Report

The aim of this project was to create an application that navigates the user through Germany’s subway system (U-bahn). The system uses 3 different search algorithms; breadth first search, depth first search and an A\* algorithm. The system requires 2 inputs; the start station and the station the user wants to reach, it then returns the path the user needs to take and the time of the trip. If a path exists between the start and the goal, the 3 algorithms will be able to find it, however only the A\* will choose the most cost-efficient path.

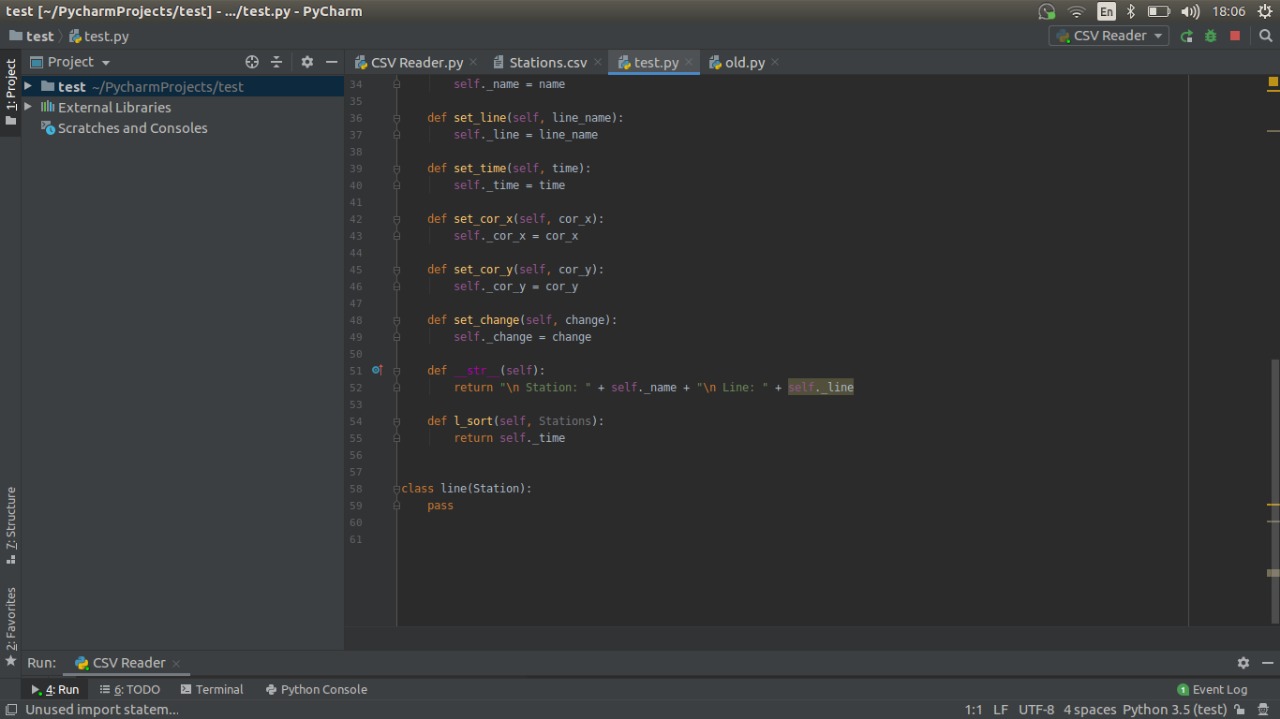
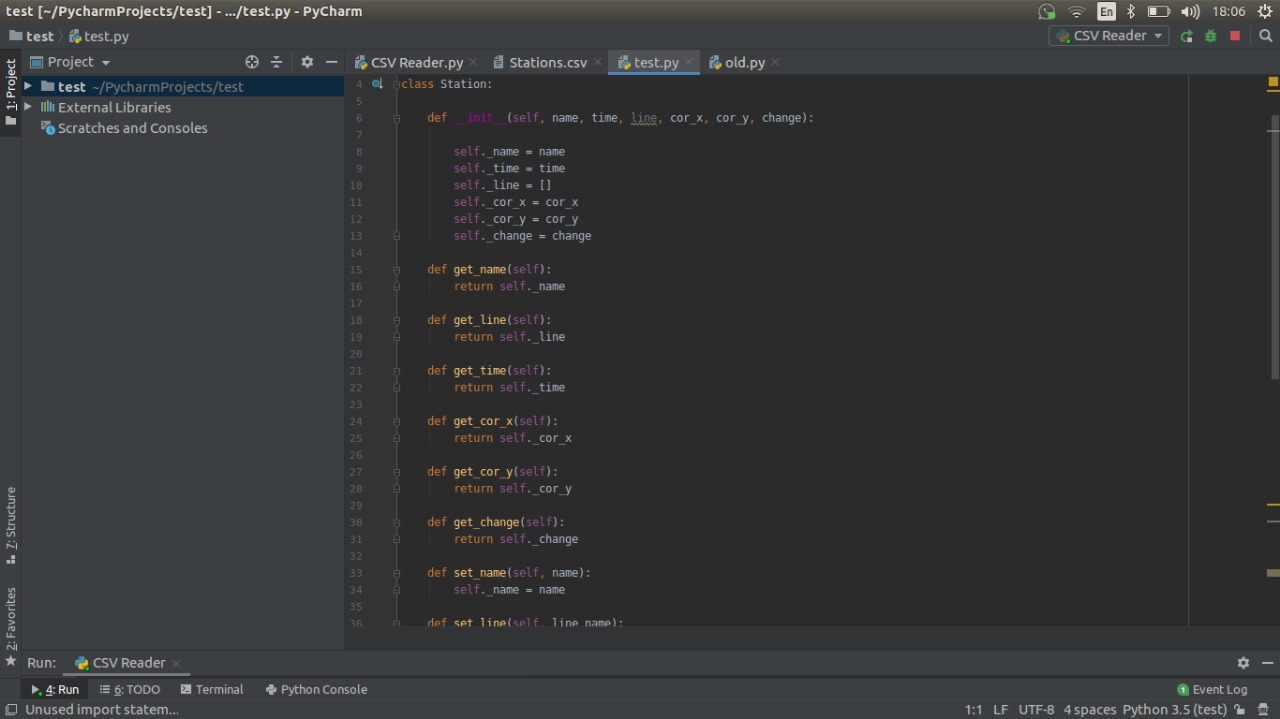
The project was implemented using python on Pycharm. NetworkX was imported to help demonstrate the graph and to make use of its already built-in methods. A csv file of all the U-bahn stations was also loaded, the file has the name, line number, whether it is a change station or not and its time relative to the start of the line. Also another file with the name of the station along with its X and Y global co-ordinates, this will be used for the heuristic for the A\*.

The A\* heuristic is the speed (average 60 km/h) divided by the Chebyshev distance giving us the time. The algorithm then chooses the best path with the lowest cost based on the estimated time. The heuristic will always be admissible as we get the time for the Chebychev distance which is the straight line distance, therefore it will always be shorter than the real path.

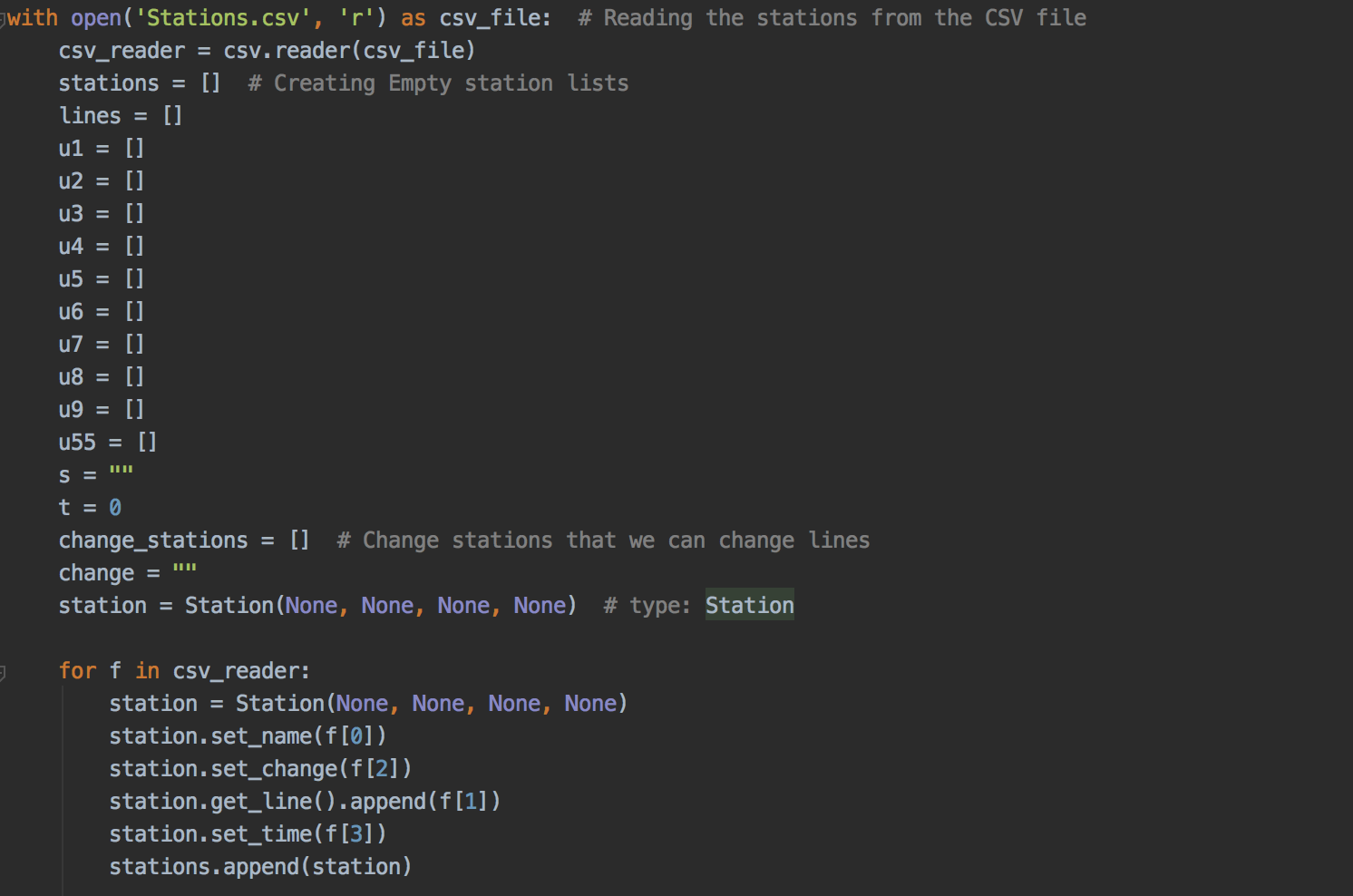
We weren’t able to include the +5 minutes whenever a line change occurs, as our search methods returned a list of strings (station names), hence we were not able to access the getLine() method. We needed this method for when iterating on our path, we check whether the current station is a change station or not, the we wanted to compare the previous and next stations, if their lines were different, that means that a line change occurred, and +5 should be added to the total trip time, however we were not able to access this data. We tried iterating on the full station list, but some stations have 3 lines, so again we could not choose which line we wanted.

Other than that all the algorithms are working perfectly and the time calculated is always correct. Another note is when running the DFS the results sometimes vary, that is because every time the graph is initialised, it is drawn with a different orientation. All the stations are in the correct order and correctly connected, however instead of a line pointing to the right it points to the left, this does not make a difference for the A\* and the BFS, but the DFS always chooses the right most branch, therefore when the orientation changes so will the result. But based on every orientation, the path to goal, time taken are correct.

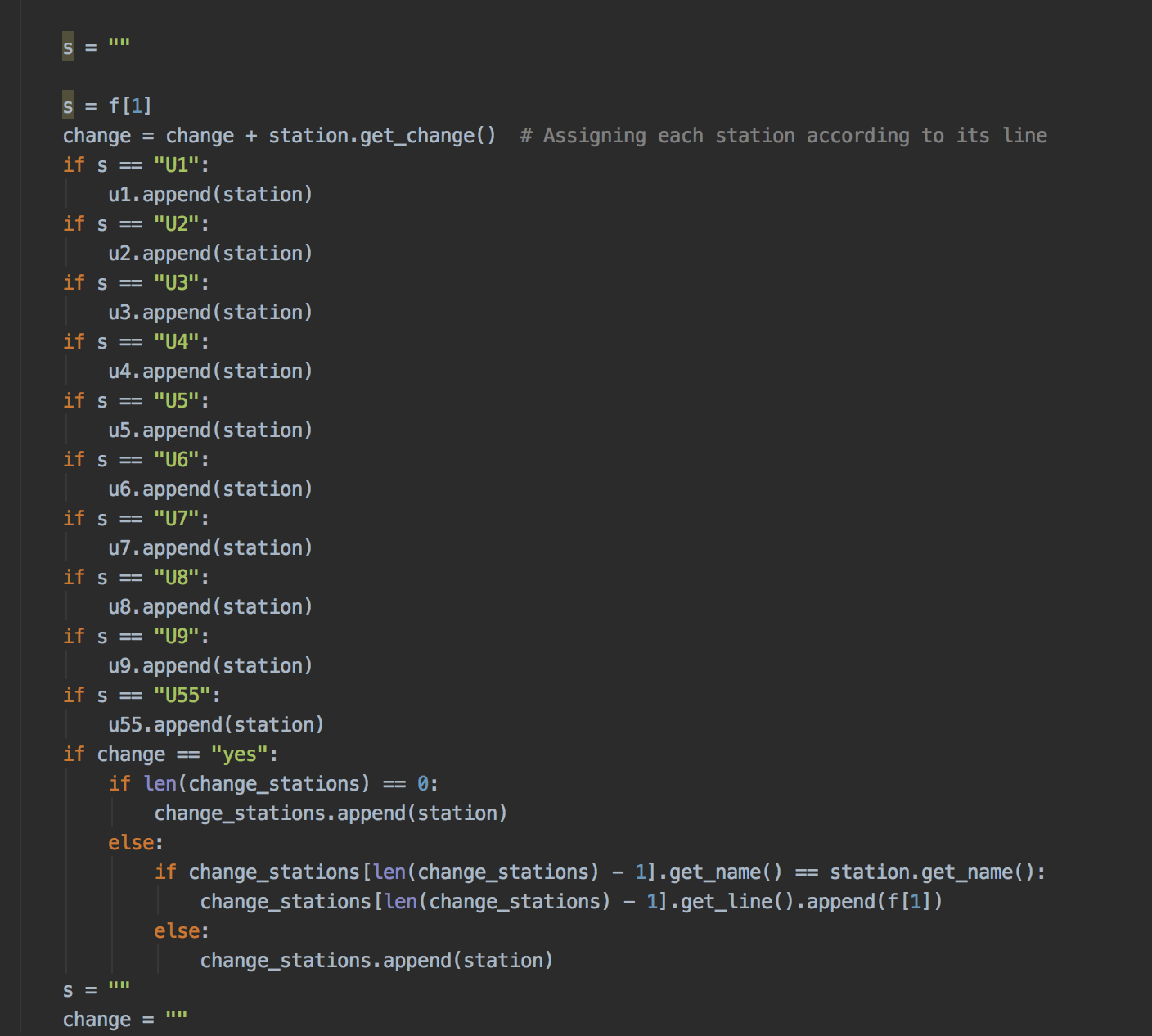
The 3 algorithms are complete, if a path exists they all will find it. Both the BFS and A\* are optimal, they always get the best path, while the DFS is not, it always chooses the right most branch, hence usually choosing longer paths. The BFS and A\* always expand the same nodes while the DFS always expands unnecessary nodes and expands a much longer path, unless there are no change stations, or if the correct path is the one to the right.



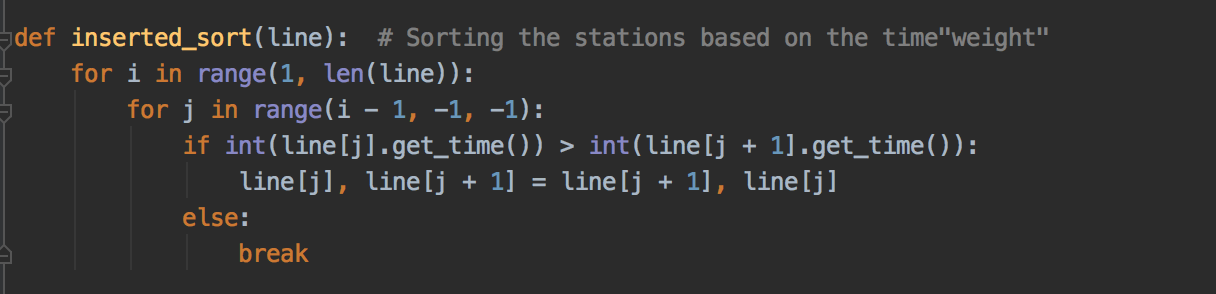
This is the class for the stations, these are the attributes with their getters and setters.



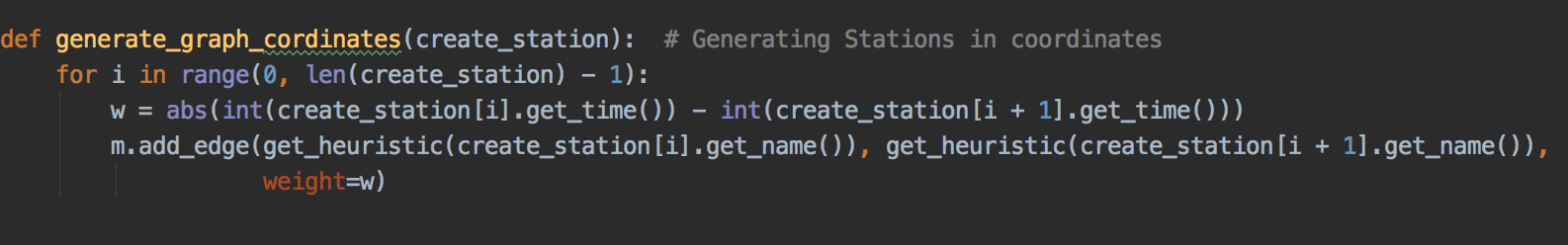
Here we prepare to access the information from the csv file by creating empty lists for each line and change stations list which has the stations where you can change between the lines. Then we create a variable of “station” for each station we have from our CSV file, assigning its according name, line, time and change. So after this step we now have all the stations with their attributes.

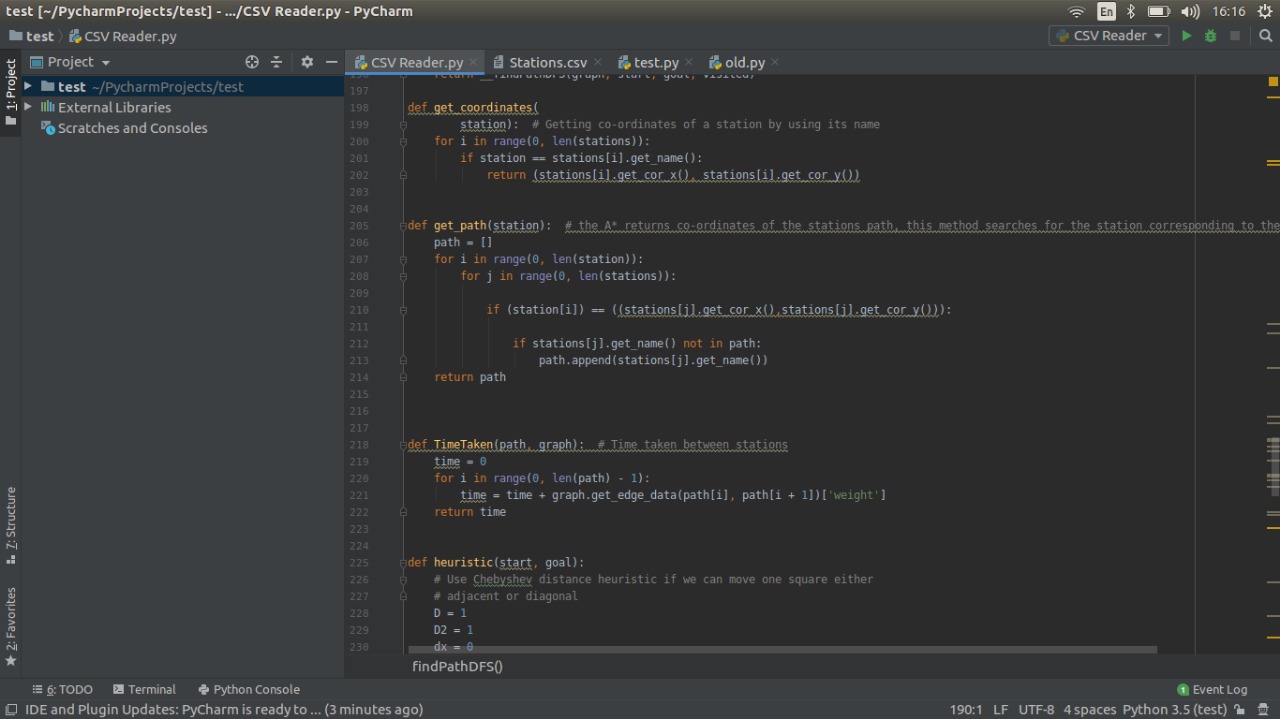


Then here we assign every station to its correct line and append them to the list of its line.

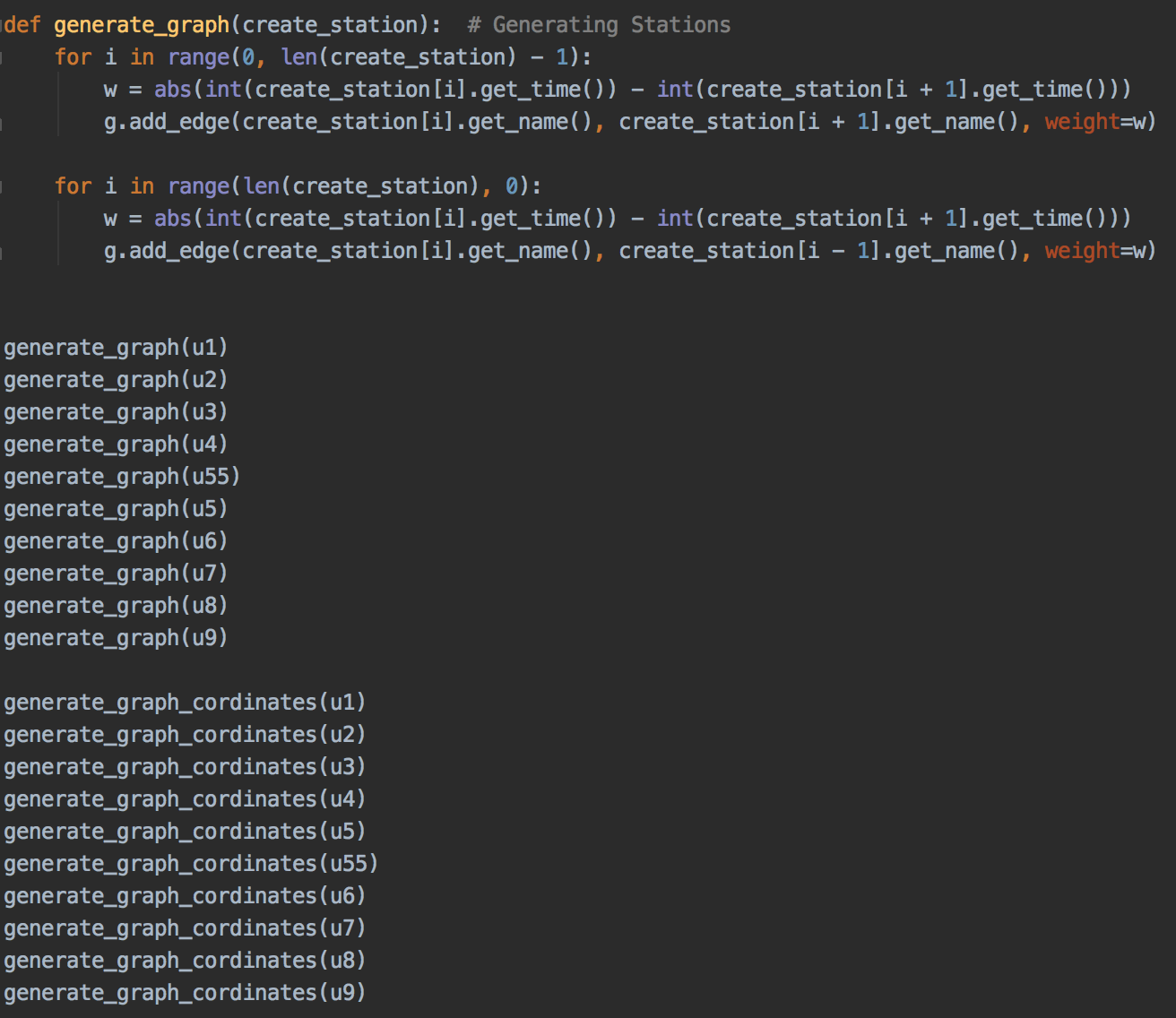


This part of the code was implemented to re-arrange the stations based on the time of each station. It gets the time of the station and compares it with the next station then if the next station’s time is less than the previous then it replaces the station. Otherwise it remains the same this keeps ongoing until all the stations in each line are sorted ascendingly.

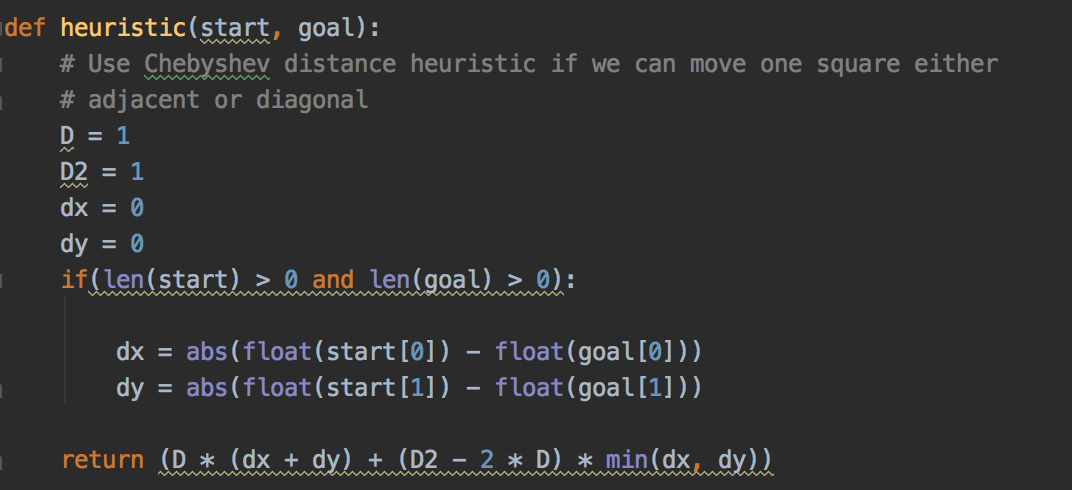




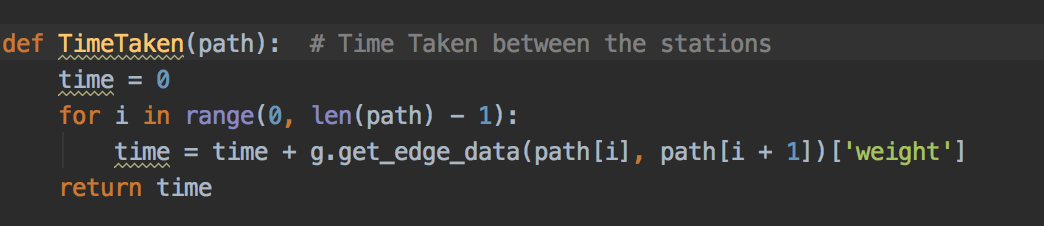
This function (generate\_graph\_cordinates) was implemented to get the weight of each edge in the graph. While the (get\_coordinates) function was implemented to get the coordinates of the stations. If the coordinates of the input station is equal to the station then we assign the coordinates to x in the form of (x , y).



generate\_graph() is responsible for generating the graph by creating the edges between the connected stations based on their names while generate\_graph\_cordinates() creates a graph based on the XY co-ordinates.



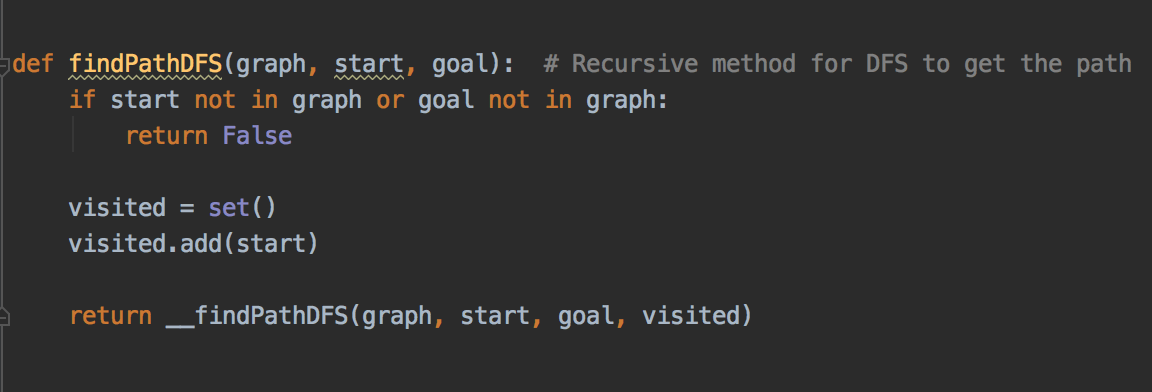
This method was implemented as a function to calculate the heuristic value that will be used in the A\* algorithm by calculating the difference between the XY co-ordinates of the start and goal stations.



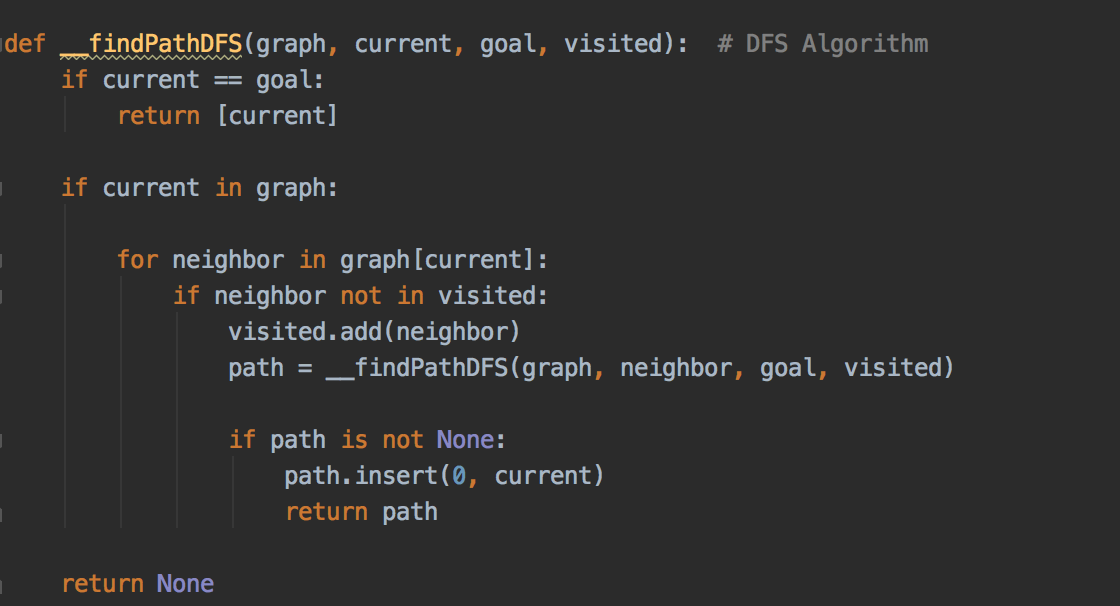
The TimeTaken() function was made to calculate the time between the stations. The time is calculated by getting the time and adding it to the sum between the stations (path[i], path[i+1]). Every edge on the graph contains the time between every station and its next station.



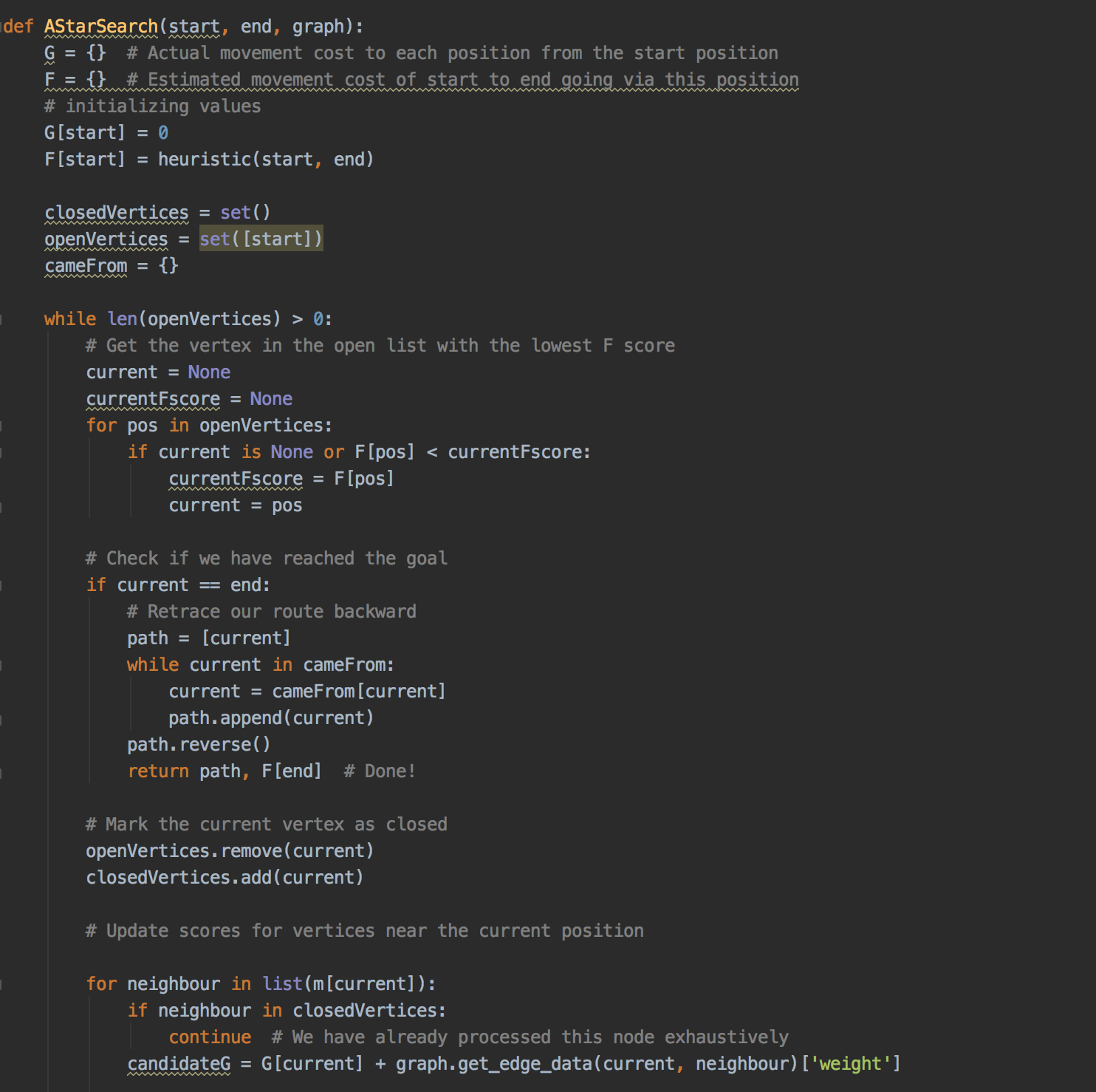
This part of the code is for implementing the breadth first search algorithm. It takes graph, start and desired goal as an input. Then we have a list that takes start (node to be explored), visited is the list for nodes that have been visited. As long as the queue is not empty, we pop the first node and put it in ‘path’. ‘Node’ is for the last term in the list, so if the node has not been visited yet we get its neighbours and put them in the list ‘neighbours’. Then we iterate on the list neighbours to get the ‘path’ and put it in a new path and append the current neighbour to it then we append ‘new path’ to ‘queue’ then we check if the current neighbour is the goal if yes, we calculate the time the we return the ‘new path’ and ‘time’ also add the current node to visited. However, if neighbour not equal to goal keep searching.



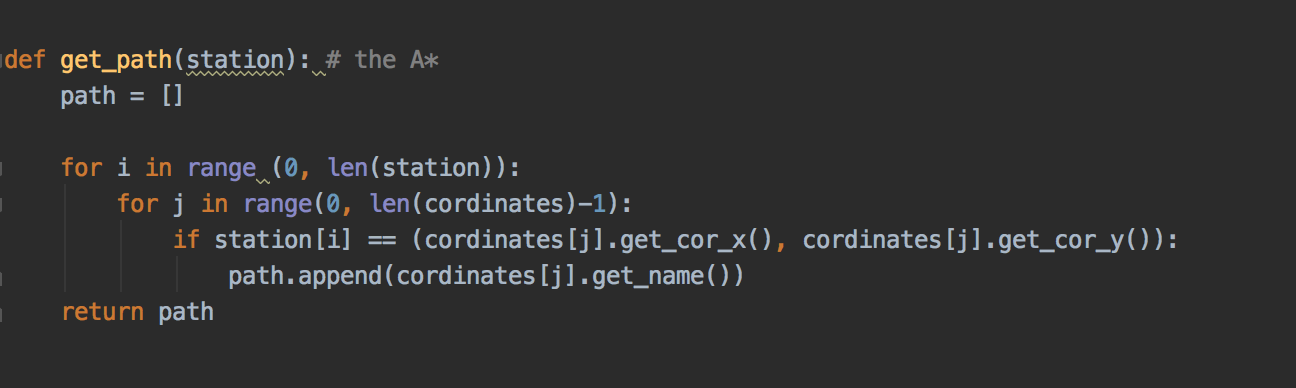
The findPathDFS() method is made for the depth first search. It takes a graph, start and goal as input. First it checks if the inputs are not in the graph then it returns false if they are not. It puts ‘start’ in ‘visited’ and calls findPathDFS (graph, start, goal, visited).



This method takes graph, current, goal, visited as input from the previous method then we start checking if the current node is the goal and return if that is. Otherwise we iterate on the neighbours of the current node and check if the current neighbour has not been visited, then we add the neighbour to ‘visited’ and recall the method new neighbour instead of ‘current’ and updated ‘visited’. Then it will iterate till it reaches the last neighbour in the graph.

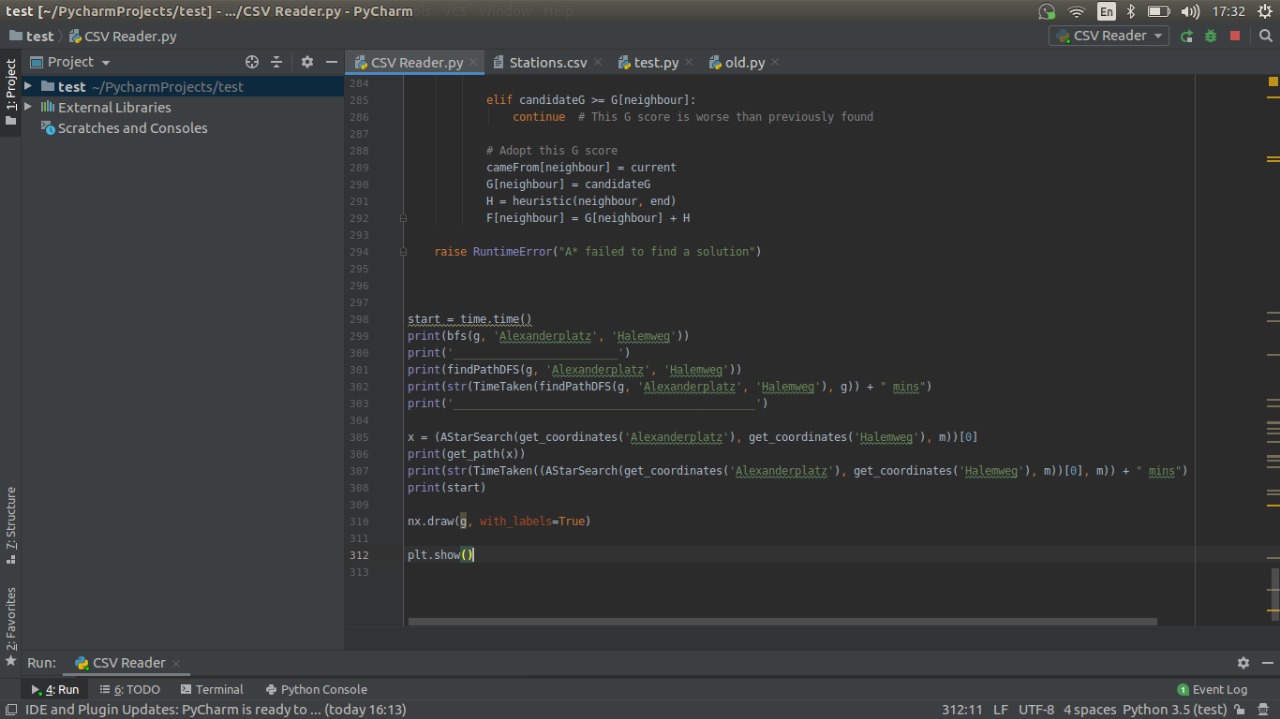


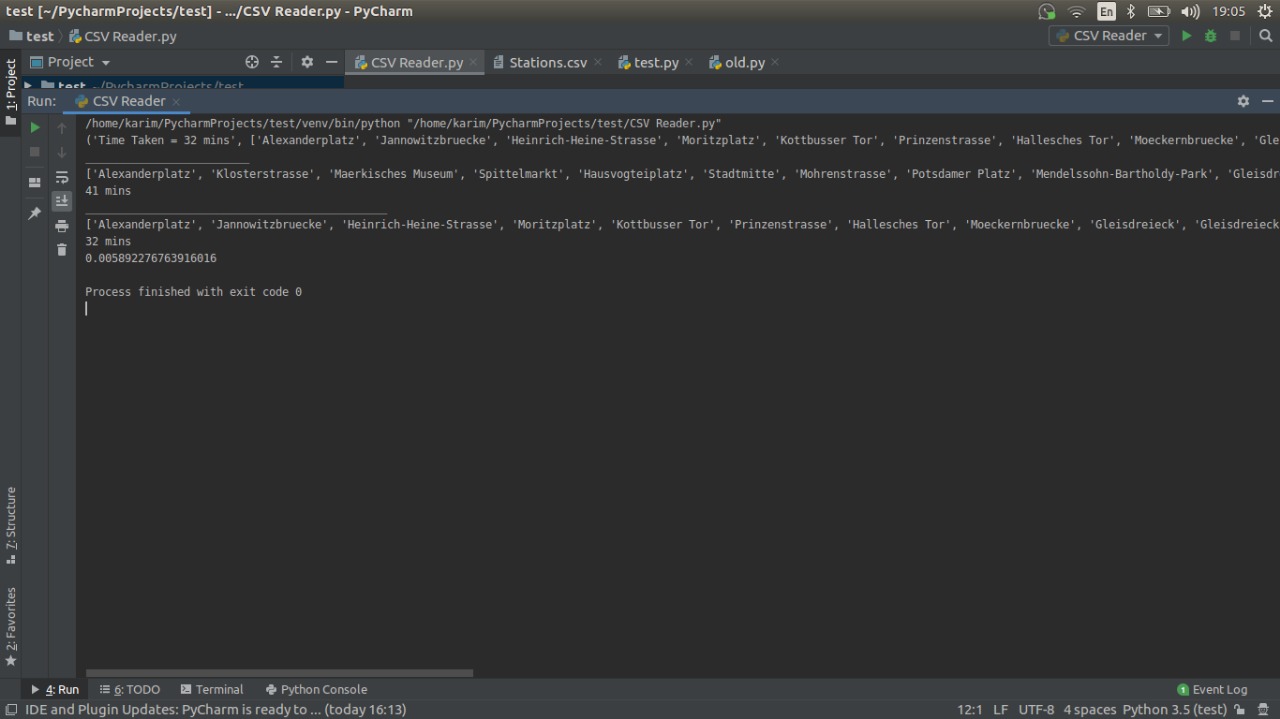


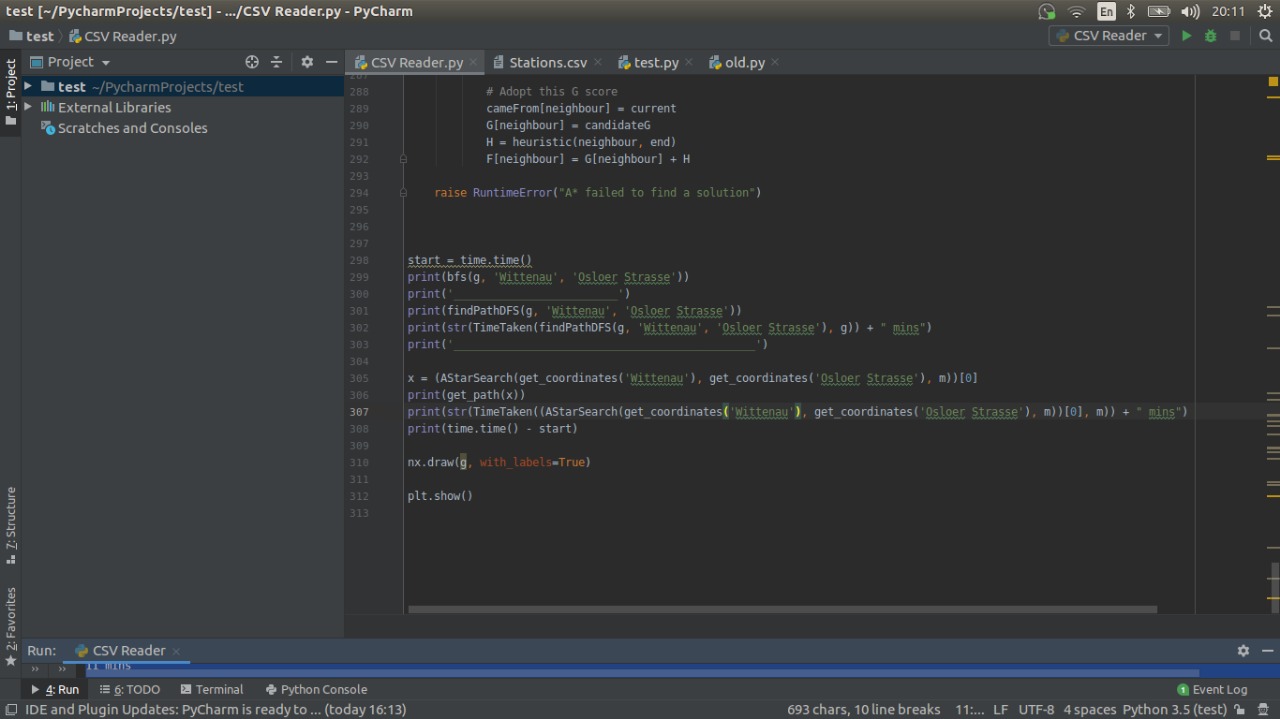


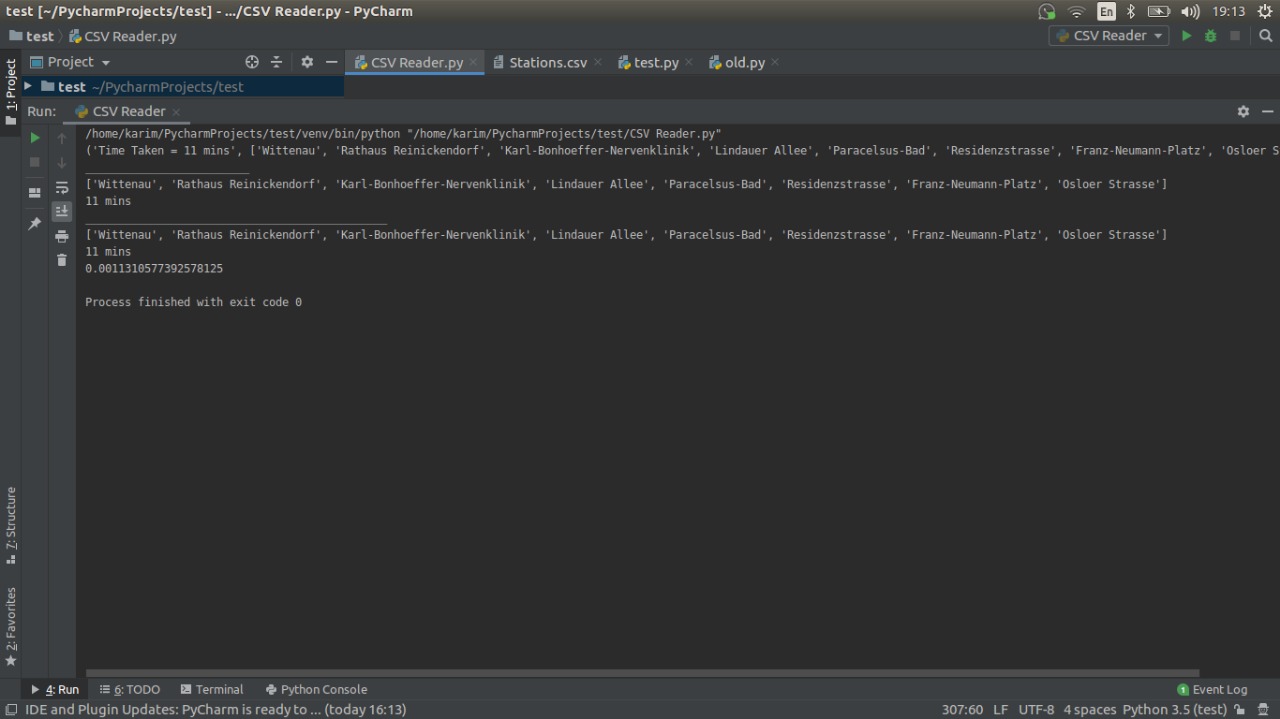
This method takes a list of station coordinates from the path of the A\* method as input and creates an empty list ‘path’ then loop on every station coordinates and compares it to all the coordinates to the CSV file then when it finds a match to the input coordinate it returns the name of the station. Then a list of strings of the station names in ‘path’ is returned.

Here are 2 examples for each search algorithm, one from Alexanderplatz to Halemweg and the other from Wittenau to Osloer Strasse. The first example has change stations through the path, wo as we can see how the DFS returns a much longer path (41 mins) while the BFS and A\* return a path with (32 mins) only. However the second example contains no change stations, therefore the 3 algorithms return the same exact time and path to goal. Also printed at the very end is the execution time.









Citations:

https://rosettacode.org/wiki/A\*\_search\_algorithm#Python

https://codereview.stackexchange.com/questions/193410/breadth-first-search-implementation-in-python-3-to-find-path-between-two-given-n

https://eddmann.com/posts/depth-first-search-and-breadth-first-search-in-python/