目录

Applications of Graph Search

Breadth-First Search: Shortest path

Want the **shortest path between vertices *start* and *goal***

- the start vertex we already have (root of the graph)
- add the goal vertex as an argument
- rename as *bfsShortestPath()*

So

切换行号显示

```
void bfsQuack(Graph g, Vertex v, int numV)
```

becomes

切换行号显示

```
void bfsShortestPath(Graph g, Vertex start, Vertex goal, int numV)
```

We need to do 2 extra things in *bfsShortestPath()*

1. know when we reach the goal vertex
2. know how to find a path back 'up' the graph (towards the root)
   - this is BFS, so this path must be the shortest

How do we find a path backwards?

- must remember who the *parent* of a vertex is
- in the function, if `isEdge(newEdge(v, w), g)` is true and *w* is unvisited
  - then we set `visited[w] = order++`
- we must do more:
  - store the parent of the vertex in an array *parent[]*: namely `parent[w] = v`
  - do this whenever we visit a vertex

The array *parent[]* enables us to 're-trace' a path back 'up' the graph when we find *goal*

- we stop traversing the graph when we reach the *goal* node

  - *visited[]* may be incomplete, *parent[]* may be incomplete, the quack may not be empty
- **that's all fine: we've got what we wanted, the shortest path**

This is implemented below:

切换行号显示

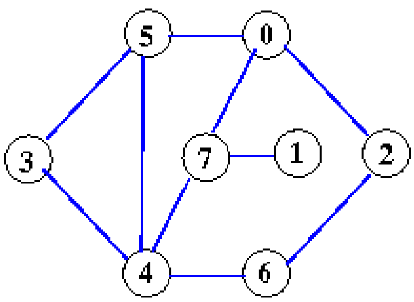
```
#include <stdio.h>
#include <stdlib.h>
#include "Graph.h"
#include "Quack.h"
#include "IOmem.h"

#define STARTVERTEX 0
#define GOALVERTEX  4

void printPath(int *, int, Vertex);
void shortestPath(Graph, Vertex, Vertex, int);

int main(void) {
   int numV;
   if ((numV = readNumV()) > 0) {        // read #vertices
      Graph g = newGraph(numV);
      if (readBuildGraph(g) == 1) {
         showGraph(g);

         printf("Find shortest path from ");
         printf("%d to %d\n", STARTVERTEX, GOALVERTEX);
         shortestPath(g, STARTVERTEX, GOALVERTEX, numV);
      }
      freeGraph(g);
      g= NULL;
   }
   else {
      printf("Error in reading #number\n");
      return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
}

void shortestPath(Graph g, Vertex start, Vertex goal, int numV) {
   int *visited = mallocArray(numV);
   int *parent = mallocArray(numV);          // THIS IS NEW

   Quack q = createQuack();
   qush(start, q);
   int order = 0;
   visited[start] = order++;
   int found = 0;
   while (!isEmptyQuack(q) && !found) {      // FOUND TELLS US TO STOP
      Vertex v = pop(q);
      for (Vertex w = 0; w < numV && !found; w++) {
```

```
         if (isEdge(newEdge(v,w), g)) {      // for adjacent vertex w ...
            if (visited[w] == UNVISITED) {   // ... if w is unvisited ...
               qush(w, q);                   // ... queue w
               printf("\t\t\t ... doing edge %d-%d\n", v, w); // DEBUG
               visited[w] = order++;         // w is now visited
               parent[w] = v;                // w's PARENT is v
               if (w == goal) {              // if w is the goal ...
                  found = 1;                 // ..FOUND IT! NOW GET OUT
               }
            }
         }
      }
   }
   if (found) {
      printf("SHORTEST path from %d to %d is ", start, goal);
      printPath(parent, numV, goal); // print path from START TO GOAL
      putchar('\n');
   }
   else {
      printf("no path found\n");
   }
   printArray("Visited: ", visited, numV);   // debug
   printArray("Parent : ", parent, numV);    // debug
   free(visited);
   free(parent);
   makeEmptyQuack(q);
   return;
}
```

Compile and execute:

```
dcc bfsShortestPath.c IOmem.c GraphAM.c Quack.c
./a.out < exsedg.inp
```

The file *exsedg.inp* is the graph:



The output is:

```
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>
Find shortest path from 0 to 4
Doing edge 0-2
Doing edge 0-5
Doing edge 0-7
Doing edge 2-6
Doing edge 5-3
Doing edge 5-4
SHORTEST path from 0 to 4 is 0-->5-->4
Visited: {0, -1, 1, 5, 6, 2, 4, 3}
Parent : {-1, -1, 0, 5, 5, 0, 2, 0}
```

The array *parent[]* stores the parent of every visited node

- ... except the start vertex, which has no parent, so stays -1

Printing the path is not trivial.

- we call *printPath()* in line 61 with the **goal** vertex as argument
- *think about that*
  - *... you have the goal vertex*
  - *you want to print the path from the start to the goal vertex*

The array *parent[]* tells you how to go back to the start vertex (i.e. from *goal* to *start*)

- but you want to print the path from *start* to *goal*!

SO, starting at the goal

- you want to print the path in reverse direction
  - which is the right direct to print the path

Use *head* recursion:

- *printPath()* calls itself immediately until it reaches ... ?
  - ... a vertex that has no parent: this is the start!
- after calling itself, the function simply prints the vertex

切换行号显示

```
// head recursion
void printPath(int parent[], int numV, Vertex v) {
   if (parent[v] != UNVISITED) { // parent of start is UNVISITED
      if (0 <= v && v < numV) {
         printPath(parent, numV, parent[v]); // head recursion
         printf("-->");
```

```
        }
        else {
            fprintf(stderr, "\nprintPath: invalid goal %d\n", v);
        }
    }
    printf("%d", v);           // print the vertices 'deepest-first'
    return;
}
```

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

We can use this recursive DFS to find cycles.

- this uses the Graph ADT (but not the Quack ADT)

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

- creates *visited[]* array
- calls the recursive function *hasCycle()* (like *dfsR()*)

The program is as follows:

切换行号显示

```
// hasCycle.c: search for a cycle in a graph using DFS
#include <stdio.h>
#include <stdlib.h>
#include "Graph.h"
#include "IOmem.h"

#define UNVISITED -1
```

```
#define STARTVERTEX 0

void searchForCycle(Graph, int, int);
int hasCycle(Graph, int, Vertex, Vertex, int *, int *);

int main(void) {
   int numV;
   if ((numV = readNumV()) > 0) {        // read #vertices
      Graph g = newGraph(numV);
      if (readBuildGraph(g) == 1) {
         showGraph(g);

         searchForCycle(g, STARTVERTEX, numV);
      }
      freeGraph(g);
      g= NULL;
   }
   else {
      printf("Error in reading #number\n");
      return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
}

// a wrapper for the recursive call to hasCycle()
void searchForCycle(Graph g, int v, int numV) {
   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");
   }
   printArray("Visited ", 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;
   for (Vertex w = 0; w < numV && !retval; w++) {
      if (isEdge(newEdge(v,w), g)) {
         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
               // WE HAVE REVISITED A VERTEX ==> CYCLE
               printf("traverse edge %d-%d\n", v, w);
```

```
                    retval = 1;
                }
            }
        }
    }
    return retval;
}
```

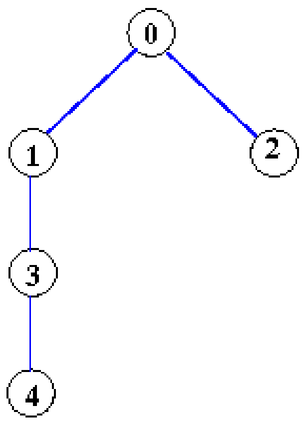
What differences are there in *hasCycle* (compared to *dfsR*)?

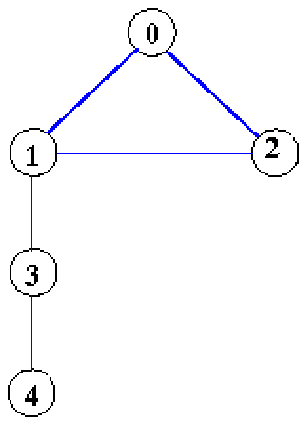
- uses a variable *found* to end the search if a cycle found
    - this variable must be passed up the recursive calls
- separate the 2 conditions in the for-loop
    - if *v*--*w* is an edge then ...
        - if *w* is UNVISITED then recurse (in other words, keep searching)
        - else *w* has been visited before and **we have found a cycle**
            - (assuming we are not going backwards to the vertex we just came from)
                - to avoid going backwards, we pass 2 vertices to *hasCycle()*

Compile:

```
dcc hasCycle.c IOmem.c GraphAM.c
```

We test with the following 2 graphs:





Execute for the graph on the left with input file *exdead.inp*

```
./a.out < exdead.inp
V=5, E=4
<0 1> <0 2>
<1 0> <1 3>
<2 0>
<3 1> <3 4>
<4 3>
traverse edge 0-1
traverse edge 1-3
traverse edge 3-4
traverse edge 0-2
no cycle found
Visited {0, 1, 4, 2, 3}
```

Execute for the graph on the right with input file *excycle.inp*

```
V=5, E=5
<0 1> <0 2>
<1 0> <1 2> <1 3>
<2 0> <2 1>
<3 1> <3 4>
<4 3>
traverse edge 0-1
traverse edge 1-2
traverse edge 2-0
found a cycle
Visited {0, 1, 2, -1, -1}
```

Depth-First Search: Eulerian cycles

An Eulerian path is a path that includes every edge exactly once

A path may include many visits to the same vertex.

An Eulerian **cycle** is an Eulerian path that starts and ends on the same vertex

Graph animation of an Euler cycle

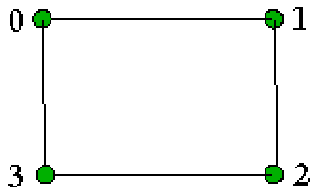
- http://www.cs.sunysb.edu/~skiena/combinatorica/animations/euler.html

Limiting ourselves to connected graphs:

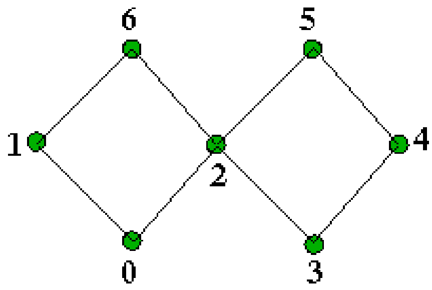
A graph:

- has an Eulerian cycle if all its vertices have even degree
  - a graph with this property is sometimes called an *Eulerian graph*
- has no Eulerian cycle if any of its vertices has odd degree
-
- has an Eulerian path if all its vertices have even degree, except for exactly 2 that have odd degree
- has no Eulerian path if it has more than 2 vertices of odd degree
-
- has neither an Euler path nor cycle if more than 2 vertices have odd degree
- cannot have both an Eulerian path and Eulerian cycle at the same time

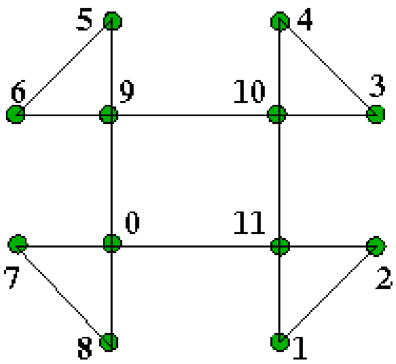
Examples of Eulerian graphs:



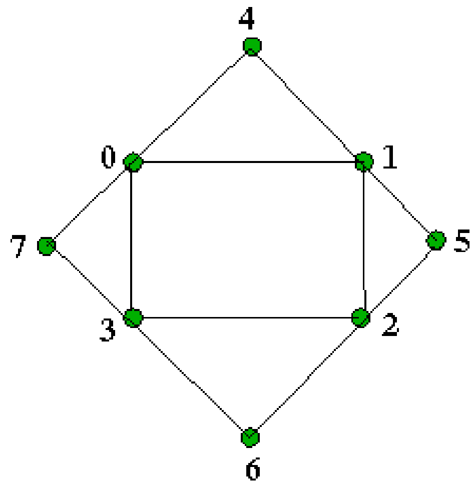
'box'



'bow'



'propbows'



'concsquares'

There is a simple algorithm to find an Eulerian cycle

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

Why the largest vertex?

- *no reason*
- the graph is arbitrarily labelled
- there are many possible Eulerian cycles
    - backtracking will depend on labelling and selection strategy

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
         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 retv = -1; // assume the worst
   for (Vertex w = numV-1; w >= 0 && retv == -1; w--) {
      Edge e = newEdge(v, w);
      if (isEdge(e, g)) {
         retv = w;
      }
   }
   return retv;
}
```

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 <-- 0 is now disconnected
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**

Depth-First Search: Finding a path

You may want to find a path between:

- a start vertex
- a goal vertex

using the DFS algorithm

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 {
      for (Vertex w = 0; w < numV && !found; w++) {
         if (isEdge(newEdge(v,w), g)) {
            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

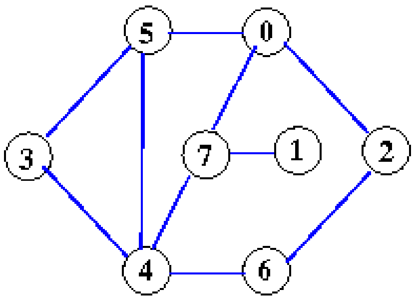
```
   dfsR(g, v, numV, &order, visited);
```

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");
}
```

If we input the graph:



then the output is:

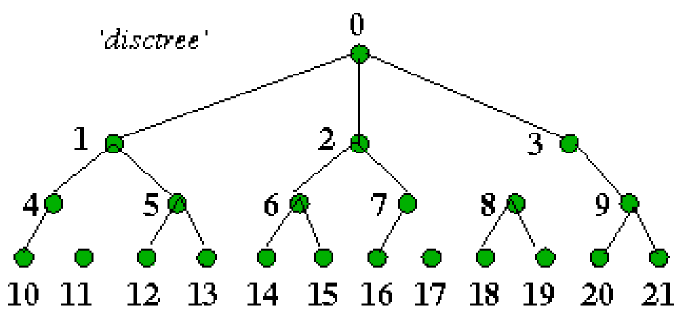
```
path 3-4
path 4-6
path 6-2
path 2-0
found path
Visited: {0, -1, 1, 4, 3, -1, 2, -1}
```

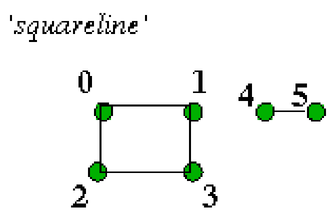
- Here we can see which vertices were visited

Depth-First Search: reachability

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.

GraphSearchApps (2019-07-25 11:11:25由AlbertNymeyer编辑)