CS2521: Graph Traversal

N. Oren
n.oren@abdn.ac.uk

University of Aberdeen

Where are we?

- We've introduced graphs - structures made up of nodes and edges.
- Considered how graphs can be represented.
- Examined different types of graphs (weighted, directed, undirected, trees).
- Defined a variety of concepts over graphs.
- We are now able to "speak in the language of graphs", allowing us to examine algorithms which manipulate and use graphs for various purposes.

Traversal

- One of the most common requirements of graph algorithms involves traversal — visiting every edge and vertex in the graph in a systematic manner.
- Another common requirement is search — visit vertices (by moving across edges) until some data is found (or other criteria achieved). Closely linked to traversal.
- A key mandatory property of algorithms to perform traversal and search is the need to avoid visiting the same node or edge twice (as we could easily then get stuck in a loop).

Knowing Where We've Been

- Basic approach: mark nodes according to whether they've been visited/processed or not.
- Three possible states
  1. undiscovered Node has not been encountered by the algorithm.
  2. discovered Node has been encountered, but some of its incident edges have not been used/explored.
  3. processed Node has been encountered and all edges have been used.
- Vertices move from undiscovered, to discovered, to processed as the algorithm progresses.

Traversal — Basic Idea

- Assume we have a data structure storing discovered but unprocessed nodes (i.e., nodes we have to do some work over).
- Initially, this contains only one node (the start node), marked as discovered.
- We evaluate each edge in the first node in the data structure; if we find a new node, mark it as discovered and add it to the data structure.
- We ignore edges that go to processed or discovered nodes.
- Once all edges are evaluated, mark node as processed, remove it from the data structure, and go to the next node in the structure.

Traversal — Properties

- If our graph is undirected, each edge will be considered how many times?

Traversal — Properties

- If our graph is undirected, each edge will be considered how many times? 2
- If our graph is directed, each edge will be considered how many times?

Traversal — Properties

- If our graph is undirected, each edge will be considered how many times? 2
- If our graph is directed, each edge will be considered how many times? 1
- Eventually, every edge and vertex in the component connected to the start node will be visited. Why?

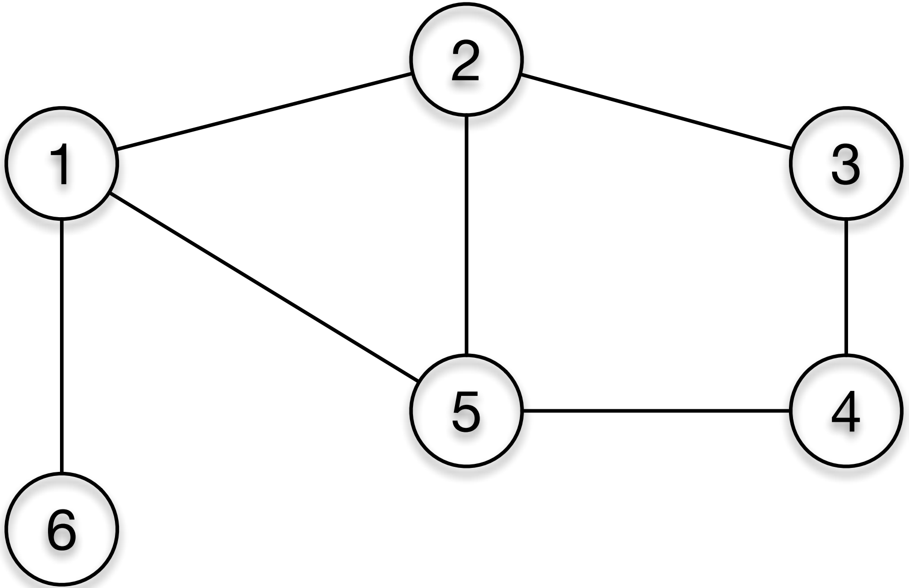
Traversal — Properties

- If our graph is undirected, each edge will be considered how many times? 2
- If our graph is directed, each edge will be considered how many times? 1
- Eventually, every edge and vertex in the component connected to the start node will be visited. Why?
- Assume an unvisited vertex u, whose neighbour v is discovered. Eventually, will be explored and processed, meaning that the (v, u) edge will be traversed.
- different rules for selecting nodes to process will yield different traversal algorithms

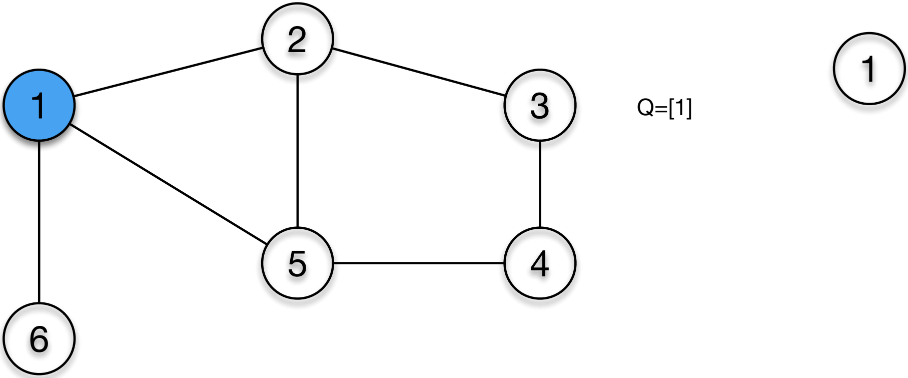
Traversal via Breadth First Search

```
function BFS(G, s)
    Q, an empty queue.
    for all v ∈ G\{s} do
        state[v] = undiscovered
        parent[v] = nil
    state[s] = discovered
    Q.enqueue(s)
    while Q is not empty do
        v = Q.dequeue
        process v
        for all w ∈ Adj[v] do
            process (v, w)
            if state[w] = undiscovered then
                state[w] = discovered
                p[w] = v
                Q.enqueue(w)
        state[v] = processed
```

BFS in Operation

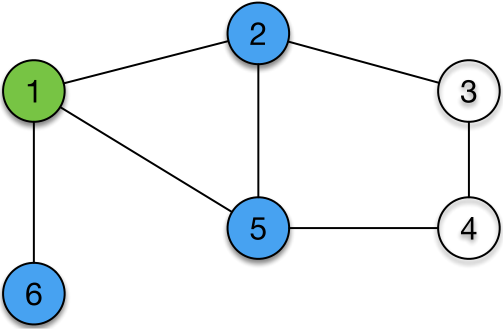


BFS in Operation

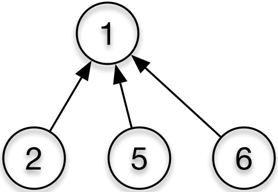


Q=[1]

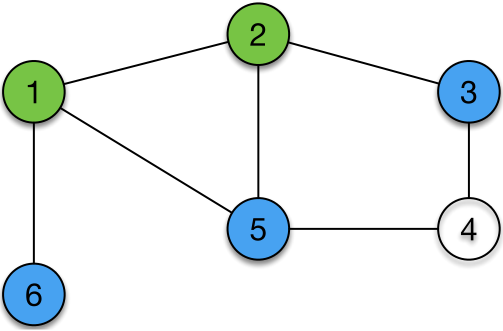
BFS in Operation



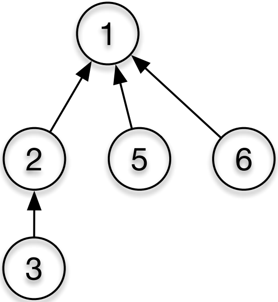
Q=[2,5,6]



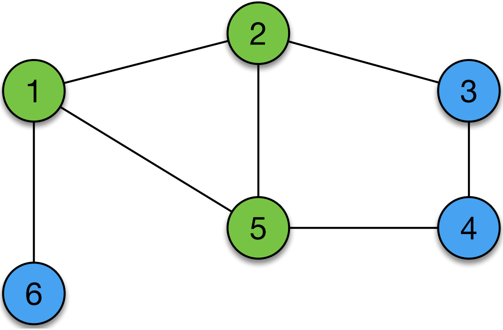
BFS in Operation



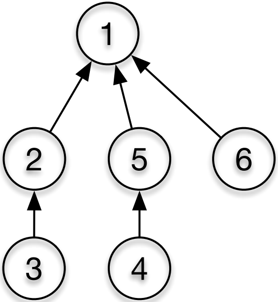
Q=[5,6,3]



BFS in Operation



Q=[6,3,4]



BFS Efficiency

- For a Graph (V, E), how efficient is BFS?
- Initialization visits each node once, so $\Theta(V)$
- Since only undiscovered nodes are enqueued, each node can only be enqueued (and dequeued) once, $\Theta(V)$
- Each edge is examined only once, $\Theta(E)$
- $\Theta(2V + E) = \Theta(V + E)$
- Recall that adjacency list memory usage is $\Theta(V + E)$
- BFS runs in time linear to the size of the adjacency list representation of the graph

What can we do with BFS?

- We store the node that discovered node[i] as an element $p[i]$ within the parent array p.
- Every node has a parent except for the start node.
- The parent relation describes a tree representing the node discovery process.
- Since vertices are discovered in order of increasing distance from the root, <u>the unique tree path from root to a node uses the smallest number of edges (or intermediate vertices) possible in the root-to-node path in the graph</u>.

Finding the Shortest Path

- We can't go from root to node, as this path can't be found from the parent list.
- Instead, we go from the target node backwards, and reverse the result.
- We can do this using a stack (how)?
- Or we can use recursion:

```
function findPath(start,end)
    if !start==end then
        print findPath(start,p[end])
    print end
```

Connected Components

- Many problems reduce to finding or counting connected components. E.g., if you consider a Rubik's cube configuration as a node, determining if it can be solved involves checking whether the graph of legal configurations is connected.
- Connected components can be found using breadth-first search, as vertex order is irrelevant.
- We can modify BFS to find connected components of a graph.
  - Pick a vertex, do BFS from there and store as one component, removing from graph.
  - Repeat for next vertex.

function CC(G)

$L = \emptyset$

$U = vert(G)$

while U!=null **do**

$u = U[0]$

do BFS starting from u and add the set of all found vertices to L

remove all found vertices from U

return L

Edge and Vertex Processing

- We can process a vertex when it is dequeued — early processing
- Or just before the next element is dequeued — late processing
- Edge processing occurs when an edge is discovered

Two-Colouring Graphs

- The vertex-coloring problem seeks to assign a label (color) to each vertex of agraph so that no edge links vertices of the same color.
- Typically, we seek to use the minimum colors possible.
- Applications in scheduling and compliers.
- A graph is bipartite if it can be legally colored using only two colors.
- Determining whether a graph is bipartite is important.
- Solution: adjust BFS so that when a vertex is discovered, it is colored opposite to its parent.
- We can then check whether any non-discovery edge links two two same coloured vertices.

Two-Colouring Graphs

- Set the start node to a color (e.g., white).
- When processing an edge, check colours of both nodes, if its identical, say the graph is not bipartite.
- Otherwise, set the other side of the edge to the complement of color of the original side (e.g., black for a white node).

Depth-First Search

- BFS uses a queue to store neighbouring nodes.
- What happens if we use a stack instead?

DFS

- Queue we explore the "oldest" unexplored vertices first. Our exploration radiates out from the starting vertex.
- Stack since we use a LIFO, we explore newest neighbours first (if available), backing up only when surrounded by previously discovered vertices. This is a depth first search.
- DFS can be defined recursively with no need for a stack.

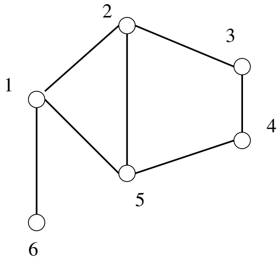
DFS

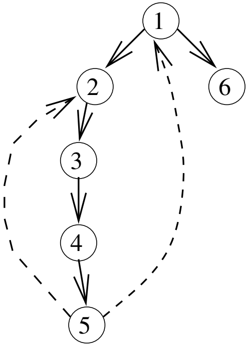
```
function DFS(G, u)
    state[u]="discovered"
    process u
    entry[u]=t
    t++
    for all v ∈ Adj[u] do
        process (u, v)
        if state[v]="undiscovered" then
            parent[v]=u
            DFS(G, v)
    state[u]="processed"
    exit[u]=t
    t++
```

DFS

- p stores the parent node
- t tracks the time a vertex is entered or exited (useful for some algorithms, but not strictly necessary)
  - If x is an ancestor of y, x must be entered before y, and y exited before x. So
    $$entry[x] < entry[y] < exit[y] < exit[x]$$
  - Since t gets incremented on entry and exit, the total number of descendants a node x has is $exit[x] - entry[x]/2$.

- DFS partitions edges (of an undirected graph) into either <u>tree edges</u> or <u>back edges</u>.
- Tree edges discover new vertices, and can be seen through the *parent* relation.
- Back edges are those whose other endpoint is an ancestor of the vertex being expanded, they point back into the tree.





DFS

- Edges can't go to a sibling, only an ancestor (or descendant), as all nodes reachable from a vertex v are expanded before traversal from v is completed.
- DFS exhaustively searches all possibilities by advancing if possible, and backing up when no unexplored possibility for further advancement exists. This concept is also known as backtracking.

Processing in DFS

- As for BFS, vertices and edges can be processed at different times.
- early processing — before traversing to child vertex
- late processing — after all (child) vertices have been processed
- edge processing — when the edge is discovered, or when edge is traversed

DFS Applications - cycle detection

- We can use DFS to find a cycle — if an edge is found to a node, and that edge doesn't lead to a parent, then a cycle has been found.

```
function processEdge(n1,n2)
    if p[n1]!=n2 ∧ n2 is undiscovered then
        print "cycle found"
```

- For this to work correctly, DFS must process undirected edges only once.

Articulation Vertices

- Suppose you are building a road network, and must take future roadworks into account — you wish to determine if roadworks anywhere will mean that travel between points can't take place.
- If roads are nodes, with connections between roads represented as edges, you want to identify vertices whose removal causes previously connected components to disconnect.
- Such vertices are articulation vertices, or cut nodes
- Any graph containing these is, in a sense, fragile.
- More generally, the connectivity of a graph is the smallest number of components whose deletion will disconnect it.

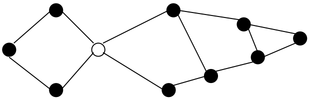


Figure 5.11: An articulation vertex is the weakest point in the graph

Articulation Vertices

- Brute force testing for a cut node is easy.

Articulation Vertices

- Brute force testing for a cut node is easy.
- Delete a vertex, do a BFS or DFS traversal and determine whether it is still connected. Complexity?

Articulation Vertices

- Brute force testing for a cut node is easy.
- Delete a vertex, do a BFS or DFS traversal and determine whether it is still connected. Complexity?
- Each traversal is $O(n+m)$, and n traversals are needed, so total cost is $O(n(n+m))$

Articulation Vertices

- Consider a search tree generated by a DFS graph traversal, which contains all vertices in a graph.
- Removing a leaf from this tree has no impact, as the leaf connects no one but itself to the tree.
- Removing the root node will split the tree if this root has more then one child.
- Removing an internal node will split the tree, unless there are back edges circumventing this split.
- So we need to determine whether such a back edge exists.
- If the subtree rooted at u has a back edge going to an ancestor of u, then u is not an articulation vertex.

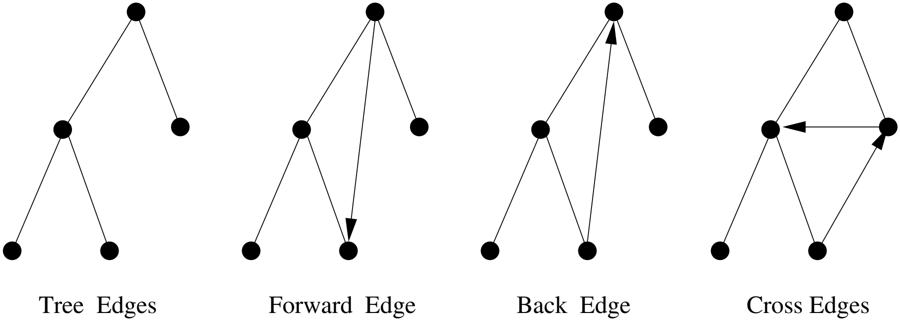
Algorithm

- Do DFS to build tree, record back edges.
- Starting at leaves of tree, identify oldest ancestor.
- Propagate value up to parent, and repeat.
- A node is an articulation node if
    - It is a root node with more than one child; or
    - It is a node with at least one child, whose oldest ancestor is its own parent.
- DFS: $O(n+m)$. Propogation: $O(n)$

Bridges

- We can instead consider cutting edges.
- An edge whose removal disconnects components is called a bridge
- A connected graph with no such edges is edge biconnected.
- An edge can be checked by removing it and checking for connectivity of graphs — linear time.
- Modifying the algorithm described previously gives a check for all bridge edges in $O(n + m)$ time — an edge is a bridge edge if it is a tree edge and no back edge connects from below to above the edge.

Edge Types



DFS on Directed Graphs

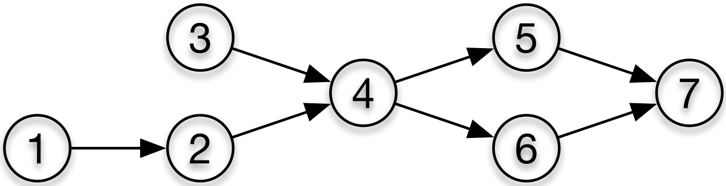
- DFS on undirected graphs will contain only tree and back edges.
  - Assume we encounter a forward edge, (x, y). This edge would have been discovered when in node y, and (y, x) should therefore be a back edge.
  - Assume we encounter a cross edge, (x, y). This again would have been encountered in y, making it a tree edge.
- For a directed graph however, all 4 edge types can be encountered.
- Each edge type can be categorised based on vertex state, parent and discovery time.

DFS on Directed Graphs

- If the parent of the new node is the other node, then it's a tree edge.
- If the edge is from a discovered, but unprocessed node, then it's a back edge.
- If the new node has been processed, and entered after the other node, then it's a forward edge.
- (otherwise), if the new node has been processed, and entered before the other node, then it's a cross node.

Topological Sort

- Given a directed acyclic graph (DAG), a topological sort returns an ordering of nodes such that — for all a, b — if a is an ancestor of b, a appears before b in the ordering.



1,2,3,4,5,6,7 3,1,2,4,6,5,7
1,3,2,4,5,6,7 1,3,3,4,6,5,7

- Each DAG has at least one topological sort.

Topological Sort

- A topological sort gives us an ordering to process vertices before successors.
- E.g., in a scheduler, edge x, y might mean that x has to be done before y. A topological sort then gives a legal schedule.
- Topological sorts lie at the heart of many critical algorithms on DAGs.
- Note that a graph is a DAG if and only if (iff) a depth first search reveals no back edges.
- Furthermore, labelling the vertices in the reverse order in which they are processed results in a topological sort.

Topological Sort

- Labelling the vertices in the reverse order in which they are processed results in a topological sort.
- Assume we are exploring vertex x, and encounter edge (x, y)
  - If y is undiscovered, we start a DFS of y before continuing with x. y will be marked completed before x, and x appears before y in the topological sort.
  - If y is discovered, but not completed, then (x, y) is a back edge, which can't exist in a DAG.
  - If y is processed, then it will have been labelled so before x, so x must appear before y in the topological sort.

Strongly Connected Components

- It is easy to test whether a graph is strongly connected using graph traversal.
- Pick a vertex v, and perform traversal. Check that all vertices can be reached.
- Reverse the direction of the edges, and do another traversal. If all nodes are again reachable, then the graph is strongly connected.
- If a graph is not strongly connected, we can seek strongly connected components.

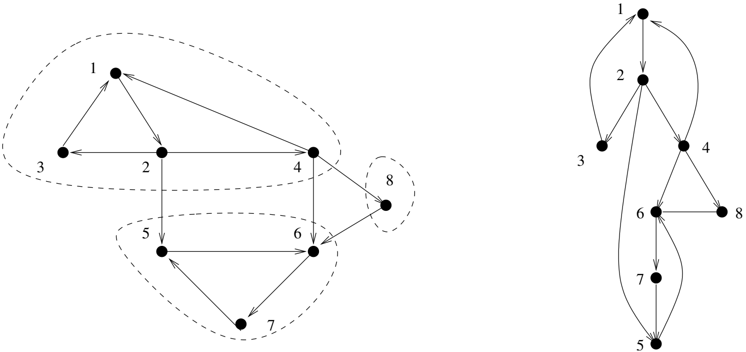


Figure 5.16: The strongly-connected components of a graph, with the associated DFS tree

- Note that any back edges in a graph, plus the path down the DFS tree gives a cycle. All nodes in this cycle are strongly connected.
- We then represent all components of this SCC as a single vertex, and repeat; when no directed cycles remain, each vertex represents a SCC.

Where are we?

- We've considered breadth and depth first traversal of a graph.
- And examined some fundamental algorithms which build on these traversals. These algorithms depend on when, and how, each node is processed, as well as the traversal approach used.
- Finding shortest path (BFS)
- Bicoloring (BFS)
- Connected component detection (BFS)
- Articulation vertices and bridges
- Topological sorting
- Strongly connected components