A Novel Solution for Simultaneously Finding the Shortest and Possible Paths in Complex Networks

Wai Mar Hlaing[1,2], Shi-Jian Liu[2], Jeng-Shyang Pan[2,3,4]
[1]Geographic Information System, University of Computer Studies Yangon, Myanmar
[2] Fujian Provincial Key Laboratory of Big Data Mining and Applications, Fujian University of Technology, China
[3] College of Computer Science and Engineering, Shandong University of Science and Technology, China
[4] Department of Information Management, Chaoyang University of Technology, Taiwan
wai_mar@yahoo.com, {liusj2003, jengshyangpan}@gmail.com

Abstract

A Novel graph approach named Combined Forward and Backward Heuristic Search (CFBHS) is proposed in this paper, which can be used to solve optimization problems in areas such as transportation and network routing. There are two major aspects distinct our method from the most cited ones. Firstly, though focuses on getting the shortest path in a graph when both source and destination are given, this work can also find other possible paths as outputs. Secondly, the proposed algorithm is a high-performance one, which is achieved by (1) reducing unnecessary nodes and edges to reach a target optimum based on dynamically calculated heuristic values and (2) finding the results by using the sub-division scheme instead of computing over the whole graph. Experiments are carried out for the complex road network of Yangon Region. The comparisons show that our algorithm is about 100 times faster than the bi-directional Dijkstra's algorithm. Besides, benefit from the heuristic forward and backward search, the proposed method can achieve very low time complexity, which is similar to the A*, but A* can only produce the shortest path. By contrast, the proposed algorithm is competent for finding not only the shortest but also many possible paths in complex road networks such as undirected graph and hypergraph networks.

Keywords: Shortest path algorithm, Bi-directional Dijkstra, Heuristic search, Distance based methods

1 Introduction

Geographic information system has been widely used in areas such as emergency services [1-2], transportation systems [3], route navigation systems [4] and life safety services [5-6]. For example, Dabhade et al. [7] presented the network analysis about the spatial data for finding shortest path in hospital information systems by using the Dijkstra's algorithm. Parmar and Trivedi [8] combined bi-directed search and shortest path bounding box to reduce the time complexity while computing the shortest paths post calculating congestion levels at different traffic junctions. Tomaszewski [9] applied the GIS software for rendering the data set on the map and achieved the disaster management and mitigation. Finding the shortest path within a few seconds is always an encountered problem in our daily lives. Given a Graph *G*, which can be created using geospatial data in transportation network and contains node set $V=\{v_1, v_2, v_3, \ldots, v_n\}$ and edge set $E=\{e_1, e_2, e_3, \ldots, e_n\}$ such that $|V|=n$ and $|E|=m$. After the graph is created, one general problem is to find the shortest path between two pre-specified nodes: a source (s) and a target (t). The path also contains other nodes that belong to *V*. Many searching techniques can be used for finding the route, but the efficiency is still a problem facing until now in real-time applications.

Most of the shortest path algorithms are based on Dijkstra's algorithm proposed in 1959 [10]. However, as we know, the complexity of Dijkstra's Algorithm is $O(n^2+m)$ if linear search is used, where n and m means the number of vertices and edges respectively. Various data structures such as fibonacci heap, fusion trees and priority queues [11] were adopted by researchers in order to improve the efficiency of the Dijkstra's Algorithm. Ahuja et al. [12] improved the efficiency to $O(m+n\log C)$, where C is the cost of the weightiest edge. Haung et al. [13] introduced the algorithm that include constraints to search the expected direction and the weighted value is flexibly changed to adapt to different network complexity. By introducing constraint function, it can omit lots of useless search path and accelerate the process of problem solving. Although the theoretical developments with various data structures and constraints based on Dijkstra's algorithm have been made, researchers are still trying to achieve better efficiency. Another approach is to search the shortest

path forward from the source and backward from the target in bi-directional simultaneously. This kind of algorithm works in $O(m)$ in the worst case and it takes in less time for path calculation, where m is the number of edges [14-15]. It is more efficient than the previous Dijkstra's algorithm, because the Dijkstra based algorithms traverses the whole graph to find the shortest path. By contrast, partitioning techniques and heuristic search [16] are applied to get the shortest path with low time complexity by removing the unnecessary data space. K-d tree is a popular data mining technique for graph partitioning [17]. Graph partitioning technique performs to find the best partitions, which close to reach the goal by using good heuristics at the pre-processing stage. However, it needs to consider the cost of pre-processing time which dominates the running time of the shortest-path computations [18].

The most popular heuristic search approach is the A* algorithm [19-21]. At each iteration of its main loop, A* needs to determine which partial paths should be expanded. It does so based on the estimation of the costs (total weight) of different paths travelling from current node to the goal node. Specifically, A* selects the path that minimizes $f(n) = g(n) + h(n)$ where n is the last node on the path, $g(n)$ is the cost of the path from the start node to n and $n(n)$ is a heuristic that estimates the cost of the cheapest path from n to the goal. Goyal et al. [22] compared the running time between A* and Dijkstra's algorithm, it is proved that the heuristic search can greatly improve the performance in terms of time complexity comparing to the linear search. However, A* considers all nodes in the open lists repeatedly.

We refer to the proposed method as a Combined Forward and Backward Heuristic Search (CFBHS) algorithm for it works forward and backward in sequential and it is based on the heuristic search like the A*. However, it differs from the A* in that the proposed method does not consider the comparison of many adjacent nodes repeatedly and it estimated values will be dynamically computed for heuristic search (see Section 3.2). And main contributions of this work includes:

- A dynamic tree-subdividing scheme is proposed for a complex network other than the conventional graph partitioning approach. This scheme is simple and easy to find the shortest path for a large network.
- A new heuristic values calculation method is adopted to avoid the pre-storing of the heuristic values for every nodes in a graph. This method calculates heuristic values dynamically and finds the appropriate node for the next level.
- A new heuristic search algorithm is proposed to prohibit about considering the nodes in the open list repeatedly and it also intends to find both shortest and possible paths simultaneously.

The rest of this paper is organized as follows. Section 2 describes Bi-directional Dijkstra's and Heuristic search algorithm. The newly proposed shortest path approach (i.e., the CFBHS algorithm) will be introduced in Section 3. Section 4 shows the simulation results and statistical analysis. Section 5 depicts the performance comparison of algorithms using different algorithm analysis approaches. Section 6 concludes the paper.

2 Problem Statement and Preliminaries

In this Section, as related works, two most cited path finding algorithms, namely the Bi-directional Dijkstra's and the A*, are introduced.

2.1 Bi-directional Dijkstra's Algorithm

This algorithm is fundamentally an extension of the Dijkstra's algorithm and it is also a popular technique for accomplishment the shortest path. The two distinct persons work a task in searching a path between source and destination on the road network. Person A starts from the source and checking the condition of Person B. Person B also starts from destination and goes in backward direction and looking for the current states of Person A. At a time, Person A and B will meet at the same point. Then they interchange the information such as the path they have traversed and store the result by combing them in the route database. After traversing all nodes in the graph, the algorithm will find the shortest path and find the minimum. Bi-directional Dijkstra's algorithm can reduce the time complexity than the popular Dijkstra's algorithm due to the working in both directions but it still traverses all nodes that exists in the whole graph while the proposed algorithm works by separating the tree-structured network into different subdivision to find the shortest path.

2.2 Heuristic Search

Heuristic means the usage of estimated values to reach the target in a short time. The popular heuristic algorithm is A* algorithm. At each iteration of its main loop, A* needs to determine which partial paths should be expanded. It does so based on the estimation of the costs (total weight) of different paths travelling from current node to the goal node. Specially, A* selects the path that minimizes

$$f(n) = g(n) + h(n) \quad \textbf{(1)}$$

In Equation (1), n is the last node on the path, $g(n)$ is the cost of the path from the start node to n and $h(n)$ is a heuristic value that estimates the cost of the cheapest path from n to the goal [22]. A* avoids expanding paths that are already expensive. It is widely used because of its good performance and accuracy. Heuristic search pre-stores the estimated weighted values between two points in the plane by using

distance based methods, such as Euclidean distance (straight-line distance method) which is popular for the plane. Table 1 depicts the straight-line distances from each node to node B in Figure 1. When a graph is huge, the pre-processing time may be long due to the calculation of the heuristic values for all nodes in the graph.

Table 1. The straight-line distances from each node to node b for the weighted graph in Figure 1

Node	Distance	Node	Distance
A	366	M	241
B	0	N	234
C	160	O	380
D	242	P	10
E	161	R	193
F	176	S	253
G	77	T	329
H	151	U	80
I	226	V	199
L	244	Z	374

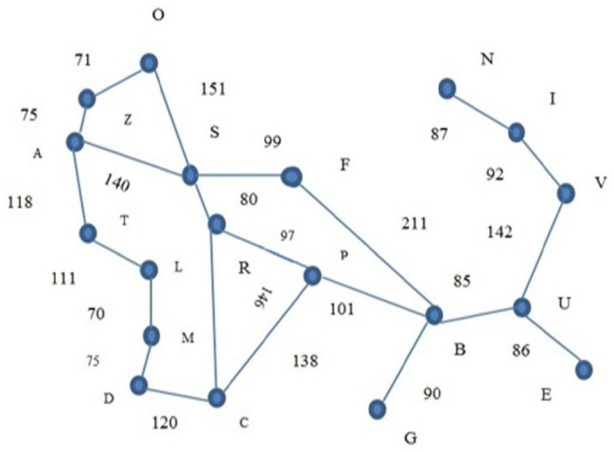


Figure 1. A weighted graph

To store the generated and the extended nodes, the table OPEN is applied. The table CLOSE is used to store the generated and selected nodes for getting the optimum route.

Now we will find the shortest path between node A and B for the graph as shown in Figure 1 by using the A* heuristic search. On the working example of the algorithm, firstly we compute the total cost $f(n)$ of the start node A according to Equation (1) in which the value of $g(n)$ always get zero at the start node. Then, the total cost value $f(n)$ is 366 because heuristic value $h(n)$ is 366 between start node and goal. This node will be added into the closed list.

Open node and close node can be seen in white box and blue box in Figure 2. The neighbors of the node A are added into the open list. Algorithm finds the cost of the neighbors and compares the values of the nodes in the open list. After computing, it selects node S in Figure 2 and node S will be added into the closed list.

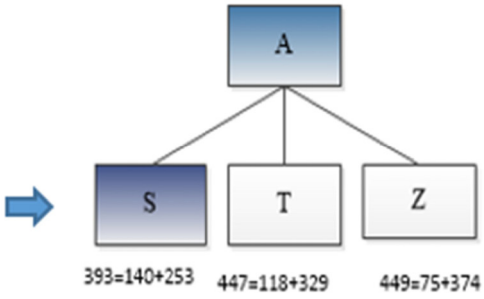


Figure 2. Choosing a node among the neighbors of node A

During the finding the minimum between node A and B in a graph, this will compare all nodes in open list as shown in Figure 3. Node T and Z are already exists in open list but it will consider to compare all nodes in open list repeatedly. When a graph that contains many incoming and outgoing edges for each vertex is traversed to find the shortest path, many comparison times will occur.

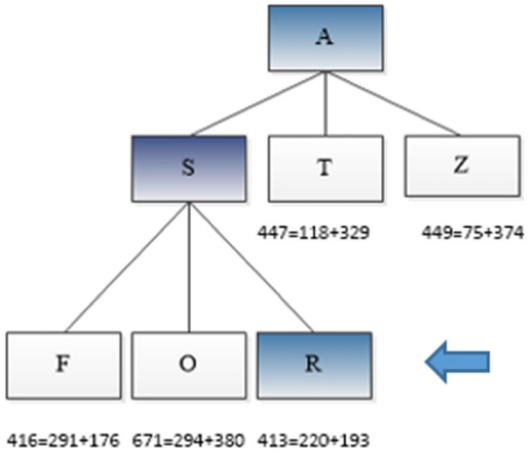


Figure 3. Choosing a node among the neighbors of node S

Inspired from this idea, a novel algorithm that uses both forward and backward heuristic search for finding the paths is proposed, description details will be given in the next Section.

3 Combined Forward and Backward Heuristic Search Algorithm

3.1 Basic Idea

In order to demonstrate the basic idea of our method, we will find the shortest path between the source node A and destination node B according to the Figure 1. Heuristic forward and backward search is applied in this system. Our system uses two threshold values. The first one is calculating the threshold value by using degree based threshold calculation method. This threshold value is used for separating a large tree into subtrees. When the graph is dense, the threshold value

may be large. The second one is to define the number of paths or shortest results $|rs|$ that will find from the subtrees. The system will check the number of neighbor nodes of all nodes at current level for a network graph. When the number of these neighbor nodes or incoming and outgoing edges of all nodes at current level are greater than or equal to the threshold, large tree is divided into the subtrees. Each subtree will find the shortest paths until the number of paths are equal to the $|rs|$. While finding the shortest path from each subtree, the heuristic search is used. The algorithm will find the shortest paths from A to B (forward search) and B to A (backward search) in this way.

Our system uses forward and backward search but they will work separately. Dijkstra with bi-directional [14] uses forward and backward approach in bi-directional search and they will meet at a point and finally give the result. However, the Dijkstra with bi-directional needs to traverse almost the whole graph to find the shortest path. The proposed algorithm does not need to traverse the all nodes of the graph and it especially computes the shortest and possible paths depend on the adjacent nodes using heuristic search. Finally, the proposed system will determine which one is the shortest according to the shortest results from A to B and from B to A.

3.2 Proposed Method

The proposed method of our system is shown in Figure 4. During the pre-processing stage, we construct the graph database and store the spatial data in it and find the distance between nodes using Haversine Distance [24] according to Equation (2), it especially focuses for the latitude and longitude data on the sphere.

$$haversin(\frac{d}{r}) = haversin(\phi_2 - \phi_1) + \cos(\phi_1)\cos(\phi_2)\, haversin(\lambda_2 - \lambda_1) \quad \textbf{(2)}$$

where d is the distance between two points with longitude λ_1, λ_2 and latitude ϕ_1, ϕ_2, r is the radius of the Earth.

Threshold value *th* is needed to divide the large tree into subtrees. Therefore, after pre-processing, a degree-based threshold calculation method is used to find it. If the network database is same, it does not need to calculate again.

The shortest and possible paths for each subtree are calculated by working the processes within the blue box in forward and backward search in Figure 4 and the paths will be stored in the database. After finding these paths in both directions, global shortest and possible paths are calculated by using quicksort.

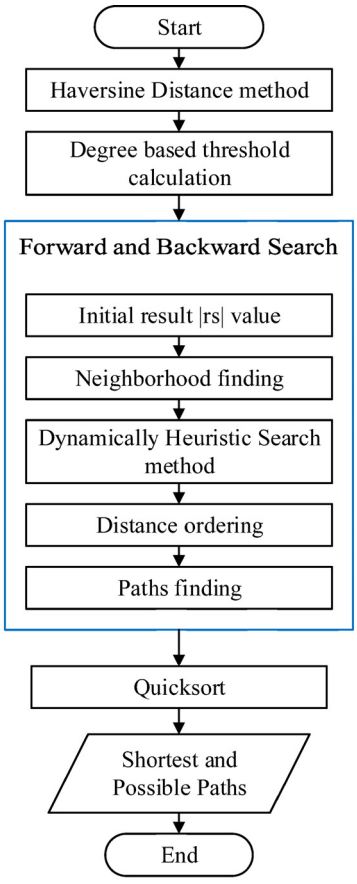


Figure 4. System flow

There are three main parts: first step is finding the threshold value *th* and dividing a large tree into small segments, second step calculates the heuristic values to know the expand node for the next level and it will find the local optimum results and final step is computing the global shortest and possible paths.

3.2.1 Subdivision Scheme

Hypergraph network is a very complex network and it can be divided into pieces to be mapped the jobs on different segments as shown in Figure 5. The system uses two threshold values *th* and *rs* as shown in Figure 4. Threshold value *th* is calculated using degree based threshold calculation method for separating the tree. Proposed method consists of four phases that are checking the degree of vertices, separating different subtrees, finding the paths and considering the accuracy of the shortest paths and performance. Threshold value *th* of a graph varies depend on the degree of vertices and nature of the graph. The degree values of the graph are used as the testing values for the threshold parameter. Then, the proposed method checks the number of edges of all nodes at the current level. If the total number of edges of all nodes at the current level is greater than or equal to the threshold value, the tree is separated into small segments. Each subtree finds the paths by using dynamically heuristic method with randomly selected start and end nodes. After the paths are finding, it chooses the threshold

value based on two factors such as the accuracy of shortest path and performance. The system needs to execute it only for the first time, when the network database is same.

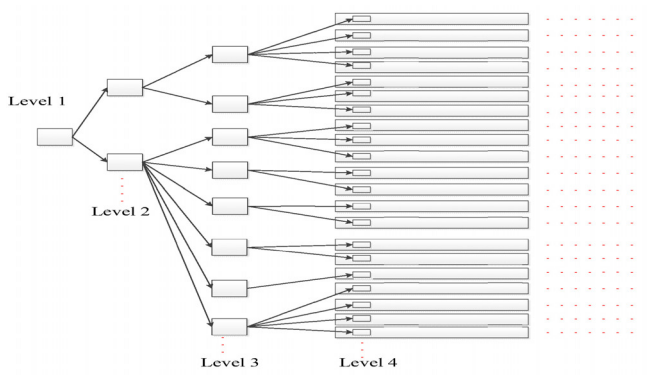


Figure 5. Demonstration of subdivision

In Figure 5, the tree is separated into the different segments at level 4. Firstly, the system traverses the paths that go from start (level 1) to the next level. During the traversing the graph, when a new level is reached, two inspections occur that are tracking the path and checking the total number of edges of current level. If a path among of these paths reaches to the destination, this path is stored into the database and it is removed from the tree. When the total number of edges of all nodes is greater than or equal to the predefined threshold value *th*, the tree is separated into the different subtrees.

At the beginning of the system, user needs to define *rs* value. *Rs* is the number of shortest path that need to find from each subtree. Each subtree finds the shortest paths using heuristic search until the user defined threshold value *rs* is reached. When the threshold value *rs* is specified as three, each subtree $\{st_i \mid 1 \le i \le 21\}$ will find the three local optimum results for their outputs.

3.2.2 Heuristic Values and Choosing the Next Expanded Node

Our system uses heuristic values while finding the results in route analysis phenomenon and it calculates these values dynamically while execution is made. Besides, the proposed system does not consider the nodes that already visited and other subtrees while choosing the next expanded node from current subtree. Each subtree works separately to find the shortest results.

Euclidean Distance [28] is used to measure the distance between two points: current neighbor node and target node (*t*). After computing the distances between the neighbors and target, ordered linked list is used to store the nodes according to descending order of these distance values. Visited linked list is used to store the result nodes while going each level to get the shortest path.

For example, if the current node n_1 contains three neighbors $\{n_{1i} \mid 1 \le i \le 3\}$ and the system will calculate the heuristic values between $\{n_{1i} \mid 1 \le i \le 3\}$ and target node (*t*). These nodes are stored in the ordered linked list according to descending order of heuristic values and the minimum heuristic value is chosen for the next level. This minimum node is stored into the visited linked list. Now, n_{13}, n_{11}, n_{12} will exists in the ordered linked list. Let assume n_{12} is the node that owns minimum heuristic value, and then n_{12} is chosen to go the next level. When the neighbors of current selected node n_{12} are n_{11}, n_{14}, n_{15} and n_{11} is an already visited node. Therefore, the system does not calculate the heuristic values of previous node n_{11} again to go the next level. By using this way, n_1 passes through the nodes and edges to reach the target. When a subtree gets the shortest result, the system checks the number of current shortest results are equal to the number of threshold value *rs*. If the number of results are less than the threshold value *rs*, it will find the next shortest path. For example, the first shortest result in visited linked list is $n_1, n_{12}, n_{14}, ..., n_t - 1, n_t$. The system will remove the last two nodes of the shortest result. The next node that owns the second minimum value is chosen from the ordered linked list and this node is added at the end of the visited linked list. Then, the system will find the second shortest path using heuristic search. When the current number of results reach to the threshold value *rs*, the jobs of current subtree will terminate.

Proposed system chooses a node for the next level depend on the heuristic values of neighbor nodes while A* considers to compare all nodes in the open list. The system finds the local optimum results of forward and backward search.

3.2.3 Computing the Shortest and Possible Paths Among of the Local Optimum Results

Proposed system based on the divide and conquer strategy. Firstly, the system divides the large tree into many small subtrees and each subtree finds the results according to the predefined threshold value. This will work in forward and backward search. Finally, all local optimum results will exist in the same database.

After finding the local optimum results of each subtree from two directional search, the system compares the different results by using quick sort. Quick sort algorithm is used to sort and compare the local optimum results because it is one of the most popular sorting methods and the time complexity of this algorithm is $O(N \log N)$ in average case and it does not need the additional memory. Besides, this algorithm's time complexity is better than other sorting algorithms such as bubble sort, selection sort and insertion sort. Finally, the system produces the best

suitable results as the outputs for a graph network by using quick sort and local optimum results.

3.3 Pseudo Code of the CFBHS

This algorithm finds the shortest path and possible paths between two points in a graph using forward searching, backward searching and heuristic approach. When the system starts, MainFunction (Algorithm 1) will work. This algorithm will create two objects with parameterized values (startnode, targetnode, flag variable) of CFBHS java class for two directional search (forward and backward) and these objects will call the CFBHS java class. Flag variable is used to check whether two directional search is finished. When the value of flag variable is equal to two, forward and backward search is finished. Firstly, calculate function (Algorithm 2) of the CFBHS java class will work. This function calls the divideLargeTree function (Algorithm 3) to divide the large tree into the small number of trees. Algorithm 3 checks whether the total number of edges $|E|$ of all nodes at the current level are greater than or equal to the predefined threshold value. $|E|$ is equal to the total number of adjacent nodes of all nodes in current level. Level 1 is checked in Algorithm 3 and the total number of edges for other levels of tree are checked using Algorithm 3 and Algorithm 4. The details of work about subdivision can be seen in (3.4) of this section. After dividing the tree by using Algorithm 3, this algorithm will call the CFBHS function (Algorithm 5). Algorithm 5 works based on the heuristic search and it uses the Euclidean Distance to find the distance between two points, current neighbor node and target and removes the unnecessary nodes and edges while going to the destination. It will work to find the local optimum results on each subtree that give from algorithm 3. Algorithm 5 finds the number of results from the subtrees according to the predefined threshold value *rs*. If the number of threshold value $|rs|$ is equal to *m*, each subtree will find *m* local optima. When a local optimum result is get at Algorithm 5, printPath (Algorithm 6) subroutine will work and it will increase the number of results. Finally, when the forward and backward direction have executed for finding the local optimum results in a graph, the calculate subroutine (Algorithm 2) will call printGPath (Algorithm 7) subroutine and it will find the best suitable shortest and possible results using quick sort algorithm and local optimum results.

Algorithm 1. MainFunction

1. Create the two objects to call CFBHS class
2. firstobj (startnode, targetnode, flag) //forwardsearch
3. secondobj (targetnode, startnode, flag) //backwardsearch

Algorithm 2. calculate

Input: start node, target node, flag
1. call divide Large Tree (startnode, threshold) function
2. **if** (flag == 2) **then**
3. call the printGPath function
4. **end if**

Algorithm 3. divideLargeTree

Input: startnode, threshold, Graph
Output: subtrees
1. j ← 0
2. adjacentLinkedlist ← adjacentNodes(startnode) // level1
3. i ←size(adjacentLinkedlist)
4. k ←i
5. **while** (i>0) **do**
6. check the neighbor nodes
7. if the current neighbor node is equal to the target node, store this traversed path into the local optimum database
8. remove this traversed path from tree
9. i←i-1
10. **end while**
11. **if** (k >= threshold) **then**
12. separate the tree into subtree
13. call the CFBHS (traversedpath) function to find the local optimum results from each subtree
14. **else**
15. work step 16 to check the number of adjacent nodes of all nodes that exist at current level // level 2 and other levels
16. **while** (k>0) **do**
17. j← j+ CheckNumofAdjNodes (adjacentLinkedlist(k))
18. do from step 6 to 8
19. k ←k -1
20. **end while**
21. **end if**
22. **if** (j>= threshold) **then**
23. do step 12 and 13
24. **else**
25. k←j
26. j ←0
27. do step 15
28. **end if**

Algorithm 4. CheckNumofAdjNodes

Input: A node that want to find adjacent nodes
Output: number of adjacent nodes of current input node
1. adjacentLinkedList ← adjacentNodes(node)
2. i ← size(adjacentLinkedlist)
3. return i

Algorithm 5. CFBHS (Combined Forward and Backward Heuristic Search)

Input: Graph, start node, target node, traversed path (visited linked list), flag, rs
Output: local optimum paths

1. get the last node from the traversed path in visited linked list of current subtree
2. find the adjacent nodes of the current node
3. insert the adjacent nodes of the current node to the Linkedlist named as adjacentnodes
4. check each node in adjacentnodes
5. **if** a node in adjacentnodes contain in visited linked list
   then
6. remove this node from adjacentnodes
7. **else**
8. compute the distance between the nodes in adjacentnodes linked list and target node by using Euclidean distance
9. **end if**
10. store the nodes in the ordered linked list according to descending order using heuristic distance values
11. choose minimum node from ordered linked list
12. add this choice node into the visited linked list
13. **if** current choice node is equal to target node **then**
14. call the printPath function
15. **else**
16. call the CFBHS function recursively
17. **end if**
18. **if** (current result number > number of predefined result named as rs)) **then**
19. terminate the job of current subtree
20. **else**
21. remove the last two nodes (target node and a node before target) from the visited linked list
22. choose the second minimum node from ordered linked list
23. add this node at the end of the visited linked list
24. call the CFBHS function recursively
25. **end if**

Algorithm 6. printPath

Input: Graph, visited Linked list

1. Calculate the total distance of a route that gives from the visited linked list
2. Store the local optimum result into the database
3. Increment the current number of result

Algorithm 7. printGPath

Input: local optimum paths
Output: shortest and possible paths

1. Use the Quick-Sort function to get the shortest and possible paths

3.4 Working Example

This section will demonstrate the working of our algorithm, we will find the shortest path between node s and o according to the Figure 6. Let assume two threshold values *th* and *rs* are 12 and 2. At the working example, the system will check the number of edges $|E|$ of all nodes for a level. If the $|E|$ is greater than or equal to the threshold value *th*, the system will divide the tree. At level 1, the number of edges for node s are 3 and at level 2, there are three nodes (d, c, b). The number of edges of d, c and b are 3, 3, 4. So, the total number of edges at level 2 are 10. At level 3, there are five nodes (h, g, c, e, f) and the total number of edges are greater than the threshold value *th*. Therefore, the tree is divided into the subtrees at level 3. We get the six subtrees for the undirected graph in Figure 6. These are (s, d, h), (s, d, g), (s, c, g), (s, b, c), (s, b, f) and (s, b, e). Each subtree finds the shortest path using heuristic search until the threshold value *rs* is reached. For example, the subtree (s, d, h) finds the neighbors of h. Among of the neighbors(d, g, k, l, m), d is the already visited node. So, the subtree (s, d, h) calculates the distance between the neighbor nodes of h and target node o except node d. Instead of using the coordinate values, we use the spatial data (latitude and longitude values) for finding the distance values between two nodes. k is the nearest node according to the heuristic value and k is chosen for the next level. We use the spatial data of Yangon, Myanmar in Google Map. In this way, the subtree (s, d, h) gets the shortest result s, d, h, k, o. Each subtree will find the two shortest paths. This work will occur in forward and backward search. Finally, the system will find the shortest and possible paths by using the local optimum results and quick sort algorithm.

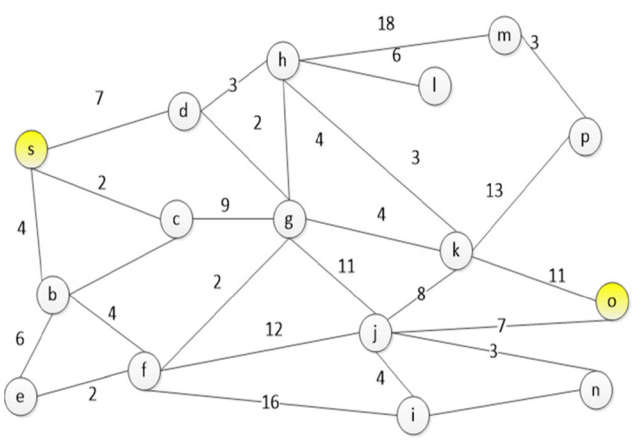


Figure 6. Undirected graph

4 Experimental Results

In order to evaluate the proposed method, experiments are carried out based on data of Yangon city in Myanmar. Yangon region is composed of 33 townships and Yangon downtown is the main transportation townships among of the townships in Yangon. Latha, Lanmadaw, Pabedan, Kyauktada,

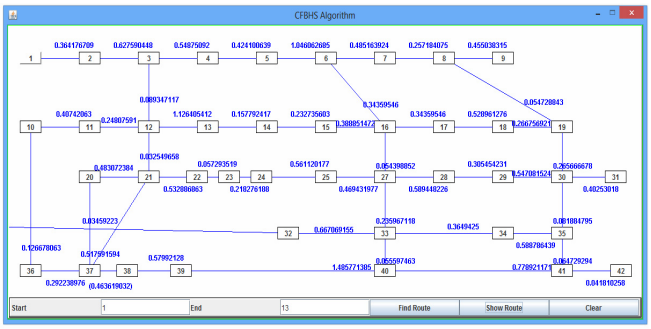
Botahtaung and Pazundaung townships exist in Yangon downtown area.

There are three major parts in this section: first part is testing the different components with different thresholds, second part is the time complexity evaluation between different algorithms and different data scalars and final part is the statistical analysis of both undirected and hypergraph network. A* and Bi-directional Dijkstra are the best ones in terms of response time in order to find the shortest path, the journal articles from [25-27]. The time complexity between Bi-directional Dijkstra [14], A*[22, 28] and the proposed CFBHS algorithm are compared with Yangon downtown data. Actually, although there exists about 300 bus stops in Yangon downtown area, only the main bus stops are considered, and we can easily inspect and analyze the results using graph structure. Besides, the time complexity of CFBHS algorithm are analysed with different data scalars. In this case, we will use the dataset of Yangon city.

4.1 Testing the Different Components Using Different Thresholds

Yangon downtown map in Figure 7(a) is constructed into a simulation graph is shown in Figure 7(b), which contains 41 nodes. Firstly, the system calculates the threshold value *th* by using the degree values of the graph as the testing parameters.

(a) Google map of Yangon downtown area

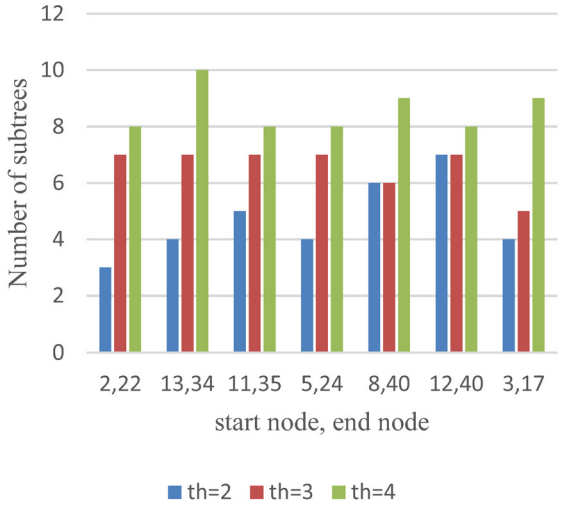


(b) Graph structure for Yangon downtown area

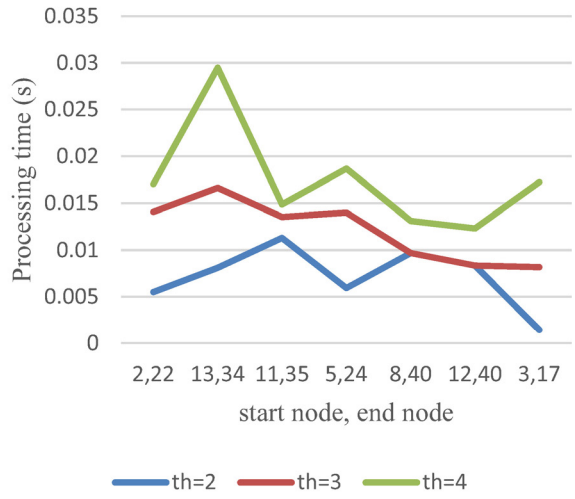
Figure 7. Building from map data into graph data

4.1.1 Testing Yangon Downtown Area with Different Thresholds

The degree of the vertices in graph for Yangon downtown area are one, two, three and four. Degree one is removed for threshold because the vertex that owns the degree "one" cannot split into the subtrees. And then, we tested different threshold values such as two, three and four. According to the experiments, although all threshold values can find the shortest path, threshold value two is the best because it's execution time is the fastest as shown in Figure 8(a) and Figure 8(b). The number of subtrees and performance is directly proportional as shown in Figure 8 because the performance of the proposed system will be faster when the number of subtrees is lesser.



(a) Number of subtrees for randomly chose nodes and different thresholds



(b) Processing time for different thresholds

Figure 8. Testing different threshold values

4.1.2 Testing Yangon Region with Different Thresholds

The degree of vertices of public bus transportation network graph of Yangon region are from degree 1 to degree 11. After testing this by using different threshold values from 2 to 11, the system found the threshold value 4 is the best to divide the tree. When the threshold value 2 and 3 are used as the testing values, at sometimes the system can produce only the second shortest path as described in Figure 9.

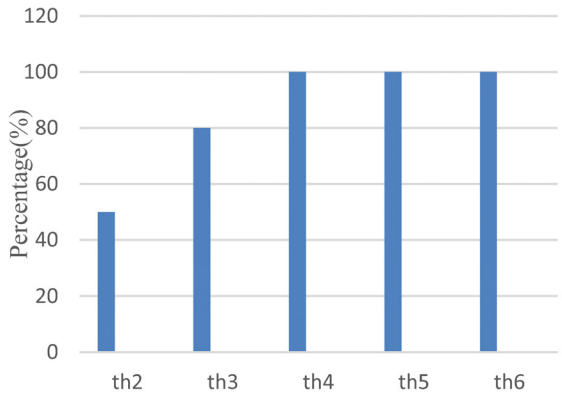
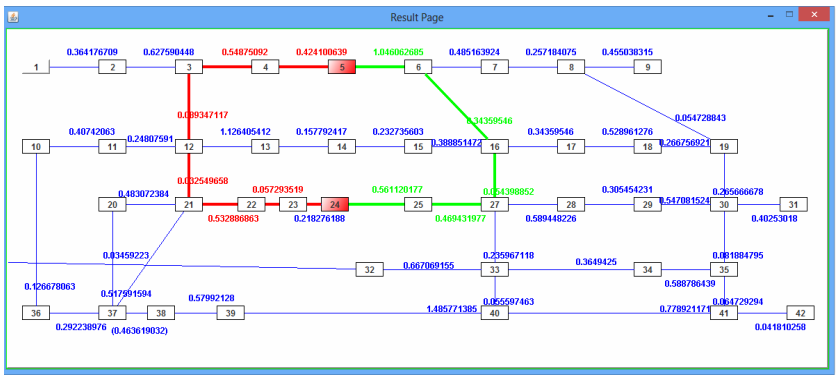


Figure 9. Yangon bus data with different thresholds

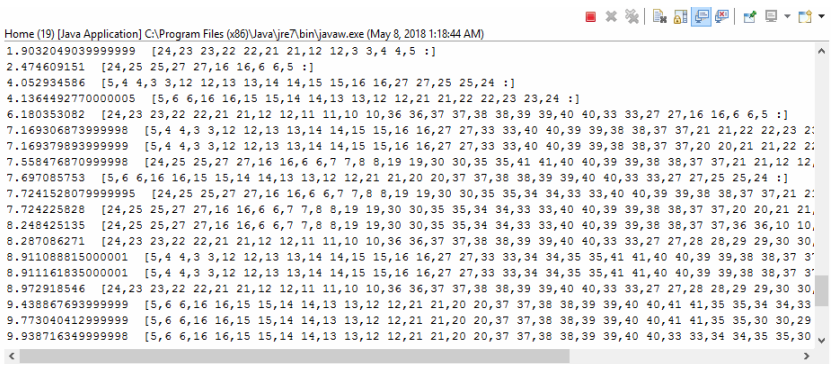
By comparing two different components by giving the specified thresholds, at sometimes, Yangon region graph cannot get the optimized shortest path by giving the threshold value two and three because it contains large degree vertices and graph's data is large. So, the most suitable threshold value for a graph varies depend on the degree of vertices and nature of the graph. Degree based threshold calculation method can give the best threshold value for different components.

4.2 Time Complexity Evaluation

The system can produce the shortest and many possible paths simultaneously. The first shortest path (red colour) and second shortest path (green colour) can be viewed on the graph for Yangon downtown area as shown in Figure 10(a) and other possible paths can be viewed in text mode in Figure 10(b). Figure 11 compared the time complexity between the proposed CFBHS algorithm and other popular shortest path algorithms such as A* and Bi-directional Dijkstra's algorithm. Yangon downtown data is used for this experiment. The proposed algorithm defines the number of results $|rs|$ that need to find from the subtrees using different parameter values such as one and ten as shown in Figure 11. According to the experimental results, the proposed algorithm can find the shortest and possible paths with very low time complexity. In addition, we can see that the number of paths are increasing when the number of predefined result value $|rs|$ is large as shown in Figure 12.



(a) Two shortest path



(b) Shortest and possible paths

Figure 10. Finding the shortest and possible paths for Yangon downtown area using threshold value two and $|rs|$ value five

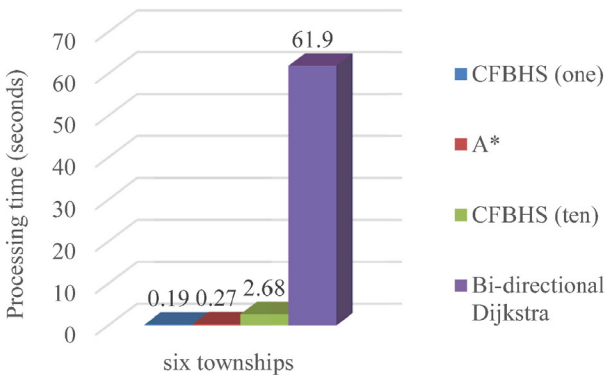


Figure 11. The time complexity of three algorithms for Yangon downtown area

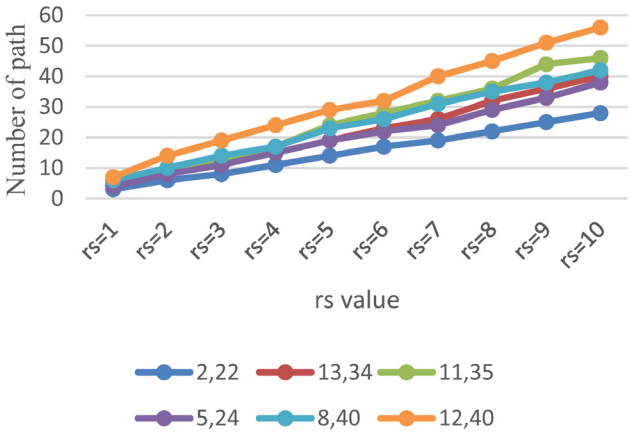


Figure 12. Number of paths according to *rs* for Yangon downtown area

The next one is to compare about the time complexity of the proposed method depends on the amount of data. Yangon city contains large amount of data about bus-lines as shown in Figure 13. The numbers of nodes in Yangon city is about 6000 and contain many connections between them. We use keyhole markup language (kml) file to load the spatial data on the map. Yellow icon is used to show the nodes (bus stops) and red line is constructed as the connections (edges) between the nodes according to the bus-line. After executing the proposed system to find the minimum on large amount of data, we get the results in java editor console and then we can trace the result on google earth. In this test, CFBHS algorithm finds the one shortest path for one subtree and threshold value *th* is used as four. There were no significant differences about the time complexity between small and large data as shown in Figure 14.

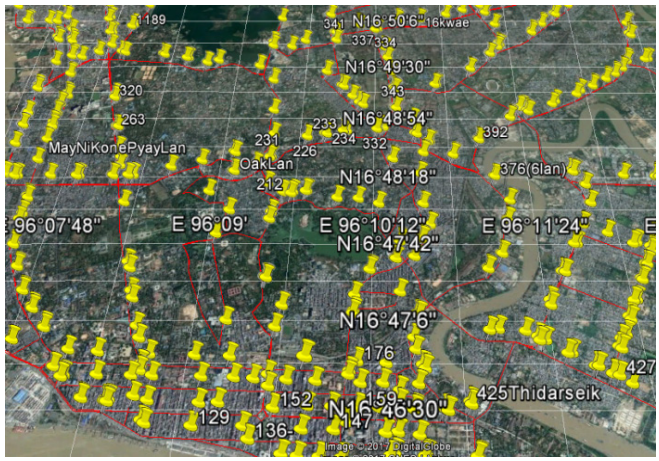


Figure 13. Yangon city

Figure 14. The time complexity of CFBHS for Yangon downtown area and Yangon city

CFBHS algorithm performs to find both shortest and possible paths at the same time in a network. Finding the one shortest path from each subtree is enough to get the optimal path for the complex network but finding the perfect possible ways are suitable by calculating the results more than one. If the threshold value $|rs|$ is equal to m, each subtree will find a local optimum result that differ about the local optima of other $m-1$ at each time. One by one, the subtrees work their jobs in sequentially. Therefore, when the number of jobs as well as the number of shortest results that will find from the system increase, time complexity increases a little. The variances of time complexity can be seen in Figure 15. CFBHS algorithm accomplishes to meet the two objectives with a very low time complexity.

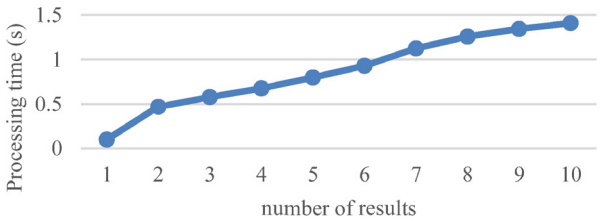


Figure 15. The time complexity of CFBHS depended on the number of results of the subtrees in 10 runs

4.3 Statistical Analysis of Both Undirected and Hypergraph Network

A hypergraph is a complex network and it is composed of many vertices and edges as the ordinary graph but an edge in this graph can join any number of vertices. For example, a hypergraph contains the set of vertices $V=\{v_1,v_2,v_3,v_4,v_5,v_6,v_7\}$ and $E=\{e_1,e_2,e_3,e_4\}$ in which $\{e_1,e_2,e_3,e_4\}$ is the same as $\{\{v_1,v_2,v_3\},\{v_2,v_3\},\{v_3,v_5,v_6\},\{v_3,v_5,v_6\}\}$. It can be constructed into ordinary graph [23] and each vertex can have many edges. Formally, a hypergraph H is a pair $H=(X,E)$ where X is a set of elements called nodes or vertices, and E is a set of non-empty subsets of X called hyper edges or edges. Yangon bus network is a complex hypergraph network. It is constituted with 61 bus lines, it contains large number nodes, and edges. The graph network in Figure 16 is constructed for four bus lines, which contains 442 nodes. Moreover, a node may own many incoming edges and outgoing edges. This section describes the comparison results between different algorithms and data size according to the statistical analysis.

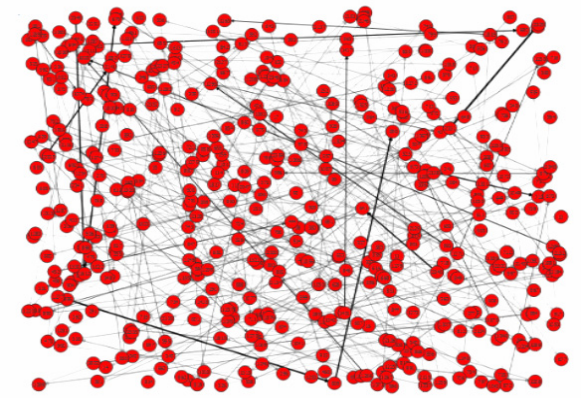


Figure 16. Simulated graph using SocNetV for hypergraph Yangon bus network that contains 442 nodes

The time complexity of the proposed CFBHS algorithm is compared with A* and Bi-directional Dijkstra by using the Yangon downtown city's data. Figure 17 shows the average time complexity of proposed algorithm is better than A* although the threshold value is changed. Figure 18(a) shows the total time complexity of these algorithms using different execution times. If we calculate only one shortest result for one subtree in CFBHS algorithm, time complexity is better than the other algorithms according to the statistical analysis. This comparison also calculates ten shortest results for proposed algorithm. In this case, our algorithm may be little delay to get the results than the A* algorithm because each subtrees works their jobs according to sequential processing, but we cannot miss the second shortest and third shortest paths. Besides, our algorithm can give both shortest and possible paths as outputs simultaneously, while A* can produce only the shortest path. The experiment describes our algorithm's time complexity is better than Bi-directional Dijkstra's algorithm for both two conditions that are *rs* value one and ten in Figure 18(a). Figure 18(b) shows the comparison of time complexity depend on the data size. Data set is used from Yangon bus system. It is a very complex hypergraph network. According to the experimental results, the time complexity of the proposed algorithm is slightly different although the data size is large as described in Figure 18(b).

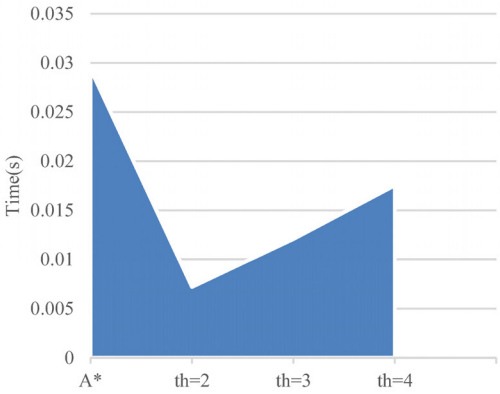
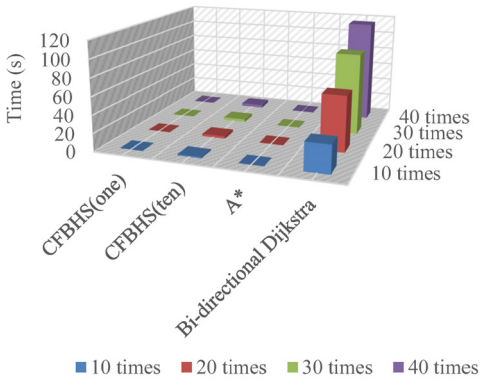
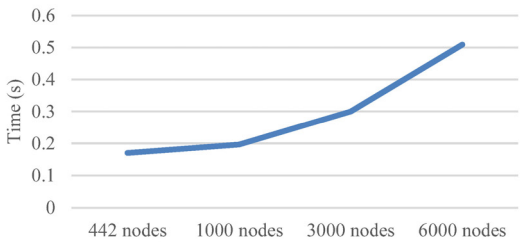


Figure 17. Average time complexity between CFBHS with different thresholds and A*



(a) Time complexity of proposed CFBHS, A* and Bi-Directional Dijkstra's algorithm depends on execution times



(b) Time complexity analysis for different data size using CFBHS algorithm.

Figure 18. Statistical analysis of both undirected graph and complex hypergraph

5 Time Complexity Analysis

To analyze the performance of proposed algorithm, asymptotic analysis is used. Asymptotic analysis is measuring the efficiency of algorithms that does not need to consider on the machine specific factors such as operating system, processors and hardware. It measures the amount of time that will take for an algorithm at running a function with the length of the input. This section describes three parts. The first two parts are describing the time complexity analysis in detail for proposed algorithm and A*. The final part is the comparison of performance between these two algorithms using the Yangon downtown data.

5.1 Detail Performance Analysis for CFBHS

The proposed algorithm is analyzed by four main factors that are forward search, backward search, number of subtrees and the comparison time for choosing a node to go the next level. After analysis, the total time complexity of CFBHS is, where n is the total number of vertices of a graph, k is the number of subtrees or threshold value th and m is subtracting one from the maximum degree. If the maximum number of degrees about the nodes of a graph is four, at most only three nodes are needed to compare for the next level. Therefore, at each step, the maximum comparison times occur m times.

If the shortest path length is n, the algorithm compares mn times for going the next level about this path. However, it does not need to consider the comparison case at $n-1$ step if the end node exits at step n because when the end node is found, comparison does not occur, and the algorithm will choose the end node. Besides, when the tree is divided at level 2, the total number of nodes that need to compare will reduce for one-step and the comparison nodes will get from n into $n-1$. Therefore, the total comparison nodes of one subtree are $n-k$. Then, $m(n-k)$ will take as the comparison time for one subtree. If the total number of subtree is k, then it will take $km(n-k)$. After computing this by two directional searches in forward and backward, the complexity is $2km(n-k)$. Finally, we use the quicksort algorithm to find the best paths. Although the complexity of quicksort is $O(n\log n)$ for average case, we will consider as the worst case. Therefore, the complexity of quicksort is $O(n^2)$, in which n is the number of paths that need to sort. The total time complexity for algorithm is resulted by using the algorithm analysis approach. It can be denoted by using asymptotic notation BigO for the worst case. Then, the time complexity of proposed algorithm for worst case is $O(kmn)$.

5.2 Detail Performance Analysis for A*

A* is the popular one among of the shortest path algorithms to find the shortest path in a short time. The time complexity of A* [29] is considered using $O(b^d)$ where, b is the branching factor and d is the depth. Otherwise, the total time complexity of A* is $\sum_{i=1}^{d-1} T_i$, where T is the time complexity for each step and d is the depth of path. Time complexity may vary depend on the shortest path length. Figure 19 finds the shortest path between vertex 'A' and vertex 'L' using A*. We will consider the worst case for this phenomenon. This algorithm works three steps to find the path between vertex A and vertex L.

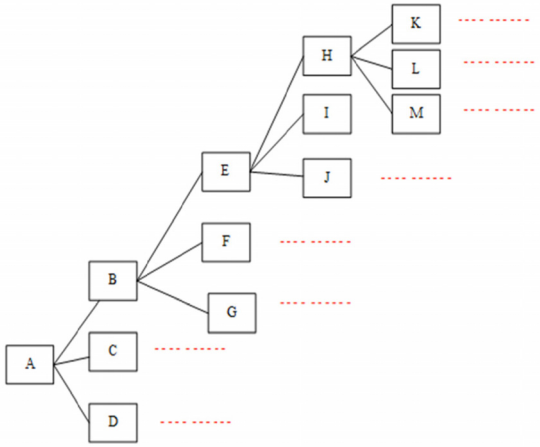


Figure 19. Shortest path finding between node A and node L

For step 1, the time complexity is T(1) = 3, because three nodes such as node B, node C and node D are compared. And then node B is chosen to expand for the next level.

For step 2, the time complexity is T(2) = 5, because node C, node D, node E, node F and node G are considered to choose the node for the next level.

For step 3, the time complexity is T(3) = 7, because node C, D, F, G, H, I, J are compared for the next step.

Now, the total comparison time for algorithm is $\sum_{i=1}^{d-1} T_i = T(1)+T(2)+T(3) = 3+5+7 = 15.$

If the graph is very large, this way is not easy to calculate the execution time. So, it uses b^d to compute the time complexity of the path. For example, Figure 19 expands three nodes for each vertex in general, then we can specify b is three and d is four for that case because d is the depth of the shortest path. Then, the total complexity is 81.

5.3 Performance Comparison between CFBHS and A*

The efficiency of these two algorithms are compared by using the data of Yangon downtown area in Section 4.1.1. This graph contains 41 vertices and m is three because the maximum degree of vertices is four. The

degree of vertices are one, two, three and four. Therefore, threshold value two, three or four can be used for the threshold parameter k according to the degree-based threshold calculation method. The execution times of these algorithms are computed by using detail time complexity analysis. The time complexity of the proposed algorithm is described in Table 2.

Table 2. Performance analysis of CFBHS

k	m	n	Checked nodes $(n-k)$	Execution time
2	3	41	39	476
3	3	41	38	702
4	3	41	37	920

A* algorithm works the total time complexity $\sum_{i=1}^{40} T_i$ for Yangon downtown data. Then, the total complexity is 1680. Therefore, the performance of proposed algorithm is better than the A* according to the time complexity analysis. The comparison of these two algorithms is considering for the worst case. In real time, the performance of these two algorithms is very fast than other classical algorithms according to the experiments and they do not need many execution times to get the shortest result.

6 Conclusion

In this paper, a novel algorithm named CFBHS is proposed for finding both shortest and possible paths. It uses forward and backward heuristic searches. Most of the shortest path algorithms use the whole graph for searching the minimum result but our algorithm divides a complex network into the small subtrees for finding the results and it works each subtree separately. Therefore, our method is more efficient. Besides, different from other methods such as A* and Bi-directional Dijkstra, which can only find one path, our algorithm can provide not only the shortest path but also the possible ones at the same time.

The comparisons between CFBHS algorithm, A* and Bi-directional Dijkstra algorithm show that our proposed algorithm outperforms others in terms of time complexity. The time consumption of the proposed method varies slightly for graph scales ascending from a collection of over 400 nodes to 6000 nodes. According to the statistical analysis and experiments, it can produce the shortest path and possible paths with reasonable time complexity for both large and small data. In further work, parallel processing and novel algorithms [30-31] will be adopted in our CFBHS algorithm for forward and backward searching and improve the efficiency of each subtree's work.

Acknowledgements

This work is supported by the Scientific Research Project of Fujian University of Technology (GY-Z160130, GY-Z160138, GY-Z160066), the Natural Science Foundation of Fujian Province (2017J05098, 2017H0003, 2018Y3001), Project of Fujian Education Department Funds (JK2017029, JZ160461), Project of Fujian Provincial Education Bureau (JAT160328, JA15325) and Project of Science and Technology Development Center, Ministry of Education (2017A13025).

The authors also gratefully acknowledge the helpful comments and suggestions of the reviewers, which have improved the presentation.

References

[1] V. Bhanumurthy, V. M. Bothale, B. Kumar, N. Urkude, R. Shukla, Route Analysis for Decesion Support System in Emergency Management through GIS Technologies, *International Journal of Advanced Engineering and Global Technology*, Vol. 3, No. 2, pp. 345-350, February, 2015.

[2] A. Gunes, J. Kovel, Using GIS in Emergency Management Operations, *Journal of Urban Planning and Development*, Vol. 126, No. 3, pp. 136-149, September, 2000.

[3] H. A. Rakha, I. EL-Shawarby, M. Arafeh, F. Dion, Estimating Path Travel-Time Reliability, *2006 IEEE Intelligent Transportation Systems Conference*, Toronto, Canada, 2006, pp. 236-241.

[4] W. Wen, S. W. Hsu, A Route Navigation System with a New Revised Shortest Path Routing Algorithm and Its Performance Evaluation, *WIT Transactions on The Built Environment*, Vol. 77, pp. 733-743, January, 2005.

[5] K. A. Korkmaz, Emergency Management for Infrastructure and Transportation Systems in the US, *2017 Baltic Geodetic Congress*, Gdansk, Poland, 2017, pp. 179-183.

[6] E. Higgins, M. Taylor, H. Francis, M. Jones, D. Appleton, *The Evolution of Geographical Information Systems for Fire Prevention Support, Fire Safety Journal*, Vol. 69, pp. 117-125, October, 2014.

[7] A. Dabhade, K. V Kale, Y. Gedam, Network Analysis for Finding Shortest Path in Hospital Information System, *International Journal of Advanced Research in Computer Science and Software Engineering*, Vol. 5, No. 7, pp. 618-623, July, 2015.

[8] R. S. Parmar, B. Trivedi, Shortest Route– Domain Dependent, Vectored Approach to Create Highly Optimized Network for Road Traffic, *International Journal of Traffic and Transportation Engineering*, Vol. 5, No. 1, pp. 1-9, May, 2016.

[9] E. Lepuschitz, Geographic Information Systems in Mountain Risk and Disaster Management, *Applied Geography*, Vol. 63, pp. 212-219, September, 2015.

[10] E. W. Dijkstra, A Note on Two Problems in Connexion with Graphs, *Numerische Mathematik*, Vol. 1, No. 1, pp. 269-271,

June, 1959.

[11] B. V. Cherkassky, A. V. Goldberg, C. Silverstein, Buckets, Heaps, Lists, and Monotone Priority Queues, *SIAM Journal on Computing*, Vol. 28, No. 4, pp. 1326-1346, January, 1999.

[12] R. K. Ahuja, K. Mehlhorn, J. Orlin, R. E. Tarjan, Faster Algorithms for the Shortest Path Problem, *Journal of the ACM*, Vol. 37, No. 2, pp. 213-223, April, 1990.

[13] Y. Huang, Q. Yi, M. Shi, An Improved Dijkstra Shortest Path Algorithm, *2nd International Conference on Computer Science and Electronics Engineering*, Paris, France, 2013, pp. 226-229.

[14] M. Dramski, Bi-directional Search in Route Planning in Navigation, *Scientific Journals of the Maritime University of Szczecin*, Vol. 39, No. 111, pp. 57-62, April, 2014.

[15] M. A. Qureshi, F. B. Hassan, S. Safdar, R. Akbar, A O(|E|) Time Shortest Path Algorithm for Non-negative Weighted Undirected Graphs, *International Journal of Computer Science and Information Security*, Vol. 6, No. 1, pp. 40-46, October, 2009.

[16] H. Wang, J. Zhou, G. Zheng, Y. Liang, HAS: Hierarchical A-Star Algorithm for Big Map Navigation in Special Areas, *Proceedings– 2014 International Conference on Digital Home*, ICDH 2014, Guangzhou, China, 2014, pp. 222-225.

[17] I. Witten, E. Frank, M. Hall, M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques* (Third Edition), Morgan Kaufmann, 2011.

[18] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, T. Willhalm, Partitioning Graphs to Speedup Dijkstra's Algorithm, *Experimental and Efficient Algorithms, 4th InternationalWorkshop, WEA 2005*, Santorini Island, Greece, 2005, pp. 1-12.

[19] I. Chabini, S. Lan, Adaptations of the A* Algorithm for the Computation of Fastest Paths in Deterministic Discrete-Time Dynamic Networks, *IEEE Transactions on Intelligent Transportation Systems*, Vol. 3, No. 1, pp. 60-74, March, 2002.

[20] J. L. Bander, C. C. White, A Heuristic Search Algorithm for Path Determination with Learning, *IEEE Transactions on Systems, Man, and Cybernetics Part A: Systems and Humans*, Vol. 28, No. 1, pp. 131-134, January, 1998.

[21] X. Liu, D. Gong, A Comparative Study of A-Star Algorithms for Search and Rescue in Perfect Maze, *2011 International Conference on Electric Information and Control Engineering, ICEICE 2011 - Proceedings*, Wuhan, China, 2011, pp. 24-27.

[22] A. Goyal, P. Mogha, R. Luthra, N. Sangwan, Path Finding: A* or Dijkstra's? *International Journal in IT and Engineering*, Vol. 2, No. 1, January, 2014.

[23] C. Guo, Y. Zhen, G. C. Jia, X. W. Zhou, Multilayer Satellite Network Routing Based on Hypergraph Theory, *International Conference on Cyberspace Technology (CCT 2013)*, Beijing, China, 2013, pp. 312-315.

[24] L. Ganesh, B. P. Vijaya Kumar, Indoor Wireless Localization using Haversine Formula, *International Advanced Research Journal in Science, Engineering and Technology*, Vol. 2, No. 7, pp. 59-63, July, 2015.

[25] S. Ergün, T. A. Ğ. An, Z. Alparslan, A Study on Performance Evaluation of Some Routing Algorithms Modeled by Game Theory Approach, *Afyon Kocatepe University Journal of Science and Engineering*, Vol. 16, pp. 170-176, May, 2016.

[26] S. K. Sharma, B. L. Pal, Shortest Path Searching for Road Network using A* Algorithm, *International Journal of Computer Science and Mobile Computing*, Vol. 4, No. 7, pp. 513-522, July, 2015.

[27] M. Karova, I. Penev, N. Kalcheva, Comparative Analysis of Algorithms to Search for the Shortest Path in A Maze, *2016 IEEE International Black Sea Conference on Communications and Networking, BlackSeaCom 2016*, Varna, Bulgaria, 2016, pp. 1-4.

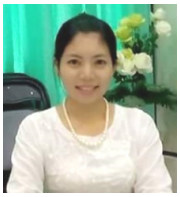
[28] B. Siregar, E. B. Nababan, J. A. Rumahorbo, U. Andayani, F. Fahmi, Nearby Search Indekos Based Android Using A Star (A*) Algorithm, *Journal of Physics: Conference Series*, Vol. 978, No. 1, pp. 1-6, March, 2018.

[29] L. Santoso, A. Setiawan, A. Prajogo, Performance Analysis of Dijkstra, A* and Ant Algorithm for Finding Optimal Path Case Study: Surabaya City Map, *MICEEI 2010,* Makassar, Indonesia, 2010, pp. 27-10-2010-28-10-2010.

[30] J. S. Pan, L. P. Kong, T. W. Sung, P. W. Tsai, V. Snasel, a-Fraction First Strategy for Hirarchical Wireless Sensor Neteorks, *Journal of Internet Technology*, Vol. 19, pp. 1717-1726, June, 2018.

[31] J. S. Pan, C. Y. Lee, A. Sghaier, M. Zeghid, J. Xie, Novel Systolization of Subquadratic Space Complexity Multipliers Based on Toeplitz Matrix-Vector Product Approach, *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 27, pp. 1614-1622, July, 2019.

Biographies



Wai Mar Hlaing received the B.C.Sc and M.C.Sc degree in Computer Science from the University of Computer Studies, Yangon in 2005 and 2009. After the graduation of her M.S. degree, she worked as a teacher in Computer University of Dawei during 2009-2011. She is working as an assistant lecturer at the Institute of Technical Innovation and Promotion (Hlaing), Myanmar from 2012 until now. She is a Ph.D. student in the University of Computer Studies, Yangon, Myanmar, and a visiting scholar in Fujian University of Technology, Fuzhou, China. Her research interests include geographical information system and spatial database.

Shi-Jian Liu received the B.S. degree in mathematics from Xiangtan University, Xiangtan, China, in 2006, the M.S. degree in computer science from Changsha University of Science and Technology, Changsha, China, in 2010, and the Ph.D. degree in computer science from Central South University, Changsha, China, in 2015. He is currently an Associate Professor in the School of

Information Science and Engineering, Fujian University of Technology, Fuzhou, China. His research interests include Petri nets, mesh/biomedical image processing, and information security.

Jeng-Shyang Pan received the B.S. degree in Electronic Engineering from the National Taiwan University of Science and Technology in 1986, the M. S. degree in Communication Engineering from the National Chiao Tung University, Taiwan in 1988, and the Ph.D. degree in Electrical Engineering from the University of Edinburgh, U.K. in 1996. Currently, he is a Dean for College of Information Science and Engineering, Fujian Universityof Technology and a director of Innovative Information Industry Research Center, Harbin Institute of Technology Shenzhen Graduate School, China. He jointed the editorial board of LNCS Transactions on Data Hiding and Multimedia Security, Journal of Computers, Journal of Information Hiding and Multimedia Signal Processing etc. His current research interests include soft computing, information security and signal processing.