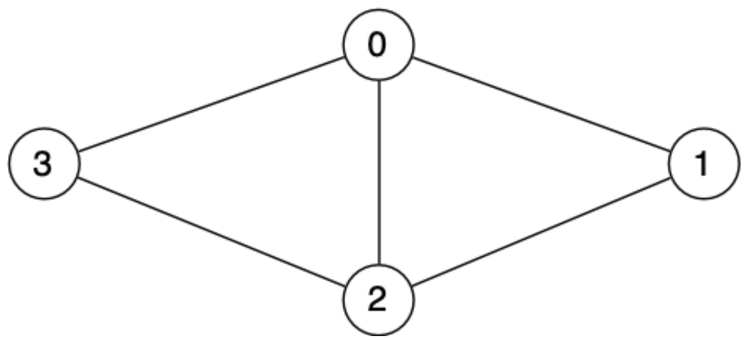
Graph Algorithms Intro

❖ Cycle Checking

A graph has a cycle if

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

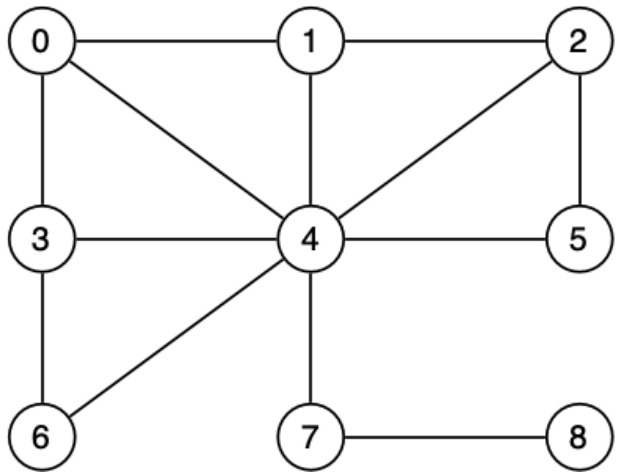
This graph has 3 distinct cycles: 0-1-2-0, 2-3-0-2, 0-1-2-3-0



("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, ...

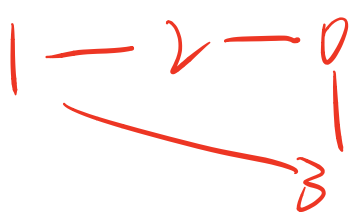
❖ ... Cycle Checking

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
```

v->w

return true



visited | if v=1 w=2

1,2 v=2 w=0, 3 true

1,2,0 v=0 w=2 3 v=2 true v=3

1,2,0,3 v=3 w=2, 0

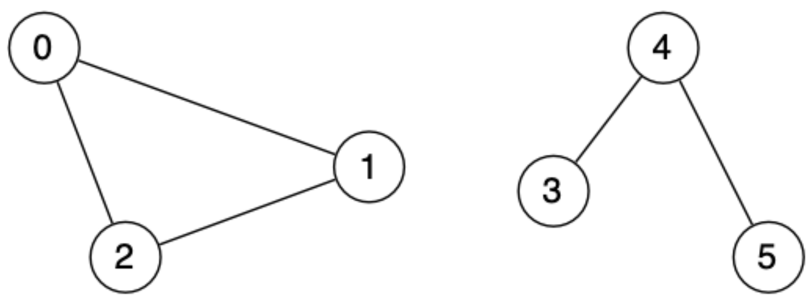
true true

❖ ... 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:



Connected Component #1 *Connected Component #2*

Bug

V—W

不能用一个 path

❖ ... Cycle Checking

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

```
                                                      v to u
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]) {没走过w
                                  u
            if (dfsCycleCheck(g, w, v))
               return true;
         }
         else if (w != u) 走过w,
            return true;
      }
   }
   return false;
}
```

v=1 u=1 w=2

v=2 { w=0 < 2 true, 3 true ; w=1 走过, w=u false ; w=3 true }

❖ ... 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;
}
```

true

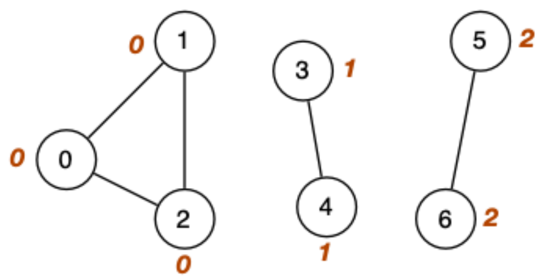
❖ Connected Components

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

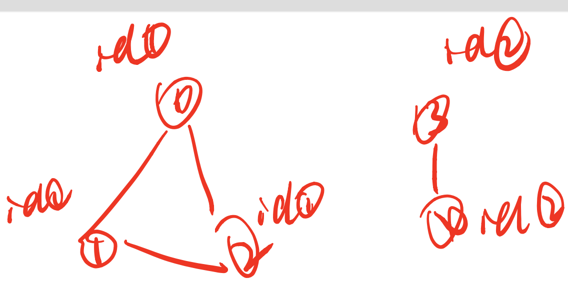
Algorithm to assign vertices to connected components:

```
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
```

可能所有
v

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

v->v

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

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

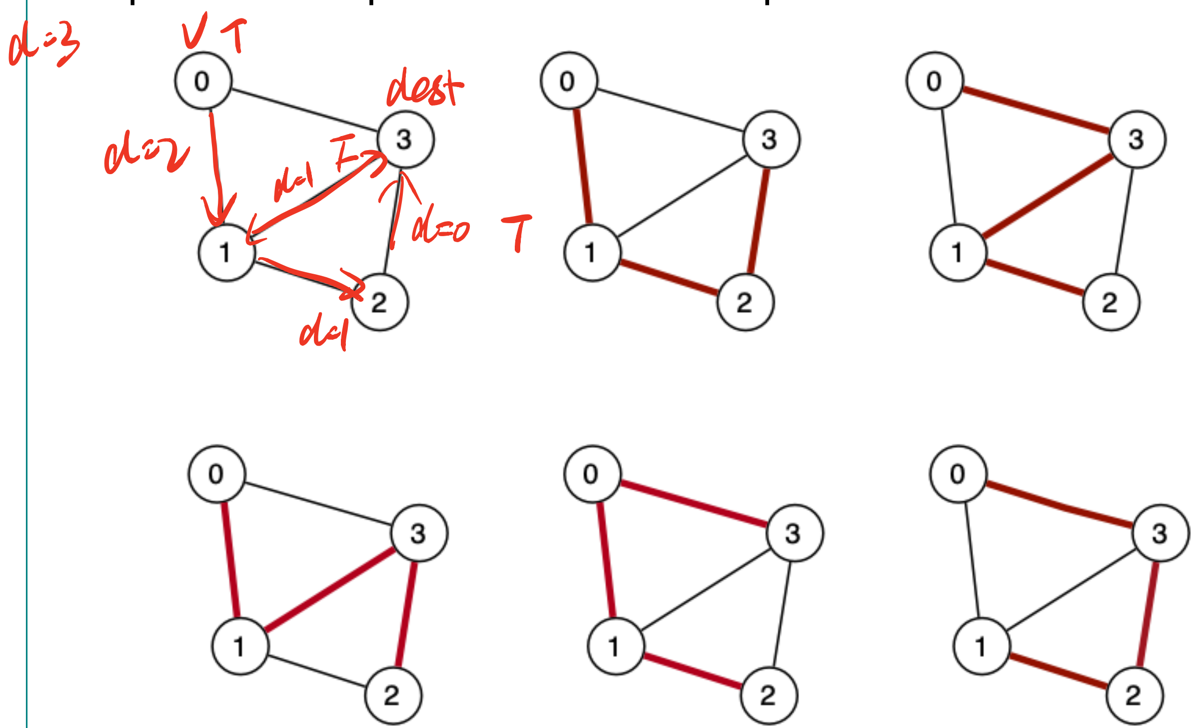
Many real-world applications require you to visit all vertices of a graph:

- Travelling salesman
- Bus routes go to every stop.
- ...

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

❖ ... Hamiltonian Path and Circuit

Graph and some possible Hamiltonian paths:



要过所有的 V 一遍

❖ ... 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:

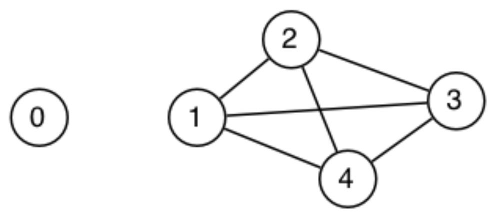
```
visited[] // array [0..nV-1] to keep track of visited vertices

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                                    4-1
|  return hamiltonR(G,src,dest,#vertices(G)-1)
```

```
                  0->2    3
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                  回到起点，并 cover all vetix
|     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.

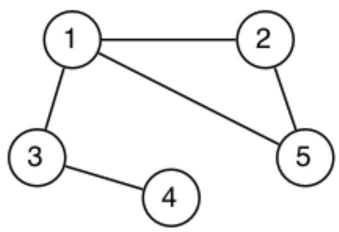
Note, however, that the above case could be solved in constant time if we had a fast check for 0 and *x* being in the same connected component

❖ Euler Path and Circuit

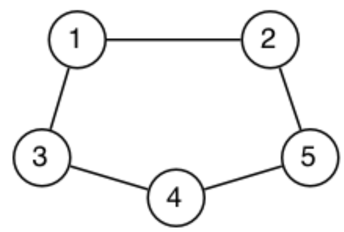
Euler path problem:

- 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

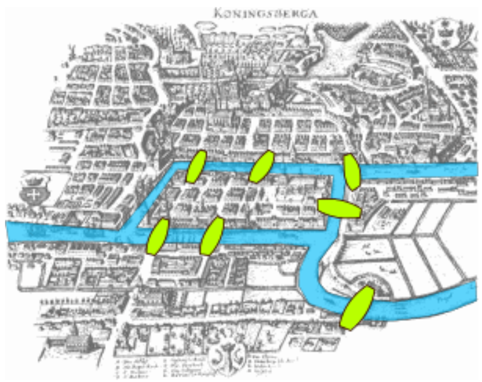
Many real-world applications require you to visit all edges of a graph:

- Postman go every street
- 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

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:

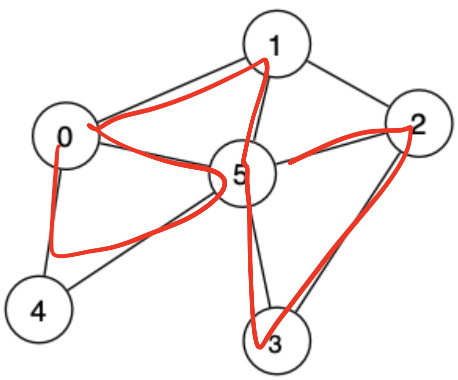
Theorem. A graph has an Euler circuit if and only if it is connected and all vertices have even degree

v w

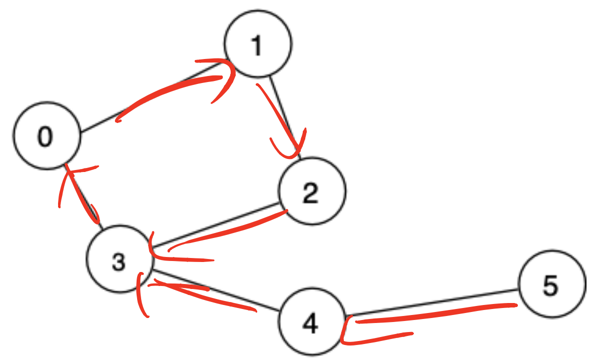
Theorem. A graph has a non-circuitous Euler path if and only if it is connected and exactly two vertices have odd degree

❖ ... 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

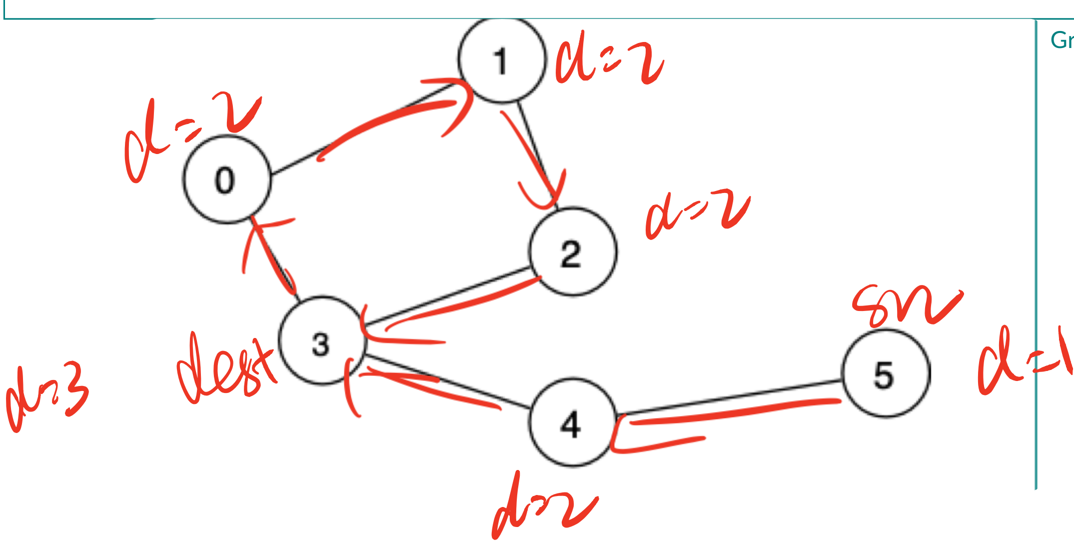
5 - 4 - 3 - 0 - 1 - 2 - 3

❖ ... 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                          or
|     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    and
|     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.

Directed/Weighted Graphs

❖ Generalising Graphs

Discussion so far has considered graphs as

- V = set of vertices, E = set of edges

Real-world applications require more "precision"

- some edges are directional (e.g. one-way streets)
- some edges have a cost (e.g. distance, traffic)

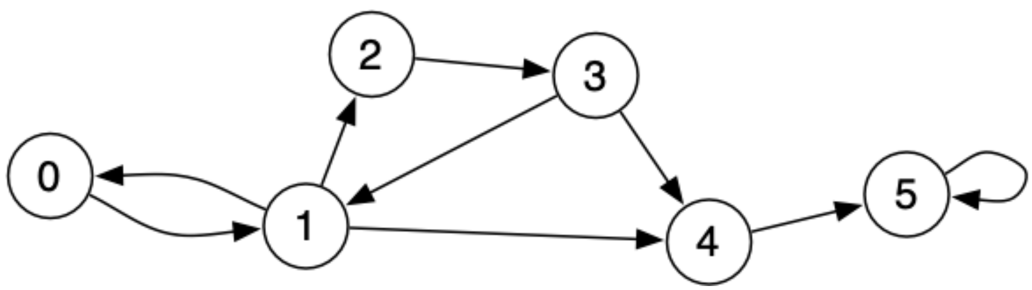
We need to consider directed graphs and weighted graphs

❖ Directed Graphs (Digraphs)

Directed graphs are ...

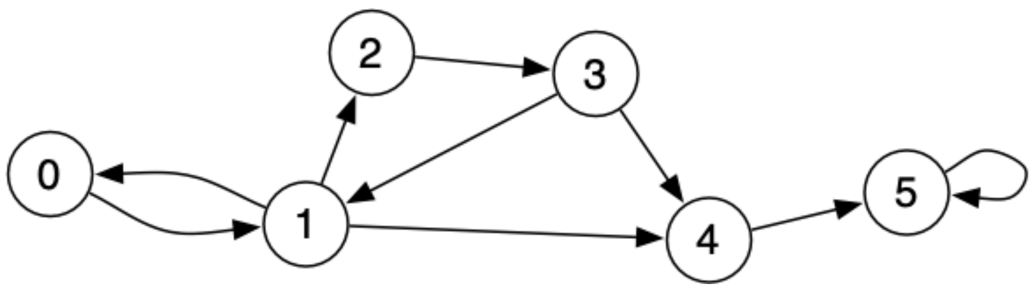
- graphs with *V* vertices, *E* edges *(v,w)*
- edge *(v,w)* has source *v* and destination *w*
- unlike undirected graphs, $v \rightarrow w \neq w \rightarrow v$

Example digraph:



❖ ... Directed Graphs (Digraphs)

Some properties of ...



- edges 1-2-3 form a cycle, edges 1-3-4 do *not* form a cycle
- vertex 5 has a self-referencing edge *(5,5)*
- vertices 0 and 1 reference each other, i.e. *(0,1)* and *(1,0)*
- there are no paths from 5 to any other nodes
- paths from 0→5: 0-1-2-3-4-5, 0-1-4-5, 0-1-2-3-1-4-5

❖ ... Directed Graphs (Digraphs)

Terminology for digraphs ...

Directed path: sequence of $n \geq 2$ vertices $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_n$

- where $(v_i,v_{i+1}) \in edges(G)$ for all v_i,v_{i+1} in sequence

If $v_1 = v_n$, we have a directed cycle

Degree of vertex: number of incident edges

$v \rightarrow$

- outdegree: $deg(v)$ = number of edges of the form $(v, \_)$
- indegree: $deg^{-1}(v)$ = number of edges of the form $(\_, v)$ $\rightarrow v$

❖ ... Directed Graphs (Digraphs)

More terminology for digraphs ...

Reachability:

- w is reachable from v if ∃ directed path $v,\dots,w$

Strong connectivity:

- every vertex is reachable from every other vertex

Directed acyclic graph (DAG):

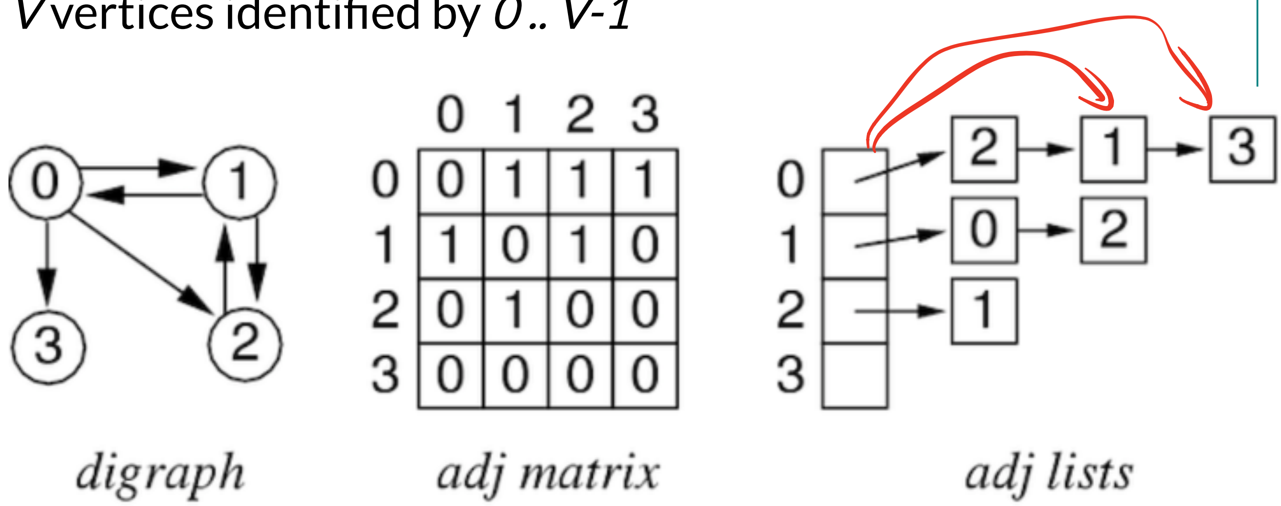
- contains no directed cycles

❖ Digraph Representation

Similar set of choices as for undirectional graphs:

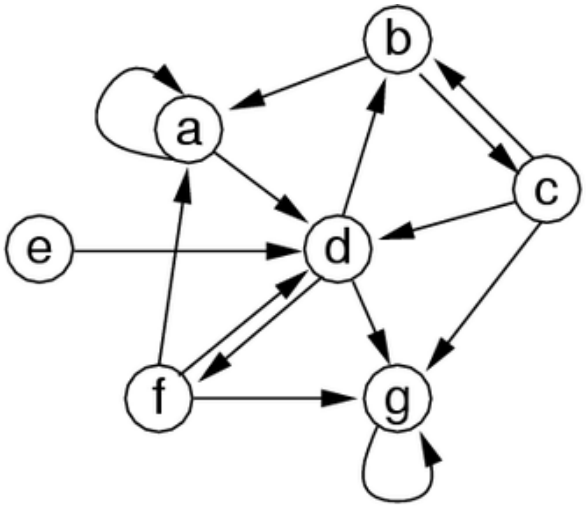
- array of edges (directed)
- vertex-indexed adjacency matrix (non-symmetric)
- vertex-indexed adjacency lists

V vertices identified by *0 .. V-1*



❖ ... Digraph Representation

Example digraph and adjacency matrix representation:



	a	b	c	d	e	f	g
a	1	0	0	1	0	0	0
b	1	0	1	0	0	0	0
c	0	1	0	1	0	0	1
d	0	1	0	0	0	1	1
e	0	0	0	1	0	0	0
f	1	0	0	1	0	0	1
g	0	0	0	0	0	0	1

Undirectional ⇒ symmetric matrix

Directional ⇒ non-symmetric matrix

Maximum #edges in a digraph with V vertices: V^2

❖ ... Digraph Representation

Costs of representations: (where degree *deg(v)* = #edges leaving *v*)

	array of edges	adjacency matrix	adjacency list
space usage	*E*	V^2	*V+E*
insert edge	*E*	*1*	*1*
exists edge *(v,w)*?	*E*	*1*	*deg(v)*
get edges leaving *v*	*E*	*V*	*deg(v)*

Overall, adjacency list representation is best

- real graphs tend to be sparse 稀疏
  (large number of vertices, small average degree *deg(v)*)
- algorithms frequently iterate over edges from *v*

❖ Weighted Graphs

Graphs so far have considered

- edge = an association between two vertices/nodes
- may be a 优先权 precedence in the association (directed)

Some applications require us to consider

- a cost or weight of an association
- modelled by assigning values to edges (e.g. positive reals)

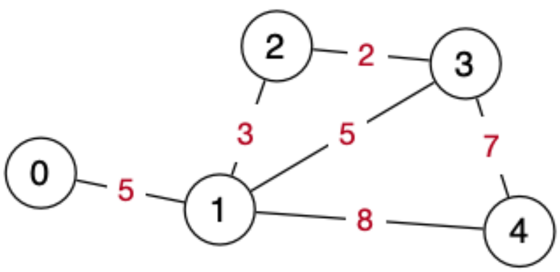
❖ ... Weighted Graphs

Weighted graphs are ...

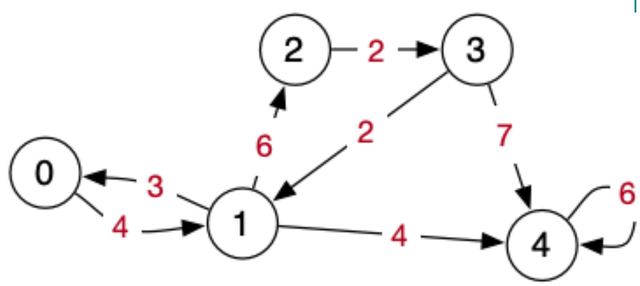
- graphs with *V* vertices, *E* edges *(s,t)*
- each edge *(s,t,w)* connects vertices *s* and *t* and has weight *w*

Weights can be used in both directed and undirected graphs.

Example weighted graphs:



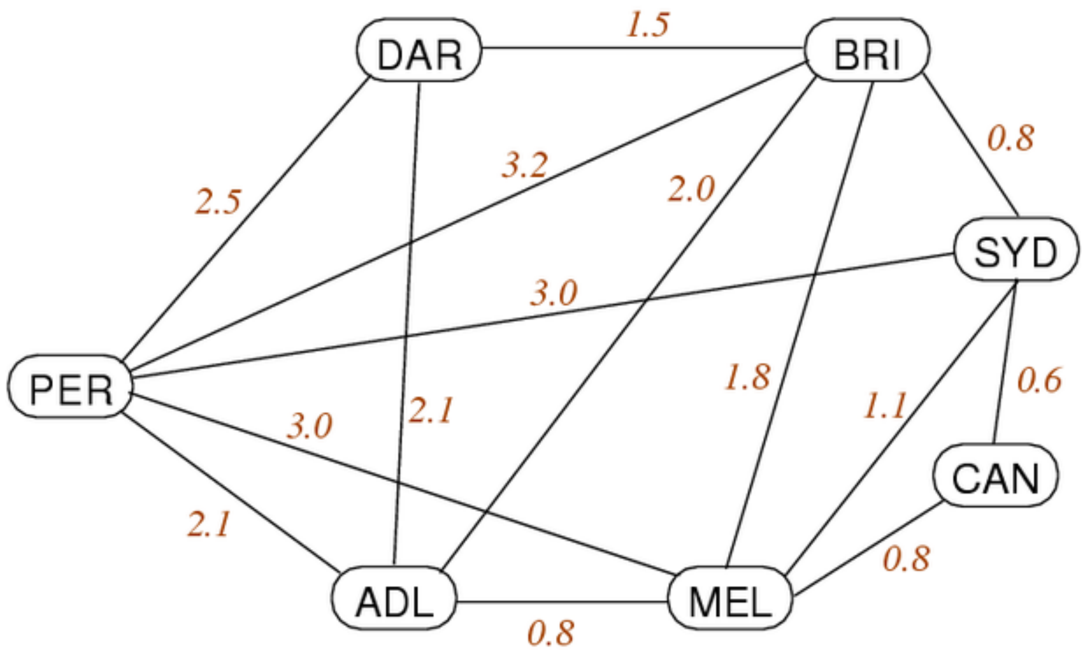
Weighted Graph



Directed Weighted Graph

❖ ... Weighted Graphs

Example: major airline flight routes in Australia

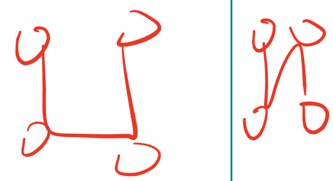


Representation: edge = direct flight; weight = approx flying time (hours)

❖ ... Weighted Graphs

最少边

Weights lead to minimisation-type questions, e.g.



1. Cheapest way to connect all vertices?
   - a.k.a. minimum spanning tree problem
   - assumes: edges are weighted and undirected
2. Cheapest way to get from *A* to *B*?
   - a.k.a shortest path problem
   - assumes: edge weights positive, directed or undirected

❖ Weighted Graph Representation

Weights can easily be added to:

- adjacency matrix representation (0/1 $\rightarrow$ int or float)
- adjacency lists representation (add int/float to list node)

The edge list representation changes to list of *(s,t,w)* triples

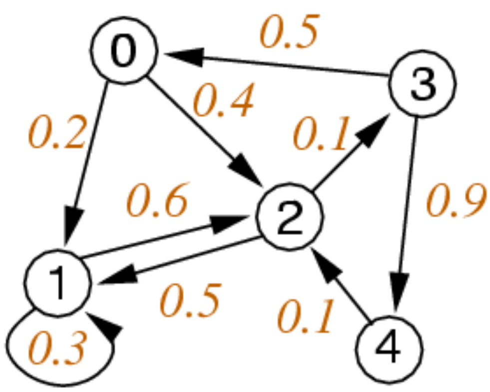
All representations can also work with directed edges

Weight values are determined by domain being modelled

- in some contexts weight could be zero or negative

❖ ... Weighted Graph Representation

Adjacency matrix representation with weights:



Weighted Digraph

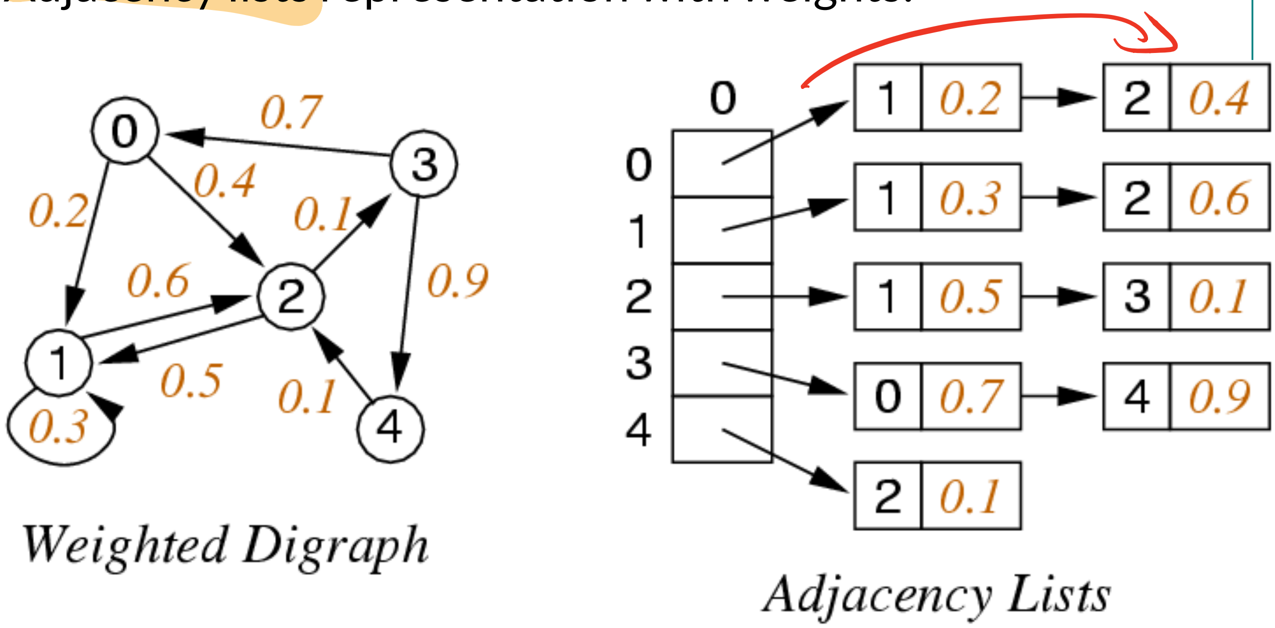
	0	1	2	3	4
0	*	0.2	0.4	*	*
1	*	0.3	0.6	*	*
2	*	0.5	*	0.1	*
3	0.5	*	*	*	0.9
4	*	*	0.1	*	*

Adjacency Matrix

Note: need distinguished value to indicate "no edge".

❖ ... Weighted Graph Representation

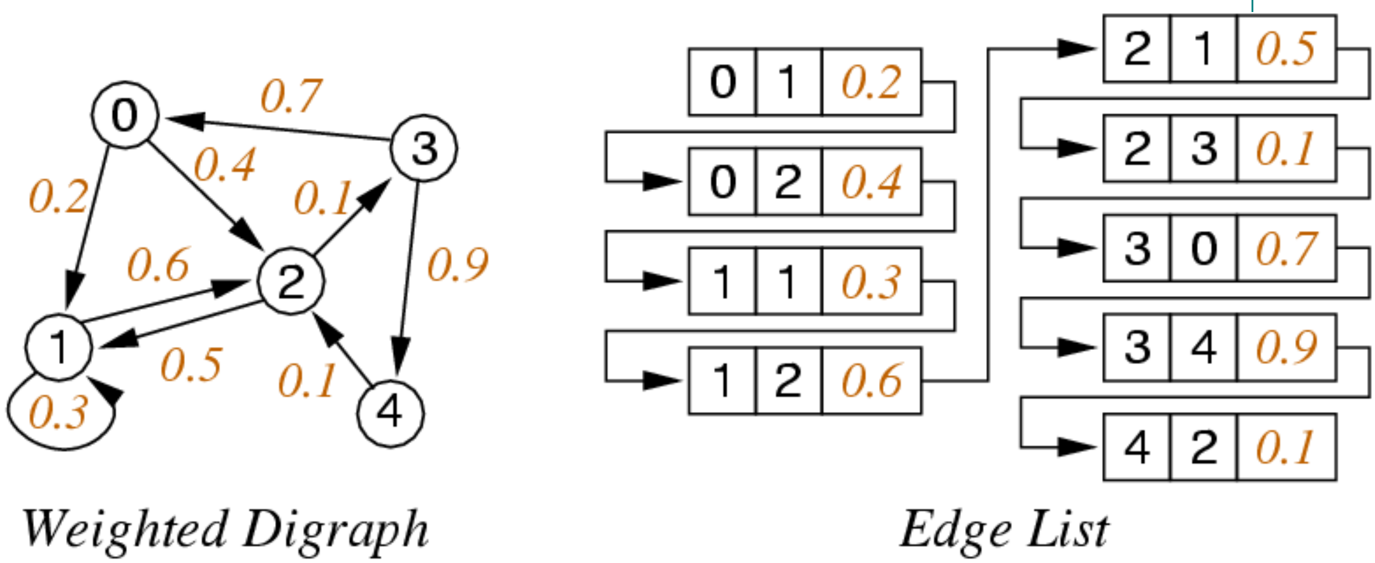
Adjacency lists representation with weights:



Note: if undirected, each edge appears twice with same weight

❖ ... Weighted Graph Representation

Edge array / edge list representation with weights:



Note: not very efficient for use in processing algorithms, but does give a possible representation for min spanning trees or shortest paths

❖ Weighted Graph Implementation

Changes to preious grpah data structures to include weights:

WGraph.h

```
// edges are pairs of vertices (end-points) plus weight
typedef struct Edge {
    Vertex v;
    Vertex w;
    int    weight;
} Edge;

// returns weight, or 0 if vertices not adjacent
int adjacent(Graph, Vertex, Vertex);
```

有无 edge

Note: here, we assume all weights are positive, but not required

❖ ... Weighted Graph Implementation

WGraph.c (assuming adjacency matrix representation)

```
typedef struct GraphRep {
   int **edges;   // adjacency matrix storing weights
                  // 0 if nodes not adjacent
   int   nV;      // #vertices
   int   nE;      // #edges
} GraphRep;

bool adjacent(Graph g, Vertex v, Vertex w) {
   assert(valid graph, valid vertices)
   return (g->edges[v][w] != 0);
}
```

❖ ... Weighted Graph Implementation

More **WGraph.c**

```
void insertEdge(Graph g, Edge e) {
   assert(valid graph, valid edge)
   // edge e not already in graph
   if (g->edges[e.v][e.w] == 0) {g->nE++;}
   // may change weight of existing edge
   g->edges[e.v][e.w] = e.weight;
   g->edges[e.w][e.v] = e.weight;
}

void removeEdge(Graph g, Edge e) {
   assert(valid graph, valid edge)
   // edge e not in graph
   if (g->edges[e.v][e.w] == 0) {return;}
   g->edges[e.v][e.w] = 0;
   g->edges[e.w][e.v] = 0;
   g->nE--;
}
```