DFS, BFS, cycle detection

- Previous lecture
  - What is a graph
  - What are they used for
  - Terminology
  - Implementing graphs

Today and tomorrow:

- Depth-first and breadth-first search
- Using DFS to detect cycles in directed graphs
- Complexity of breadth-first search
- Complexity of depth-first search

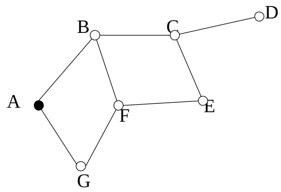
Breadth first search

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
```

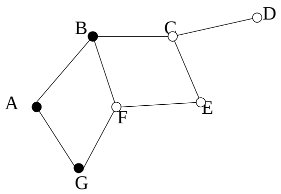
BFS starting from A:

Q={A}



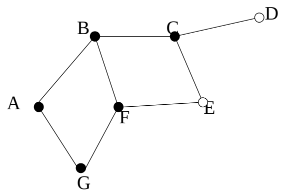
BFS starting from A:

Q={A}
Q={B,G}

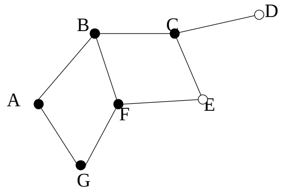


BFS starting from A:

Q={A}
Q={B,G}
Q={G,C,F}

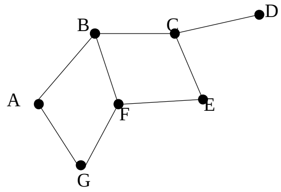


BFS starting from A:



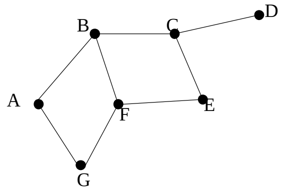
Q={A}
Q={B,G}
Q={G,C,F}
Q={C,F}

BFS starting from A:



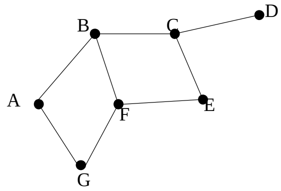
Q={A}
Q={B,G}
Q={G,C,F}
Q={C,F}
Q={F,D,E}

BFS starting from A:



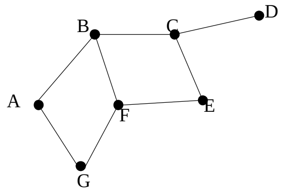
Q={A}
Q={B,G}
Q={G,C,F}
Q={C,F}
Q={F,D,E}
Q={D,E}

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={}

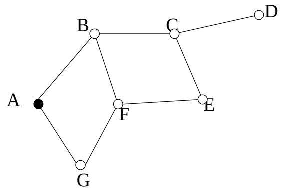
Simple DFS

DFS starting from vertex v:

```
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
```

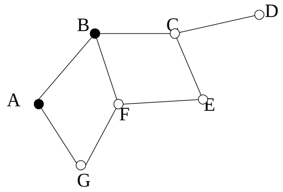
DFS starting from A:

S={A}



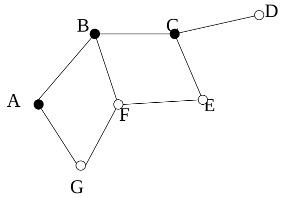
DFS starting from A:

S={A}
S={A,B}



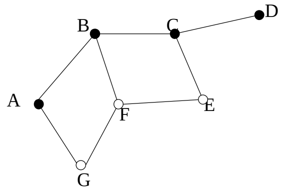
DFS starting from A:

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



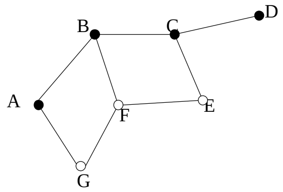
DFS starting from A:

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



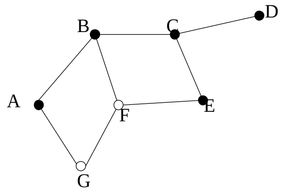
DFS starting from A:

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



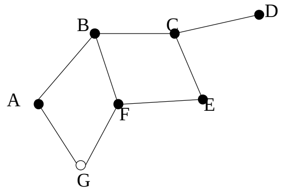
DFS starting from A:

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



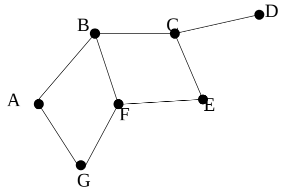
DFS starting from A:

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}



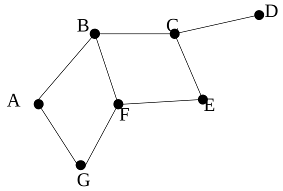
DFS starting from A:

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}



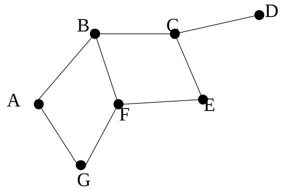
DFS starting from A:

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



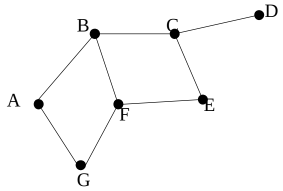
DFS starting from A:

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



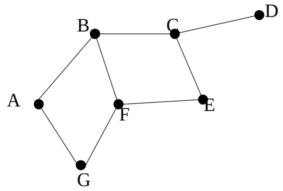
DFS starting from A:

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

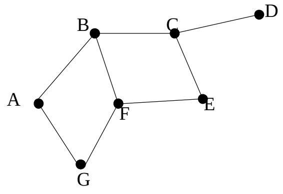


DFS starting from A:

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

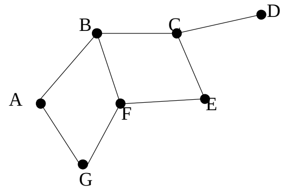


DFS starting from A:



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={}

Modification of depth first search

- How to get DFS to detect cycles in a directed graph:

idea: if we encounter a vertex which is already on the stack, we found a loop (stack contains vertices on a path, and if we see the same vertex again, the path must contain a cycle).

- Instead of visited and unvisited, use three colours:
  - **white =** unvisited
  - **gray =** on the stack
  - **black =** finished (we backtracked from it, seen everywhere we can reach from it)

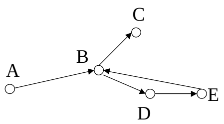
Modification of depth first search

Modified DFS starting from **`v:`**

```
all vertices coloured white
create a stack S
colour v gray and push v onto S
while S is non-empty
  peek at the top u of S
  if u has a gray neighbour, there is a
  cycle
  else if u has a white neighbour w,
  colour w gray and push it onto S
  else colour u black and pop S
```

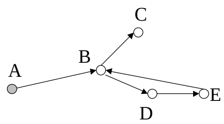
Tracing modified DFS from A

S = {}

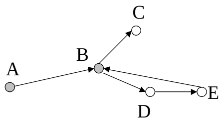


Tracing modified DFS from A

S = {}
S = A



Tracing modified DFS from A

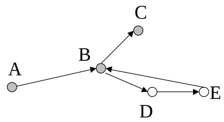


S = {}

S = A

B
S = A

Tracing modified DFS from A



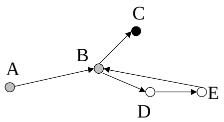
S = {}

S = A

B
S = A

C
B
S = A

Tracing modified DFS from A



S = {}

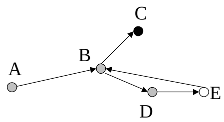
S = A

B
S = A

C
B
S = A

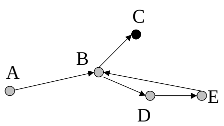
B
pop: S = A

Tracing modified DFS from A



push: D
B
S = A

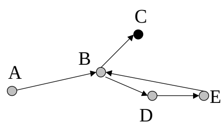
Tracing modified DFS from A



push: D
B
S = A

E
D
B
S = A

Tracing modified DFS from A



push: D
B
S = A

E
D
B
S = A

E has a gray neighbour: B!
Found a loop!

Pseudocode for BFS and DFS

- To compute complexity, I 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.

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){
      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);
   }
}
```

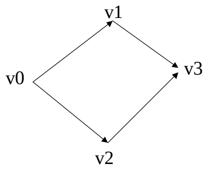
Space Complexity of BFS and DFS

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

Time Complexity of BFS and DFS

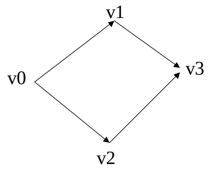
- In terms of the number of vertices V: two nested loops over V, hence $O(V^2)$.
- More useful complexity estimate is in terms of the number of edges. Usually, the number of edges is less than V^2.

Time complexity of BFS



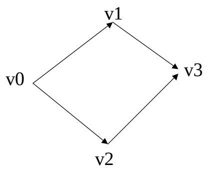
Adjacency lists:
V E
v0: {v1,v2}
v1: {v3}
v2: {v3}
v3: {}

Time complexity of BFS



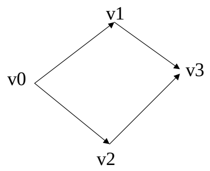
Adjacency lists:
V E
v0: {v1,v2} mark, enqueue v0
v1: {v3}
v2: {v3}
v3: {}

Time complexity of BFS



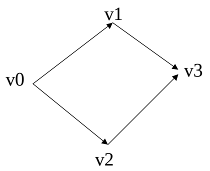
Adjacency lists:
V E
v0: {v1,v2} dequeue v0; mark, enqueue v1,v2
v1: {v3}
v2: {v3}
v3: {}

Time complexity of BFS



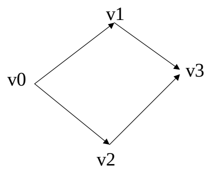
Adjacency lists:
V E
v0: {v1,v2}
v1: {v3} dequeue v1; mark, enqueue v3
v2: {v3}
v3: {}

Time complexity of BFS



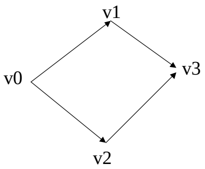
Adjacency lists:
V E
v0: {v1,v2}
v1: {v3}
v2: {v3} dequeue v2, check its adjacency list (v3 already marked)
v3: {}

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.