## Breadth First Search

*Description:*

*Breadth First Search* is the simplest Graph Algorithm, and it is also an important prototype of many Graph Algorithm. For example, Prim Algorithm and Dijkstra Algorithm all use the Breadth First Search Algorithm.

For a given Graph G = (V, E), and source node s, Breadth First Search can be used to find all accessible nodes starting from source node s. The algorithm can be used to calculate the distance from source node s to each accessible nodes, and finally generate the *“Breadth First Search Tree”*.

The tree treats the Source Node s as Root Node, including all accessible nodes. For each accessible Node v starting from Source Node u, *in the Breadth First Search Tree, the simple routine from Node s to Node v is the Shortest Routine from Node s to Node v in Graph, which includes the least edge routine.*

*This algorithm can be used in Directed Graph and Undirected Graph.*

Breadth First Search Tree is so famous because the algorithm always enlarge the boundary between Known Node and Unknown Node. Enlarge Breadth First Search Tree through its breadth direction, which means that *the algorithm needs to finish finding all nodes away from node s by k, and then it can find all nodes away from node s by k + 1.*

*Procedure:*

The Breadth First Search Tree needs to print each node with color white, gray, or black.

During the Search Process, the node which first met would mean “finding the node”, and the Color of node has been changed. However, the color of Black or Gray means that the node has been visited. But Breadth First Search Tree would differentiate between the Black node and Gray node.

If the edge ( u, v ) belongs to E and node u is black, then the node v would be gray or black. *The white node means that the node has not been visited before. The black node u means that all nodes that connected with node u have already been visited. The gray node v means that there still exist white node that connected with node u.*

During the procedure of Breadth First Search, *Adjacent Linked List Structure* has been used. Also we keep three extra information for each node, which are *the Parent Node of the current Node, the distance between the current Node and the source Node s, and the color of the current Node.*

*structure Node {*

*int distance; ( Distance stands for the shortest distance between node s and v. )*

*Node Parent Node; ( Parent Node of current node v. )*

*string color; ( Color of current node v. )*

*};*

*Rule:*

1. Starts from the source node s, and visit all nodes in the Adjacent Linked List. As long as the node has been visited and the color of the node equals to white, then we just add the node v and the edge ( u, v ) into the Breadth First Search Tree.

*The distance of the node v = s.distance + 1*

*The parent node of node v = s*

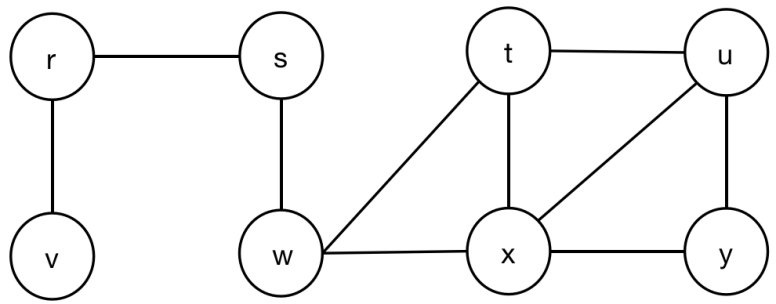
*The color of node v = gray*

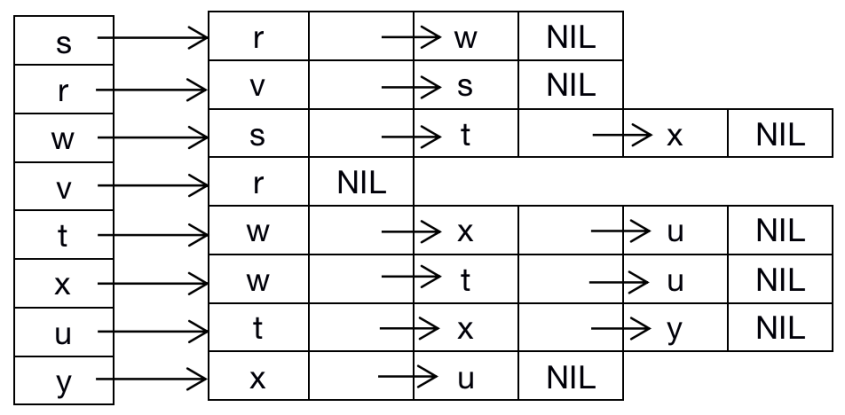
However, we do not really add the node v and the edge ( u, v ) into the Breadth First Search Tree. We just add the node into the queue and update its related information.

1. After finishing finding all nodes which have connected with node s, what we need to do is to mark the color of node s, which equals to black.
2. *In the queue, we need to pop out the next node from queue and finding all nodes which have been connected with it, just as the step 1, after finding all nodes v which connected with the node, then we need to update all related information related with the node, just as the same, the information including the distance of the node v, the parent node of node v, and the color of node v.*
3. After finishing all these step, just as the same step, update the color of the node as black.

*Example:*

The Graph is as below, and it totally has 8 nodes and 10 edges. Through finding all nodes which have been connected with node s, and mark the color of node s, we can finish Breadth First Search Algorithm on the Graph.

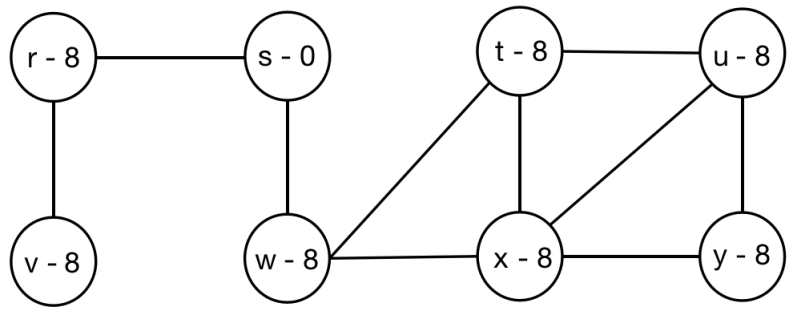




Initialize all nodes with extra information, including distance, parenting node, and color of node.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | Infinite | Infinite | Infinite | Infinite | Infinite | Infinite | Infinite | Infinite |
| Parent | NIL | NIL | NIL | NIL | NIL | NIL | NIL | NIL |
| Color | White | White | White | White | White | White | White | White |

Here, we use 0 to stand for 0 distance between source node s and source node s. Use 8 to stand for Infinite distance between node and source node s.



Prepare the queue to store each node into the queue. Also, initialize the queue with the size equals to the number of nodes.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|  |  |  |  |  |  |  |  |

1. Push the source node s into the queue.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| s |  |  |  |  |  |  |  |

1. Recursively run all below several steps:
2. Pop out the first node s.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|  |  |  |  |  |  |  |  |

1. Update the information table of source node s:

* Distance(s, s) = 0.
* Parent Node(s) = NIL.
* Color(s) = Gray.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | Infinite | Infinite | 0 | Infinite | Infinite | Infinite | Infinite | Infinite |
| Parent | NIL | NIL | NIL | NIL | NIL | NIL | NIL | NIL |
| Color | White | White | Gray | White | White | White | White | White |

3) Recursively visit all adjacent nodes of source node s.

1. Visit the first adjacent node r, and update all related information of node s.

* Distance(s, r) = 1.
* Parent Node(r) = s.
* Color(s) = Gray.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | Infinite | 1 | 0 | Infinite | Infinite | Infinite | Infinite | Infinite |
| Parent | NIL | s | NIL | NIL | NIL | NIL | NIL | NIL |
| Color | White | Gray | Gray | White | White | White | White | White |

1. Push the adjacent node r into queue.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| r |  |  |  |  |  |  |  |

1. Visit the second adjacent node w, and update all related information w.

* Distance(s, w) = 1.
* Parent Node(w) = s.
* Color(w) = Gray.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | Infinite | 1 | 0 | 1 | Infinite | Infinite | Infinite | Infinite |
| Parent | NIL | s | NIL | s | NIL | NIL | NIL | NIL |
| Color | White | Gray | Gray | Gray | White | White | White | White |

1. Push the adjacent node w into queue.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| r | w |  |  |  |  |  |  |

1. Update the color of the first Node s, s.color = Black.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | Infinite | 1 | 0 | 1 | Infinite | Infinite | Infinite | Infinite |
| Parent | NIL | s | NIL | s | NIL | NIL | NIL | NIL |
| Color | White | Gray | Black | Gray | White | White | White | White |

4) Visit all adjacent nodes of node r.

1. Pop out the first node r out of queue.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| w |  |  |  |  |  |  |  |

1. Recursively visit all adjacent nodes of node r:
   1. Visit the first adjacent node v, and update all related information.

* Distance(r, v) = Distance(s, v) + 1 = 2.
* Parent Node(v) = r.
* Color(v) = Gray.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | 2 | 1 | 0 | 1 | Infinite | Infinite | Infinite | Infinite |
| Parent | r | s | NIL | s | NIL | NIL | NIL | NIL |
| Color | Gray | Gray | Black | Gray | White | White | White | White |

* 1. Push the adjacent node v into the queue.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| w | v |  |  |  |  |  |  |

* 1. Update the color of Node r, r.color = black.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | 2 | 1 | 0 | 1 | Infinite | Infinite | Infinite | Infinite |
| Parent | r | s | NIL | s | NIL | NIL | NIL | NIL |
| Color | Gray | Black | Black | Gray | White | White | White | White |

1. Visit all adjacent nodes of node w.
2. Pop out the first node w out of queue.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| v |  |  |  |  |  |  |  |

1. Recursively visit all adjacent nodes of node w:
2. Visit the first adjacent node t, and update all related information.

* Distance(w, t) = Distance(s, w) + 1 = 2.
* Parent Node(t) = w.
* Color(t) = Gray.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | 2 | 1 | 0 | 1 | 2 | Infinite | Infinite | Infinite |
| Parent | r | s | NIL | s | w | NIL | NIL | NIL |
| Color | Gray | Black | Black | Gray | Gray | White | White | White |

b) Push the adjacent node t into the queue.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| v | t |  |  |  |  |  |  |

c) Visit the second adjacent node x, and update all related information.

* Distance(w, x) = Distance(s, w) + 1 = 2.
* Parent Node(x) = w.
* Color(x) = Gray.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | 2 | 1 | 0 | 1 | 2 | 2 | Infinite | Infinite |
| Parent | r | s | NIL | s | w | w | NIL | NIL |
| Color | Gray | Black | Black | Gray | Gray | Gray | White | White |

d) Push the adjacent node x into queue.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| v | t | x |  |  |  |  |  |

e) Update the color of Node w, w.color = Black.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | 2 | 1 | 0 | 1 | 2 | 2 | Infinite | Infinite |
| Parent | r | s | NIL | s | w | w | NIL | NIL |
| Color | Gray | Black | Black | Black | Gray | Gray | White | White |

6) Visit all adjacent nodes of node v:

1. Recursively visit all adjacent nodes of node v:
2. Pop out the first node v out of queue.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| t | x |  |  |  |  |  |  |

1. All adjacent nodes of node v have been visited.
2. Update the color of Node v, v.color = Black.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | 2 | 1 | 0 | 1 | 2 | 2 | Infinite | Infinite |
| Parent | r | s | NIL | s | w | w | NIL | NIL |
| Color | Black | Black | Black | Black | Gray | Gray | White | White |

7) Visit all adjacent nodes of node t:

1. Visit all adjacent nodes of node t:
   1. Pop out the first node t of queue.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| x |  |  |  |  |  |  |  |

* 1. Visit the first adjacent node u of node t, and update all related information:
* Distance(t, u) = Distance(t, s) + 1 = 2 + 1 = 3.
* Parent Node(u) = t.
* Color(u) = Gray.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | 2 | 1 | 0 | 1 | 2 | 2 | 3 | Infinite |
| Parent | r | s | NIL | s | w | w | t | NIL |
| Color | Black | Black | Black | Gray | Black | Gray | Gray | White |

* 1. Push the adjacent node u into the queue.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| x | u |  |  |  |  |  |  |

* 1. Update the color of node u, u.color = Black.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | 2 | 1 | 0 | 1 | 2 | 2 | 3 | Infinite |
| Parent | r | s | NIL | s | w | w | t | NIL |
| Color | Black | Black | Black | Gray | Black | Gray | Black | White |

1. Visit all adjacent nodes of node x:
2. Visit all adjacent nodes of node x:
   1. Pop out the first node x of queue.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| u |  |  |  |  |  |  |  |

b) Visit the first adjacent node y of node x, and update all related information:

* Distance(x, y) = Distance(x, s) + 1 = 2 + 1 = 3.
* Parent Node(y) = x.
* Color(y) = Gray.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | 2 | 1 | 0 | 1 | 2 | 2 | 3 | 3 |
| Parent | r | s | NIL | s | w | w | t | x |
| Color | Black | Black | Black | Gray | Black | Gray | Gray | Gray |

c) Push the adjacent node y into the queue:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| u | y |  |  |  |  |  |  |

d) Update the color of node x, x.color = Black.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | 2 | 1 | 0 | 1 | 2 | 2 | 3 | 3 |
| Parent | r | s | NIL | s | w | w | t | x |
| Color | Black | Black | Black | Gray | Black | Black | Gray | Gray |

1. Visit all adjacent nodes of node u:
2. Recursively visit all adjacent nodes of node u:
3. Pop out the first node u out of queue.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| y |  |  |  |  |  |  |  |

1. All adjacent nodes of node u have been visited.
2. Update the color of node u, u.color = Black.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | 2 | 1 | 0 | 1 | 2 | 2 | 3 | 3 |
| Parent | r | s | NIL | s | w | w | t | x |
| Color | Black | Black | Black | Black | Black | Black | Black | Gray |

1. Visit all adjacent nodes of node y:
2. Recursively visit all adjacent nodes of node y:
   1. Pop out the first node y out of queue.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|  |  |  |  |  |  |  |  |

* 1. All adjacent nodes of node y have been visited.
  2. Update the color of node y, y.color = Black.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | 2 | 1 | 0 | 1 | 2 | 2 | 3 | 3 |
| Parent | r | s | NIL | s | w | w | t | x |
| Color | Black | Black | Black | Black | Black | Black | Black | Black |

*Code:*

structure Node {

int distance; *// The distance between two nodes.*

string color; *// The color of node.*

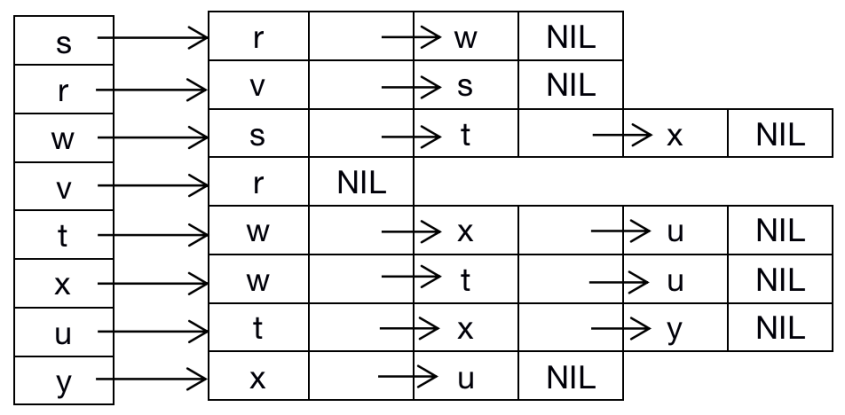
Node \* parent; *// The parent node of current node.*

Node\* next; *// The next field is used to link the next node.*

char value; *// The value field is used to record the value of node.*

};

*The structure of Graph g is given below, and it is passed as parameter in the following BFS function. The size of Graph g is also given.*



*The structure of hash table is given below, and it is used to store the final node. The size of hash table is the same as the number of nodes in the Graph g.*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Node v | Node r | Node s | Node w | Node t | Node x | Node u | Node y |
| Dist | Infinite | Infinite | Infinite | Infinite | Infinite | Infinite | Infinite | Infinite |
| Parent | NIL | NIL | NIL | NIL | NIL | NIL | NIL | NIL |
| Color | White | White | White | White | White | White | White | White |

*The structure of queue is given as below, and it is used to store the intermediate node, and further check all related nodes around it ---- therefore calls Breadth First Search Algorithm.Just as the same before, the size of queue is kept as the number of nodes.*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... | n |
|  |  |  |  |  |  |  |  |  |  |  |

*The size stands for the number of nodes in Graph, Queue and Hash Table.*

void BFS(Graph \* g, int size) {

/\*

\*\* The queue is used to store all nodes pointer of Graph g.

\*/

queue<Node \*> qu;

*/\**

*\*\* The unordered map is used to stored the final result. The key equals*

*\*\* to the value of all nodes ---- Here is the type of char. Of course, the value*

*\*\* of the unordered\_map is the pointer of Node.*

*\*/*

unordered\_map<char, Node \*> table;

*/\**

*\*\* The first step is to update the related information of all nodes in the Graph g.*

*\*/*

int i;

for ( i = 0; i < size; i++ ) {

g[ i ].distance = 8; *// Initialize the distance of the node in Graph g - here 8*  *// stands for infinite.*

g[ i ].color = “White”; *// Initialize White color of node in Graph g.*

g[ i ].parent = NULL; *// Initialize Parent node of node in Graph g.*

}

*/\**

*\*\* The second step is to initialize all related information of source node s in*

*\*\* Graph g.*

*\*/*

Node \* node = g[ 0 ];

node->distance = 0;

node->color = ‘Gray’;

node->parent = NULL;

*/\**

*\*\* The nextnode here stands for the next node of node.*

*\*/*

Node \* nextnode;

*/\**

*\*\* The third step is to push the node into queue.*

*\*/*

qu.push\_back(node);

while ( !qu ) {

node= qu.front();

qu.pop\_out();

*/\**

*\*\* Store the current node into hash table. The key is the value field of*

*\*\* node, and the value field here is the node itself.*

*\*/*

table[node->value] = node;

*/\**

*\*\* The color of next node should equal to “White”, otherwise it means that*

*\*\* all information of the node have been updated, the distance of the node*

*\*\* has already known.*

*\*/*

while ( node->next != NULL &&

node->next->color == “White” ) {

*/\**

*\*\* Initialize the content of nextnode.*

*\*/*

nextnode = node->next;

*/\**

*\*\* Initialize all information of nextnode.*

*\*/*

nextnode->distance = node->distance + 1;

nextnode->color = “Gray”;

nextnode->parent = node;

*/\**

*\*\* Push the nextnode into the queue.*

*\*/*

qu.push\_back(nextnode);

*/\**

*\*\* Meanwhile, save nextnode into Hash table.*

*\*/*

table[nextnode->val] = nextnode;

}

*/\**

*\*\* After recursively check all nodes of the current node, what we need to do here*

*\*\* is to update the color information of the node, and make it “Black”, which*

*\*\* means that the shortest routine of the node has been calculated.*

*\*/*

node->color = “Black”;

}

}

*The Shortest Routine:*

The Breadth First Search can be used to find all distance from source node to all accessible node. Here, we define the shortest routine distance d(u, v) as the least edge from source node u to definition node v. *The routine distance from node s to v is called as the shortest routine from s to v.*

However, if there has no routine from the node u to v, then d(u, v) = infinite.

*Key:*

*Breadth First Search can be used to calculate the Shortest Routine Distance from source node s to all accessible node v.*

*Routine Display - Display the Shortest Routine from Node s To Node v:*

*void Display ( Graph\* g, Node\* s, Node\* y ) {*

*if ( s == y ) {*

*print( s->value );*

*} else if ( y->priority != NULL ) {*

*Display( g, s, y->priority );*

*} else {*

*print( y->value );*

*}*

*}*