Employer/Worker Algorithm Project Document

Overview of the algorithm

Our algorithm is split into 4 separate files that correlate with each separate part of the algorithm and we'll go into more detail below in separate sections.

1. final_Main.py
2. Employer.py
3. Worker.py
4. Methods.py

The algorithm works in a broad sense by using filtering methods that check the employer's values against the workers' values and return a True or a False based on gender checking, their skills/job types, their schedule for both time and days of the week, their minimum pay rate, and finally by their distance. We used a nested if loop to ensure our algorithm only returns perfect matches based on their inputs. As the algorithm checks each employer to each worker, the possible matches are saved to lists stored inside of each worker and employer object. At this point, our algorithm has a list of all possible employers for each worker that has passed the filters. Once we have these lists, we run a distance check that calculates the driving distance between each worker and the possible employers and stores them into a distance dictionary that is utilized throughout the algorithm and assigned to the workers and employer objects.

The next part of the algorithm is the part where jobs get assigned. First, the algorithm creates lists of employers with multiple jobs and assigns them to the worker that can work the most of them and is the closest to the employer. Once this iteration of the algorithm is done, it goes through the main iteration of the algorithm which assigns the rest of the jobs to the rest of the employers.

After the algorithm is done running, it displays a schedule for each worker that displays the order of jobs that the worker will be assigned and can be returned to the database to be displayed on the website.

final_Main.py

Our algorithm in its current state works by creating test objects that hold the values for several workers and employers objects that showcase multi-job matching and the filtering methods work. This portion in its final form will be connected to the database and auto-populate. In the previous project, we connected directly to a test Postgres database using the pyscogs2 library and commented these in to showcase how our team accomplished this before.

After the employer and worker objects are populated, they are appended to two lists, an Employer_List and Worker_List. The list is used to iterate through each combination of worker and employer so that each worker object is matched against each employer object.

After the list is populated, we perform the initial matching which calls match(worker, employer). During this, each worker and employer object is filtered through the list of checks and if they pass each if statement, they are added to a list of possible matches stored inside of each worker object:

- Example: If worker1 matched all filters aside from distance with Employer1 and Employer5, the resulting matched_employers list for Worker1 would look like this:
    - Worker1.matched_employers: Employer1, Employer5

After they are added, they can be checked through the print statement that displays the workers and their matched Employers.

After the lists are populated, we call CalcDistanceDict2(Employer_List, Worker_List, dist_dict). For this, the algorithm creates a distance dictionary that stores the driving distance from each worker to each employer's address as well as the distance between employers in each worker matched_employers list. This is to allow us to prioritize distance and to have the distance in miles for each possible match to use during the multi-job matching. Once the calculation is done, we add them to the matched lists with addDistanceToMatches(Worker_List, dist_dict). We then sort the Worker_List and Employer_List by shortest distance to prioritize short driving times for each worker.

The algorithm then makes a copy of the list and creates a job_id dictionary to hold the types of jobs and a worker_id dictionary that are used throughout the multi-job matching.

For the multi-job matching portion, we use several lists to hold values:

- finished_list
    - hold the finalized list of matched jobs
- job_taken_list
    - Stores the list of job_id's taken to ensure the jobs aren't assigned twice
    - Note: can easily change this to a dictionary to get O(1) search and improve performance
- max_distance
    - A modifiable variable that limits the max driving distance a job can be
    - Currently set to 30 miles because that is what the group and Aisha decided to use, but can be easily modified
- max_multi_distance
    - Was added to allow workers that are assigned multiple jobs to drive further because it would be "more worth it" to drive further for multiple jobs
    - Currently set to 45 miles but can be easily modified or made to be longer depending on the number of jobs the worker will have

- worker_list
    - After this list is populated, it is contains a list of all the jobs each employer has for each worker.
    - For example, if we have Worker3 (worker_id 3), Employer1 (employer_id 1, job_id 4), and Employer1 (employer_id 1, job_id 5):
        - worker_list[3][1] will show a list containing [4, 5] because
- worker_list_sorted
    - This is a sorted version of the worker_list sorted by the max number of jobs per employer for each worker primarily, and then by the distance secondarily.

What the algorithm does next is it prioritizes employers with multiple jobs. In order to do that, it creates the worker_list and worker_list_sorted which are both used for this. Once the lists are created and sorted, it goes down the lists and gives all the jobs to the employer that is both the closest and has the skills and time to perform all of the jobs from the employer.

After the employer with multiple job iteration of the algorithm, the algorithm goes into its main loop which assigns single jobs at a time based on distance to each worker. In this loop, after several checks to confirm the worker is able to take the job, the time array of the job gets subtracted from the time array of the worker and an extra hour at the end is also subtracted to account for travel time or extra time to their next job. The job is then also added to the job_taken_list to prevent other employers from adding a job already taken. Then, the job gets added to the finished_list of the worker using the worker as a key to get access to the list of jobs the worker has. After this, the algorithm sets the workers current address to the address of the job it just took and recalculates the distances between it and the other possible jobs it can take. Once the matched_employer list is either empty or the closest job is out of range, the algorithm moves onto the next worker and continues this until it is done. Print statements are available displaying what exactly the algorithm is doing by setting the “show_print_statements” to True.

At this point, the program prints out the finished list. The finished list of each worker contains a list of references to employer objects in the order each worker should go to them in.

Employer.py

This file was created to store employer class information. It has a class Employer that hold all the values that the database should have including a few key values like employer_id, job_id and timeslot_id that are unique to each employer.

Example/explanations from the main for each class value:
ex_Employer1 = Employer(

"Employer1", # employer name
1, # employer id
"Male", # employer gender
"Female", # employer preferred gender for the job
0, # 0 = gender does not matter, 1 = gender matters
1, # time slot id
[0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], # time job should be at
["Gardening"], # skill required for job
"3041 RIDGETOP ROAD Ottawa, Ontario", # employer address
0, 1, 1, 1, 1, 1, 0, # day of the week the job should be on
15, # pay for the job
0, # job id
)

The above portion will be auto-populated by the main.
*Note: the time_array holds 24 boolean variables starts at 0 = midnight and ends at 23 = 11 PM. This allows us to remove timeslots in the methods.py during the multi-job matching process.

Worker.py

Essentially the same as Employer.py but with respect to worker variable names for easier understanding and can have different values than employer if needed. There is also no job_id associated with a worker.

Methods.py

This file stores all the methods that are used for matching. We separated the file from the main to make the main easier to understand. We also left the connect_to_db() which was used to connect to the Postgres database in our previous objects as a reference for adding database connections to this algorithm.
*Note: the variable geolocater = Nominatim(user_agent=”http”) is where the map API is set. If google maps API is needed, replace this value to use google maps API.

test_route(address):

This was used simply for ensure our addresses don’t return None when geolocater.geocode(address) is run.

get_route2(address1, address2):

This method uses OSRM and the nomatim openstreetmap API to simply calculate the distance. It first ensures that the addresses aren’t the same and returns 0 miles if they are. It then assigns

a worker_Location variable using address1 and geocoding the address to a usable format as well as employer_Location using address2.
The method then extracts the longitude and latitude from each location and assigns them to pickup_long and pickup_lat for workers (starting address) and dropoff_long and dropoff_lat for employers' locations (ending location).
After the algorithm formats the information and sends it to the API and returns the distance in miles to the distance variable in meters. We then convert the meters to miles and returns distance_Miles to the algorithm.

filter_Days():

This method is used to filter the days that the workers and employers match. It initially starts with an empty list called match_Days and checks if the employers and workers match for each day and append them to a list if they are matched. For example, if the worker and employer both have availability for Monday and Wednesday but the worker is also available Friday, the method will check equality and append [“Monday”, “Wednesday”] to the list and since they both don’t include Friday, Friday will not be appended.

This method is utilized in checkDays() for the match() function

checkDays():

This method calls filter_Days and stores them to a days_list. It simply checks that if there are days that are matched, i.e. the list is not empty, it returns True because it passes the day check.

CheckGender():

This method accounts for if gender matters for either the employer or the worker. It uses the boolean variable “gender_matter” and checks each possible scenario for gender_matters.

It checks if gender doesn’t matter for both employer and worker and assigns True to pass the check.
It checks if Gender matters for both and ensures the preferred gender for both worker and employer are matched and returns True if they match and False if they don’t match.
It checks both scenarios for when it matters for only one. If the worker decides gender matters and the worker doesn’t care, it checks that the employer is the preferred gender and returns True if they are and false if it is not the preferred gender and vice versa for the employer to the worker.

checkSkills():

This method creates a temporary set for the list of jobs skills/jobs required. By doing so, we can check for intersections in the list and if the worker has the same skill as the required job, it returns True and if the intersection is empty, meaning no matching job skills, it returns False.

checkTimeArray():

This method utilizes the time_array variable in the employer and worker class and creates a numpy array with each worker/employer time_array. With numpy, we subtract the two arrays and if there are any -1 values in the result_array, it means the time's don't match for the job and it returns False for the check and if the time_arrays work for each other, it will have no -1's and returns True. So if a worker is available from 3 pm to 7 pm and the employer needs work done from 4 pm to 6 pm, it would subtract the 4-6 from 3-7 and return no 0's. Or if a worker is available from 12 pm to 2 pm and the employer needs work done from 4 pm to 6 pm, it would return -1 for the misaligned time slots.

checkPay():

Check pay simply checks that the employer's pay rate is greater than or equal to the worker's pay rate and returns True if the pay rate is accepted and False if the pay is below the worker's requested pay.

Match()

This is our initial filter method. It works through nested if statements that if any of the above check methods return False, it breaks the nested ifs and will not add the employer in the worker's match_employers list. This ensures only perfect matches are selected and the if statements are ordered by the predetermined priority of filters.

addDistanceToMatches():

This method simply adds the distance matrix to each worker's employer in their employer list. The for statement iterates the employees that are inside of the worker.matched_employers list (after they are filtered to reduce API calls). After it adds the employers and their distance in miles to new_list. It changes the worker.matched_employer list to hold two values with their name and the miles instead of holding just their names.

SortMatchedWorkers and SortSingleWorker, and sortMatchedEmployers():

These all are to just simply to sort the lists created by our Match() and addDistanceToMatches function. They use selection sort to sort the list.

CalcDistanceDict2():

This function is an updated version of CalcDistanceDict(). This function is used to do the actual distance calculations for each worker to each employer. This helps creates a distance matrix that is used to determine which worker should get each job.

*Note: the time.sleep(1) calls is required for OSRM and their test server to allow us to use their server. Too many API calls in a short period of time get rejected and we found sleeping the code for 1 second will allow us to make all the API calls we need. With Google Maps API, this part is unneeded and will allow the code to run with a shorter runtime.

RecalcDistance():

RecalcDistance iterates through the worker's list of matched_employers and returns an updated distance calculation. Once the algorithm finds the match after the multi-job matching is completed, it assigns a new distance that calculates the total driving distance between the newly assigned/final jobs.