Graph traversals

- In this lecture, we look at two ways of visiting all vertices in a graph: breadth-first search and depth-first search.
- 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.

Breadth first search

BFS starting from vertex v:

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

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}
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}
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}
S={A}
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
    - grey = 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 \( v \) grey and push \( v \) onto \( S \)
  - while \( S \) is non-empty:
    - peek at the top \( u \) of \( S \)
    - if \( u \) has a grey neighbour, there is a cycle
    - else if \( u \) has a white neighbour \( w \), colour \( w \) grey and push it onto \( S \)
    - else colour \( u \) black and pop \( S \)

Tracing modified DFS from A

- Start with the initial stack: \( S = \{ \} \)

- \( S = A \)

- \( S = \{ A, B \} \)

- \( S = \{ A, B, C \} \)

- \( S = \{ A, B, C, E \} \)

- \( S = \{ A, B, C, E, F \} \)
Tracing modified DFS from A

S = {}

S = A

B

S = A

C

B

S = A

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

Tracing modified BFS from A

push: D
B
S = A

E has a grey 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:

```java
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.

Pseudocode for breadth-first search starting from vertex s

```java
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

```java
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:

- v0: [v1, v2]
- v1: [v3]
- v2: [v3]
- v3: []
Time complexity of BFS

Adjacency lists:

\[ \begin{array}{c|c}
V & E \\
--- & --- \\
v0: \{v1,v2\} & mark, enqueue v0 \\
v1: \{v3\} & mark, enqueue v1, v2 \\
v2: \{v3\} & v1: \{v3\} \\
v3: \{\} & v2: \{v3\} \\
\end{array} \]

Time complexity of BFS

Adjacency lists:

\[ \begin{array}{c|c}
V & E \\
--- & --- \\
v0: \{v1,v2\} & dequeue v0; mark, enqueue v1,v2 \\
v1: \{v3\} & v1: \{v3\} \\
v2: \{v3\} & v2: \{v3\} \\
v3: \{\} & v3: \{\} \\
\end{array} \]

Time complexity of BFS

Adjacency lists:

\[ \begin{array}{c|c}
V & E \\
--- & --- \\
v0: \{v1,v2\} & dequeue v1; mark, enqueue v3 \\
v1: \{v3\} & v1: \{v3\} \\
v2: \{v3\} & v2: \{v3\} \\
v3: \{\} & v3: \{\} \\
\end{array} \]

Time complexity of BFS

Adjacency lists:

\[ \begin{array}{c|c}
V & E \\
--- & --- \\
v0: \{v1,v2\} & dequeue v2, check its adjacency list (v3 already marked) \\
v1: \{v3\} & v3: \{\} \\
v2: \{v3\} & \text{check its adjacency list} \\
v3: \{\} & \text{check its adjacency list} \\
\end{array} \]

Time complexity of BFS

Adjacency lists:

\[ \begin{array}{c|c}
V & E \\
--- & --- \\
v0: \{v1,v2\} & |E_0| = 2 \\
v1: \{v3\} & |E_1| = 1 \\
v2: \{v3\} & |E_2| = 1 \\
v3: \{\} & |E_3| = 0 \\
\end{array} \]

Total number of steps:

\[ |V| + |E_0| + |E_1| + |E_2| + |E_3| = |V| + |E| \]
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.