COMP 250

Lecture 29

graph traversal

Nov. 15/16, 2017
• breadth first
• depth first

• Non-recursive graph traversal
• depth first

• Recursive graph traversal
• depth first
Heads up!

There were a few mistakes in the slides for Sec. 001 for today’s lecture. So if you are following the lecture recordings and using these (corrected) slides, then you will notice some differences.
Recall: tree traversal (recursive)

def depthfirst__Tree (root):
    if (root is not empty):
        root.visited = true  // “preorder”
        for each child of root
            depthfirst__Tree( child )

}
Graph traversal (recursive)

Need to specify a starting vertex.

Visit all nodes that are “reachable” by a path from a starting vertex.
Graph traversal (recursive)

def depthFirst_Graph(v):
    v.visited = true
    for each w such that (v,w) is in E            // w in v.adjList
        _______?____________________
    }

    // Here “visiting” just means “reaching”
Graph traversal (recursive)

```java
depthFirst_Graph(v){
    v.visited = true
    for each w such that (v,w) is in E  // w in v.adjList
        if ! (w.visited)  // avoids cycles
            depthFirst_Graph(w)
}
```

// Here “visiting” just means “reaching”
Call Stack for depthFirst(a)

def depthFirst_Graph(v):
    v.visited = true
    for each w such that (v,w) is in E
        if ! w.visited
            depthFirst_Graph(w)

depthFirst_Graph(a)
Call Stack for depthFirst(a)

depthFirst_Graph(v){
    v.visited = true
    for each w such that (v,w) is in E
        if ! (w.visited)
            depthFirst_Graph(w)
}
Call Stack for depthFirst(a)

depthFirst_Graph(v){
    v.visited = true
    for each w such that (v,w) is in E
        if ! (w.visited)
            depthFirst_Graph(w)
}
Call Stack for depthFirst(a)

depthFirst_Graph(v){
    v.visited = true
    for each w such that (v,w) is in E
        if ! (w.visited)
            depthFirst_Graph(w)
}
Call Stack for `depthFirst(a)`

```
def depthFirst_Graph(v):
    v.visited = True
    for each w such that (v,w) is in E
        if ! (w.visited)
            depthFirst_Graph(w)
```
What is the call tree for depthFirst( a )?

Adjacency List

a - (b,d)
b - (a,c,e)
c - (b,f)
d - (a,e,g)
e - (b,d,f,h)
f - (c,e,i)
g - (d,h)
h - (e,g,i)
i - (f,h)
Example 2

call tree for depthFirst(a)
Q: Non-recursive graph traversal?

A: Similar to tree traversal: Use a stack or a queue.
Recall: depth first tree traversal (with a slight variation)

```python
treeTraversalUsingStack(root){
    initialize empty stack s
    visit root
    s.push(root)
    while s is not empty {
        cur = s.pop()
        for each child of cur{
            visit child
            s.push(child)
        }
    }
}
```

Visit a node before pushing it onto the stack.

Every node in the tree gets visited, pushed, and then popped.
Generalize to graphs...

```java
graphTraversalUsingStack(v){
    initialize empty stack s
    v.visited = true
    s.push(v)
    while (!s.empty) {
        u = s.pop()
        for each w in u.adjList{
            if (!w.visited){                        // the only new part
                w.visited = true
                s.push(w)
            }
        }
    }
}
```
Example: graphTraversalUsingStack(a)
Example: `graphTraversalUsingStack(a)`

The traversal defines a tree, but it is not a “call tree”. Why not?

’a’ is popped and both ‘b’ and ‘d’ are pushed.
Example: `graphTraversalUsingStack(a)`

'\textcolor{red}{d'} is popped and both 'e' and 'g' are pushed.
Example: graphTraversalUsingStack(a)

'g' is popped and 'h' is pushed.
Example: `graphTraversalUsingStack(a)`

`g h i`

`d e e e e`

`a b b b b b`

‘h’ is popped and ‘i’ is pushed.
Example: `graphTraversalUsingStack(a)`

```
g h i f
---
d e e e e e
a b b b b b
```

‘i’ is popped and ‘f’ is pushed.
Example: `graphTraversalUsingStack(a)`

`g` `h` `i` `f` `c`  
`d` `e`  
`a` `b` `b` `b` `b` `b` `b`  

‘f’ is popped and ‘c’ is pushed.
Example: graphTraversalUsingStack(a)

Order of nodes visited: abdeghifc
Recall: breadth first tree traversal (see lecture 20)

for each level $i$
visit all nodes at level $i$

```python
def treeTraversalUsingQueue(root):
    initialize empty queue q
    q.enqueue(root)
    while q is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
```
Breadth first graph traversal

Given an input vertex, find all vertices that can be reached by paths of length 1, 2, 3, 4, ....
Breadth first graph traversal

```javascript
graphTraversalUsingQueue(v) {
  initialize empty queue q
  v.visited = true
  q.enqueue(v)
  while (! q.empty) {
    u = q.dequeue()
    for each w in u.adjList{
      if (!w.visited) {
        w.visited = true
        q.enqueue(w)
      }
    }
  }
}
```
Example

graphTraversalUsingQueue(c)

queue
c

![Graph Diagram]

- a → c
- c → d
- c → f
- e → f
- f → b
Example

graphTraversalUsingQueue(c)

queue
cf
Example

graphTraversalUsingQueue(c)

queue

\begin{itemize}
  \item c
  \item f
  \item be
\end{itemize}

Both ‘b’, ‘e’ are visited and enqueued before ‘b’ is dequeued.
Example

`graphTraversalUsingQueue(c)`

queue

```
c
f
be
e
```
graphTraversalUsingQueue(c)

It defines a tree whose root is the starting vertex. It finds the shortest path (number of vertices) to all vertices reachable from starting vertex.
Example: `graphTraversalUsingQueue(a)`
Example: `graphTraversalUsingQueue(a)`

```plaintext
1  2  3
a  b  c
  d  e  f
g  h  i
```

Output: `a bd`
Example: `graphTraversalUsingQueue(a)`

```
1                     2                     4
a---b---c
  |     |
  v     v
3     5
d---e---f
  |     |
  v     v
g---h---i
```

Output:
```
a
bd
dce
```
Example: graphTraversalUsingQueue(a)
Example: graphTraversalUsingQueue(a)

1. a
2. b
3. d
4. c
5. e
6. g
7. f
8. h
9. i

1. a
2. bd
3. dce
4. ceg
5. egf
Example: graphTraversalUsingQueue(a)
Example: graphTraversalUsingQueue(a)
Example: graphTraversalUsingQueue(a)

```
1                     2                    4
a                       b                       c
3                       5                       7
6                      8                       9
da                      e                       f
6                      8                       9
g                       h                       i
```

```
1                     2                    4
a                       b                       c
3                       5                       7
6                      8                       9
da                      e                       f
6                      8                       9
g                       h                       i
```

```
a
bd
dce
ceg
egf
gfh
fh
hi
```
Example: `graphTraversalUsingQueue(a)`

Note order of nodes visited:
paths of length 1, 2, 3, 4
Example: graphTraversalUsingQueue(a)

The traversal defines a tree, but it is not a “call tree”. Why not?
Recall: How to implement a Graph class in Java?

class Graph<T> {
    HashMap<String, Vertex<T>> vertexMap;
}

class Vertex<T> {
    ArrayList<Edge> adjList;
    T element;
    boolean visited;
}

class Edge {
    Vertex endVertex;
    double weight;
}
}
HEADS UP! Prior to traversal, ...

for each w in V
  w.visited = false

How to implement this?
HEADS UP! Prior to traversal, ....

for each w in V
  w.visited = false

How to implement this?

class Graph<T> {
  HashMap<String, Vertex<T>> vertexMap;
  :
  public void resetVisited() {
    }
  }
}
HEADS UP! Prior to traversal, ....

for each w in V
w.visited = false

How to implement this?

class Graph<T> {
    HashMap<String, Vertex<T>> vertexMap;
    
    public void resetVisited() {
        for (Vertex<T> v : vertexMap.values()) {
            v.visited = false;
        }
    }
}

[ASIDE: I did something unnecessarily complicated on the Sec.001 slides. What I have above is better.]