Performance Analysis of BFS & DFS Algorithms for Various Applications

Amritam Sarcar
Dept of Computer Science,
500 W. University Avenue
El Paso, TX -79902
asarcar@miners.utep.edu

1. Abstract

One of the most simplest and intuitive algorithms for graph traversal are Breadth First Search and Depth First Search algorithms. In this paper, we try to investigate the application domain of these algorithms, in particular, how well they perform against each other for specific applications. This paper documents the several experiments that were carried out, along with their results and findings. Through this experimental study, our hypothesis that BFS or DFS works better in these conditions or applications are proved or disproved. It takes into account the performance issues including time, space and quality of results.

2. Introduction

In graph theory, breadth-first search (BFS) is one of the simplest graph searching algorithms that traverses the entire graph starting from the root node and visiting all the neighboring nodes. Then for each of these nodes it in turn again visits their neighboring nodes, and so on, until it finds its goal. BFS is an uninformed search algorithm that aims to expand all nodes of a given graph until it finds its solution. In other words, it blindly searches every node in the graph without considering the goal until it finds it. It does not use any heuristic.

Depth-First search (DFS) is an uninformed search that searching by visiting the first child node of the graph and moves deeper and deeper until it finds its goal or the node has no further child nodes. At his point, the search backtracks and returns to the most recent node that it has not yet explored. And this process continues.

3. Method / Algorithm

Algorithm BFS(G)	*1*

```
1. Input a graph G
2. for all n ∈ G, vertices
   2.1 setLabel(n, unexplored)
3. for all e ∈ G, edges
   3.1 setLabel(e, unexplored)
4. for all v ∈ G, vertices
   4.1 if getLabel(v) == unexplored
   4.2 BFS(G, v)
```

Algorithm BFS(G,v)	*2*

1. $L_0 \leftarrow$ new empty sequence
2. L_0.insertLast (s)
3. setLabel(s, visited)
4. $i \leftarrow 0$
5. while $\sim L_i$.isEmpty()
   5.1. $L_{i+1} \leftarrow$new empty sequence()
   5.2. for all v $\in L_{i+1}$.elements()
      5.2.1. for all e $\in$ G. incidentEdges(v)
      5.2.2. if getLabel(e) == *unexplored*
         5.2.2.1 w ← opposite(v, e)
         5.2.2.2 if getLabel(w) == *unexplored*
            5.2.2.2.1 setLabel(e, *discovery)*
            5.2.2.2.1 setLabel(w, *visited)*
            5.2.2.2.2 L_{i+1}.insertLast(w)
         5.2.2.3 else setLabel(e, *cross*)
6. i ← i+1

Algorithm DFS(G)	*1*

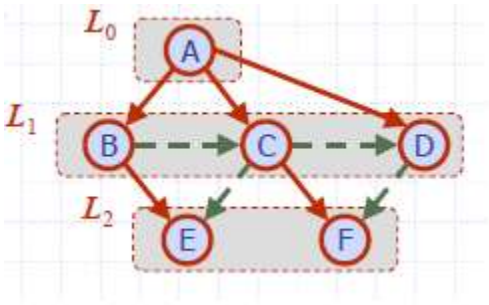
1. Input a graph G
2. for all n $\in$ G, vertices
   2.1 setLabel(n, *unexplored*)
3. for all e $\in$ G, edges
   3.1 setLabel(e, *unexplored*)
4. for all v $\in$ G, vertices
   4.1 if getLabel(v) == *unexplored*
   4.2 DFS(G, v)

Algorithm DFS(G,v)	*2*

1. Input graph G and a start vertex of G
2. setLabel(v, visited)
3. for all e $\in$G. incidentEdges(v)
   4.1 if getLabel(e) == *unexplored*
      3.1.1. w ← opposite(v, e)
      3.1.2. if getLabel(w) == *unexplored*
         3.1.2.1. setLabel(e, *discovery)*
         3.1.2.2. DFS(G, w)
      4.1.3 else setLabel(e, *cross*)
4. i ← i+1

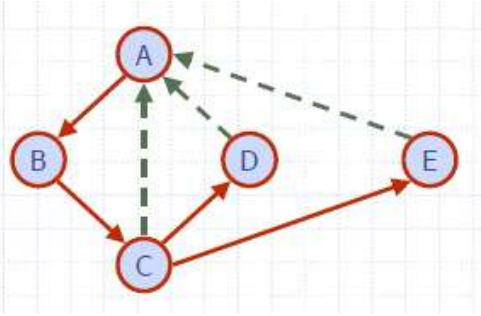
5. Properties of BFS, DFS

5.1. BFS



Property 1	Property 2	Property 3
BFS*(*G, s*)** visits all the vertices and edges in the connected component of ***Gs.	The discovery edges labeled ***BFS*(*G, s*)** form a spanning tree of the connected component of ***Gs.***	For each vertex ***v*** in ***Li*** The path of ***Ts*** from ***s*** to ***v*** has ***i*** edges Every path from ***s*** to ***v*** has at least ***i*** edges.

5.2. DFS



Property 1	Property 2
DFS*(*G, v*)** visits all the vertices and edges in the connected component of ***v	The discovery edges labeled ***DFS*(*G, v*)** form a spanning tree of the connected component of ***v.***

6. Index of diagrams

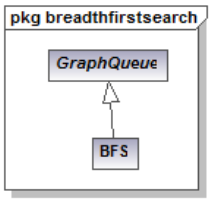
Class Diagram	Content of breadthfirstsearch Content of main	Content of depthfirstsearch Content of mazegen	Content of graphgen Content of src	Content of library
Package Diagram	Package dependencies of src			

7. Index of elements

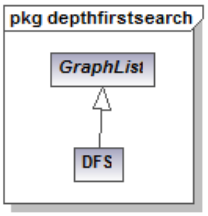
Class	BFS Exception GraphList Iterator List<T1->Integer> List<T1->String> Random SET<Key->Key> ST<Key->String,Val->SET<Key->String>> TreeMap URL	Comparable FindNode GraphQueue Iterator<T1->Key> List<T1->Long> Long RandomGenerator SET<Key->String> Stats TreeMap<T1->Key,T2->Val> Visitor	Comparable<T1->Key> Graph In Iterator<T1->Key> List<T1->String> Main Scanner Socket String TreeSet	DFS GraphGenerator Integer List List<T1->String> MazeGenerator SET ST TestCase TreeSet<T1->Key>
Component	breadthfirstsearch main	depthfirstsearch mazegen	graphgen	library
Interface	Iterable	Iterable<T1->Key>	Iterable<T1->Key>	Iterable<T1->String>
Package	breadthfirstsearch library src	Component View main Unknown Externals	depthfirstsearch mazegen	graphgen Root

8. Diagrams

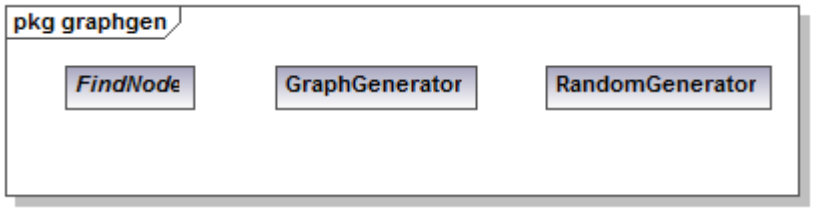
ClassDiagram **Content of breadthfirstsearch** (breadthfirstsearch)



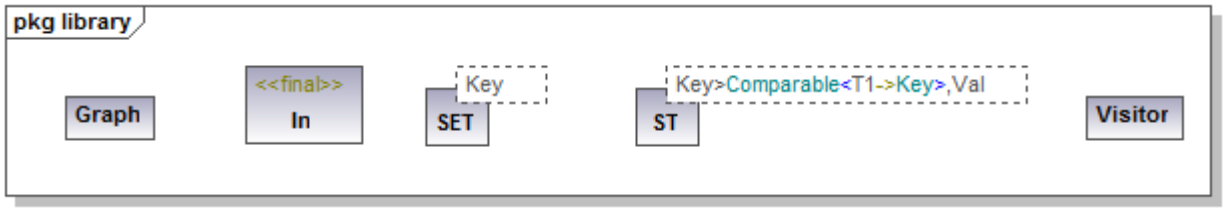
ClassDiagram **Content of depthfirstsearch** (depthfirstsearch)



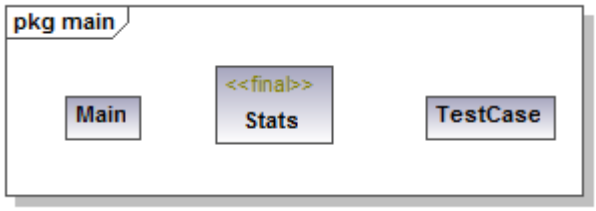
ClassDiagram **Content of graphgen** (graphgen)



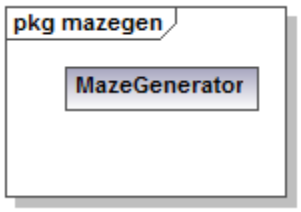
ClassDiagram **Content of library** (library)



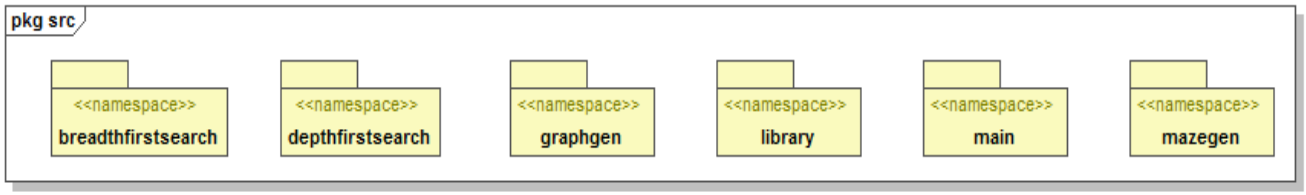
ClassDiagram **Content of main** (main)



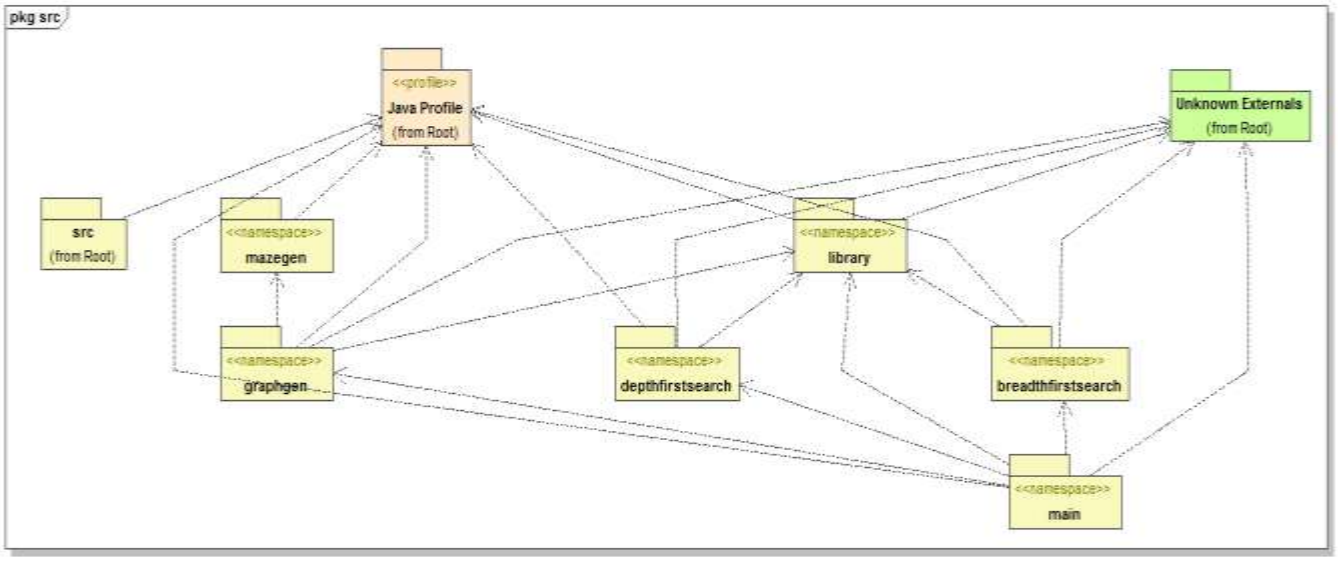
ClassDiagram **Content of mazegen** (mazegen)



ClassDiagram **Content of src** (src)



PackageDiagram **Package dependencies of src** (src)



9. Elements

Component **breadthfirstsearch**

owner	**Component View**
properties	qualified name Component View::breadthfirstsearch visibility public leaf false abstract false indirectlyInstantiated true code language Java6.0 (1.6) directory C:\Users\Amritam\Desktop\DFSandBFS\src\breadthfirstsearch use for code engineering true
source of relation	ComponentRealization **BFS** **GraphQueue**

Component **depthfirstsearch**

owner	**Component View**
properties	qualified name Component View::depthfirstsearch visibility public leaf false abstract false indirectlyInstantiated true code language Java6.0 (1.6) directory C:\Users\Amritam\Desktop\DFSandBFS\src\depthfirstsearch use for code engineering true
source of relation	ComponentRealization **DFS** **GraphList**

Component **graphgen**

owner	**Component View**
properties	qualified name Component View::graphgen visibility public leaf false abstract false indirectlyInstantiated true code language Java6.0 (1.6) directory C:\Users\Amritam\Desktop\DFSandBFS\src\graphgen use for code engineering true
source of relation	ComponentRealization **FindNode** **GraphGenerator** **RandomGenerator**

Component **library**

owner	**Component View**
properties	qualified name Component View::library visibility public leaf false abstract false indirectlyInstantiated true code language Java6.0 (1.6) directory C:\Users\Amritam\Desktop\DFSandBFS\src\library use for code engineering true
source of relation	ComponentRealization **Graph** **In** **SET** **ST** **Visitor**

Component **main**

owner	**Component View**
properties	qualified name Component View::main visibility public leaf false abstract false indirectlyInstantiated true code language Java6.0 (1.6) directory C:\Users\Amritam\Desktop\DFSandBFS\src\main use for code engineering true
source of relation	ComponentRealization **TestCase** **Main** **Stats**

Component **mazegen**

owner	**Component View**
properties	qualified name Component View::mazegen visibility public leaf false abstract false indirectlyInstantiated true code language Java6.0 (1.6) directory C:\Users\Amritam\Desktop\DFSandBFS\src\mazegen use for code engineering true
source of relation	ComponentRealization **MazeGenerator**

10. Packages & Classes

Package **src**

owner	**Root**
properties	qualified name src visibility public
ownedDiagrams	**Content of src** **Package dependencies of src**
ownedMember	**breadthfirstsearch** **depthfirstsearch** **graphgen** **library** **main** **mazegen**
source of relation	Dependency **Java Profile** ProfileApplication **Java Profile**
shown on diagram	**Package dependencies of src**
hyperlinks	**Content of src**

Package **breadthfirstsearch**

owner	**src**
properties	qualified name src::breadthfirstsearch visibility public <<namespace>> true
ownedDiagrams	**Content of breadthfirstsearch**
ownedMember	**BFS** **GraphQueue** **List<T1->String>**
source of relation	Dependency **library** **Unknown Externals** **Java Profile**
target of relation	Dependency **main**
shown on diagram	**Content of src** **Package dependencies of src**
hyperlinks	**Content of breadthfirstsearch**

Package **depthfirstsearch**

owner	**src**
properties	qualified name src::depthfirstsearch visibility public <<namespace>> true
ownedDiagrams	**Content of depthfirstsearch**
ownedMember	**DFS** **GraphList** **List<T1->String>**
source of relation	Dependency **library** **Unknown Externals** **Java Profile**
target of relation	Dependency **main**
shown on diagram	**Content of src** **Package dependencies of src**
hyperlinks	**Content of depthfirstsearch**

Package **graphgen**

owner	**src**
properties	qualified name src::graphgen visibility public <<namespace>> true
ownedDiagrams	**Content of graphgen**
ownedMember	**FindNode** **GraphGenerator** **List<T1->Integer>** **RandomGenerator**
source of relation	Dependency **mazegen** **library** **Unknown Externals** **Java Profile**
target of relation	Dependency **main**
shown on diagram	**Content of src** **Package dependencies of src**
hyperlinks	**Content of graphgen**

Package **library**

owner	**src**
properties	qualified name src::library visibility public <<namespace>> true
ownedDiagrams	**Content of library**
ownedMember	**Comparable<T1->Key>** **Graph** **In** **Iterable<T1->Key>** **Iterable<T1->Key>** **Iterable<T1->String>** **Iterator<T1->Key>** **Iterator<T1->Key>** **SET** **SET<Key->Key>** **SET<Key->String>** **ST** **ST<Key->String,Val->SET<Key->String>>** **TreeMap<T1->Key,T2->Val>** **TreeSet<T1->Key>** **Visitor**
source of relation	Dependency **Unknown Externals** **Java Profile**
target of relation	Dependency **breadthfirstsearch** **main** **graphgen** **depthfirstsearch**
shown on diagram	**Content of src** **Package dependencies of src**
hyperlinks	**Content of library**

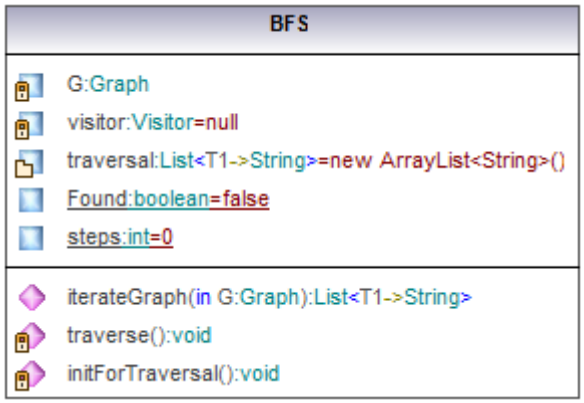
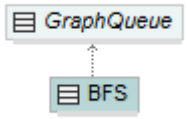
Package **main**

owner	**src**
properties	qualified name src::main visibility public <<namespace>> true
ownedDiagrams	**Content of main**
ownedMember	**List<T1->Long>** **List<T1->String>** **Main** **Stats** **TestCase**
source of relation	Dependency **breadthfirstsearch** **library** **Unknown Externals** **graphgen** **depthfirstsearch** **Java Profile**
shown on diagram	**Content of src** **Package dependencies of src**
hyperlinks	**Content of main**

Package **mazegen**

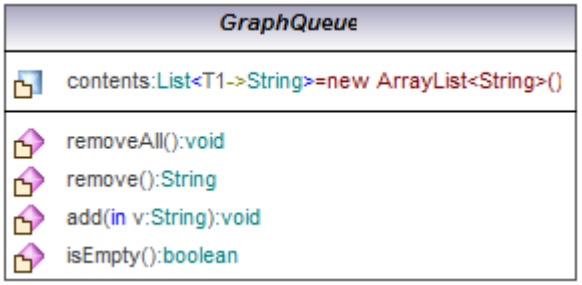
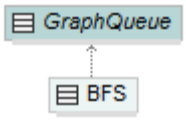
owner	**src**
properties	qualified name src::mazegen visibility public <<namespace>> true
ownedDiagrams	**Content of mazegen**
ownedMember	**MazeGenerator**
source of relation	Dependency **Java Profile**
target of relation	Dependency **graphgen**
shown on diagram	**Content of src** **Package dependencies of src**
hyperlinks	**Content of mazegen**

Class **BFS**

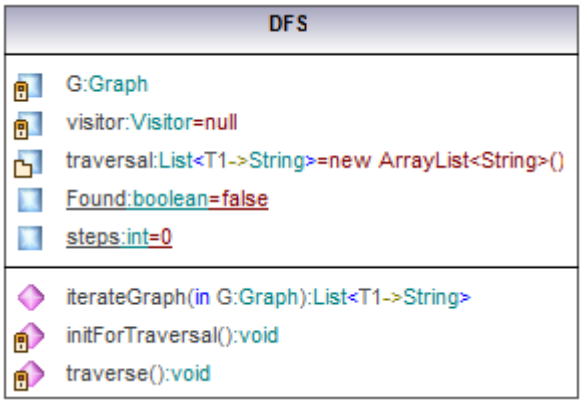
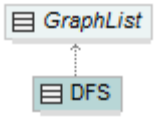
diagram	
hierarchy	
owner	**breadthfirstsearch**
properties	qualified name src::breadthfirstsearch::BFS visibility public leaf false abstract false active false code file name BFS.java code file path C:\Users\Amritam\Desktop\DFSandBFS\src\breadthfirstsearch\BFS.java <<annotations>> false <<static>> false <<final>> false <<strictfp>> false
ownedMember	**Found** **G** **initForTraversal** **iterateGraph** **steps** **traversal** **traverse** **visitor**
general	**GraphQueue**

target of relation	ComponentRealization **breadthfirstsearch**
typedElements	Class **TestCase** Property **bfs**
shown on diagram	**Content of breadthfirstsearch**
parameter	name direction type multiplicity default **return** **return** **void**
ownedMember	**return**

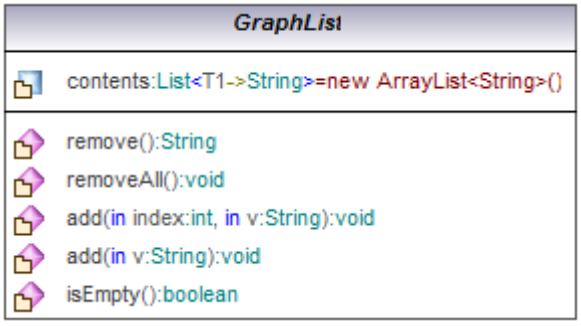
Class **GraphQueue**

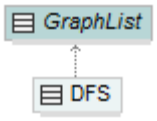
diagram	
hierarchy	
owner	**breadthfirstsearch**
properties	qualified name src::breadthfirstsearch::GraphQueue visibility public leaf false abstract true active false code file name GraphQueue.java code file path C:\Users\Amritam\Desktop\DFSandBFS\src\breadthfirstsearch\GraphQueue.java <<annotations>> false <<static>> false <<final>> false <<strictfp>> false
ownedMember	**add** **contents** **isEmpty** **remove** **removeAll**
specific	**BFS**
target of relation	ComponentRealization **breadthfirstsearch**
shown on diagram	**Content of breadthfirstsearch**

Class **DFS**

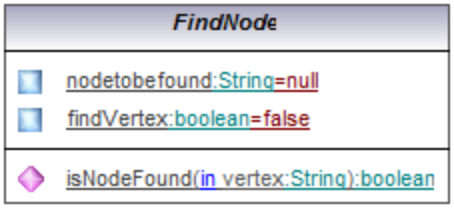
diagram	
hierarchy	
owner	**depthfirstsearch**
properties	qualified name src::depthfirstsearch::DFS visibility public leaf false abstract false active false code file name DFS.java code file path C:\Users\Amritam\Desktop\DFSandBFS\src\depthfirstsearch\DFS.java <<annotations>> false <<static>> false <<final>> false <<strictfp>> false
ownedMember	**Found** **G** **initForTraversal** **iterateGraph** **steps** **traversal** **traverse** **visitor**
general	**GraphList**
target of relation	ComponentRealization **depthfirstsearch**
typedElements	Class **TestCase** Property **dfs**
shown on diagram	**Content of depthfirstsearch**

Class **GraphList**

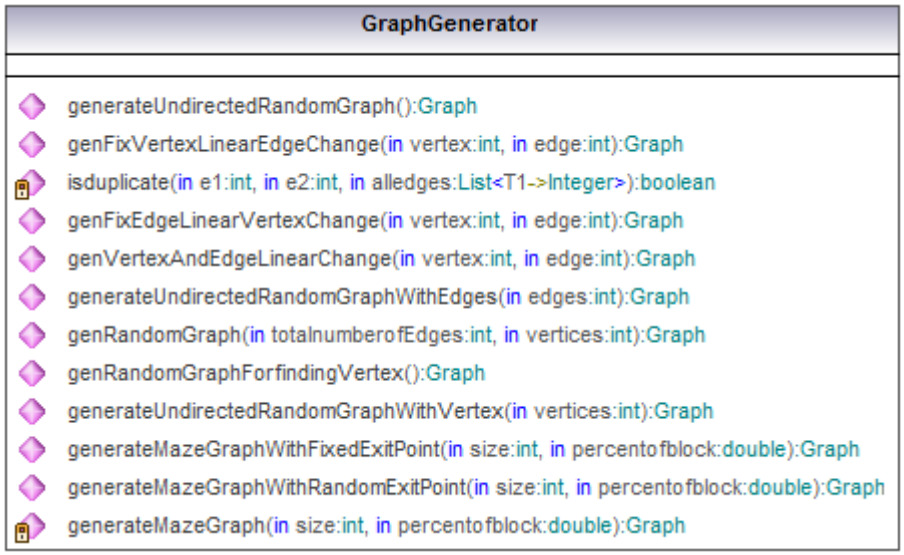
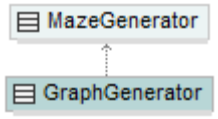
diagram	

hierarchy	
owner	**depthfirstsearch**
properties	qualified name src::depthfirstsearch::GraphList visibility public leaf false abstract true active false code file name GraphList.java code file path C:\Users\Amritam\Desktop\DFSandBFS\src\depthfirstsearch\GraphList.java <<annotations>> false <<static>> false <<final>> false <<strictfp>> false
ownedMember	**add** **add** **contents** **isEmpty** **remove** **removeAll**
specific	**DFS**
target of relation	ComponentRealization **depthfirstsearch**
shown on diagram	**Content of depthfirstsearch**

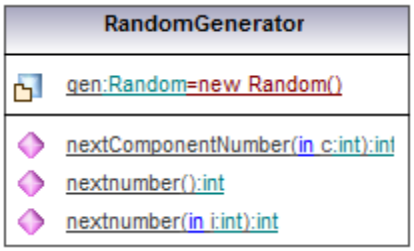
Class **FindNode**

diagram	
owner	**graphgen**
properties	qualified name src::graphgen::FindNode visibility public leaf false abstract true active false code file name FindNode.java code file path C:\Users\Amritam\Desktop\DFSandBFS\src\graphgen\FindNode.java <<annotations>> false <<static>> false <<final>> false <<strictfp>> false
ownedMember	**findVertex** **isNodeFound** **nodetobefound**
target of relation	ComponentRealization **graphgen**
shown on diagram	**Content of graphgen**

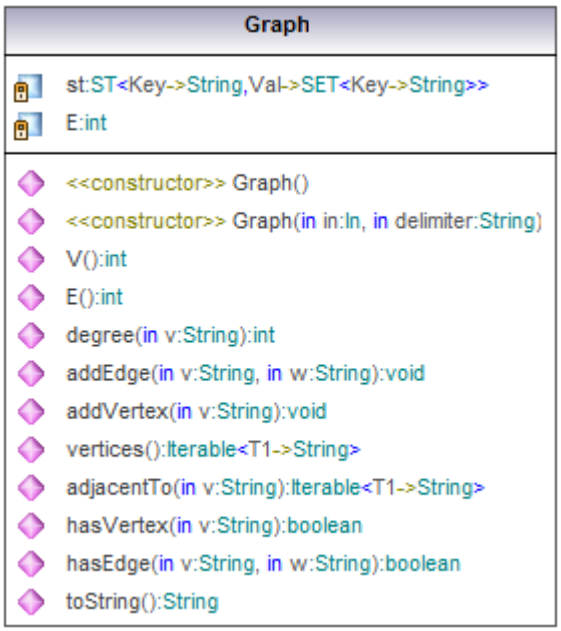
Class **GraphGenerator**

diagram	
hierarchy	
owner	**graphgen**
properties	qualified name src::graphgen::GraphGenerator visibility public leaf false abstract false active false code file name GraphGenerator.java code file path C:\Users\Amritam\Desktop\DFSandBFS\src\graphgen\GraphGenerator.java <<annotations>> false <<static>> false <<final>> false <<strictfp>> false
ownedMember	**generateMazeGraph** **generateMazeGraphWithFixedExitPoint** **generateMazeGraphWithRandomExitPoint** **generateUndirectedRandomGraph** **generateUndirectedRandomGraphWithEdges** **generateUndirectedRandomGraphWithVertex** **genFixEdgeLinearVertexChange** **genFixVertexLinearEdgeChange** **genRandomGraph** **genRandomGraphForfindingVertex** **genVertexAndEdgeLinearChange** **isduplicate**
general	**MazeGenerator**
target of relation	ComponentRealization **graphgen**
typedElements	Class **TestCase** Property **gen**
shown on diagram	**Content of graphgen**

Class **RandomGenerator**

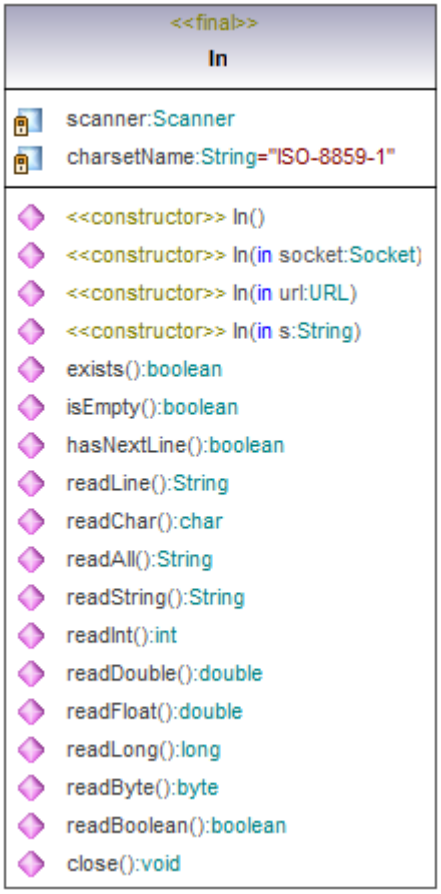
diagram	
owner	**graphgen**
properties	qualified name src::graphgen::RandomGenerator visibility public leaf false abstract false active false code file name RandomGenerator.java code file path C:\Users\Amritam\Desktop\DFSandBFS\src\graphgen\RandomGenerator.java <<annotations>> false <<static>> false <<final>> false <<strictfp>> false
ownedMember	**gen** **nextComponentNumber** **nextnumber** **nextnumber**
target of relation	ComponentRealization **graphgen**
shown on diagram	**Content of graphgen**

Class **Graph**

diagram	
owner	**library**
properties	qualified name src::library::Graph visibility public leaf false

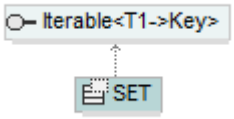
	abstract false active false code file name Graph.java code file path C:\Users\Amritam\Desktop\DFSandBFS\src\library\Graph.java <<annotations>> false <<static>> false <<final>> false <<strictfp>> false
ownedMember	**addEdge addVertex adjacentTo degree E E Graph Graph hasEdge hasVertex st toString V vertices**
target of relation	ComponentRealization **library**
typedElements	Class **BFS** Property **G** Operation **iterateGraph** Class **DFS** Property **G** Operation **iterateGraph** Class **GraphGenerator** Operation **generateMazeGraph generateMazeGraphWithFixedExitPoint generateMazeGraphWithRandomExitPoint generateUndirectedRandomGraph generateUndirectedRandomGraphWithEdges generateUndirectedRandomGraphWithVertex genFixEdgeLinearVertexChange genFixVertexLinearEdgeChange genRandomGraph genRandomGraphForfindingVertex genVertexAndEdgeLinearChange** Class **TestCase** Operation **runBFS runDFS** Class **Visitor** Operation **init**
shown on diagram	**Content of library**

Class **In**

diagram	

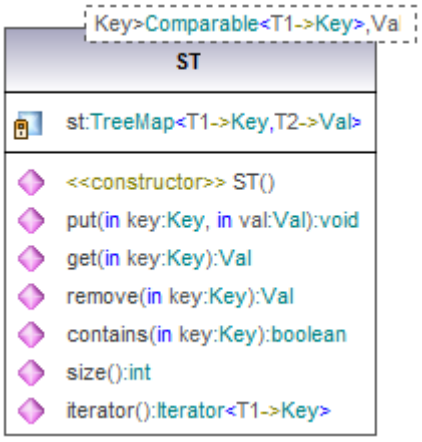
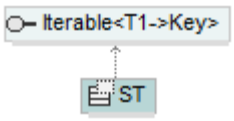
owner	**library**
properties	qualified name src::library::In visibility public leaf false abstract false active false code file name In.java code file path C:\Users\Amritam\Desktop\DFSandBFS\src\library\In.java <<annotations>> false <<static>> false <<final>> true <<strictfp>> false
ownedMember	**charsetName close exists hasNextLine In In In In isEmpty readAll readBoolean readByte readChar readDouble readFloat readInt readLine readLong readString scanner**
target of relation	ComponentRealization **library**
typedElements	Class **Graph** Operation **Graph**
shown on diagram	**Content of library**

Class **SET**

Diagram	
hierarchy	
owner	**library**
template parameters	name: **Key** kind: **Class** constrainingClassifier default
properties	qualified name src::library::SET visibility public leaf false abstract false active false code file name SET.java code file path C:\Users\Amritam\Desktop\DFSandBFS\src\library\SET.java

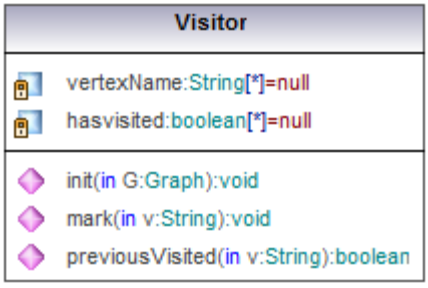
	<<annotations>> false <<static>> false <<final>> false <<strictfp>> false
ownedMember	**add contains intersects iterator remove size st toString union**
implemented interfaces	**Iterable<T1->Key>**
source of relation	InterfaceRealization **Iterable<T1->Key>**
target of relation	ComponentRealization **library**
bound elements	**SET<Key->Key> SET<Key->String>**
shown on diagram	**Content of library**

Class **ST**

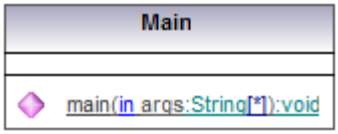
Diagram	
hierarchy	
owner	**library**
template parameters	name: **Key**; kind: **Class**; constrainingClassifier: **Comparable<T1->Key>**; default: name: **Val**; kind: **Class**
properties	qualified name src::library::ST visibility public leaf false abstract false active false code file name ST.java code file path C:\Users\Amritam\Desktop\DFSandBFS\src\library\ST.java <<annotations>> false <<static>> false

	<<final>> false <<strictfp>> false
ownedMember	**contains** **get** **iterator** **put** **remove** **size** **ST** **st**
implemented interfaces	**Iterable<T1->Key>**
source of relation	InterfaceRealization **Iterable<T1->Key>**
target of relation	ComponentRealization **library**
bound elements	**ST<Key->String,Val->SET<Key->String>>**
shown on diagram	**Content of library**

Class **Visitor**

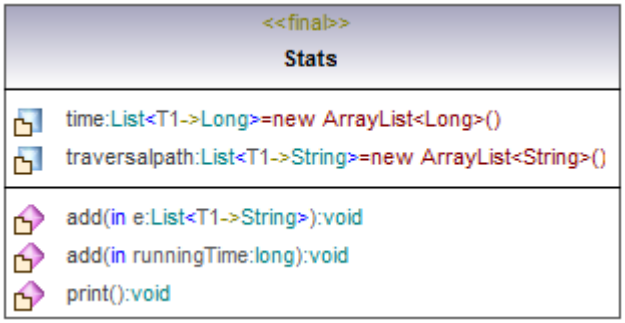
diagram	
owner	**library**
properties	qualified name src::library::Visitor visibility public leaf false abstract false active false code file name Visitor.java code file path C:\Users\Amritam\Desktop\DFSandBFS\src\library\Visitor.java <<annotations>> false <<static>> false <<final>> false <<strictfp>> false
ownedMember	**hasvisited** **init** **mark** **previousVisited** **vertexName**
target of relation	ComponentRealization **library**
typedElements	Class **BFS** Property **visitor** Class **DFS** Property **visitor**
shown on diagram	**Content of library**

Class **Main**

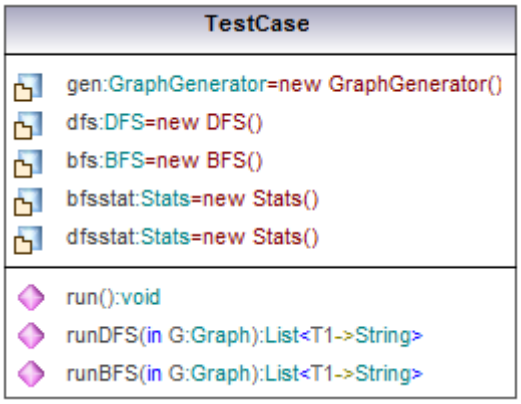
diagram	

owner	**main**
properties	qualified name src::main::Main visibility public leaf false abstract false active false code file name Main.java code file path C:\Users\Amritam\Desktop\DFSandBFS\src\main\Main.java <<annotations>> false <<static>> false <<final>> false <<strictfp>> false
ownedMember	**main**
target of relation	ComponentRealization **main**
shown on diagram	**Content of main**

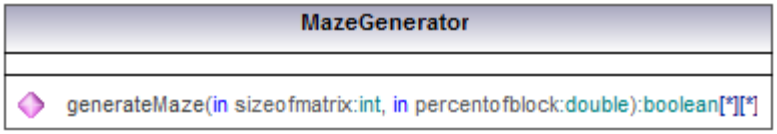
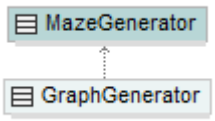
Class **Stats**

diagram	
owner	**main**
properties	qualified name src::main::Stats visibility public leaf false abstract false active false code file name Stats.java code file path C:\Users\Amritam\Desktop\DFSandBFS\src\main\Stats.java <<annotations>> false <<static>> false <<final>> true <<strictfp>> false
ownedMember	**add** **add** **print** **time** **traversalpath**
target of relation	ComponentRealization **main**
typedElements	Class **TestCase** Property **bfsstat** **dfsstat**
shown on diagram	**Content of main**

Class **TestCase**

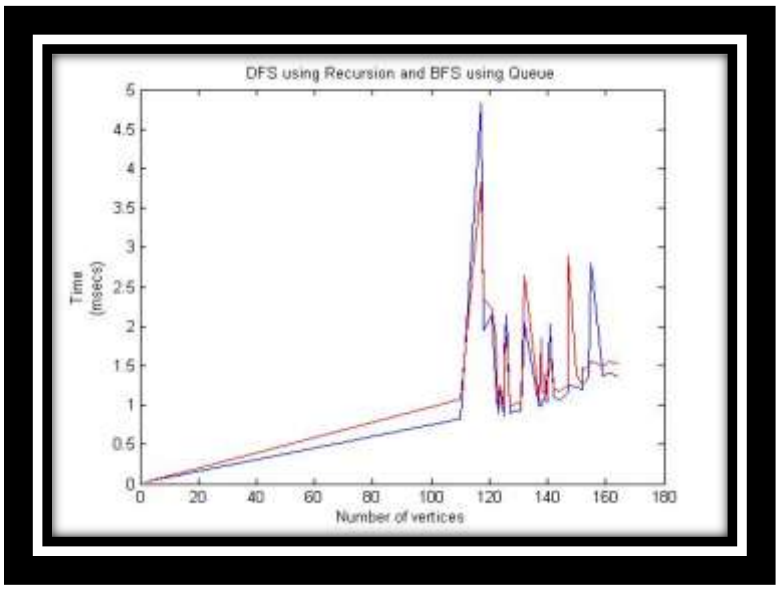
diagram	
owner	**main**
properties	qualified name src::main::TestCase visibility public leaf false abstract false active false code file name TestCase.java code file path C:\Users\Amritam\Desktop\DFSandBFS\src\main\TestCase.java <<annotations>> false <<static>> false <<final>> false <<strictfp>> false
ownedMember	**bfs** **bfsstat** **dfs** **dfsstat** **gen** **run** **runBFS** **runDFS**
target of relation	ComponentRealization **main**
shown on diagram	**Content of main**

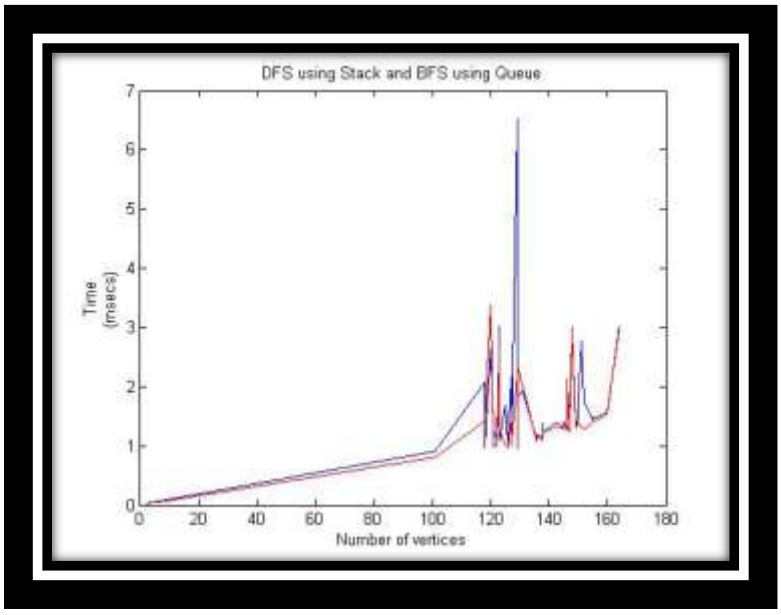
Class **MazeGenerator**

diagram	
hierarchy	
owner	**mazegen**
properties	qualified name src::mazegen::MazeGenerator visibility public leaf false abstract false active false code file name MazeGenerator.java code file path C:\Users\Amritam\Desktop\DFSandBFS\src\mazegen\MazeGenerator.java <<annotations>> false

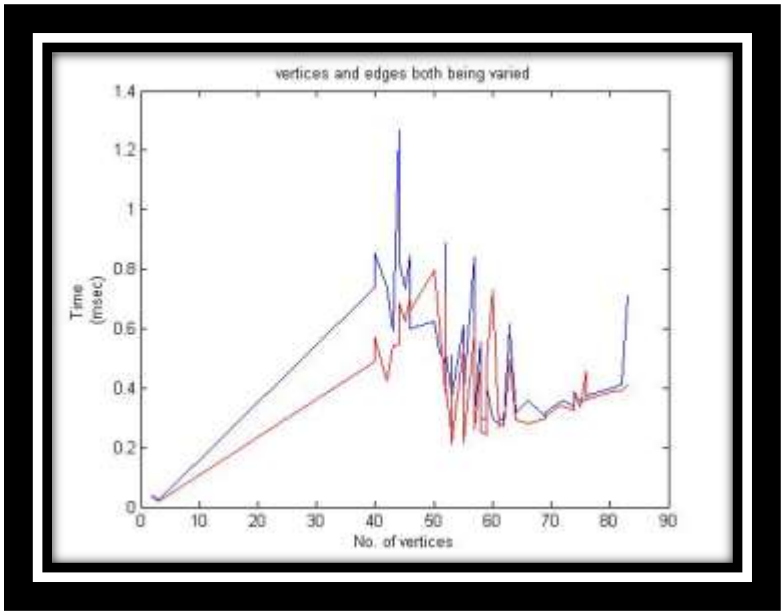
	<<static>> false <<final>> false <<strictfp>> false
ownedMember	**generateMaze**
specific	**GraphGenerator**
target of relation	ComponentRealization **mazegen**
shown on diagram	**Content of mazegen**

11. Results

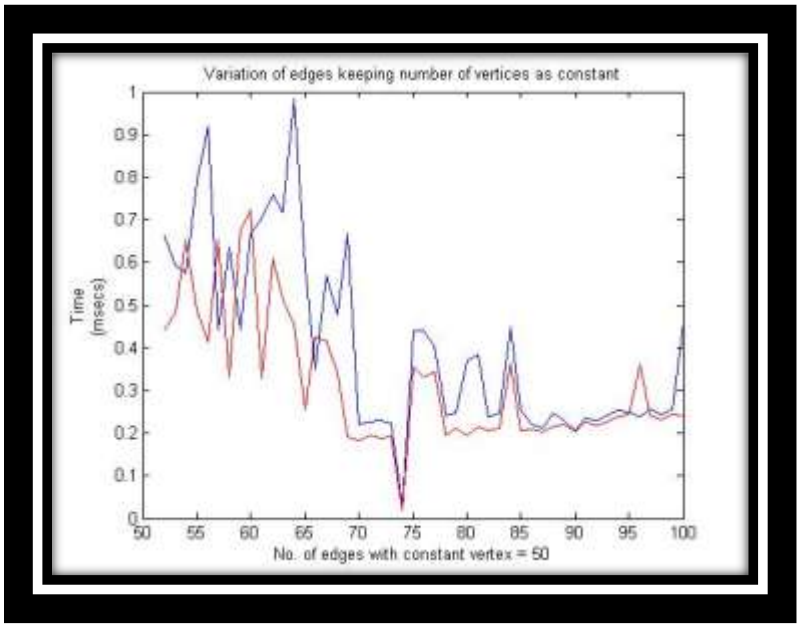




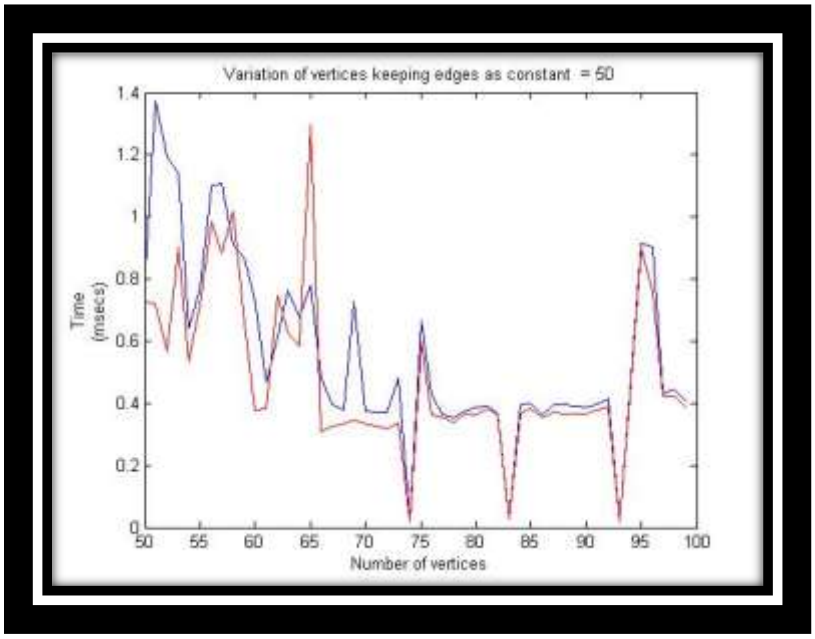
The above two graphs shows what happens when the implementation can go wrong. In the first graph, using recursion we see that DFS always take more time to travel the entire graph, which may not always hold true. However this happens because Recursion does not use pushing and popping of elements, since it is done implicitly by the JVM. Thus using a stack for the DFS is better suited and shows that there is no correlation between vertices and the type of search. It shows that under certain circumstances, BFS wins over DFS and other times the reverse happens.



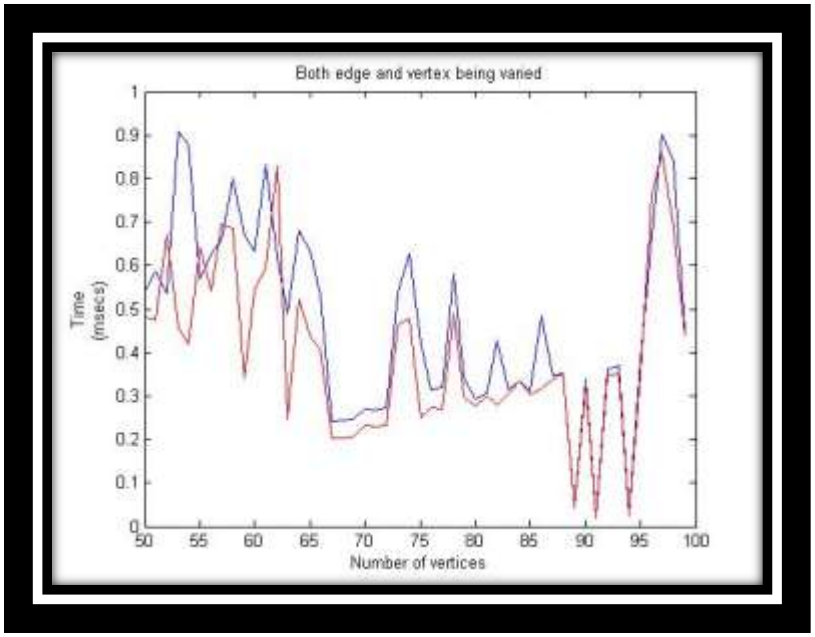
The above graph shows how DFS and BFS varies in their time of traversal when both the number of vertices and the number of edges are varied. This graph also shows that although there is no clear winner, however BFS is almost always better than DFS for finding '***whether the graph is connected or not'***.



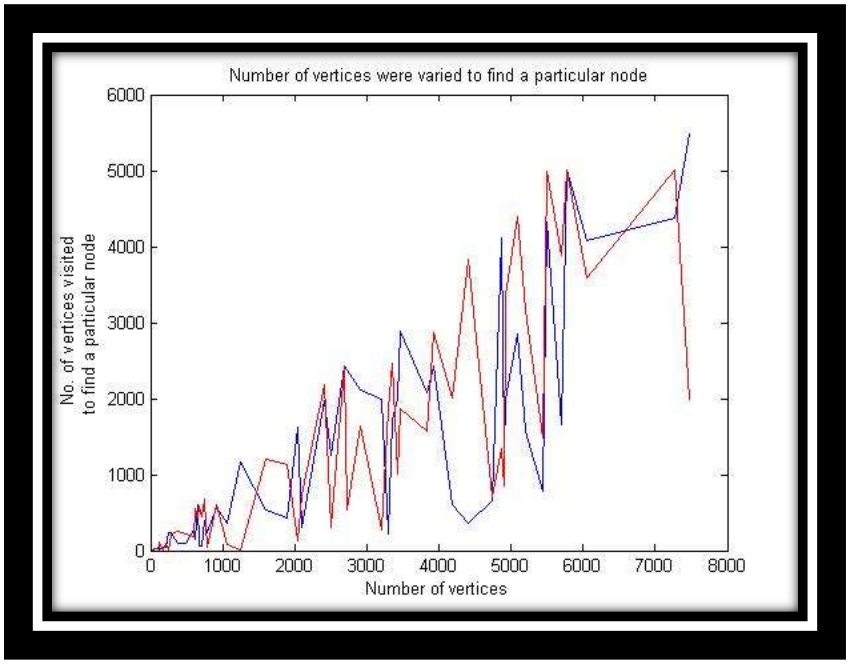
This shows how DFS and BFS reacts when edges are varied keeping the number of vertices fixed. BFS does perform better than DFS, however, we can also observe that when edges are almost equal to the number of vertices, DFS tend to perform a bit better.

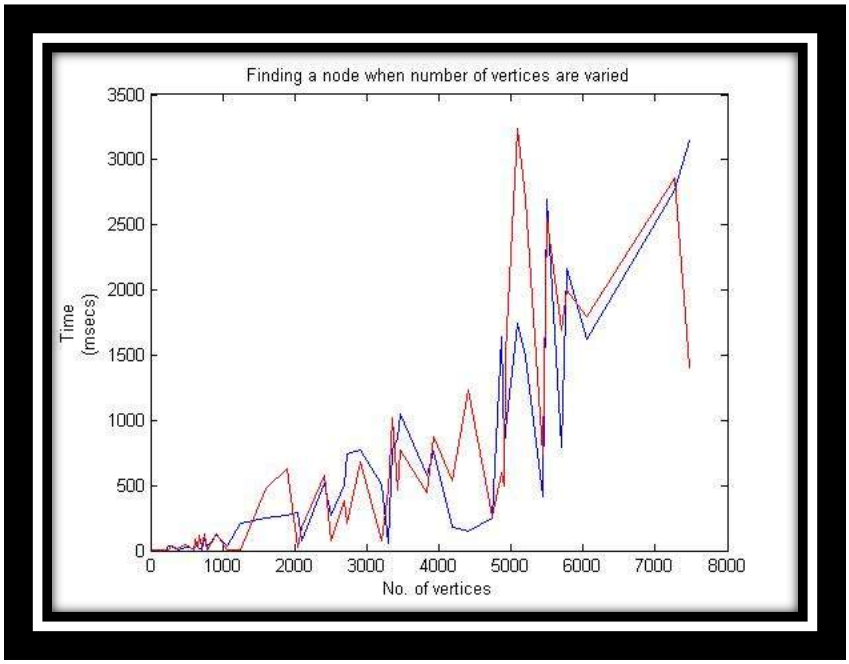


The graph above shows how vertices are varied keeping the number of edges to be constant. We can note that both DFS and BFS are very close to each other, in terms of performance time. We may also note that both strategies tend to decrease with increase number of vertices as edges are fixed. This observation is clear with our theoretical knowledge that since for a fixed number of edges, increasing the number of edges has no appreciable change to BFS or DFS.

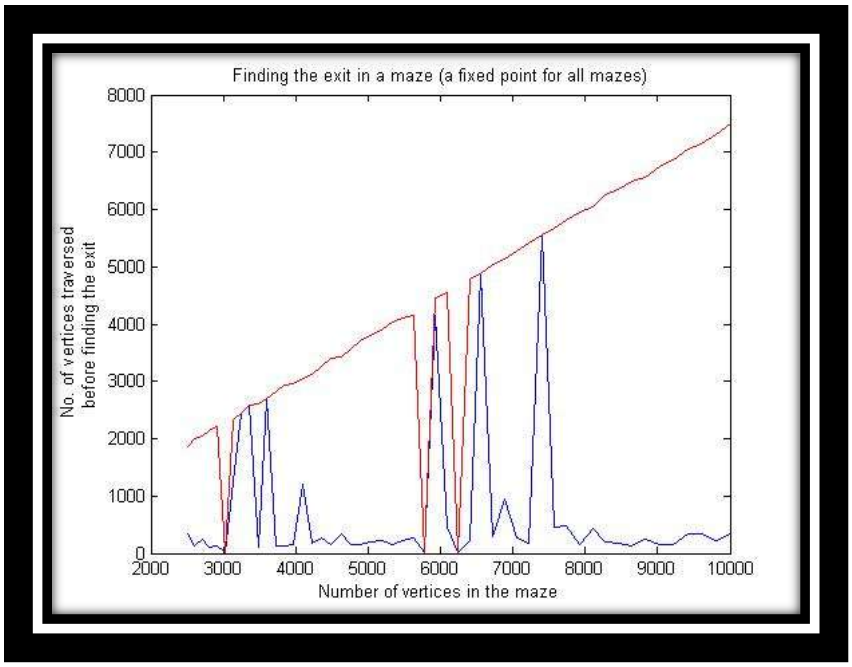


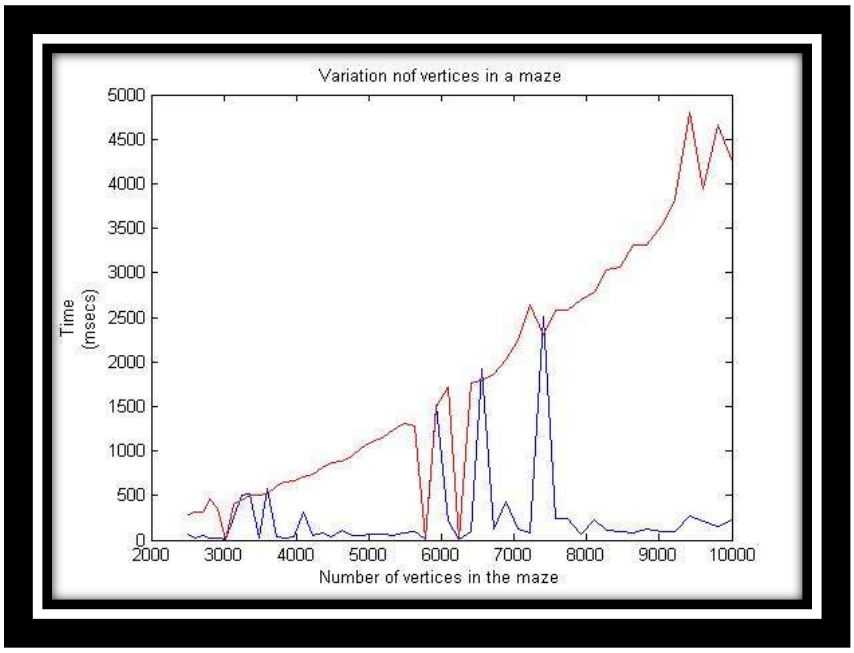
How both components of the Graph that is edge and vertex are changed, and so is the performance of the two strategies. It does show that by increasing both the components, BFS and DFS react in the same manner.



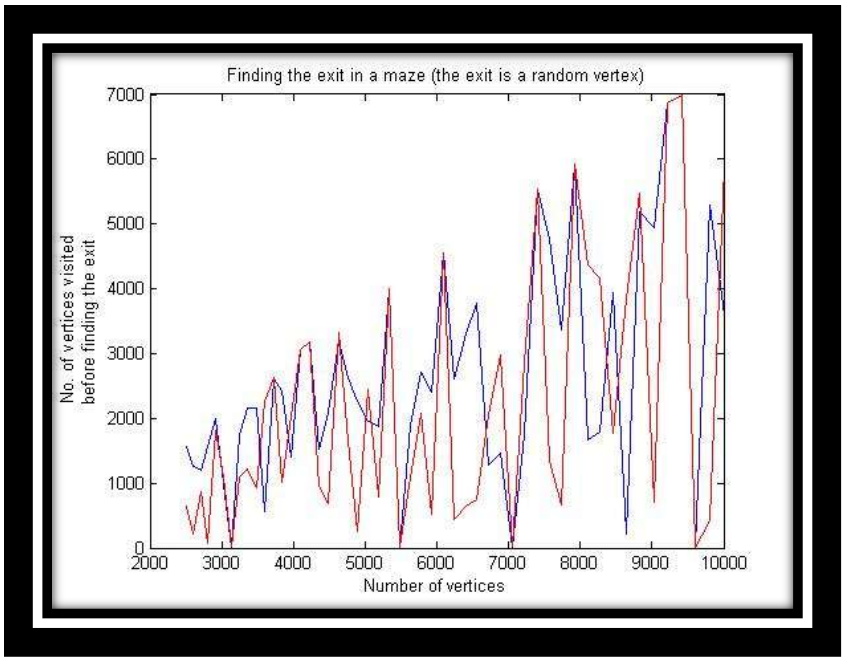


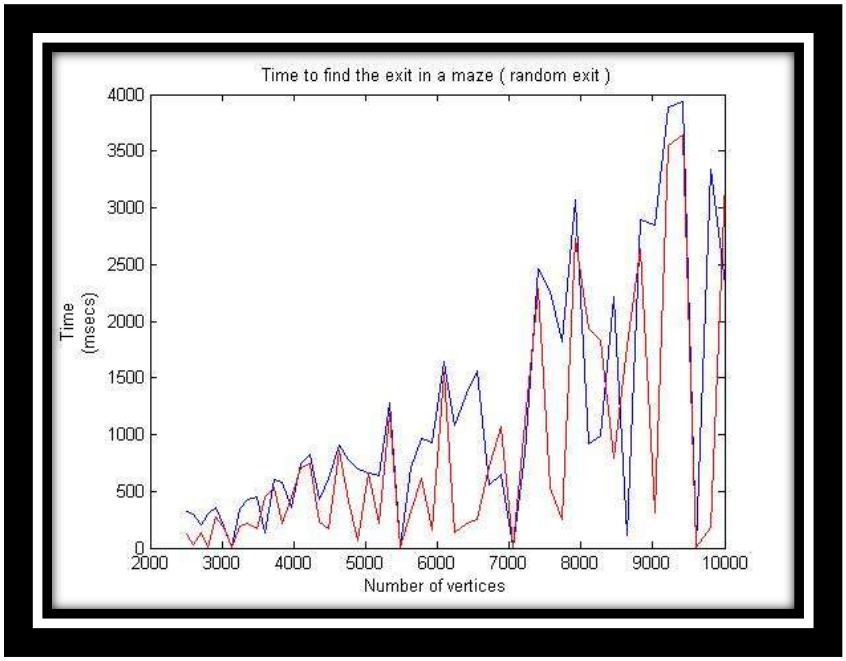
The above two graphs depicts the results when a particular node (or vertex) is to be found. This was simulated by varying the number of vertices and a random vertex was chosen to be found. The first graph shows the number of vertices that were traversed before the particular node was found. It shows that both DFS and BFS are very closely matched to each other. The second graph shows the execution time taken to find the particular node. This also shows how time and number of vertices graph show almost the same nature, showing that the execution is nearly perfect free from ***noise***.





The classic problem of traversal through a maze has been simulated here. The first graph shows how the no. of vertices is traversed before they can get out of the maze. Although, the author and the audience is aware of the fact that DFS works better for ***'traversal through a maze'***. However we can observe that in both the graphs, the BFS algorithm performs very poorly. And it actually grows linearly with increase number of vertices. Since this simulation was for a fixe point Exit which was basically situated at the last cell of the maze, it is evident that DFS would traverse much faster than BFS which in fact was traversing the entire graph/maze.





The classic problem of traversal through a maze is being re-visited. The first graph shows how the no. of vertices is traversed before they can get out of the maze. However this time we can observe that in both the graphs, since now the exit point has been randomly fixed and not the last cell, the BFS algorithm does not perform as poorly as in the previous graph. In fact it actually performs better in some cases than DFS, showing that choosing the exit point is crucial to chose which strategy is better.

12. Conclusion & Future Work

From the results and analysis, we clearly see that there is no clear winner in terms of performance across all applications. Our hypothesis started by stating that BFS and DFS are individually better than each other for certain specific applications.

We thus conclude by stating that, ***It is indeed true that DFS and BFS are better than each other for certain applications.***

For the future, it remains to be seen whether novel algorithms which may use hybrid techniques and may outperform BFS and DFS individually.