Graph Algorithms Intro

- Problems on Graphs
- Cycle Checking
- Connected Components
- Hamiltonian Path and Circuit
- Euler Path and Circuit

❖ Problems on Graphs

What kind of problems do we want to solve on/via graphs?

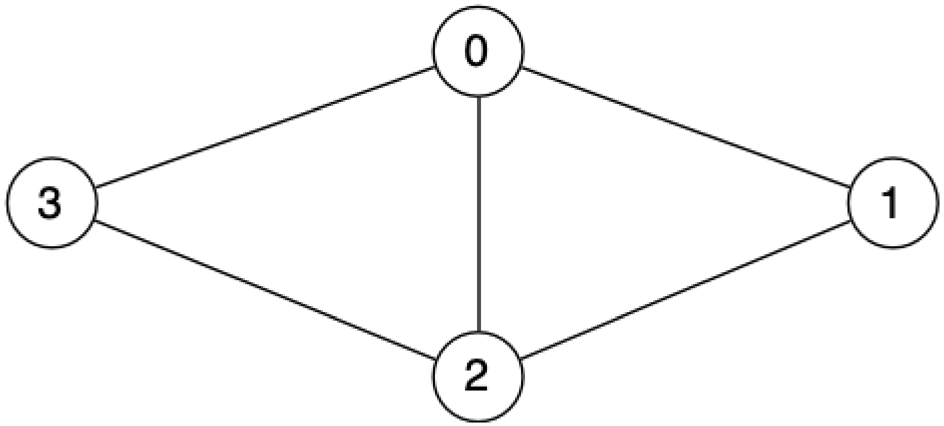
- how many connected components in the graph?
- is one vertex reachable from some other vertex? (path finding)
- what is the cheapest cost path from *v* to *w*?
- which vertices are reachable from *v*? (transitive closure)
- is there a cycle somewhere in the graph?
- is there a cycle that passes through all vertices? (circuit)
- is there a tree that links all vertices? (spanning tree)
- what is the minimum spanning tree?
- ...
- can a graph be drawn in a plane with no crossing edges? (planar graphs)
- are two graphs "equivalent"? (isomorphism)

❖ Cycle Checking

A graph has a cycle if

- it has a path of length > 2
- with start vertex *src* = end vertex *dest*
- and without using any edge more than once

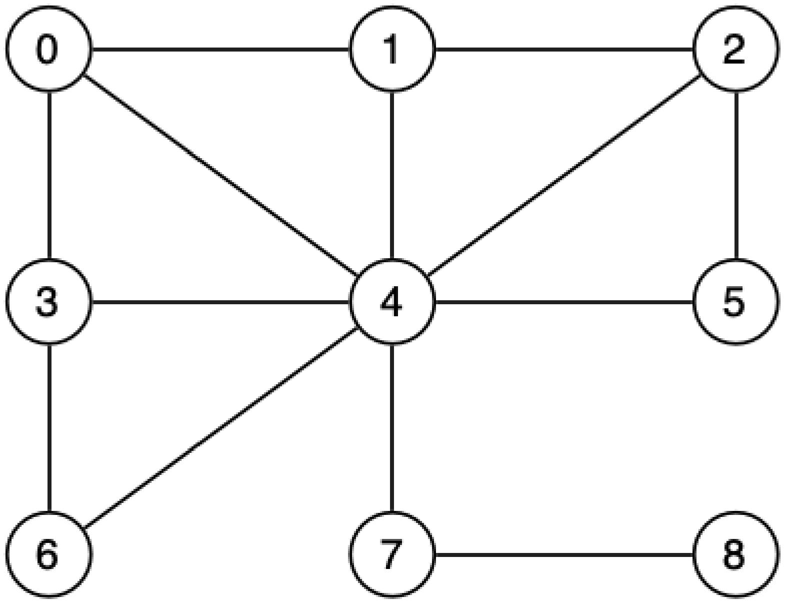
This graph has 3 distinct cycles: 0-1-2-0, 2-3-0-2, 0-1-2-3-0



("distinct" means the *set* of vertices on the path, not the order)

❖ ... Cycle Checking

Consider this graph:



This graph has many cycles e.g. 0-4-3-0, 2-4-5-2, 0-1-2-5-4-6-3-0, ...

❖ ... Cycle Checking

First attempt at checking for a cycle

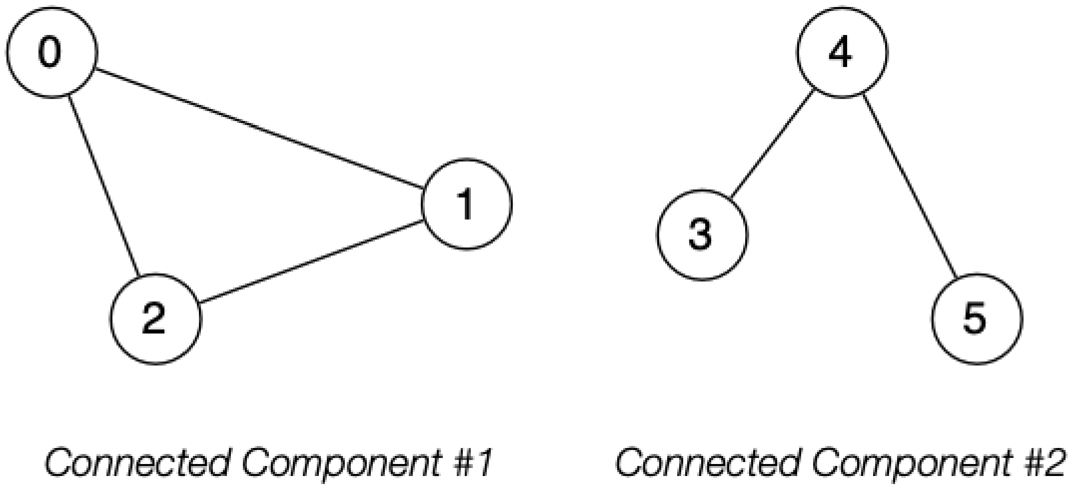
```
hasCycle(G):
|  Input  graph G
|  Output true if G has a cycle, false otherwise
|
|  choose any vertex v ∈ G
|  return dfsCycleCheck(G,v)

dfsCycleCheck(G,v):
|  mark v as visited
|  for each (v,w) ∈ edges(G) do
|  |  if w has been visited then  // found cycle
|  |     return true
|  |  else if dfsCycleCheck(G,w) then
|  |     return true
|  end for
|  return false  // no cycle at v
```

❖ ... Cycle Checking

The above algorithm has two bugs ...

- only one connected component is checked
- the loop **`for each (v,w) ∈ edges(G) do`** should exclude the neighbour of *v* from which you just came, so as to prevent a single edge *w*-*v* being classified as a cycle.

If we start from vertex 5 in the following graph, we don't find the cycle:



❖ ... Cycle Checking

Version of cycle checking (in C) for one connected component:

```
bool dfsCycleCheck(Graph g, Vertex v, Vertex u) {
   visited[v] = true;
   for (Vertex w = 0; w < g->nV; w++) {
      if (adjacent(g, v, w)) {
         if (!visited[w]) {
            if (dfsCycleCheck(g, w, v))
               return true;
         }
         else if (w != u)
            return true;
      }
   }
   return false;
}
```

❖ ... Cycle Checking

Wrapper to ensure that all connected components are checked:

```
Vertex *visited;

bool hasCycle(Graph g, Vertex s) {
   bool result = false;
   visited = calloc(g->nV,sizeof(int));
   for (int v = 0; v < g->nV; v++) {
      for (int i = 0; i < g->nV; i++)
         visited[i] = -1;
      if dfsCycleCheck(g, v, v)) {
         result = true;
         break;
      }
   }
   free(visited);
   return result;
}
```

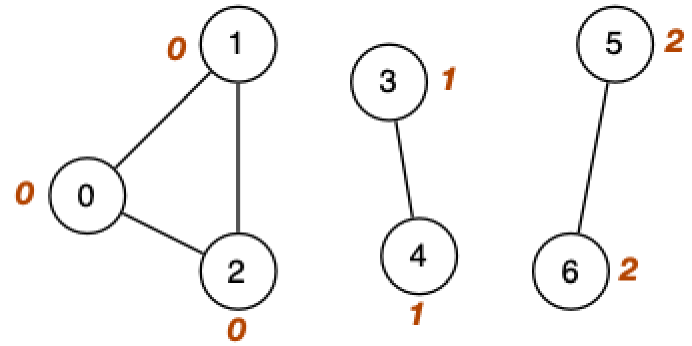
❖ Connected Components

Consider these problems:

- how many connected subgraphs are there?
- are two vertices in the same connected subgraph?

Both of the above can be solved if we can

- build **`componentOf[]`** array, one element for each vertex *v*
- indicating which connected component *v* is in




```
nComponents(g) = 3

componentOf[1] = 0
componentOf[5] = 2

sameComponent(3,4) = true
sameComponent(3,5) = false
sameComponent(0,6) = false
```

❖ ... Connected Components

Algorithm to assign vertices to connected components:

```
components(G):
|  Input  graph G
|  Output componentOf[] filled for all V
|
|  for all vertices v ∈ G do
|  |  componentOf[v]=-1
|  end for
|  compID=0  // component ID
|  for all vertices v ∈ G do
|  |  if componentOf[v]=-1 then
|  |     dfsComponent(G,v,compID)
|  |     compID=compID+1
|  |  end if
|  end for
```

❖ ... Connected Components

DFS scan of one connected component

```
dfsComponent(G,v,id):
|  componentOf[v]=id
|  for each (v,w) ∈ edges(G) do
|     if componentOf[w]=-1 then
|        dfsComponent(G,w,id)
|     end if
|  end for
```

❖ ... Connected Components

Consider an application where connectivity is critical

- we frequently ask questions of the kind above
- but we cannot afford to run **`components()`** each time

Add a new fields to the **`GraphRep`** structure:

```
typedef struct GraphRep *Graph;

struct GraphRep {
    ...
    int nC;   // # connected components
    int *cc;  // which component each vertex is contained in
    ...       // i.e. array [0..nV-1] of 0..nC-1
}
```

❖ ... Connected Components

With this structure, the above tasks become trivial:

```
// How many connected subgraphs are there?
int nConnected(Graph g) {
   return g->nC;
}
// Are two vertices in the same connected subgraph?
bool inSameComponent(Graph g, Vertex v, Vertex w) {
   return (g->cc[v] == g->cc[w]);
}
```

But ... introduces overheads ... maintaining **`cc[]`**, **`nC`**

❖ ... Connected Components

Consider maintenance of such a graph representation:

- initially, **`nC = nV`** (because no edges)
- adding an edge may reduce **`nC`**
  (adding edge between *v* and *w* in different components)
- removing an edge may increase **`nC`**
  (removing edge between *v* and *w* in same component)
- **`cc[]`** can simplify path checking
  (ensure **`v,w`** are in same component before starting search)

Additional cost amortised by lower cost for **`nConnected()`** and **`inSameComponent()`**

Is it simpler to run **`components()`** after each edge change?

❖ Hamiltonian Path and Circuit

Hamiltonian path problem:

- find a simple path connecting two vertices *v*,*w* in graph *G*
- such that the path includes each vertex exactly once

If *v* = *w*, then we have a Hamiltonian circuit

Simple to state, but difficult to solve (*NP*-complete)

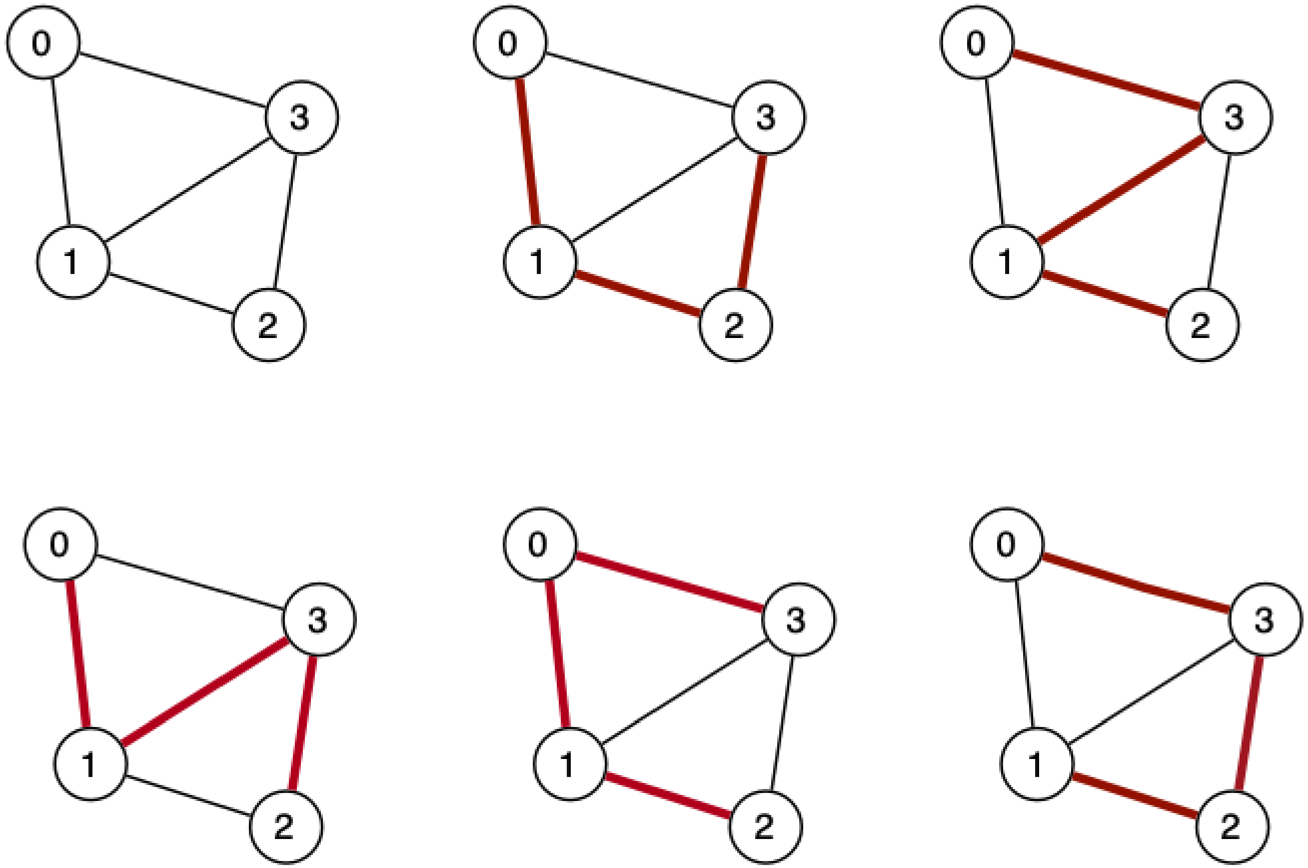
Many real-world applications require you to visit all vertices of a graph:

- Travelling salesman
- Bus routes
- ...

Named after Irish mathematician/physicist/astronomer Sir William Hamilton (1805-1865)

❖ ... Hamiltonian Path and Circuit

Graph and some possible Hamiltonian paths:



❖ ... Hamiltonian Path and Circuit

Approach:

- generate all possible simple paths (using e.g. DFS)
- keep a counter of vertices visited in current path
- stop when find a path containing V vertices

Can be expressed via a recursive DFS algorithm

- similar to simple path finding approach, except
  - keeps track of path length; succeeds if length = v
  - resets "visited" marker after unsuccessful path

❖ ... Hamiltonian Path and Circuit

Algorithm for finding Hamiltonian path:

```
visited[] // array [0..nV-1] to keep track of visited vertices

hasHamiltonianPath(G,src,dest):
|   Input  graph G, plus src/dest vertices
|   Output true if Hamiltonian path src...dest,
|          false otherwise
|
|   for all vertices v ∈ G do
|      visited[v]=false
|   end for
|   return hamiltonR(G,src,dest,#vertices(G)-1)
```

❖ ... Hamiltonian Path and Circuit

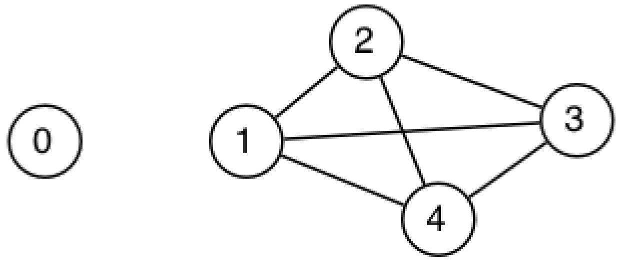
Recursive component:

```
hamiltonR(G,v,dest,d):
|  Input G    graph
|        v    current vertex considered
|        dest destination vertex
|        d    distance "remaining" until path found
|
|  if v=dest then
|     if d=0 then return true else return false
|  else
|  |  visited[v]=true
|  |  for each (v,w) ∈ edges(G)  where not visited[w] do
|  |     if hamiltonR(G,w,dest,d-1) then
|  |        return true
|  |     end if
|  |  end for
|  end if
|  visited[v]=false             // reset visited mark
|  return false
```

❖ ... Hamiltonian Path and Circuit

Analysis: worst case requires *(V-1)!* paths to be examined

Consider a graph with isolated vertex and the rest fully-connected



Checking **`hasHamiltonianPath(g,`*`x`*`,0)`** for any *x*

- requires us to consider every possible path
- e.g 1-2-3-4, 1-2-4-3, 1-3-2-4, 1-3-4-2, 1-4-2-3, ...
- starting from any *x*, there are 3! paths ⇒ 4! total paths
- there is no path of length 5 in these *(V-1)!* possibilities

There is no known simpler algorithm for this task ⇒ *NP*-hard.

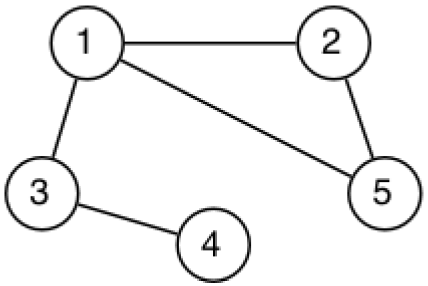
Note, however, that the above case could be solved in constant time if we had a fast check for 0 and x being in the same connected component

❖ Euler Path and Circuit

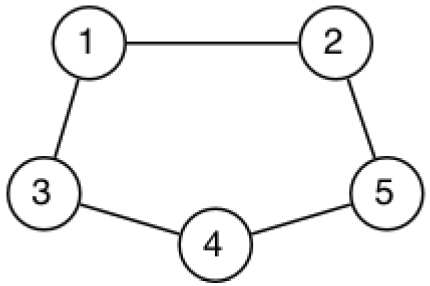
Euler path problem:

- find a path connecting two vertices v,w in graph G
- such that the path includes each edge exactly once
  (note: the path does not have to be simple ⇒ can visit vertices more than once)

If $v = w$, the we have an Euler circuit



Euler Path: 4-3-1-5-2-1



Euler Circuit: 1-2-5-4-3-1

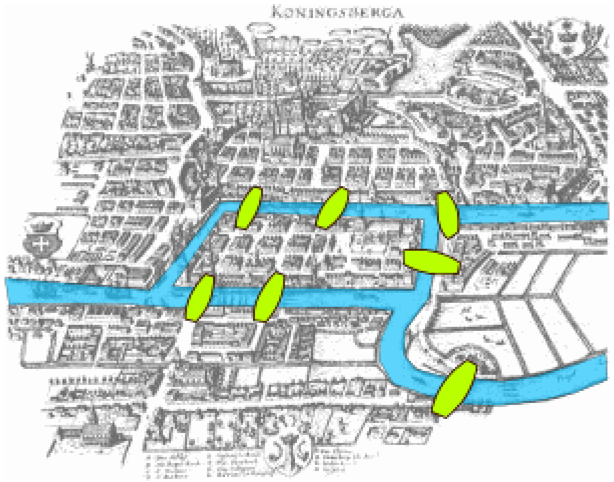
Many real-world applications require you to visit all edges of a graph:

- Postman
- Garbage pickup
- ...

❖ ... Euler Path and Circuit

Problem named after Swiss mathematician, physicist, astronomer, logician and engineer Leonhard Euler (1707 - 1783)

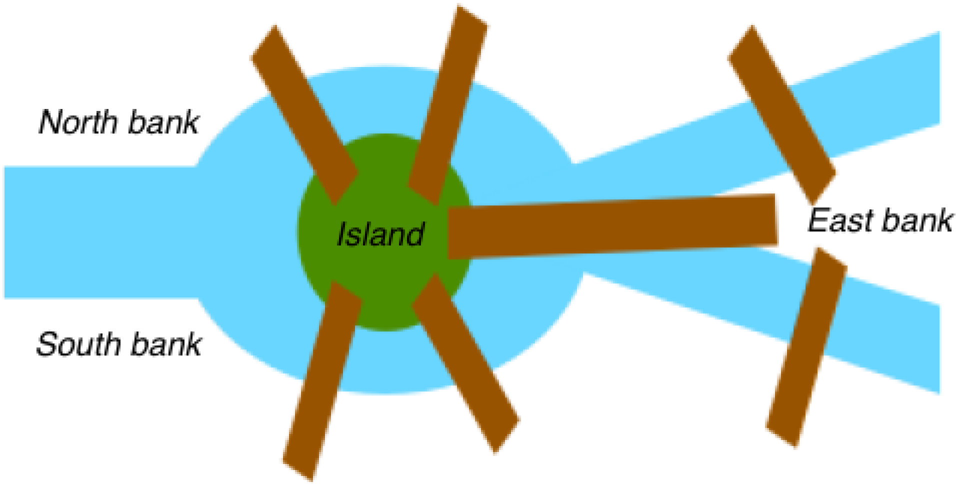
Based on a circuitous route via bridges in Konigsberg



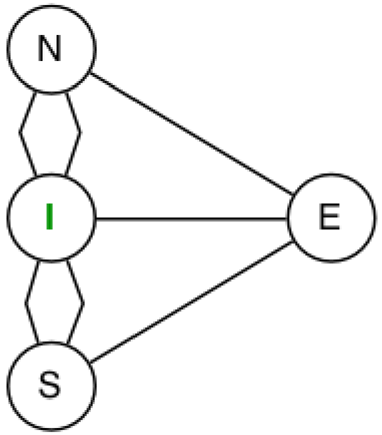
❖ ... Euler Path and Circuit

Is there a way to cross all the bridges of Konigsberg exactly once on a walk through the town?

- treat land as nodes; bridges as edges



Bridges as schematic



Bridges as graph

❖ ... Euler Path and Circuit

One possible "brute-force" approach:

- check for each path if it's an Euler path
- would result in factorial time performance

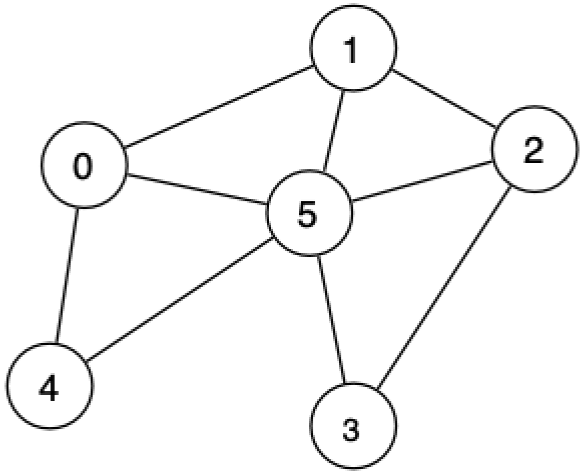
Can develop a better algorithm by exploiting:

Theorem. A graph has an Euler circuit if and only if it is connected and all vertices have even degree

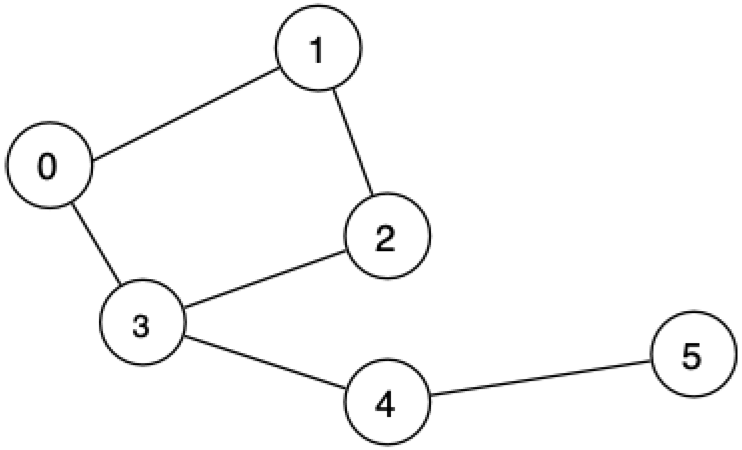
Theorem. A graph has a non-circuitous Euler path if and only if it is connected and exactly two vertices have odd degree

❖ ... Euler Path and Circuit

Graphs with an Euler path are often called Eulerian Graphs



Has neither Eulerian path or circuit



Has no Eulerian circuit, but does have path

5 - 4 - 3 - 0 - 1 - 2 - 3

❖ ... Euler Path and Circuit

Assume the existence of **degree(g,v)**

Algorithm to check whether a graph has an Euler path:

```
hasEulerPath(G,src,dest):
|  Input  graph G, vertices src,dest
|  Output true if G has Euler path from src to dest
|         false otherwise
|
|  if src≠dest then
|     if degree(G,src) is even ∨ degree(G,dest) is even then
|        return false
|     end if
|  end if
|  for all vertices v ∈ G do
|     if v≠src ∧ v≠dest ∧ degree(G,v) is odd then
|        return false
|     end if
|  end for
|  return true
```

❖ ... Euler Path and Circuit

Analysis of **`hasEulerPath`** algorithm:

- assume that connectivity is already checked
- assume that **`degree()`** is available via $O(1)$ lookup
- single loop over all vertices ⇒ $O(V)$

If degree requires iteration over vertices

- cost to compute degree of a single vertex is $O(V)$
- overall cost is $O(V^2)$

⇒ problem tractable, even for large graphs (unlike Hamiltonian path problem)

For the keen, a linear-time (in the number of edges, E) algorithm to compute an Euler path is described in [Sedgewick] Ch.17.7.

Produced: 25 Jul 2020