

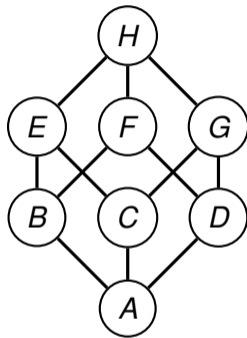
Cosc 201
Algorithms and Data Structures
Lecture 16 (22/4/2025)
Graphs: terminology and traversals

Brendan McCane
brendan.mccane@otago.ac.nz
And
Michael Albert



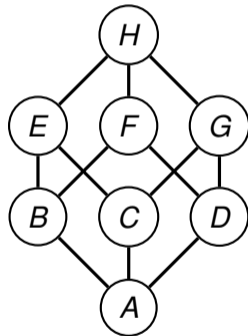
What is a graph?

- ▶ A graph represents a set of things, with some relationships between pairs of them.
- ▶ It's another "data concept" because there are an enormous range of possible interpretations and conditions, and no clear interface.
- ▶ We'll be looking (mostly) at one of the simpler interpretations. It's actually called *simple graph* in mathematics.



What is a simple graph?

- ▶ A *simple graph* (just *graph* for the rest of this lecture) consists of:
 - ▶ a set of *vertices* (A to H),
 - ▶ some *edges* between them (there are 12, AB , AC , etc.)
- ▶ The picture is just a help
- ▶ And, usually, the labels are just for convenience
- ▶ Some graphs even have **names like Q_3** .



Applications of graphs

Graphs are used to model many things, including:

- ▶ Social networks (Facebook, LinkedIn, etc.)
- ▶ Road networks (Google Maps, etc.)
- ▶ The internet (Google, etc.)
- ▶ Molecular structures (drug design, etc.)
- ▶ Scheduling (timetables, airlines, etc.)
- ▶ The power grid
- ▶ Data communications (computer networks, cell phone networks, etc.)
- ▶ Scene graphs in computer graphics
- ▶ Neural networks
- ▶ Game states (chess, etc.)
- ▶ etc

Graph definition and terminology

- ▶ A graph G is a pair (V, E) where V is a set of vertices and E is a set of edges.
- ▶ An edge is a pair of vertices, $e = (v, w)$.
- ▶ We say that v and w are *adjacent* or *neighbours* if there is an edge between them.
- ▶ The *degree* of a vertex is the number of edges incident on it.
- ▶ A *simple graph* has no self-loops (edges from a vertex to itself) and no multiple edges (more than one edge between the same pair of vertices).
- ▶ A *path* is a sequence of *distinct* vertices v_1, v_2, \dots, v_k such that (v_i, v_{i+1}) is an edge for $1 \leq i < k$.
- ▶ A *cycle* is a path where $v_1 = v_k$.
- ▶ A *connected graph* is one where there is a path between every pair of vertices.

Data structures to represent simple graphs

- ▶ There is no single ‘best’ way to represent graphs, even just simple graphs.
- ▶ The kinds of questions we typically want to be able to answer quickly, are things like:
 - ▶ Are vertices v and w *neighbours* i.e., is there an edge between them?
 - ▶ What are all the neighbours of v ?
- ▶ The simplest way to represent a graph is to have an array of vertices, and each vertex has a list of its neighbours.
- ▶ Determining if v and w are neighbours is $O(\text{degree}(v))$.
- ▶ We could also have an array of edges, and each edge is a pair of vertices (no explicit representation of vertices).
- ▶ Determining if v and w are neighbours is $O(|E|)$.

More complex representations

- ▶ More efficient would be an array with an element for each vertex, and each element of the array is a `HashSet` of its neighbours.
- ▶ If we wanted more structure/flexibility we could define a `Vertex` class and use a `HashMap` from `Vertex` to `HashSet<Vertex>` to represent the neighbours.
- ▶ And lots of other options too – so we'll be pretty agnostic about the details and treat the graph as an abstract data type.

Questions we might want to answer about graphs

- ▶ Are two vertices connected?
- ▶ What is the shortest path between two vertices?
- ▶ What is the shortest path between a vertex and all other vertices?
- ▶ What groups of vertices are connected?
- ▶ What is the minimum connected subgraph that includes all vertices?
- ▶ What is the maximum flow through the graph?

We will look at some, but not all, of these questions.

Traversals of simple graphs

- ▶ In a simple graph G , given vertices v and w we say they are *connected* if there is some sequence of edges that we could follow to get from v to w .
- ▶ Or, in a union-find instance where we put all pairs of neighbouring vertices in the same group, v and w belong to the same group! That group is called the *connected component* of v . This is the idea used in assignment 1!
- ▶ If we think of edges as representing interaction of some kind, then vertices in different components don't influence one another, so we often implicitly assume the graph is *connected* i.e., has only one component.
- ▶ The resident of v wants to visit all the vertices in their component. How might they do that?
- ▶ The *wide searcher* concentrates on visiting all the immediate neighbours, then all their (unvisited) neighbours, then all theirs, ...
- ▶ The *deep searcher* concentrates on visiting a neighbour, then one of their neighbours, then one of theirs, ...
- ▶ These strategies define the two fundamental *traversals* in graphs.

Breadth-first traversal

We want to traverse starting from v . We initialise an empty queue, q , and add a *seen* marker to each vertex (or make a set of *seen* vertices).

```
1:  $q.add(v)$ ,  $v.seen \leftarrow \mathbf{true}$ 
2: while  $q$  is not empty do
3:    $w \leftarrow q.remove()$ 
4:   visit( $w$ )
5:   for  $n$  in  $w.neighbours$  do
6:     if  $n.seen = \mathbf{false}$  then
7:        $q.add(n)$ 
8:        $n.seen \leftarrow \mathbf{true}$ 
9:     end if
10:  end for
11: end while
```

- ▶ (1) Add the start vertex to a queue
- ▶ (2) While the queue is not-empty.
- ▶ (3) Remove from the front of the queue.
- ▶ (4) Do something with the item (visit)
- ▶ (5-10) Look at all its neighbours and add any unseen ones onto the queue, marking them as seen.
- ▶ (11) Repeat.

Depth-first traversal

We want to traverse starting from v . We initialise an empty stack, s , and add a *seen* marker to each vertex (or make a set of *seen* vertices).

```
1:  $s.push(v)$ ,  $v.seen \leftarrow \mathbf{true}$ 
2: while  $s$  is not empty do
3:    $w \leftarrow s.pop()$ 
4:   visit( $w$ )
5:   for  $n$  in  $w.neighbours$  do
6:     if  $n.seen = \mathbf{false}$  then
7:        $s.push(n)$ ,
8:        $n.seen \leftarrow \mathbf{true}$ 
9:     end if
10:  end for
11: end while
```

- ▶ (1) Add the start vertex to a stack
- ▶ (2) While the stack is not-empty.
- ▶ (3) Pop off the top element of the stack.
- ▶ (4) Do something with the item (visit)
- ▶ (5-10) Look at all its neighbours and push any unseen ones onto the stack, marking them as seen.
- ▶ (11) Repeat.

Recursive DFT

Stack-based algorithms are often more easily expressed recursively.

```
function RECDFT(v)  
  v.seen ← true, visit(v)  
  for n in v.neighbours do  
    if n.seen = false then  
      RECDFT(n)  
    end if  
  end for  
end function
```

The cost of traversal

- ▶ In graph algorithms, we usually measure complexity both in terms of the number of vertices, V , and the number of edges E .
- ▶ The two traversal algorithms are so similar that one analysis will do for both.
- ▶ We assume that a single *push*, *pop*, *add* or *remove* is $O(1)$.
- ▶ What happens to each vertex?
 - ▶ It occurs in some neighbour list for the first time at which point it is added to the queue or stack, and marked as seen (6-7).
 - ▶ Then at some future point it is removed from the stack or queue (3).
 - ▶ That's $O(1)$ work per vertex, so $O(V)$.
- ▶ Where do the edges come in?
 - ▶ The edges arise implicitly in the neighbour lists (4).
 - ▶ Each one appears twice, once per endpoint.
 - ▶ When one appears, it causes a constant amount of work to happen (5).
 - ▶ That's $O(1)$ work per edge, so $O(E)$.
- ▶ The total complexity is $O(V + E)$.