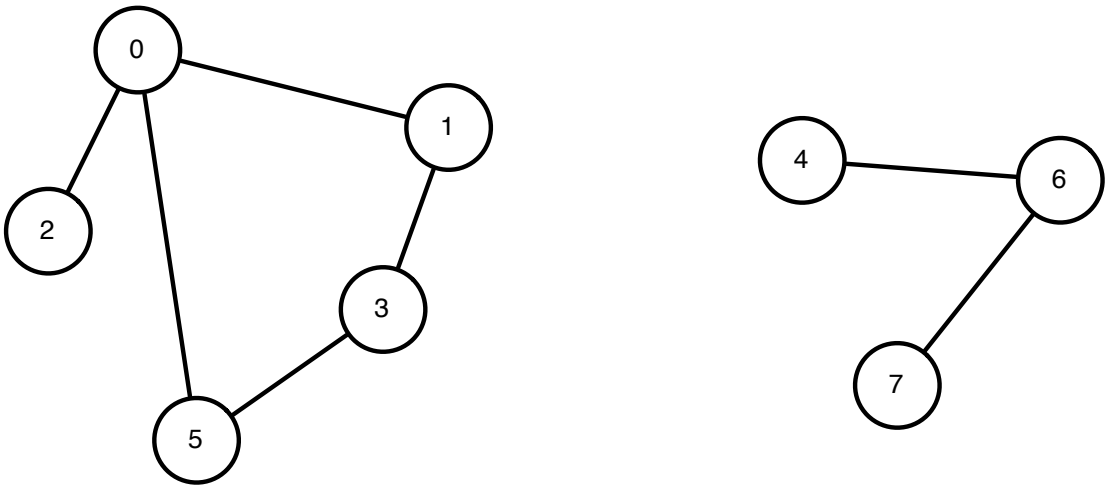
CS 234

Module 9

November 19, 2018

Graph terminology

A **simple undirected graph** G is a set $V(G)$ of **vertices** and a set $E(G)$ of **edges**, that is, unordered pairs of vertices $\{u, v\}$ such that $u \neq v$.

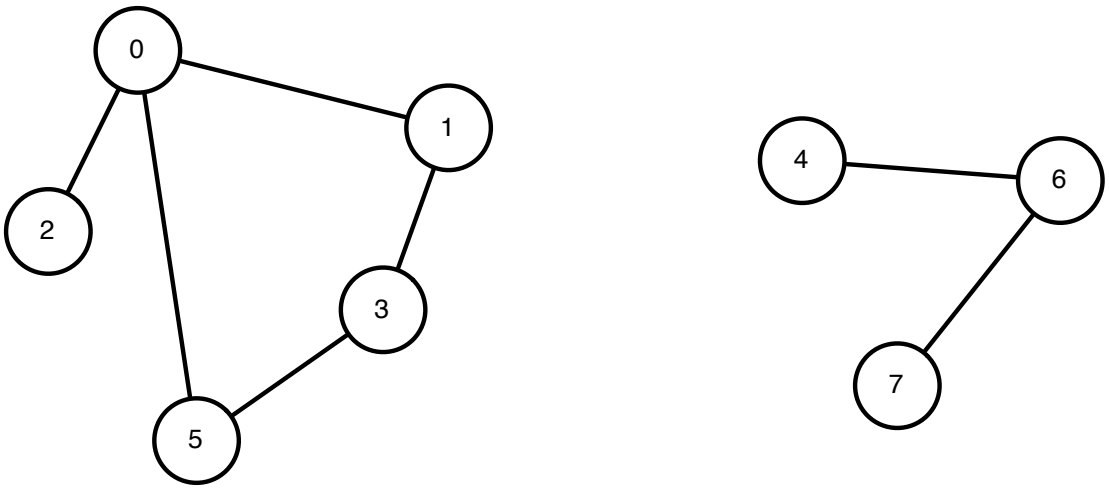


Useful terms:

- Two vertices u and v are **adjacent** if $\{u, v\} \in E(G)$.
- The edge $\{u, v\}$ is **incident on** the vertices u and v.

Degree and neighbours

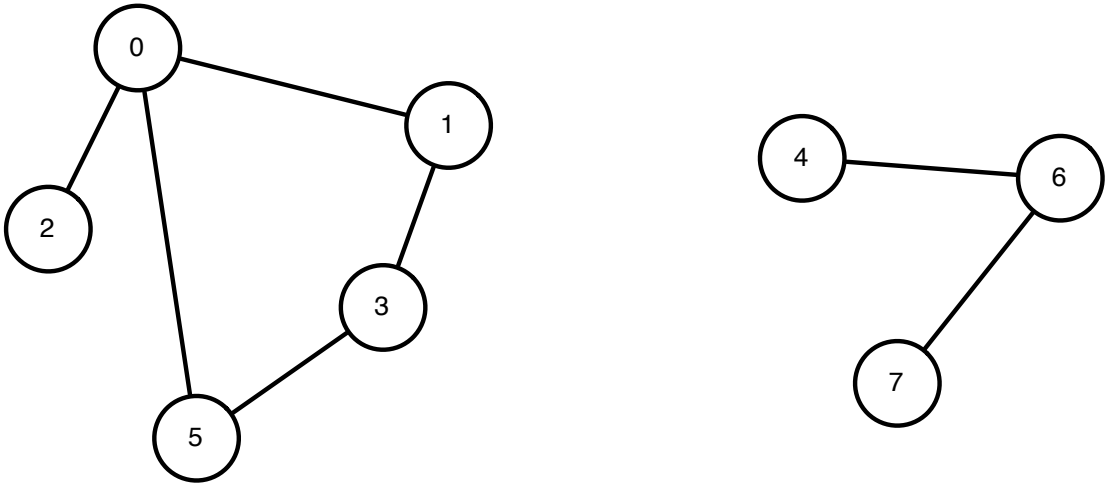
- The set of all vertices adjacent to u is the set of **neighbours** of u.
- The **degree** of a vertex is the number of incident edges.



Questions:

- What is the minimum degree of a vertex in a graph?
- What is the maximum degree of a vertex in a graph?
- How is degree related to number of neighbours?

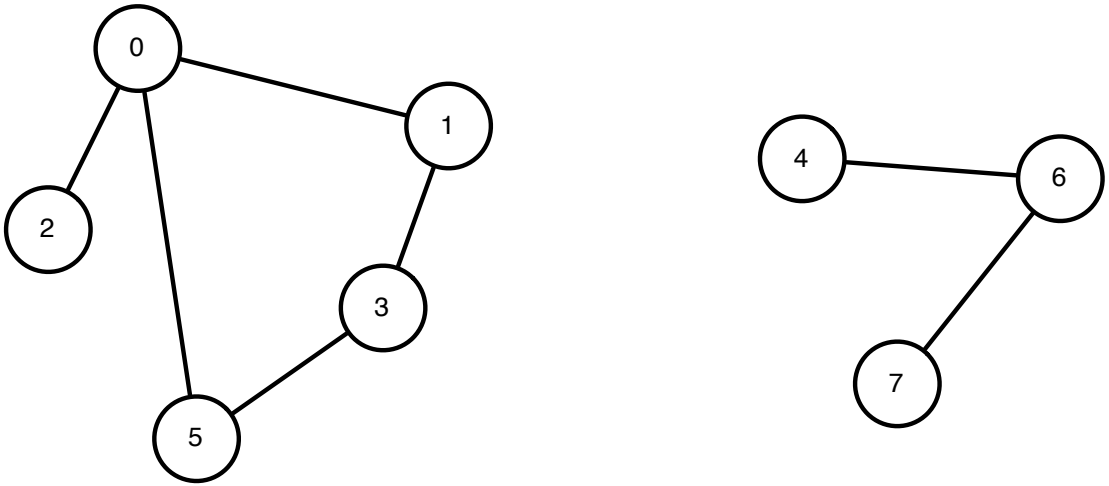
Paths and cycles



- A **path** between vertices v_0 and v_k is a sequence of vertices $\{v_0, \ldots, v_k\}$ such that $\{v_i, v_{i+1}\} \in E(G)$ for all $0 \leq i < k$.
- A **cycle** is a path such that $v_0 = v_k$ and $k > 1$.
- A graph without a cycle is **acyclic**.

Connected graphs

A graph is **connected** if there is path between any pair of vertices in $V(G)$.



Questions:

- What is the minimum number of edges in a connected graph?
- What is the maximum number of edges in a graph?
- What is another name for a connected graph with the minimum number of edges?
- Why is such a graph acyclic?

Calculating running times for graph operations

Conventions used:

- Unless other information is known, running times are expressed as functions of n (the number of vertices) and m (the number of edges).
- If extra information is known, the function can be simplified.

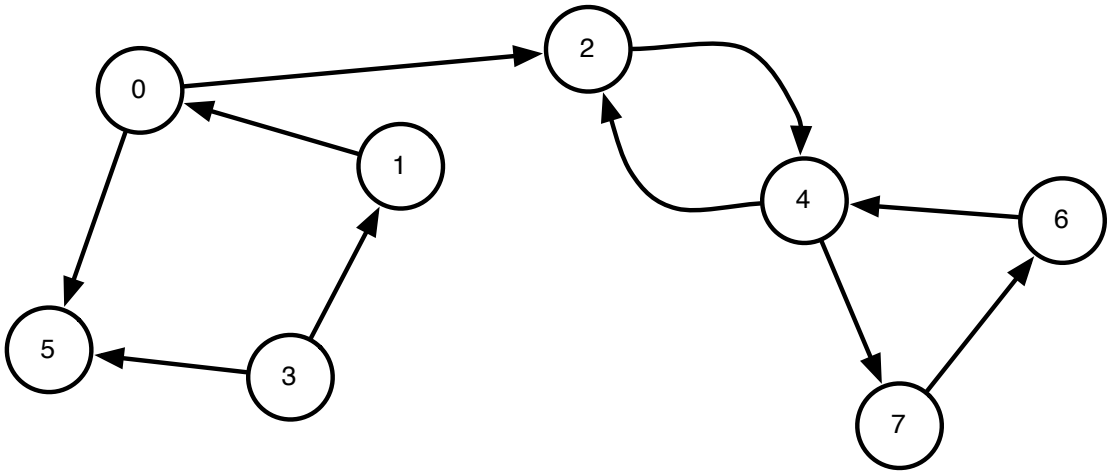
For any graph, $m \in O(n^2)$.
For a connected graph $m \in \Omega(n)$.

Examples:

Function	General	Connected
$n + m^2$	no simpler	$\Theta(m^2)$
$n^2 + m$	$\Theta(n^2)$	$\Theta(n^2)$
$n \log n + m$	no simpler	no simpler

Directed graphs

In a **directed graph**, each edge has a direction.

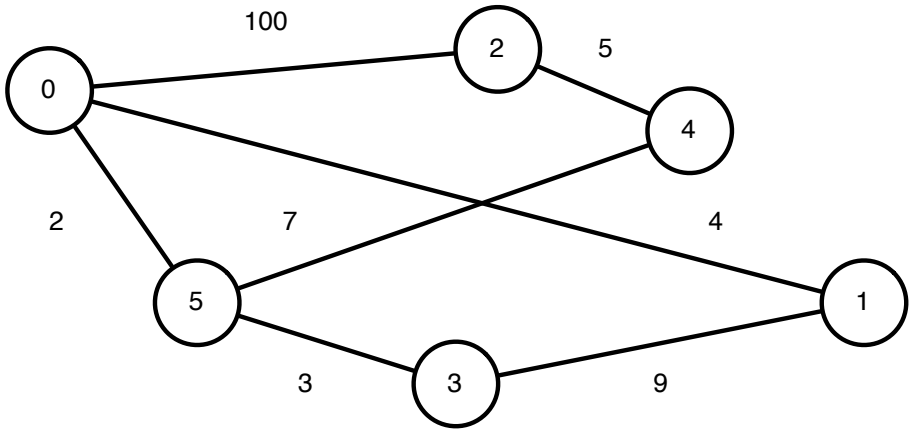


Observations:

- There are edges $(2,4)$ and $(4,2)$.
- The vertex 0 has one **in-neighbour** (1) and two **out-neighbours** (2 and 5). It has **in-degree** one and **out-degree** two.
- There is a **directed path** from vertex 3 to vertex 7.

Weighted graphs

In a **weighted graph**, there can be weights assigned to edges and/or vertices.



Observations:

- This graph has weights on edges.
- The weight of the path from 0 to 2 through 5 and 4 has a total weight of $2+7+5=14$, which is less than the weight of the edge from 0 to 2.

It is also possible for a directed graph to be weighted.

ADT Undirected Graph

Preconditions: For all G is a graph, and u and v are vertices in G.

Postconditions: Mutation by AddVertex, AddEdge, DeleteVertex, DeleteEdge.

Name	Returns
Create()	a new empty graph
IsEmpty(G)	true if empty, else false
AreAdjacent(G, u, v)	true if an edge, else false
Edges(G)	all edges (ADT Set)
Neighbours(G,v)	all neighbours of v (ADT Set)
AddVertex(G)	added vertex
AddEdge(G, u, v)	
DeleteVertex(G, v)	
DeleteEdge(G, u, v)	

Ideas for directed and weighted graphs

Directed graph

Modified operations: All edges interpreted as having a direction.

Additional operations:

- InNeighbours(G, v)
- OutNeighbours(G, v)

Weighted graph

Additional data: Store weights with vertices and edges.

Modified operations:

- AddVertex(G, weight)

Additional operations:

- VertexWeight(G, v)
- SetVertexWeight(G, v)
- EdgeWeight(G, u, v)
- SetEdgeWeight(G, u, v)

Data structures for graphs

Assume $V = \{0, 1, \ldots, n-1\}$), though this causes problems for removing vertices - the complexity will be based on the total number of vertices used, not the total in the graph.

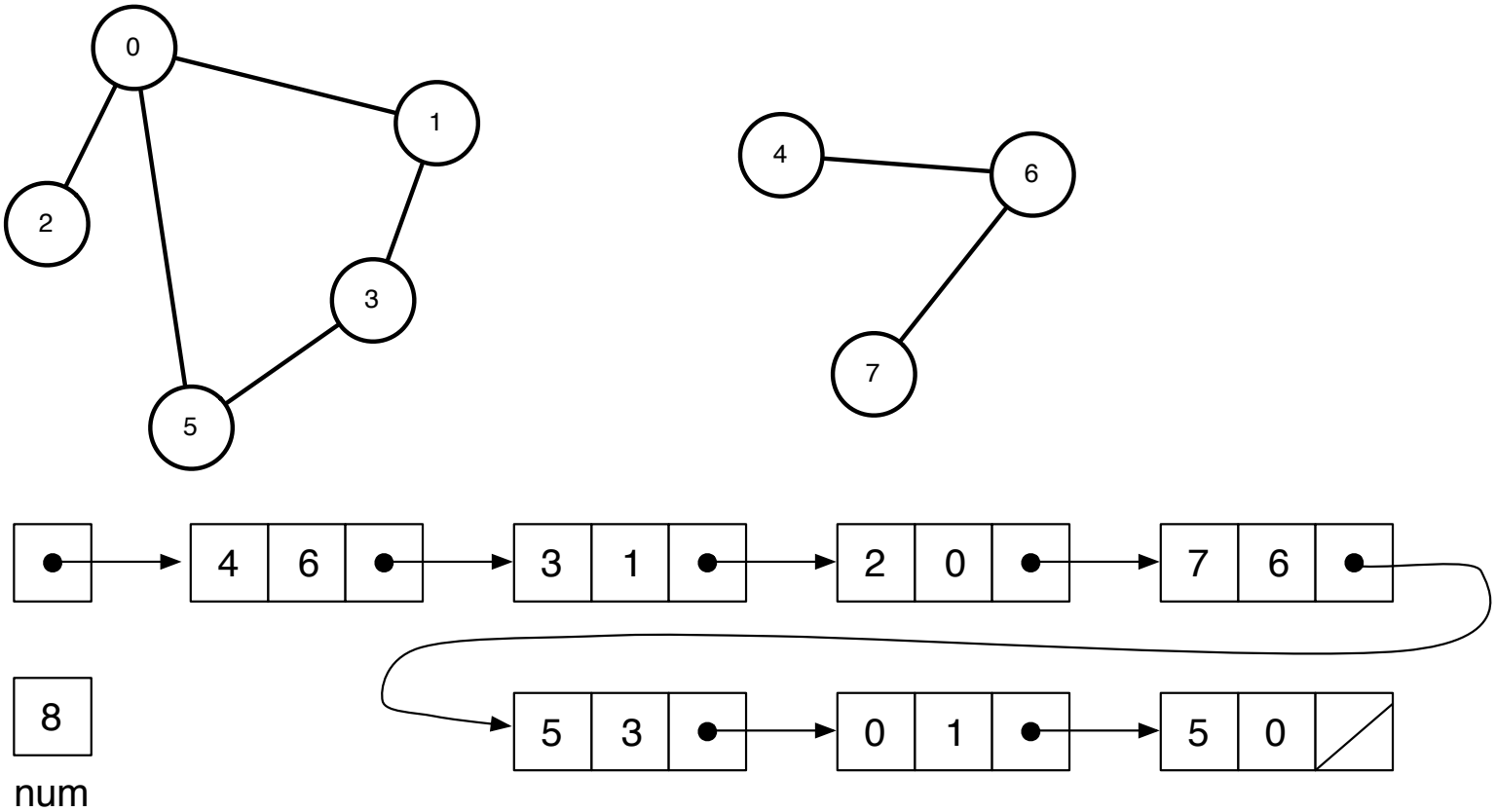
Ideas:

- Adapt linked implementation for binary trees - but one pointer per possible neighbour in each node takes too much space.
- Adapt linked implementation for ordered trees - but what is the order of "children"?
- What would an array implementation look like?

Focus on just small number of operations for now:

- AddEdge(G, u, v)
- AreAdjacent(G, u, v)
- Neighbours(G, v)
- DeleteVertex(G, v)

Data structure: **edge list**



Data structure: **edge list**

Data structure:

- Linked list where each node stores the two incident vertices.
- Variable *num* storing number of vertices in graph.

Worst-case running time of operations in terms of n and m:

- AddEdge(G, u, v): $\Theta(m)$
- AreAdjacent(G, u, v): $\Theta(m)$
- Neighbours(G, v): $\Theta(m)$
- DeleteVertex(G, v): $\Theta(m)$

Variants of edge list data structure

Variant:
Sort in terms of lower-numbered vertex.

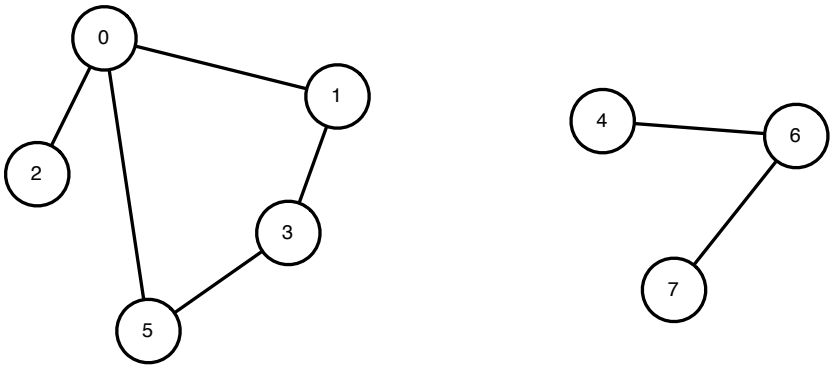
Notes:

- Directed edges can be handled by having order of the pair of values matter.
- Weighted edges can be handled as a triple of values (make clear which is the weight).
- Weighted vertices could be implemented using an array with weights.

Other ideas:
Store vertices of degree zero as edge to selves - changes the running time since number of items stored in list is now $\Theta(n+m)$.

New idea:
Can we use a bit vector to keep track of all neighbours of a vertex? One bit vector per vertex?

Data structure: **adjacency matrix**



	0	1	2	3	4	5	6	7
0	0	1	1	0	0	1	0	0
1	1	0	0	1	0	0	0	0
2	1	0	0	0	0	0	0	0
3	0	1	0	0	0	1	0	0
4	0	0	0	0	0	0	1	0
5	1	0	0	1	0	0	0	0
6	0	0	0	0	1	0	0	1
7	0	0	0	0	0	0	1	0

Data structure: **adjacency matrix**

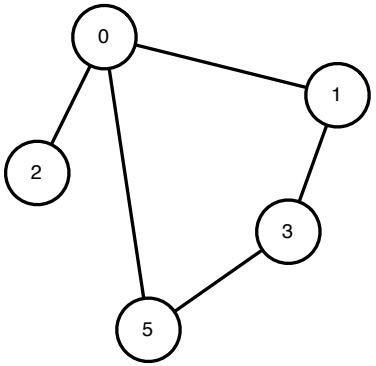
Data structure:

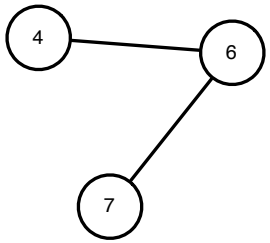
- ADT Grid with bit (exists or not) for edge (u, v) stored at (u, v)

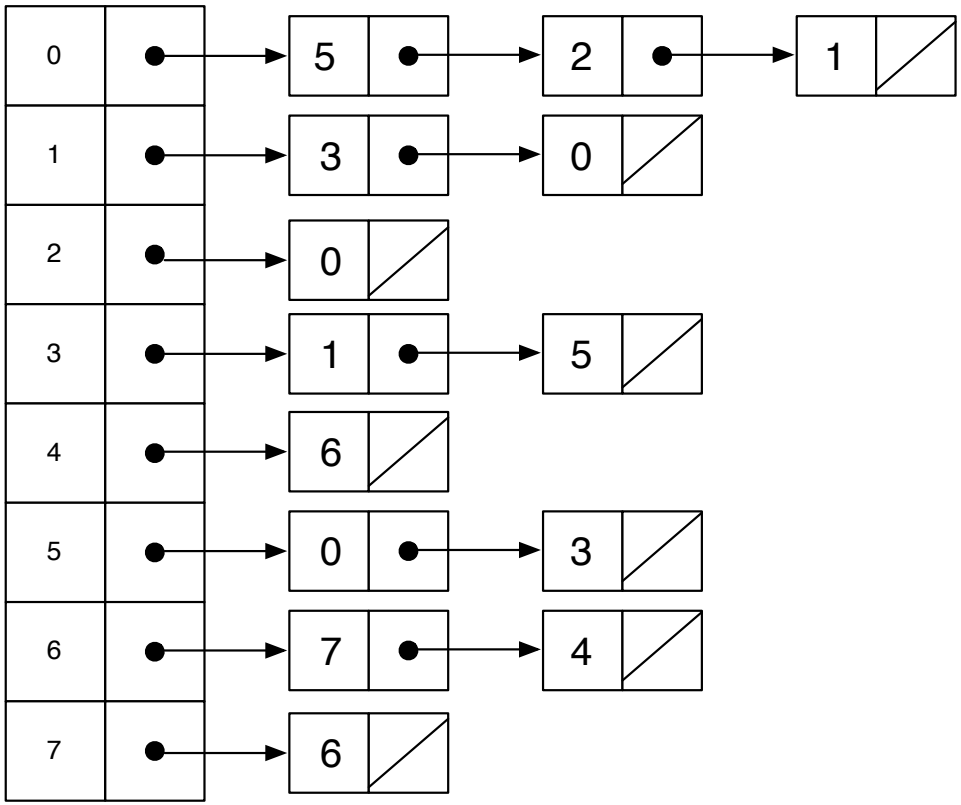
Worst-case running time of operations in terms of n and m:

- AddEdge(G, u, v): $\Theta(1)$
- AreAdjacent(G, u, v): $\Theta()$
- Neighbours(G, v): $\Theta(n)$
- DeleteVertex(G, v): $\Theta(n^2)$

Data structure: **adjacency list**







Data structure: **adjacency list**

Data structure:

- Array of size n with pointers to linked lists.
- Linked list for vertex v has a node for each neighbour.

Worst-case running time of operations in terms of n, m $deg(v)$. and $deg(u)$:

- AddEdge(G, u, v): $\Theta(deg(u) + deg(v))$
- AreAdjacent(G, u, v): $\Theta(deg(v))$ - check v list only
- Neighbours(G, v): $\Theta(deg(v))$
- DeleteVertex(G, v): $\Theta(m + n)$ - update all lists, renumber one

Data structure: adjacency list with degrees

Extra field in each array storing the degree of that vertex.

Worst-case running time of operations in terms of n, m $deg(v)$. and $deg(u)$:

- AddEdge(G, u, v): $\Theta(deg(u) + deg(v))$
- AreAdjacent(G, u, v): $\Theta(\min\{deg(u), deg(v)\})$ - check v list only
- Neighbours(G, v): $\Theta(deg(v))$
- DeleteVertex(G, v): $\Theta(m + n)$ - update all lists, renumber one

Variants of adjacency list data structure

Variants:
Sort by vertex number.

Notes:

- Directed edges can be handled by having order of the pair of values matter
- But then how to compute in and out neighbours?
- could have two links, one to list of in-neighbours, one to list of out-neighbours
- Weighted edges can be handled as a triple of values (make clear which is the weight)
- Weighted vertices with ADT List of weights or possibly as entry in ADT Set.

Special case: small degree, connected graph

Data structure:

- Variable *graph* pointing to an arbitrary node.
- Nodes storing data items and d pointers to neighbours (using some implementation of ADT Set).

Worst-case running time of operations in terms of n, m, and d:

- AddEdge(G, u, v) - $\Theta(d)$
- AreAdjacent(G, u, v) - $\Theta(d)$
- Neighbours(G, v) - $\Theta(d)$
- DeleteVertex(G, v) - $\Theta(d^2)$ - check d neighbours of each of d neighbours

Case study revisited

For airport and flights example, what if we wish to use names of airports, not numbers?

- Idea: Use data structures discussed, but have a way of mapping names to numbers.
- ADT requirements: pairs of values, given a number find a name and given a name find a number.
- ADT Dictionary for both?
- Going from number to name we can use an array.
- Going from name to number, consider BST, AVL, (2,3), maybe even hashing.

What if we wish to store multiple flights between cities since there are different times and costs of each one?

- Edge list with multiple edges, plus extra values for each
- Adjacency matrix with pointer to an ADT in each array entry.
- Adjacency list with multiple edges.

Further options for ADTs and data structures

ADTs for adjacency matrix:

- priority queue by times?
- priority queue by costs?
- pointers to multiple ADTs
- ADTs that store pointers to actual data instead of data itself so that it doesn't need to be copied in full everywhere

Other data structures for graphs:

- Modify so that vertices and edges are both more complex objects
- Include pointers
- Operations like Neighbours will return pointers to the vertices instead of the vertices themselves

Pointer to CS 338.

Case study: getting from point A to point B

Problem: Given u and v, determine if there is a path from u to v.

Ask what to do if:

- u and v adjacent (use neighbours)
- otherwise (keep track of neighbours of neighbours)

New problem: Given v, what is the set of vertices connected by a path to v?

Questions:

- How do we not get stuck in a cycle? (Mark seen)
- How to mark visited vertices? (Bit vector or colour)
- How keep track of what was seen?

Colouring vertices

Colours:

- unvisited - white
- visited but not yet completed processed - gray
- completely processed - black), where completely processed includes all neighbours having been checked

Customizing the ADT:

- VertexColour(G, v)
- SetVertexColour(G, v, c)

Provider view:

- How to modify each data structure?
- Constant cost per operation.

Breadth-first search pseudocode

Initialize all vertices to white; add processing steps as needed.

```
Q ← CreateQueue()
Enqueue(Q, u)
SetVertexColour(Q, u, gray)
while not IsEmptyQueue(Q)
    v ← Dequeue(Q)
    for w in Neighbours(G, v)
        if VertexColour(G, w) == white
            Enqueue(Q, w)
            SetVertexColour(G, w, gray)
    SetVertexColour(G, v, black)
```

Complexity analysis of BFS

Each vertex is enqueued and dequeued, total $\Theta(n)$.
When processed:

- All neighbours are checked.
- All other operations are constant time.

Total cost: sum over all vertices of cost of Neighbours

Edge list: $O(nm)$

Adjacency matrix: $O(n^2)$ (for each of n vertices, $O(n)$ to check all neighbours)

Adjacency list: $O(n+m)$ (sum of degrees is twice number of edges)

Depth-first search

```
function depth(G, start)
    S ← CreateStack()
    Push(S, start)
    while not IsEmptyStack(S)
        v ← Pop(S)
        process(v)
        for N in Neighbours(G, v)
            Push(S, N)
```

Comparison

- Each traces out a tree within the graph.
- Same complexity.
- Sometimes either fine, sometimes not.

Examples for both:

- Determine if G is connected.
- Find connected components (search from each vertex not already found in previous searches).
- Determine if G contains a cycle.

Examples for one:

- Escape from a maze.
- Shortest paths
- Finding something within a fixed distance in an infinite graph.

Search and tree traversals

Which type of search can be used for the various tree traversals?

Preorder and **postorder**: Use depth-first search, adding a vertex when it is first seen (preorder) or when it is last seen (postorder).

Inorder: Use depth-first search, always going to the left before the right, and adding a vertex when its left subtree has been visited.

Level order: Use breadth-first search.

More graph algorithms

Either as user or provider

- All pairs shortest path (uses techniques not known yet).
- Longest path (hard)
- Deliver flowers to all airline personal by sending to just some airports (vertex cover) (hard)
- Deliver flowers to airports so that all airports just a flight away (dominating set) (hard)

Pointer to CS 231