Software Design Document

Purple Parrots
Leighann Astolfi - astolfi.l@husky.neu.edu
Ryan Bigelow - bigelow.r@husky.neu.edu
Jeffrey Guion - guion.j@husky.neu.edu
Nicholas Labich - labich.n@husky.neu.edu

To Dr Jessica Young Schmidt,

The purpose of this memo is to discuss the basic inputs, outputs, data structures and algorithms that will go into the MBTA Real Time Tracker application. This is only a high level overview and not a detailed design document. The items in this memo are subject to change with implementation. Detailed data structures and algorithms are listed in full at the bottom.

There are a few basic types of requests that a user can do. First, a user can request to see where they can go on the T. In this case, a request for JSON data is not needed. Just a simple map of the T would work. It would be best to have an instance of the T created when the application starts. This would consist of the following data structures: a class Station and a class TMap. The Station would have its name, a list of colors for the lines it is on and a list of adjacent stations. Each Station will be part of a Line. A Line will be an list of Stations in the order they appear on each line. All of this information would static and hard coded so they need to be maintained if any changes occur to the T line. The output of requesting where a user can go on the T would be a map displayed visually on the screen.

The next case is if a user asks for the current location of all of the trains. To do this, first we need a Train class. The Train class would have a color for the line it is on, its end destination and an ordered list of predictions which consist of a station and an estimated amount of time to get to that station. To get the next location of all the trains, the system must make a request to http://developer.mbta.com/lib/rthr/{{line}}.json for each of the lines. To get the next location, loop through each trip in the Trips array (of the JSON data), the first item in the Predictions array (from the Trips array) will be the stop that the train is arriving at next. The current location of the train will be between the first station listed and the station adjacent to it the opposite direction of the end destination. To display the current location, overlay each train onto the map at its current location.

A similar case to the previous one would be if a user asks when the next trains will get to a specific stop. This case would require a user to input a stop. They will also be asked to enter a line and a direction to narrow down their choices for the application to show the most relevant data to them. We can return to the user a sorted list of times. To get this list of times, request json data for the specific line inputted by the user. To get the next arriving trains, loop through each trip in the trip list. If the destination matches where the user's inputted destination, loop through the Predictions array. If you find the stop in the array, add the predicted seconds to the list of arrival times. (Can optimize by short circuiting if predictions are headed in wrong direction).

Next are direction based cases. The first is the user wants to know how to get from stop A to stop B. This requires the user to input exactly two stops. They will be asked to enter a line for each stop to optimize the algorithm. To get the directions, we would use our knowledge of the T. It would be quickest to already have a HashMap that mapped each station's name to the color line it is on and the direction it is from the transfer station. This way, our search algorithm can quickly know where to go. At a high level, how the algorithm will work is first, look up the line of the end destination. Then starting at the current station, go through the adjacency list of stations on the line. If the destination is on the current line, then search each station on the line until you find it. If it is not, search each station until you find a transfer station. If the transfer station is for the destination line, take it and iterate through the list until you find the stop. If the transfer station is not for the current line, continue to the end of the list. If you find another transfer station, take that. Otherwise, go back to the first transfer station and loop through until you find the correct transfer station. Instead of using Dijkstra's algorithm or another shortest path algorithm to find the path, I believe since we already know information about the T map (two transfer stations), we can create a better solution. We also need a Route class which has a list of Stations stopped at in the directions. Other meta-data should be included in the Route such as total time data and number of transfers. While looping through the stations, the Route is generated which will be the output of asking for directions.

The next case is a user wants to get to an ordered list of stops. This scenario is very similar to getting directions from Stop A to Stop B. As an input, the user can input any number of stops in the order which he would like to visit them. Using the same algorithm described in the previous case, for each stop in the list of stops, find directions from the current stop to the next stop (if there is one). Keep appending directions onto the Route object until the final destination is reached.

The next case is getting to an unordered list of stops. This is a little more complicated. The user will input any number of stops and indicate that order does not matter. In order to find the quickest route to navigate between them, we can form a graph where the cost of each edge is the number of stations between two stations. Each station would have an edge to each other station. The quickest route between the stations would be the minimum spanning tree of the graph. The minimum spanning tree would then be converted to a Route.

The next type of case is when the user wants to add constraints on their trip. The first is when a user wants to add a start time or an end time. Instead of displaying only a route based on the shortest path, the Predictions array from the JSON data needs to be used. If the user wants to add a start time, then, calculate the amount of time the start time is from the current time is in seconds. Then, filter the JSON data so any Trip where the delta seconds is less than the Predicted seconds for the starting stop is not included. Using the filtered JSON data, the Route will be created and the specific train ids will added to the trainIds list. Adding an end time will be more complicated. First, generate a Route based on the directions. Then, for each Trip that will arrive at the starting station, while there are still transfers left on the Route, calculate the time it will take to get to the next transfer station. For that transfer station, look at all the Predictions that will arrive on the transferred line. Filter out any predictions that are less than the amount of time it will take to arrive at that station (+1 minute for transfer time). Continue doing this until there are no more transfers left. Then calculate directions based on the filtered JSON data.

The other case is if the user wants to know the fastest route, the earliest departure, the earliest arrival, and fewest transfers. The fastest route case and the earliest arrival case are already accounted for in our algorithms which will return the fastest time. The earliest departure case, although I do not know why it would be useful to anyone, would only be different for getting directions to an unordered list of stations. In that case, it would create directions with a starting point being the station with the train that is predicted to arrive the soonest. In the fewest transfers case, for the list of stations, do not transfer until all of the stops on a specific line have been reached.

When the user wants to use the software with old data, the system should be able to read a specified input a file and parse it the same way as it would for the live MBTA JSON data. The software should be able to run all of the previously mentioned algorithms on the data and get the proper results or error messages.

If you have any questions, comments or concerns, please contact Jeffrey Guion at guion.j@husky.neu.edu or any other members of the Purple Parrots at the above listed email addresses.

Data Structures

Station	name	String	the name of the stop
	lines	List[String]	the names of the lines of this stop (at most two for DC and State)
Line	name	String	the name of the line, one of {"red", "blue", "orange"}
	stops	List[String]	the names of the stops of the line, ordered from one (arbitrary but internally consistent) end to the other
Train	id	String	the ID of this train (from the MBTA json)
	line	String	the name of the line this train is on
	destination	String	the name of the destination station
	predictions	List[Pair]	to reflect the json data, predictions would be an ordered list of a tuples which stores the future stops (so that the head of the list is the next stop) and the estimated time to each stop.
Route	trainIds	List[String]	a list (of at most three) of IDs of the trains which will be used on this trip, ordered such that the train used earlier will be at the head
	stops	List[String]	an ordered list of adjacent stops
	schedule	List[Int]	the number of seconds it will take to get to each of the stops in the the stops of this trip, in the same order (next at head)
	transfers	int	the number of transfers in a trip
	totalTime	int	the amount of time from now until arrival
	elapsedTime	int	the amount time from departure to arrival
	legs	List<Leg>	the list of two station sets that make up a route
Leg	trainId	String	the ID of the train that connects this leg
	startStation	String	the first station of this leg

	| endStation	: String	: the last station of this leg
	| line	: String	: the line that this leg is on
	| lineDestination	: String	: the direction this train is headed
	| startTime	: int	: the time that this leg is departing
	| endTime	: int	: the time that this leg is arriving
Prediction	| stopId	: String	: the id of this stop
	| stop	: String	: the name of this stop
	| seconds	: int	: how many seconds out the next train is from this stop

Algorithms

```
getAllTrains(line)
    train_list = []
    json = getJSON(line)
    triplist = json[“TripList”]
    for trip in triplist[“Trips”]
        train = Train(triplist[“Line”], tripList[“Destination”], tripList[“Predictions”][0]
        train_list.append(train)
    return train_list


getTrainsAtStop(stop, line, direction)
    json = getJSON(line)
    approachingTrains = []
    triplist = json[“TripList”]
    for trip in triplist[“Trip”]
        if direction != trip[“Destination”]
            continue
        predictions = trip[“Predictions”]
        for each stop in predictions
            if stop[“Stop”] == stop
                approachingTrains.append(stop[“seconds”])

getDirections(stopA, stopB)
    a = getStation(stopA)
    b = getStation(stopB)
    stations = []
    found = False
    // iterate_inbound will follow the adjacency to the next station.
    // based on knowledge of the transfer stations, it will switch lines if necessary
    for station in iterate_inbound(a, b.color)
```

```
            if a.name = station.name
                found = True
                return new Route(stations.append(station))
            else
                stations.append(station)
        if not found
            stations = []
            for station in iterate_outbound(a, b.color)
                if a.name = station.name
                    found = True
                    return new Route(stations.append(station))
                else
                    stations.append(station)
        if not found
            raise Error

    getOrderedDirections(stations)
        ss = []
        for i = 0 , i < len(stations) - 1, i++
            ss.append(getDirections(stations[i], stations[i+1]).stations)
        return Route(ss)

    getUnorderDirections(stations)
        //forms a graph where the edges are the number of stops between each station
        // in the list
        g = calculateGraph(stations)
        //creates a minimum spanning tree for the graph
        mst = calculateMST(g)
        //creates a route by picking one end of the MST and traversing it
        return generateRouteFromMST(mst)
```