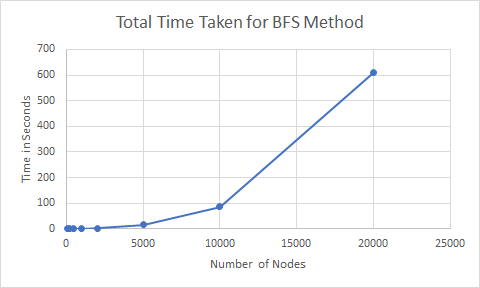
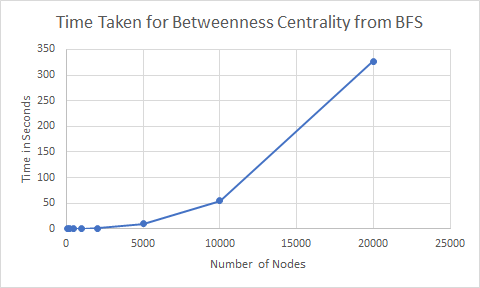
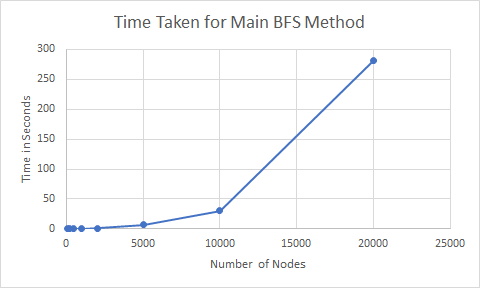
**Execution Time of BFS**

The BFS method takes **1 minutes 29 seconds**on a graph of 10k nodes. Further it takes, only **10  minutes 6 seconds** on a graph of 20k nodes

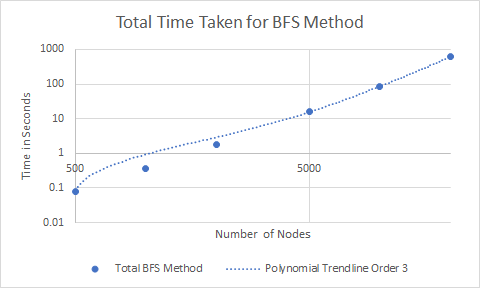
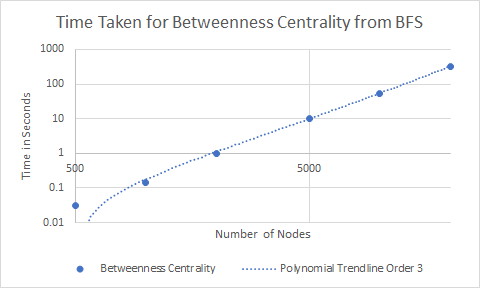
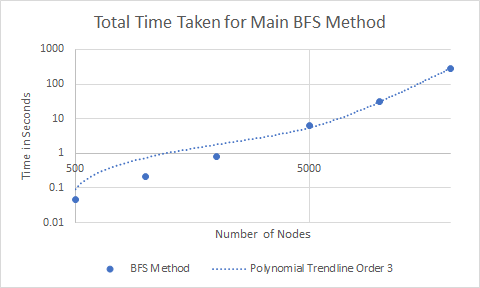
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Plots for BFS**

I am attaching below a graph of the execution time for BFS  
  


--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

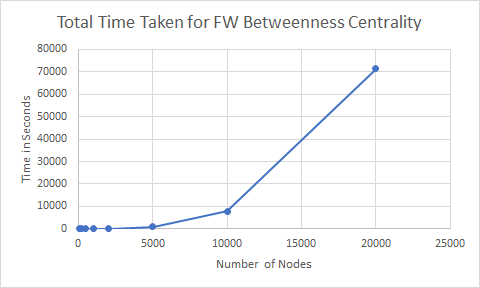
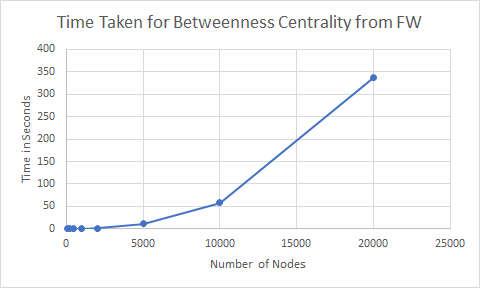
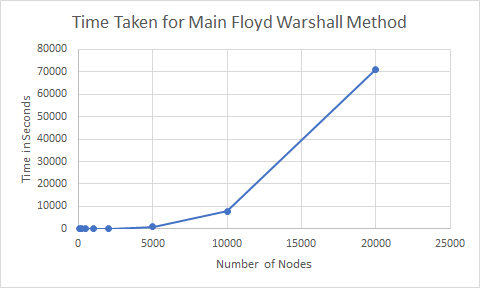
**Log-Log Plots**

I am attaching below a log-log plot of the execution time for BFS for further clarity  


--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Plots for FW**

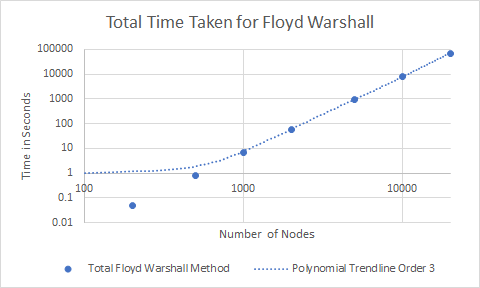
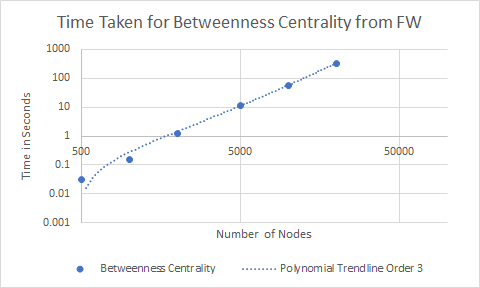
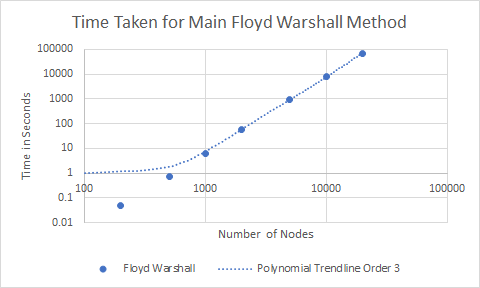
The below charts show the relationship between the time taken for the FW method execution vs the number of nodes in the graph.



--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Log-Log Plots**

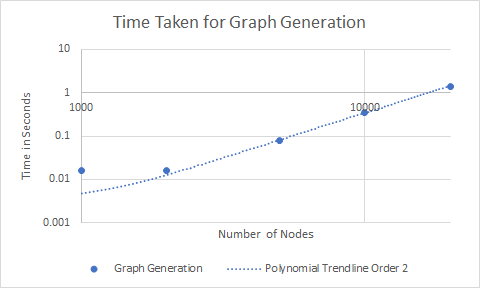
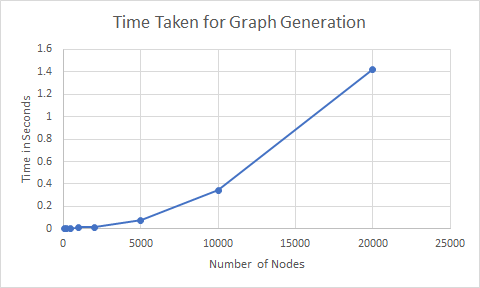
We plot the log log plots based on the above data so that we are not overwhelmed by the highest point in the graph.



--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Graph Generation**

I have used the Erdos method to generate the graphs. Since in my original implementation, I didn't understand that we only have to implement unweighted graphs, I have created methods for weighted graph creation as well. Currently, the Graph generation takes order n^2 time.



--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Table of Execution Times**

For the purpose of easy viewing, I am attaching the actual tabular values for all the execution times below

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Number of Nodes | Graph Generation | Floyd Warshall | Betweenness Centrality | Total Floyd Warshall Method | BFS Method | Betweenness Centrality | Total BFS Method |
| 100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 200 | 0 | 0.046875 | 0 | 0.046875 | 0.015625 | 0 | 0.015625 |
| 500 | 0 | 0.75 | 0.03125 | 0.78125 | 0.046875 | 0.03125 | 0.078125 |
| 1000 | 0.015625 | 6.484375 | 0.15625 | 6.65625 | 0.21875 | 0.140625 | 0.375 |
| 2000 | 0.015625 | 58.96875 | 1.203125 | 60.1875 | 0.828125 | 0.984375 | 1.828125 |
| 5000 | 0.078125 | 947.546875 | 11.40625 | 959.03125 | 6.484375 | 10.109375 | 16.671875 |
| 10000 | 0.34375 | 7783.0625 | 57.875 | 7841.28125 | 30.546875 | 54.475 | 85.365625 |
| 20000 | 1.421875 | 71125.4975 | 337.15975 | 71464.07913 | 280.90625 | 326.5 | 608.828125 |

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**File Structure:**

For the purposes of modularity, I have created implementation files as below. Each comes with (1) its own header (with nested headers if required), (2) its own test client (3) it's own object file and linkages (4) Test Results. I am mentioning the structure with a brief description below

1. graph\_functions
   1. graph\_functions.c: This is the implementation for all things to do with the graph, stored as an adjacency list.
   2. graph\_functions.o: This is the object file
   3. test\_graph\_functions.c: This is the client file that tests the graph functions
   4. test\_graph\_functions.out: This is the executable that can be run to test the graph functions
   5. test\_graph\_functions.txt: This is used to test output
   6. GraphFunctions.h: This is the header to be included for clients of graph\_functions
2. build\_graph
   1. build\_graph.c: Used for building the graph
   2. build\_graph.o: This is the object file
   3. test\_build\_graph.c: We test some basic graph building here including from the test file given
   4. test\_build\_graph.out: This is the executable that can be run to test the required outputs are met
   5. test\_build\_graph.txt: This is the output file for testing
3. nodesArrayFunctions
   1. nodesArrayFunctions.c: I have used this to test the processed graphs. I store data structures like nodeArrays (storing only half of matrix) and pathmatrix here.
   2. nodesArrayFunctions.o: This is the object file for above
   3. manual\_tests.xlsx: This is a manual set of tests done to ensure correctness.
   4. ProcessedGraph.h: This is the header used to access the data structures used to process Graphs by both the methods.
4. floyd\_warshall
   1. floyd\_warshall.c: This is a small file implementing the Floyd Warshall Method
   2. floyd\_warshall.o: This is the object file
   3. test\_floyd\_warshall.c: This is the client file used to test the Floyd Warshall Method
   4. test\_floyd\_warshall.out: This is the executable that can be run to test the graph functions
   5. test\_floyd\_warshall.txt: This is used to test output on the sample graph
   6. FloydWarshall.h: This is the specific header used for clients of this file.
5. bfs
   1. bfs.c: Main source file used to process graph through BFS method.
   2. bfs.o: This is the object file
   3. test\_bfs.c: This is the client file used for running the tests. Here we compare results with the FW method and they match
   4. test\_bfs.out: This is the executable used for tests
   5. test\_bfs.txt: This is the text file with outputs of the path matrix and the betweenness centrality values
   6. BFS.h: Header used
6. random\_connected\_graph\_generator
   1. random\_connected\_graph\_generator.c: This is the implementation of random graph generator
   2. random\_connected\_graph\_generator.o: This is the object file
   3. test\_random\_connected\_graph\_generator.c: This is the client file that tests the random graph generator
   4. test\_random\_connected\_graph\_generator.out: This is the executable that can be run the tests
   5. test\_random\_connected\_graph\_generator.txt: The outputs for the test are stored here
   6. RandomGraph.h: Header used by clients of the random connected graph generator.
   7. test\_large\_graphs.c: another set of tests for generation of large graphs.
   8. test\_large\_graphs.out: The executable for above.
7. betweenness\_centrality
   1. betweenness\_centrality.c: The source file used to implement betweenness centrality values. Note the tests for this are run within the FW test, BFS and the large graph tests
   2. betweenness\_centrality.o: This is the object file
   3. BetweennessCentrality.h: header file to run Betweenness Centrality in any client.

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Thorough Testing**

1) I have created a thorough test Suite giving all the required functionality. I am attaching the files and the outputs in the attached sharepoint drive. (note that even after zipping test outputs are too large to attach on email). I have divided these into appropriate folders for ease of access. Note that some of the executables would need the sample graphs in the '/data' folder. i have accordingly placed the data folder within the executables directory

[[](https://indianinstituteofscience-my.sharepoint.com/:f:/g/personal/mrahul_iisc_ac_in/EpeEds_1rrBJuVqIUcVEFuABWyhZ6xGHCuOeY3t0UIwqQw?e=K4BW4v)TestSuite](https://indianinstituteofscience-my.sharepoint.com/:f:/g/personal/mrahul_iisc_ac_in/EpeEds_1rrBJuVqIUcVEFuABWyhZ6xGHCuOeY3t0UIwqQw?e=K4BW4v)

2) I would specifically draw your attention to the files in the below folder.

The file **test\_large\_graphs.txt**may be of your interest for the large graphs case.

[[](https://indianinstituteofscience-my.sharepoint.com/:f:/g/personal/mrahul_iisc_ac_in/Eox4D3KR5y5CiLlP64FPcv8B5buhqDh9kk1cJfU8HdBRWw?e=1net95)testOutputs](https://indianinstituteofscience-my.sharepoint.com/:f:/g/personal/mrahul_iisc_ac_in/Eox4D3KR5y5CiLlP64FPcv8B5buhqDh9kk1cJfU8HdBRWw?e=1net95)

3) Lastly, I would like to draw your attention to **manual tests**that I have done to ensure correctness of the betweenness centrality values. This has been matched with a sample graph I created

<https://graphonline.ru/en/?graph=xCpmWyGaJBxdpNbi>

[Manual Test File](https://indianinstituteofscience-my.sharepoint.com/:x:/g/personal/mrahul_iisc_ac_in/EQwc_fbIeeVPi1ixSpVPyjQBYRZ9Y6x3J5gBkDzoyglN4A?e=iHCMUj)

Main test files as already described:

1. test\_build\_graph
2. test\_graph\_functions
3. test\_floyd\_warshall
4. test\_BFS
5. test\_random\_connected\_graph\_generator
6. test\_large\_graph\_generation
7. test\_large\_graphs

The last one is the most important one of all, as it runs all tests together. Further, we have main.c (which generates BetweennessCentrality.out, which includes all tests for the sample file, but does not generate large graphs as they take long to run

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Efficiency: 0 memory Leaks.**

Since the memory requirements for the program were relatively high, it called for 0 memory leaks. I have attached this in the screenshots below

[[](https://indianinstituteofscience-my.sharepoint.com/:f:/g/personal/mrahul_iisc_ac_in/Epq8zfEytXJBn_wnm-ZgfRwBAORrQkGu2B2EnN8mxgXmFw?e=eFShsV)valgrind](https://indianinstituteofscience-my.sharepoint.com/:f:/g/personal/mrahul_iisc_ac_in/Epq8zfEytXJBn_wnm-ZgfRwBAORrQkGu2B2EnN8mxgXmFw?e=eFShsV)

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**FW Pseudo code**

We will briefly describe the method of implementation below of the Floyd Warshall method to compute Betweenness Centrality. Adequate description of the data structures involved has not been included for brevity

**>** Take two passes of n^3.

> In the first pass, 

> loop in the order Intermedate Node, Source Node, Destination Node

> Within each loop,

> If (Source-Destination) distance <  (Source-Intermediate) distance + (Intermediate-Destination) distance

1. Compute new Shortest Path Distance

2. Compute new Num Shortest Paths (Source-Destination) = Numshortest Paths (Source-Intermediate) \* Numshortest Paths (Intermediate-Destination)

> If (Source-Destination) distance == (Source-Intermediate) distance + (Intermediate-Destination) distance

1. Add to Numshortest Paths (Source-Destination) += Numshortest Paths (Source-Intermediate) \* Numshortest Paths (Intermediate-Destination)

> Initialize Betweenness Centrality array for each node to 0.

> In the second pass,

> loop in the order Intermedate Node, Source Node, Destination Node

> Within each loop,

> If (Source-Destination) distance ==  (Source-Intermediate) distance + (Intermediate-Destination) distance

> Compute Betweenness Centrality (Intermediate Node) +=

(NumshortestPaths (Source-Intermediate) \* NumshortestPaths (Intermediate-Destination))/NumshortestPaths (Source-Destination)

> return Betweenness Array

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**BFS Pseudo code**

We will briefly describe the method of implementation below of the BFS method to compute Betweenness Centrality.

**>** Take two passes.

> In the first pass,

> loop for each intermediate node

> Within each loop,

> Carry out a bfs

> if node not visited, mark visited

> mark predecessor node

> mark shortest path from source vertex to vertex visited

> if node visited, then check if shortest path same

> if same, add to number of shortest paths

> add to predecessor nodes list (subgraph)

> Initialize Betweenness Centrality array for each node to 0.

> In the second pass,

> loop in the order Source Node, Destination Node

> Within each loop,

> Visit the predecessor nodes in DFS order based on predecessor subgraph stored.

> Compute Betweenness Centrality (Predecessor node) +=

(NumshortestPaths (Source-Predecessor) \* NumshortestPaths (Predecessor-Destination))/NumshortestPaths (Source-Destination)

> return Betweenness Array

**BFS is therefore much faster as computation of predecessor subgraph is very fast.** Note that this is more memory intensive than the standard implementation. If the lower memory requirement implementation is required, it has been saved in the folder below

**--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------  
  
Alternative Implementation**

For all cases where memory is an issue or the graphs are not sparse, we provide an alternative implementation (submission) as below. This is one of the many iterations on the code that was carried out. Amongst the alternatives created, the main submission as well as the below submission are optimized for different circumstances. For example the below is optimized for minimized memory usage. Its running time is very similar for Floyd Warshall as our given submission.

[[](https://indianinstituteofscience-my.sharepoint.com/:f:/g/personal/mrahul_iisc_ac_in/ElgaZoeLdV5DopM04lLHwSoBuIkDETiRL5C3p7MPBiiNrQ?e=MKxnbc)BetweennessCentralityLowmem](https://indianinstituteofscience-my.sharepoint.com/:f:/g/personal/mrahul_iisc_ac_in/ElgaZoeLdV5DopM04lLHwSoBuIkDETiRL5C3p7MPBiiNrQ?e=MKxnbc)

**Possible improvement through Reachability Graph**

As per the class today on spatial locality in malloc, I said to you in class that such spatial locality may enable me to improve my submission. I have to unfortunately report that the Reachability Matrix method (with my implementation of spatial locality) produces no better results than just the pathmatrix alone. The possible explanation I gave myself is that the constants for random accesses into matrices is much lower in case one accesses local elements rather than far away elements.

In other words, *order of access into matrices matters more than a (small) reduction in number of accesses*.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | Without Pathmatrix, Without Reachability Matrix | | | With Pathmatrix, Without Reachability Matrix | | | Without Pathmatrix, With Reachability Matrix | | | With Pathmatrix, With Reachability Matrix | | | |
| Number of Nodes | Build Graph | Process Graph | Betweenness Centrality | Total Time Taken | Process Graph | Betweenness Centrality | Total Time Taken | Process Graph | Betweenness Centrality | Total Time Taken | Process Graph | Betweenness Centrality | Total Time Taken |
| 100 | 0.00 | 0.00 | 0.00 | 0.00 | 0.02 | 0.00 | 0.02 | 0.00 | 0.00 | 0.00 | 0.02 | 0.00 | 0.02 |
| 200 | 0.00 | 0.03 | 0.03 | 0.06 | 0.05 | 0.02 | 0.06 | 0.03 | 0.05 | 0.08 | 0.05 | 0.00 | 0.05 |
| 300 | 0.02 | 0.11 | 0.13 | 0.25 | 0.19 | 0.00 | 0.20 | 0.20 | 0.13 | 0.34 | 0.19 | 0.02 | 0.22 |
| 400 | 0.00 | 0.31 | 0.30 | 0.61 | 0.41 | 0.02 | 0.42 | 0.31 | 0.30 | 0.61 | 0.42 | 0.02 | 0.44 |
| 500 | 0.00 | 0.63 | 0.64 | 1.27 | 0.83 | 0.03 | 0.86 | 0.66 | 0.58 | 1.23 | 0.88 | 0.05 | 0.92 |
| 600 | 0.00 | 1.13 | 1.13 | 2.25 | 1.45 | 0.05 | 1.50 | 1.27 | 1.14 | 2.41 | 1.63 | 0.06 | 1.69 |
| 700 | 0.00 | 1.89 | 1.89 | 3.78 | 2.44 | 0.08 | 2.52 | 2.06 | 2.02 | 4.08 | 2.80 | 0.11 | 2.91 |
| 800 | 0.02 | 2.83 | 3.08 | 5.92 | 3.63 | 0.13 | 3.77 | 3.45 | 3.09 | 6.56 | 4.73 | 0.14 | 4.89 |
| 900 | 0.00 | 4.17 | 4.52 | 8.69 | 5.23 | 0.17 | 5.41 | 5.78 | 4.53 | 10.31 | 7.39 | 0.20 | 7.59 |
| 1000 | 0.00 | 5.84 | 6.36 | 12.20 | 7.25 | 0.23 | 7.48 | 8.28 | 6.52 | 14.80 | 10.09 | 0.27 | 10.36 |

**PathMatrix explanation**

> During the calculation of number of shortest paths(first k,i,j loop),

> we can store the predecessor subgraph. This can be done by just storing the last hop node in a path.

> In case of multiple paths, this may be more than 1 node

> During the calculation of betweenness centrality, we don't need to do the triple loop again, we can just go through the predecessor subgraph to access nodes on the shortest paths (Note that we are nowhere storing shortest path, just the last node)

**Pros:**

1. Drastic reduction in running time for Betweenness Centrality for sparse graphs

**Cons:**

1. Higher memory usage.
2. Not worthwhile for non-sparse graphs

**Reachability Matrix explanation**

> The typical floyd warshall triple loop consists of

i)   intermediate node -(k)

ii)  source node -(i)

iii) destination node -(j)

But we can note that :

1. For every intermediate node (k), especially in the starting stages (low k values) of sparse graphs, we may not be able to reach all other vertices.
2. Thus looping over every i and j may not be worthwhile
3. We may save a lot of calculations by storing the vertices that can be reached from each k.
4. Then our loop runs as follows:

1. For each intermediate node(k):

2. > For *i\_index*in 0:num\_reachable\_nodes(k):

3. i = reachables[k][i\_index];

4. > For *j\_index*in i\_index+1:num\_reachable\_nodes(k):

5. j = reachables[k][j\_index];

6. Do regular loop tasks

I was earlier storing the list of vertices that can be reached from a given intermediate node by a linked list/adjacency list (in the submission given to you). I realized that this would be extremely slow due to malloc allocating memory non-contiguously based on the class today. I therefore switched to a matrix where I was allocating nodes as they are seen. This would be in random order, but the blocks would be contiguous. Further this is a definite reduction in number of loops made over the naive implementation.

Yet the naive implementation runs faster. This could be because

a) the graphs get connected very fast, ie in just a few passes over the intermediate nodes, the entire graph becomes connected and you can reach anywhere from anywhere (then the method gives no added benefits)

b) the nodes are added in random order, and therefore our access into the main structure (nodesarray is what i call it) where we store the shortest paths etc becomes random. This causes the code to lose locality in access there.

Thus even though the method has fewer passes through the central part of the loop, it is slower.

**Possible Improvements**

If the number of nodes reachable from some intermediate node is under some percentage of nodes (or some heuristically decided %), then go with this method, else go back to triple loop.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Heuristic | Process Graph | Betweenness Centrality | Time Taken to Build Graph | Total Time Taken | Time Taken with Pathmatrix Without Reachability | %age Improvement |
| 0.05 | 7.015625 | 0.25 | 0.00 | 7.27 | 7.40625 | 1.94% |
| 0.10 | 7.015625 | 0.23 | 0.00 | 7.25 | 7.40625 | 2.16% |
| 0.15 | 7.000000 | 0.25 | 0.00 | 7.25 | 7.40625 | 2.16% |
| 0.20 | 7.015625 | 0.20 | 0.00 | 7.22 | 7.40625 | 2.60% |
| 0.25 | 7.046875 | 0.23 | 0.00 | 7.28 | 7.40625 | 1.72% |
| 0.30 | 7.015625 | 0.23 | 0.00 | 7.25 | 7.40625 | 2.16% |
| 0.35 | 7.046875 | 0.23 | 0.00 | 7.28 | 7.40625 | 1.72% |
| 0.40 | 7.015625 | 0.23 | 0.00 | 7.25 | 7.40625 | 2.16% |
| 0.45 | 7.015625 | 0.23 | 0.00 | 7.25 | 7.40625 | 2.16% |
| 0.50 | 7.046875 | 0.20 | 0.00 | 7.25 | 7.40625 | 2.16% |
| 0.55 | 7.000000 | 0.23 | 0.00 | 7.23 | 7.40625 | 2.38% |
| 0.60 | 7.015625 | 0.23 | 0.00 | 7.25 | 7.40625 | 2.16% |
| 0.65 | 6.984375 | 0.23 | 0.00 | 7.22 | 7.40625 | 2.60% |
| 0.70 | 7.015625 | 0.23 | 0.00 | 7.25 | 7.40625 | 2.16% |
| 0.75 | 7.031250 | 0.22 | 0.00 | 7.25 | 7.40625 | 2.16% |
| 0.80 | 7.062500 | 0.23 | 0.00 | 7.30 | 7.40625 | 1.50% |
| 0.85 | 7.015625 | 0.23 | 0.00 | 7.25 | 7.40625 | 2.16% |
| 0.90 | 7.109375 | 0.25 | 0.00 | 7.36 | 7.40625 | 0.64% |
| 0.95 | 7.031250 | 0.23 | 0.00 | 7.27 | 7.40625 | 1.94% |

In this case, we see an improvement over the pathmatrix without reachability, but it is nothing to write home about