Comparative Study of Various Approaches of Dijkstra Algorithm

1st Divyam Verma
dept. of Informatics
(School of Computer Science)
University of Petroleum and Energy Studies
Dehradun, Uttarakhand, India
divyam0216@gmail.com

2nd Devansh Messon
dept. of Informatics
(School of Computer Science)
University of Petroleum and Energy Studies
Dehradun, Uttarakhand, India
devansh.messon28@gmail.com

3rd Mayank rastogi
dept. of Informatics
(School of Computer Science)
University of Petroleum and Energy Studies
Dehradun, Uttarakhand, India
Mayikmj@gmail.com

4th Amit Singh
dept. of Informatics
(School of Computer Science)
University of Petroleum and Energy Studies
Dehradun, Uttarakhand, India
amit.singh1@ddn.upes.ac.in

***Abstract*—Dijkstra algorithm is not improved in terms of space and time-complexity, the algorithm is muddled to apply in network analysis for enormous test cases. To handle this huge time complexity, advanced structures are applied to enhance Dijkstra algorithm. The use of a keen and numerical procedure speeds up the enhanced Dijkstra by multiple times. In this paper four approaches using advanced data structures like linked list, Fibonacci heap, binary heap and bi-directional Dijkstra are applied and their performance is also analyzed and compared on two classes of graphs, dense and sparse graphs.**

Keywords—Dijkstra Algorithm, Binary Heap, Fibonacci Heap, Bi-directional Dijkstra

I. INTRODUCTION

With the progression of innovation and the broad exchange of information, the Dijkstra algorithm is applied to some real-world problems like digital mapping services, social networking applications, telephone and road networking, flighting agenda, designating file server, robotic path, financial market, business, and other network-oriented problems [1]. As the network is developing and growing step by step, the multifaceted nature of such issues is expanding complex. Subsequently, best fitting algorithms and their methodologies become significant. The current paper focuses around execution time analysis of different implementations of the Dijkstra algorithm for dense and sparse graphs.

A. Dijkstra algorithm

The shortest path algorithm is a traditional algorithm in graph theory with the purpose to locate the shortest path from a specific source to various destinations. Dijkstra is a solitary source shortest path algorithm and the goal is to improve the execution time as well as memory utilization of the algorithm. Dijkstra algorithm does not have negative edge weights. It applies a greedy strategy to ideally tackle single-source shortest path problem.

The algorithm picks up a vertex with the minimum distance from the source, and this picked vertex relaxes up its adjacent vertices and these two stages run |V| times, where V is the number of vertices in the graph. Brute-force based solution of Dijkstra algorithm takes $O(|V|^2)$ time and $O(|V|^2)$ space [2].

B. Dense and Sparse Graphs

Regarding the density of a graph, there are two classes, sparse and dense graphs [3].

Dense graph: Dense graph is a graph where E the number of edges is which near to the maximal number of edges.
The mathematical representation of dense graph is:
Let G (V, E) is a graph where V and E are vertices and edges respectively then dense graph is: $|E|= \theta(|V|^2)$.

Sparse graph: Sparse graph is a graph in which the quantity of edges is near the insignificant number of edges.
The mathematical representation of dense graph is:
Let G (V, E) is a graph where V and E are vertices and edges respectively then sparse graph is: $|E|= O(|V|)$.

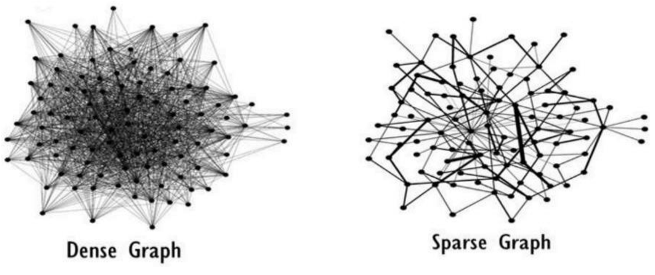


Fig. 1. Various types of graphs on the basis of graph density [4]

C. The approaches to optimize the Dijkstra algorithm

Dijkstra algorithm isn't improved in terms of space and time complexity, the algorithm is muddled to apply in network analysis for enormous test cases. For example, in the event that $V=10^{18}$ (number of vertices), at that point finding a minimum element by brute-force based solution will take 10^{18} iterations.

To handle this huge time complexity, advanced structures are applied to enhance Dijkstra algorithm. The use of a keen and numerical procedure speeds up the enhanced Dijkstra by multiple times. Here are some advanced data-structures applied in this analysis -

1) Linked list: A linked list is used to store the graph proficiently. A linked list is more proficient than a matrix as far as storing the graph. For example, on the off chance that V=10 and E=5 (sparse graph), at that point the linked list will take 5 units of space, yet the matrix will take 100 units of space. A linked list is the execution of a modern adjacency list which helps in decreasing the space. Structures and pointers are used to connect heterogenous information related with graph. Linked list is the implementation of modern adjacency list which helps in reducing the space from $O(|V|^2)$ to O(|E|) where E is the number of edges.

2) Fibonacci heap: A Fibonacci heap is a particular usage of the heap data structure that uses Fibonacci numbers. Fibonacci heap is utilized to actualize the need line component in Dijkstra algorithm, giving the calculation an exceptionally effective running time.

The Fibonacci heap is likewise used to execute a priority queue that finds and deletes the minimum component in O (log n) iterations(amortized), where n is the number of components in the heap. Reduction key activity is done in amortized cost of O(1). Although Fibonacci loads have gained notoriety for being delayed by and by [5].

3) Binary heap: A binary heap is the execution of a modern priority queue that proficiently finds and erases the minimum element in O(log n) iterations, where n is the number of elements in the heap. Relaxation of a vertex additionally takes O(log n) iterations [6].

In the event that $V=10^{18}$(number of vertices), at that point finding a minimum element by brute-force based solution will take 10^{18} cycles, and then again, the binary heap will take only 60 iterations to discover the minimum element. Binary heap and linked list aggregately diminished the running time from $O(|V|^2)$ to O(|E| log |V|).

4) Bidirectional Dijkstra: Bi-directional method runs the Dijkstra algorithm from the source vertex and objective vertex at the same time and the algorithm ends when a vertex is handled by forward binary heap and backward binary heap [7]. In the event that the graph structure is straight and on the off chance that $V=10^5$, at that point the unidirectional Dijkstra calculation will take 10^5 cycles to get ended yet then again, the bidirectional Dijkstra calculation will take $5*10^4$ iterations to end, which is improving the running time multiple times quicker. The primary component of this method is the amazing decrease in the quantity of cycles/iterations and the simplicity in finding the shortest path/route from one source to destination.

II. EXPERIMENTAL STUDY

In the present paper, all four approaches were implemented in ANSI C programing and executed on GCC compiler. The flowchart and algorithm are mentioned.

A. Brute-force method

1. Distance of nodes(except source node) from source node is infinity.
2. Find the unvisited node with minimum distance //complexity in O(n).
3. Mark the picked node as visited.
4. Relax the nodes adjacent to picked node.
5. Adjacent nodes which are marked as visited cannot be relaxed further.
6. Relaxing condition:
   if(dist[picked vertex] + edge(picked vertex, adjacent vertex) < dist[adjacent vertex])

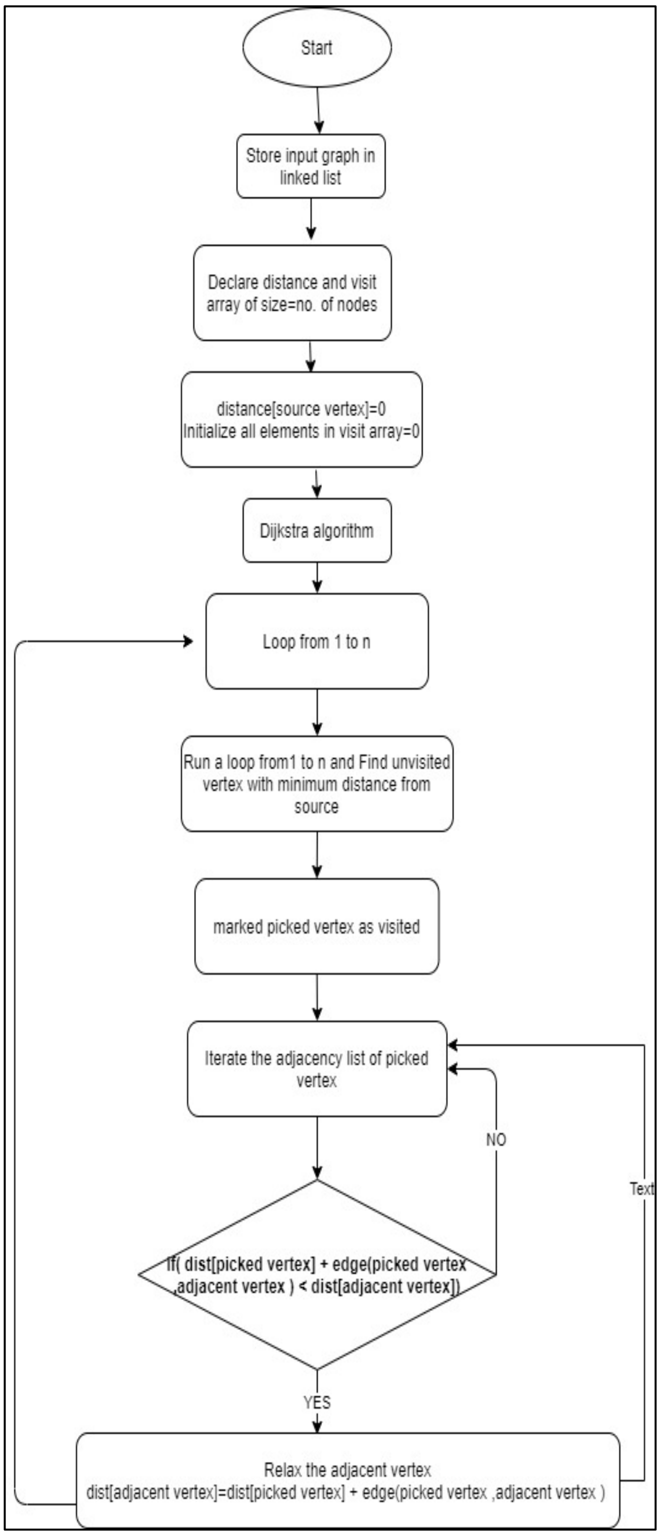

Start
Store input graph in linked list
Declare distance and visit array of size=no. of nodes
distance[source vertex]=0
Initialize all elements in visit array=0
Dijkstra algorithm
Loop from 1 to n
Run a loop from1 to n and Find unvisited vertex with minimum distance from source
marked picked vertex as visited
Iterate the adjacency list of picked vertex
NO
Text
if(dist[picked vertex] + edge(picked vertex ,adjacent vertex) < dist[adjacent vertex])
YES
Relax the adjacent vertex
dist[adjacent vertex]=dist[picked vertex] + edge(picked vertex ,adjacent vertex)

Fig. 2. Flowchart of Brute-force method

B. *Fibonacci Heap Implementation*

***Sub-algorithm Building Fibonacci heap*:**

```
1.  Allocate a new node.
2.  Make its parent,child,left, right as NULL.
3.  degree, mark=0
4.  Distance of nodes from source node is infinity except
    source node.
5.  If heapsize==0:
6.      Minimum element=input
7.  if heapsize>0:
8.      Minimum element=min (Minimum element, input)
```

Sub-algorithm Deleting Minimum element fromFibonacci heap :

```
1.  Unlink the minimum node from the root list and add
    all its children to the root list.
2.  Degree of node=number of children.
3.  All Subtrees should have unique degree.
4.  Max. Degree in the Root list= Log2 (N)
5.  Declare array pointer which is pointing to
    node which has degree=array's index
6.  Traverse the root list of Fibonacci heap.
7.  If (array[current node's degree]==NULL):
8.  then array[current node's degree] will point
    to current node.
9.  If (array[current node's degree]!=NULL):
10.     If (array[current node's degree] value >
        current node value)
11.     then make current node as parent   of
        array [current node's degree]
12.     array[current node's degree] will point to
        current node
13. else vice versa
14. After every subtree has unique degree in the
    Fibonacci heap.
15. Traverse the array pointer and find the
    minimum element of the Fibonacci heap
```

Algorithm Decreasing key of Fibonacci heap:

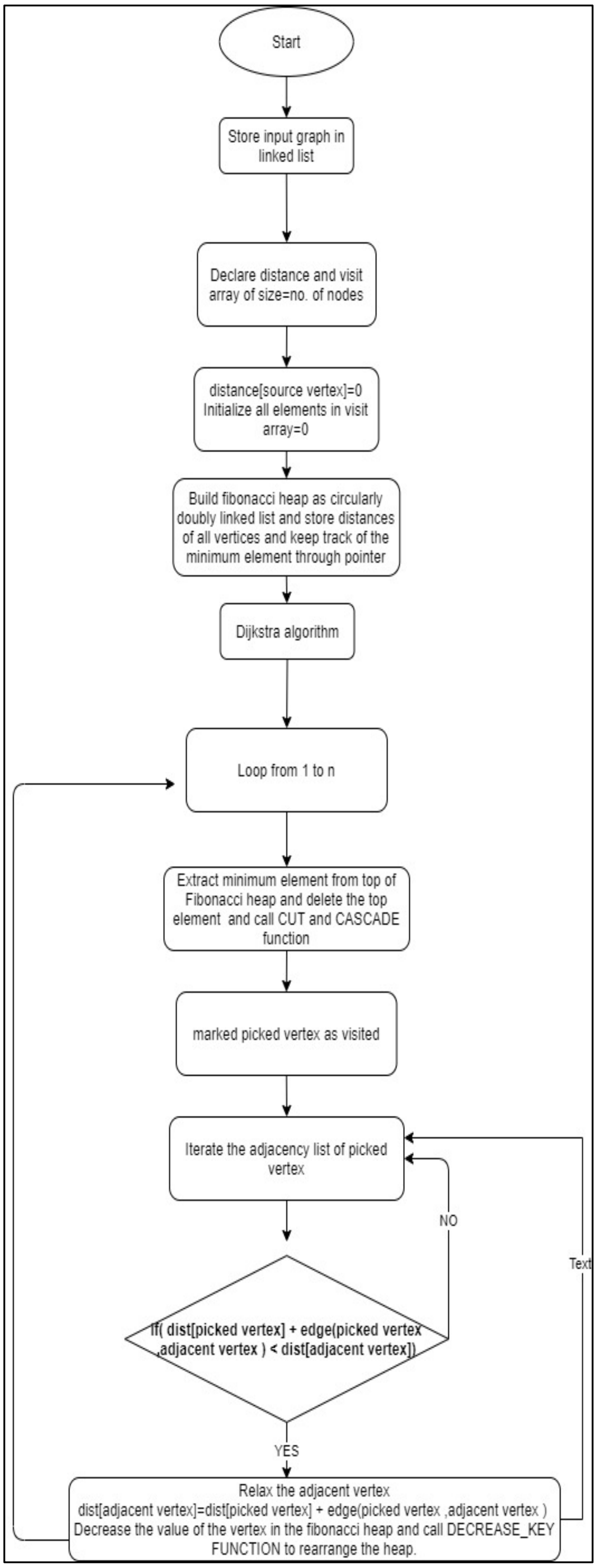
```
1.    Map array is used to get the position of a
      particular element in heap.
2.    Assign: value of $i^{th}$ element=Decreased value.
```

Sub-Algorithm CUT Function():

```
1.    If value of parent of $i^{th}$ node > value of $i^{th}$ node:
2.        then :
3.        Move $i^{th}$ node to the root list and mark it 0.
          Degree of parent is reduced by 1
```

Sub-Algorithm CASCADE Function ():

```
1.  If parent of $i^{th}$ node is marked as 0:
2.      then parent of $i^{th}$ node is marked as 1
3.  else
4.      if (parent of $i^{th}$ node is already marked as 1).
5.          then: CUT()
6.                CASCADE_CUT()
```



Start
Store input graph in linked list
Declare distance and visit array of size=no. of nodes
distance[source vertex]=0 Initialize all elements in visit array=0
Build fibonacci heap as circularly doubly linked list and store distances of all vertices and keep track of the minimum element through pointer
Dijkstra algorithm
Loop from 1 to n
Extract minimum element from top of Fibonacci heap and delete the top element and call CUT and CASCADE function
marked picked vertex as visited
Iterate the adjacency list of picked vertex
NO
Text
If(dist[picked vertex] + edge(picked vertex ,adjacent vertex) < dist[adjacent vertex])
YES
Relax the adjacent vertex
dist[adjacent vertex]=dist[picked vertex] + edge(picked vertex ,adjacent vertex)
Decrease the value of the vertex in the fibonacci heap and call DECREASE_KEY FUNCTION to rearrange the heap.

Fig. 3. Flowchart Fibonacci heap implementation

C. *Binary Heap Implementation*

***Sub-algorithm Building Minimum Heap*:**

1. Distance of nodes from source node is infinity except source node.
2. Put distance [source node] = 0 as root of the heap.
3. Traverse the heap array from 2nd index and put distance of all other nodes.

***Sub-algorithm Deleting Minimum Node from binary heap*:**

```
1.   Move the last element of heap to 1st index of
     heap array.
2.   Rearrange the heap such that min. element Is on
     top.
3.   Start from top element:
4.   Child1=2*(parent's index)+1
5.   Child2=2*(parent's index)+2

6.   If (distances[children] < distance[parent]):
7.           Take min. of distances of Child1,
             Child2 and swap the minimum one with
             the parent node.
8.           Make the minimum element as parent and
             Repeat the above steps.
9.   else: Do nothing
```

***Sub-algorithm Decrease a value of ith element in heap*:**

```
1.   Map array is used to get the position of a
     particular element in heap.
2.   Assign value of ith element=Decreased value
3.   Heap might get disturbed so rearrange it.
4.   Start from the ith element and go upwards In
     heap
5.   Compare its value with parent.
6.   Parent's index in heap=i / 2;

7.   while Parent's value>ith element's value:
8.           swap(parent,ith element)
9.           update the map array
10.          ith element=parent(going upwards).
```

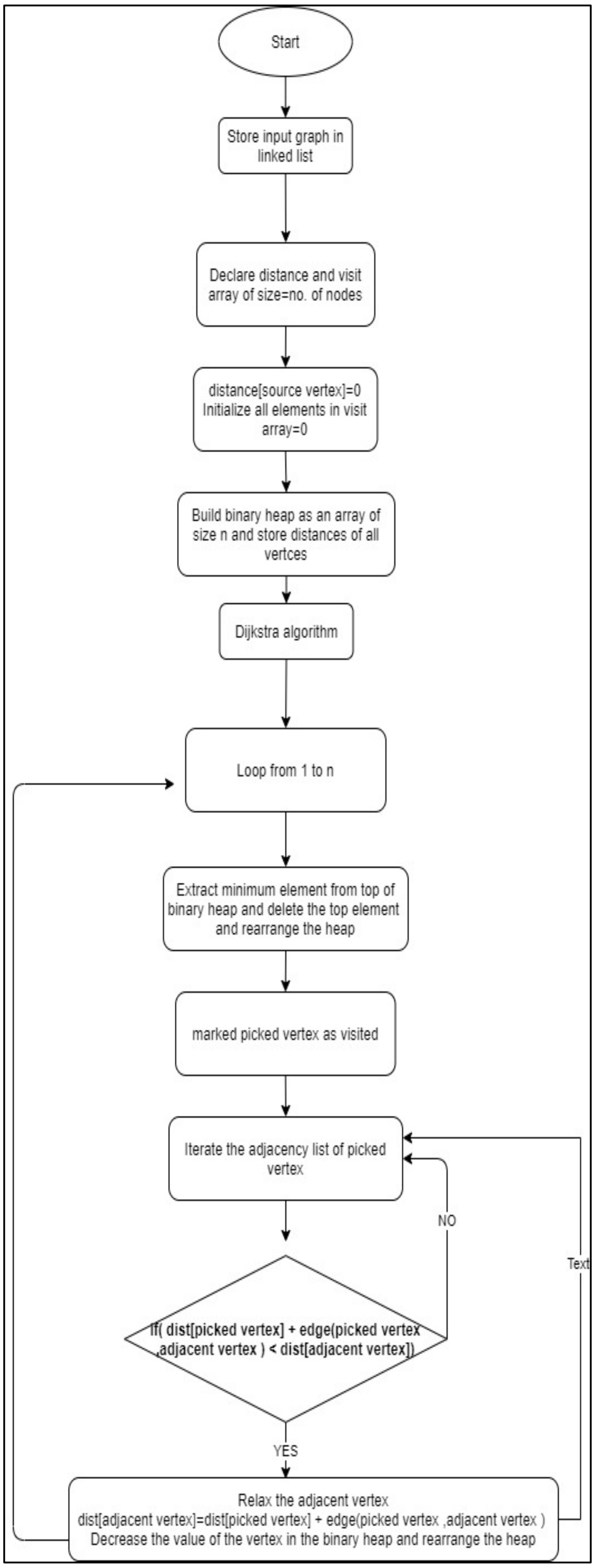

Start
Store input graph in linked list
Declare distance and visit array of size=no. of nodes
distance[source vertex]=0
Initialize all elements in visit array=0
Build binary heap as an array of size n and store distances of all vertces
Dijkstra algorithm
Loop from 1 to n
Extract minimum element from top of binary heap and delete the top element and rearrange the heap
marked picked vertex as visited
Iterate the adjacency list of picked vertex
If(dist[picked vertex] + edge(picked vertex ,adjacent vertex) < dist[adjacent vertex])
NO
YES
Text
Relax the adjacent vertex
dist[adjacent vertex]=dist[picked vertex] + edge(picked vertex ,adjacent vertex)
Decrease the value of the vertex in the binary heap and rearrange the heap

Fig. 4. Flowchart binary heap Implementation

D. Bi-directional Dijkstra Implementation

Optimized_Dijkstra():

```
1.  input(number_of_nodes)
2.  input(number_of_edges)
3.  FOR LOOP till number_of_edges:
4.      input(node_1)
5.      input(node_2)
6.      input(edge_weight)
7.      Call Build_Graph(node_1,node_2,edge_weight)
8.      Call Build_Graph(node_2,node_1,edge_weight)
9.  declare Forward_distance [] and Backward_distance
    []array
10. Forward_distance[source]=0
11. Backward_distance [destination]=0
12. Forward_distance [ all_nodes_except_source ] =
    INFINITY
13. Backward_distance[all_nodes_except_destination
        ] = INFINITY
```

Sub Algorithm Build_Graph (node_1, node_2, edge_weight):

```
1.  put node_2 into linked_list of node_1
2.  put node_1 into linked_list of node_2
```

Sub Algorithm Build_Binary_Heap (Binary_Heap_type, top)

```
1.  //Binary_Heap_Type means Forward_heap OR
    Backward_heap
2.  declare Binary_Heap_type[] array
3.  Top_of_binary_heap=distance[top]
4.  Put distance [Other_nodes_except_top] below the
    top node element
```

Algorithm Bidirectional_Dijkstra( )

```
1.  processed[all_nodes]=0
2.  //processed[i] signifies that node i's distance   from
    source node is minimizied.
3.  WHILE Forward_Heap_size>0 AND
    Backward_Heap_size>0 :
4.  //Forward Dijkstra Starts
5.  Dijkstra(Forward_Dijkstra)
6.  //Check if final path is going through 'pick' node and
    if it is minimizing
7.  //the final distance from source to destination
8.          if (Final_distance > Forward_distance[pick] +
    Backward_distance[pick]):
9.          Final_distance =
    Forward_distance[pick] +
    Backward_distance[pick]

10. //Above line is minimizing the final distance
    from source to destination
11. If (processed[pick]==1): break
    //if picked distance is already minimized
12. Then Bidirectional algorithm terminates.
13. else processed[pick]=1
14.    //pick's distance from source node is minimizied.
15.     //Backward Dijkstra Starts
16.     Dijkstra(Backward_Dijkstra)
17.     //Check if final path is going through
    'pick' node and if it is minimizing
18.     //the final distance from source to destination
19.     if Final_distance >
    Forward_distance[pick] +
    Backward_distance[pick]:
20.             Final_distance =
    Forward_distance[pick] +
    Backward_distance[pick]
21.             //Above line is minimizing the
    final distance from source to destination
22.      if processed[pick]==1: break
23.      //if pick's distance is already
    minimized then Bidirectional
    algorithm terminates.
24.      else processed[pick]=1
25. //pick's distance from source node is
    minimized.
```

Sub Algorithm Dijkstra (Dijkstra_type)

```
1.  //Binary_heap_type means Forward_Dijkstra OR
    Backward_Dijkstra
2.  int pick=top_element_of_heap
3.  //pick signifies the node with minimum distance from
    source node.
4.  //Delete Top element from Forward_Binary_Heap to
    access next node in next iteration
5.  Delete_Top_Element (Binary_Heap_Type)
    //Binary_Heap_Type means Forward_heap OR
    Backward_heap
6.  //Traverse the adacency_list of pick to relax the
    adjacnent nodes of pick
7.  FOR LOOP till size_of_adjacencylist_of_pick:
8.  //Relax the adjacent node if possible
9.  If (distance_type [pick] + edge_weight
    (pick,adjacent_node)
    < distance_type[adjacent]):
10. //distance_type means Forward_distance OR
    backward_distance
11. //Relax the adjacent node as condition is satisfied
12. distance_type [adjacent_node] = distance_type[pick]
    + edge_weight (pick,adjacent_node)
    Relax_Value_Of_Node(adjacent_node,Binary_Heap
    _ Type)
    //Reflect the changes in minimum heap
```

Sub Algorithm Delete_Top_Element (Binary_Heap_Type):

```
1.  //Binary_Heap_Type means Forward_heap OR
    Backward_heap
2.  MOVE last_node_in_heap TO
    Top_position_in_heap
3.  //Top element deleted
4.  //Rearrange the heap
5.  WHILE True:
6.  LEFT_CHILD=2*(Index_of_parent_in_heap)
    //left child node
```

```
7.  RIGHT_CHILD=2*(Index_of_parent_in_heap) +1
    //right child node
8.  //Go downwards in heap if minimum heap property
    is violated
9.  If (Parent > LEFT_CHILD OR RIGHT_CHILD):
10. SWAP Parent and Minimum (LEFT_CHILD,
    RIGHT_CHILD)
11. Parent=Minimum (LEFT_CHILD, RIGHT_CHILD)
12. else break
```

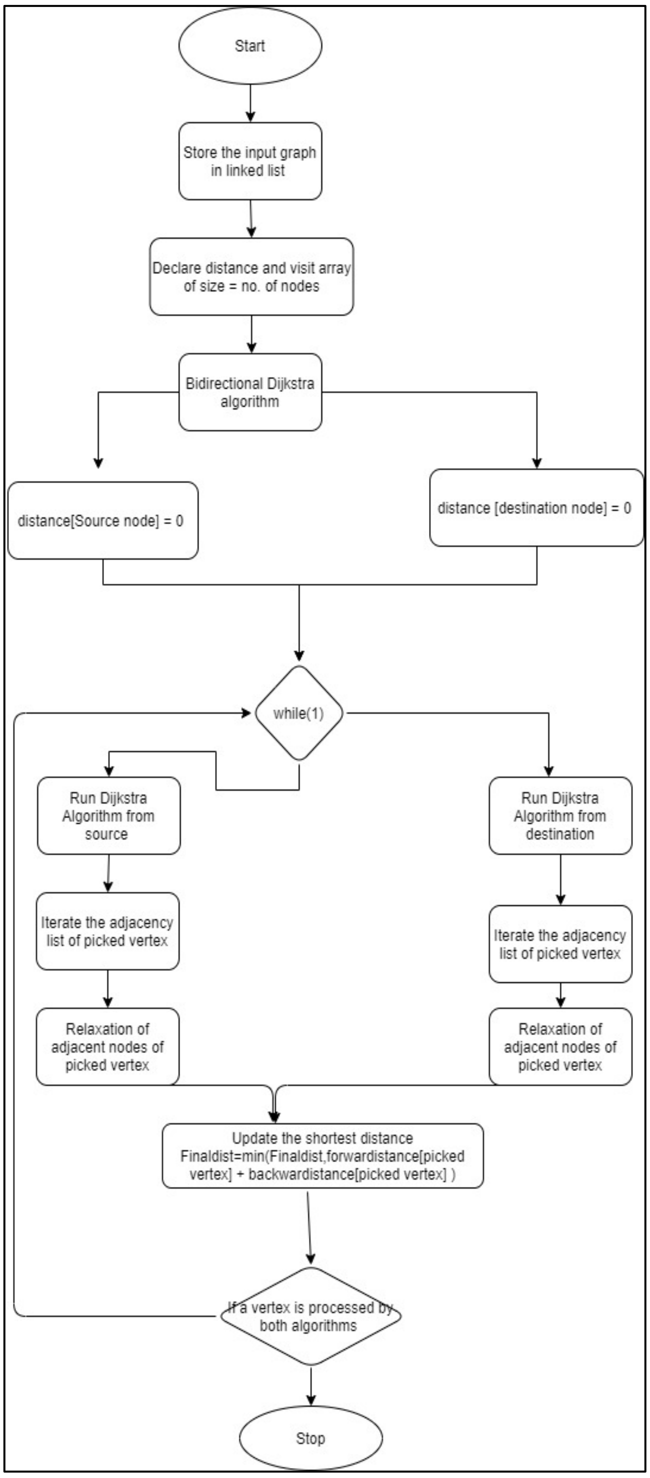

Start
Store the input graph in linked list
Declare distance and visit array of size = no. of nodes
Bidirectional Dijkstra algorithm
distance[Source node] = 0
distance [destination node] = 0
while(1)
Run Dijkstra Algorithm from source
Run Dijkstra Algorithm from destination
Iterate the adjacency list of picked vertex
Iterate the adjacency list of picked vertex
Relaxation of adjacent nodes of picked vertex
Relaxation of adjacent nodes of picked vertex
Update the shortest distance Finaldist=min(Finaldist,forwardistance[picked vertex] + backwardistance[picked vertex])
If a vertex is processed by both algorithms
Stop

Fig. 5. Flowchart of Bi-directional Dijkstra

III. RESULT ANALYSIS AND DISCUSSION

The outcome of execution time (in milliseconds) was done on two distinct classes of the graphs on the basis of graph density (dense and sparse) as shown in the Table I. There are 18 test cases in the experiment, which are used for all four proposed implementations (Brute-force based solution: BFB, Fibonacci heap implementation: FHI, Binary heap implementation: BHI and Bi-Directional Dijkstra: BDD) on same processor (i5 7th generation, 8GB RAM). In these test cases a graph is represented by N*V where N is number of nodes and V is number of vertices. Decision point is best performance implementation on the basis of execution time.

TABLE I EXECUTION TIME (ms)

*N*V*	*Case No*	*BFB*	*BHI*	*BDD*	*FHI*	*D-point*
1E4 * 1E5	Case 1	392	8	1	282	BDD
1E4 * 5E5	Case 2	485	32	2	291	BDD
1E4 * 10E5	Case 3	526	80	3	303	BDD
1E4 * 500	Case 4	16	1	1	292	BDD/BHI
1E4 * 2000	Case 5	79	1	1	294	BDDBHI
1E4 * 4000	Case 6	156	2	2	296	BDD/BHI
5E4 * 1E5	Case 7	9417	22	1	9647	BDD
5E4 * 5E5	Case 8	10442	44	4	9664	BDD
5E4 * 10E5	Case 9	10828	66	4	9723	BDD
5E4 * 500	Case 10	125	2	3	9682	BHI
5E4 * 2000	Case 11	437	3	4	9643	BHI
5E4 * 4000	Case 12	806	4	3	9682	BDD
1E5 * 1E5	Case 13	38559	34	2	7023	BDD
1E5 * 5E5	Case 14	39672	68	2	7054	BDD
1E5 * 10E5	Case 15	37172	94	3	7103	BDD
1E5 * 500	Case 16	187	4	1	7162	BDD
1E5* 2000	Case 17	798	7	3	7191	BDD
1E5 * 4000	Case 18	1515	8	3	7215	BDD

TABLE II RANKING OF IMPLEMENTATION

Case No	*BFB*	*BHI*	*BDD*	*FHI*
Case 1	4	2	1	3
Case 2	4	2	1	3
Case 3	4	2	1	3
Case 4	3	1	1	4
Case 5	3	1	1	4
Case 6	3	1	1	4
Case 7	3	2	1	4
Case 8	4	2	1	3
Case 9	4	2	1	3
Case 10	3	1	2	4
Case 11	3	1	2	4
Case 12	3	2	1	4
Case 13	4	2	1	3
Case 14	4	2	1	3
Case 15	4	2	1	3
Case 16	3	2	1	4
Case 17	3	2	1	4
Case 18	3	2	1	4

Table II is a continuation of Table I where it represents ranking of all four proposed implementations according to the best to worst execution time for each case (Case 1-18). Numbering from 1-4 have following meaning in Table II -

i. *Best in terms of execution time*
ii. *Second best in terms of execution time*
iii. *Third best in terms of execution time*
iv. *Forth best in terms of execution time*

Based on Table I and II the performance is examined as follows –.

A. *Analysis 1 (Performance Comparison between BFB and FHI for Sparse graph)*

Table III represents an execution time analysis between BFB and FHI for sparse graph test cases. As per the outcomes, BFB performs in a better way than FHI for sparse graphs which can also be concluded from Figure14. As appeared in the line graphs of Figure 6, BFB is continually increasing in normal pattern as the number of nodes and vertices increment all the while. While on account of FHI, as the number of nodes increment from 1E5 to 5E5 the values and line graph shoots up. Yet, as the diagram turned a little dense, the execution time of FHI got slightly diminished.

TABLE III EXECUTION TIME OF BFB AND FHI FOR SPARSE GRAPH

*N*V*	*BFB*	*FHI*
1E4 * 500	16	292
1E4 * 2000	79	294
1E4 * 4000	156	296
5E4 * 500	125	9682
5E4 * 2000	437	9643
5E4 * 4000	806	9682
1E5 * 500	187	7162
1E5* 2000	798	7191
1E5 * 4000	1515	7215

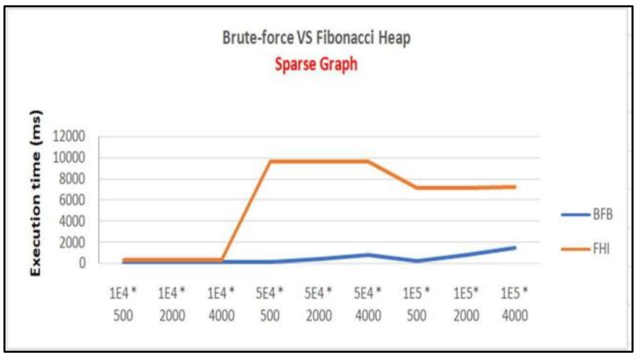


Fig. 6. Performance curve between BFB and FHI for Sparse Graph

B. *Analysis 2 (Performance Comparison between BFB and FHI for Dense graph)*

Table IV shows to an execution time analysis between BFB and FHI for dense graph test cases. As per the outcomes, FHI performs far better than BFB for dense graphs. As indicated by Table I and II, there is just one case (Case 7) in which BFB has demonstrated better execution time when contrasted with FHI by the distinction of 230ms.

Figure 7, FHI's executing time is continually increasing consistently as dense graph is getting expanded or getting complexed. While on account of BFB, as the number of nodes increase, the line graph shoots-up. In the case of FHI, there is not much difference between the time complexities of sparse graph test cases and dense graphs test cases. According to analysis 1, BFB shows a constant increment in execution time but in the dense graph BFB has shown an abnormal increment in execution time as the number of nodes and vertices increases.

TABLE IV EXECUTION TIME OF BFB AND FHI FOR DENSE GRAPH

*N*V*	*BFB*	*FHI*
1E4 * 1E5	392	282
1E4 * 5E5	485	291
1E4 * 10E5	526	303
5E4 * 1E5	9417	9647
5E4 * 5E5	10442	9664
5E4 * 10E5	10828	9723
1E5 * 1E5	38559	7023
1E5 * 5E5	39672	7054
1E5 * 10E5	37172	7103

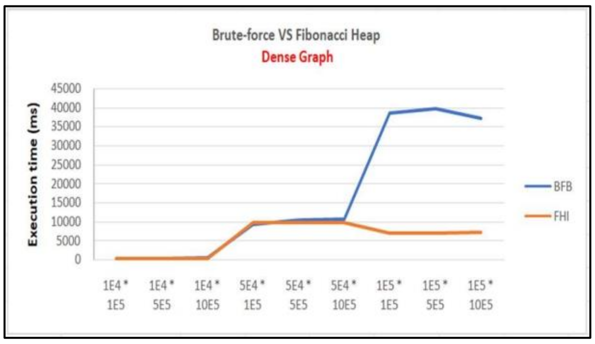


Fig. 7. Performance curve between BFB and FHI for dense Graph

C. *Analysis 3 (Performance Comparison between BHI and BDD for Sparse graph)*

Table V and figure 8 represent execution time analysis among BHI and BDD for sparse graph test cases. As per the outcomes, both BDD and BHI have similar execution time with some minute differences in some cases. According to Table II, in case 4, 5 and 6 both BHI and BDD have same execution time. In case 10 and 11 BHI has demonstrated better execution time when contrasted with BDD by the distinction of 1ms.

TABLE V EXECUTION TIME OF BHI AND BDD FOR SPARSE GRAPH

*N*V*	*BHI*	*BDD*
1E4 * 500	1	1
1E4 * 2000	1	1
1E4 * 4000	2	2
5E4 * 500	2	3
5E4 * 2000	3	4
5E4 * 4000	4	3
1E5 * 500	4	1
1E5* 2000	7	3
1E5 * 4000	8	3

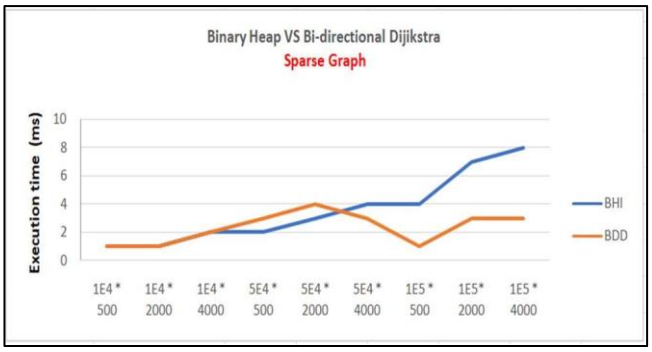


Fig. 8. Performance curve between BHI and BDD for Sparse Graph

D. *Analysis 4 (Performance Comparison between BHI and BDD for dense graph)*

Table V represents execution time analysis among BHI and BDD for dense graph test cases. As per the outcomes, BDD performs in a way that is better than BHI for dense graphs which can be concluded from Table II and Figure 9. According to Table V The execution time of BDD lies between 1-4 while in the BHI the execution time is in double digit with decreasing-increasing variations, as the number of nodes and vertices increases. These variations can be seen in the Figure 16 as a Saw tooth pattern. The line graph of BDD is practically consistent with a slight aggravation pattern.

TABLE-V EXECUTION TIME OF BHI AND BDD FOR DENSE GRAPH

*N*V*	*BHI*	*BDD*
1E4 * 1E5	8	1
1E4 * 5E5	32	2
1E4 * 10E5	80	3
5E4 * 1E5	22	1
5E4 * 5E5	44	4
5E4 * 10E5	66	4
1E5 * 1E5	34	2
1E5 * 5E5	68	2
1E5 * 10E5	94	3

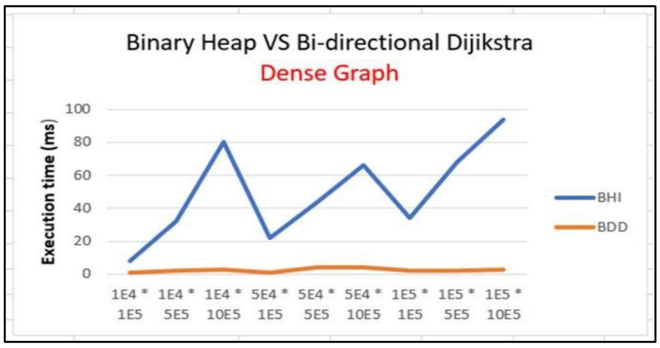


Fig. 9. Performance curve between BHI and BDD for Dense Graph

IV. CONCLUSION

After closely comparing the execution time of four algorithms on various real-world test cases, Bidirectional Dijkstra algorithm comes out to be the most efficient algorithm out of the four algorithms on the basis of time complexity. The chosen algorithm wins the race among all other algorithms in almost every test case. Therefore, it is proved that this algorithm can process huge real-world graphs in a very small amount of time. The present paper also investigates the performances of various implementation approaches applied to Dijkstra algorithm for sparse and dense graphs. As a future scope of the present research, an intelligent system may be developed in future to decide the applicability of an algorithm by seeing the nature of the problem in network-oriented domains.

REFERENCES

[1] R.K. Arjun, P. Reddy, Shama, and M. Yamuna, “Research on the optimization of Dijkstra‟s algorithm and its applications”, Int. J. Sci. Tech. Manag., vol. 4, no. 1, pp. 304 – 309, 2015.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, "Dijkstra's algorithm". Introduction to Algorithms, 2nd ed., MIT Press and McGraw–Hill. pp. 595–601, 2001, ISBN 0-262-03293-7..

[3] E.W Dijkstra, “A note on two problems in connection with graphs”, Numerische Mathematik, vol. 1 , pp. 269–271, 1959.

[4] https://blog.metaflow.fr/sparse-coding-a-simple-exploration-152a3c900a7c

[5] T.B. de Silveira, E. M. Duque, S.J.F. Guimarães, H. T. Marques-Neto and H. C. de Freitas, “Proposal of Fibonacci heap in the Dijkstra algorithm for low-power ad-hoc mobile transmissions”. IEEE Latin America Transactions, vol. 18, issue 3, pp. 623-630, 2020, doi: 10.1109/TLA.2020.9082735.

[6] A. -D. Andreiana, C. Bădică and E. Ganea, "An Experimental Comparison of Implementations of Dijkstra’s Single Source Shortest Path Algorithm Using Different Priority Queues Data Structures," 2020 24th International Conference on System Theory, Control and Computing (ICSTCC), Sinaia, Romania, 2020, pp. 124-129, doi: 10.1109/ICSTCC50638.2020.9259693

[7] F. Moo Mena, R. Hernandez Ucan, V. Uc Cetina and F. Madera Ramirez, "Web service composition using the bidirectional Dijkstra algorithm," in IEEE Latin America Transactions, vol. 14, no. 5, pp. 2522-2528, 2016, doi: 10.1109/TLA.2016.7530454.