目录

Graph search

Searching a graph can have many aims:

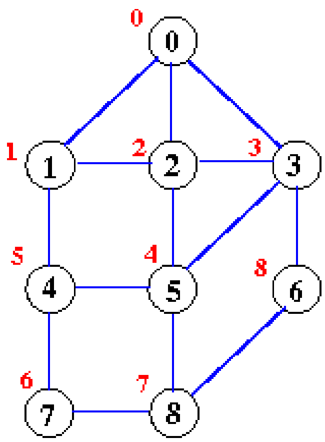
- can I reach every vertex in the graph (is it connected)?
- is one vertex reachable starting from some other vertex?
- what is the shortest path from vertex *v* to *w*?
- which vertices are reachable from a vertex? (transitive closure)
- is there a cycle that passes through all the graph? (*tour*)
-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

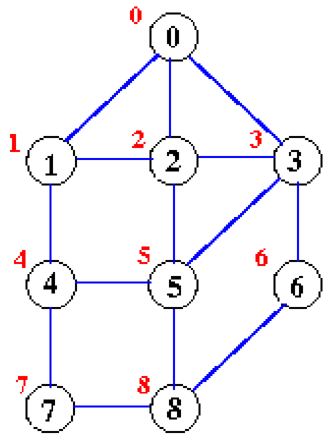
- depth-first search DFS
- breadth-first search BFS

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

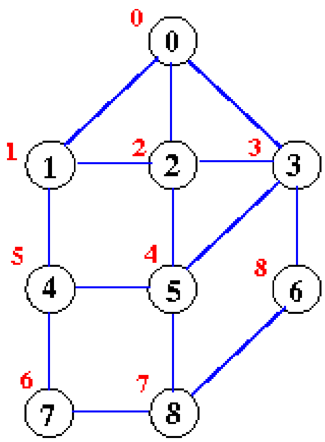
When searching we need to remember which nodes we've *visited*:

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

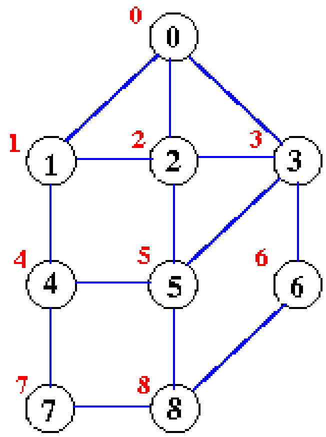
Generally an array **visited[0 .. numVertices-1]** is used

- 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



Depth–First Search



Breadth–First Search

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
any node	**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
any node	**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}

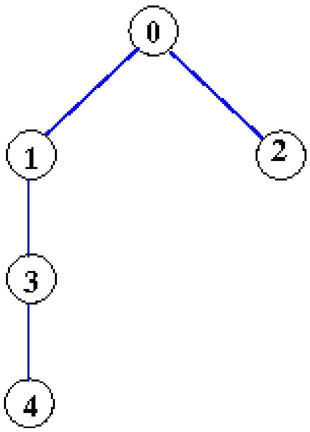
The array *visited[]* here **is the depth-first order**

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

```
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
any node	**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**}
	finished?	

- 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*

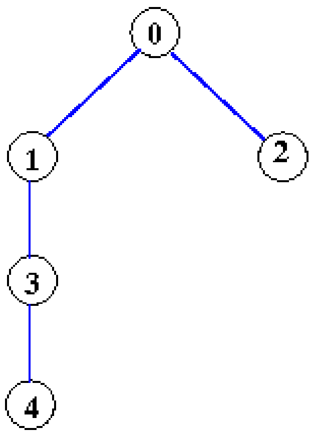
But how do we backtrack?

- 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!*

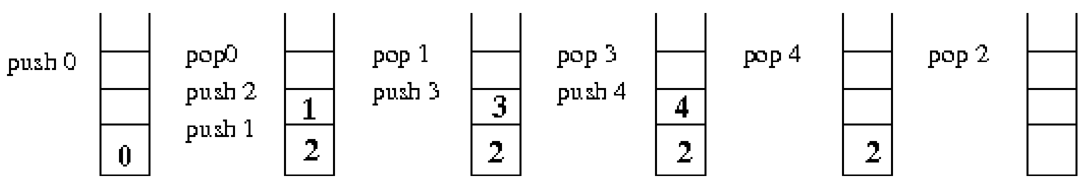
Using a stack in DFS means:

- 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:



Code:

切换行号显示

```
// dfsStack.c: traverse a graph using DFS and stacking (graph may be disconnected)
// Compile using:
//       dcc -o dfsStack dfsStack.c IOmem.c GraphAM.c Quack.c
//
#include <stdio.h>
#include <stdlib.h>
#include "Graph.h"
#include "Quack.h"
#include "IOmem.h"

#define STARTVERTEX 0   // start the depth-first search at this vertex

void dfsQuack(Graph, Vertex, int);
```

```

int main (void) {
    int numV;
    if ((numV = readNumV()) > 0) {
        Graph g = newGraph(numV);
        if (readBuildGraph(g)) {
            showGraph(g);
            dfsQuack(g, STARTVERTEX, numV);
        }
        g = freeGraph(g);
        g = NULL;
    }
    else {
        printf("Error in reading #number\n");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}

void dfsQuack(Graph g, Vertex v, int numV) {
   int *visited = mallocArray(numV);
   Quack s = createQuack();
   push(v, s);
   showQuack(s);
   int order = 0;
   while (!isEmptyQuack(s)) {
      v = pop(s);
      if (visited[v] == UNVISITED) {   // we visit only unvisited vertices
         printArray("Visited: ", visited, numV);
         visited[v] = order++;
         for (Vertex w = numV - 1; w >= 0; w--) { // push adjacent vertices
            if (isEdge(newEdge(v,w), g)) {        // ... in reverse order
               push (w, s);                       // ... onto the stack
            }
         }
      }
      showQuack(s);
   }
   printArray("Visited: ", visited, numV);
   free(visited);
   return;
}
```

The helper ADT IOmem

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

切换行号显示

```
// IOmem.h
// Interface to IOmem ADT that reads input data, builds and print graphs and manages memory.

#include <stdio.h>
#include <stdlib.h>

int readNumV();                // read an int (numV) from stdin
int readBuildGraph(Graph);     // read int pairs from stdin
int* mallocArray(int);         // malloc an array of length int * sizeof(int)
void printArray(char *, int *, int); // print an int array of length int

```

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

  ```
  dcc dfsStack.c IOmem.c GraphAM.c Quack.c
  ```

  or

  ```
  dcc dfsStack.c IOmem.c GraphAL.c Quack.c
  ```

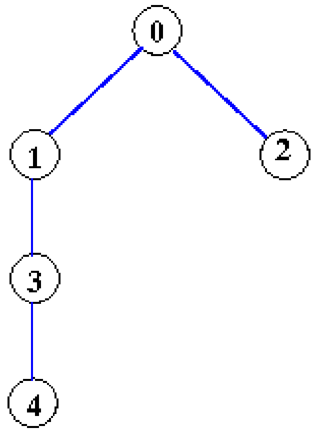
- also a choice between the array-based ADT *Quack* and linked-list version *QuackLL*
- in total, 4 combinations of *Graph* and *Quack* ADTs possible!

Testing Stack-Based Depth-First Search

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}
```

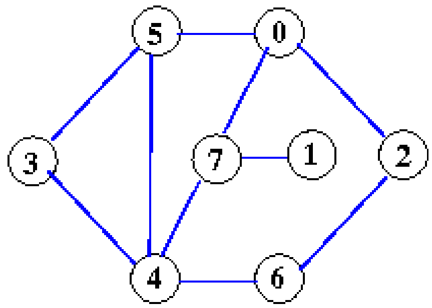
Here we see:

- 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*

Note the role of the stack here: it enables backtracking.

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}
```

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

切换行号显示

```
// dfsRec.c: traverse a graph using DFS (graph may be disconnected)
// Compile using:
//       dcc -o dfsRec dfsRec.c IOmem.c GraphAM.c
//
#include <stdio.h>
#include <stdlib.h>
#include "Graph.h"
#include "IOmem.h"

void dfs(Graph, Vertex, int);
void dfsR(Graph, Vertex, int, int *, int *);

int main(void) {
   int numV;
   if ((numV = readNumV()) >= 0) {
       Graph g = newGraph(numV);
       if (readBuildGraph(g)) {
          showGraph(g);
          dfs(g, 0, numV); // DEPTH-FIRST SEARCH FROM NODE 0
       }
       g = freeGraph(g);
       g = NULL;
   }
   else {
      return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
}

void dfs(Graph g, Vertex v, int numV) { // a 'wrapper' for recursive dfs
   int *visited = mallocArray(numV);  // ... handles disconnected graphs
   int order = 0;
   Vertex newv = v;                   // this is the starting vertex
   int allVis = 0;                    // assume not all visited
```

```
    while (!allVis) {                       // as long as there are vertices
        dfsR(g, newv, numV, &order, visited);
        allVis = 1;                         // are all visited now?
        for (Vertex w = 0; w < numV && allVis; w++) { // look for more
           if (visited[w] == UNVISITED) {
              printf("Graph is disconnected\n"); // debug
              allVis = 0;                   // found an unvisited vertex
              newv = w;                     // next loop dfsR this vertex
           }
        }
    }
    printArray("Visited: ", visited, numV);
    free(visited);
    return;
}

void dfsR(Graph g, Vertex v, int numV, int *order, int *visited) {
    visited[v] = *order;                     // records the order of visit
    printf("Visiting vertex %d in order %d\n", v, *order);
    *order = *order+1;
    for (Vertex w = 0; w < numV; w++) {
        if (isEdge(newEdge(v,w), g) && visited[w]==UNVISITED) {
           dfsR(g, w, numV, order, visited);
        }
    }
    return;
}
```

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

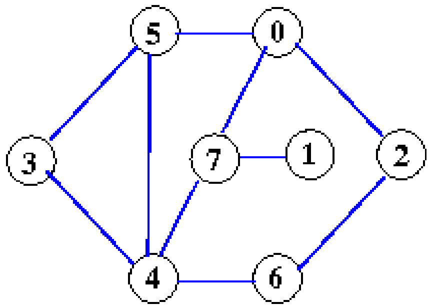
- this function is a *wrapper*
- it does 'housekeeping' (initialising *order* and the array *visited*)
- it calls the recursive function *dfsR()*

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

Testing recursive DFS

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:

```
dcc -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 shown 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

Globally visited

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)

Almost always implemented as global variables.

- For example, in recursive DFS, *dfsRec.c*
  - *visited[]* and *order* are initialised in the 'wrapper'
  - if global variables are used, they

GraphSearchDFSglobal

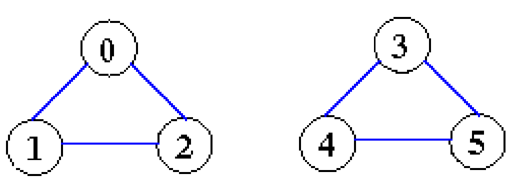
Testing a disconnected graph

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:

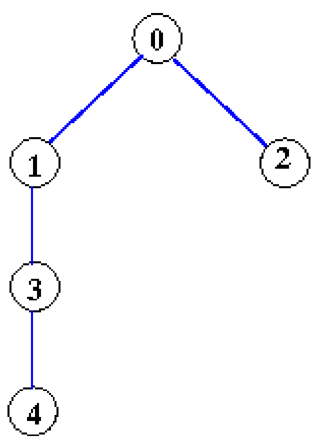
- the first (sub) graph's DFS search begins at vertex 0
- the second (sub) graph's DFS search begins at vertex 3

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 pushed all the adjacent vertices of a vertex onto a stack
  - so we would remember the vertices we need to 'still visit'
- BFS instead uses a queue:
  - we push all the adjacent vertices of a vertex onto a queue (not a stack)
  - but we visit them in the order they occur (as we must in a queue)
- the change in the *quack* version is trivial:

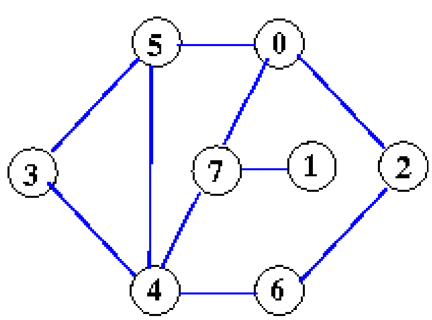
- just 4 changes

切换行号显示

```
void bfsQuack(Graph g, Vertex v, int numV) { //name change
   int *visited = mallocArray(numV);
   Quack q = createQuack();
   qush(v, q);                              //qush, not push
   showQuack(q);
   int order = 0;
   while (!isEmptyQuack(q)) {
      v = pop(q);
      if (visited[v] == UNVISITED) {
         printf("\t\t\t ... visit %d\n", v);
         visited[v] = order++;
         Vertex w;
         for (w = 0; w < numV; w++) {      //vertex order
            if (isEdge(newEdge(v,w), g)) {
               qush(w, q);                  //qush, not push
            }
         }
      }
      showQuack(q);
   }
   printArray("Visited: ", visited, numV);
   free(visited);
   makeEmptyQuack(q);
   return;
}
```

What about the graph we considered above for DFS 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
```



Starting at vertex 0, what did DFS do:

- *Visited[]* = {0, 7, 1, 4, 3, 5, 2, 6}
- corresponds to the vertices: 0 2 6 4 3 5 7 1

What does *bfs()* do:

```
Quack: <<0>>                           <== start node
Visited: {-1, -1, -1, -1, -1, -1, -1, -1}
Quack: <<2, 5, 7>>                     <== 2,5,7 pushed
Visited: {0, -1, -1, -1, -1, -1, -1, -1}
Quack: <<5, 7, 0, 6>>                  <== 0,6 quashed
Visited: {0, -1, 1, -1, -1, -1, -1, -1}
Quack: <<7, 0, 6, 0, 3, 4>>            <== 0,3,4 pushed
Visited: {0, -1, 1, -1, -1, 2, -1, -1}
Quack: <<0, 6, 0, 3, 4, 0, 1, 4>>      <== 0,1,4 pushed
Quack: <<6, 0, 3, 4, 0, 1, 4>>
Visited: {0, -1, 1, -1, -1, 2, -1, 3}
Quack: <<0, 3, 4, 0, 1, 4, 2, 4>>      <== etc
Quack: <<3, 4, 0, 1, 4, 2, 4>>
Visited: {0, -1, 1, -1, -1, 2, 4, 3}
Quack: <<4, 0, 1, 4, 2, 4, 4, 5>>
Visited: {0, -1, 1, 5, -1, 2, 4, 3}
Quack: <<0, 1, 4, 2, 4, 4, 5, 3, 5, 6, 7>>
Quack: <<1, 4, 2, 4, 4, 5, 3, 5, 6, 7>>
Visited: {0, -1, 1, 5, 6, 2, 4, 3}
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}
```

In the first few lines of this output we see:

- vertex 0 is qushed, then popped
- vertices 2, 5, 7 are qushed, then successively
  - 2 is popped
  - 5 is popped
  - 7 is popped
- 'qushing' means everything gets added to the end of the quack
- the order that vertices are visited is:
  - 0 2 5 7 6 3 4 1
  - note:
    - 0 is a level-0 vertex
    - 2 5 7 are level-1
    - 6 3 4 1 are level-2
    - no vertex is more than a path length of 2 away from the starting vertex

Applications of Depth-First Search

Depth-First Search: Cycle Detection

A graph that does not contain a cycle is called a tree, so

- asking whether a graph contains no cycles is equivalent to
- asking whether a graph is a tree

The *recursive* DFS algorithm, *dfsR()* we saw earlier is:

切换行号显示

```
void dfsR(Graph g, Vertex v, int numV, int *order, int *visited) {
   visited[v] = *order;
   *order = *order+1;
   Vertex w;
   for (w=0; w < numV; w++) {
      if (isEdge(g, newE(v,w)) && visited[w]==UNVISITED) {
         dfsR(g, w, numV, order, visited);
      }
   }
   return;
}
```

To search for a cycle, the function *main()* calls *searchForCycle()*:

- which does housekeeping and in turn
- calls the recursive function *hasCycle()*

The function *hasCycle()* is a simple modification of *dfsR()*. The changes are:

- introduce a variable *found* to terminate the search if a cycle is found
  - this is a sort of early-exit
  - this variable must be passed up the recursive calls
- separate the 2 conditions in the for-loop
  - if *w* is adjacent to *v* then
    - if *w* is UNVISITED then recurse (in other words, keep searching)
    - else *w* has been visited before and **we have a cycle**
      - (assuming we are not going backwards along the same path)
        - to avoid going backwards, we pass 2 vertices to *hasCycle()*

The program is as follows:

切换行号显示

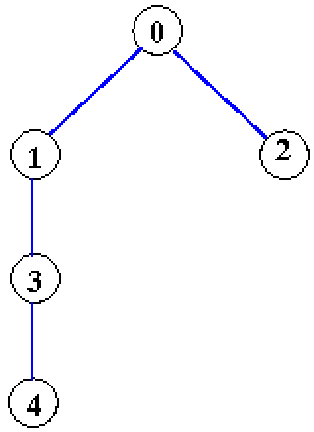
```
void searchForCycle(Graph g, int v, int numV) {
   int *mallocArray(int numV) {
      // as before
   }
```

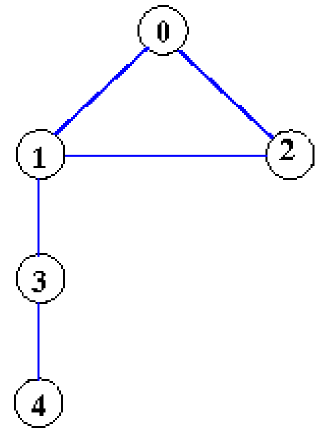
```c
   void showArray(int *vis, int numV) {
      // as before
   }
   int *visited = mallocArray(numV);
   int order = 0;

   if (hasCycle(g, numV, v, v, &order, visited)) {
      printf("found a cycle\n");
   }
   else {
      printf("no cycle found\n");
   }
   showArray(visited, numV);
   free(visited);
   return;
}

int hasCycle(Graph g, int numV, Vertex fromv, Vertex v, int *order, int *visited) {
   int retval = 0;
   visited[v] = *order;
   *order = *order+1;
   Vertex w;
   for (w=0; w<numV && !retval; w++) {
      if (isEdge(g, newE(v,w))) {
         if (visited[w]==UNVISITED) {
            printf("traverse edge %d-%d\n", v, w);
            retval = hasCycle(g, numV, v, w, order, visited);
         }
         else {
            if (w != fromv) { // exclude the vertex we've just come from
               printf("traverse edge %d-%d\n", v, w);
               retval = 1;
            }
         }
      }
   }
   return retval;
}
```

If we input the graph on the left below:





a program that calls this function will output:

```
found a cycle
```

Alternatively, if the input is the graph on the right, it will output:

```
no cycle found
```

Eulerian cycles (using DFS)

An Eulerian **path** in a graph is a path that includes every edge exactly once

- this path may include many visits to the same vertex

In turn, an Eulerian **cycle** is a special case of a Eulerian path in which the start and end points are the same

Animation of the *Konigsburg Bridge Problem*

- http://www.youtube.com/watch?v=3bUvjajakGM

Graph animation of finding an Euler cycle

- http://www.cs.sunysb.edu/~skiena/combinatorica/animations/euler.html

Well-known properties of Eulerian paths and cycles:

```
A connected graph has an Eulerian cycle if all its vertices
have even degree.
```

```
A connected graph has no Eulerian cycle if any of its vertices
has odd degree.
```

```
A connected graph has an Eulerian path if it has exactly 2
vertices of odd degree, and all the rest have even degree.
```

```
A connected graph has no Eulerian path if it has more than 2
vertices of odd degree.
```

So, a graph with more than 2 vertices of odd degree has no Eulerian cycle or path.

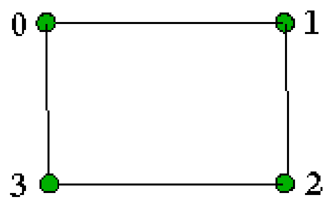
```
If a graph contains a Eulerian path, it cannot contain a
Eulerian cycle, and vice versa.
```

```
A graph that consists of vertices all with even degree is
often called an '''''Eulerian graph'''''
```

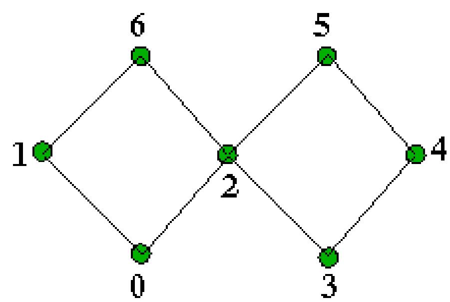
There is a simple algorithm to find an Eulerian cycle in an Eulerian graph

- it uses a DFS, hence requires a stack (to backtrack when you reach a deadend).

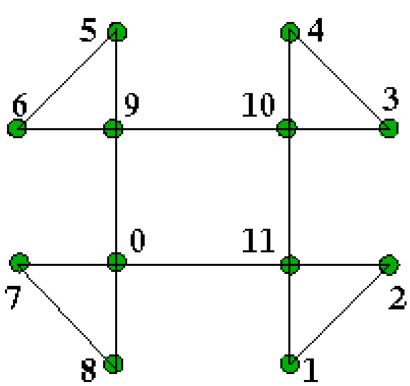
Examples of Eulerian graphs:



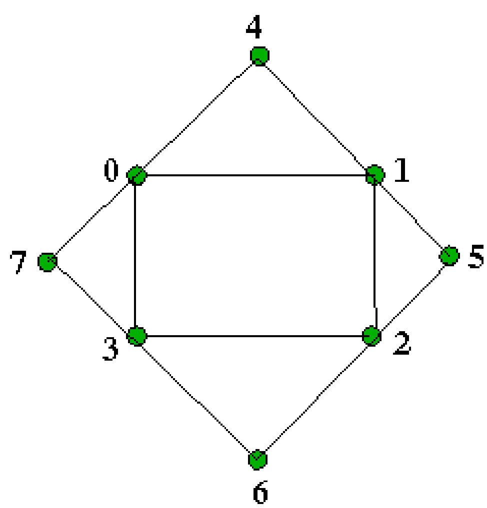
'box'



'bow'



'propbows'



'concsquares'

Eulerian Cycle Algorithm

Assume graph is connected and is Eulerian.

1. Read the Eulerian graph and initialise a stack.

2. Choose any vertex *v* and push *v*.
3. While the stack is not empty:
   - if the top of stack vertex has one of more adjacent vertices
     - select arbitrarily the largest vertex *w*
     - push *w*
     - remove edge *v*-*w*
   - else pop the top vertex and print it

The sequence of vertices that is printed is an Eulerian cycle. How does the algorithm work?

- we traverse the graph by pushing vertices on the stack and removing edges that lead to them
  - the vertex that is pushed is the neighbour of the previous vertex that is pushed
- we continue to push vertices as long as we can
- if we have a vertex on the stack with no neighbours, we start to pop and print

The function *findEulerCycle()* below implements the algorithm:

切换行号显示

```
void findEulerCycle(Graph g, int numV, Vertex startv) {
   Quack s = createQuack();
   printf("Eulerian cycle: ");

   push(startv, s);
   while (!isEmptyQuack(s)) {
      Vertex v = pop(s); // v is the top of stack vertex and ...
      push(v, s);        // ... the stack has not changed
      Vertex w;
      if ((w = getAdjacent(g, numV, v)) >= 0) {
         push(w, s);     // push a neighbour of v onto stack
         removeE(g, newE(v, w)); // remove edge to neighbour
      }
      else {
         w = pop(s);
         printf("%d ", w);
      }
   }
   putchar('\n');
}

Vertex getAdjacent(Graph g, int numV, Vertex v) {
   // returns the Largest Adjacent Vertex if it exists, else -1
   Vertex w;
   Vertex lav = -1; // the adjacent vertex
   for (w=numV-1; w>=0 && lav==-1; w--) {
      Edge e = newE(v, w);
```

```
            if (isEdge(g, e)) {
               lav = w;
            }
         }
         return lav;
      }
```

For example: for the **box** graph above:

```
push 0
push 3 and remove 0-3
push 2 and remove 3-2
push 1 and remove 2-1
push 0 and remove 1-0 <-- at this point all the edges have
been removed, and 0 is isolated
pop 0 and print 0
pop 1 and print 1
pop 2 and print 2
pop 3 and print 3
pop 0 and print 0
```

- **Eulerian cycle: 0 1 2 3 0**
- *It is not obvious why you need a stack here: 5 pushes were followed by 5 pops.*

We can reach a deadend during the traversal

- if the top vertex on the stack has no adjacent vertices, the traversal has reached a deadend
  - (remember that we are removing edges as we traverse the graph)
- but there may be vertices on the stack that **do** have branches to vertex neighbours
  - if there is one, then a vertex neighbour is pushed and the traversal continues
  - this is a process of **back-tracking** of course
- the process stops when branches have been taken
  - this will happen when all edges have been removed
  - every vertex on the stack will have been popped and printed
    - we may see the same vertex many times, but each edge is traversed just once

In the next example, for the **bow** graph above, we see backtracking in action:

```
push 0
push 2 and remove 0-2
push 6 and remove 2-6
push 1 and remove 6-1
push 0 and remove 1-0
pop 0 and print 0
pop 1 and print 1
pop 6 and print 6
push 5 and remove 2-5
push 4 and remove 5-4
push 3 and remove 4-3
push 2 and remove 3-2
```

```
pop 2 and print 2
pop 3 and print 3
pop 4 and print 4
pop 5 and print 5
pop 2 and print 2
pop 0 and print 0
```

- **Eulerian cycle: 0 1 6 2 3 4 5 2 0**

You may need to backtrack many times of course. Consider the **propbows** graph above:

```
push 0
pushed 11 and remove edge 0-11
pushed 10 and remove edge 11-10
pushed 9 and remove edge 10-9
pushed 6 and remove edge 9-6
pushed 5 and remove edge 6-5
pushed 9 and remove edge 5-9
pushed 0 and remove edge 9-0
pushed 8 and remove edge 0-8
pushed 7 and remove edge 8-7
pushed 0 and remove edge 7-0
popped 0 and print 0
popped 7 and print 7
popped 8 and print 8
popped 0 and print 0
popped 9 and print 9
popped 5 and print 5
popped 6 and print 6
popped 9 and print 9
pushed 4 and remove edge 10-4
pushed 3 and remove edge 4-3
pushed 10 and remove edge 3-10
popped 10 and print 10
popped 3 and print 3
popped 4 and print 4
popped 10 and print 10
pushed 2 and remove edge 11-2
pushed 1 and remove edge 2-1
pushed 11 and remove edge 1-11
popped 11 and print 11
popped 1 and print 1
popped 2 and print 2
popped 11 and print 11
popped 0 and print 0
```

- **Eulerian cycle: 0 7 8 0 9 5 6 9 10 3 4 10 11 1 2 11 0**

Path searching (using DFS)

You could want to search for a path between two given vertices:

- the starting vertex we already have
- add the goal vertex as a parameter and test 'is there a path' in the *dfs()* function.

切换行号显示

```
int isPath(Graph g, Vertex v, Vertex goalv, int numV, int *order, int *visited) {
   int found = 0;
   visited[v] = *order;
   *order = *order+1;
   if (v == goalv) {
      found = 1;
   }
   else {
      Vertex w;
      for (w=0; w < numV && !found; w++) {
         if (isEdge(g, newE(v,w))) {
            if (visited[w] == UNVISITED) {
               found = isPath(g, w, goalv, numV, order, visited);
               printf("path %d-%d\n", w, v);
            }
         }
      }
   }
   return found;
}
```

- this function does a DFS traversal, exactly like *dfsR()* ...
  - ... but at the same time, it searches for a path to a vertex *goalv*

So, instead of the call to *dfsR()* (in *dfs()*):

切换行号显示

```
   dfsR(g, v, numV, &order, visited);
```

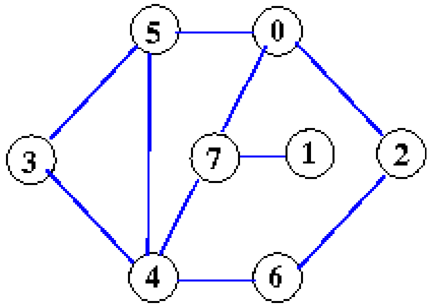
we call the function *isPath()*:

切换行号显示

```
   #define STARTV 0
   #define GOALV 3
   if (isPath(g, STARTV, GOALV, numV, &order, visited)) { //notice the STARTV and GOALV arguments
      printf("found path\n");
   }
   else {
      printf("no path\n");
   }
```

- Here a #define is used to name the start and goal vertices: this could better be input interactively of course.

If we input the graph:



then the output is:

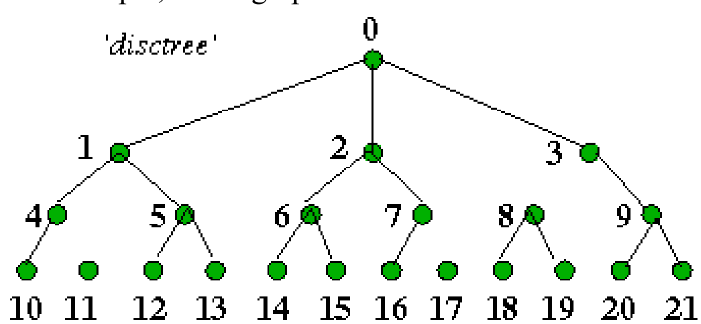
```
path 3-4
path 4-6
path 6-2
path 2-0
found path
Visited: {0, -1, 1, 4, 3, -1, 2, -1}
```

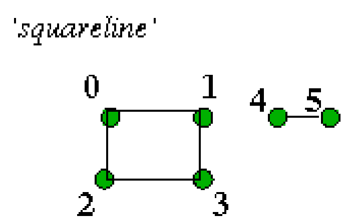
- Here we can see which vertices were visited

Reachability Analysis

In many problems we are interested in knowing which vertices are *reachable* from some start vertex.

- for example, in the graphs:





some of the vertices are unreachable from the start vertex 0, others are not

- if the graph is undirected, it is obviously disconnected
- (if the graph is directed then it may not be disconnected of course)

A DFS algorithm can be used to find all the reachable vertices

- simply run the algorithm from the start vertex
- on conclusion, check the visited array
  - any vertex (except the start vertex) that is unvisited is unreachable

An alternative method is to use so-called 'fixed-point' computation.

1. initialise:
    - a reachable set comprising of just the start vertex
    - every other vertex is considered unreachable
2. check every unreachable vertex *v*
    - if there is an edge from a vertex in the reachable set to *v*
        - then add *v* to the reachable set
3. repeat the previous step until the reachable set does not change
    - if the reachable set does not change, terminate

When the set does not change, we have reached a 'fixed point'

- the set of vertices in the reachable set can be reached from the start vertex
- all other vertices cannot be reached

Example: consider the graph *squareline* above

- let *R* be the set of reachable vertices, and the start vertex be 0
    - initially $R = \{0\}$
- consider vertices 1..5
    - 1 is adjacent to 0, add to *R*
    - 2 is adjacent to 0, add to *R*
    - $R = \{0, 1, 2\}$
    - *R* has changed, so repeat
- consider vertices 3..5
    - 3 is adjacent to 1, add to *R*
    - $R = \{0, 1, 2, 3\}$
    - *R* has changed, so repeat
- consider vertices 4 and 5
    - neither vertex is adjacent to a vertex in *R*
    - *R* does not change
- terminate the algorithm

Notice that you do not needs to use a stack here, or recursion. There is no backtracking.

Application of Breadth-First Search: path searching

You could want to search for a path between two given vertices, *start* and *goal* say:

- the starting vertex we already have
- add the goal vertex as a parameter and test for it in the *bfs()* function

But, we don't follow any 'path' during BFS (as we did in DFS): we traverse by level:

- whenever we 'visit' a node, we must remember its parent
    - we store the parent of each vertex in an array called *parent[]*

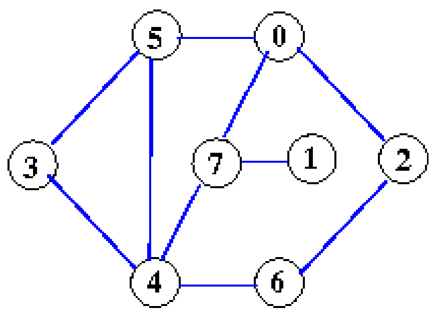
- we hence need 2 arrays, *visited[]* and *parent[]*
- when we find the goal node, we're finished
    - the path 'backwards' to the goal node will be stored in *parent[]*
- we print the path from the *start* to the *goal* stored in *parent[]*

切换行号显示

```
void searchPath(Graph g, Vertex start, Vertex goal, int numV) {
   int *visited = mallocArray(numV);
   int *parent = mallocArray(numV);
   Quack q = createQuack();
   qush(start, q);
   int order = 0;
   visited[start] = order++;
   int found = 0;
   while (!isEmptyQuack(q) && !found) {
      Vertex x = pop(q);
      for (Vertex y = 0; y < numV && !found; y++) {
         if (isEdge(newEdge(x,y), g)) {      // for adjacent vertex y ...
            if (visited[y] == UNVISITED) {  // ... if y is unvisited ...
               qush(y, q);                   // ... queue y
               printf("\t\t\t ... trying edge %d-%d\n", x, y);
               visited[y] = order++;      // y is now visited
               parent[y] = x;             // y's parent is x
               if (y == goal) {           // if y is the goal ...
                  found = 1;              // ... SUCCESS! now get out
               }
            }
         }
      }
   }
   if (found) {
      printf("SHORTEST path from %d to %d is ", start, goal);
      printPath(parent, numV, goal); // an extern function
      putchar('\n');
   }
   else {
      printf("no path found\n");
   }
   printArray("Visited: ", visited, numV);
   printArray("Parent : ", parent, numV);
```

```
    free(visited);
    free(parent);
    makeEmptyQuack(q);
    return;
}
```

If we input the graph:



then the output is:

```
SHORTEST path from 0 to 6 is 6<--2<--0
```

The array *parent[]* stores the parent of every visited node in the graph, but:

- the start vertex has no parent
- unvisited nodes have no parents

To print the path:

- we print *goal*
- we print *parent[goal]*
- we print *parent[parent[goal]]*
- we print *parent[parent[parent[goal]]]*
- we print *parent[parent[parent[parent[goal]]]]*
- ...
- until we find the *start* vertex

That's what the function below does:

切换行号显示

```
void printPath(int parent[], int numV, Vertex v) {
   printf("%d", v);
   if (0<=v && v<numV) {
      Vertex p = parent[v];
      while (p != UNVISITED) {
          printf("<--%d", p);
          p = parent[p];
      }
   }
   else {
```

```
        fprintf(stderr, "printPath: illegal vertex in parent[]\n");
    }
}
```

As we start with *goal* and work backwards, we print the path in reverse direction.

GraphSearch (2019-07-22 17:59:26由AlbertNymeyer编辑)