Algorithms Design Document

## Shortest Path between 2 bus-stops

#### Underlying data-structure

The underlying data-structure used to implement the shortest path algorithm is an edge weighted digraph with an adjacency list graph representation, with stops as vertices and trips between stops as edges. While this would usually be implemented using arrays with the vertices as indexes, this was not possible in this case, as the stops were not numbered from 0 to n, and were not sequential in number. So instead, wherever a vertex indexed array would be used, a hashmap with the stop number as key would be used.

The adjacency-list representation was used as it has the best time/space trade-off, taking up ~(E + V) space (where E is the number of edges and V is the number of vertices) while only taking ~degree(v) time to iterate over vertices adjacent to v. While the list of edges representation only takes up ~E space, it takes ~E time to iterate over the vertices adjacent to v, which is too long as this is the action taken most often in this application. The adjacency matrix representation is entirely inappropriate taking up space ~(V²). So I found the adjacency-list representation to be most appropriate.

#### Underlying algorithm

I chose to use Dijkstra's shortest path algorithm to find the shortest paths between two stops, as it is the best algorithm for single source shortest path with no negative weights. I have implemented it with what is essentially an unordered array as it is the most feasible to implement with input as complex as that which is present here.

#### Adding efficiencies

I added a class to provide a way to increase the overall efficiency of the application. This stores the edgeTo and distanceTo HashMaps for any source vertex shortest paths which have already been calculated. This means a shortest path is never calculated more than once.

## Searching for bus stop by name

#### Underlying data structure and algorithm

The underlying data structure for this part of the application must be a ternary search trie (TST) as defined in the assignment. The implementation was taken directly from the book, as recommended. This is the most efficient overall for this application as almost all of the processing time is spent in building the trie, with each search being very efficient. This also takes up only 4N space.

## Searching for all trips with a given arrival time

#### Underlying data structure and algorithm

I also chose to use a TST for this part of the application, where the string key is the arrival time, and the value is an ArrayList of Trips. I chose this as it combined sorting and searching into one data structure. For the ArrayList of returned trips, I sorted them using insertion sort, as it is the most efficient for small arrays.