Algorithms COMP0005, Individual Coursework | Arslan Aftab 20006029 | Analysis Report

The London Railway Network

Introduction
The aim of this assignment is to efficiently compute various factors given a train network as a graph. This report will highlight some subtleties regarding data structures and algorithm choices in pursuit of efficiency, as well as a comparison/justification with respect to alternative options. The report will also encompass the theoretical and experimental (real life) computational complexity of the implementation.

Representing the London Railway Network
In order to implement the railway structure, it seemed intuitive to represent the graph, stations and locations using classes. These classes provide a wrapping interface for implementing the API.

- A location object stores longitude and latitude.
- A station object has a name, a location object, and a dictionary of neighbouring stations. The dictionary has entries as such; stationName: (distanceToStation, railwayLine).
- The station graph itself is a dictionary of station objects, and has entries as such - stationName: stationObject. Size is an instance variable for *O(1)* size lookup.

The reasoning behind using dictionaries was simply for *O(1)*[‡] lookup of stations based on their name, this is something we heavily depend on, therefore a crucial factor in maintaining performance. Distances to neighbouring stations are computed as each station is initialised, this is a compromise for some memory, but the reward is never having to compute distances at runtime again, thus distance lookups between two neighbouring stations is *O(1)*.

Calculating distance between two locations
Assuming the earth is a perfect sphere, the Haversine formula determines the distance between two points on a sphere traveling along the surface. This is close to the true distance between stations, variations in curvature of the earth are not noticeable at the scale of a metropolis.
Given two location objects, we extract their longitude and latitude, and apply functions sequentially from the standard math library in order to reproduce the Haversine formula and thus calculate our answer in metres. This is converted to miles and rounded to 4 dp.

Some Important Metrics: Time Complexities, with Amortized worst cases

Data Structure	Operation	Average Case	Worst Case
Dict	Check Membership	*O(1)*	*O(n)*
	Get/Set Item	*O(1)*	*O(n)*
	Iterate all keys/values	*O(n)*	*O(n)*
Set	Check Membership	*O(1)*	*O(n)*
	Insert Item	*O(1)*	*O(n)*
List	Append	*O(1)*	*O(1)*
	Get/Set Item	*O(1)*	*O(1)*
	Iterate all	*O(n)*	*O(n)*
	Check Membership	*O(n)*	*O(n)*
	Take slice	*O(k)*	*O(k)*
Deque	Append left/right	*O(1)*	*O(1)*
	Pop left/right	*O(1)*	*O(1)*
Heapq	Push/Pop	*O(log n)*	*O(log n)*

[‡] *At multiple times during the analysis we will mention dict/set operations which have average performance O(1), but keep in mind that it may deteriorate to O(n) upon collisions when hashing.*

LoadStationsAndLines: Theoretical Time Complexity[1]
We begin by iterating each station from the `londonstations.csv` file, creating a station object and its location object before adding that to our stations dictionary. Creating a location object and station object is both constant time, inserting our station object into a dict has average complexity *O(1)*[‡]. Thus our theoretical average and worst case case would be *Θ(S)*.

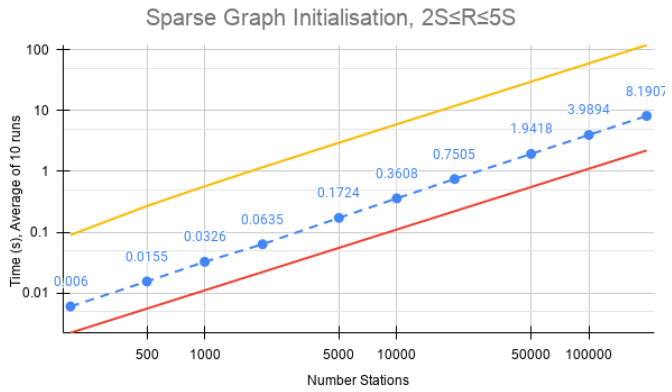
Then in order to create all connections between stations, we take each pair of connected stations and calculate the distance between them in constant time. We then create an outward connection for both stations to its paired station, since this is an undirected weighted graph; thus having to call `addRoute` twice giving a complexity of *Θ(R)*[‡]. Dictionaries perform much better than lists because

[1] For this report we will assume that S and R when mentioning complexities, both experimentally and theoretically, refers to the number of stations and routes (i.e. Vertices and Edges)

we do not need to linearly scan all stations in the graph to fetch a station object. In the average case we will be able to lookup a station in constant time given no collisions.

Thus this produces an overall theoretical average and worst case $\Theta(S) + \Theta(R) = \Theta(S+R)$, but may take $O(S^2) + O(R^2) = O(S^2+R^2)$ for initialising our data structures if collisions occur at every stage. This is very unlikely and will be as a result of python's implementation of dicts. Keeping in mind that the $O(R)$ itself may vary between[2] $O(S)$ and $O(S^2)$ which can be interpreted as the worst case.

LoadStationsAndLines: Experimental Time Complexity (Images full size in references)[3]



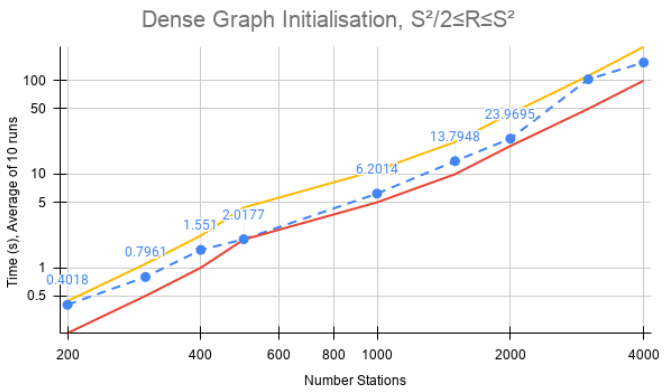


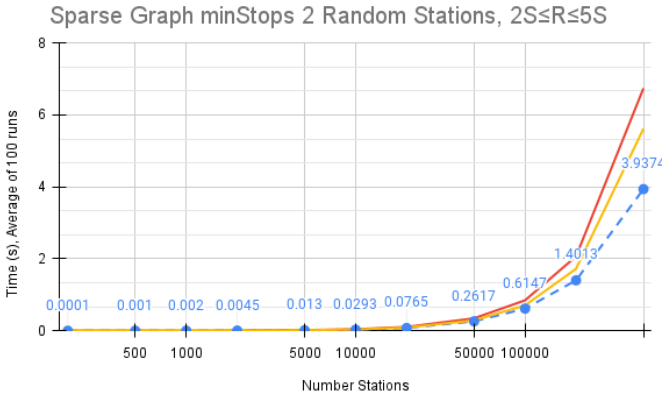
Figure 1 & 2 , Each time result is the average of 10 initialisations. Sparse Graph bounded between runtime 1e-5(S) and 6e-3(S). Thus our average case has linear growth. Dense Graph bounded between runtime 1e-3(S) and 1e-9(S^2) + 2.2e-3(S), as mentioned - quadratic growth.

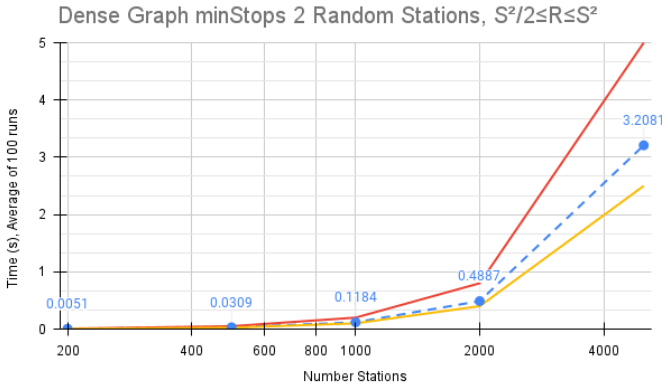
MinStops: Theoretical Time Complexity

In order to find the minimum number of stops connecting two stations, the logical solution is to utilise a queue based Breadth First search, we start at the `fromStation` which has a distance of 0 hops to itself and discover stations in an increasing number of hops from where we are. We would visit each node at least once, and mark it as visited which has $O(S)^{\ddagger}$. We would also have to explore every outgoing edge and add unmarked stations to our queue (pushing/popping from the queue is O(1)) which takes $O(R)^{\ddagger}$. As we discover every station and its routes, we construct a dictionary that maps each station to its distance from `fromStation` in hops which takes $O(S)$. This assumes that there are no collisions when hashing into our dict‡. To then find the number of hops connecting us to `toStation`, we simply run a get operation to fetch the stored value.

Instead of using a normal dict, which we would have to initialise by iterating each station which is $O(S)$, we instead use a default dict that means unless we state otherwise; every station has a default distance of -1 from `fromStation` thus we do not need to have any initialisation of our dict. In the average case, our dictionary set and get operations would take $O(1)^{\ddagger}$ thus resulting in $O(S) + O(R) = O(S+R)$. Again, the worst case arises in the condition that $|R| = |S^2|$, thus giving worst case $O(S) + O(S^2) = O(S^2)$ for a dense graph. This may deteriorate to $O(S^2+R^2)^{\ddagger}$.

MinStops: Experimental Time Complexity





[2] Assuming a station has at least one connected station, which is a given for a train network.
[3] Sparse refers to a graph (experimental testing purposes) in which every station has 2-5 routes, whereas dense graphs have stations with $S^2/2$ to S^2 routes per station.

Figure 3 & 4 , Each time result is the average of minStops between 100 random stations. Sparse Graph bounded between runtime 2.5e-7($S^{1.29}$) and 3e-7($S^{1.29}$). Thus our average case has growth reflective of (R+S) since 2S ≤R≤5S . Dense Graph bounded between runtime 2e-7(S^2) and 1e-7(S^2), as expected-quadratic growth.

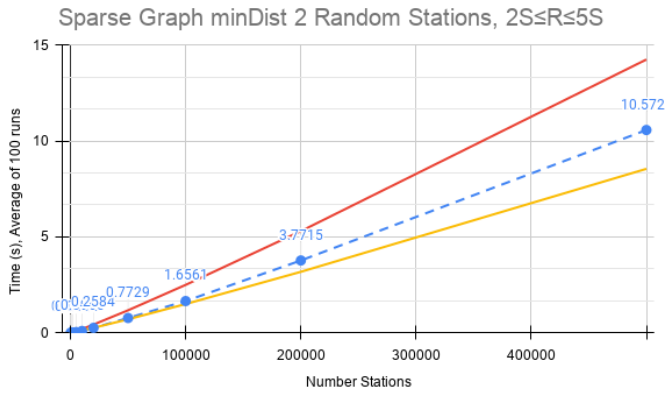
MinDistance: Theoretical Time Complexity

In order to find the minimum distance connecting two stations, it seemed intuitive to use Djikstra's shortest path algorithm. This is implemented using a binary heap which shines in its efficiency for sparse graphs. This is a clear choice since a dense graph, i.e. one where each station connects to many others is inefficient in the real world, each station in our network connects to ~ 2-4 others.

We track all visited stations using a set, which allows us to track in $O(1)^{\dagger}$, which is clearly faster than a list which has $\Theta(S)$ for checking membership. We track distances to stations from `fromStation` using a dict which has set/get in $O(1)^{\dagger}$. Again the use of defaultdict allows to imply all distances are initially infinite without having to initialise via iterating all stations which has $O(S)$. Each time the main loop runs, we extract a station from the heap and its distance, starting with `fromStation` which has a distance of 0 to itself. There may be up to S stations in the queue, thus giving $O(S)$ and extracting from the heap[4] takes $O(\log S)$ ignoring previously visited stations in $O(1)^{\dagger}$. Thus the main loop can be completed in $O(S \log S)$. We then consider all the unvisited connections of the current station and determine whether we can change it to a shorter route, and update our dict in O(1)†, if we must again push to the heap it can be done in $O(\log S)$. We must consider all edges as such, giving $O(R)$ deteriorating to $O(S^2)$ for a dense graph (which is why we can safely use it, as our graph has |R| << |S^2|) and thus this step can be completed in $O(R \log S)$. Thus $O(R+S) * O(\log S)$ gives us the average and worst case complexity of $O(R \log S)$.

Using an unordered array, our overall time complexity would have shot up to $O(S^2)$ since popping the minimum priority station would have taken $O(S)$ via a linear scan. We can safely use a heap as R log S << S^2 in our sparse graph. Djiktra's is dominated by operations on the priority queue.
I opted against using A star for the reason that the heuristic would heavily depend upon calculating the distance to `toStation` to judge the cost traversing between nodes, which involves multiple floating point and mathematical operations and may be severely taxing to a system.

MinDistance: Experimental Time Complexity



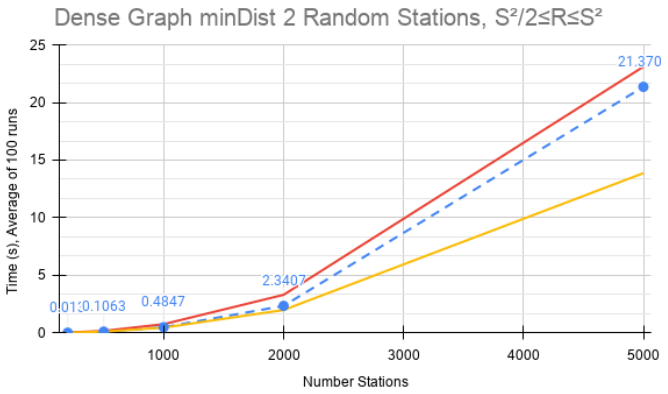


Figure 5 & 6 , Each time result is the average of minDistance between 100 random stations. Sparse Graph bounded between runtime 3e-6(R log S) and 5e-6(R log S). Thus our average case has linearithmic growth. Dense Graph bounded between runtime 1.5e-7(S^2 log S) and 2.5e-7(S^2 log S), as expected since R ≈ S^2.

NewRailwayLine: Theoretical Time Complexity

To construct the new railway line we begin by constructing a smaller graph of only the relevant stations. This is a graph, where each station connects to every other station. We take every station from the list and make a copy of it, ignoring its connections which can be completed in $\Theta(S_L{}^5)$. Grabbing a station from the graph takes $O(1)^{\dagger}$ and hence grabbing its location object also takes $O(1)^{\dagger}$. We iterate each station which takes $\Theta(S_L)$ and make a copy of `inputList` excluding the current station which can be completed in $\Theta(S_L)$ since we splice up until the station and after the station (1, 2, …, k-1 and k+1, k+2, …, n). We then iterate every station in the sublist, $\Theta(S_L)$ and

[4] Keeping in mind that Djikstra's observes the smallest distances first, disregarding whether they lead towards the destination or not.
[5] Note that S_L is the number of stations in the `inputList`, likewise R_L is the number of routes.

connect it to our current station via addRoute which takes $O(1)^{\ddagger}$. Thus we can construct this temporary graph in $O(S_L) + O(S_L^2) = O(S_L^2)$. This is just a nested for loop.

Now that we have such a graph, we can construct its minimum spanning tree using a lazy Prim's algorithm. We also construct an unconnected (initially) graph with copies of the stations and their locations which takes $\Theta(S_L)$[6]. We track the visited stations using a set for $O(1)^{\ddagger}$ membership checking, and again use a min heap to view routes in order of smallest distance. While there are routes we have not considered, $\Theta(R_L)$ we pop a route of the min heap which takes $O(\log R_L)$ [7]. We ignore routes we have considered already $O(1)^{\ddagger}$. For every route we add it to our graph which takes $O(1)^{\ddagger}$. If either of the two stations it connects are unvisited then we visit them, and push their outgoing routes to our priority queue and mark them as visited. This step takes $O(R_L \log S_L)$. We also keep track of the last station we popped, and store that as a variable which takes $O(1)$. Thus we can find the MST in $O(R_L \log S_L) + O(S_L \log S_L) = O(R_L \log S_L)$.

Now that we have found the MST, we can simply traverse the MST starting at the final station which we popped from the priority queue from using Depth First search. We start at the given node, and add it to the set of visited stations $O(1)^{\ddagger}$ and the ordered list with $O(1)$. We then recursively travel as far from that station using the first available route until we can no longer, and backtrack once we have exhausted the unvisited stations down that path. This stage takes $O(S_L + R_L)$ at most. We maintain the visited set and ordered list separately because membership checking takes $O(1)$ in sets but our API demands we return a list, thus converting our set to list would carry a cost of $O(S_L)$. We can compromise on some space and simply maintain the list in parallel.

Thus, this gives a complexity of $O(S_L^2) + O(R_L \log S_L) + O(S_L + R_L)$. Thus $O(S_L^2)$

NewRailwayLine: Experimental Time Complexity

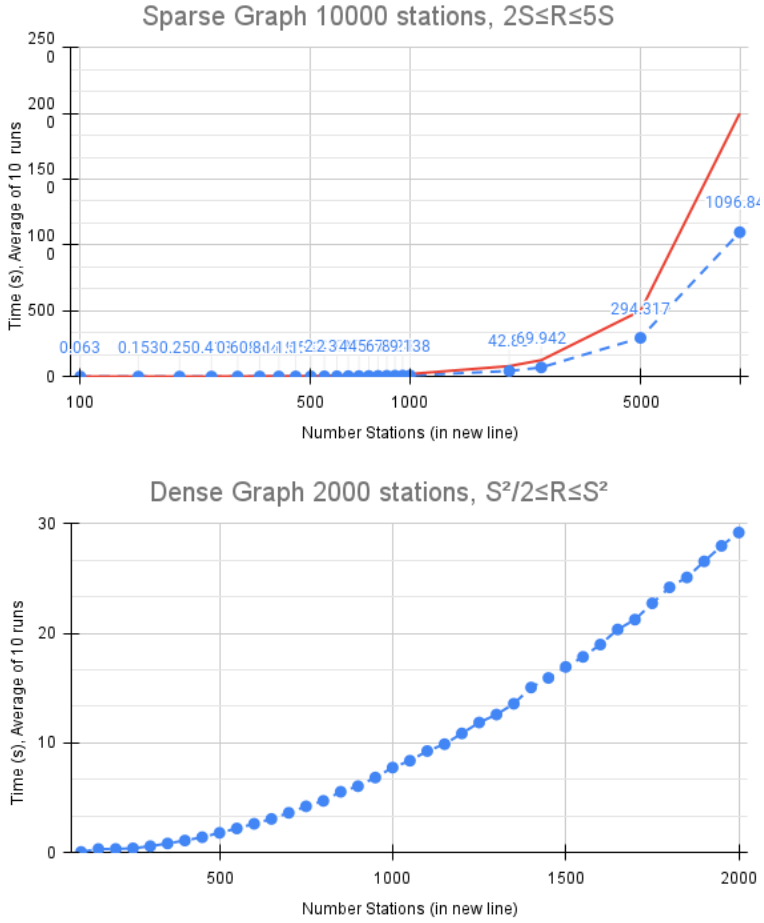
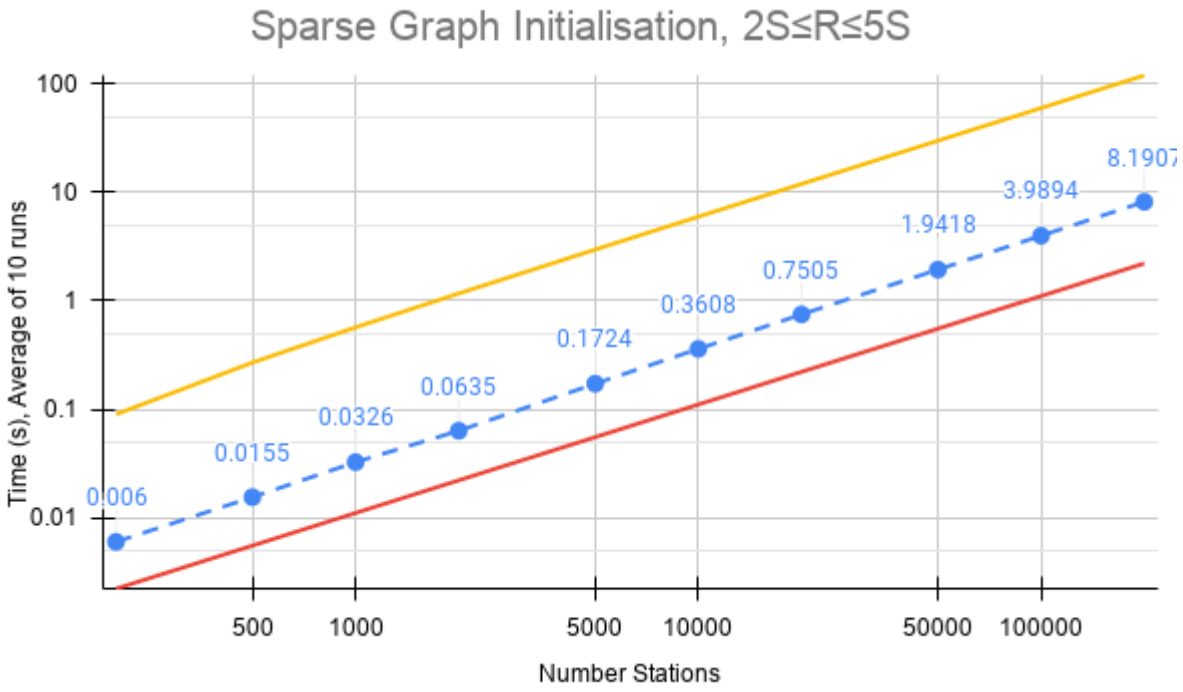


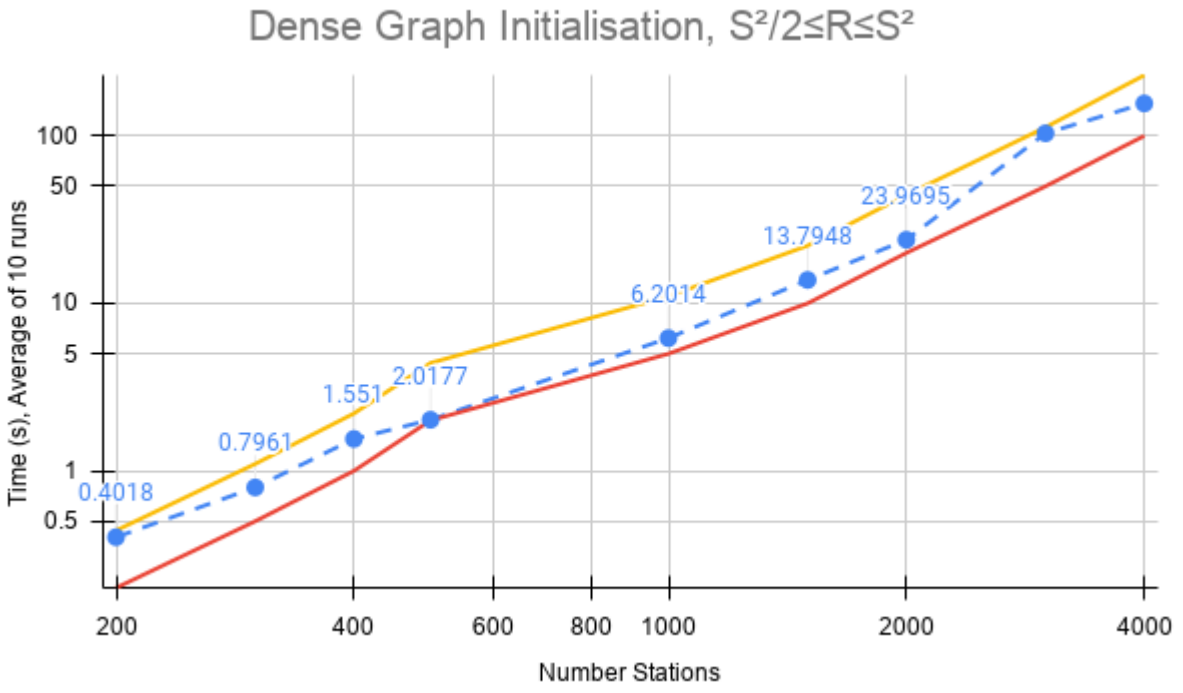
Figure 7 & 8 , Each time result is the average of 10 new railway lines being added. Each of these railway lines consists of randomly chosen stations. Sparse Graph bounded under 2e-5(S^2). Thus our computational complexity grows as expected (Quadratic). Dense Graph could not be run past about 2000 stations due to memory constraints and sparse Graph could not be pushed past 10000 stations due to time constraints.

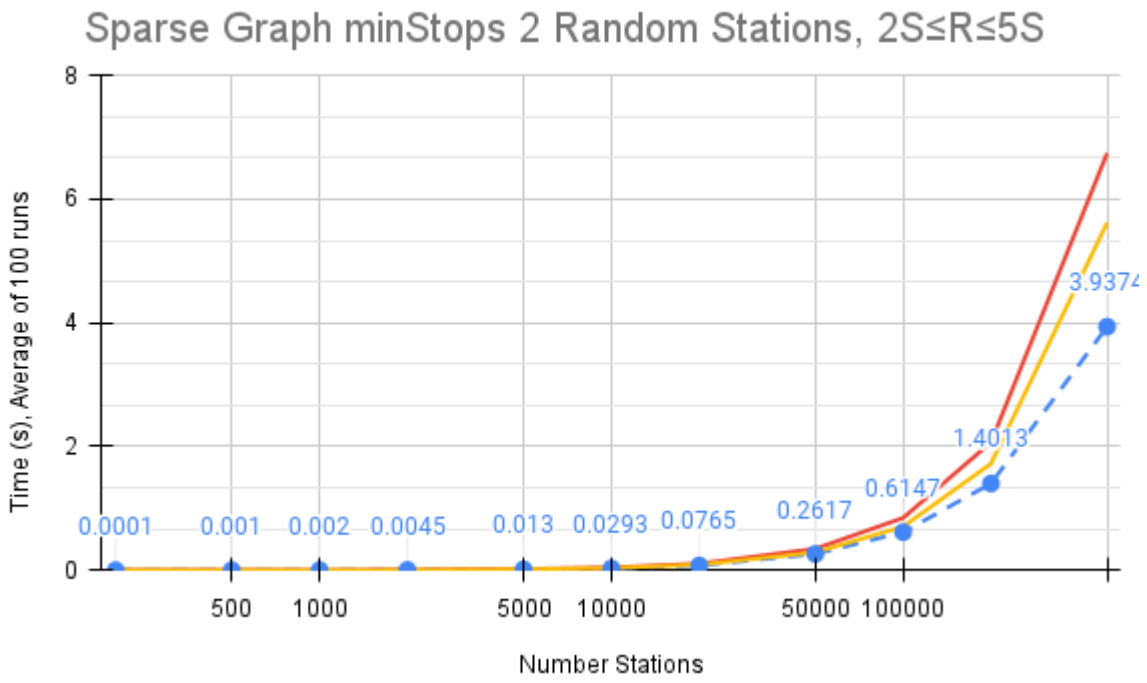
[6] Similar to as we have done above
[7] Thus far we have $O(S_L \log S_L)$

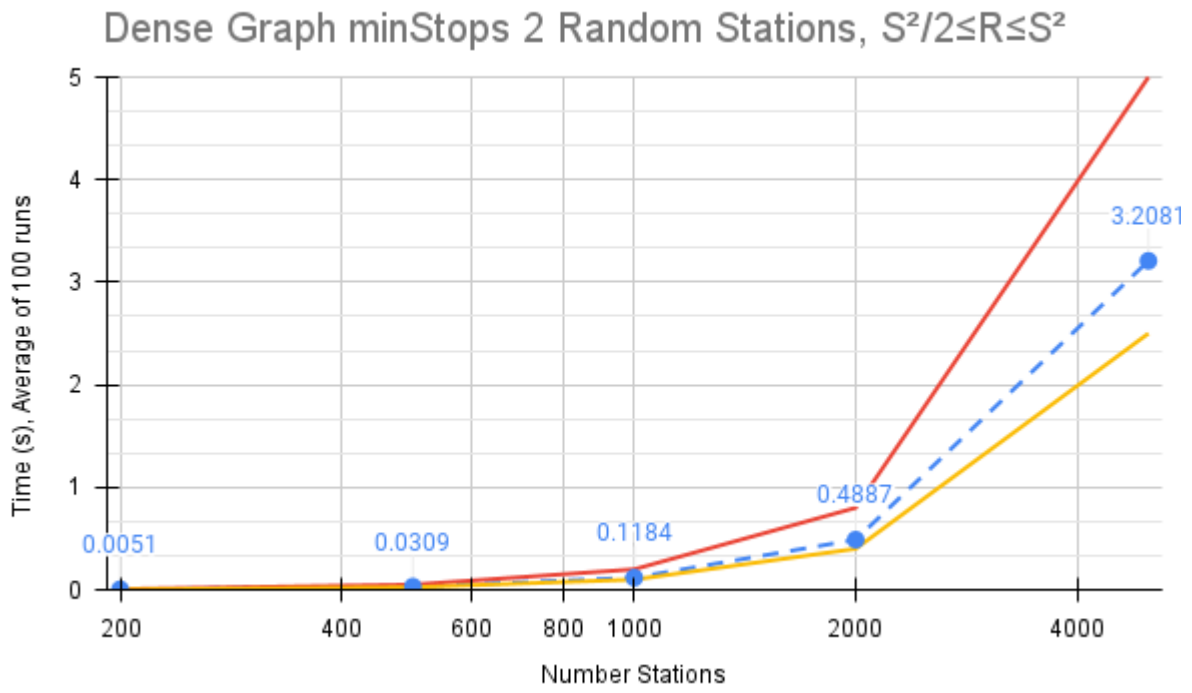
References:

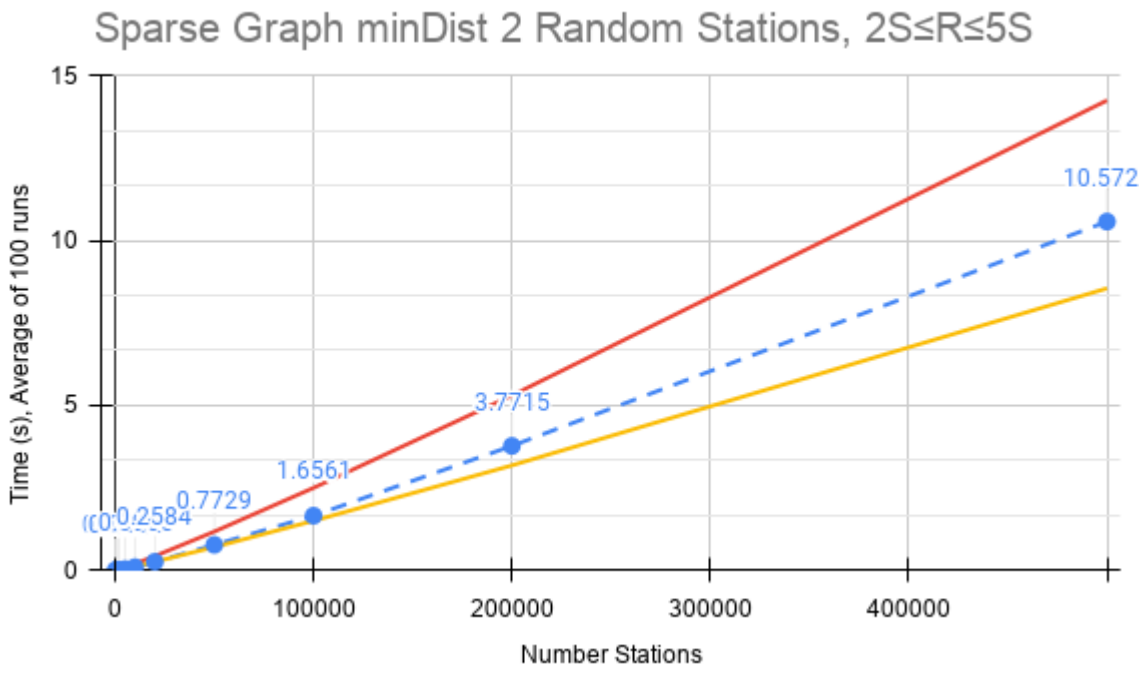
- Dict Complexities.
- Set Complexities.
- List Complexities.
- collections.deque Complexities

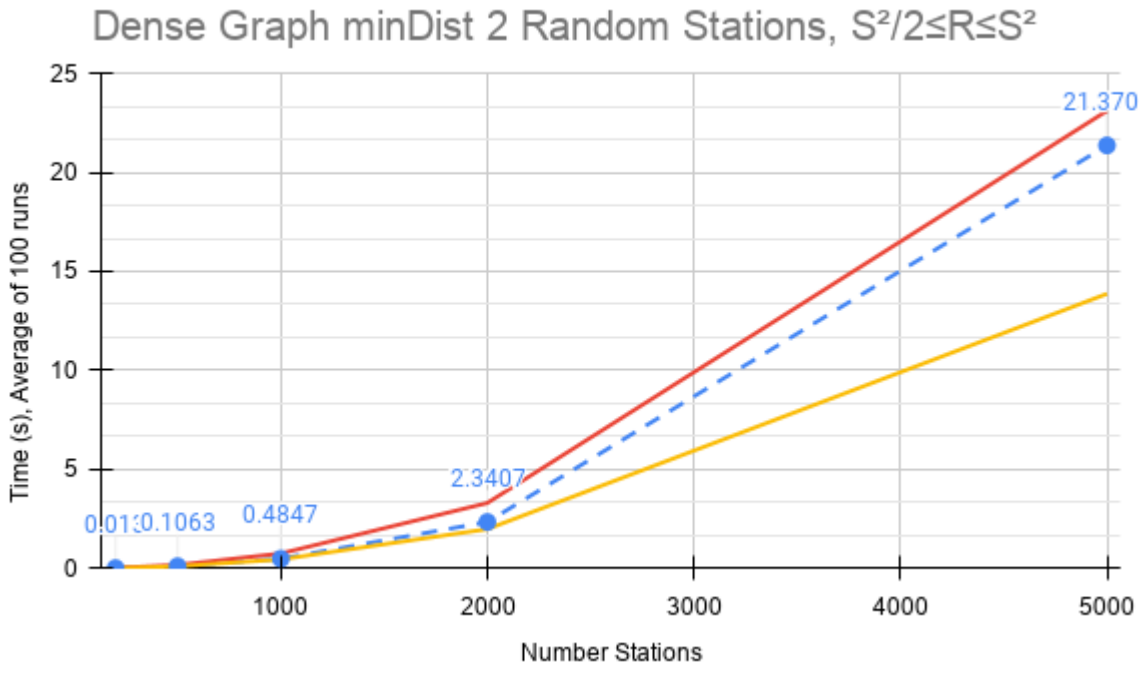


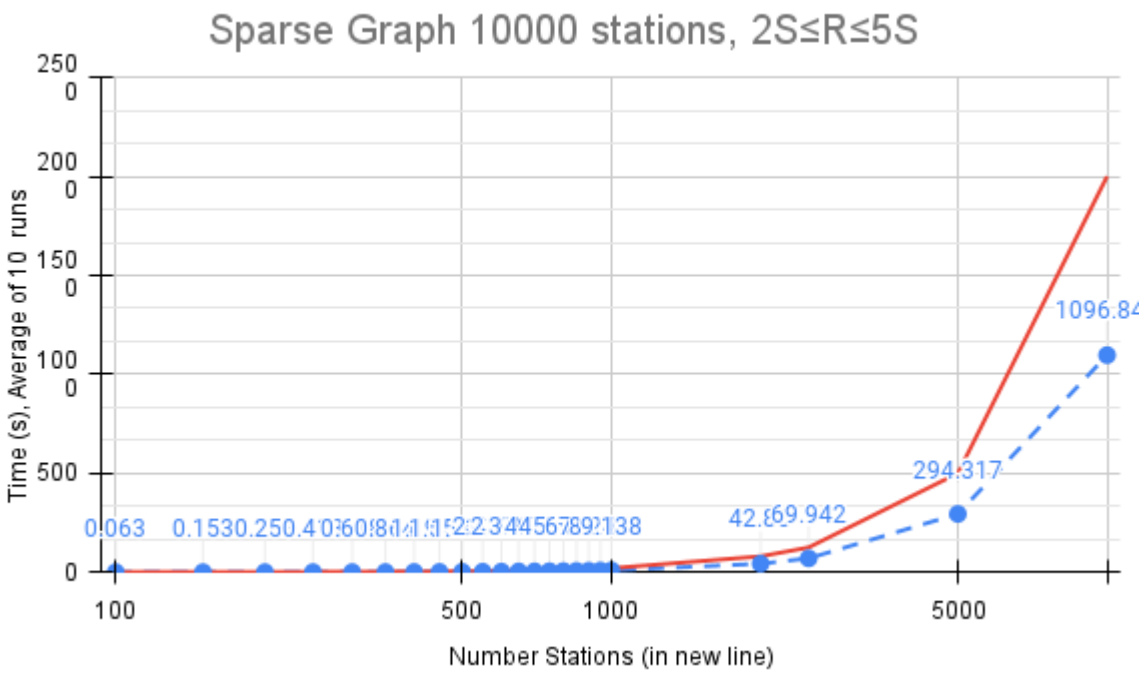


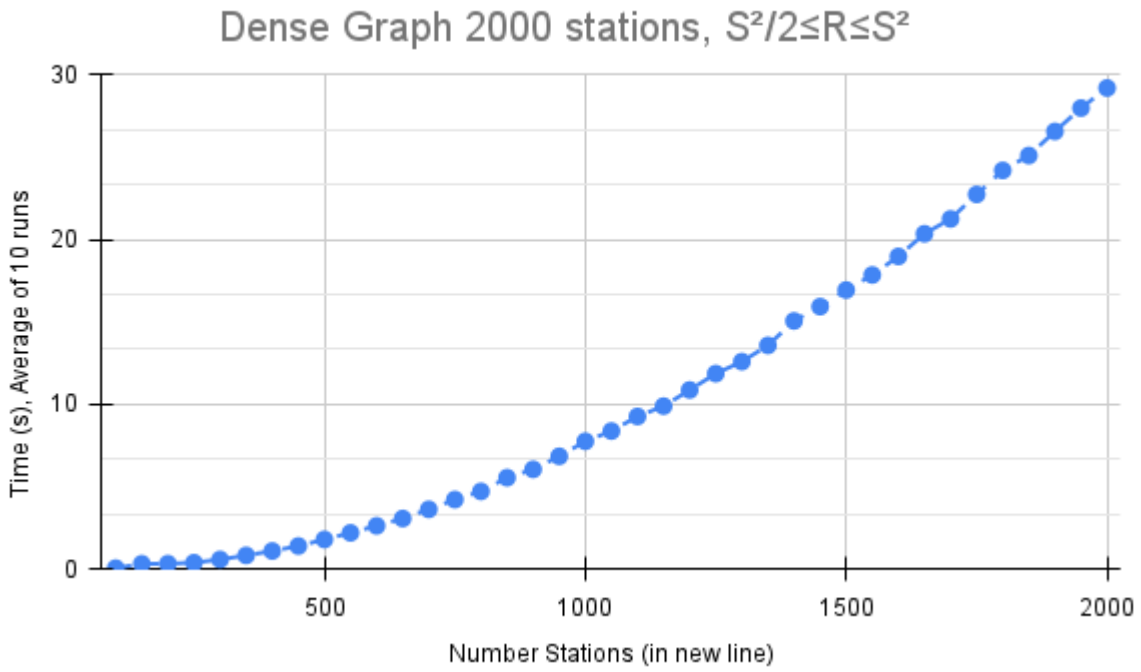

Sparse Graph minStops 2 Random Stations, 2S≤R≤5S
Time (s), Average of 100 runs
Number Stations
0
2
4
6
8
500
1000
5000
10000
50000
100000
0.0001
0.001
0.002
0.0045
0.013
0.0293
0.0765
0.2617
0.6147
1.4013
3.9374


Dense Graph minStops 2 Random Stations, S²/2≤R≤S²
Time (s), Average of 100 runs
Number Stations
0
1
2
3
4
5
200
400
600
800
1000
2000
4000
0.0051
0.0309
0.1184
0.4887
3.2081


Sparse Graph minDist 2 Random Stations, 2S≤R≤5S
Time (s), Average of 100 runs
Number Stations
0
5
10
15
0
100000
200000
300000
400000
0.2584
0.7729
1.6561
3.7715
10.572


Dense Graph minDist 2 Random Stations, S²/2≤R≤S²
Time (s), Average of 100 runs
Number Stations
0
5
10
15
20
25
1000
2000
3000
4000
5000
0.1063
0.4847
2.3407
21.370


Sparse Graph 10000 stations, 2S≤R≤5S
Time (s), Average of 10 runs
250
0
200
0
150
0
100
0
500
0
1096.84
294.317
42.869.942
0.063
0.153
0.25
0.4
0.6
0.8
1
1.5
2
2.3
3
4
4.5
5
6
7
8
9
10
11
38
100
500
1000
5000
Number Stations (in new line)


Dense Graph 2000 stations, S²/2≤R≤S²
Time (s), Average of 10 runs
30
20
10
0
500
1000
1500
2000
Number Stations (in new line)