Submitted by: **Jibin Rajan Varghese** **UIN: 725002873**

List of Contents

1. Introduction

The project aims to give us a feel of graph algorithms for network optimisation by evaluating three well-known algorithms in network optimisation, namely : Modified-Dijkstra's Algorithm, Modified-Dijkstra's Algorithm with Heap, and Kruskal's Algorithm. In order to do the project, we also have to implement a Binary Max-Heap data-structure , which is used by the algorithm. Also we need to generate random graphs which will be used by the project to evaluate the above algorithms.

2. Implementation Details

a. Graph

Data Structure:

The graph is stored as a nested dictionary of vertices in Python.
Here, a vertex say A, is connected to a vertex say B with a weight of W, then we represent this as follows:

{ A : { B : w } }
Where the ({ }) indicates a dictionary which is basically a hashmap in Python
Now, vertex B also has the same information about A, hence the graph (A ↔ B) will be as follows

Graph G = { A : { B : w } , B : { A : w } }

Likewise , we add a third vertex C, connected to A then,

G = { A : { B : w , C: w1 } , B : { A : w }, C : { A : w1} }, ….and so on

Helper Function:

get_vertices : returns a list of all vertices of the graph
get_edges : returns a list of all edges of the graph, given as (u , v , edgeWeight_u-to-v)
add_edges : adds an edge to the graph
add_vertices: adds a vertex to the graph
get_adjacent_vertices : returns the adjacent connected vertices to that vertex
get_connected_edges : returns the connected edges to that vertex
get_edge : returns the edge-weight between two vertices u,v if any, else gives -1
get_degree : returns the number of connected vertices to the particular vertex
grap_plot : plots the graph for debugging

b. Random Graph Generation

Random graph generation is done by the function generate_graph, which takes as inputs number of vertices, max weight of edge and degree (which is basically, number of vertices connected to any vertex v).

The way the algorithm works is as follows:

1. Add vertices upto (n_vertices) to the graph
2. While list of start-end points < 5,
   Generate a random s,t where s ≠ t
   Create a path between s-t by iterating over all vertices and adding edges of random weights s - u - v….. and so on till all vertices are covered
   Append (s,t) to s-t list
3. Once the path is generated, we need to take care that all the vertices are connected upto their average degree of connection to every other vertex. This we do taking the list of every vertex and iterating over all vertices u and v adding random edges as follows
4. For p in range (degree(u), degree) (degree ⇒ degree of connectivity)
   Choose random vertex v from list of vertices
   If degree(v) < degree and u ≠ v , then add edge u-v
   If degree(v) becomes = degree , remove(v) from vertex list

This way we ensure all the vertices are connected to each other up to the extent of their average degrees. The algorithm also ensures that all the vertices of the graph are connected, hence ensuring connectivity.

c. Heap Implementation

We implement the heap as a list of Node elements of the Node base class. The elements of the node class are self._key and self._value, where we use the key of the node to sort them. This is a major limitation of python as python doesn't have arrays and uses lists that are far slower in processing time compared to counterparts in C++ or JAVA. More information here (https://stackoverflow.com/questions/801657/is-python-faster-and-lighter-than-c). Having said that, the fundamental implementation of heap is similar to that of max heap in any language.

Data structure:

The following data structure is used to implement the heap:

Base class : Node
Contains : class (_Element)
Contains :self._key
self._value
Class HeapPQ (inherits Node)
contains : list[Node]

Class DijkstrasHeap (inherits HeapPQ)
Contains : class Indexer(adds to HeapPQ._Element)
Contains : self._index
(which we will use for Modified Dijkstra's with heap, later...)

The standard implementations of _bubbleup, _bubbledown, remove_largest etc are implemented, and lambda functions are used to speed up the searching for parent, left child, right child etc. The Update function in the DijkstrasHeap is worth mention, because it changes the value of the locator by reference, which is how it updates the indexer . Then the values are bubbled up/down by using the _index, of the DijkstrasHeap.

d. Modified - Dijkstra's without Heap

The implementation of the modified dijkstra's function is pretty straightforward. The algorithm scans through the vertices of the graph to get the connected edges, and while the status of a vertex is 'un_visited' it adds it to the fringe, and if the status of the vertex is in 'fringe', it updates the bandwidth if we get a larger bandwidth path to the vertex. The time of finding the fringe vertex of a vertex with max bandwidth is O(n) and iterating over n vertices in the graph would take eventually $O(n^2)$. Thus the runtime complexity of the algorithm is $O(n^2)$ which is the standard case for Dijkstra's Algorithm.

Practically also, we see that this algorithm runs pretty fast as the min, max algorithms on lists are implemented by python's core libraries in O(n) time.

e. Modified - Dijkstra's with Heap

We use the heap defined in section 2.c for the heap implementation. While this is supposed to be faster than using lists in python, we see that in practical case, the nested calls to a large number of member functions such as bubbleup, bubbledown, etc on list of Nodes in python take up a lot of

time, making the constants higher, which eventually slows down the algorithm. The faster way would have been to use pre-implemented heaps and arrays provided by libraries such as numpy which use C++ in their back-end in order to speed up the calculations.

In order to update the maximum bandwidth of a vertex in the heap, I used an update function, and maintained a dictionary of elements in the indexer{} dictionary. Whenever the algorithm updates the the value of the vertex in the heap, we use the indexer dictionary's to get the desired element's location in the heap, change its value and finally perform a bubbleup / bubbledown operation on the heap based on the weight of the element's parent, finally the new element is updated in the indexer array. *Why go through all this process?* Because otherwise, searching for an element in the heap would take O(n) time. For details see : *heap.py, dijkstras_withHeap.py*

Theoretically, the complexity of the algorithm is O((m+n)log(n)) which better than Dijkstra's algorithm. However we can see that in practice, the constants incurred in every step of the internal heap functions which operate on lists slow down the program to give sometimes worse performance than Dijkstra's without heap.

f. Modified - Kruskal's

Kruskal's algorithm is the slowest among the algorithms, because it generates a Maximum Spanning Tree (MST) for the graph over all vertices, and doesn't terminate with the target vertex. It also uses a heap implementation to get the maximum bandwidth element.This is a one-time operation though, and further on the other s-t paths can be easily found by traversing the already generated MST using a breadth first search. We are also required to implement make-set , union, and find algorithms in Kruskal's in order to generate the MST. The union algorithm also does path compression in order to make the subsequent searches faster

The complexity of Kruskal's algorithm is theoretically O(m logn), however the operations on the MST and the time to generate all the vertices and edges again and add it to the MST makes kruskal the slowest algorithm amongst all

g. Other Implementation Details

Runtime of the algorithms are computed using the time module in python. The time.time() command basically takes a snapshot of the system time when the program makes the function call, and we use start-time and end-time in order to compute the time taken for a calculation. Hence for most accurate results, it is advised to run this program on a system dedicated to this task, and not perform parallel operations such as browsing internet etc during the program run, as time-sharing systems could degrade the performance of the program sporadically.

We record the times of the algorithms as follows :

1.Kruskal Runtime : (taking MST generation + one BFS(s-t))
2. Mod-Dijkstras Runtime per problem
3. Mod-Dijkstras Runtime with heap per problem

The *main.py* function also provides 3 ways of running the algorithm : Sample graph type 1 (5000 vertices, avg connectivity = 8) , Sample graph type 2 (5000 vertices, avg connectivity = 1000 (ie 20% of 5000)) and a custom mode, where you can specify the number of vertices, weights and connectivity.

Instructions for running the program:

On any linux machine, open terminal, go to the folder, type python3 main.py
On any windows machine, use IDLE or other similar editors to open the program main.py and run

3. Test Results

Graph No	Start	Destination	Graph Type	Max Bandwidth	Dijkstra's(sec)	Dijkstra's with Heap(sec)	Kruskal's (sec)
	2383	2468	Sparse	831	0.001111269	0.003163099	1.945236444
	3190	1015	Sparse	778	0.706404686	0.934761286	1.89720273
	1850	1868	Sparse	805	0.531928062	0.710656881	1.906707525
	1077	1197	Sparse	793	0.658624411	0.854783297	1.833345175
1	783	4017	Sparse	821	0.217322826	0.103624344	1.776287317
	1996	6	Sparse	753	0.769405842	1.056867599	1.884935379
	3774	4227	Sparse	757	0.676749468	0.867685318	1.577048779
	4286	2454	Sparse	760	0.232151031	0.938663006	1.734577179
	4330	2246	Sparse	672	0.811680794	1.105698586	1.972182512
2	2082	654	Sparse	791	0.683340311	0.935895443	1.856136322
	48	648	Sparse	771	0.000356913	0.814195633	1.925996065
	185	4973	Sparse	762	0.783617973	1.005557775	1.938604116
	1198	706	Sparse	831	0.076955795	0.211340904	1.847801208
	2481	2740	Sparse	684	0.482106924	0.556978941	1.831478834
3	1457	1721	Sparse	758	0.749184132	1.006978512	1.898296118
	113	251	Sparse	766	0.019877434	0.014865875	1.95642662
	744	3949	Sparse	753	0.582321644	0.015926361	1.936281443
	646	3880	Sparse	829	0.089420319	0.189158201	1.890906096
	1520	1592	Sparse	594	0.804329634	1.095822573	1.741241693
4	4054	4875	Sparse	822	0.391150236	0.151984215	1.807379961
	881	4671	Sparse	722	0.788713217	1.054846287	1.885478258
	1969	65	Sparse	758	0.763999701	1.015849352	1.592232943
	2385	1464	Sparse	819	0.0017488	0.006428719	1.957171917
	280	2140	Sparse	822	0.059029102	0.062603712	1.807250738
5	722	2392	Sparse	594	0.812413931	1.086475849	1.795089245

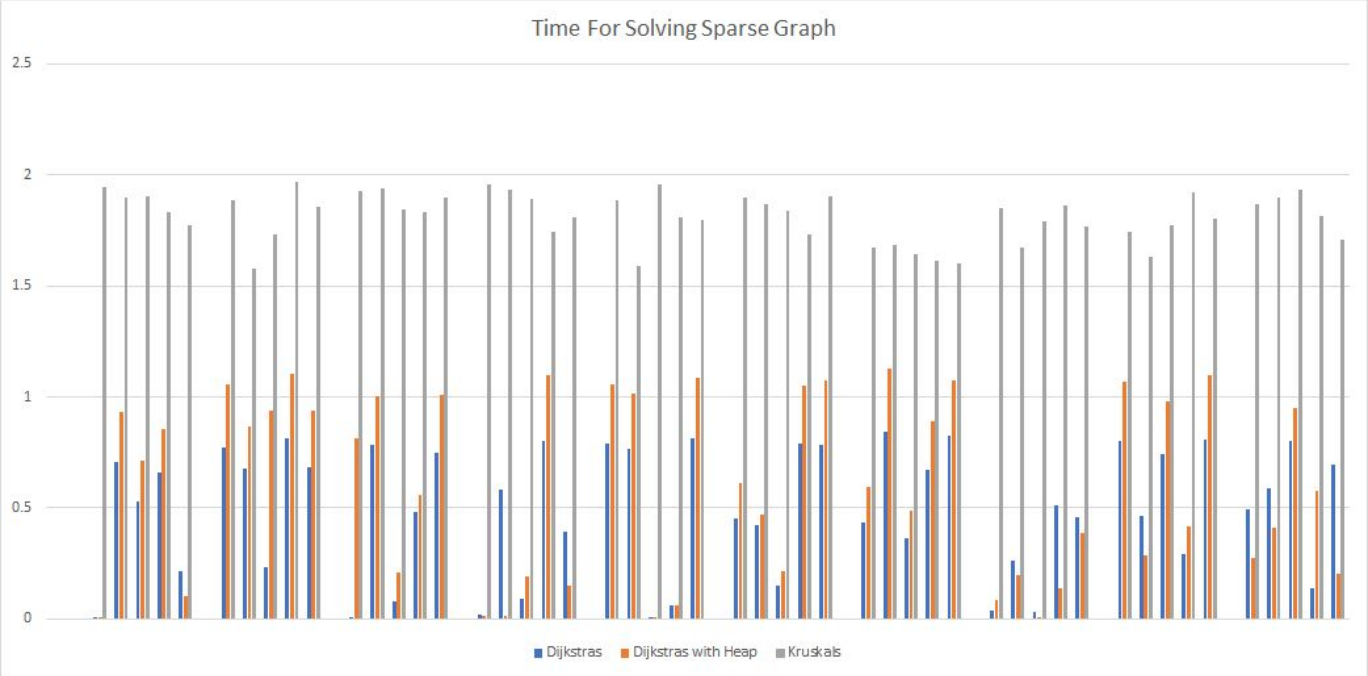
	925	2895	Sparse	827	0.454753637	0.612916231	1.897234917
	3077	113	Sparse	787	0.42238903	0.46762991	1.866211176
	862	1546	Sparse	850	0.149845362	0.217299461	1.841783762
	2196	3344	Sparse	720	0.792511225	1.049772263	1.730278254
6	4089	1468	Sparse	558	0.782558918	1.076106787	1.901629686
	2727	86	Sparse	830	0.432816982	0.59519434	1.6738832
	3791	869	Sparse	629	0.841871023	1.127866745	1.684013128
	1337	4967	Sparse	839	0.362541914	0.487477779	1.64526844
	2493	2081	Sparse	797	0.669561625	0.889633179	1.612102509
7	2774	1162	Sparse	722	0.822855711	1.076582909	1.604333162
	200	4225	Sparse	820	0.038532734	0.084230185	1.850225925
	1923	3057	Sparse	838	0.26393342	0.194748402	1.671947956
	4088	298	Sparse	804	0.032772541	0.01054287	1.792319059
	1590	2829	Sparse	664	0.511556149	0.140086651	1.862214327
8	1814	3867	Sparse	823	0.458301306	0.387879133	1.764925718
	4568	4794	Sparse	716	0.800799131	1.066658497	1.742059708
	2326	1720	Sparse	638	0.463559389	0.283715725	1.633620977
	2640	3855	Sparse	775	0.741197109	0.978826046	1.775003672
	1456	663	Sparse	833	0.295178175	0.414975405	1.921204805
9	3212	880	Sparse	612	0.810302734	1.095571756	1.8028934
	1277	3967	Sparse	784	0.494889736	0.272700071	1.868638515
	187	416	Sparse	799	0.585981846	0.411017179	1.899947166
	2177	4390	Sparse	777	0.79940486	0.952094793	1.932707071
	1620	3048	Sparse	802	0.140858173	0.575659513	1.815918684
10	4152	4779	Sparse	772	0.697349548	0.204534292	1.708341599
Graph No	Start	Destination	Graph Type	Max Bandwidth	Dijkstra's (sec)	Dijkstra's with Heap (sec)	Kruskal's (sec)
	1305	3753	Dense	997	2.433158636	2.654476881	247.3721077
	1308	944	Dense	998	0.020287514	2.456273556	247.4889805
	4952	1189	Dense	997	2.305312395	2.522923946	247.6242757
	851	4573	Dense	995	2.363024712	2.685335636	247.5381434
1	1914	3738	Dense	998	1.710468292	2.393016815	247.734472

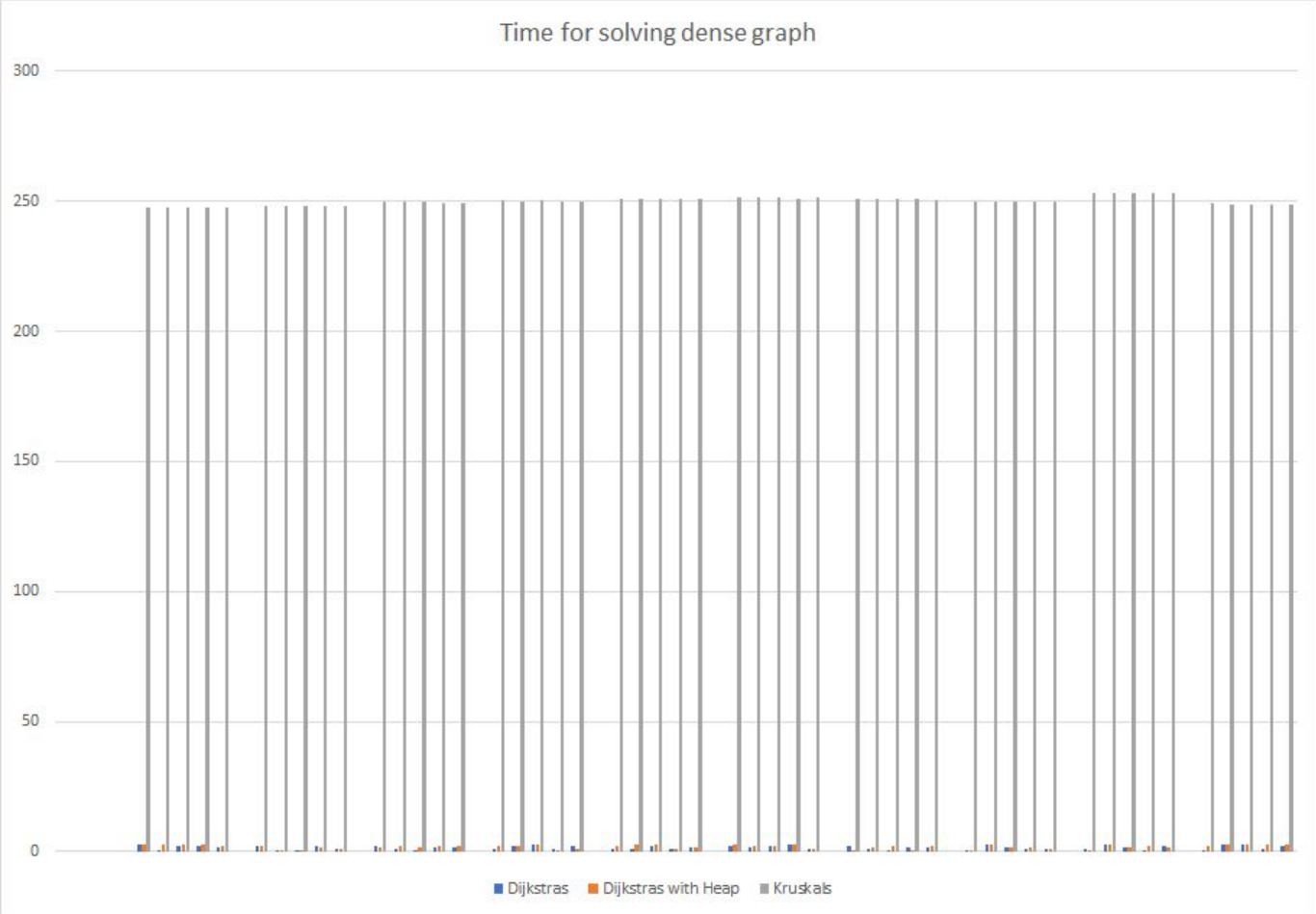
	4681	3336	Dense	995	1.888895035	2.384024858	248.1535666
	3843	2546	Dense	998	0.626591206	0.237205029	248.445405
	2791	457	Dense	998	0.737918377	0.644842148	248.2235031
	3230	4791	Dense	998	2.158873558	1.806940794	248.0479853
2	2957	3085	Dense	998	1.058392525	1.11457634	248.4093935
	2256	3436	Dense	998	1.905411482	1.322086096	249.6208222
	2988	3409	Dense	998	1.236439943	2.036936522	249.6531687
	1977	2256	Dense	998	0.345890522	1.85160017	249.6626818
	2648	3047	Dense	997	1.554548264	2.405081272	249.403939
3	4019	2308	Dense	997	1.548929691	2.36915946	249.3511813
	4766	3152	Dense	998	1.00080061	2.047325134	250.3657694
	1935	4404	Dense	998	2.075209141	2.279929876	250.0373313
	1968	4409	Dense	996	2.615745783	2.790989161	250.2518809
	4161	3204	Dense	998	1.046255589	0.243010283	250.0046663
4	1682	4335	Dense	998	2.089597225	0.923006535	250.1136727
	4313	2438	Dense	998	1.088866472	2.159248352	251.056505
	1981	1505	Dense	998	1.037086248	2.495435953	250.9116666
	1242	3021	Dense	998	2.063750029	2.594474077	250.9328415
	3622	1146	Dense	998	1.004027128	1.167860508	250.738564
5	3917	2487	Dense	996	1.378180027	1.651025057	251.0050478
	4533	193	Dense	997	2.316796303	2.734642744	251.2870693
	3593	3370	Dense	997	1.768087626	2.412296295	251.2820661
	4561	4893	Dense	997	2.386524677	1.866137266	251.3473327
	2000	4477	Dense	997	2.533018351	2.692176819	251.1349189
6	2075	3435	Dense	998	1.25652957	1.001054525	251.311902
	2166	4604	Dense	997	2.296496868	0.52844429	251.0428638
	4988	1939	Dense	998	0.781417131	1.325712204	251.047796
	4289	48	Dense	998	0.14581728	2.022686243	251.0181303
	2384	1694	Dense	998	1.550988197	0.526453018	250.8474679
7	4904	3138	Dense	998	1.641355515	2.128684759	250.6972027
	1997	682	Dense	997	0.378904819	0.247051716	249.6077824
8	2630	4419	Dense	997	2.535284519	2.541092634	249.7413275

	2023	1790	Dense	998	1.357339382	1.815079451	249.9595165
	233	1418	Dense	997	1.23368597	1.606793165	249.5950737
	759	3374	Dense	998	1.14089489	1.163100004	249.9545255
	2021	2577	Dense	997	1.136331081	0.417392731	253.3966601
	2838	669	Dense	993	2.612388134	2.831917524	253.1975672
	4112	2985	Dense	997	1.481634855	1.454424858	253.3953328
	2236	2013	Dense	998	0.504899502	1.996191978	253.3726201
9	486	1965	Dense	997	1.985363007	1.335858822	253.1756938
	12	3129	Dense	998	0.736916304	1.978775263	249.1601906
	2977	2266	Dense	996	2.588026762	2.776119947	249.0244322
	931	2478	Dense	992	2.590489864	2.807924747	248.750587
	341	2201	Dense	998	0.852081776	2.476042986	248.8893886
10	2738	3611	Dense	997	2.156574965	2.46205616	248.9511943

4. Analysis of Results

'The results are easier to analyse if plotted in a graph:





The results are as predicted when we implemented the problem. The performance of the Dijkstra's algorithm (with or without heap) is much faster than kruskal's for sparse or dense graphs. In fact the results are much more noticeable in dense graphs, because of the longer time taken to create the MST in kruskal's. For sparse graphs, the creation of heaps are an overkill as python lists are more efficient than a custom implementation (Heaps) that use lists internally. Hence we can see that the runtime of Dijkstra's with heap is slightly worse than dijkstra's without heap in sparse graphs. For dense graphs we observe that Dijkstra's with heap performs slightly better than without heap, but the difference is negligible because of the large constants incurred in the heap generation in python. However this is in alignment with the theoretical understanding of the runtime complexities of the algorithm.

As we can see in comparison, Kruskal's algorithm takes the most time for sparse and dense graphs. The main reason here is the overhead cost of constructing the Maximum Spanning tree for the entire graph. However, once the MST is constructed, the BFS can be done repeatedly for multiple s - t vertices using the same MST, unlike Dijkstra's Algorithm which has to do the whole computation again and again.

Lastly, the time complexity of all these algorithms are in the order of O(m log n), and this can be seen from the fact that dense graphs take more time than sparse graphs.

Further Improvements

There are many improvements that can be done to speed up the network optimisation problem:

1. Choice of Programming Language :
   The speed of basic data structures depends heavily on the level of abstraction of the programming language. Python being a very high level programming language (4th generation language) , is often up to 100 times slower than C/C++. Hence the algorithm would have been much faster if implemented in C / C++ or JAVA for that matter.

2. Use of Dictionaries instead of lists in Heap:
   The heap implementation uses python lists which are slower compared to dictionaries (which use hash tables in the backend). While I was able to convert the graph into a set of nested dictionaries, I wasn't successful in doing so for heaps, because managing the indices became challenging. The next immediate improvement would be the conversion of heaps into a data-structure that uses dictionaries in the back end.

3. Improving algorithms using parallel processing and multi-threading architecture.
   By use of simple changes to the program, Kruskal's algorithm can be made faster if make-set union find is run parallelly instead of sequentially. This is also true in the case of graph generation, where iterating over vertices and adding edges can be done parallelly as well

4. Other Improvements:
   We can also improve our algorithm further by using many other optimisations. For example, we can improve the runtimes of the Kruskal's algorithm by adding edges to a list and implementing Quicksort or python's native sort algorithm rather than implementing heaps. We can use numpy arrays which (are basically C++ back-end) in place of the lists in our program. We can also use 2-3 trees and similar data-structures to store the fringes for example.