Graph Data Structures and Algorithms

Many applications require

- a collection of items (i.e. a set)
- relationships/connections between items

Collection types you're familiar with

- lists: linear sequence of items (week 3, COMP9021)
- trees: branched hierachy of items (COMP9021)
  But Graphs are more general and allow arbitrary connections.

Graph algorithms are generally more complex than tree/list ones:

- <u>no implicit order</u> of items
- graphs may contain <u>cycles</u> (which trees do not have)
- concrete representation is less obvious
- algorithm complexity depends on connection complexity

Category of graphs:

- simple graph
- directed graph
- weighted graph

Common algorithms for graphs:

1. connectivity (simple graphs)

2. path finding (simple/directed graphs)
3. minimum spanning trees (weighted graphs)
4. shortest path (weighted graphs)

Graph Terminology and Property

Terminology

A ***Graph*** $G = (V, E)$

- V is a set of ***vertices***
- E is a set of ***edges*** (subset of $V \times V$)

For an edge e that connects vertices v and w

- v and w are adjacent (neighbours)
- e is incident on both v and w

Degree of a vertex v: number of edges incident on e

Synonyms:
vertex = node, edge = arc = link (Note: some people use arc for directed edges)

Path and Cycle

- ***Path***: a sequence of <u>vertices</u> where each vertex has an edge to its predecessor.
- ***Cycle***: a path where <u>last vertex</u> in path is same as <u>first vertex</u> in path
- ***Length*** of path or cycle: <u>#edges</u>

Actually, a tree is a special graph

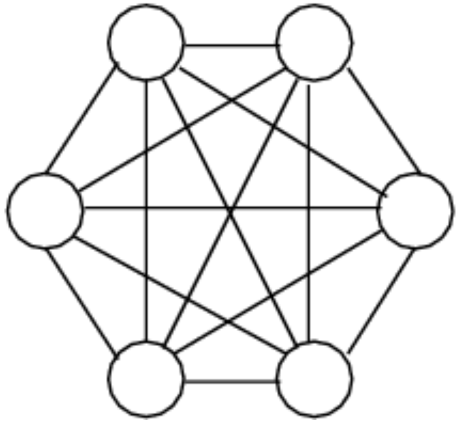
- no cycle
- every vertex has just 1 edge

Graph Terminology

Types of Graph

Connected Graph and Complete Graph:

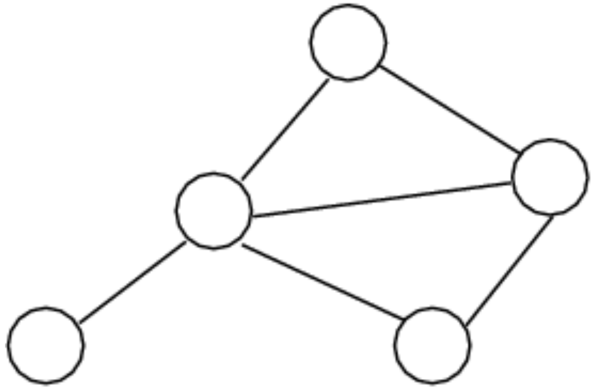
- ***Connected graph***
  - there is a *path* from each vertex to every other vertex
  - if a graph is not connected, it has ≥2 connected components
- ***Complete graph** KV*
  - there is an *edge* from each vertex to every other vertex
  - in a complete graph, $E = V(V-1)/2$



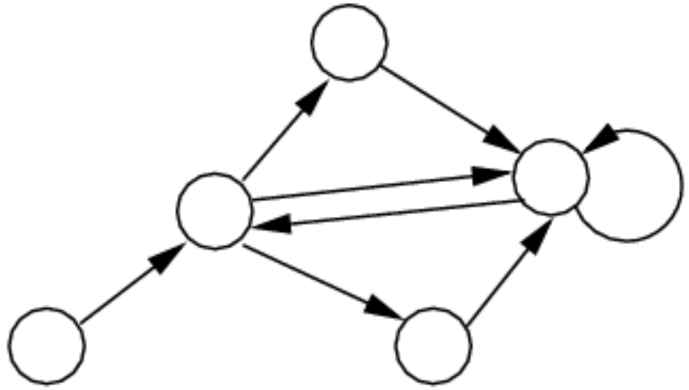
Complete Graph

Undirected Graph and Directed Graph:

- ***Undirected graph***: *edge(u,v)* = *edge(v,u)*, no self-loops (i.e. no *edge(v,v)*)
- ***Directed graph***: *edge(u,v)* ≠ *edge(v,u)*, can have self-loops (i.e. *edge(v,v)*)



Undirected graph



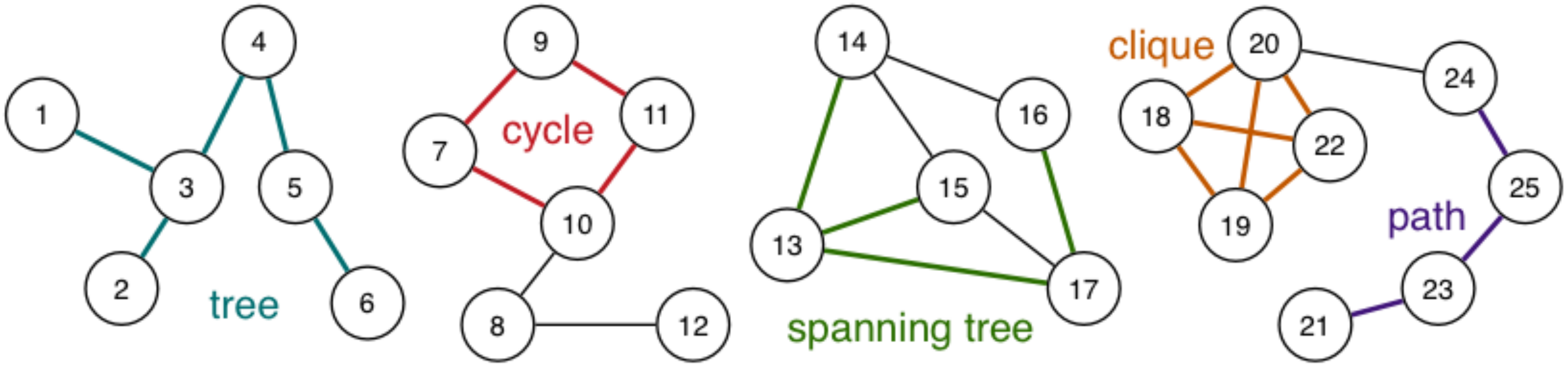
Directed graph

Weighted and Multi Graph:

- ***Weighted graph***
  - each edge has an associated value (weight)
  - e.g. road map (weights on edges are distances between cities)
- ***Multi-graph***: allow multiple edges between two verticese.g. function call graph (`f()` calls `g()` in several places)

Subgraph

- ***Tree***: connected (sub)graph with no cycles
- ***Spanning tree***: tree containing all vertices
- ***Clique***: complete subgraph



Property

A graph with V vertices has at most $V(V-1)/2$ edges.

The ratio $E:V$ can vary considerably:

- if E is closer to V^2, the graph is *dense*
- if E is closer to V, the graph is *sparse*

Graph ADT

Data:

- set of *vertices*: Normally, we use index from `0` to `nV-1` to represent vertex.
- set of *edges*: We have 3 different ways to represent edges: array, matrix and list.

Basic Operations:

- building: create graph, add edge
- deleting: remove edge, drop whole graph

- scanning: check if graph contains a given edge

Things to note:

- set of vertices is fixed when graph initialised
- we treat vertices as `int`s, but could be arbitrary `Item`s

Edges Representation

Represent the edges is a critical part of Graph ADT. We will discuss 3 ways to represent edges:

1. ***Array of edges***: use a E-sized array
2. ***Adjacency matrix***: use a V^2-sized 2d array
3. ***Adjacency list***: use a $O(V^2)$-sized linked list

Array of edges Representation

Graph initialisation:

```
newGraph(V):
|  Input  number of nodes V
|  Output new empty graph
|
|  g.nV = V   // #vertices (numbered 0..V-1)
|  g.nE = 0   // #edges
|  allocate enough memory for g.edges[]
|  return g
```

Edge insertion:

```
insertEdge(g,(v,w)):
```

```
|  Input graph g, edge (v,w)
|
|  i=0
|  while i<g.nE ∧ (v,w)≠g.edges[i] do
|     i=i+1
|  end while
|  if i=g.nE then      // (v,w) not found
|     g.edges[i]=(v,w)
|     g.nE=g.nE+1
|  end if
```

Edge removal:

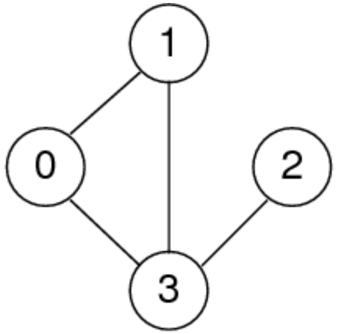
```
removeEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  i=0
|  while i<g.nE ∧ (v,w)≠g.edges[i] do
|     i=i+1
|  end while
|  if i<g.nE then                 // (v,w) found
|     g.edges[i]=g.edges[g.nE-1] // replace by last edge in array
|     g.nE=g.nE-1
|  end if
```

Performance:

- Storage cost: $O(E)$
- Cost of operations:
  - initialisation: $O(1)$
  - insert edge: $O(E)$ (assuming edge array has space)
    - If array is full on insert: allocate space for a bigger array, copy edges across ⇒ still $O(E)$

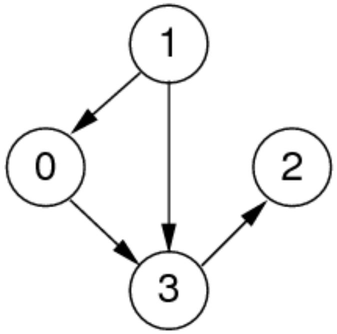
- delete edge: $O(E)$ (need to find edge in edge array)
- If we maintain edges in order: use binary search to find edge ⇒ $O(\log E)$

Adjacency Matrix Representation



Undirected graph

A	0	1	2	3
0	0	1	0	1
1	1	0	0	1
2	0	0	0	1
3	1	1	1	0



Directed graph

A	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	0	0	0	0
3	0	0	1	0

Advantages:

- easily implemented as *2-dimensional array* (quick to access in constant time)
- can represent graphs, digraphs and weighted graphs
  - graphs: symmetric boolean matrix
  - digraphs: non-symmetric boolean matrix

    - weighted: non-symmetric matrix of weight values

Disadvantages:

- if few edges (sparse) ⇒ memory-inefficient (V^2)

Performance:

- Storage cost: $O(V^2)$
    - If the graph is sparse, most storage is wasted.
- Cost of operations:
    - initialisation: $O(V^2)$ (initialise *V*×*V* matrix)
    - insert edge: $O(1)$ (set two cells in matrix)
    - delete edge: $O(1)$ (unset two cells in matrix)

```
struct vNode {
    vertex v;
    struct vNode *next;
}

struct graphRep {
    int nV;
    int nE;
    int **egdes;    // create a 2D array
};

// connect two vertex
g->edges[v][w] = g->edges[w][v] = 1;
```

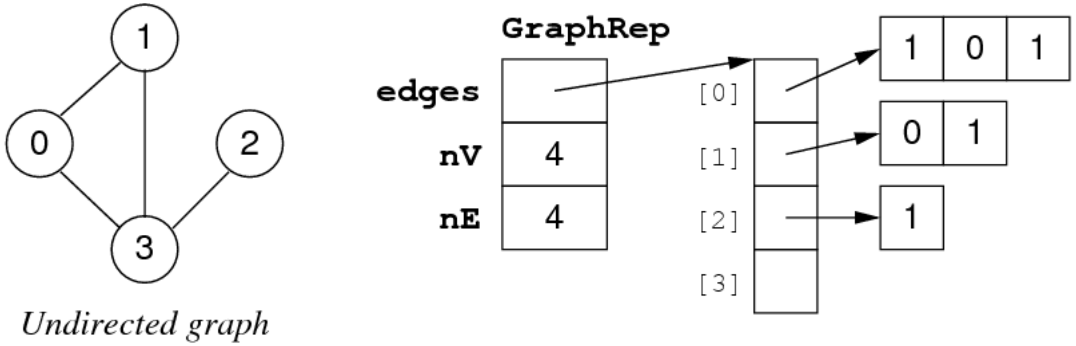
Adjacency List Representation

Most graphs are very sparse. So we

Since for an undirected graph, their edges are symmetric, we can just store half of the matrix.

A storage optimisation: store only top-right part of matrix.

- New storage cost: $V-1$ int ptrs + $V(V+1)/2$ ints (but still $O(V^2)$)
- Requires us to always use edges *(v,w)* such that *v* < *w*.
- implemented by an array and linked list



Performance:

- Storage cost: $O(V+E)$
- Cost of operations:
    - initialisation: $O(V)$ (initialise *V* lists)
    - insert edge: $O(1)$ (insert one vertex into list)
        - if you don't check for duplicates
    - delete edge: $O(E)$ (need to find vertex in list)
    - If vertex lists are sorted:
        - insert requires search of list ⇒ $O(E)$
        - delete always requires a search, regardless of list order

Graph initialisation:

```
newGraph(V):
|  Input  number of nodes V
|  Output new empty graph
|
|  g.nV = V   // #vertices (numbered 0..V-1)
|  g.nE = 0   // #edges
|  allocate memory for g.edges[]
|  for all i=0..V-1 do
|     g.edges[i]=NULL   // empty list
|  end for
|  return g
```

Edge insertion:

```
insertEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if inLL(g.edges[v],w) then  // (v,w) not in graph
|     insertLL(g.edges[v],w)
|     insertLL(g.edges[w],v)
|     g.nE=g.nE+1
|  end if
```

Edge removal:

```
removeEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if inLL(g.edges[v],w) then  // (v,w) in graph
|     deleteLL(g.edges[v],w)
|     deleteLL(g.edges[w],v)
```

```
|     g.nE=g.nE-1
|   end if
```

```
struct vNode {
    vertex v;
    struct vNode *next;
}

struct graphRep {
    int nV;
    int nE;
    int *egdes;    // create a 1D array
};
```

Comparision of Graph Representations

	array of edges	adjacency matrix	adjacency list
space usage	*E*	V^2	*V+E*
initialise	*1*	V^2	*V*
insert edge	*E*	*1*	*1*
remove edge	*E*	*1*	*E*

Other operations:

	array of edges	adjacency matrix	adjacency list
disconnected(v)?	*E*	*V*	*1*

isPath(x,y)? (key operation)	*E·log V* (Best)	V^2	*V+E*
copy graph	*E*	V^2	*V+E*
destroy graph	*1*	*V*	*V+E*

Basic Graph Algo: DFS & BFS

Finding a Path

Questions on paths:

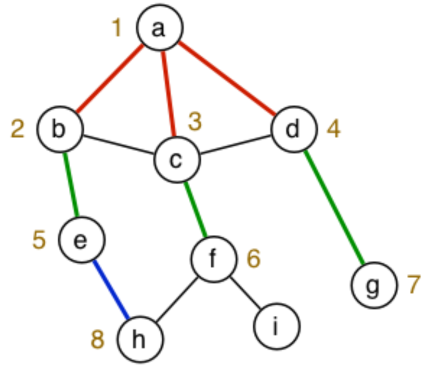
1. is there a path between two given vertices (*src*,*dest*)?
2. what is the path (sequence of vertices) from *src* to *dest*?

Approach to solving problem:

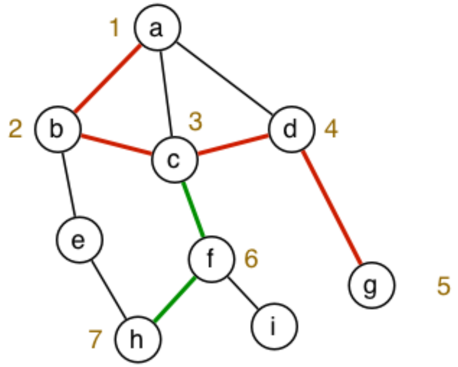
1. examine vertices adjacent to *src*
2. if any of them is *dest*, then done
3. otherwise try vertices two edges from *v*
4. repeat looking further and further from *v*

Two strategies for graph traversal/search: ***depth-first***, ***breadth-first***

- ***DFS*** follows one path to completion before considering others (can be implemented via recursion)
- ***BFS*** "fans-out" from the starting vertex ("spreading" subgraph)
- They just vary on the searching order.



Breadth-first Search



Depth-first Search

Depth-first:

- favour following path rather than neighbours
- <u>can be implemented recursively or iteratively (via *stack*, FILO)</u>
- full traversal produces a *depth-first spanning tree*

Breadth-first:

- favour neighbours rather than path following
- <u>can be implemented iteratively (via *queue*, FIFO)</u>
- full traversal produces a *breadth-first spanning tree*

Compared to trees,

- BFS is like level-order traversal
- DFS is like prefix-order traveral

```
void breadthFirst (Graph g, Vertex v)
{
    int *visited = calloc (g->nV, sizeof (int));
```

```
    Queue q = newQueue ();
    QueueJoin (q, v);
    while (QueueLength (q) > 0) {
        Vertex x = QueueLeave (q);
        if (visited[x])
            continue;
        visited[x] = 1;
        printf ("%d\n", x);
        for (Vertex y = 0; y < g->nV; y++) {
            if (!g->edges[x][y])
                continue;
            if (!visited[y])
                QueueJoin (q, y);
        }
    }
}

void depthFirst (Graph g, Vertex v)
{
    int *visited = calloc (g->nV, sizeof (int));
    Stack s = newStack ();
    StackPush (s, v);
    while (!StackIsEmpty (s)) {
        Vertex x = StackPop (s);
        printf ("%d\n", x);
        for (Vertex y = g->nV - 1; y >= 0; y--) {
            if (!g->edges[x][y])
                continue;
            if (!visited[y]) {
                StackPush (s, y);
                visited[y] = 1;  // this line is important that it can avoid pushing sa
me vertex into a stake
            }
```

```
        }
    }
}
```

DFS

Depth-first search can be described recursively as

depthFirst(G,v):
for each `(v,w)` ∈*edges*(G), if `w` not visited then do:

1. mark `v` as visited and store which vertex the path come from
2. then
    i. if `v` == `dest`, return true
    ii. else if depthFirst(w) is true, return false
    iii. else return false

The recursion induces ***backtracking***

Recursive DFS path checking:

```
hasPath(G,src,dest):
|  Input  graph G, vertices src,dest
|  Output true if there is a path from src to dest in G,
|         false otherwise
|
|  return dfsPathCheck(G,src,dest)

dfsPathCheck(G,v,dest):
|  mark v as visited
|  for all (v,w)∈edges(G) do
```

```
|  |  if w=dest then       // found edge to dest
|  |     return true
|  |  else if w has not been visited then
|  |     if dfsPathCheck(G,w,dest) then
|  |        return true    // found path via w to dest
|  |     end if
|  |  end if
|  end for
|  return false           // no path from v to dest
```

Finding a path in C:

```
#define MAX_NODES 1000

bool dfsPathCheck (Graph g, int nV, Vertex v, Vertex dest. int visited[])
{
    Vertex w;
    for (w = 0; w < nV; w++) {
        // if an adjacent node is not visited
        if ( adjacent(g, v, w) && visited[w] == -1 ) {
            visited[w] = v; // means w is visited from v
            if (w == dest)
                return true;
            else if ( dfsPathCheck(g, nV, w, dest, visited) )
                return true;
            }
    }
    return false;
}

// a helper function
void findPathDFS (Graph g, int nV, Vertex src, Vertex dest) {
    // if src is the dest
```

```
    if (src == dest)
        printf("...");

    int visited[MAX_NODES];  // array to store visiting order indexed by vertex 0..nV-1
, which will be passed as a pointer
    Vertex v;
    // initialize visited[]
    for (v = 0; v < nV; v++)
        visited[v] = -1;
    visited[src] = src; // begin from src

    if dfsPathCheck(g, nV, src, dest,  visited) {
        // print the path backwards
        v = dest;
        while (v != src) {
            printf("%d - %d\n", visited[v], v);
            v = visited[v];
        }
    }
}
```

Cost analysis:

- each vertex visited at most once ⇒ cost = $O(V)$
- visit all edges incident on visited vertices ⇒ cost = $O(E)$ (assuming using an adjacency list representation)
- Total time complexity of DFS: $O(V + E)$ (using an adjacency list representation)

Other Applications

Connected Components

Algorithm to assign vertices to connected components:

```
components(G):
|  Input graph G
|  # initialize the array
|  for all vertices v∈G do
|     componentOf[v]=-1
|  end for
|  # begin marking
|  compID=0
|  for all vertices v∈G do
|  |  if componentOf[v]=-1 then
|  |     dfsComponents(G,v,compID)
|  |     compID=compID+1
|  |  end if
|  end for

dfsComponents(G,v,id):
|  componentOf[v]=id  # marked by component's id
|  for all vertices w adjacent to v do
|     if componentOf[w]=-1 then
|        dfsComponents(G,w,id)
|     end if
|  end for
```

This can be applied to: check whether two vertex are connected very quickly (compared to using DFS to query)

Hamiltonian and Euler Paths

Hamiltonian Path

Hamiltonian path problem:

- find a simple path connecting two vertices `v`, `w` in graph `G`, such that the path <u>includes each *vertex* exactly once</u>
- If `v` = `w`, then we have a ***Hamiltonian circuit***.
- If a graph doesn't have a circle, it won't have a hamiltonian path.
- Simple to state, but difficult to solve (NP-complete)

hamiltonSearch:

1. if v==dest and d=0, return true
2.

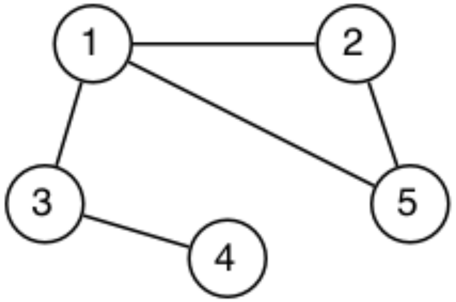
```
visited[] // array [0..nV-1] to keep track of visited vertices

hasHamiltonianPath(G,src,dest):
|  for all vertices v∈G do
|     visited[v]=false
|  end for
|  return hamiltonR(G,src,dest,#vertices(G)-1)

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) ∧ !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
```

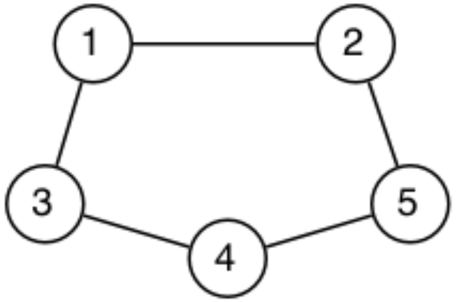
Euler Path

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***.
- If every vertex doesn't have even degree (one in and one out), Euler path won't exist.



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



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

Theorem:

1. A graph has an Euler circuit if and only if it is connected and all vertices have even degree.
2. A graph has a non-circuitous Euler path if and only if it is connected and exactly two vertices have odd degree.

```
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
|  else if degree(G,src) is odd then
|     return false
|  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
```

In many real-world applications of graphs:

- ***Directed Graph***: edges are directional (v → w ≠ w → v)
- ***Weighted Graph***: edges have a weight (cost to go from v → w)

Comparison

Euler Path:

1. include each *edge* exactly once
2. Existence: every vertex have even degree

Hamiltonian Path:

1. include each *vertex* exactly once
2. Existence: at least one circle

Directed Graphs (digraph)

Undirectional ⇒ symmetric matrix
Directional ⇒ non-symmetric matrix

Terminology for digraphs

Directed path: sequence of n ≥ 2 vertices v1 → v2 → … → vn

- where (vi,vi+1) ∊ edges(G) for all vi,vi+1 in sequence
- if v1 = vn, we have a directed cycle

Degree of vertex: deg(v) = number of edges of the form (v, _) ∊ edges(G)

- ***Indegree*** of vertex: deg-1(v) = number of edges of the form (_, v) ∊ edges(G)

Reachability: `w` is reachable from `v` if ∃ directed path `v,…,w`

Strong connectivity: every vertex is reachable from every other vertex

Directed acyclic graph (DAG): graph containing no directed cycles

Digraph Applications

- ***transitive closure***: is there a directed path from s to t?

- ***shortest path***: what is the shortest path from s to t?
- ***strong connectivity***: are all vertices mutually reachable?
- ***topological sort***: how to organise a set of tasks?
- ***PageRank***: which web pages are "important"?
- ***graph traversal***: how to build a web crawler?

Digraph Algo

Reachability / Transitive Closure

```
reachable(G,s,t):
|  return G.tc[s][t]
```

Goal: produce a matrix of reachability values

- if `tc[s][t]` is `1`, then `t` is reachable from `s`
- if `tc[s][t]` is `0`, then `t` is not reachable from `s`

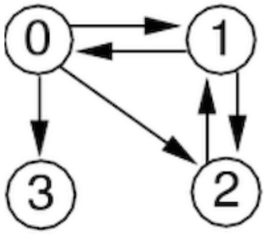
Observation:

- if `tc[s][t]` and `tc[t][i]` are true, `tc[s][i]` is true
- We can apply this rule iteratively.
- At last, we just need to know
  all the 1-length path.

Warshall's algorithm:

```
make tc[][] a copy of edges[][]
for all i∈vertices(G) do
```

```
    for all s∈vertices(G) do
        for all t∈vertices(G) do
            if tc[s][i]=1 ∧ tc[i][t]=1 then
                tc[s][t]=1
            end if
        end for
    end for
end for
```



digraph

	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	0	1	0	0
3	0	0	0	0

adj matrix

1st iteration `i=0`:

tc	[0]	[1]	[2]	[3]
[0]	0	1	1	1
[1]	1	1	1	1
[2]	0	1	0	0
[3]	0	0	0	0

2nd iteration `i=1`:

tc	[0]	[1]	[2]	[3]
[0]	1	1	1	1
[1]	1	1	1	1
[2]	1	1	1	1
[3]	0	0	0	0

3rd iteration `i=2`: unchanged

4th iteration `i=3`: unchanged

Web Crawling

PageRank

Approach: mimic a random web surfer: if we randomly follow links in the web more likely to re-discover pages with many inbound links

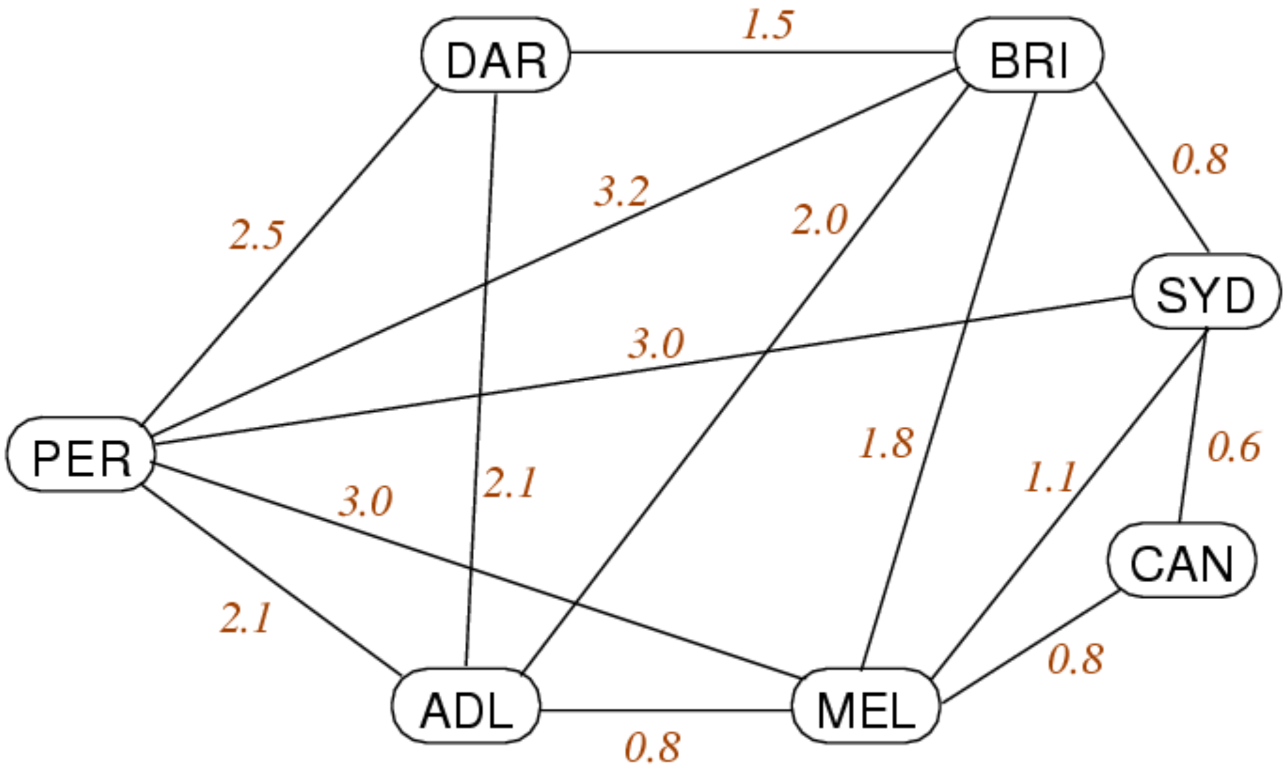
```
curr=random page, prev=null
for a long time do
|  if curr not in array ranked[] then
|     rank[curr]=0
|  end if
|  rank[curr]=rank[curr]+1
|  if random(0,100)<85 then            // with 85% chance ...
|     prev=curr
|     curr=choose hyperlink from curr  // ... crawl on
|  else
|     curr=random page                 // avoid getting stuck
|     prev=null
|  end if
end for
```

Weighted Graph

Some applications require us to consider

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

Weights can be used in both directed and undirected graphs.



Application

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

Shortest Path Algo

Path = sequence of edges in graph G $p = (v_0, v_1), (v_1, v_2), \ldots, (v_{m-1}, v_m)$

cost(path) = sum of edge weights along path

Shortest path between vertices `s` and `t`

- a simple path $p(s, t)$ where $s = first(p), t = last(p)$
- no other simple path $q(s, t)$ has $cost(q) < cost(p)$

Assumptions: weighted digraph, no negative weights.

Finding shortest path between two given nodes known as *source-target* SP problem

Variations:

- *single-source* SP: find the shortest path from one particular source vertex
- *all-pairs* SP

Single-source Shortest Path (SSSP, Dijkstra's Algorithm)

Given: weighted digraph `G`, *source vertex* `s`

Result: shortest paths from `s` to all other vertices

Data:

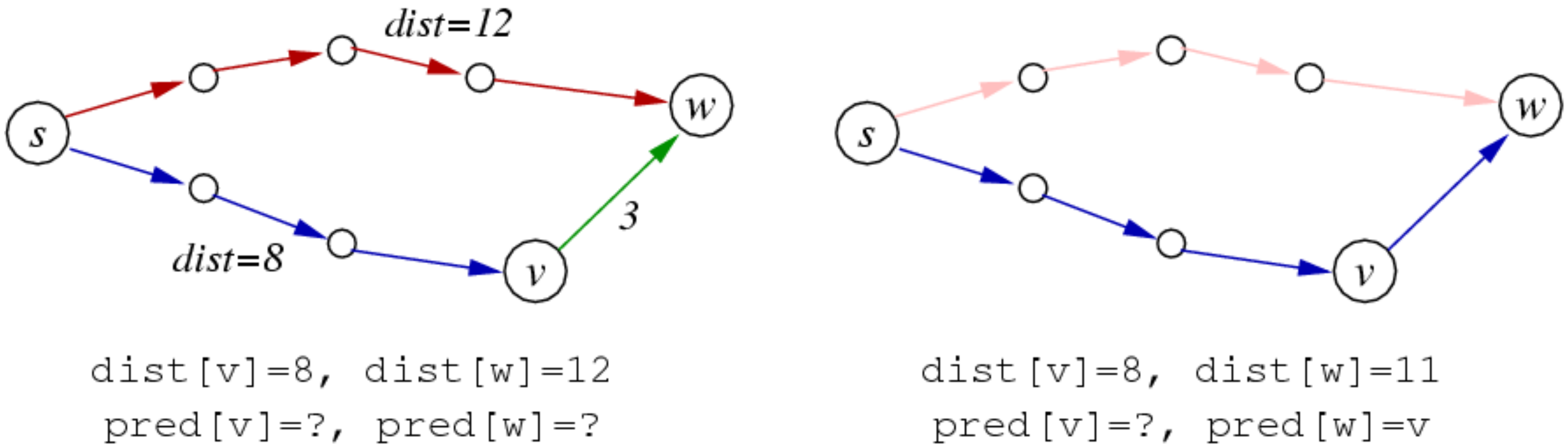
- `dist[]` V-indexed array of cost of shortest path from `s`, which stores length of shortest known path from `s` to other vertex
  - This will be initialized to infinity (a very big value).
- `pred[]` V-indexed array of predecessor in shortest path from `s`, which stores the shortest known path from `s` to other vertex
  - This will be initialized to a dummy value (like -1).

- `vSet` : set of vertices whose shortest path from s is unknown (or known)

Basic Steps:

1. initialize
2. when not all vertices are visited, do:
    - take a vertex with minimum distance from the Priority Queue
    - for its every adjancancy node, relax, and add into PQ
    - Mark the vertex as found the minimum distance

Key idea: ***Relaxation***, updates data for `w` if we find a shorter path from `s` to `w` :



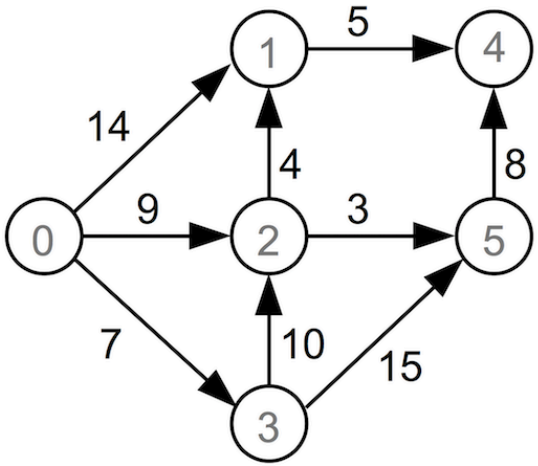
- if `dist[v]+weight < dist[w]` , then update `dist[w]:=dist[v]+weight` and `pred[w]:=v`

> The notion of "relaxation" comes from an analogy between the estimate of the shortest path and the length of a helical tension spring, which is not designed for compression. Initially, the cost of the shortest path is an overestimate, likened to a stretched out spring. As shorter paths are found, the estimated cost is lowered, and the spring is relaxed. Eventually, the shortest path, if one exists, is found and the spring has been relaxed to its resting length.
>
> The relaxation process in Dijkstra's algorithm refers to updating the cost of all vertices connected to a

> vertex v, if those costs would be improved by including the path via v.

Basic

```
dist[]  // array of cost of shortest path from s
pred[]  // array of predecessor in shortest path from s

dijkstraSSSP(G,source):
|  Input graph G, source node
|
|  initialise dist[] to all ∞, except dist[source]=0
|  initialise pred[] to all -1
|  vSet=all vertices of G
|
|  while vSet≠∅ do   // loop through all the vertex in the vset
|  |  find the s∈vSet with minimum dist[s] // very important step
|  |  // Since we just take the vertex with minimum distance
|  |  // its distance must be the smallest for itself
|  |  // otherwise, other vertex should have smaller distance
|  |  for each (s,t,w)∈edges(G) do
|  |     relax along (s,t,w)
|  |  end for
|  |  vSet=vSet\{s}
|  end while
```



	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	∞	∞	∞	∞	∞
pred	–	–	–	–	–	–

dist	0	14	9	7	∞	∞
pred	–	0	0	0	–	–

dist	0	14	9	7	∞	22
pred	–	0	0	0	–	3

dist	0	13	9	7	∞	12
pred	–	2	0	0	–	2

dist	0	13	9	7	20	12
pred	–	2	0	0	5	2

dist	0	13	9	7	18	12
pred	–	2	0	0	1	2

Finding Minimum Spanning Trees

Recap ***Spanning tree*** `ST` of *graph* `G=(V,E)`

- spanning = all vertices, tree = no cycles
- ST is a subgraph of G (G'=(V,E') where E' ⊆ E)
- ST is connected and acyclic
- If a graph has nV vertices, then its spanning tree has $nV - 1$ edges.

In a *weighted graph*, we have a special spanning tree: minimum spanning tree:
Minimum spanning tree `MST` of *graph* `G`:

- MST is a spanning tree of G
- sum of edge weights is no larger than any other ST

Two algos to find a MST in a weighted and undirected graph:

1. Kruskal's Algorithm: iterate edges in the order of weight, if it does not make a circle, then add it
2. Prim's Algorithm: iterate vertex already in the mst, and find the cheapest way to add new vertex

Simplified assumption: edges in `G` are weighted but not directed (MST for digraphs is harder)

Input: graph G with V nodes
Output: MST

Kruskal's Algorithm

Basic idea:

1. start with an empty `MST`

2. consider every edges of `G` in increasing weight order:
    - if it does not form a cycle in the `MST`, add edge
    - else, continue
3. repeat until $V-1$ edges are added

Critical operations:

- iterating over edges in weight order: ensure it only adds smallest edge
- checking for not forming a cycle: ensure it is acyclic

This is actually also an incremental algo (like relaxation). We improve our solution incrementally.

```
KruskalMST(G):
|  Input  graph G with n nodes
|  Output a minimum spanning tree of G
|
|  MST=empty graph
|  sort edges(G) by weight
|  for each e∈sortedEdgeList do
|  |  MST = MST ∪ {e}
|  |  if MST has a cyle then
|  |     MST = MST \ {e}
|  |  end if
|  |  if MST has n-1 edges then
|  |     return MST
|  |  end if
|  end for
```

```
typedef Graph MSTree;
MSTree kruskalFindMST (Graph g)
{
```

```
    MSTree mst = newGraph (); // MST initially empty
    Edge eList[g->nV]; // sorted array of edges
    edges (eList, g->nE, g);
    sortEdgeList (eList, g->nE);
    for (int i = 0; mst->nE < g->nV - 1; i++) {
        Edge e = eList[i];
        insertE (mst, e);
        if (hasCycle (mst))
            removeE (mst, e);
    }
    return mst;
}
```

Performance:

- sorting edge list is $O(E \cdot \log E)$
- at least V iterations over sorted edges
- on each iteration …
  - getting next lowest cost edge is O(1)
  - checking whether adding it forms a cycle: cost = not sure

Possibilities for cycle checking:

- DFS: too expensive?
- Union-Find data structure

Prim's Algorithm

Basic Ideas:

1. start from any vertex `v` and empty `MST`
2. choose edge not already in `MST` to add to `MST`, if it

    - be incident between a vertex `s` already connected to `v` in `MST` and a vertex `t` not already connected to `v` in `MST`
    - have minimal weight of all such edges
3. repeat until `MST` covers all vertices

Critical operations:

- checking for vertex being connected in a graph: ensure it is acyclic
- finding min weight edge in a set of edges: ensure it only adds smallest edge

```
PrimMST(G):
|  Input  graph G with n nodes
|  Output a minimum spanning tree of G
|
|  MST=empty graph
|  usedV={0}
|  unusedE=edges(g)
|  while |usedV|<n do
|  |  find e=(s,t,w)∈unusedE such that {
|  |     s∈usedV ∧ t∉usedV ∧ w is min weight of all such edges
|  |  }
|  |  MST = MST ∪ {e}
|  |  usedV = usedV ∪ {t}
|  |  unusedE = unusedE \ {e}
|  end while
|  return MST
```

```
MSTree primMST(Graph g) {
    MSTree mst = newGraph(g->nV); // a new graph for MST
    VertexSet seen = newSet(); // vertices in MST
    EgdeSet fringe = newSet(); // edges at fringe
```

```
    // add starting vertex and its adjancent edges
    SetInclude(seen, 0);
    SetInclude(fringe, edgesAt(0));

    Vertex curr;
    Edge e1, e2;

    while (SetSize(seen) < g->nV) {

        e1 = getSmallestEdge(fringe);
        SetRemove(fringe, e1);
        if (isElem(seen, e.w))  continue;

        SetInclude(seen, e.w);
    }
}
```

Sidetrack: Priority Queues

Some applications of queues require: items processed in order of "key", rather than in order of entry (FIFO — first in, first out).

Priority Queues (PQueues) provide this via:

- `join` : insert item into PQueue (replacing `enqueue`)
- `leave` : remove item with largest key (`replacing dequeue`)