

目录

1. Graph search
 1. Breadth-first versus Depth-first search
2. Stack-Based Depth-First Search
 1. dfsStack.c (for connected graphs)
 2. dfsQuack() (disconnected graphs)
 3. The helper ADT IOmem
 4. Test 1
 5. Test 2
 6. Test 3: a disconnected graph
 7. Performance
3. Recursive Depth-First Search
 1. dfsRec.c (disconnected graphs)
 2. Test 1
 3. Test 2: a disconnected graph
 4. Global variables
4. Breadth First Search
 1. bfsQuack() (connected)

Graph search

Searching a graph can have many aims:

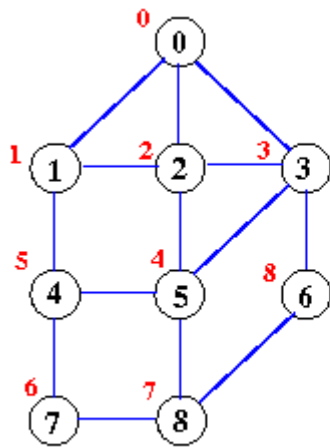
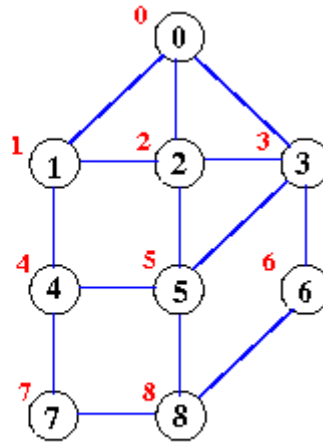
- can I reach every vertex in the graph (is it connected)?
- is one vertex reachable starting from some other vertex?
- what is the shortest path from vertex v to w ?
- which vertices are reachable from a vertex? (transitive closure)
- is there a cycle that passes through all the graph? (*tour*)
- is there a tree that links all vertices? (*spanning tree*)
 - what is the *minimum* spanning tree?
- are two graphs "equivalent"? (*isomorphism*)

A search is almost never 'random': it uses an underlying strategy:

- depth-first search DFS
- breadth-first search BFS

Breadth-first versus Depth-first search

Example:

**Depth-First Search****Breadth-First Search**

Order is given by the 'red' labels

- in this example the label ordering is breadth-first (layer by layer)

DFS descends by selecting the first available unvisited node

- select 0
- adjacent {1,2,3}
 - select 1
 - adjacent {2, 4}
 - select 2
 - adjacent {3, 5}
 - select 3
 - adjacent {5, 6}
 - select 5
 - adjacent {4, 8}
 - select 4
 - adjacent {7}
 - select 7
 - adjacent {8}
 - select 8
 - adjacent {6}
 - select 6
 - adjacent {} no sites left unvisited

BFS descends by systematically visiting the nodes in order of level

- select 0
- adjacent {1,2,3}
 - select 1
 - adjacent {4}
 - select 2
 - adjacent {5}
 - select 3
 - adjacent {6}
 - select 4
 - adjacent {7}
 - select 5
 - adjacent {8}
 - select 6
 - adjacent {}
 - select 7
 - adjacent {}

- select 8
- adjacent {} no sites left unvisited

These two 'strategies' actually use the same algorithm. They differ only in their use of data structure:

- DFS uses a stack
- BFS uses a queue

Here is the pseudo-algorithm for **Depth/Breadth**-first search:

```
push the root node onto a stack/queue
while (stack/queue is not empty) {
    pop a node from the stack/queue
    if (node is a goal node)
        return 'success'
    push all children of node onto the stack/queue
}
return 'failure'
```

If the aim is not to find a goal node, but to search the whole graph:

- leave out the conditional 'return' (i.e if (node is ...)
- return when complete

Stack-Based Depth-First Search

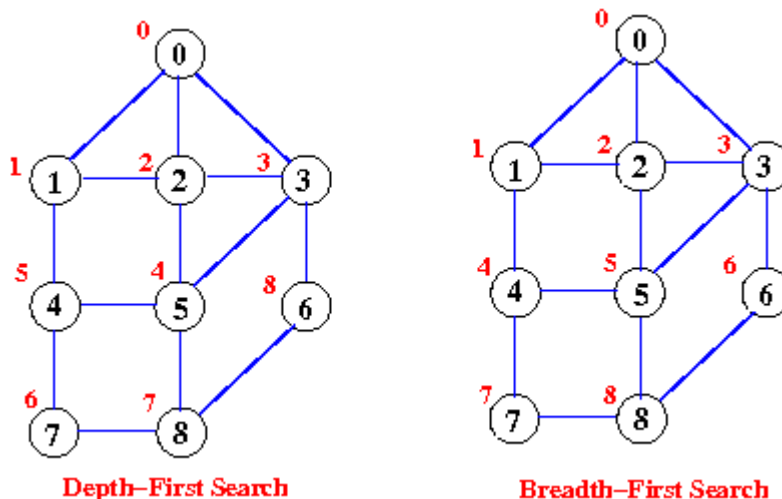
When searching we need to remember which nodes we've *visited*:

- to avoid cycles
- to make sure every node gets visited

Generally an array **visited[0 .. numVertices-1]** is used

- array indices correspond to vertices
- initialise all elements to -1, meaning unvisited
- when a vertex is visited, the index is set to its 'visit order' number
 - this is simply a 'count' that gets incremented each time a new node is visited

For example, here is the earlier graph again



The *visited* array starts as {-1,-1,-1,-1,-1,-1,-1,-1,-1}

We select the root **0** first

adjacent	visit	resulting visited array
<i>any node</i>	0	{ 0 , -1, -1, -1, -1, -1, -1, -1, -1 }
1 2 3	1	{ 0, 1 , -1, -1, -1, -1, -1, -1, -1 }
0 2 4	2	{ 0, 1, 2 , -1, -1, -1, -1, -1, -1 }
0 1 3 5	3	{ 0, 1, 2, 3 , -1, -1, -1, -1, -1 }
0 2 5 6	5	{ 0, 1, 2, 3, -1, 4 , -1, -1, -1 }
2 3 4 8	4	{ 0, 1, 2, 3, 5 , 4, -1, -1, -1 }
1 5 7	7	{ 0, 1, 2, 3, 5, 4, -1, 6 , -1 }
4 8	8	{ 0, 1, 2, 3, 5, 4, -1, 6, 7 }
5 6 7	6	{ 0, 1, 2, 3, 5, 4, 8 , 6, 7 }

Let's try a different starting vertex: this time start at vertex **5**:

adjacent	visit	resulting visited array
<i>any node</i>	5	{ -1, -1, -1, -1, -1, 0 , -1, -1, -1 }
2 3 4 8	2	{ -1, -1, 1 , -1, -1, 0, -1, -1, -1 }
0 1 3 5	0	{ 2 , -1, 1, -1, -1, 0, -1, -1, -1 }
1 2 3	1	{ 2, 3 , 1, -1, -1, 0, -1, -1, -1 }
0 2 4	4	{ 2, 3, 1, -1, 4 , 0, -1, -1, -1 }
1 5 7	7	{ 2, 3, 1, -1, 4, 0, -1, 5 , -1 }
4 8	8	{ 2, 3, 1, -1, 4, 0, -1, 5, 6 }
5 6 7	6	{ 2, 3, 1, -1, 4, 0, 7 , 5, 6 }
3 8	3	{ 2, 3, 1, 8 , 4, 0, 7, 5, 6 }

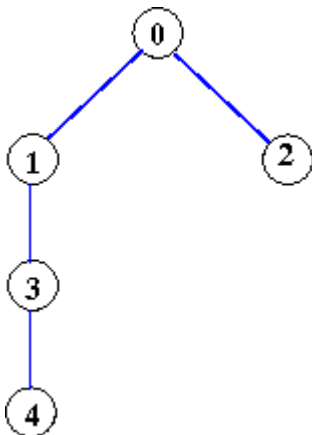
The array *visited[]* here **is the depth-first order**

- It says: { 2nd, 3rd, 1st, 8th, 4th, 0th, 7th, 5th, 6th }

The visited array indicates the order of the search.

Can anything go wrong during the traversal?

- *Yes, we can hit a deadend!*



choice	visit	resulting visited array
<i>any node</i>	0	{ 0 , -1, -1, -1, -1, }

1 2	1	{ 0, 1 , -1, -1, -1 }
0 3	3	{ 0, 1, -1, 2 , -1 }
1 4	4	{ 0, 1, -1, 2, 3 }
	<i>finished?</i>	

- 4 is a leaf node, we can go no further
- **there is still an unvisited vertex in the array**

How do we 'find' it?

- we need to **backtrack**
 - we go back to vertex 0, and then visit vertex 2
 - this is also a leaf node
 - ... but all nodes have been visited, so we are really finished this time

Final DFS path is visited = {0, 1, 4, 2, 3}

So we cannot expect DFS to visit every vertex in a single forward traversal

- we sometimes need to *backtrack*

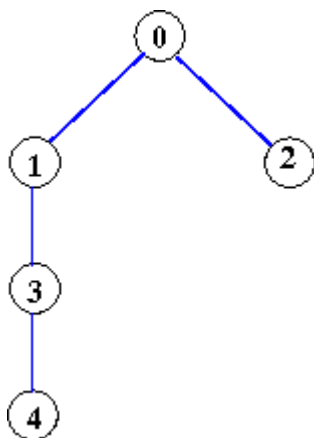
But how do we backtrack?

- we use a stack!
 - vertices are pushed onto the stack when we have 1 or more adjacent vertices to visit
 - to actually visit a vertex, we simply pop it from the stack
- *only when the stack is empty have we visited everyone!*

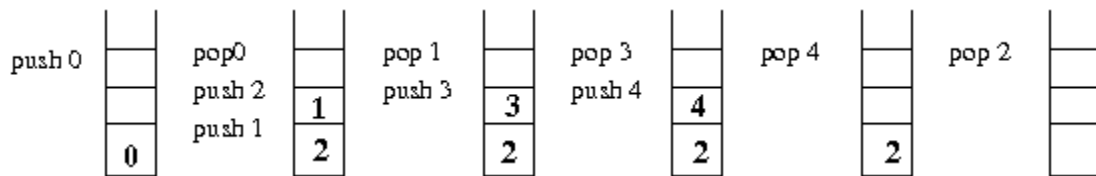
Using a stack in DFS means:

- when we *visit*, we *pop* the next vertex off the stack
- after a visit, we *push* the adjacent vertices onto the stack
- when we land on a leaf node, we cannot *push* any nodes onto the stack
 - we then *pop* a vertex instead
 - ... this is backtracking (to an earlier vertex)
- only when the stack is empty have we visited every vertex

Consider the above graph again:



The following stack operations are carried out:



dfsStack.c (for connected graphs)

切换行号显示

```

1 // dfsStack.c: traverse a graph using DFS and stacking (graph may
be disconnected)
2 // Compile using:
3 //     dcc -o dfsStack dfsStack.c IOmem.c GraphAM.c Quack.c
4 //
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include "Graph.h"
8 #include "Quack.h"
9 #include "IOmem.h"
10
11 #define STARTVERTEX 0 // start the depth-first search at this
vertex
12
13 void dfsQuack(Graph, Vertex, int);
14
15 int main (void) {
16     int numV;
17     if ((numV = readNumV()) > 0) {
18         Graph g = newGraph(numV);
19         if (readBuildGraph(g)) {
20             showGraph(g);
21             dfsQuack(g, STARTVERTEX, numV);
22         }
23         g = freeGraph(g);
24         g = NULL;
25     }
26     else {
27         printf("Error in reading #number\n");
28         return EXIT_FAILURE;
29     }
30     return EXIT_SUCCESS;
31 }
32
33 //HANDLES CONNECTED GRAPHS ONLY
34 void dfsQuack(Graph g, Vertex v, int numV) {
35     int *visited = mallocArray(numV);
36     Quack s = createQuack();
37     push(v, s);
38     showQuack(s);
39     int order = 0;
40     while (!isEmptyQuack(s)) {
41         v = pop(s);
42         if (visited[v] == UNVISITED) { // we visit only unvisited
vertices
43             printArray("Visited: ", visited, numV);
44             visited[v] = order++;
45             for (Vertex w = numV - 1; w >= 0; w--) { //push adjacent
vertices
46                 if (isEdge(newEdge(v,w), g)) { // ... in
reverse order
47                     push (w, s); // ... onto the
stack
48                 }
49             }
50         }
51         showQuack(s);

```

```

52     }
53     printArray("Visited: ", visited, numV);
54     free(visited);
55     return;
56 }

```

The loop formed by lines 40-52 eventually empty the stack

- ... suggesting that the traversal is complete ...
 - ... and that all nodes have been visited

This may not be true.

If the graph is disconnected then *isEdge()* on line 46 is insufficient ...

- ... there will be no edges to a disconnected part of the graph

We need to also check that every vertex has been visited before we return from the function

This involves:

1. looking for a vertex in *visited[]* that is -1
 - this vertex has not yet been visited before
2. *push* this vertex onto the stack
3. start a new traversal with this node as 'root'

The code of the 'disconnected version' of *dfsQuack()* is as follows:

dfsQuack() (disconnected graphs)

切换行号显示

```

1  // HANDLES DISCONNECTED GRAPHS
2  void dfsQuack(Graph g, Vertex rootv, int numV) {
3      int *visited = mallocArray(numV);
4      Quack s = createQuack();
5      push(rootv, s);
6      showQuack(s);
7      int order = 0;
8      int allVis = 0;
9      while (!allVis) {          // as long as there are unvisited
vertices
10         while (!isEmptyQuack(s)) {
11             Vertex v = pop(s);
12             if (visited[v] == UNVISITED) {
13                 printArray("Visited: ", visited, numV); // debug
14                 visited[v] = order++;
15                 for (Vertex w = numV - 1; w >= 0; w--) {
16                     if (isEdge(newEdge(v,w), g)) {
17                         push (w, s);
18                     }
19                 }
20             }
21             showQuack(s);
22         }
23         // stack is empty, but are we finished?
24         allVis = 1;
25         for (Vertex w = 0; w < numV && allVis; w++) {
26             if (visited[w] == UNVISITED) {
27                 printf("Graph is DISCONNECTED\n"); // debug
28                 allVis = 0;      // found an unvisited vertex
29                 push(w, s);      // push vertex onto stack

```

```

30         showQuack(s);
31     }
32 }
33 }
34 printArray("Visited: ", visited, numV);
35 free(visited);
36 return;
37 }

```

The helper ADT IOmem

Input/output and memory management is controlled by an ADT called **IOmem**. Its interface is:

切换行号显示

```

1 // IOmem.h
2 // Interface to IOmem ADT that reads input data, builds and print
  graphs and manages memory.
3
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int readNumV();           // read an int (numV) from stdin
8 int readBuildGraph(Graph); // read int pairs from stdin
9 int* mallocArray(int);    // malloc an array of length int *
  sizeof(int)
10 void printArray(char *, int *, int); // print an int array of
  length int
11

```

This ADT allows the amount of graph search code to be kept minimal.

We can now compile the graph search algorithm with the *Graph*, *Quack* and *IOmem* ADTs:

- we can either the *GraphAM* or *GraphAL* ADTs

```
gcc dfsStack.c IOmem.c GraphAM.c Quack.c
```

or

```
gcc dfsStack.c IOmem.c GraphAL.c Quack.c
```

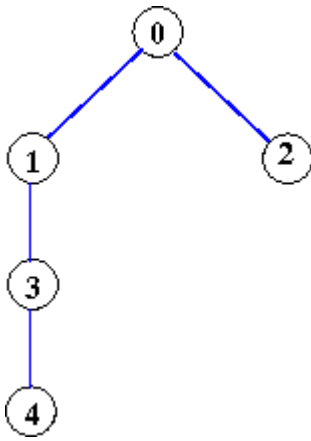
- also a choice between the array-based ADT *Quack* and linked-list version *QuackLL*
- in total, 4 combinations of *Graph* and *Quack* ADTs possible!

Test 1

The input file we use is:

```
#5
0 1 0 2 1 3 3 4
```

which corresponds to the simple graph we saw before:



Executing *a.out* using this input file results in the following:

```

V=5, E=4
<0 1> <0 2>
<1 0> <1 3>
<2 0>
<3 1> <3 4>
<4 3>
Quack: <<0>>
Visited: {-1, -1, -1, -1, -1}
Quack: <<1, 2>>
Visited: {0, -1, -1, -1, -1}
Quack: <<0, 3, 2>>
Quack: <<3, 2>>
Visited: {0, 1, -1, -1, -1}
Quack: <<1, 4, 2>>
Quack: <<4, 2>>
Visited: {0, 1, -1, 2, -1}
Quack: <<3, 2>>
Quack: <<2>>
Visited: {0, 1, -1, 2, 3}
Quack: <<0>>
Quack: << >>
Visited: {0, 1, 4, 2, 3}

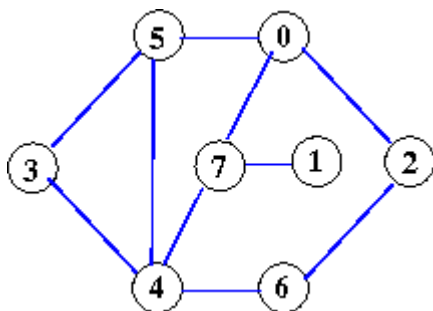
```

Here we see:

- the starting vertex 0 is pushed
- 0 is popped and its neighbours 1 and 2 are pushed
 - `visited[0] = 0`
- 1 is popped and its neighbours 0 and 3 are pushed
 - `visited[1] = 1`
- 0 is popped and ignored as it is in array *visited*
- 3 is popped and its neighbours 1 and 4 are pushed
 - `visited[3] = 2`
- 1 is popped and is ignored
- 4 is popped and its neighbour 3 is pushed
 - `visited[4] = 3`
- 3 is popped and is ignored
- 2 is popped and its neighbour 0 is pushed
 - `visited[2] = 4`
- 0 is popped
- *quack is empty*

Test 2

What about a more substantial graph:



It is represented by the input data:

#8
0 2 0 5 0 7 2 6 1 7 4 7 4 6 4 3 3 5 4 5

If we want to do a DFS starting from vertex 0 (remember: a *#define* in the code):

```
V=8, E=10
<0 2> <0 5> <0 7>
<1 7>
<2 0> <2 6>
<3 4> <3 5>
<4 3> <4 5> <4 6> <4 7>
<5 0> <5 3> <5 4>
<6 2> <6 4>
<7 0> <7 1> <7 4>
Quack: <<0>>
Visited: {-1, -1, -1, -1, -1, -1, -1, -1}
Quack: <<2, 5, 7>>
Visited: {0, -1, -1, -1, -1, -1, -1, -1}
Quack: <<0, 6, 5, 7>>
Quack: <<6, 5, 7>>
Visited: {0, -1, 1, -1, -1, -1, -1, -1}
Quack: <<2, 4, 5, 7>>
Quack: <<4, 5, 7>>
Visited: {0, -1, 1, -1, -1, -1, 2, -1}
Quack: <<3, 5, 6, 7, 5, 7>>
Visited: {0, -1, 1, -1, 3, -1, 2, -1}
Quack: <<4, 5, 5, 6, 7, 5, 7>>
Quack: <<5, 5, 6, 7, 5, 7>>
Visited: {0, -1, 1, 4, 3, -1, 2, -1}
Quack: <<0, 3, 4, 5, 6, 7, 5, 7>>
Quack: <<3, 4, 5, 6, 7, 5, 7>>
Quack: <<4, 5, 6, 7, 5, 7>>
Quack: <<5, 6, 7, 5, 7>>
Quack: <<6, 7, 5, 7>>
Quack: <<7, 5, 7>>
Visited: {0, -1, 1, 4, 3, 5, 2, -1}
Quack: <<0, 1, 4, 5, 7>>
Quack: <<1, 4, 5, 7>>
Visited: {0, -1, 1, 4, 3, 5, 2, 6}
Quack: <<7, 4, 5, 7>>
Quack: <<4, 5, 7>>
Quack: <<5, 7>>
Quack: <<7>>
Quack: << >>
Visited: {0, 7, 1, 4, 3, 5, 2, 6}
```

Test 3: a disconnected graph

The third test is a disconnected graph *exdiscon.inp*:

```
#6
0 1 0 2
1 2
3 4 3 5
4 5
```

In essence, this graph consists of 2 subgraphs, which are triangles.

Executing *a.out* using this input file results in the following:

```
V=6, E=6
<0 1> <0 2>
<1 0> <1 2>
<2 0> <2 1>
<3 4> <3 5>
<4 3> <4 5>
<5 3> <5 4>
Quack: <<0>>
Visited: {-1, -1, -1, -1, -1, -1}
Quack: <<1, 2>>
Visited: {0, -1, -1, -1, -1, -1}
Quack: <<0, 2, 2>>
Quack: <<2, 2>>
Visited: {0, 1, -1, -1, -1, -1}
Quack: <<0, 1, 2>>
Quack: <<1, 2>>
Quack: <<2>>
Quack: << >>
Graph is disconnected
Quack: <<3>>
Visited: {0, 1, 2, -1, -1, -1}
Quack: <<4, 5>>
Visited: {0, 1, 2, 3, -1, -1}
Quack: <<3, 5, 5>>
Quack: <<5, 5>>
Visited: {0, 1, 2, 3, 4, -1}
Quack: <<3, 4, 5>>
Quack: <<4, 5>>
Quack: <<5>>
Quack: << >>
Visited: {0, 1, 2, 3, 4, 5}
```

Notice that after traversing the first 'triangle', the program detects the program is disconnected.

- it then starts at the lowest unvisited vertex (3) and traverses the second triangle.

Performance

- number of pushes and pops
 - should be the same (stack is empty at the end)
- number of pushes of a vertex v = vertex degree of v
- total number of pushes
 - = sum of all the vertex degrees of vertices v in the graph

The sum of vertex degrees is equal to twice the number of edges.

- this means the complexity is linear in the number of edges, $O(E)$
 - **what does this mean? ...**
 - *how many edges are there?*

- the worst case is a dense graph: $E = V*(V-1)/2$
- so the complexity is quadratic in V : i.e. $O(V^2)$
- if it is sparse, then it will be less than quadratic
- often said that DFS is *linear in the size of the graph* ...
 - ... where 'size' is the number of edges
 - ... which is another way of saying *quadratic in the number of vertices*

Recursive Depth-First Search

DFS above used our own stack to 'remember' which path it was traversing and do backtracking.

The system also has a stack, called a *call stack*, which is used to execute functions

- a function call causes a *function frame* to be pushed onto the call stack
 - ... upon a function return, the *frame* is popped off the call stack

This works even for recursive functions of course.

The call stack can be used instead of the 'stack' ADT we used above

- so when we compile we do not need the ADT quack

It works by recursion:

- a function *dfsR()* calls itself recursively ...
- in essence adjacent vertices are being *pushed* onto the system 'call' stack
 - the *for-loop* in *dfsR()* is over all all unvisited adjacent vertices
 - in the *for-loop*, *dfsR()* is called for every vertex
 - these calls *stack up* as you descend down the tree

dfsRec.c (disconnected graphs)

切换行号显示

```

1 // dfsRec.c: traverse a graph using DFS (graph may be
disconnected)
2 // Compile using:
3 //     dcc -o dfsRec dfsRec.c IOmem.c GraphAM.c
4 //
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include "Graph.h"
8 #include "IOmem.h"
9
10 #define START 0 // the start vertex
11
12 void dfs(Graph, Vertex, int);
13 void dfsR(Graph, Vertex, int, int *, int *);
14
15 int main(void) {
16     int numV;
17     if ((numV = readNumV()) >= 0) {
18         Graph g = newGraph(numV);
19         if (readBuildGraph(g)) {
20             showGraph(g);
21             dfs(g, START, numV); // DEPTH-FIRST SEARCH FROM START
22         }
23         g = freeGraph(g);

```

```

24     g = NULL;
25 }
26 else {
27     return EXIT_FAILURE;
28 }
29 return EXIT_SUCCESS;
30 }
31
32 void dfs(Graph g, Vertex rootv, int numV) {'wrapper' for
recursive dfs
33     int *visited = mallocArray(numV); // ... handles disconnected
graphs
34     int order = 0;
35     Vertex startv = rootv;           // this is the starting
vertex
36     int allVis = 0;                  // assume not all visited
37     while (!allVis) {                // as long as there are
vertices
38         dfsR(g, startv, numV, &order, visited);
39         allVis = 1;                  // are all visited now?
40         for (Vertex w = 0; w < numV && allVis; w++) { // look for
more
41             if (visited[w] == UNVISITED) {
42                 printf("Graph is disconnected\n"); // debug
43                 allVis = 0;                // found an unvisited vertex
44                 startv = w;                // next loop dfsR this vertex
45             }
46         }
47     }
48     printArray("Visited: ", visited, numV);
49     free(visited);
50     return;
51 }
52
53 void dfsR(Graph g, Vertex v, int numV, int *order, int *visited) {
54     visited[v] = *order;              // records the order of
visit
55     *order = *order+1;
56     for (Vertex w = 0; w < numV; w++) {
57         if (isEdge(newEdge(v,w), g) && visited[w]==UNVISITED) {
58             dfsR(g, w, numV, order, visited);
59         }
60     }
61     return;
62 }

```

Here the function *dfs()* is called by *main()*

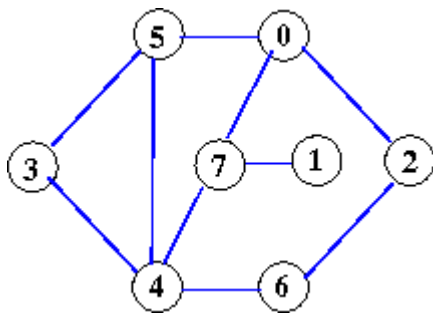
- this function is a *wrapper*
- it does 'housekeeping' (initialising *order* and the array *visited*)
- it calls the recursive function *dfsR()*

Remember: in the stack version the main function called *dfsQuack()*

- it does 'housekeeping' (initialising *order* and the array *visited*)
- *pops* and *pushes* off/on the quack until the quack is empty

Test 1

Let's run this recursive DFS on the graph we had above:



Remember, it is represented by the input data:

```
#8
0 2 0 5 0 7 2 6 1 7 4 7 4 6 4 3 3 5 4 5
```

Compiling:

```
gcc -o dfsRec dfsRec.c IOmem.c GraphAM.c
```

notice, no *Quack* ADT, and executing

```
V=8, E=10
<0 2> <0 5> <0 7>
<1 7>
<2 0> <2 6>
<3 4> <3 5>
<4 3> <4 5> <4 6> <4 7>
<5 0> <5 3> <5 4>
<6 2> <6 4>
<7 0> <7 1> <7 4>
Visiting vertex 0 in order 0
Visiting vertex 2 in order 1
Visiting vertex 6 in order 2
Visiting vertex 4 in order 3
Visiting vertex 3 in order 4
Visiting vertex 5 in order 5
Visiting vertex 7 in order 6
Visiting vertex 1 in order 7
Visited: {0, 7, 1, 4, 3, 5, 2, 6}
```

Comparing that with the stack version, the last lines are shown again below:

```
.
.
.
Visited: {0, -1, 1, 4, 3, 5, 2, 6}
Quack: <<7, 4, 5, 7>>
Quack: <<4, 5, 7>>
Quack: <<5, 7>>
Quack: <<7>>
Quack: << >>
Visited: {0, 7, 1, 4, 3, 5, 2, 6}
```

In summary:

- we've seen 2 versions of depth-first search:
 - an explicit stack version *dfsStack.c* that uses a *Quack* ADT
 - a call-stack version *dfsRec.c* that uses recursion

You could argue that the stack version:

- requires much less system resources (no recursion)
- does backtracking in an *iterative* manner, so will be much faster

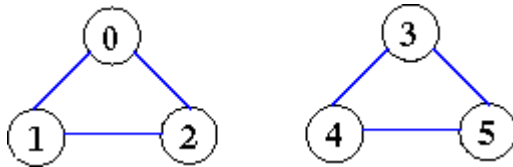
Test 2: a disconnected graph

We saw in *dfsRec.c* that the recursion handles disconnected graphs

Let's check that.

```
#6
0 1 0 2 1 2 3 4 3 5 4 5
```

corresponding to:



and assuming the starting vertex is 0, then *dfsRec* produces:

```
V=6, E=6
<0 1> <0 2>
<1 0> <1 2>
<2 0> <2 1>
<3 4> <3 5>
<4 3> <4 5>
<5 3> <5 4>
Visiting vertex 0 in order 0
Visiting vertex 1 in order 1
Visiting vertex 2 in order 2
Graph is disconnected
Visiting vertex 3 in order 3
Visiting vertex 4 in order 4
Visiting vertex 5 in order 5
Visited: {0, 1, 2, 3, 4, 5}
```

Notice:

- the first (sub) graph's DFS search begins at vertex 0
- the second (sub) graph's DFS search begins at vertex 3

Global variables

Crucial in both versions is the array *visited[]* and integer variable *order*

- *visited[]* records unvisited vertices and the *order* of visiting
- stop cycles occurring (remember, we are dealing with graphs)

In *dfsRec.c*:

- *visited[]* and *order* are initialised in the 'wrapper'
- are parameters to the recursive call
 - a vertex is visited on every call so they will change every call

切换行号显示

```
1 void dfsR(Graph g, Vertex v, int numV, int *order, int
*visited) {
```

```

2      visited[v] = *order;           // records the order
of visit
3      *order = *order+1;
4      . . .

```

It is easier to implement them as global variables (most people do this)

- ... don't need to pass them to *dfsR()*

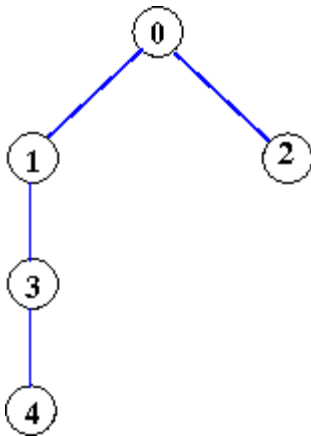
GraphSearchDFSglobal

Breadth First Search

All adjacent vertices are visited before moving to another vertex

- each level of vertices is visited before the next level's vertices are considered

For example:



1. visit vertex 0
2. visit vertex 1 and 2
3. visit vertex 3
4. visit vertex 4

In essence, the vertices are processed ***in order*** (top to bottom, left to right)

- DFS used a stack:
 - we *push* all the adjacent vertices of a vertex onto a stack
 - we *pop* off the top until the stack is empty (this allows backtracking)
 - vertices are being pushed and popped off the 'same place'
- BFS instead uses a queue:
 - we *push* all the adjacent vertices onto a queue (not a stack)
 - ... so they get added to the bottom
 - we *pop* off the top
 - all adjacent vertices are handled 'together'
- the change from stack to queue is almost trivial:
 - just 4 changes

bfsQuack() (connected)

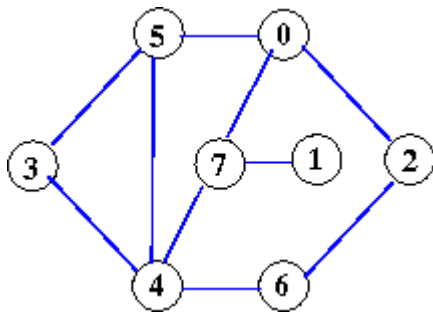
切换行号显示


```

1 void bfsQuack(Graph g, Vertex v, int numV) { //name change
2     int *visited = mallocArray(numV);
3     Quack q = createQuack();
4     qush(v, q); //qush, not push
5     showQuack(q);
6     int order = 0;
7     while (!isEmptyQuack(q)) {
8         v = pop(q);
9         if (visited[v] == UNVISITED) {
10             printf("Visit %d\n", v); // debug
11             visited[v] = order++;
12             for (Vertex w = 0; w < numV; w++) { //vertex order
13                 if (isEdge(newEdge(v,w), g)) {
14                     qush(w, q); //qush, not push
15                 }
16             }
17         }
18         showQuack(q);
19     }
20     printArray("Visited: ", visited, numV);
21     free(visited);
22     makeEmptyQuack(q);
23     return;
24 }

```

Test it using the graph:



This is the file *exsedg.inp*.

```

#8
0 2 0 5 0 7 2 6 1 7 4 7 4 6 4 3 3 5 4 5

```

Compile and execute:

```

dcc bfsQueue.c IOmem.c GraphAM.c Quack.c
./a.out < exsedg.inp
Quack: <<0>>
Visit 0
Quack: <<2, 5, 7>>      <=== 2,5,7 qushed
Visit 2
Quack: <<5, 7, 0, 6>>    <=== 0,6 qushed
Visit 5
Quack: <<7, 0, 6, 0, 3, 4>> <=== 0,3,4 qushed
Visit 7
Quack: <<0, 6, 0, 3, 4, 0, 1, 4>> <==== 0,1,4, qushed
Quack: <<6, 0, 3, 4, 0, 1, 4>>
Visit 6
Quack: <<0, 3, 4, 0, 1, 4, 2, 4>>
Quack: <<3, 4, 0, 1, 4, 2, 4>>
Visit 3
Quack: <<4, 0, 1, 4, 2, 4, 4, 5>>
Visit 4
Quack: <<0, 1, 4, 2, 4, 4, 5, 3, 5, 6, 7>>

```

```
Quack: <<1, 4, 2, 4, 4, 5, 3, 5, 6, 7>>
Visit 1
Quack: <<4, 2, 4, 4, 5, 3, 5, 6, 7, 7>>
Quack: <<2, 4, 4, 5, 3, 5, 6, 7, 7>>
Quack: <<4, 4, 5, 3, 5, 6, 7, 7>>
Quack: <<4, 5, 3, 5, 6, 7, 7>>
Quack: <<5, 3, 5, 6, 7, 7>>
Quack: <<3, 5, 6, 7, 7>>
Quack: <<5, 6, 7, 7>>
Quack: <<6, 7, 7>>
Quack: <<7, 7>>
Quack: <<7>>
Quack: << >>
Visited: {0, 7, 1, 5, 6, 2, 4, 3}
```

Note:

- 0 is a level-0 vertex
- 2 5 7 are level-1
- 6 3 4 1 are level-2

DFS generated the following:

- $Visited[] = \{0, 7, 1, 4, 3, 5, 2, 6\}$

GraphSearch (2019-07-25 17:36:42由AlbertNymeyer编辑)