Scientia Africana, Vol. 14 (No.2), December 2015. Pp 82-98
© College of Natural and Applied Sciences, University of Port Harcourt, Printed in Nigeria *ISSN 1118 – 1931*

AN EMPIRICAL TIME COMPLEXITY ANALYSIS FOR A CLASS OF GRAPH ROUTE PROBLEMS

U. A. Okengwu[1] and M. O. Musa[2]

[1]*Department of Computer Science,*
University of Port Harcourt, Port Harcourt, Nigeria
Email: ugodepaker@yahoo.com
[2]*Department of Computer Science,*
University of Port Harcourt, Port Harcourt, Nigeria
Email: marthaozy@yahoo.com

Received: 10-07-15
Accepted: 14-10-15

ABSTRACT

In this paper, we investigate Empirical Time Complexity of a Modified Dijkstra algorithm for the multi-source to multi-destination problem. We use the Netbeans profiler tool in Java language to perform several runs on an experimental data-set derived with GPS and processed with Arc-Map Software. From the means plot of the runs, it was observed that the model has an experimental linear time complexity with a slightly better running time than Standard Dijkstra Algorithm.

Key-words: Empirical Time Complexity, Modified Dijkstra-Algorithm, Multi-Source to Multi-Destination

INTRODUCTION

Shortest-path algorithms are a class of computational intensive routines that basically seek to minimize the path to reach a destination from a given source point. One of the most popular algorithms for such class of problems is the Dijkstra Algorithm of which several variants exist. Some of these algorithms are dependent on the cardinality of the input-output for which we have the Multi-pairs Wang et al (2005), and some recent hierarchical approaches (Geisberger et al, 2010), Schultes et al (2008), Okengwu et al (2015). These approaches seek to minimize the cost of path discovery in very large datasets. However, very little studies have been done in the area of empirical time complexity of a Modified Dijkstra Algorithm for Multi-Sources and Multi-Destinations. In this paper, ETC studies will be performed on a Modified Dijkstra Algorithm developed specifically for Multi-Source to Multi-Destination problem. The aim of this research is to carry out an Empirical Time Complexity (ETC) analysis on a modified version of Dijkstra's Algorithm and verify if there are slight improvements over the standard algorithm. The study shall focus on a Multiple-Source to Multiple-Destination shortest path problem using a real time data set.

MATERIALS AND METHODS

Dijkstra algorithm shown in algorithm 1 below was conceived by computer scientist Edsger Dijkstra in 1956 and published in 1959, is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree. This Algorithm is often used in routing and as subroutine in other graph algorithm.

The Dijkstra Algoritm

```
function Dijkstra (Graph, source):
    for each vertex v in Graph:          // Initializations
        dist[v]:= infinity;              // Unknown distance function from
                                         // source to v
        previous[v]:= undefined;         // Previous node in optimal path
    end for                              // from source

    dist[source] := 0 ;                  // Distance from source to source
    Q: = the set of all nodes in Graph;  // All nodes in the graph are
                                         // unoptimized – thus are in Q
  while Q is not empty:                  // The main loop
    u: = vertex in Q with smallest distance in dist[] ; // Source node in first case
      remove u from Q ;
      if dist[u] = infinity:
         break ;                         // all remaining vertices are
      end if                             // inaccessible from source
    for each neighbor v of u: // where v has not yet been removed from Q
          alt := dist[u] + dist_between(u, v) ;
          if alt < dist[v]:              // Relax (u,v,a)
              dist[v] := alt ;
              previous[v] := u ;
              decrease-key v in Q;       // Reorder v in the Queue
          end if
    end for
  end while
 return dist;
end function
```

Algorithm 1: *Dijsktra Algorithm (Source: Melissa, 2012)*

For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path first is widely used in network routing protocols.

In the algorithm 1 above, the code u: = vertex in Q with smallest dist[], searches for the vertex u in the vertex set Q that has the least dist[u] value. That vertex is removed from the priority queue Q and returned to the user. dist_between (u, v) calculates the length between the two neighbour-

Scientia Africana, Vol. 14 (No.2), December 2015. Pp 82-98
© College of Natural and Applied Sciences, University of Port Harcourt, Printed in Nigeria *ISSN 1118 – 1931*

nodes *u* and *v*. The variable *alt* on lines 18 & 20 is the length of the path from the root node to the neighbour node *v* if it were to go through *u*. If this path is shorter than the current shortest path recorded for *v*, that current path is replaced with this *alt* path. The previous array is populated with a pointer to the "next-hop" node on the source graph to get the shortest route to the source.

Suppose you would like to find the shortest path between two intersections on a city map, a starting point and a destination. The order is conceptually simple: to start, mark the distance to every intersection on the map with infinity. This is done not to imply there is an infinite distance, but to note that intersection has not yet been *visited*; some variants of this method simply leave the intersection unlabeled. Now at each iteration, select a *current* intersection. For the first iteration, the current intersection will be the starting point and the distance to it (the intersection's label) will be zero. For subsequent iterations (after the first), the current intersection will be the closest unvisited intersection to the starting point—this will be easy to find.

From the current intersection, update the distance to every unvisited intersection that is directly connected to it. This is done by determining the sum of the distance between an unvisited intersection and the value of the current intersection, and relabeling the unvisited intersection with this value if it is less than its current value. In effect, the intersection is relabeled if the path to it through the current intersection is shorter than the previously known paths. To facilitate shortest path identification, in pencil, mark the road with an arrow pointing to the relabeled intersection if you label or relabel it, and erase all others pointing to it. After you have updated the distances to each neighboring intersection, mark the current intersection as *visited* and select the unvisited intersection with lowest distance (from the starting point) – or the lowest label—as the current intersection. Nodes marked as visited are labeled with the shortest path from the starting point to it and will not be revisited or returned to. Continue this process of updating the neighboring intersections with the shortest distances, then marking the current intersection as visited and moving onto the closest unvisited intersection until you have marked the destination as visited. Once you have marked the destination as visited (as is the case with any visited intersection) you have determined the shortest path to it, from the starting point, and can trace your way back, following the arrows in reverse. Note that this algorithm makes no attempt to direct "exploration" towards the destination as one might expect. Rather, the sole consideration in determining the next "current" intersection is its distance from the starting point. This algorithm, therefore "expands outward" from the starting point, interactively considering every node that is closer in terms of shortest path distance until it reaches the destination. When understood in this way, it is clear how the algorithm necessarily finds the shortest path; however, it may also reveal one of the algorithm's weaknesses: its relative slowness in some topologies.

The Modified Dijkstra Algorithm:

PROCEDURE Parallel-Dijkstra *Graph, Source-Vertices*

//M^n:= all possible Queues holding the set of all nodes (V) in the Graph (G) ∉ Source-Vertices

For each Vertex (V) in the Graph (G) ∉ Source -Vertices (SV);

//V: = Hospitals in Rivers State, SV: = Communities, Towns all in Rivers State

Dist [V^n]:= ∞; ***// Unknown distance for all non- source vertices in the graph***
Previous [V^n]: = Undefined;
End for
IN PARALLEL DO
BEGIN
Dist [SV^n] := 0 ***// No previous nodes with optimal path from SV^n***
M^n: = All Queues holding the set of all nodes (V) in the Graph (G) ∉ Source-Vertices (SV^n)
While $M^n \neq$ **empty:**
Y^n:= Vertices in M^n with Shortest distance in Dist [V];
Remove Y^n from M^n;
If Dist[Y^n]:=∞//***Unknown distance from neighboring nodes of Y^n in the Graph;***
Break;
End if;
For each neighboring V of Y^n with Dist [V^1,V^2,V^3…V^n]
K^n:=neighboring Vertices (V) of Y^n
K^n:= Dist_between [SV^n,Y^n] + Dist_between [Y,K]
If K^n[Dist]< Dist[V^1,V^2,V^3…V^n]
Dist[V^n]:=K^n

Previous (V^n):=Y^n

Decrease-Key Y^n in M^n
While M^n = **empty**
End if
End For
End while
END

Algorithm 2: Modified Dijkstra Algorithm

The Modified Dijkstra Algorithm illustrated in Algorithm 2 above handles multiple-source to multiple-destination starts with initialization of parameters, where M^n represents queues of all nodes. Source vertices are specified as SV and other vertices as V. Dist [V] represents hospital locations which are initialized as infinity (because distance to a particular hospital is not known yet. Previous [V] is undefined, the communities are represented as SV^n, Dist SV^n=0 shows the multiple source our algorithm is addressing. Dist SV^n is assigned zero because no optimal paths have been previously assigned to those communities.

IN PARALLEL DO will concurrently initiate separate threads that will handle each queue in M^n. M^n refers to all possible queues holding set of all nodes in the graph represented by V.

While M^n is not empty means that as long as all the queues in M^n still have nodes to consider then select Y^n, which should be the shortest paths from communities to hospitals. Then remove Y^n from M^n, since they have been identified as shortest paths. If the Dist(Y^n) assigned infinity while considering neighbouring nodes that are inaccessible, the algorithm breaks. That is, the system will keep searching until it finds out that, there is no unknown distance from neighboring nodes that exists. The first shortest nodes (Y^n) have neighbouring nodes V that have not been removed from M^n. K^n are neighbouring vertices V^n to the first nearest nodes Y^n. The value of K^n is the distance of the first shortest path plus the distance from Y^n to the new vertice K for the queues in M^n. If K^n is less than V^n then remove K^n from M^n, then reorder Y^n in the queues. While M^n is empty, the search ends.

Time Complexity Analysis using BIG-O Notation

In the time complexity analysis of an Algorithm, the Big-O notation, a theoretical time complexity notation is routinely employed. This may be in the best case, worst case or average case. In every instance where the variable "n" is used it represents the size of the operation's input.

Scientia Africana, Vol. 14 (No.2), December 2015. Pp 82-98
© College of Natural and Applied Sciences, University of Port Harcourt, Printed in Nigeria *ISSN 1118 – 1931*

Table 1 shows illustrations of BIG-O notations. A useful guide may also be obtained from [Big-O Cheat sheet, 200].

***Table 1**: Illustrations of Big-O notations*

S/N	Specific Command Operation	Big-O Notation	Sample Operations
1	Constant-Time Operation	O (1)	Variable declarations, if-else, while
2	Linear Operation	O (n)	Single for-loop
3	Quadratic Operation	O (n^2)	Single for-loop with one inner for-loop, bubble sort, quick sort (worst case).
4	Log Operation	O (logn)	Input operations cut-down by half or a factor during loop processing. E.g. by using break, if-else command and probably some stopping/ and or an alternate logical transfer criterion.
5	Log-linear Operation	O (n*logn)	Single loops cut down by n* (half or a factor of) the operations e.g. as in Heap sort functions.

However, one major drawback of Big-O notation is the manual restriction in its interpretation. Also, it is not entirely clear how well this translates into practice as there might be some algorithmic dependent factors not seen in the algorithm but will appear in the simulations. Thus, there is yet to be a universally agreed approach to the theoretical approach. An alternative to Big-O is the Empirical Time Complexity (ETC) which is gradually gaining grounds with the improvement in processing power of modern PC's.

The materials employed include the hardware, software part for running the simulations and the test field data.

The Hardware part includes:

i. A Samsung R60 (GPS) plus PC
ii. Processor Speed – 1.5Ghz
iii. Ram – 2GB
iv. Hard disk- 100Gb

The Software tools used for Simulations study include:

i. Netbeans IDE for Java Code Development
ii. Microsoft Excel for further Graph Plots

Data Source and Data Collection

The test field data was obtained using GPS and processed with ArcMap GIS Tool, ArcMap (2015) using fly-distance measures. A sample map of this extraction process is shown in Figure 1. The distance matrix for four source communities to several target hospitals was obtained as shown in Table 2. In figure 1 below shows Graph transverse for Modified Dijkstra Algorithm, the evaluation of the hospital route (for instance, the route to the nearest hospital in cases of emergency) involves the assessment of the graphical network produced by the targeted hospital locations and communities where the hospitals are situated. An edge of the graph of the network represents the distance or road network between hospitals whereas the

vertices represent the target hospitals, all in Rivers State. The edge is weighted by the cost incurred from multiple sources to multiple destinations. Input to the algorithm is in the form of a digraph, D={V,E} where V represents the nodes (communities or hospitals) and E represents the edges (distance between the communities or hospitals).

The data for the study area collected through personal interviews, administration of questionnaires and field survey using GPS (Global positioning system). A GPS is a tool in GIS for data acquisition. It can acquire data on the location of hospitals in geographic x, y, and z co-ordinates. The GPS is a constellation of 27 NAVSTAR satellites orbiting the earth at a height of 12,600miles; it has 5 monitoring stations and individual receivers. By reading the radio signals broadcast from as few as three of these satellites simultaneously (a process known as triangulation), a receiver on earth can pin point its exact location on the ground. This location is expressed in latitude and longitude co-ordinates or UTM (Universal Traverse Mercator) as the case may be. Using the GPS, the exact location of hospitals in Port Harcourt city can be identified and mapped in a GIS environment like Arcview 3.2, ArcGIS 9.1, and ArcMap. Figure21below shows the fly distances from assumed points within Ogbunabali, Rumuogbai, Woji and Rumuomasi communities to hospitals in these areas, considering GPS coordinates of these hospitals acquired using hand held GPS of accuracy3-4meters. These hospital distances as shown in table 2 were derived by plotting the GPS coordinates using ArcMap Software to generate maps for the communities as shown in figure 2, figure 3, figure 4, figure 5 respectively.

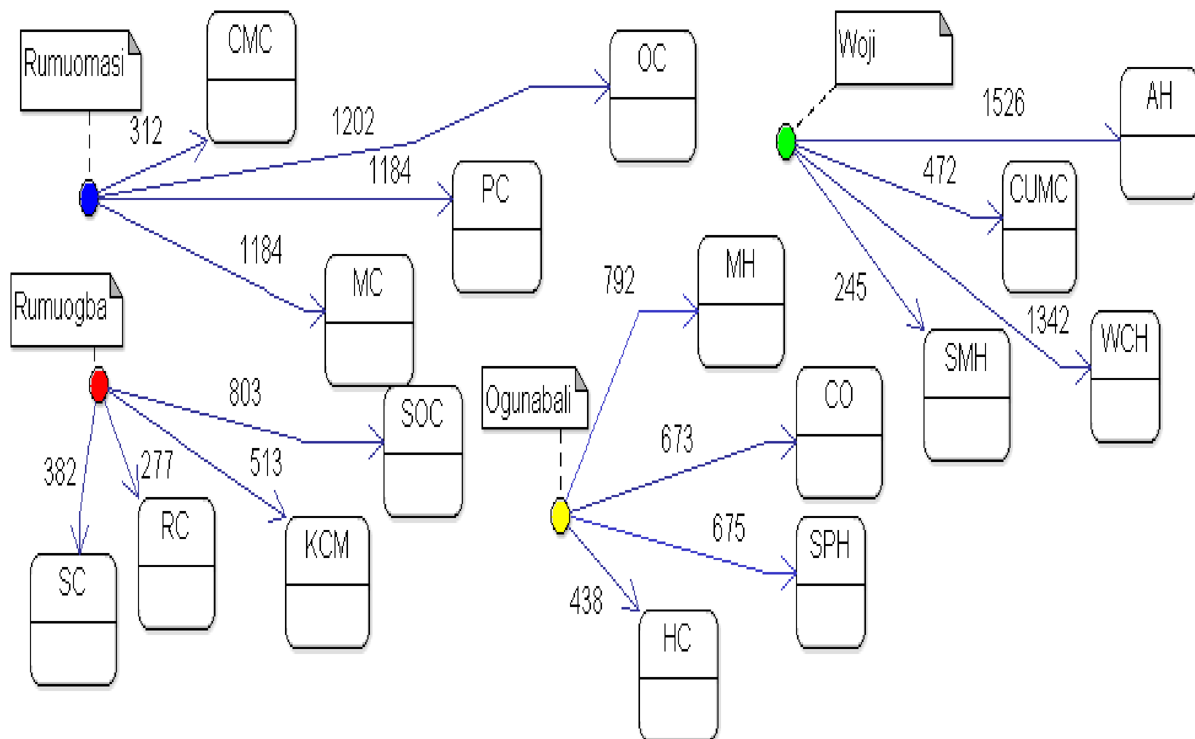


Fig 1: illustration of GraphTranverse from multiple-source to multiple-destination (shortest path)

Scientia Africana, Vol. 14 (No.2), December 2015. Pp 82-98
© College of Natural and Applied Sciences, University of Port Harcourt, Printed in Nigeria *ISSN 1118 – 1931*

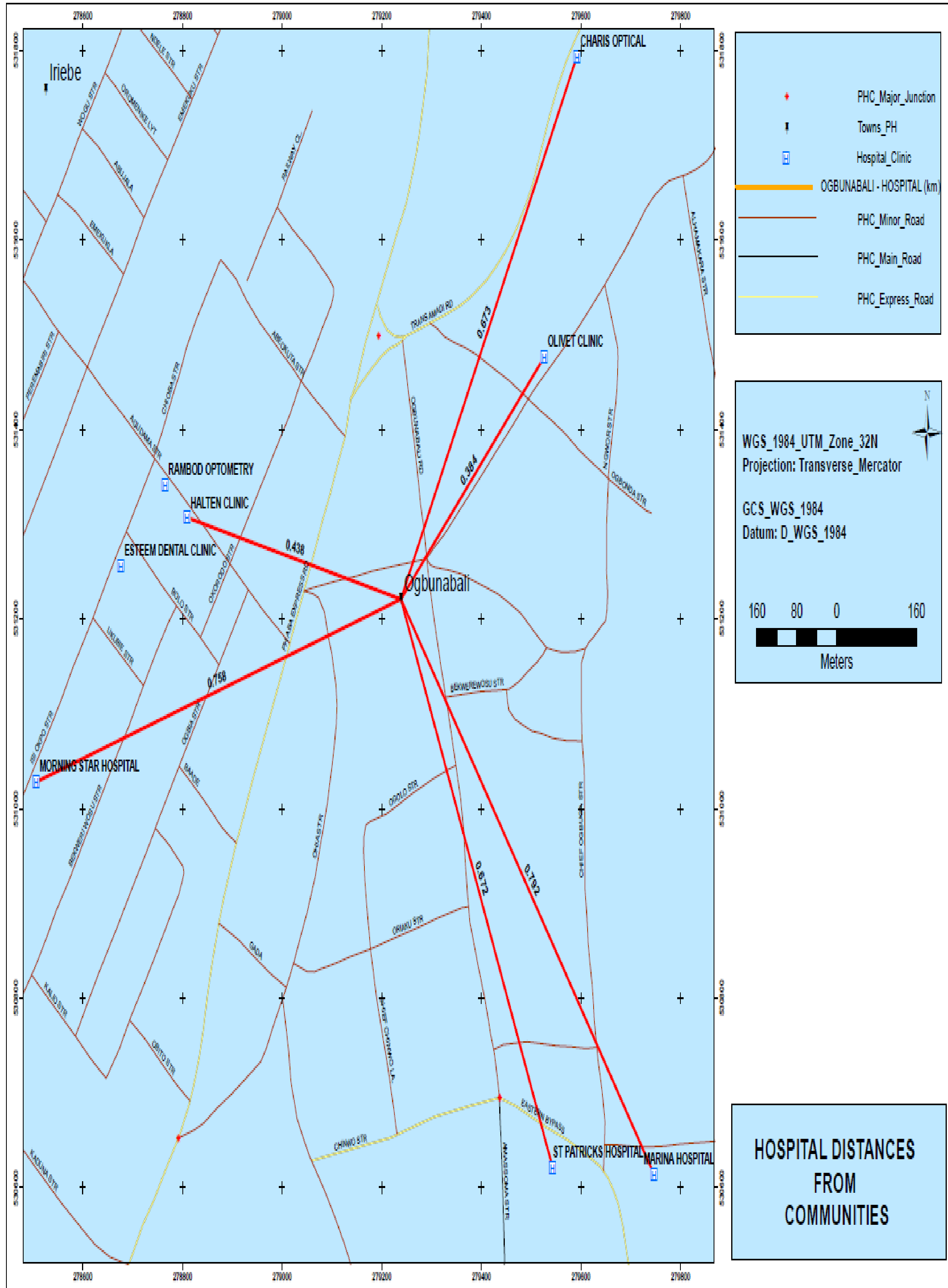


Fig 2: Fly distance of hospitals in Ogbunabali using ArcMap software

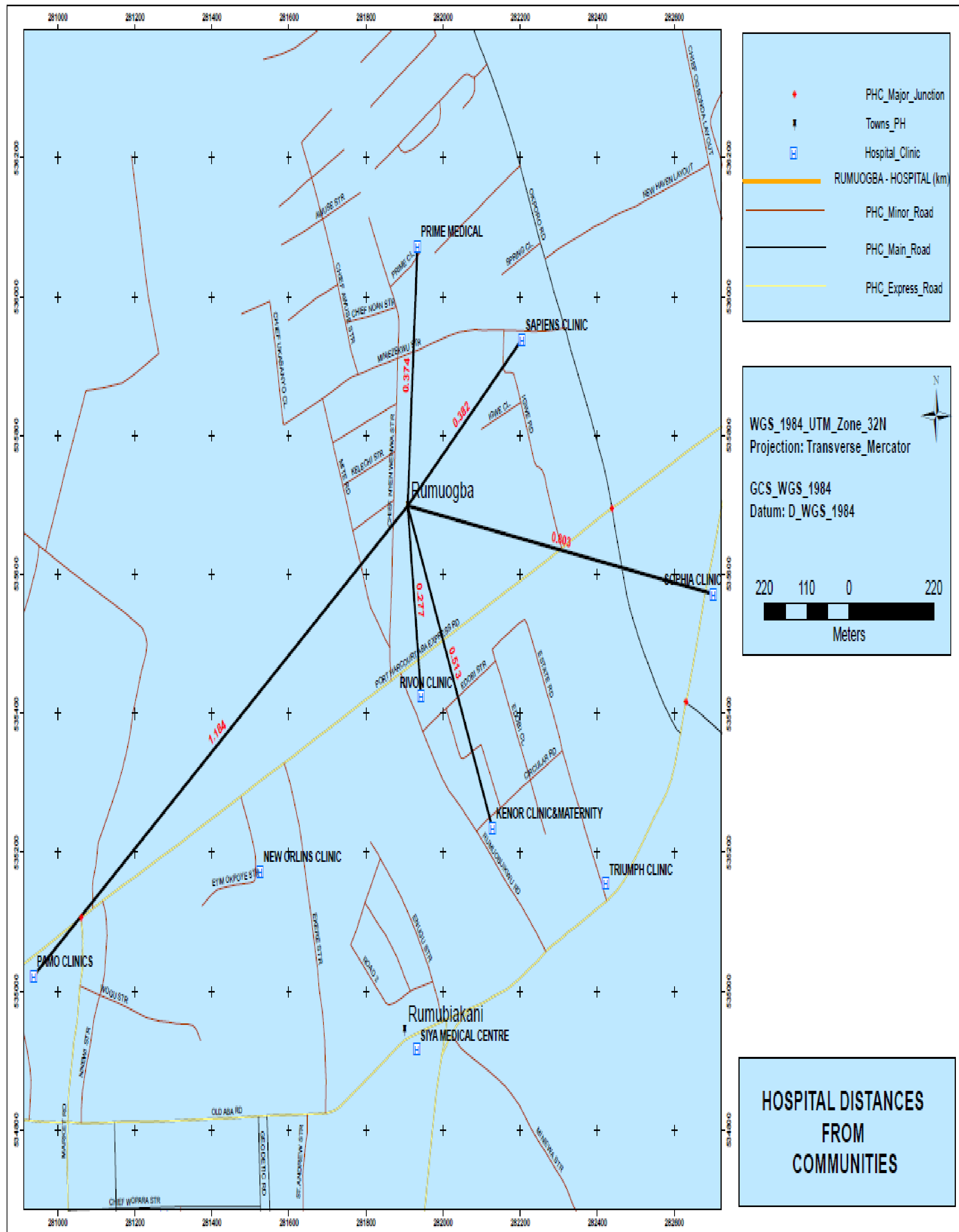


Fig 3: Fly distance of hospitals in Rumuogba using ArcMap software

Scientia Africana, Vol. 14 (No.2), December 2015. Pp 82-98
© College of Natural and Applied Sciences, University of Port Harcourt, Printed in Nigeria *ISSN 1118 – 1931*

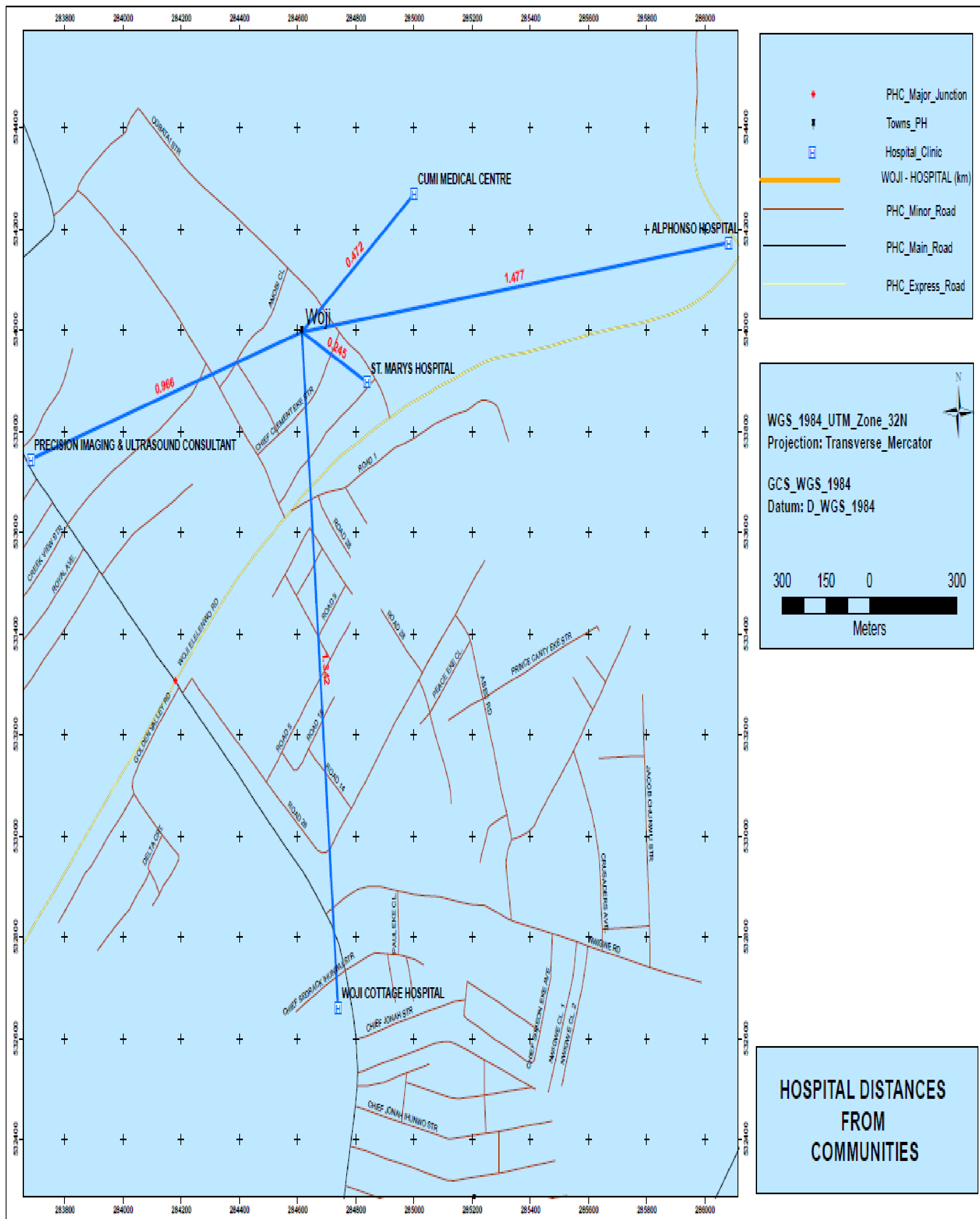


Fig 4: Fly distance of hospitals in Woji using ArcMap software

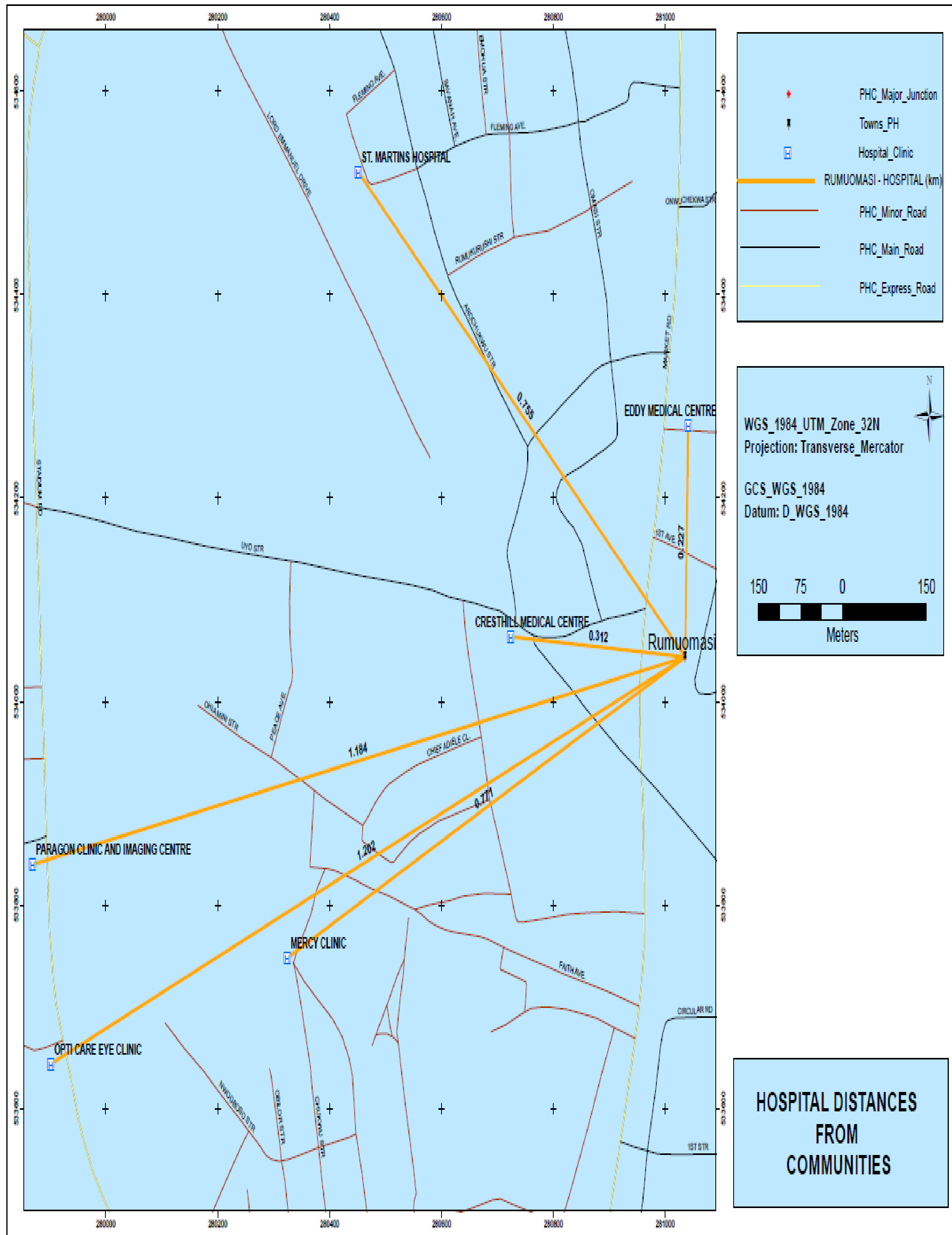


***Fig 5:** Fly distance of hospitals in Rumuomasi using ArcMap software*

Scientia Africana, Vol. 14 (No.2), December 2015. Pp 82-98
© College of Natural and Applied Sciences, University of Port Harcourt, Printed in Nigeria *ISSN 1118 – 1931*

Table 2: *Distance of Hospitals from centre of the communities*

S/N	Communities	Hospitals	Distance
1.	RUMUOMASI	eddy medical centre	0.227km
		cresthill medical centre	0.312km
		mercy clinic	0.771km
		paragon clinic and imaging centre	1.184km
		opticare eye clinic	1.202km
2.	RUMUOGBA	pamo clinics	0.374km
		sapiens clinic	0.382km
		rivon clinic	0.277km
		sophia clinic	0.803km
		kenor clinic and maternity	0.513km
		morning star hospital	0.392km
3	WOJI	precision imaging & ultrasound consultant	0.966km
		cumi medical centre	0.472km
		st. marys hospital	0.245km
		woji cottage hospital	1.342km
		alphonso hospital	1.526km
4	OGBUNABALI	olivet clinic	0.384km
		halten clinic	0.438km
		charis optical	0.673km
		st patricks hospital	0.675km
		marina hospital	0.792km

In order to characterize the time complexity effects in a Multi-Source to Multi-Destination environment, adequate methodologies need to be deployed. For an ETC simulation study, the following steps were performed:

i. Incremental modification of field data size in orders of 2 and this is done for 3-steps (see Appendix I)

ii. Time simulation runs using Modified Dijkstra program (see Appendix II). The simulations are done using the Netbeans Profiler Tool. For a total of 12 runs, the snapshot image is saved and recorded.

iii. In data collation stage, all running times are merged and tabulated for further analysis

iv. Mean computation of tabulated readings
v. Graphical Analysis of the tabulated readings

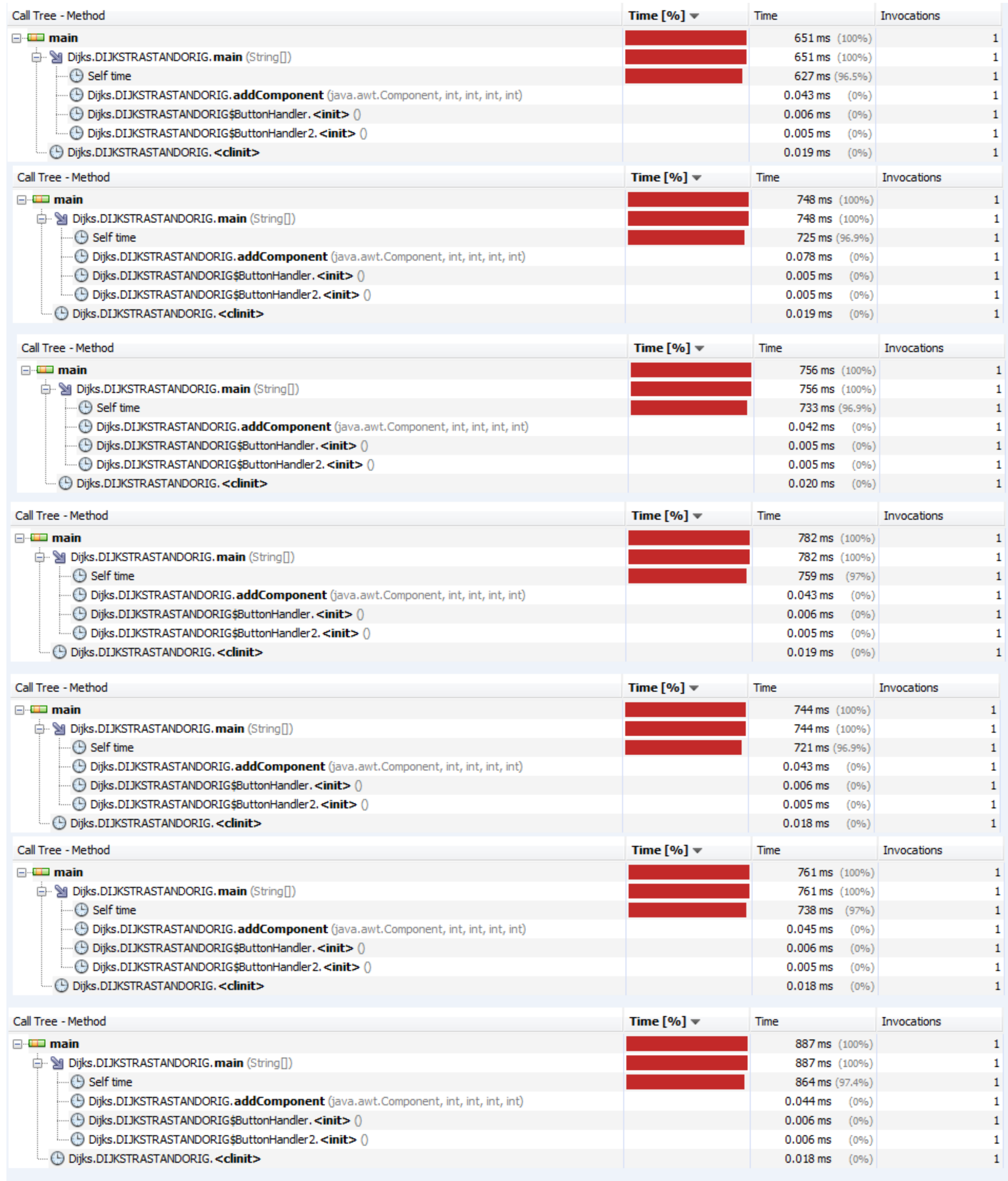
The software design procedure will typically include the following:

i. Launching the Netbeans IDE Interface
ii. Creating the Main Application Java Project and copying the program to space (See Appendix II for program listing)
iii. Creating required Inf.java class to handle distance updates (see Appendix III)
iv. Running the Program from the Java Projects Tree.

RESULTS

Table 3 below illustrates the time complexity analysis results of both Dijkstra Algorithm for Single-Source to Single-Destination (SS-SD) approach and Modified Dijkstra Algorithm for Multiple-Source to Multiple-Destination (MS-MD) approach using JAVA Netbeans Profiler. These two algorithms where subjected to the same data (fly distances of the hospitals from communities in Rivers State) in three Test Cases. The results show that it takes less time for Modified Dijkstra Algorithm for MS-MD to derive shortest paths from multiple-source than the time it takes Dijkstra Algorithm for SS-SD to derive shortest path from single-source. While figure 6 shows snapshots of the Netbeans Profiler Dijkstra Algorithm for Test Case 1.

Scientia Africana, Vol. 14 (No.2), December 2015. Pp 82-98
© College of Natural and Applied Sciences, University of Port Harcourt, Printed in Nigeria *ISSN 1118 – 1931*



Call Tree - Method	Time [%]	Time	Invocations
main		651 ms (100%)	1
Dijks.DIJKSTRASTANDORIG.**main** (String[])		651 ms (100%)	1
Self time		627 ms (96.5%)	1
Dijks.DIJKSTRASTANDORIG.**addComponent** (java.awt.Component, int, int, int, int)		0.043 ms (0%)	1
Dijks.DIJKSTRASTANDORIG$ButtonHandler.**<init>** ()		0.006 ms (0%)	1
Dijks.DIJKSTRASTANDORIG$ButtonHandler2.**<init>** ()		0.005 ms (0%)	1
Dijks.DIJKSTRASTANDORIG.**<clinit>**		0.019 ms (0%)	1

Call Tree - Method	Time [%]	Time	Invocations
main		748 ms (100%)	1
Dijks.DIJKSTRASTANDORIG.**main** (String[])		748 ms (100%)	1
Self time		725 ms (96.9%)	1
Dijks.DIJKSTRASTANDORIG.**addComponent** (java.awt.Component, int, int, int, int)		0.078 ms (0%)	1
Dijks.DIJKSTRASTANDORIG$ButtonHandler.**<init>** ()		0.005 ms (0%)	1
Dijks.DIJKSTRASTANDORIG$ButtonHandler2.**<init>** ()		0.005 ms (0%)	1
Dijks.DIJKSTRASTANDORIG.**<clinit>**		0.019 ms (0%)	1

Call Tree - Method	Time [%]	Time	Invocations
main		756 ms (100%)	1
Dijks.DIJKSTRASTANDORIG.**main** (String[])		756 ms (100%)	1
Self time		733 ms (96.9%)	1
Dijks.DIJKSTRASTANDORIG.**addComponent** (java.awt.Component, int, int, int, int)		0.042 ms (0%)	1
Dijks.DIJKSTRASTANDORIG$ButtonHandler.**<init>** ()		0.005 ms (0%)	1
Dijks.DIJKSTRASTANDORIG$ButtonHandler2.**<init>** ()		0.005 ms (0%)	1
Dijks.DIJKSTRASTANDORIG.**<clinit>**		0.020 ms (0%)	1

Call Tree - Method	Time [%]	Time	Invocations
main		782 ms (100%)	1
Dijks.DIJKSTRASTANDORIG.**main** (String[])		782 ms (100%)	1
Self time		759 ms (97%)	1
Dijks.DIJKSTRASTANDORIG.**addComponent** (java.awt.Component, int, int, int, int)		0.043 ms (0%)	1
Dijks.DIJKSTRASTANDORIG$ButtonHandler.**<init>** ()		0.006 ms (0%)	1
Dijks.DIJKSTRASTANDORIG$ButtonHandler2.**<init>** ()		0.005 ms (0%)	1
Dijks.DIJKSTRASTANDORIG.**<clinit>**		0.019 ms (0%)	1

Call Tree - Method	Time [%]	Time	Invocations
main		744 ms (100%)	1
Dijks.DIJKSTRASTANDORIG.**main** (String[])		744 ms (100%)	1
Self time		721 ms (96.9%)	1
Dijks.DIJKSTRASTANDORIG.**addComponent** (java.awt.Component, int, int, int, int)		0.043 ms (0%)	1
Dijks.DIJKSTRASTANDORIG$ButtonHandler.**<init>** ()		0.006 ms (0%)	1
Dijks.DIJKSTRASTANDORIG$ButtonHandler2.**<init>** ()		0.005 ms (0%)	1
Dijks.DIJKSTRASTANDORIG.**<clinit>**		0.018 ms (0%)	1

Call Tree - Method	Time [%]	Time	Invocations
main		761 ms (100%)	1
Dijks.DIJKSTRASTANDORIG.**main** (String[])		761 ms (100%)	1
Self time		738 ms (97%)	1
Dijks.DIJKSTRASTANDORIG.**addComponent** (java.awt.Component, int, int, int, int)		0.045 ms (0%)	1
Dijks.DIJKSTRASTANDORIG$ButtonHandler.**<init>** ()		0.006 ms (0%)	1
Dijks.DIJKSTRASTANDORIG$ButtonHandler2.**<init>** ()		0.005 ms (0%)	1
Dijks.DIJKSTRASTANDORIG.**<clinit>**		0.018 ms (0%)	1

Call Tree - Method	Time [%]	Time	Invocations
main		887 ms (100%)	1
Dijks.DIJKSTRASTANDORIG.**main** (String[])		887 ms (100%)	1
Self time		864 ms (97.4%)	1
Dijks.DIJKSTRASTANDORIG.**addComponent** (java.awt.Component, int, int, int, int)		0.044 ms (0%)	1
Dijks.DIJKSTRASTANDORIG$ButtonHandler.**<init>** ()		0.006 ms (0%)	1
Dijks.DIJKSTRASTANDORIG$ButtonHandler2.**<init>** ()		0.006 ms (0%)	1
Dijks.DIJKSTRASTANDORIG.**<clinit>**		0.018 ms (0%)	1

Fig 6: Snapshots of the Netbeans profiler DIJKSTRA ALGORITHM (**Test1**)

Table 3: *Results of run time as obtained from the Netbeans Profiler*

S/N	TEST1		TEST2		TEST3	
	MS-MD (ms)	SS-SD (ms)	MS-MD (ms)	SS-SD (ms)	MS-MD (ms)	SS-SD (ms)
1	723	719	984	986	962	986
2	646	630	995	997	1046	1011
3	717	724	955	959	1053	1082
4	731	712	970	990	954	979
5	718	611	983	979	984	998
6	760	651	974	978	999	1005
7	632	748	990	993	988	1000
8	726	756	983	989	1021	1025
9	770	887	986	997	1012	1018
10	645	782	999	999	1128	1165
11	644	744	952	994	1033	1037
12	775	761	1026	1079	992	1009
Meanx	707	729	983	995	1014	1026

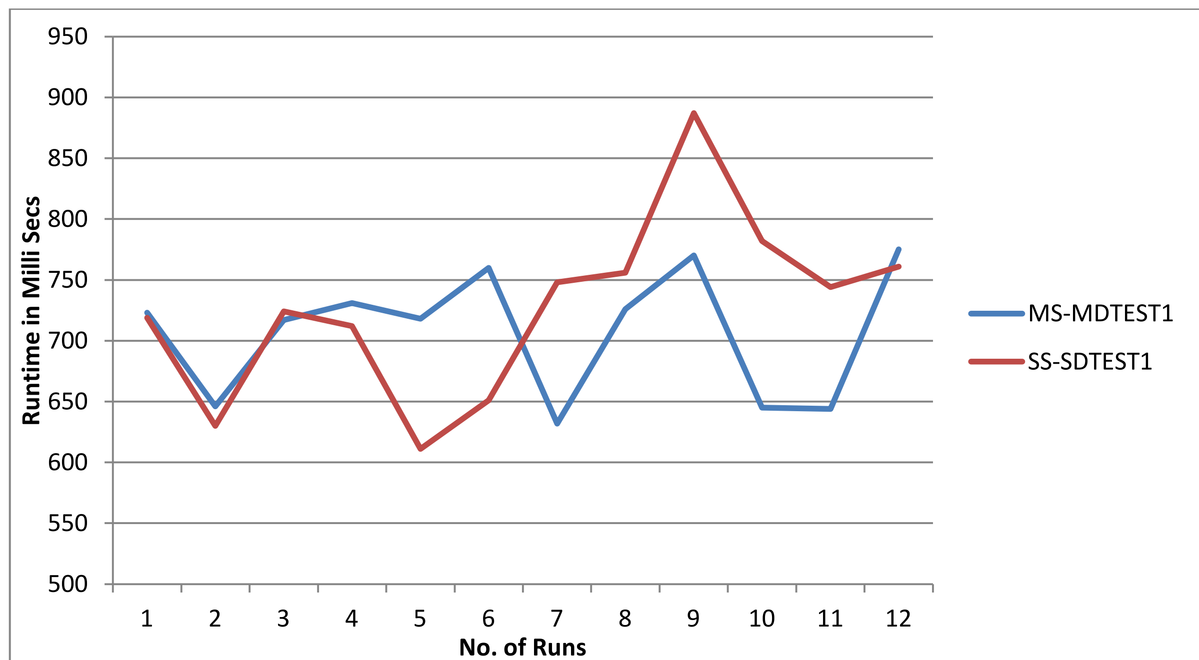


Fig 7: Graph showing results for MS-MD AND SS-SD Algorithms in Test 1

Scientia Africana, Vol. 14 (No.2), December 2015. Pp 82-98
© College of Natural and Applied Sciences, University of Port Harcourt, Printed in Nigeria *ISSN 1118 – 1931*

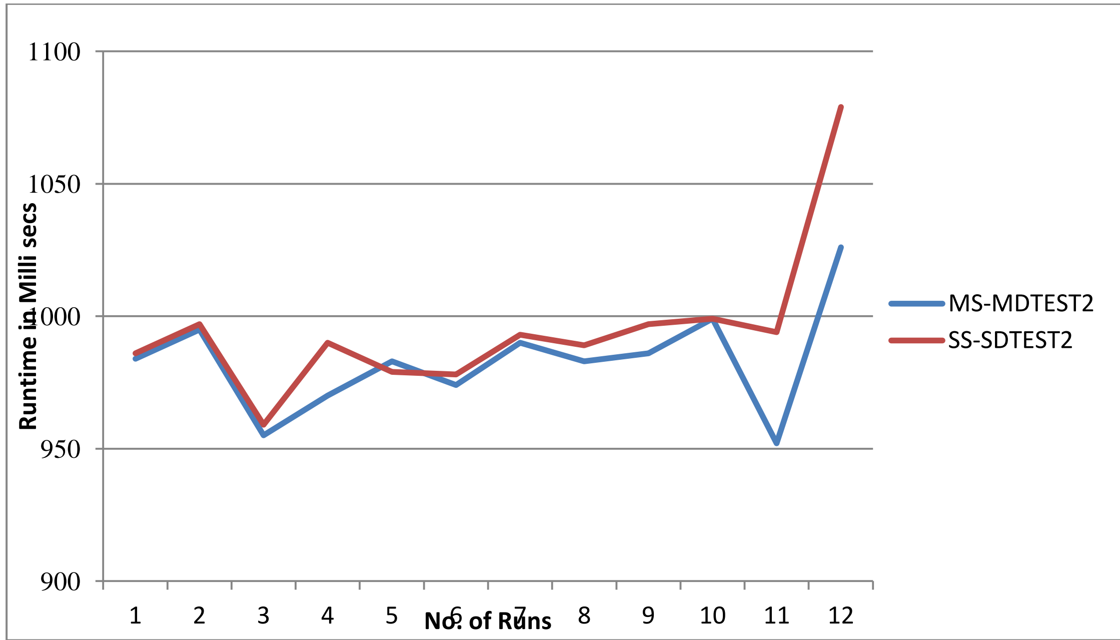


Fig 8: Graph showing results for MS-MD AND SS-SD Algorithims in Test 2

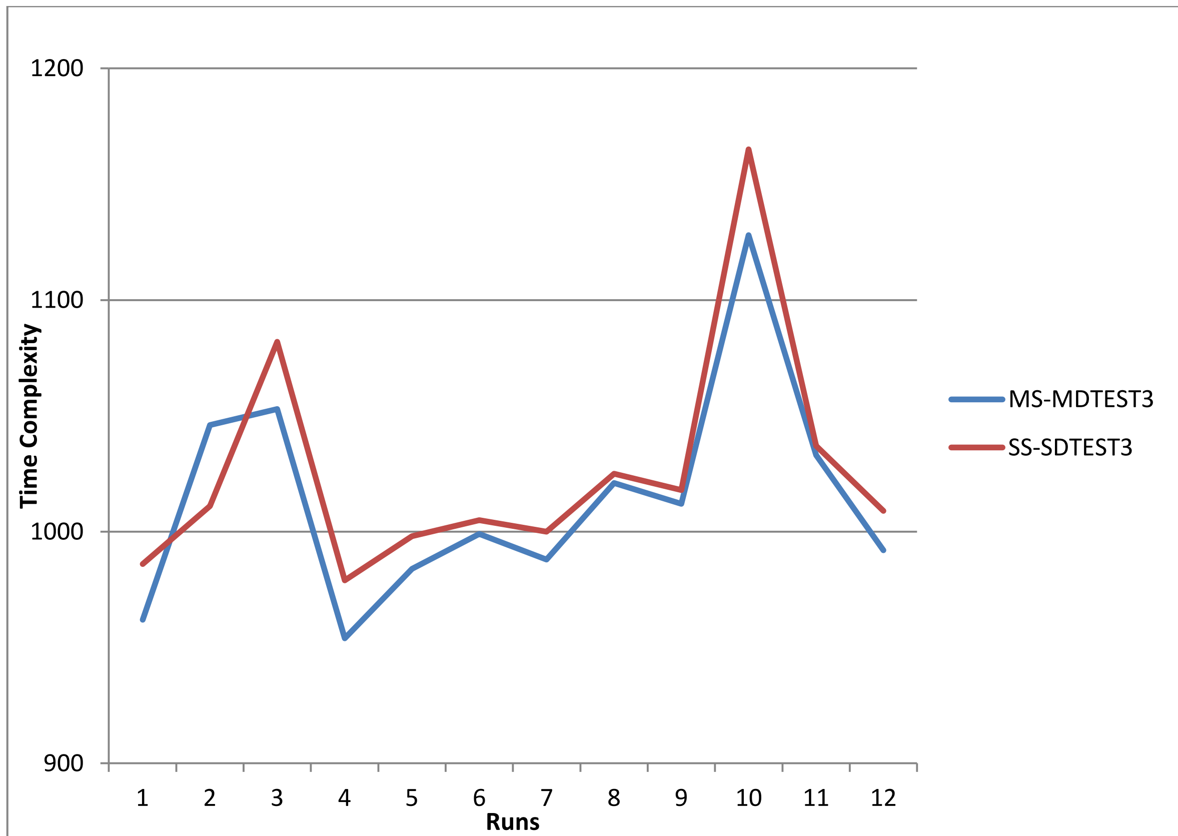


Fig 9: Graph showing results for MS-MD AND SS-SD Algorithims in Test 3

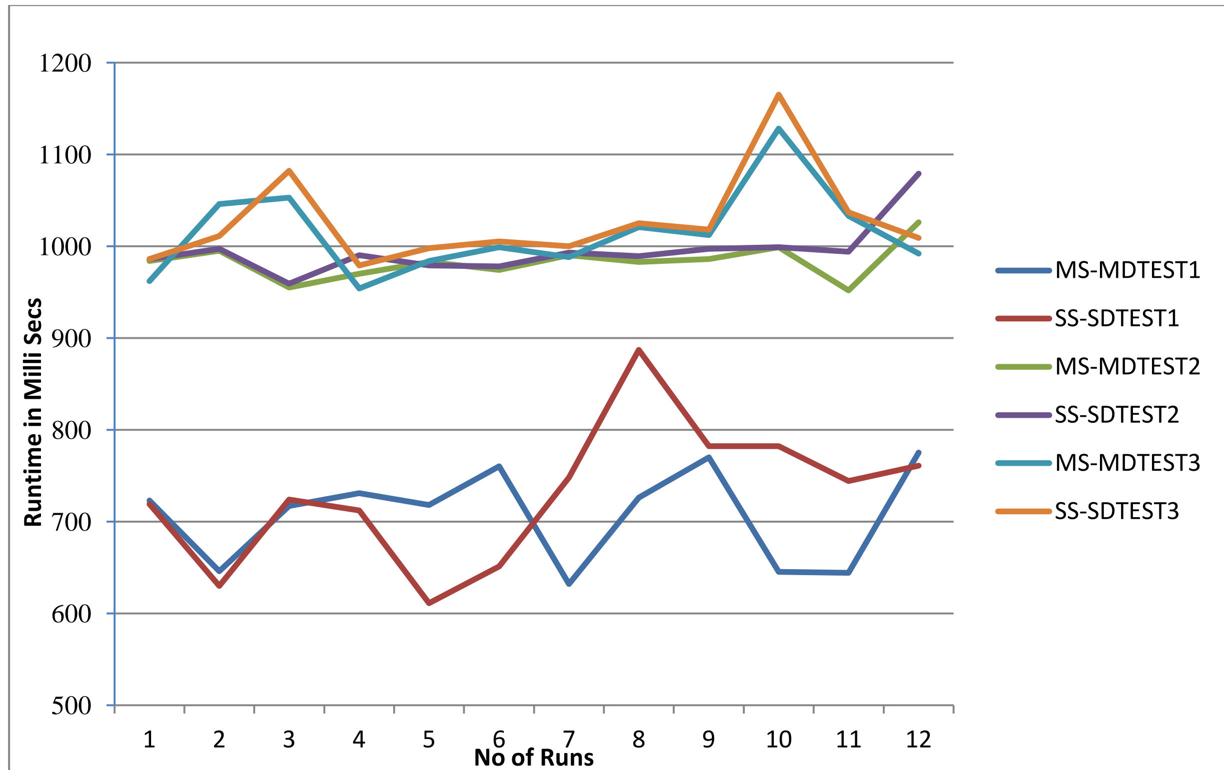


Fig10: Graph showing results from combined three Test cases

Figure 7 Test Case1 illustrates graphical representation of both algorithms time complexity results. Comparing the run time of Multiple-Source to Multiple-Destination and Single-Source to Single-Destination, we can infer that on the first iteration both algorithms start off with almost the same running time (MS-MD at 723ms) and (SS-SD at 719ms). As the number of iterations increases precisely at the fifth iteration, the SS-SD is shown to have the least running time after that, there is an increase in the running time of the SS-SD, conclusively the SS-SD running time is more computationally expensive relative to the MS-MD.

Figure 8 Test Case2 illustrates graphical representation of both algorithms time complexity results, it can be seen that at the initial iteration, the run time for the MS-MD is lower relative to the SS-SD but as the iteration progresses, the running time for the MS-MD becomes better compared with the SS-SD. conclusively the SS-SD running time is more computationally expensive relative to the MS-MD.

Figure 9 Test Case3 illustrates graphical representation of both algorithms time complexity results, it can be inferred that the MS-MD has a lower run time than the SS-SD which implies that the Modified Dijkstra Algorithm is more efficient compared to Dijkstra Algorithm when both algorithms are subjected to the same data set.

Combining the three Test cases, it is observed in figure 10 that MS-MD has the lowest running time while SS-SD has the highest running time, considering the fact that the MS-MD algorithm performs concurrent operations unlike the SS-SD algorithm that handles single operation at a time. This difference is clearly visible as the number of iterations increases. Conclusively, the MS-MD does not perform poorly given the fact that it performs more operations than the SS-SD algorithm.

Scientia Africana, Vol. 14 (No.2), December 2015. Pp 82-98
© College of Natural and Applied Sciences, University of Port Harcourt, Printed in Nigeria *ISSN 1118 – 1931*

DISCUSSION

Dijkstra algorithm is a useful shortest path optimisation routine when dealing with datasets that are small in size. However, the larger sizes of data, modifications of Dijkstra algorithm are necessary. This paper has performed empirical time complexity (ETC) studies using Netbeans Profiler Tool on a Modified Dijkstra Algorithm for multi-source to multi-destinations with better running times for the modified case. Thus, performance of Dijkstra Algorithm and modifications can be studied empirically. In future, we would like to automate ETC tasks for carrying out ETC studies on different versions of the Dijkstra algorithm. The shortest path problem is the problem of finding the paths in a graph for which a given traversal is a minimum. We also conclude that given any number of vertices N, the time complexity for the Modified Dijkstra Algorithm is $O(N^2)$ and time complexity for Dijkstra Algorithm is also $O(V^2)$ where V is the number of vertices. Hence the multiple-source to multiple-destination implementation of the Modified Dijkstra Algorithm yields the same time complexity as the single source to single destination implementation of Dijkstra Algorithm knowing that for Modified Dijkstra Algorithm more vertices are considered while for Dijkstra Algorithm considers one vertice.

REFERENCES

Wang, I.L., Johnson E.l., and Sokol J.S., (2005). A Multiple Pairs Shortest Path Algorithm. Journal of Transportation Science, Vol. 39, No.4, pp. 465-476.

Geisberger, R., Sanders, P., Schultes, D., and delling, D., (2008). Contraction Hierachies: faster and Simpler Hierarchical Routing in Road networks. In: 7th Workshop on Experimental Algorithms. Springer-Verlag, pp. 319-333.

Bast, H., Funke, S., Matijevic, D., Sanders, P., and Schultes, D., (2007, January). In Transit to Constant Time Shortest-Path Queries in Road Networks. In ALENEX.

OKENGWU, U. A., Nwachukwu E.O., and Osegi, E. N., (2015). Modified Dijkstra Algorithm with Invention Hierarchies Applied to a Conic Graph. arXiv preprint arXiv: 1503.02517.

Big-O Cheat sheet. Retrieved October 06, 2014 from http://bigocheatsheet.com

ArcGIS version 10.3 [Computer Software] (1999-2014). Redlands, CA: Enviromental Systems Research Institute.

NetBeans IDE version 7.1.2 [Computer Software] (2009-2010). Redwood City, CA: Oracle Incorporation.

Melissa Y. (2012), Shortest part Algorithm, retrieved from math.mit.edu/~roth, May 2013.