Part 1.¶

You are tasked with analyzing public transportation networks. You are allowed to use the *collections* module for part 1, please do not use any other external modules.

For each of the questions below, you should develop a correct, efficient algorithm, implement the algorithm in Python, and discuss the running time and efficiency of your implementation in your *pdf*. Your discussion should include an estimate of the asymptotic running time and a concise description of how you constructed this estimate. You should also include a brief description of each algorithm.

1) (2 pts) You are given a set of airports and connections corresponding to a air transportation network. Specifically, you know if there is a direct connection between any two airports in the network. Complete the function, *flightLegs*, so that it efficiently computes the minimum number of flights required to travel between two distinct airports on the network. The function should also efficiently determine the number of distinct routes between the two airports which require this minimum number of flights. The function is provided an *N*- element adjacency list, *Alist*, as input, and airports are numbered from *0* to *N-1*. The *ith* element of *Alist* is a list containing the airports which can be reached directly from airport *i*. The network is symmetric, so if *i* can be reached directly from *j*, then *j* can be reached directly from *i*. The starting (*start*) and destination (*dest*) airports are also provided. The function output should be a two-element list containing the minimum number of flights required to travel between *start* and *dest* and the number of distinct journeys between *start* and *dest* with this minimum number.

2) (5 pts) You will now analyze the journeys on a subway network. You are provided with a list of stations and direct routes between stations.

i) You are also provided with *densities* for the network where d_{ij}dij corresponds to the average number of people on a direct journey from station *i* to *j* (no stations other than *i* and *j* are visited during a direct journey). For convenience, we will assume that $d_{ij}=d_{ji}$dij=dji. A safety factor, S_{ab}Sab, for a journey between stations *a* and *b* (not necessarily a direct journey) is defined to be the maximum density encountered during the journey (lower *S* corresponds to higher safety). Given a starting and destination station, develop an algorithm to efficiently find the *safest* journey between the two stations. Implement your method in *safeJourney* in *part1.py*. Along with the starting and destination stations, the function is provided an adjacency list, *Alist*, for the network. The *ith* element of the list is a list of tuples. Each tuple contains a station which can be directly reached from *i*, and the density for travel between that station and *i*. The function should return a list containing the sequence of stations encountered during the safest journey and the safety factor for the journey. E.g., if the function returns [[1,5,3],10] then the full journey is 1 -> 5 -> 3 and the safety factor of the journey is 10. If multiple journeys produce the same minimum *S*, you can return any one of those journeys. If no path exists between the starting and destination stations, return an empty list.

ii) Now consider the fastest journeys between stations *start* and *dest*. You are provided with durations of journeys between stations *i* and *j*, t_{ij}tij (rounded to the nearest minute and $t_{ij}=t_{ji}$tij=tji), and if there are multiple shortest journeys, you should select the one with the least number of steps. Complete *shortJourney* so that it efficiently finds this fastest journey and returns the stations included in the journey (as a list) and the duration of the journey. If no path exists between the starting and destination stations, return an empty list. The overall setup of the function is very similar to *safeJourney*, and further details are provided in the function docstring.

3) (3 pts) You are provided a network of train stations and, for a given station outside of this network (*x*), you know the minimum fare for reaching each in-network station from *x*, and you also know the minimum fare for returning to *x* from any in-network station. You are also given a list of 2-way dedicated cycling routes connecting pairs of in-network stations. Complete the function, *cheapCycling*, so that it efficiently finds the cheapest cyling journey and returns the pair of distinct in-network stations corresponding to this journey. Here, a journey consists of 1) arrival at an in-network station from *x*, 2) some amount of cost-free cycling using dedicated routes, and 3) return to *x* from an in-network station which is distinct from the arrival station. The train stations are numbered from *0* to *N-1*, and the function is provided two *N*-element lists as input (*Slist* and *Clist*). The *ith* element of *Slist* contains two numbers – the minimum arrival and return fares for routes between stations *i* and *x*. *Clist* is an *adjacency list*. Its *ith* element contains a list of integers which correspond to in-network stations that can be reached directly by bicycle from station *i*.

Note for Part 1: You are not allowed to use *heapq*, however, if you determine that the best approach to a problem requires a binary heap, you should choose this approach, implement it without a heap, and then discuss the benefits of the heap (and its influence on the asymptotic running time) in your analysis.

Part 2.

1) (5 pts) For this question, you will investigate random walks on unweighted, undirected graphs. One step of a random walk is defined as follows. The probability of a walker moving from node *i* to node *j* is $p_{ij}=A_{ij}/q_i$pij=Aij/qi where A_{ij}Aij is the adjacency matrix for the graph, and $q_i$qi is the degree of node *i*.

i) Complete *rwgraph* so that it efficiently simulates *M Nt*- step random walks on the NetworkX graph provided as input. All walkers should start at the same node with the this node set via input variable, *i0*. Provide a 1-2 sentence description of your algorithm for 1 simulation step in your *pdf*.

For the next two questions, you should use Barabasi-Albert graphs with *n=2000* and *m=4* (see NetworkX documentation for the definitions of *m* and *n*). You should also fix *seed* so that the same graph is generated each simulation.

ii) Analyze your simulation results as *Nt* and *M* are increased. The initial positions for all walkers should be the node with maximum degree. If multiple nodes have the same maximum degree, select the node with the smallest node number. Compare large-*Nt* results to the node degrees for the graph. Make 1-2 figures supporting your main findings, and place them along with accompanying discussion in your *pdf*. The code used to generate the figures should be placed in *rwgraph_analyze1*

iii) Recall that linear diffusion on an unweighted graph was defined using the graph Laplacian matrix, $L=Q-A$L=Q−A where *Q* is a diagonal matrix with the graph degrees on the diagonal, and *A* is the adjacency matrix. The scaled Laplacian, L_sLs, is defined as $L_s=I-Q^{-1}A$Ls=I−Q−1A. Consider the linear spreading produced by both this operator and L^T_sLsT. Investigate if any one of these three diffusion operators produces dynamics with significant similarities to your simulated random walk results (all generated using the same B-A graph).

Identify 2-3 key points and provide explanations of what you have found in your *pdf*. These may be accompanied by figures as needed. Place any relevant code in *rwgraph_analyze2*. You are not required to establish quantitative correspondence between simulated diffusion and random walks on a time-step by time-step basis.

2)(5 pts) Consider the following two models for transport processes on graphs:

Model A:

$$\frac{di_j}{dt} = -\beta i_j + \sum_{k=0}^{N-1} \gamma A_{jk} i_k \left(1 - i_j\right)$$

Model B:

$$\frac{ds_j}{dt} = i_j$$

$$\frac{di_j}{dt} = \sum_{k=0}^{N-1} \alpha L_{jk} \left(s_k - s_j\right)$$

Here, L_{jk}Ljk is the Laplacian matrix and A_{jk}Ajk is the adjacency matrix for an N-node graph.

i) Complete *modelA* so that it computes simulations of *modelA* on the NetworkX graph provided as input. Complete the function, *RHS*, so that it computes and returns the right-hand side of the model equations (see the function docstring). You should assume that RHS will be called a very large number of times within one call to model A and construct your code accordingly. Construct an estimate of the number of operations required to compute $di_j/dt, j=0,...,N-1$dij/dt,j=0,...,N−1 in one call to RHS. Add this estimate to your *pdf* along with a concise explanation of how it was constructed. Add the needed code below *RHS* to simulate the model for *Nt* time steps from *t=0* to *t=tf* with the initial condition $i=i_0$i=i0 at node *x* with *x* and i_0i0 provided as input, and finally return $i_j(t)$ij(t).

ii) Investigate the similarities and differences between the dynamics generated by model A, model B, and linear diffusion on Barabasi-Albert graphs. You can initially consider $\alpha=-0.01$α=−0.01 (previously was $\alpha=0.01$α=0.01), $\beta=0.5$β=0.5, and $\gamma=0.1$γ=0.1, though you are free to vary these parameters as you like. You should initially (at *t=0*) set all variables to zero except for *i* at the node with maximum degree.

You should design your own approach to the problem, and identify 3-4 key points, carefully discuss them (and provide explanations for highlighted trends), and produce a few figures illustrating numerical results related to the key points. Among the points you could consider is the initial spread of a perturbation placed at some node *x*. Add code for generating your figures to the function, *transport* and add the figures and accompanying discussion to your *pdf*. It is sufficient to consider B-A graphs with *n=100* and *m=5*.