

# COMP2521 25T2

## Graphs (III)

### Graph Problems

Sim Mautner

`cs2521@cse.unsw.edu.au`

cycle checking  
connected components  
hamiltonian paths/circuits  
eulerian paths/circuits

Cycle  
Checking

Connected  
Components

Hamiltonian  
Path/Circuit

Eulerian  
Path/Circuit

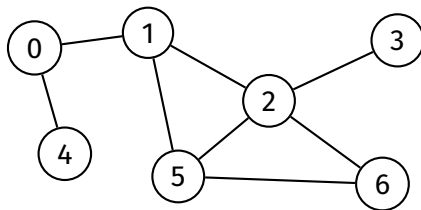
Other  
Problems

## Basic graph problems:

- Is there a cycle?
- How many connected components are there?
- Is there a simple path/cycle that passes through all vertices?
- Is there a path/cycle that passes through each edge exactly once?

Cycle  
CheckingAttempt 1  
Attempt 2  
Solution  
AnalysisConnected  
ComponentsHamiltonian  
Path/CircuitEulerian  
Path/CircuitOther  
Problems

A **cycle** is a path of length  $> 2$   
where the start vertex = end vertex  
and no edge is used more than once



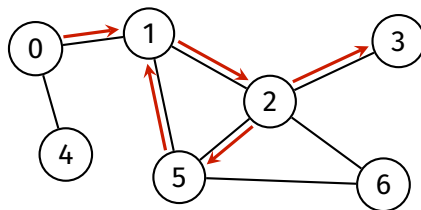
This graph has three distinct cycles:  
1-2-5-1, 2-5-6-2, 1-2-6-5-1

(two cycles are distinct if they have different sets of edges)

## How to check if a graph has a cycle?

### Idea:

- Perform a DFS, starting from any vertex
- During the DFS, if the current vertex has an edge to an already-visited vertex, then there is a cycle



tests/cycle1.txt

```
hasCycle( $G$ ):
```

```
    Input: graph  $G$ 
```

```
    Output: true if  $G$  has a cycle, false otherwise
```

```
    pick any vertex  $v$  in  $G$ 
```

```
    create visited array, initialised to false
```

```
    return dfsHasCycle( $G$ ,  $v$ , visited)
```

```
dfsHasCycle( $G$ ,  $v$ , visited):
```

```
    visited[ $v$ ] = true
```

```
    for each neighbour  $w$  of  $v$  in  $G$ :
```

```
        if visited[ $w$ ] = true:
```

```
            return true
```

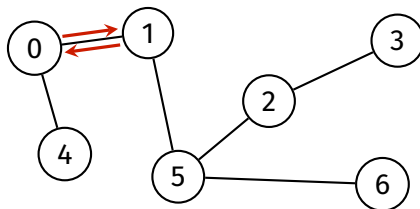
```
        else if dfsHasCycle( $G$ ,  $w$ , visited):
```

```
            return true
```

```
    return false
```

## Problem:

- The algorithm does not check whether the neighbour  $w$  is the vertex that it just came from
- Therefore, it considers moving back and forth along a single edge to be a cycle (e.g., 0-1-0)



tests/cycle2.txt

Improved idea:

- Perform a DFS, starting from any vertex
- **Keep track of previous vertex during DFS**
- During the DFS, if the current vertex has an edge to an already-visited vertex **which is not the previous vertex**, then there is a cycle

```
hasCycle( $G$ ):
```

```
    Input: graph  $G$ 
```

```
    Output: true if  $G$  has a cycle, false otherwise
```

```
    pick any vertex  $v$  in  $G$ 
```

```
    create visited array, initialised to false
```

```
    return dfsHasCycle( $G$ ,  $v$ ,  $v$ , visited)
```

```
dfsHasCycle( $G$ ,  $v$ ,  $prev$ , visited):
```

```
    visited[ $v$ ] = true
```

```
    for each neighbour  $w$  of  $v$  in  $G$ :
```

```
        if  $w = prev$ :
```

```
            continue
```

```
        if visited[ $w$ ] = true:
```

```
            return true
```

```
        else if dfsHasCycle( $G$ ,  $w$ ,  $v$ , visited):
```

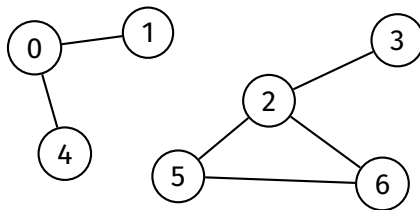
```
            return true
```

```
    return false
```



## Problem:

- The algorithm only checks one connected component
  - The connected component that the initially chosen vertex belongs to



tests/cycle3.txt

## Working idea:

- Perform a DFS, starting from any vertex
- Keep track of previous vertex during DFS
- During the DFS, if the current vertex has an edge to an already-visited vertex which is not the previous vertex, then there is a cycle
- **After the DFS, if any vertex has not yet been visited, perform another DFS, this time starting from that vertex**
- **Repeat until all vertices have been visited**

```
hasCycle( $G$ ):
```

```
    Input: graph  $G$ 
```

```
    Output: true if  $G$  has a cycle, false otherwise
```

```
    create visited array, initialised to false
```

```
    for each vertex  $v$  in  $G$ :
```

```
        if visited[ $v$ ] = false:
```

```
            if dfsHasCycle( $G$ ,  $v$ ,  $v$ , visited):
```

```
                return true
```

```
    return false
```

```
dfsHasCycle( $G$ ,  $v$ ,  $prev$ , visited):
```

```
    visited[ $v$ ] = true
```

```
    for each neighbour  $w$  of  $v$  in  $G$ :
```

```
        if  $w = prev$ :
```

```
            continue
```

```
        if visited[ $w$ ] = true:
```

```
            return true
```

```
        else if dfsHasCycle( $G$ ,  $w$ ,  $v$ , visited):
```

```
            return true
```

```
    return false
```

Cycle  
Checking

Attempt 1

Attempt 2

Solution

Analysis

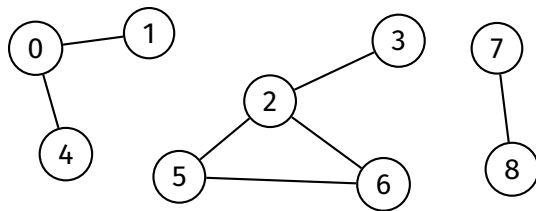
Connected  
ComponentsHamiltonian  
Path/CircuitEulerian  
Path/CircuitOther  
Problems

Analysis for adjacency list representation:

- Algorithm is a slight modification of DFS
- A full DFS traversal is  $O(V + E)$
- Thus, worst-case time complexity of cycle checking is  $O(V + E)$

A **connected component**  
is a maximally connected subgraph

For example, this graph has three connected components:



## DEFINITIONS:

### subgraph

a subset of vertices and edges of original graph

### connected subgraph

there is a path between every pair of vertices in the subgraph

### maximally connected subgraph

no way to include more edges/vertices from original graph into the subgraph  
such that subgraph is still connected

Cycle  
Checking

**Connected  
Components**

Hamiltonian  
Path/Circuit

Eulerian  
Path/Circuit

Other  
Problems

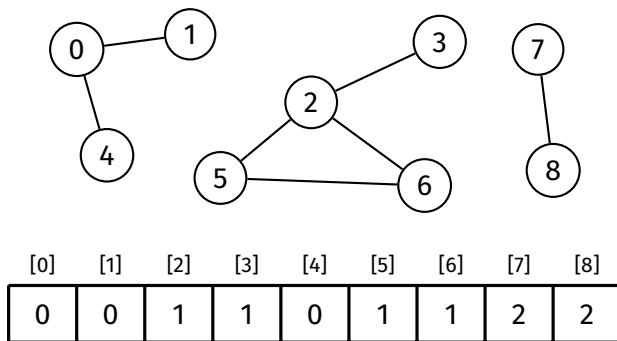
Problems:

How many connected components are there?

Are two vertices in the same connected component?

## Goal:

- Compute an array which indicates which connected component each vertex is in
  - Let this array be called `componentOf`
  - `componentOf[v]` contains the component number of vertex  $v$
- For example:





## Idea:

- Choose a vertex and perform a DFS starting at that vertex
  - During the DFS, assign all vertices visited to component 0
- After the DFS, if any vertex has not been assigned a component, perform a DFS starting at that vertex
  - During this DFS, assign all vertices visited to component 1
- Repeat until all vertices are assigned a component, increasing the component number each time

```
components( $G$ ):  
    Input: graph  $G$   
    Output: componentOf array  
  
    create componentOf array, initialised to -1  
  
    compNo = 0  
    for each vertex  $v$  in  $G$ :  
        if componentOf[ $v$ ] = -1:  
            dfsComponents( $G$ ,  $v$ , componentOf, compNo)  
            compNo = compNo + 1  
  
    return componentOf  
  
dfsComponents( $G$ ,  $v$ , componentOf, compNo):  
    componentOf[ $v$ ] = compNo  
    for each neighbour  $w$  of  $v$  in  $G$ :  
        if componentOf[ $w$ ] = -1:  
            dfsComponents( $G$ ,  $w$ , componentOf, compNo)
```

Analysis for adjacency list representation:

- Algorithm performs a full DFS, which is  $O(V + E)$

Suppose we frequently need to answer the following questions:

- How many connected components are there?
- Are  $v$  and  $w$  in the same connected component?
- Is there a path between  $v$  and  $w$ ?

Note: The last two questions are actually equivalent in an undirected graph.

## Solution:

- Cache the components array in the graph struct

```
struct graph {  
    ...  
    int nC; // number of connected components  
    int *cc; // componentOf array  
};
```

This allows us to answer the questions very easily:

```
// How many connected components are there?  
int numComponents(Graph g) {  
    return g->nC;  
}  
  
// Are v and w in the same connected component?  
bool inSameComponent(Graph g, Vertex v, Vertex w) {  
    return g->cc[v] == g->cc[w];  
}  
  
// Is there a path between v and w?  
bool hasPath(Graph g, Vertex v, Vertex w) {  
    return g->cc[v] == g->cc[w];  
}
```

However, this information needs to be maintained as the graph changes:

- Inserting an edge may reduce  $nC$ 
  - If the endpoint vertices were in different components
- Removing an edge may increase  $nC$ 
  - If the endpoint vertices were in the same component *and* there is no other path between them

Cycle  
Checking

Connected  
Components

Hamiltonian  
Path/Circuit

Eulerian  
Path/Circuit

Other  
Problems

A **Hamiltonian path** is  
a path that includes each vertex exactly once

A **Hamiltonian circuit** is  
a cycle that includes each vertex exactly once



Cycle  
Checking

Connected  
Components

Hamiltonian  
Path/Circuit

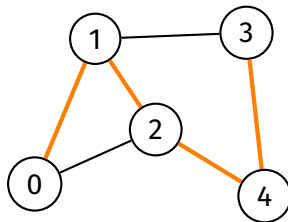
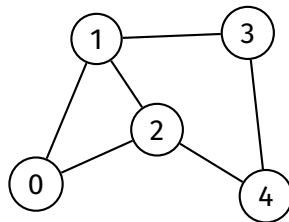
Eulerian  
Path/Circuit

Other  
Problems

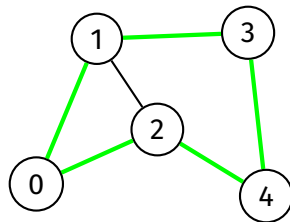
Named after  
Irish mathematician, astronomer and physicist  
Sir William Rowan Hamilton (1805-1865)



Consider the following graph:



Hamiltonian path



Hamiltonian circuit

## How to check if a graph has a Hamiltonian path?

### Idea:

- Brute force
- Use DFS to check all possible paths
  - Recursive DFS is perfect, as it naturally allows backtracking
- Keep track of the number of vertices left to visit
- Stop when this number reaches 0

```
hasHamiltonianPath( $G$ ):
```

```
    Input: graph  $G$ 
```

```
    Output: true if  $G$  has a Hamiltonian path  
            false otherwise
```

```
    create visited array, initialised to false
```

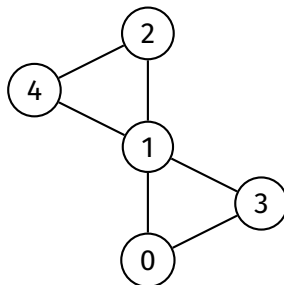
```
    for each vertex  $v$  in  $G$ :
```

```
        if dfsHamiltonianPath( $G$ ,  $v$ , visited, #vertices( $G$ )):  
            return true
```

```
    return false
```

```
dfsHamiltonianPath( $G$ ,  $v$ , visited, numVerticesLeft):  
    visited[ $v$ ] = true  
    numVerticesLeft = numVerticesLeft - 1  
  
    if numVerticesLeft = 0:  
        return true  
  
    for each neighbour  $w$  of  $v$  in  $G$ :  
        if visited[ $w$ ] = false:  
            if dfsHamiltonianPath( $G$ ,  $w$ , visited, numVerticesLeft):  
                return true  
  
    visited[ $v$ ] = false  
    return false
```

Why set `visited[v]` to false at the end of `dfsHamiltonianPath`?



How to check if a graph has a Hamiltonian *circuit*?

- Similar approach as Hamiltonian path
- Don't need to try all starting vertices
- After a Hamiltonian path is found, check if the final vertex is adjacent to the starting vertex

```
hasHamiltonianCircuit( $G$ ):
```

```
    Input: graph  $G$ 
```

```
    Output: true if  $G$  has a Hamiltonian circuit  
             false otherwise
```

```
    if #vertices( $G$ ) < 3:
```

```
        return false
```

```
    create visited array, initialised to false
```

```
    return dfsHamiltonianCircuit( $G$ , 0, visited, #vertices( $G$ ))
```

```
dfsHamiltonianCircuit( $G$ ,  $v$ , visited, numVerticesLeft):
```

```
    visited[ $v$ ] = true
```

```
    numVerticesLeft = numVerticesLeft - 1
```

```
    if numVerticesLeft = 0 and adjacent( $G$ ,  $v$ , 0):
```

```
        return true
```

```
    for each neighbour  $w$  of  $v$  in  $G$ :
```

```
        if visited[ $w$ ] = false:
```

```
            if dfsHamiltonianCircuit( $G$ ,  $w$ , visited, numVerticesLeft):
```

```
                return true
```

```
    visited[ $v$ ] = false
```

```
    return false
```

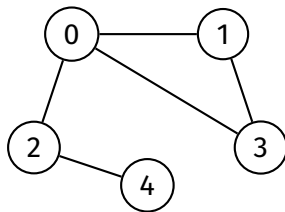


## Analysis:

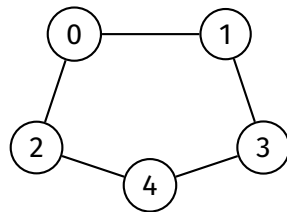
- Worst-case time complexity:  $O(V!)$
- There are at most  $V!$  paths to check ( $\approx \sqrt{2\pi V} (V/e)^V$  by Stirling's approximation)
- There is no known polynomial time algorithm, so the Hamiltonian path problem is NP-hard

An **Eulerian path** is  
a path that visits each edge exactly once

An **Eulerian circuit** is  
an Eulerian path that starts and ends at the same vertex



Eulerian path:  
4-2-0-1-3-0

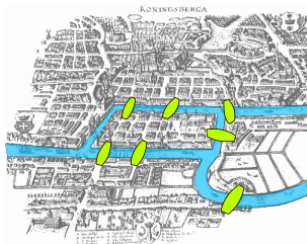


Eulerian circuit:  
4-2-0-1-3-4

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



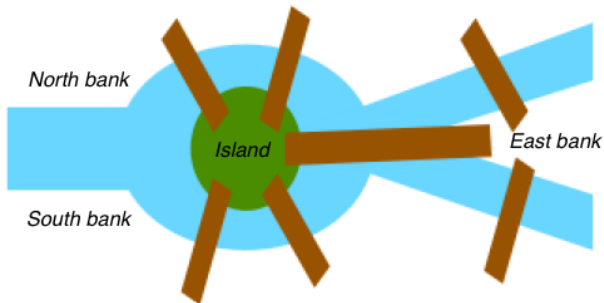
Problem was introduced by Euler while trying to solve the Seven Bridges of Königsberg problem in 1736.



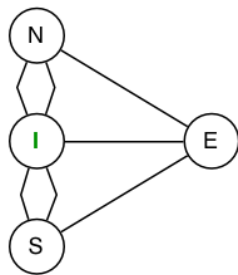
Is there a way to cross all the bridges exactly once on a walk through the town?

Cycle  
CheckingConnected  
ComponentsHamiltonian  
Path/CircuitEulerian  
Path/CircuitOther  
Problems

This is a graph problem:  
**vertices** represent pieces of land  
**edges** represent bridges



Bridges as schematic



Bridges as graph

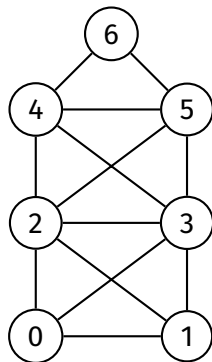
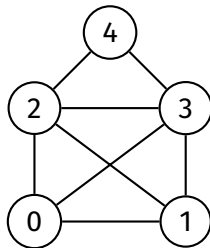
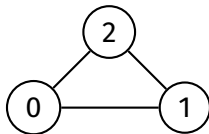
How to check if a graph has an Eulerian path or circuit?

Can use the following theorems:

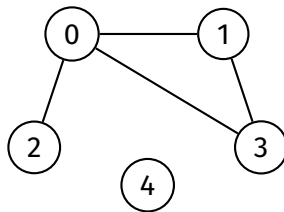
A graph has an **Eulerian path** if and only if  
exactly zero or two vertices have odd degree,  
and all vertices with non-zero degree belong to the same connected  
component

A graph has an **Eulerian circuit** if and only if  
every vertex has even degree,  
and all vertices with non-zero degree belong to the same connected  
component

Which of these graphs have an Eulerian path? How about an Eulerian circuit?



Why  
“all vertices with non-zero degree belong to the same connected component”?





Cycle  
CheckingConnected  
ComponentsHamiltonian  
Path/CircuitEulerian  
Path/CircuitOther  
Problems

```
hasEulerianPath( $G$ ):  
    Input: graph  $G$   
    Output: true if  $G$  has an Eulerian path  
               false otherwise  
  
    numOddDegree = 0  
    for each vertex  $v$  in  $G$ :  
        if degree( $G$ ,  $v$ ) is odd:  
            numOddDegree = numOddDegree + 1  
  
    return (numOddDegree = 0 or numOddDegree = 2) and  
           eulerConnected( $G$ )
```

```
eulerConnected( $G$ ):  
    Input: graph  $G$   
    Output: true if all vertices in  $G$  with non-zero degree  
              belong to the same connected component  
              false otherwise  
  
    create visited array, initialised to false  
  
    for each vertex  $v$  in  $G$ :  
        if degree( $G$ ,  $v$ ) > 0:  
            dfsRec( $G$ ,  $v$ , visited)  
            break  
  
    for each vertex  $v$  in  $G$ :  
        if degree( $G$ ,  $v$ ) > 0 and visited[ $v$ ] = false:  
            return false  
  
    return true
```

Cycle  
CheckingConnected  
ComponentsHamiltonian  
Path/CircuitEulerian  
Path/CircuitOther  
Problems

```
hasEulerianCircuit( $G$ ):
```

```
    Input: graph  $G$ 
```

```
    Output: true if  $G$  has an Eulerian circuit  
             false otherwise
```

```
    for each vertex  $v$  in  $G$ :
```

```
        if degree( $G$ ,  $v$ ) is odd:  
            return false
```

```
    return eulerConnected( $G$ )
```

### Analysis for adjacency list representation:

- Finding degree of every vertex is  $O(V + E)$
- Checking connectivity requires a DFS which is  $O(V + E)$
- Therefore, worst-case time complexity is  $O(V + E)$

So unlike the Hamiltonian path problem, the Eulerian path problem can be solved in polynomial time.

Cycle  
Checking

Connected  
Components

Hamiltonian  
Path/Circuit

Eulerian  
Path/Circuit

Other  
Problems

Many graph problems are **intractable** – that is,  
there is no known “efficient” algorithm to solve them.

In this context, “efficient” usually means polynomial time.

A **tractable** problem is one that is known to have a  
polynomial-time solution.

Cycle  
Checking

Connected  
Components

Hamiltonian  
Path/Circuit

Eulerian  
Path/Circuit

Other  
Problems

tractable

what is the shortest path  
between two vertices?

intractable

how about the *longest* path?

Cycle  
Checking

Connected  
Components

Hamiltonian  
Path/Circuit

Eulerian  
Path/Circuit

Other  
Problems

tractable

what is the shortest path  
between two vertices?

does a graph contain a clique?

intractable

how about the *longest* path?

what is the *largest* clique?

Cycle  
Checking

Connected  
Components

Hamiltonian  
Path/Circuit

Eulerian  
Path/Circuit

Other  
Problems

tractable

what is the shortest path  
between two vertices?

does a graph contain a clique?

given two colors, is it possible to  
colour every vertex in a graph such  
that no two adjacent vertices are the  
same colour?

intractable

how about the *longest* path?

what is the *largest* clique?

what about *three* colours?



Cycle  
Checking

Connected  
Components

Hamiltonian  
Path/Circuit

Eulerian  
Path/Circuit

Other  
Problems

tractable

what is the shortest path  
between two vertices?

does a graph contain a clique?

given two colors, is it possible to  
colour every vertex in a graph such  
that no two adjacent vertices are the  
same colour?

does a graph contain an Eulerian path?

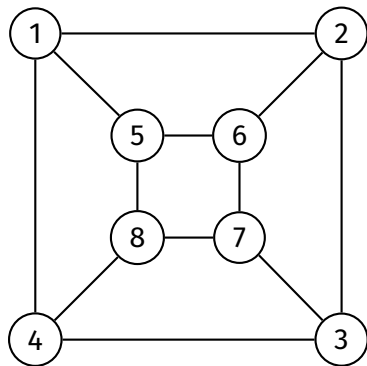
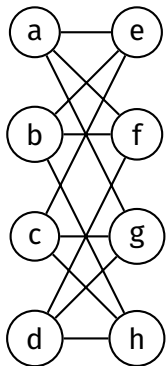
intractable

how about the *longest* path?

what is the *largest* clique?

what about *three* colours?

how about a Hamiltonian path?

Cycle  
CheckingConnected  
ComponentsHamiltonian  
Path/CircuitEulerian  
Path/CircuitOther  
Problems

**Graph isomorphism:**

Can we make two given graphs identical by renaming vertices?