Lecturer: Andrew Parkes
Web page for the course, follow links from
http://www.cs.nott.ac.uk/~ajp/

G52ADS 2014-15
Graph Traversals

Breadth-First and Depth-First Search

"Dive dive dive"

Graph traversals

- We look at two ways of visiting all vertices in a graph:
  - breadth-first search (BFS)
  - depth-first search (DFS)
- Traversal of the graph is used to perform tasks such as searching for a certain node
- It can also be slightly modified to search for a path between two nodes, check if the graph is connected, check if it contains loops, and so on.
- Example: webcrawlers

How to think about graph algorithms. 1

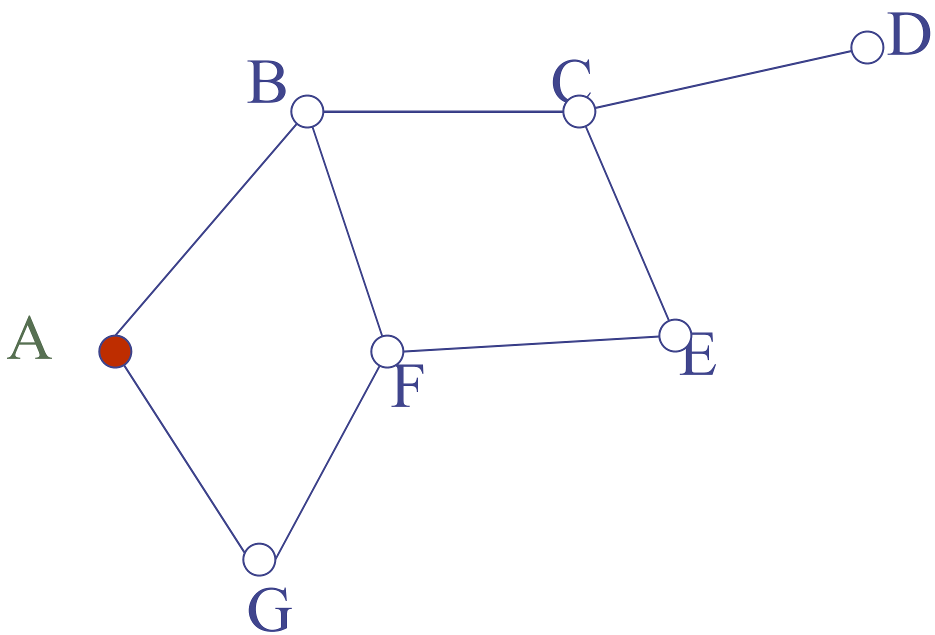
- Brain is
  - massive parallel processor
  - subconscious
- often can look at whole graph and “see” things immediately.
- But computer:
  - does not see whole graph – just some set of ‘working nodes’
  - works sequentially

How to think about graph algorithms. 2

- Hence, reasoning based on “Graph on a piece of paper” can be misleading
- Better models (“ways of thinking”) might be
  - graph as “websites & links”, and you only ‘see’ what you explicitly access
  - graph as a maze – no ‘birds-eye’ view, but only a local view
  - graph theory as potholing
    - a set of caves and tunnels but no overall map

Graph Traversal starting from A

- *Exercise: What might we do!?*



Graph Traversals

- Generally have three sets of nodes

1. Nodes that have not yet been discovered
2. “Working Set” – nodes we are currently processing in some way
3. Nodes that we have finished with

The names for these sets might vary, but they are often (implicitly) present

Graph Traversals: General View

- “Processing a node” will generally mean looking at its neighbours and (generally) adding them to the working set
- The working set is stored in some data structure
  - Need a policy to pick which node of the working set is next selected for processing: FIFO? LIFO? something else?
  - Once selected, in some algorithms, the node might be moved to a data structure storing “finished nodes”
  - Usually continue until the working set is empty

Breadth first search

BFS $\leftrightarrow$ Queue

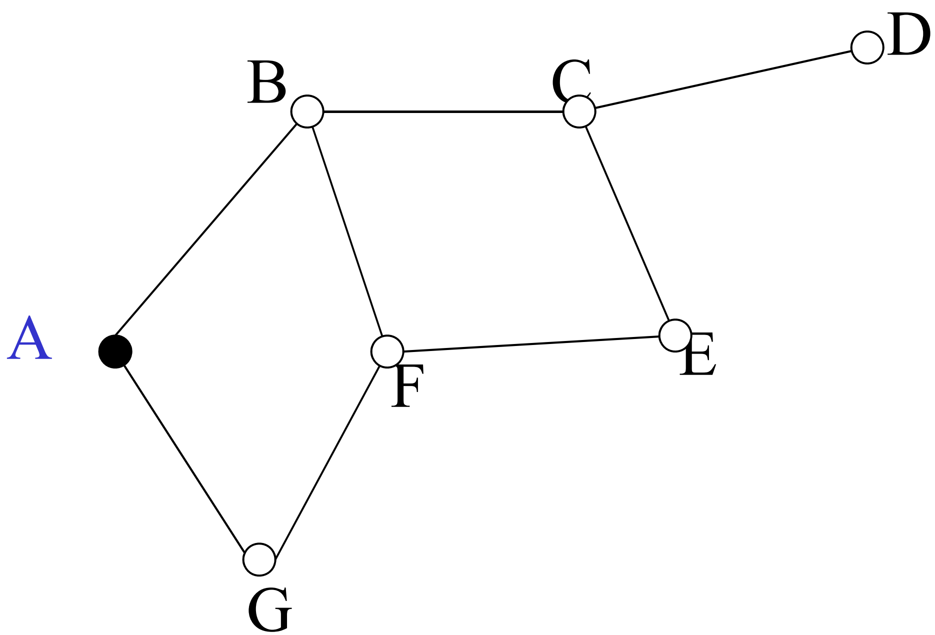
BFS starting from vertex v:

```
create a queue Q
mark v as visited and put v into Q
while Q is non-empty
   remove the head u of Q
   mark and enqueue all (unvisited)
   neighbours of u
```

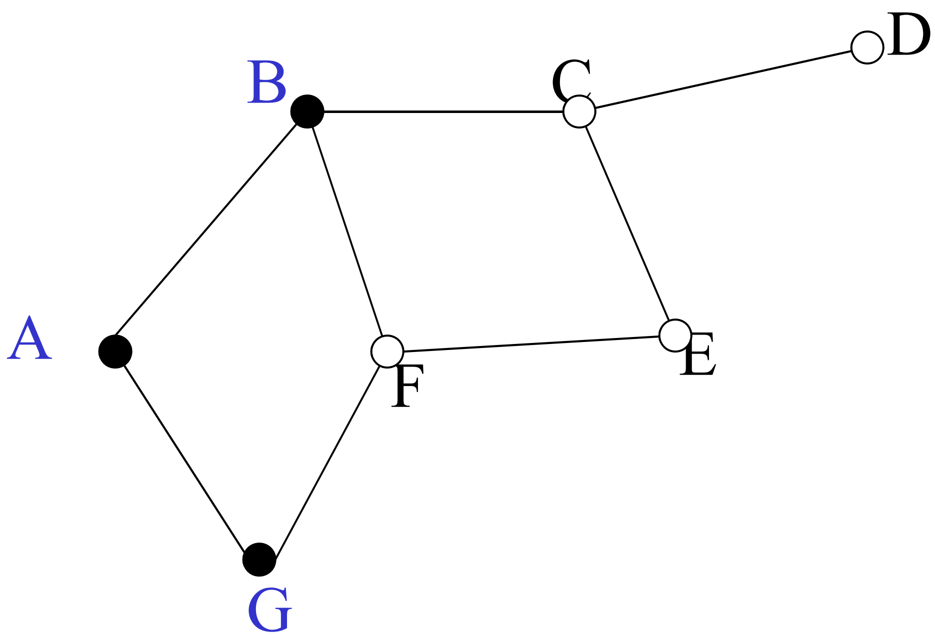
add **all** neighbours at same time

BFS starting from A:

Q={A}



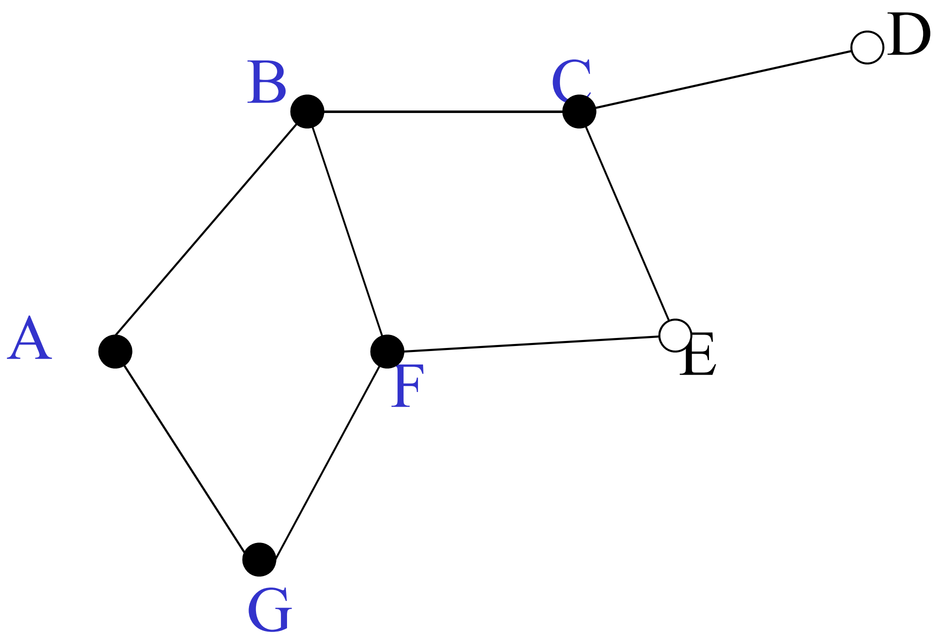
BFS starting from A:



Q={A}

Q={B,G}

BFS starting from A:

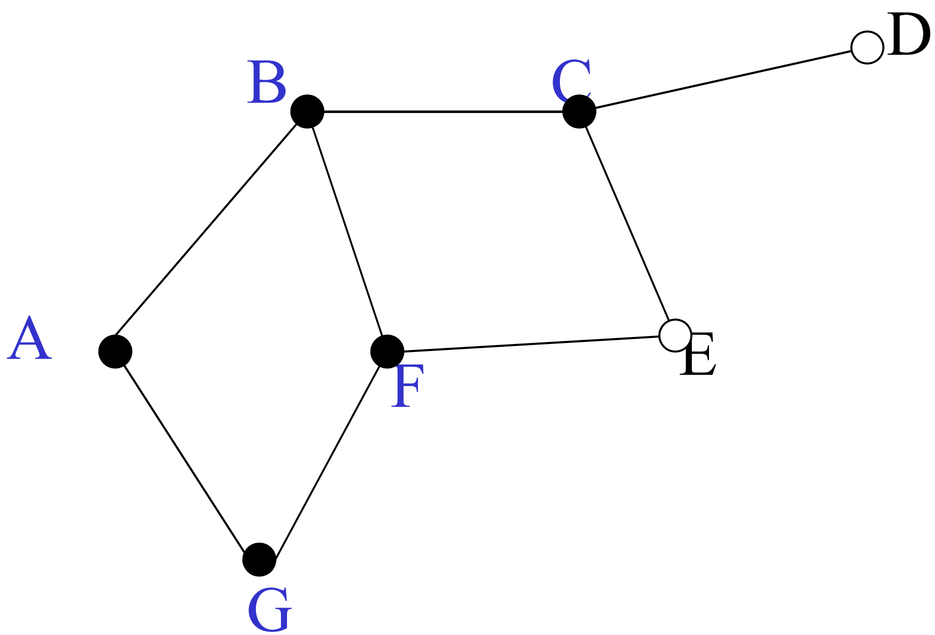


Q={A}

Q={B,G}

Q={G,C,F}

BFS starting from A:



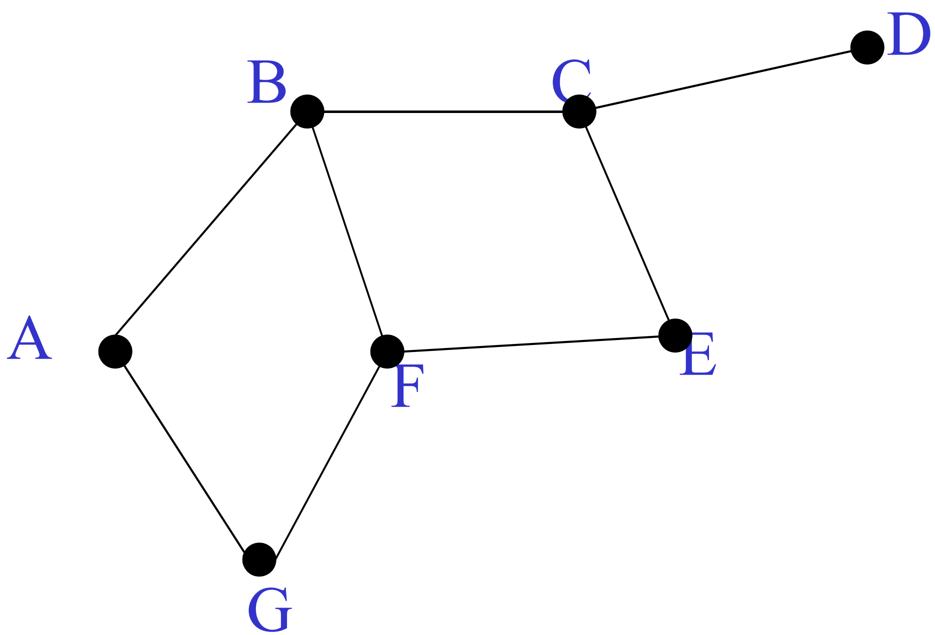
Q={A}

Q={B,G}

Q={G,C,F}

Q={C,F}

BFS starting from A:



Q={A}

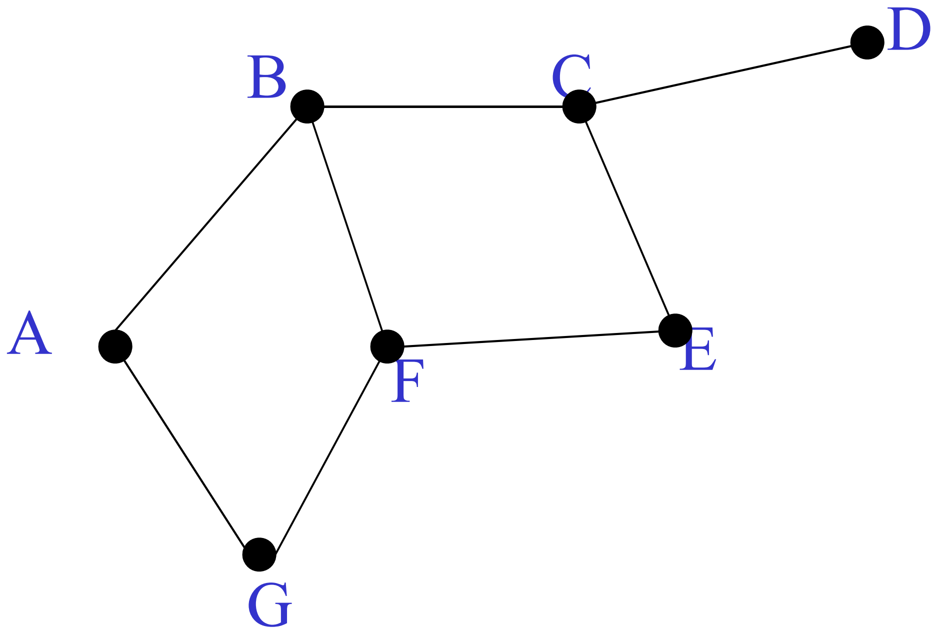
Q={B,G}

Q={G,C,F}

Q={C,F}

Q={F,D,E}

BFS starting from A:



Q={A}

Q={B,G}

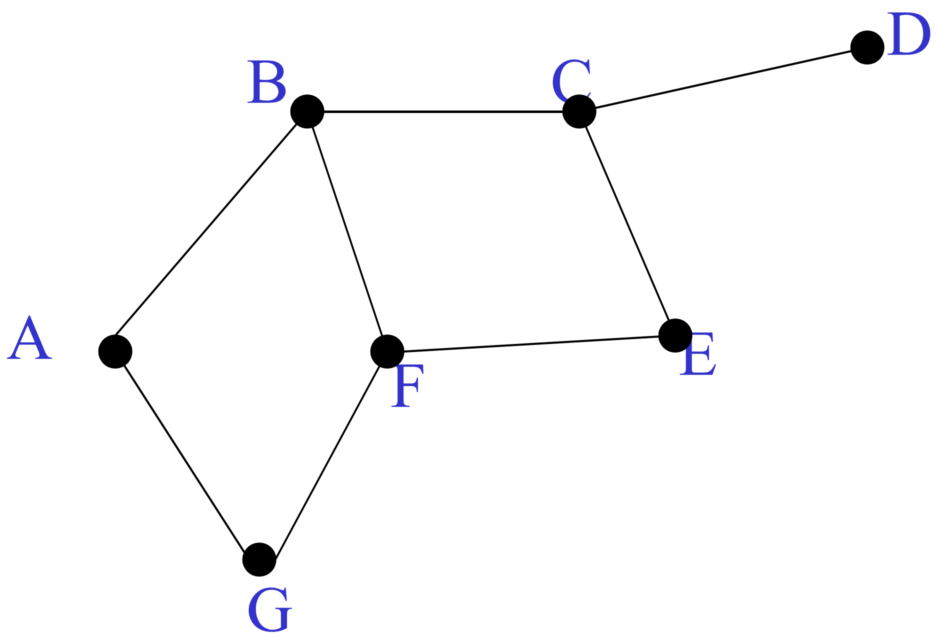
Q={G,C,F}

Q={C,F}

Q={F,D,E}

Q={D,E}

BFS starting from A:



Q={A}

Q={B,G}

Q={G,C,F}

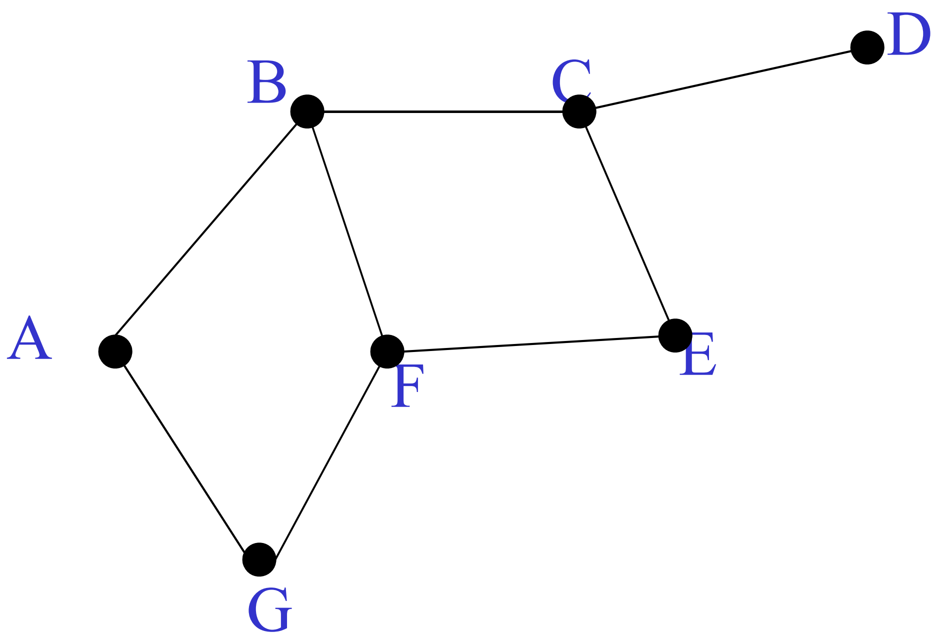
Q={C,F}

Q={F,D,E}

Q={D,E}

Q={E}

BFS starting from A:



Q={A}

Q={B,G}

Q={G,C,F}

Q={C,F}

Q={F,D,E}

Q={D,E}

Q={E}

Q={}

Overall Traversal Order: BFS

- In this example the nodes are traversed from the starting point A in the order:
  A B G C F D E
- Note that the BFS order is that those closest to the start point A occur earliest
- Note that the order is not generally unique; e.g. either of B or G could occur first

Simple DFS

DFS starting from vertex v:

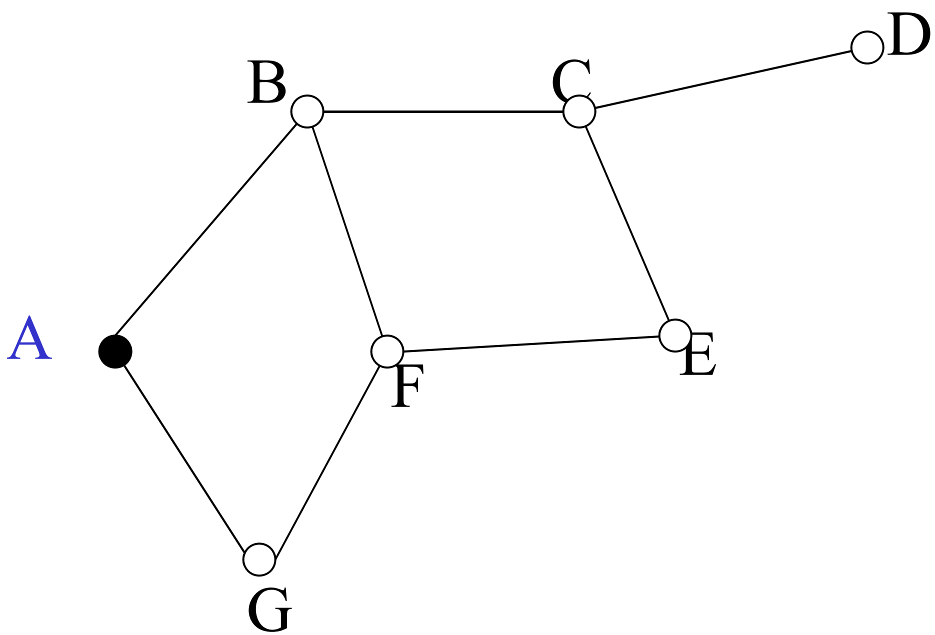
DFS ↔ Stack

```
create a stack S
mark v as visited and push v onto S
while S is non-empty
  peek at the top u of S
  if u has an (unvisited)neighbour w,
          mark w and push it onto S
  else pop S
```

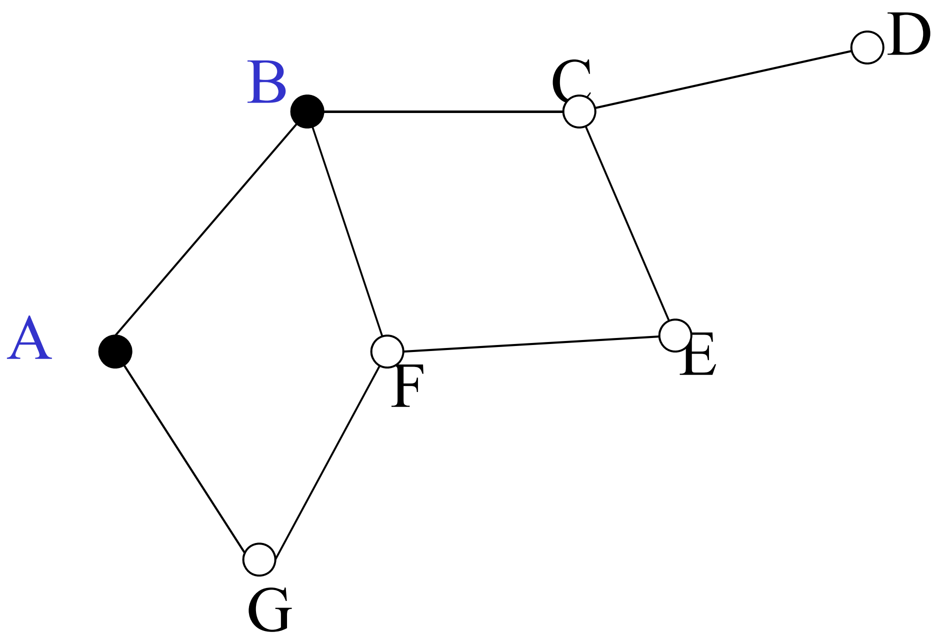
add **one** neighbour at a time

DFS starting from A:

S={A}



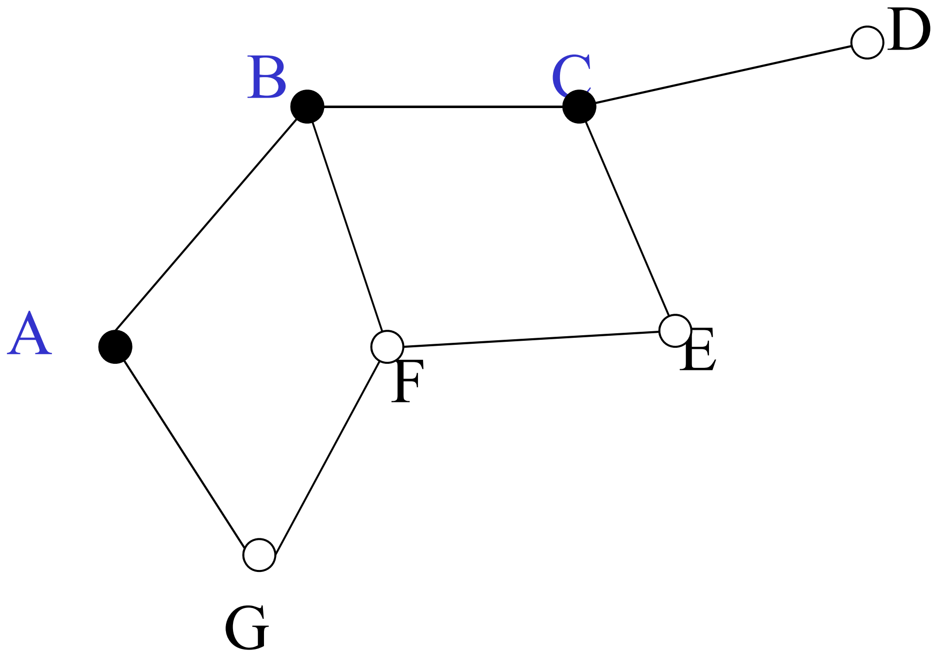
DFS starting from A:



S={A}

S={A,B}

DFS starting from A:

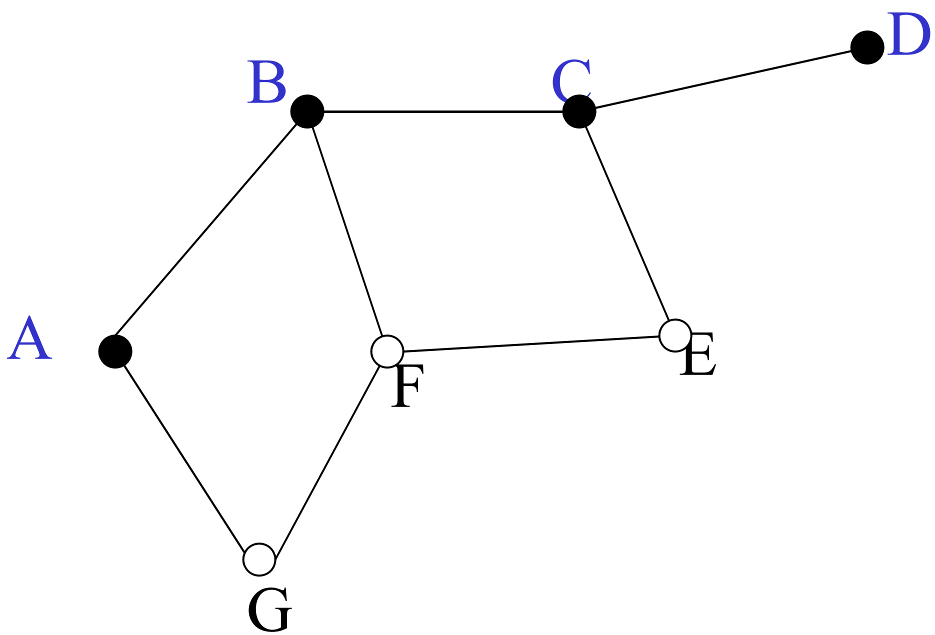


S={A}

S={A,B}

S={A,B,C}

DFS starting from A:



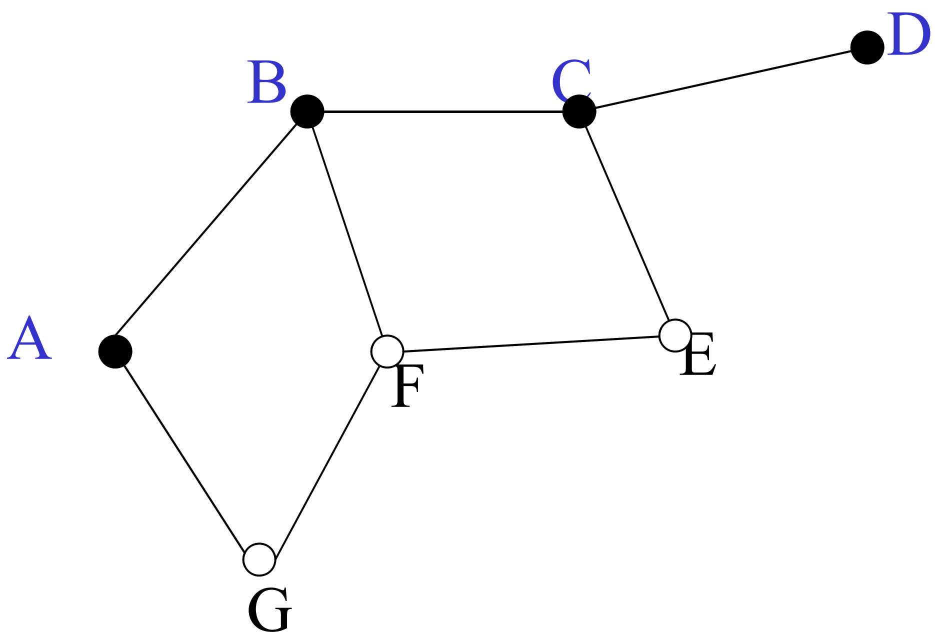
$S=\{A\}$

$S=\{A,B\}$

$S=\{A,B,C\}$

$S=\{A,B,C,D\}$

DFS starting from A:



S={A}

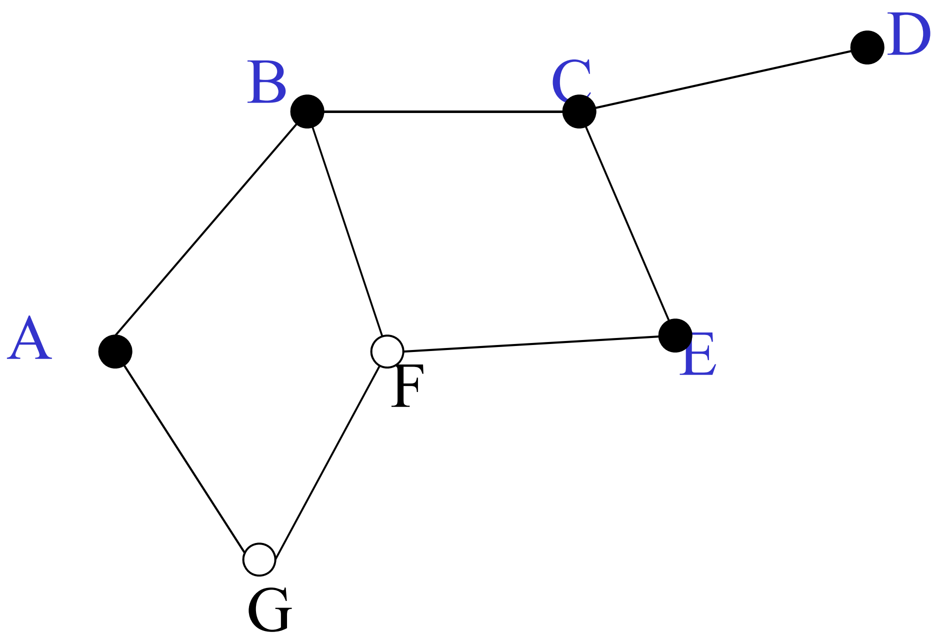
S={A,B}

S={A,B,C}

S={A,B,C,D}

S={A,B,C}

DFS starting from A:



S={A}

S={A,B}

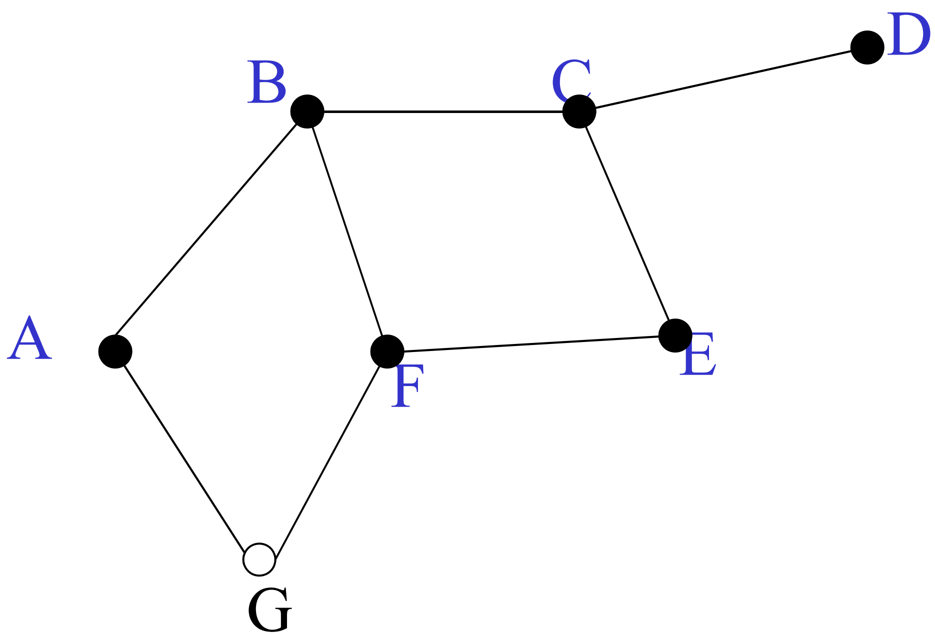
S={A,B,C}

S={A,B,C,D}

S={A,B,C}

S={A,B,C,E}

DFS starting from A:



S={A}

S={A,B}

S={A,B,C}

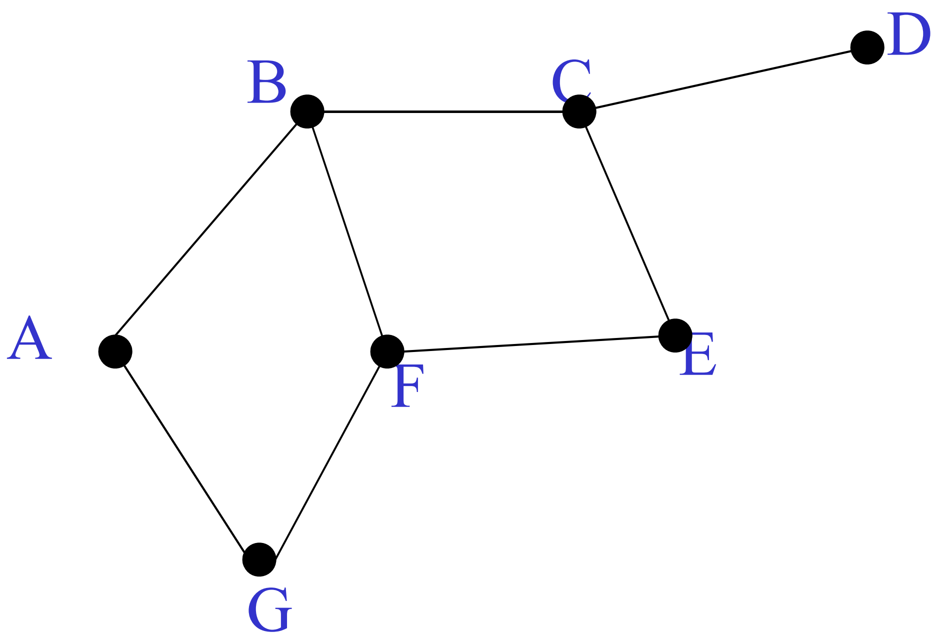
S={A,B,C,D}

S={A,B,C}

S={A,B,C,E}

S={A,B,C, E, F}

DFS starting from A:



S={A}

S={A,B}

S={A,B,C}

S={A,B,C,D}

S={A,B,C}

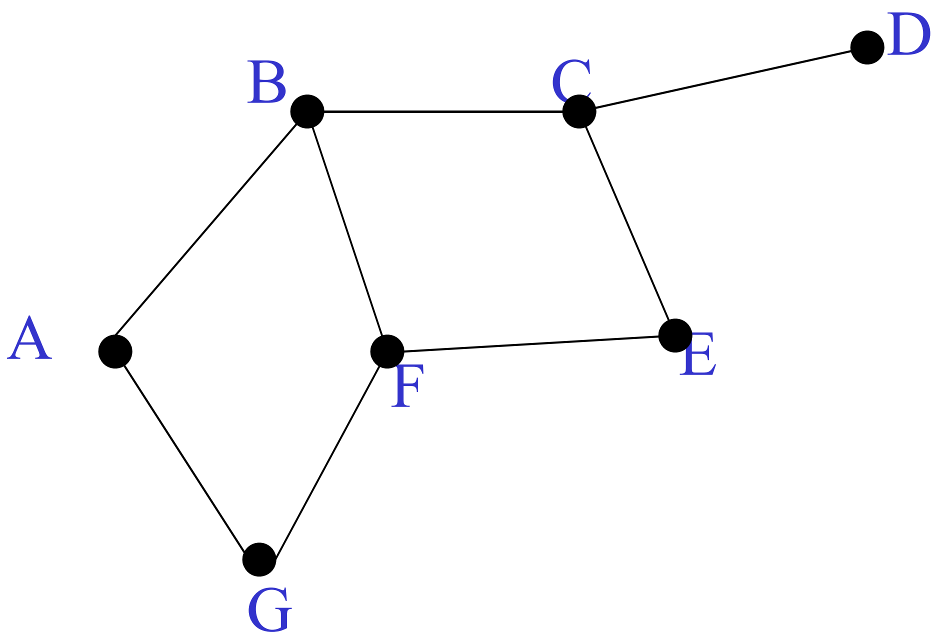
S={A,B,C,E}

S={A,B,C,E,F}

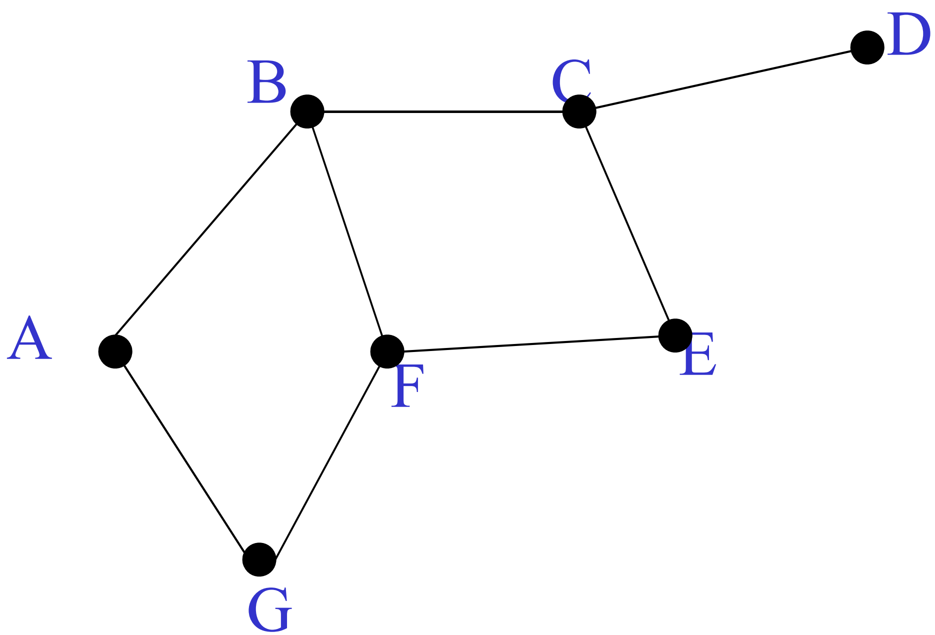
S={A,B,C,E,F,G}

DFS starting from A:

S={A,B,C,E,F}



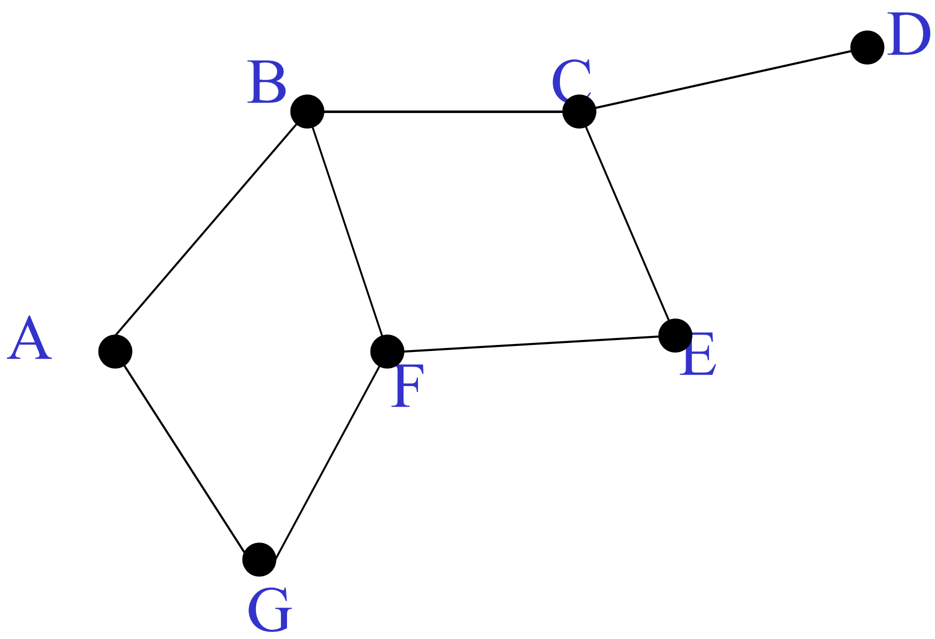
DFS starting from A:



S={A,B,C,E,F}

S={A,B,C,E}

DFS starting from A:

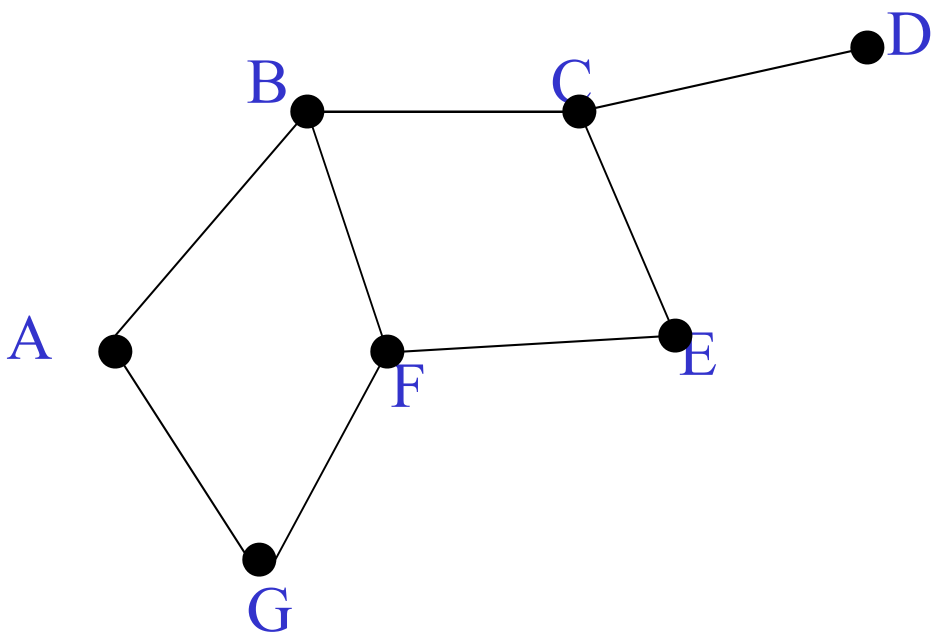


S={A,B,C,E,F}

S={A,B,C,E}

S={A,B,C}

DFS starting from A:



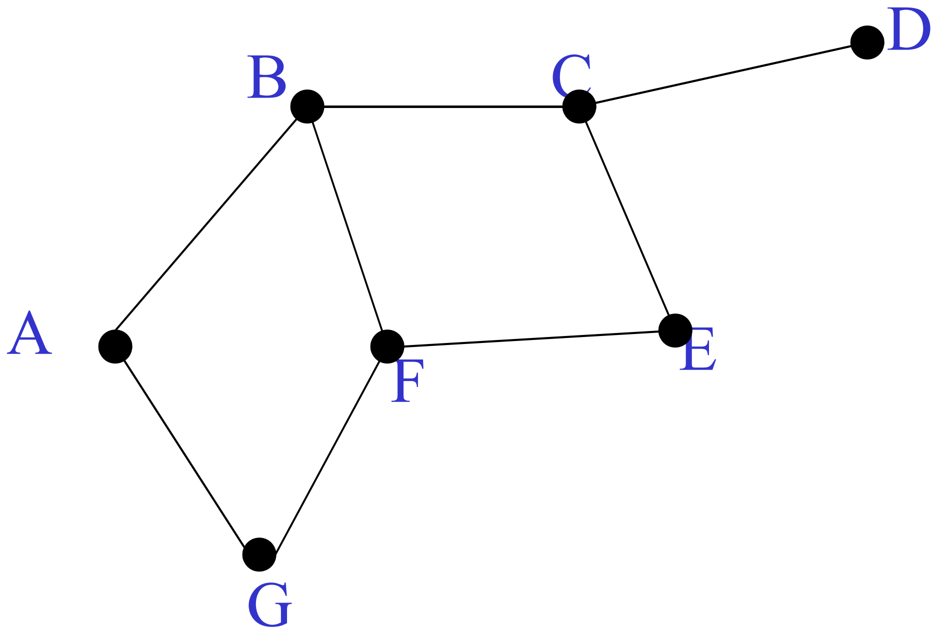
S={A,B,C,E,F}

S={A,B,C,E}

S={A,B,C}

S={A,B}

DFS starting from A:



S={A,B,C,E,F}

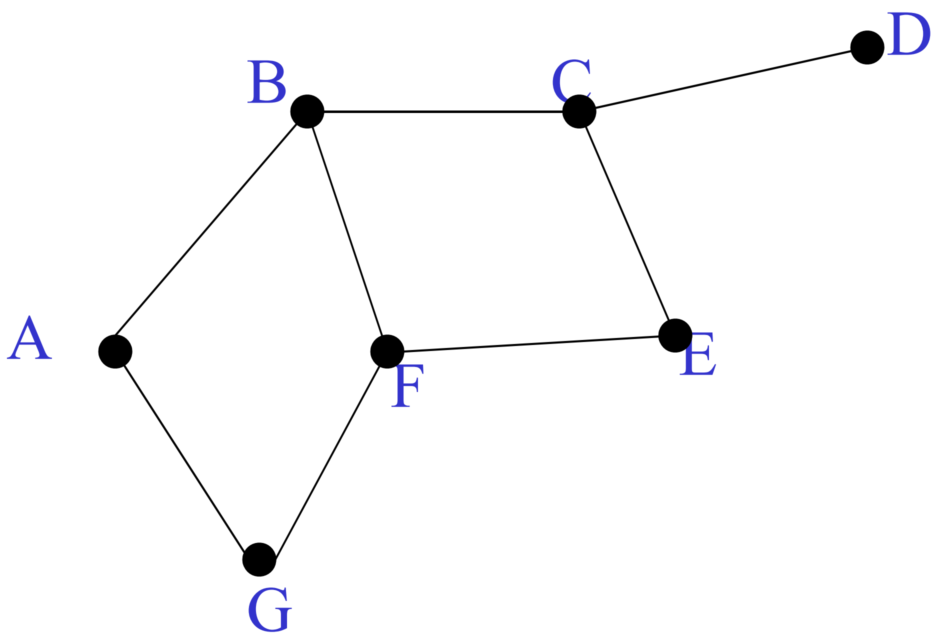
S={A,B,C,E}

S={A,B,C}

S={A,B}

S={A}

DFS starting from A:



S={A,B,C,E,F}

S={A,B,C,E}

S={A,B,C}

S={A,B}

S={A}

S={}

Overall Traversal Order: DFS

- In this example the nodes are traversed from the starting point A in the order:
  A B C D E F G
- Note that the DFS search tends to “dive”.
- Note that the order is not generally unique; e.g. either of B or G could occur first, and if G were selected first then the order would be quite different.

Remarks

- If we discover some node – then the state of the stack provides some path to reach that node
- Might want a directed path
- Could just allowed directed neighbours
  - does not provide shortest path
  - advantage is that is space efficient
  - See auxiliary slides

Complexity of BFS and DFS

- To compute complexity, we will be referring to an adjacency list implementation
- Assume that we have a method which returns the first unmarked vertex adjacent to a given one:
  `GraphNode firstUnmarkedAdj(GraphNode v)`

list of v's neighbours

v → u1(marked) → u2(unmarked) → u3(unmarked)

↑ bookmark

Implementation of firstUnmarkedAdj()

- We keep a pointer into the adjacency list of each vertex so that we do not start to traverse the list of adjacent vertices from the beginning each time.
- Or we use the same iterator for this list, so when we call next() it returns the next element in the list – again does not start from the beginning.

v → u1(marked) → u2(unmarked) → u3(unmarked)

currUnmarkedAdj

Pseudocode for breadth-first search starting from vertex s

```
s.marked = true; // marked is a field in
                 // GraphNode
Queue Q = new Queue();
Q.enqueue(s);
while(! Q.isempty()) {
   v = Q.dequeue();
   u = firstUnmarkedAdj(v);
   while (u != null){ // enqueue & mark all unmarked
      u.marked = true;
      Q.enqueue(u);
      u = firstUnmarkedAdj(v);}}}
```

Pseudocode for DFS

```
s.marked = true;
Stack S = new Stack();
S.push(s);
while(! S.isempty()){
   v = S.peek();
   u = firstUnmarkedAdj(v);
   if (u == null) S.pop();
   else {
      u.marked = true;
      S.push(u);
   }
}
```

Space Complexity of BFS and DFS

For a general graph

- Need a queue/stack of size |V| (the number of vertices).
- Space complexity O(V).

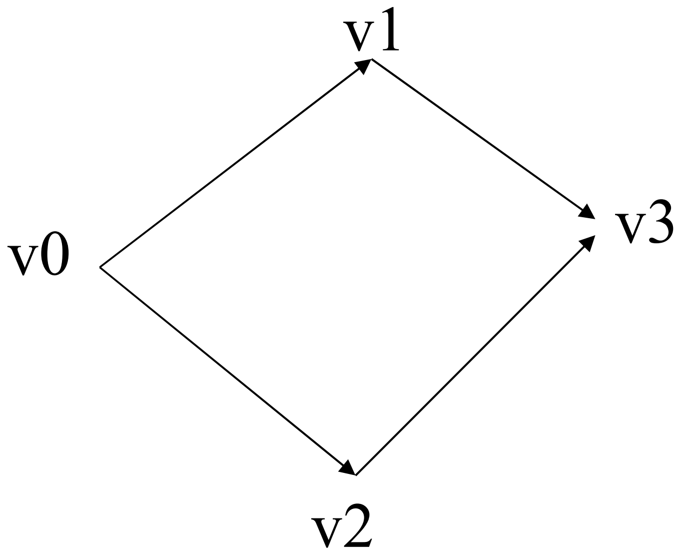
Space Complexity in Trees

- If the graph has special properties then these complexities can be reduced
- Example: suppose the graph is a tree, and we search from the root (G51IAI,etc)
  - In DFS the stack will be O(height),
    - this can be as good as O(log n)
  - In BFS we still need to store all nodes of a level,
    - hence is still O(n)
- Hence in trees, DFS can be a lot more space efficient than BFS

Time Complexity of BFS and DFS

- In terms of the number of vertices V: two nested loops over V, hence at worst $O(V^2)$.
- More useful complexity estimate is in terms of the number of edges.
  - Usually, the number of edges is much less than V^2.

Time complexity of BFS



Adjacency lists:

V E

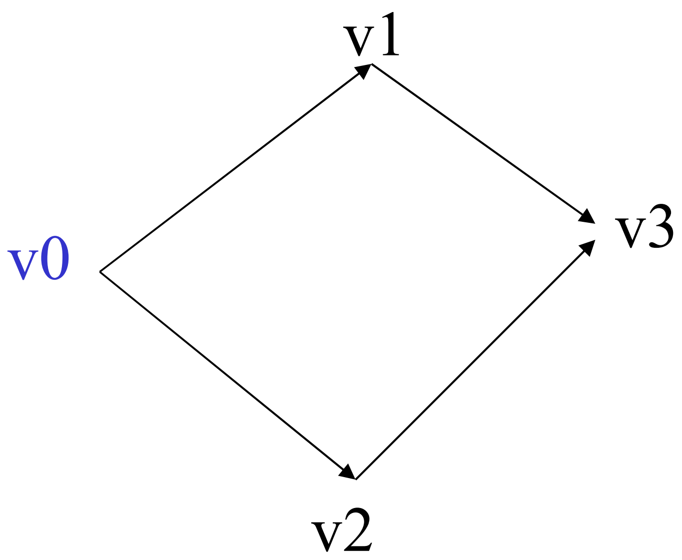
v0: {v1,v2}

v1: {v3}

v2: {v3}

v3: {}

Time complexity of BFS



Adjacency lists:

V E

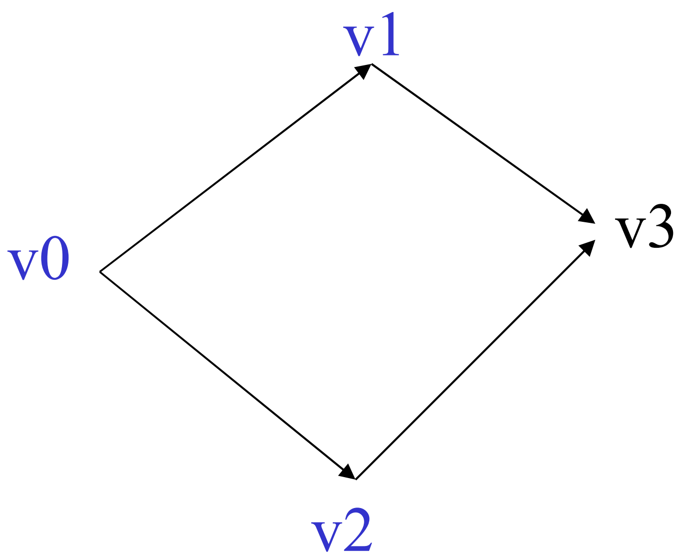
v0: {v1,v2} mark, enqueue v0

v1: {v3}

v2: {v3}

v3: {}

Time complexity of BFS



Adjacency lists:

V E

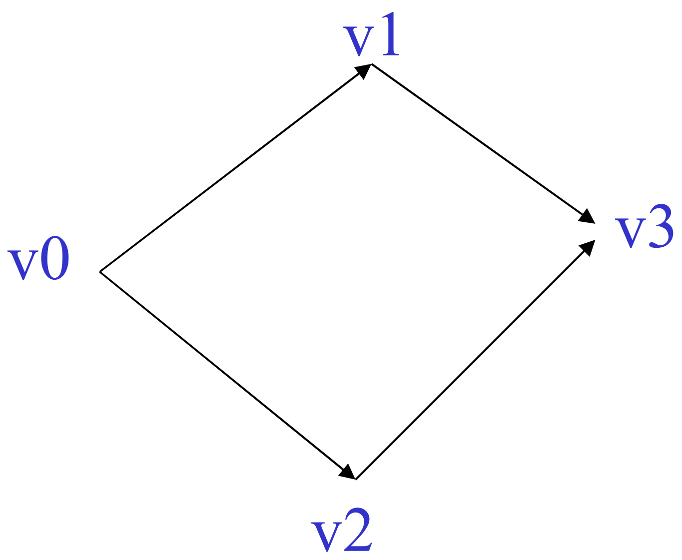
v0: {v1,v2} dequeue v0;
mark, enqueue v1,v2

v1: {v3}

v2: {v3}

v3: {}

Time complexity of BFS



Adjacency lists:

V E

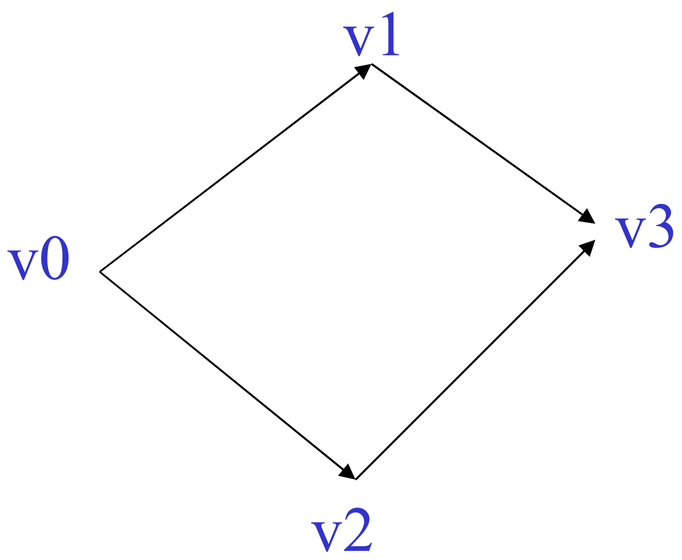
v0: {v1,v2}

v1: {v3} dequeue v1; mark, enqueue v3

v2: {v3}

v3: {}

Time complexity of BFS



Adjacency lists:

V E

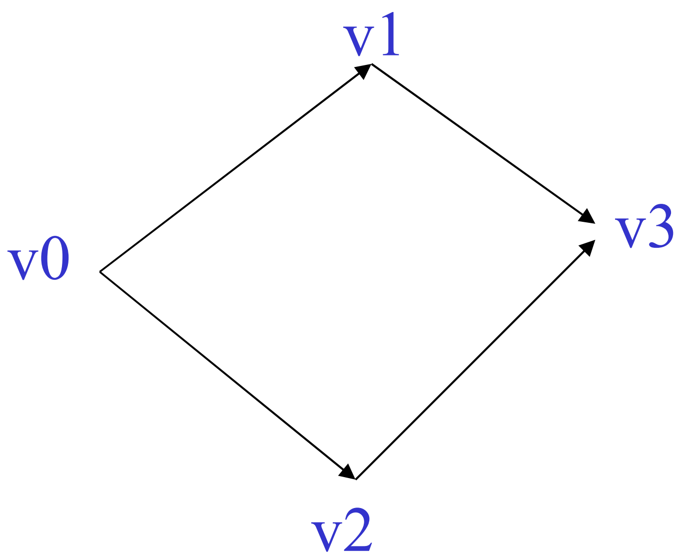
v0: {v1,v2}

v1: {v3}

v2: {v3} dequeue v2, check its adjacency list (v3 already marked)

v3: {}

Time complexity of BFS



Adjacency lists:

V E

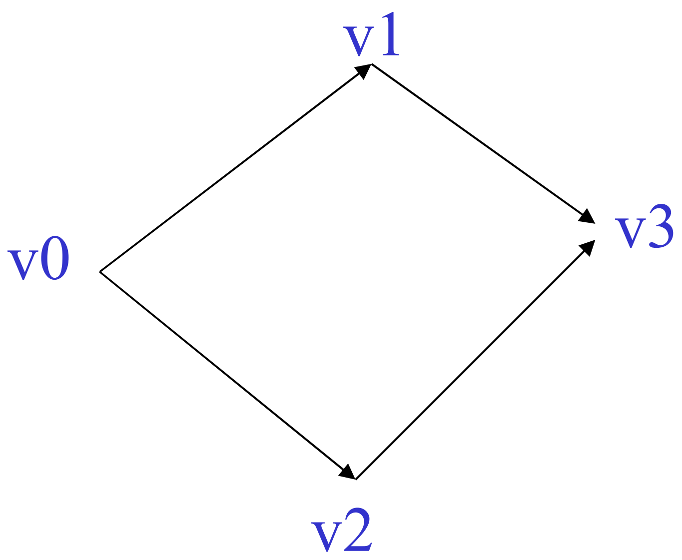
v0: {v1,v2}

v1: {v3}

v2: {v3}

v3: {} dequeue v3; check its adjacency list

Time complexity of BFS



Adjacency lists:

V E

v0: {v1,v2} |E0| = 2

v1: {v3} |E1| = 1

v2: {v3} |E2| = 1

v3: {} |E3| = 0

Total number of steps:
$|V| + |E0| + |E1| + |E2| + |E3|$

$= |V|+|E|$.

Complexity of breadth-first search

- Assume an adjacency list representation, V is the number of vertices, E the number of edges.
- Each vertex is enqueued and dequeued at most once.
- Scanning for all adjacent vertices takes $O(|E|)$ time, since sum of lengths of adjacency lists is $|E|$.
- Gives a $O(|V|+|E|)$ time complexity.

Complexity of depth-first search

- Each vertex is pushed on the stack and popped at most once.
- For every vertex we check what the next unvisited neighbour is.
- In our implementation, we traverse the adjacency list only once. This gives $O(|V|+|E|)$ again.

Exercise

- If you had to implement a webcrawler (e.g. to provide the data for a search engine) then would you use
  - DFS?
  - BFS?
  - something else?

Exercises

For each of DFS and BFS

- Take the pseudo-code and annotate it with appropriate conditions and loop invariants.
  - use these to argue for why the code is correct – i.e. on a connected graph it really will
    - visit every node?
    - visit each node only once?

Summary

- Standard Traversal methods
  - DFS
  - BFS
- (DFS can be modified to detect cycles)
- Complexities:
  - Space is O (|V|)
  - Time is O(|V| + |E|)