***GRAPH ADT DESCRIPTION***

There is an almost endless amount of choices for representing a graph with a data structure. You can use various combinations of arrays, types of lists, queues, maps etc. All have various advantages and disadvantages. This resulted in there being 2 guiding concepts on how to implement a graph in code, an adjacency matrix and an adjacency list. The matrix-style is normally used for a dense graph, where the number of edges approaches number of vertices squared. The simplest way this can be done is by using something along the likes of a 2D array. The matrix-style methods have super-fast indexing and ease of use, but the downside is that they have a ginormous memory cost, particularly for bigger graphs. An adjacency list is normally used for a sparse graph. These need slightly more complicated index management and take more effort to maintain, however the space cost is massively reduced. As far as indexing speed goes, it depends on what implementation you go with.

The Boston metro system we are working with is a sparse graph. For this reason, we have gone with the adjacency list concept. For the implementation, we decided to use a map, specifically a HashMap. It will have a key of a node’s (called station in our case) ID and a value of an array of edges (called tracks) connected to it.

An edge will know what line they are on, IDs of the connected nodes and their own weight (all of which are set to 1 for this exercise to avoid any weird edge cases where the algorithm takes a longer path). Edges will also have fields for the names of arriving and leaving nodes that are filled out once the algorithm has worked out a route. This is because we only return a linked list of edges that the frontend uses to display the route.

A node will know its ID and what edges are connected to it. All nodes will also have a weight that is added on only when switching lines, in order to make sure there are as few line switches as possible. We will use all this information to build our search algorithm for the graph. We will be using Dijkstra’s algorithm for its ability to traverse uninformed, weighted graphs.

The are a few reasons we chose to use a HashMap with a numerical node ID mapped to an array of edges. The HashMap is an officially supported library implementation and is therefore likely to be very fast, efficient and reliable. This gives us more flexibility to implement our own search algorithm in a more efficient and reliable way. We will be using an Arraylist over a standard array to store edges as it is extremely useful when building the graph. We do not know how many edges are connected to each node on the graph and therefore having it dynamically adjust its size will save some hassle. Therefore, whilst standard arrays have their benefits i.e. less overhead and more speed, an Arraylist will end up saving us on complexity with its easier indexing control and ability to auto adjust its size. For this implementation and size of graph, the slightly slower speed of an Arraylist is pretty much irrelevant.