COMP20003
Algorithms and Data Structures
Traversing Trees and Graphs

Nir Lipovetzky
Department of Computing and Information Systems
University of Melbourne
Semester 2

Traversal

- **Traverse**: to pass or move over, along, or through
- **Tree traversal**: the process of visiting (examining or updating) each node exactly once, in a systematic way
- **Graph traversal**: the process of visiting all the nodes in a graph

Tree traversal is a special case of graph traversal

Traversal

Graph traversal *vs.* Tree traversal

Graph traversal complications due to:

- Possible cycles
- Not necessarily connected

… Lets start with Tree traversal

Starting with trees: bst dfs traversal depth-first search

Depth-first tree search can be done as:

- In-order
- Pre-order
- Post-order

Recursive in-order search: binary tree

```
void inorder(node_t* t)
{
  if(t==NULL) return();
  inorder(t->left);
  visit(t);   /* e.g. print value */
  inorder(t->right);
}
```

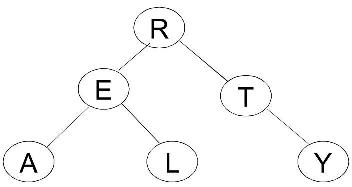

Recursive pre-order search: binary tree

```
void preorder(node_t* t)
{
 if(t==NULL) return();
 visit(t);   /* visit first */
 preorder(t->left);
 preorder(t->right);
}
```


Non-recursive pre-order search: DFS - explicit stack

```
void preorder(stack_t* st, node_t* t)
{
    push(st,t);
    while(!stackempty(st))
    {
          t= pop(st); visit(t);
          if(t->r != NULL) push(st,t->r);
          if(t->l != NULL) push(st,t->l);
    }
}
/* note: stack contains pointers into the tree */
```

Depth-first search *vs.* breadth-first search



Breadth-first tree search: use a queue

```
void preorder(queue* Q, node_t* t)
{
    enQ(Q,t);
    while(!emptyQ(Q))
    {
          t = deQ(Q); visit(t);
          if(t->l != NULL) enQ(Q,t->l);
          if(t->r != NULL) enQ(Q,t->r);
    }
}
/* note: queue contains pointers into the tree */
```

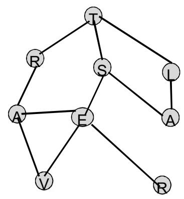

Tree traversal: assumptions

- **Assumes** every node is reachable from the root
- **Assumes** every node has only one parent, can only be visited once

Graph traversal needs to make sure that:

- Every node is reached
- Every node is visited only once

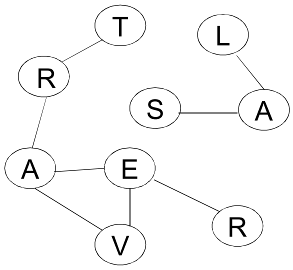
Traversing a connected graph (depth first)



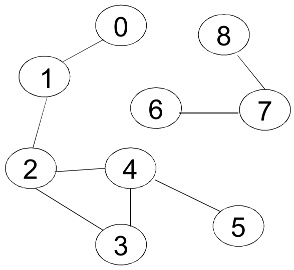
Need to mark nodes as visited

Traversing an unconnected graph (depth first)

- Need to traverse each connected component
- Still need to mark nodes as visited

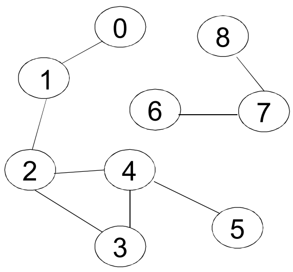


Traversing an unconnected graph (depth first)



Traversing an unconnected graph (depth first)

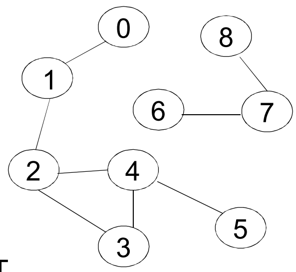
```
int  order=0;
```



Traversing an unconnected graph (depth first)

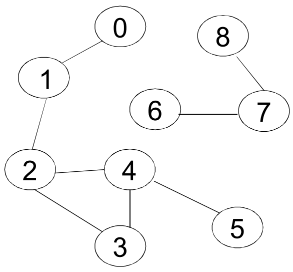
Matrix

	0	1	2	3	4	5	6	7	8
0		T							
1	T		T						
2		T		T	T				
3			T		T				
4			T	T		T			
5					T				
6								T	
7							T		T
8								T	



Traversing an unconnected graph (depth first)

Adjacency List
0 → 1
1→2
2→3→4→1
3→2→4
4→3→2→5
5→4
6→7
7→8→6
8→7



`visited[]` array: keeping track of what's been done

```
/* invoke an array to track whether or not a
node has already been visited */
int  visited[V];
listdfs()
{
    int k;
    /* initialize – no nodes yet visited */
    for(k = 0; k < V; k++)
        visited[k]= 0;
}
```


Adjacency list node

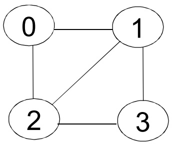
```
/* adjacency list is an array of pointers to
nodes; node is struct with value (nodeID)
and next ptr*/
struct node{
    int  value;
    struct node *next;
};
struct node*  adj[V];
```


Visiting nodes; updating the `visited[]` array

```
int  visited[V];
int  order=0;  /*keeps track of the order in
                 which nodes are visited */
void visitDFS(int k)
{
   struct node* t;
   visited[k] = ++order;
   for(t = adj[k]; t != NULL; t = t->next){
        if( !visited[t->v] )
             visitDFS( t->v );
   }
}
```

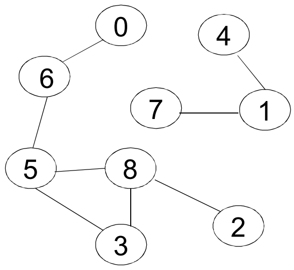
Example dfs graph traversal

Adjacency List
0→1→2
1→0→2→3
2→0→1→3
3→1→2



Example dfs graph traversal

Adjacency List
0→6
1→4→7
2→8
3→5→8
4→1
5→3→6→8
6→0→5
7→1
8→2→3→5



Graph dfs: Analysis

- Fill in the `visited[]` array:
- Examine (at most) each edge twice:

Graph dfs: Analysis

- Fill in the `visited[]` array:
  - |V|
- Examine (at most) each edge twice:
  - |E|
- Overall: |V|+|E|

Graph *breadth*-first search

- Again, modify the tree bfs, to make sure that:

Graph *breadth*-first search

- Again, modify the analogous tree search, to make sure that:
  - Every node is visited, even if the graph is not connected, and
  - Every node is visited only once

brfs `visit()`

```
int visited[V]; int order=0;
void visitBFS(int k){
   struct node* t;
   enQ(Q,k);
   while(!Qempty(Q)){
       k = deQ(Q);
       if( !visited[k] ){
            visited[k] = ++order;
            for(t = adj[k]; t != NULL; t = t->next){
                if( !visited[t->num] )
                        enQ(Q,t->num);
                }
        }
   }
}
```


Breadth-first graph search

```
int  visited[V];
void listbfs()
{
     int k;
     for( k = 0; k < V; k++ ) visited[k]= 0;
     for( k = 0; k < V; k++ )
          if(!visited[k])
               visitBFS(k);
}
```

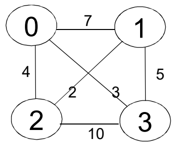
Weighted graphs

So far, we used arbitrary ordering of the connected nodes (determined by position in adjacency list or matrix)

- For weighted graphs, it might be nice to get the nodes out in order of distance
  - Distance = sum of weights

Example weighted graph brfs

Adjacency List
0→1→2
1→0→2→3
2→0→1→3
3→1→2



Previous visit order from node 0:

- But if these are restaurants and bars, and we want to go to a nearby bar

From restaurant 0…

Priority Queues

We can still use a queue, but we make that a priority queue (PQ).

- Chapter 5, Skiena book