CSE 643: Artificial Intelligence

Assignment 2: Report

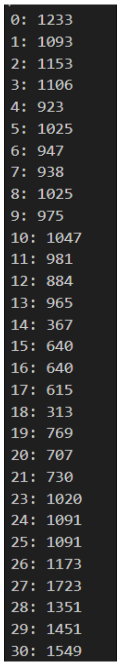
Aditya Aggarwal **2022028**

Question 1` - Knowledge Base Creation

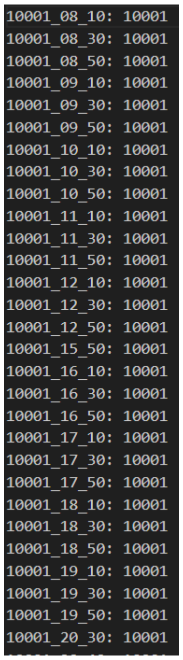
A. *Trip_to_route:* A dictionary containing a one-to-one mapping of trips and routes. Each trip has a unique *trip_id*, and multiple trips can share a single route.

B. *Route_to_stops*: A dictionary mapping each route to a list of stops that are visited on that route. This dictionary was created by merging the *trips* and *stop_times* dataframes.

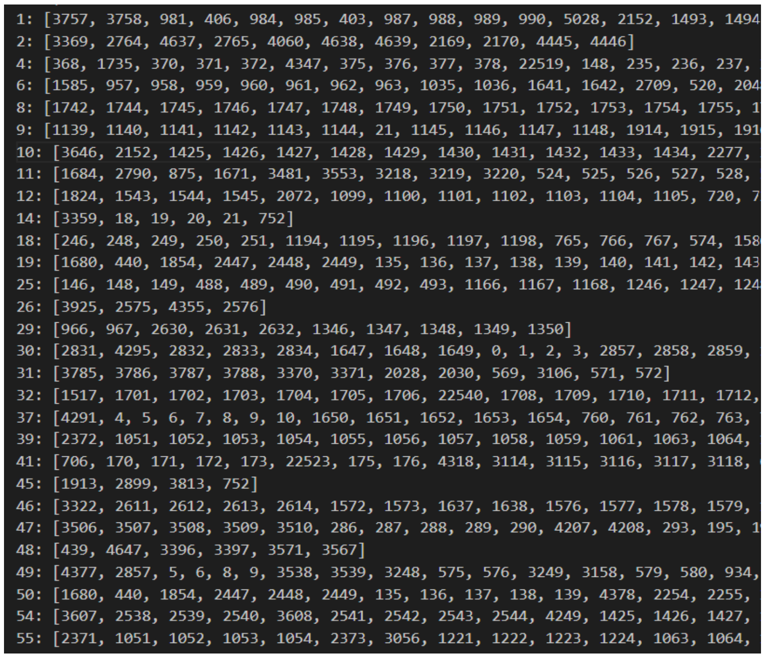
C. *Stop_trip_count:* A dictionary mapping each stop to the number of trips that go via that stop.



```
0: 1233
1: 1093
2: 1153
3: 1106
4: 923
5: 1025
6: 947
7: 938
8: 1025
9: 975
10: 1047
11: 981
12: 884
13: 965
14: 367
15: 640
16: 640
17: 615
18: 313
19: 769
20: 707
21: 730
23: 1020
24: 1091
25: 1091
26: 1173
27: 1723
28: 1351
29: 1451
30: 1549
```



```
10001_08_10: 10001
10001_08_30: 10001
10001_08_50: 10001
10001_09_10: 10001
10001_09_30: 10001
10001_09_50: 10001
10001_10_10: 10001
10001_10_30: 10001
10001_10_50: 10001
10001_11_10: 10001
10001_11_30: 10001
10001_11_50: 10001
10001_12_10: 10001
10001_12_30: 10001
10001_12_50: 10001
10001_15_50: 10001
10001_16_10: 10001
10001_16_30: 10001
10001_16_50: 10001
10001_17_10: 10001
10001_17_30: 10001
10001_17_50: 10001
10001_18_10: 10001
10001_18_30: 10001
10001_18_50: 10001
10001_19_10: 10001
10001_19_30: 10001
10001_19_50: 10001
10001_20_30: 10001
```

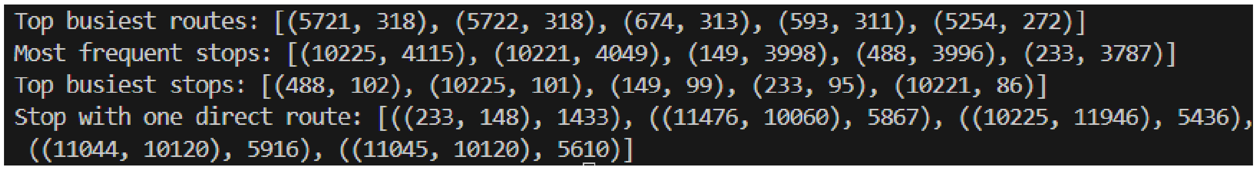


```
1: [3757, 3758, 981, 406, 984, 985, 403, 987, 988, 989, 990, 5028, 2152, 1493, 1494
2: [3369, 2764, 4637, 2765, 4060, 4638, 4639, 2169, 2170, 4445, 4446]
4: [368, 1735, 370, 371, 372, 4347, 375, 376, 377, 378, 22519, 148, 235, 236, 237,
6: [1585, 957, 958, 959, 960, 961, 962, 963, 1035, 1036, 1641, 1642, 2709, 520, 204
8: [1742, 1744, 1745, 1746, 1747, 1748, 1749, 1750, 1751, 1752, 1753, 1754, 1755, 1
9: [1139, 1140, 1141, 1142, 1143, 1144, 21, 1145, 1146, 1147, 1148, 1914, 1915, 191
10: [3646, 2152, 1425, 1426, 1427, 1428, 1429, 1430, 1431, 1432, 1433, 1434, 2277,
11: [1684, 2790, 875, 1671, 3481, 3553, 3218, 3219, 3220, 524, 525, 526, 527, 528,
12: [1824, 1543, 1544, 1545, 2072, 1099, 1100, 1101, 1102, 1103, 1104, 1105, 720, 7
14: [3359, 18, 19, 20, 21, 752]
18: [246, 248, 249, 250, 251, 1194, 1195, 1196, 1197, 1198, 765, 766, 767, 574, 158
19: [1680, 440, 1854, 2447, 2448, 2449, 135, 136, 137, 138, 139, 140, 141, 142, 143
25: [146, 148, 149, 488, 489, 490, 491, 492, 493, 1166, 1167, 1168, 1246, 1247, 124
26: [3925, 2575, 4355, 2576]
29: [966, 967, 2630, 2631, 2632, 1346, 1347, 1348, 1349, 1350]
30: [2831, 4295, 2832, 2833, 2834, 1647, 1648, 1649, 0, 1, 2, 3, 2857, 2858, 2859,
31: [3785, 3786, 3787, 3788, 3370, 3371, 2028, 2030, 569, 3106, 571, 572]
32: [1517, 1701, 1702, 1703, 1704, 1705, 1706, 22540, 1708, 1709, 1710, 1711, 1712,
37: [4291, 4, 5, 6, 7, 8, 9, 10, 1650, 1651, 1652, 1653, 1654, 760, 761, 762, 763,
39: [2372, 1051, 1052, 1053, 1054, 1055, 1056, 1057, 1058, 1059, 1061, 1063, 1064,
41: [706, 170, 171, 172, 173, 22523, 175, 176, 4318, 3114, 3115, 3116, 3117, 3118,
45: [1913, 2899, 3813, 752]
46: [3322, 2611, 2612, 2613, 2614, 1572, 1573, 1637, 1638, 1576, 1577, 1578, 1579,
47: [3506, 3507, 3508, 3509, 3510, 286, 287, 288, 289, 290, 4207, 4208, 293, 195, 1
48: [439, 4647, 3396, 3397, 3571, 3567]
49: [4377, 2857, 5, 6, 8, 9, 3538, 3539, 3248, 575, 576, 3249, 3158, 579, 580, 934,
50: [1680, 440, 1854, 2447, 2448, 2449, 135, 136, 137, 138, 139, 4378, 2254, 2255,
54: [3607, 2538, 2539, 2540, 3608, 2541, 2542, 2543, 2544, 4249, 1425, 1426, 1427,
55: [2371, 1051, 1052, 1053, 1054, 2373, 3056, 1221, 1222, 1223, 1224, 1063, 1064,
```

A B C

Inference

The following are the results from the functions *get_busiest_routes, get_most_frequent_stops, get_top_5_busiest_stops,* and *get_stops_with_one_direct_route respectively.*



```
Top busiest routes: [(5721, 318), (5722, 318), (674, 313), (593, 311), (5254, 272)]
Most frequent stops: [(10225, 4115), (10221, 4049), (149, 3998), (488, 3996), (233, 3787)]
Top busiest stops: [(488, 102), (10225, 101), (149, 99), (233, 95), (10221, 86)]
Stop with one direct route: [((233, 148), 1433), ((11476, 10060), 5867), ((10225, 11946), 5436),
 ((11044, 10120), 5916), ((11045, 10120), 5610)]
```

Question 2 - Reasoning

Part (a) - Execution Time Analysis

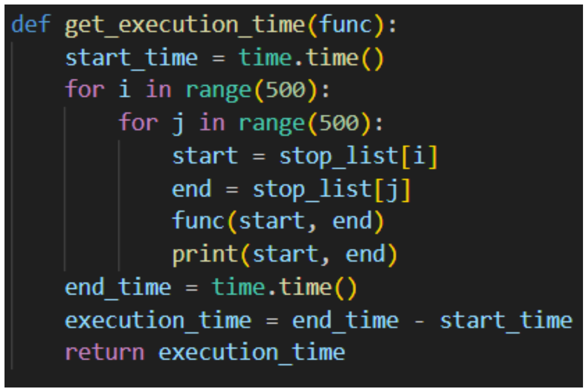
On analyzing the runtime for both functions, it is concluded that the *query_direct_routes* function takes lesser execution time in comparison to the *direct_route_brute_force* method. This difference is notable when both functions are executed for a large number of pairs of stops.

When both algorithms are run for all possible combinations of the first 500 stops (250000 iterations), and their runtimes are calculated using the time library, the *direct_route_brute_force* function takes 634.59 seconds to complete its execution, whereas *query_direct_routes* takes 192.26 seconds to run. It is apparent from the results that the *query_direct_routes* function is much faster than the *direct_route_brute_force* function.

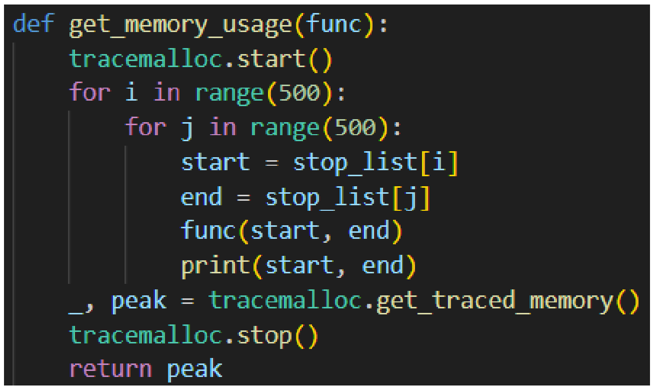
Part (a) - Memory Usage Analysis

On comparing both functions, it is seen that the *direct_route_brute_force* function takes less memory space compared to the *query_direct_routes* function. This is because the former only needs a space complexity of $O(R)$ to store the list of direct routes as its answer. However, the latter first prepares its knowledge base by adding the predicate *RouteHasStop(R, S)* for each route, which requires an additional space complexity of $O(R \times S)$.

It is clear from the above analysis that *direct_route_brute_force* requires much less memory usage. This analysis is confirmed when both algorithms are run for all possible combinations of the first 500 stops (250000 iterations), and their memory usages are tracked by using the tracemalloc library. The peak memory usage of the functions *direct_route_brute_force* and *query_direct_routes* are 0.001064 MB and 5.788477 MB, respectively.



```python
def get_execution_time(func):
    start_time = time.time()
    for i in range(500):
        for j in range(500):
            start = stop_list[i]
            end = stop_list[j]
            func(start, end)
            print(start, end)
    end_time = time.time()
    execution_time = end_time - start_time
    return execution_time
```



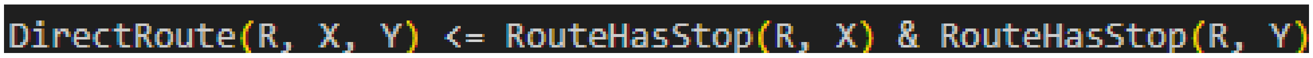
```python
def get_memory_usage(func):
    tracemalloc.start()
    for i in range(500):
        for j in range(500):
            start = stop_list[i]
            end = stop_list[j]
            func(start, end)
            print(start, end)
    _, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    return peak
```

Functions used to obtain execution time and memory usage

Part (b) - Intermediate Steps

direct_route_brute_force: In this function, in one step, we iterate twice through the list of stops for one specific route. In the first iteration, we check whether the given starting stop is present in the list, and in the second iteration, we check the same for the ending stop. If both stops are present in the route, we append the *route_id* to the list of answers.

query_direct_routes: In this function, we define the predicate that signifies that R is a direct route between two stops, X and Y, with the condition that the predicate that route R has X and the predicate that route R has Y are true simultaneously.



```python
DirectRoute(R, X, Y) <= RouteHasStop(R, X) & RouteHasStop(R, Y)
```

Part (c) - Number of steps

For a given pair of stops, the function *direct_route_brute_force* checks each route and determines whether it is a direct route or not. It traverses through the *route_to_stops* dictionary and traverses each list of stops twice to check whether both stops are present in the list or not. Thus, the brute force method has a total number of steps of the order of $O(R \times S)$.

The function *query_direct_routes* uses the pyDatalog library to first create its knowledge base by adding all the routes in its knowledge base. Hence, the function does not have to traverse through the entire list of stops corresponding to each route. Thus, the brute force method has a total number of steps of the order of $O(R)$.

Question 3 - Planning

Part (a) - Execution Time Analysis

When the runtime of forward chaining and backward chaining algorithms are compared, it is found that the latter runs faster. Since the knowledge base is large, forward chaining takes more time as it applies the rules to all facts and generates all possible inferences, even if they are irrelevant to the query. Backward chaining runs faster as it only considers the facts relevant to the specific goal of finding an optimal route with given constraints.

When both algorithms are run for all possible combinations of the first 100 stops (10000 iterations), with the stop to be included being generated randomly, the forward chaining takes 327.71 seconds to execute. Backward chaining only takes 16.05 seconds to execute. It can be concluded from the results that backward chaining is much faster than forward chaining.

Part (a) - Memory Usage Analysis

Forward Chaining typically uses more space than backward chaining because it needs to store all the facts and all rules, even if they are irrelevant to the goal. As the number of facts grows, the memory required to store intermediate facts also increases.

This is evident by the results we obtained. While the forward chaining algorithm has a peak memory usage of 18.81 MB, the backward chaining algorithm only requires a peak memory of 3.46 MB.

Part (b) - Intermediate Steps

In both algorithms, the definition of the predicate that determines the optimal route remains the same. With the maximum number of transfers set to 1, the optimal route predicate determines whether two routes, R1 and R2, exist such that R1 is a direct route from the starting stop to the stop where the transfer occurs and R2 is a direct route from the stop where the transfer occurs to the ending stop. The backward chaining algorithm works in a similar fashion except that it first defines the rules by using general stops X, Y, and Z, and after establishing the rule, goes backward to find the list of optimal routes for the X being the starting stop, Y being the ending stop and Z being the point of transfer.

Part (c) - Number of Steps

Both the algorithms run for a number of steps of the order $O(R \times S)$. This is because both algorithms require the search of the given stops in the lists corresponding to all the routes to reach the required goal. However, as evident, backward chaining reaches the optimal path faster than forward chaining as it starts from the end and only searches the paths that lead to the goal, eliminating all the other paths.

Visualizations and Results

Algorithm	Execution Time (seconds)	Peak Memory Usage (MBs)	Iterations
direct_route_brute_force	634.59	0.001064	250000
query_direct_routes	192.26	5.78	250000
forward_chaining	327.71	18.80	10000
backward_chaining	16.05	3.46	10000

Stop-Route Network Graph