COMP0005 Individual Coursework: The London Railway Network Report

Ce Cao

Data Structures

During planning and evaluation phase I browsed the tasks and initially decided to use *Graph* as main data structure, because there are shortest-path problems and TSP problems that are related to graphs. *Adjacency Matrix* or *Adjacency List* are the most commonly used representations of a graph. Since what is going to be realized is a graph of railway network, the number of edges (in this case, 976) is much smaller than the square of number of vertices (653^2), so most of the matrices would be "INF" if we decided to use an adjacency matrix, which causes a huge waste of space. Therefore, the adjacency list is used, which is represented by a d*ictionary* of a dictionary. Self-created *Vertex* classes are the keys of the outer dictionary, vertices connected to each vertex are the keys of the inner dictionary, weight of each edge connect two vertices is the value of inner dictionary, and these key-value pairs are the values of outer dictionary. In addition, *Queue* data structure with O(1) complexity of pop() operation and *Heap* data structure with O(logN) complexity of pop() and insert() operation are also implemented to improve algorithm efficiency.

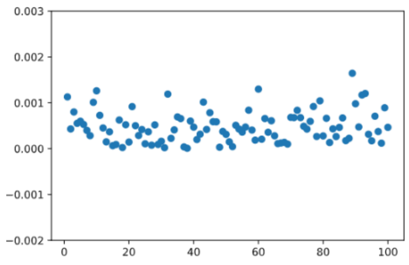
APIs *(Note: Assume station names will always be from the londonstations.csv file)*

1. loadStationsAndLines()

The function of this API is to construct the graph of the transportation network according to the topological relationship of the actual railway network and initialize anything needed for further development. Use csv library to iterate through each row and read from csv files, and then use haversine() function to calculate the distance between two points on earth and finish building the graph. One file stores the location information of vertex (station), the other stores the information of edges (pairwise connections). Since there is no requirement on "transfers", different tube lines are ignored when constructing the graph, however the information is stored in dictionary "stationLine". The overall computational complexity would be O(V+E) (V: number of vertices; E: number of edges, same below). For testing loadStationsAndLines() method, since there is no parameter representing input file, I modified different csv files of different rows, changed the name of file from code "with open('xxxx.csv', newline='') as csvfile:" above and make the test program repeat 100 times. The result meets the pattern. Unfortunately, due to the limitation of no csv file can be uploaded and no changes of skeleton code in Jupyter Notebook, I had to delete the code because it would generate error alert. Same problem with the other methods, only with distinct graphs, the number of vertices and edges can vary and the tests on algorithms can actually be meaningful. We cannot simply generate random graphs either, the structure would not be same as a "Railway Network Graph", hence tests are still meaningless. As a result, I can only demonstrate tests on the actual API - showing actual cost on the given graph, comparing between different algorithms I implemented by fixing the graph and varying the input. The average loading time is 0.005s. However, if the user requires to use an alternative approach for minDistance(), a two-dimensional list would also be constructed here with time complexity of O(V^3) and space complexity of O(V^2). The average loading time is 48.709s. More details will be discussed later in the report.

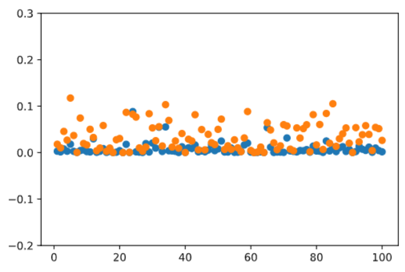
2. minStops()

The graph can be seen as unweighted and undirected graph in this API. We can use depth-first or breadth-first search here, and I choose BFS. In theory they have the same complexity of O(V+E), but BFS is faster because it ends immediately after the destination is found while DFS does not. We do not want to use algorithms like Dijkstra because without optimization, the complexity is O(V^2). Experimentally, the "while" loop visits each vertex at most once with complexity of O(V). The "for" loop nested in "while" makes each vertex dequeued at most once, so edges are checked at most once with complexity of O(E). So overall it has linear time complexity of O(V+E). Testing it 100 times and the average searching time is 0.0003s.



3. minDistance()

We cannot use BFS in this API because BFS only returns what has least edges when it is unweighted. With weighted graph the route that has least edge may not be the route with shortest distance. Based on the breadth-first algorithm, Dijkstra algorithm uses greedy algorithm to improve, from starting point as the center to expand. Each iteration the current shortest path is calculated and stored, and the shortest path is searched each time until it expands to the end point or every vertex is visited. Using heap data structure as an optimization when fetching and storing the nearest vertex can reduce its complexity to O(ElogV). There is also Bellman-Ford algorithm of the same type as Dijkstra (solving single-source shortest path), but its efficiency is lower with the time complexity of O(VE), and since the subway line has no negative weight edges, so it is not considered in our solution. For the Dijkstra I found that one of its drawbacks is that the search is blind and not directional. For example, if we want to find out the shortest distance from London Bridge to King's Cross, we do not actually need to consider station in the opposite direction, that is, to the south of London Bridge. But the algorithm does not know. What is more beneficial for the algorithm is to check the stations to the north first, since we know it is the direction of destination. we can greatly reduce the amount of time needed if we modify Dijkstra with some heuristic h(n) like the distance from one station v to destination in this case, select the path that minimizes f(n) = g(n) + h(n) where n is the next vertex on the path, g(n) is the cost of the path from the starting point to n, and h(n) is a heuristic function that estimates the cost of the cheapest path from n to the goal. This is basically the A* algorithm. If make h = 0, we get f = g, which is the Dijkstra algorithm. For A* implementation of this graph, first we mark starting point as "searched", then we calculate the weight of vertices adjacent to starting point with haversine() function (If on two-dimensional plane we can use Manhattan Distance: $|x1 - x2| + |y1 - y2|$ instead of Euclidean Distance: $\sqrt{(x1 - x2)^2 + (y1 - y2)^2}$ to improve efficiency). The specific calculation method is haversine(s, n) + haversine(n, d), where s is starting point, n is the vertex and d is destination. Then we choose the vertex with minimum weight, mark it as "searched", calculate the weight of vertices adjacent to "searched" points, choose again and repeat the process until reach the target. A* is not perfect, however. It requires some sort of estimation to form the basis of heuristic. When it is not possible to do so, we cannot use A*. If the difference between the estimated value and the actual value is too large, it may be less efficient than the Dijkstra algorithm, or even impossible to get the correct answer. But in this subway system case, it is relatively simple, and the result is always correct. Experimentally I implemented both A* and Dijkstra, average searching time for A* (0.008s, blue in the figure) is less than Dijkstra's (0.040s, orange in the figure), it is indeed more efficient than Dijkstra, so I settled on A* for this API.

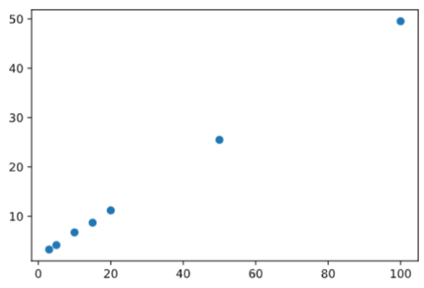


During implementation, it comes to my mind that railway network is relatively regular, and the frequency of changes is low (no need to reconstruct graph often), another method is to trade space for time - calculate the shortest path between any two points in advance and store it in a table somewhere. It only takes O(1) time to look up the table when the user searches. This can be achieved with the multi-source shortest path Floyd algorithm. The core idea is either the shortest path from A to C is the shortest path found so far from A to C, or the shortest path from A to B plus the shortest path from B to C. So, we create a table which records shortest distance between any 2 stations, update it each iteration of start vertex, transfer vertex and ending vertex. The complexity of Floyd is O(V^3). Of course, we can also do Dijkstra for each point, because Dijkstra can get the shortest distance from the desired point to all points, and it has a lower complexity of O(EVlogV), but Floyd is much more elegant with only five lines of code. The average searching time is only 8.5e-7s. I commented out the test of this method because the loading time would be nearly 48s. However, it only needs to load once, and store the table elsewhere for further use that may reach tens of thousands of times. How to choose between these two methods depends entirely on user requirements. Here we are calculating the shortest distance, without taking transfer time, transfer route, or the flow of people at different times into consideration. We will have more complicated algorithms in real-world situation. If transfer is considered, an improvement is to use different nodes representing the same transfer station of different lines, and then set a larger distance between these nodes.

4. newRailwayLine()

The most intuitive way to solve this problem is brute force algorithm, enumerating all possible permutations with a complexity of O(n!) - its efficiency is so low that even if there are only 10 stations, it will have to iterate 3628800 times! However as mentioned before, the tube lines are already compact and fixed, it is not realistic to select stations from existing stations and create new lines on a large scale. The most likely way is to select a small number from existing stations then build some new stations to create a new line, or just build brand new stations. So, when the scale is small, we can actually consider this kind of solution. With the idea of Dynamic Programming, we can create a table[s][i] that records cost of the minimum cost path visiting each vertex in subset s exactly once ending at i. This gives us O($2^n n^2$) time complexity and O(2^n) space complexity performance, which is better than O(n!). But if n larger, it still becomes a computer's nightmare. This problem is a NP-Hard problem, which means that there is no known algorithm to arrive at precise solution in polynomial time. Greedy algorithm is what came to my mind next, by randomly choosing a starting point, and then choose the shortest path to move forward. Even if we iterate through all starting point, the time complexity is only O(n^3). But the problem is that it only selects the currently optimal solution, and it is easy to get only the local optimal solution in the end. This local optimal solution may not be the global optimal solution. Then I did some research, and settled on Simulated Annealing algorithm, which is derived from the principle of solid annealing. The solid is heated to a sufficient height, and then slowly cooled. When heating, the internal particles of the solid become disordered with the temperature rise, and the internal energy increases, while the particles gradually cool down. It tends to be ordered, reaching an equilibrium state at every temperature, and finally reaching the ground state at room temperature, and the internal energy is reduced to a minimum. The Simulated Annealing algorithm is like a clever version of Greedy algorithm, the core idea is accepting a solution that is worse than the current solution with a certain probability, and then use the worse solution to continue searching in order to avoid the shortcoming of simple Greedy algorithm. At first create a path, calculate the approximate cost of it with Manhattan distance, then create a new path with randomly swap two stations, calculate the cost again, if the cost is lower, then accept it, otherwise accept with Metropolis acceptance rule (figure on the right) or discard it. Repeat the progress until temperature smaller than 1e-8. The performance of it depends on the set of initial value of temperature T0, number of iterations L and the annealing coefficient q. Normally if T0, L or q is high, it is more likely to find the global optimal solution, but it will take a lot of calculation time; otherwise, the calculation time can be shortened, but it will be less likely to get best solution. In actual application process, these parameters generally need to be adjusted several times based on the experimental results. For example, if length of inputSet n is small, we can obviously change to smaller parameters. However, it’s not possible to change when testing, because this API only has one parameter: inputSet, so I set q = 0.98, L = 800, T0 = 100 for testing. During each iteration, the complexity will be O(n), because I use slicing operation with complexity of O(n) when creating new list, the complexity of calculating costs is O(n), and complexity of creating new stationList is O(1). Hence the overall complexity would be O($\log_q^{1e-8(L/T0)}$ n). The constant $\log_q^{1e-8(L/T0)}$ is about 800. This is much more efficient than the traditional brute force or DP algorithm. Experimentally, I test the case when n = [3,5,10,15,20,50,100], and the execution times(s) = [3.2562063999999964, 4.150580899999994, 6.72751550000001, 8.697102700000002, 11.184402500000004, 25.48449819999999, 49.52347520000001]

$$p = \begin{cases} 1 & \text{if} \quad E(x_{new}) < E(x_{old}) \\ \exp(-\dfrac{E(x_{new}) - E(x_{old})}{T}) & \text{if} \quad E(x_{new}) \geq E(x_{old}) \end{cases}$$



Similar to this solution, some other mainstream methods at present also use some random and heuristic search algorithms, such as Genetic algorithm, Ant Colony Optimization, Particle Swarm Optimization and so on. These algorithms all have one shortcoming, that is, they may not be able to find the optimal solution but can only converge to (approach) the optimal solution, and maybe get a sub-optimal solution, because they are all essentially random algorithms.