Trees and Graphs

CSE 2320 – Algorithms and Data Structures
Vassilis Athitsos
University of Texas at Arlington
Graphs

• A graph is formally defined as:
  – A set $V$ of **vertices** (also called **nodes**).
  – A set $E$ of **edges**. Each edge is a pair of two vertices in $V$.

• Graphs can be directed or undirected.

• In a directed graph, edge $(A, B)$ means that we can go (using that edge) from $A$ to $B$, but **not** from $B$ to $A$.
  – We can have both edge $(A, B)$ and edge $(B, A)$ if we want to show that $A$ and $B$ are linked in both directions.

• In an undirected graph, edge $(A, B)$ means that we can go (using that edge) from both $A$ to $B$ and $B$ to $A$. 
Example: of an Undirected Graph

• A graph is formally defined as:
  – A set $V$ of vertices.
  – A set $E$ of edges. Each edge is a pair of two vertices in $V$.

• What is the set of vertices on the graph shown here?
  – $\{0, 1, 2, 3, 4, 5, 6, 7\}$

• What is the set of edges?
  – $\{(0, 1), (0, 2), (0, 5), (0, 6), (0, 7), (3, 4), (3, 5), (4, 5), (4, 6), (4, 7)\}$. 
Trees

• Trees are a natural data structure for representing several types of data.
  – Family trees.
  – Organizational chart of a corporation, showing who supervises who.
  – Folder (directory) structure on a hard drive.
  – Parsing an English sentence into its parts.
A Family Tree (from Wikipedia)
An Organizational Chart (from Wikipedia)
A Parse Tree (from Wikipedia)

Constituency-based parse tree

John, hit, the, ball.
Paths

• A path in a tree is a list of distinct vertices, in which successive vertices are connected by edges.
  – No vertex is allowed to appear twice in a path.
• Example: ("Joseph Wetter", "Jessica Grey", "Jason Grey", "Hanna Grey")
Trees and Graphs

• Are trees graphs?
  – Always?
  – Sometimes?
  – Never?

• Are graphs trees?
  – Always?
  – Sometimes?
  – Never?
Trees and Graphs

• All trees are graphs.
• Some graphs are trees, some graphs are not trees.
• What is the distinguishing characteristic of trees?
• What makes a graph a tree?
Trees and Graphs

• All trees are graphs.
• Some graphs are trees, some graphs are not trees.
• What is the distinguishing characteristic of trees?
  – What makes a graph a tree?
• A tree is a graph such that any two nodes (vertices) are connected by precisely one path.
  – If you can find two nodes that are not connected by any path, then the graph is not a tree.
  – If you can find two nodes that are connected to each other by more than one path, then the graph is not a tree.
Example

- Are these graphs trees?
• Are these graphs trees?

Yes, this is a tree. Any two vertices are connected by exactly one path.

No, this is not a tree. For example, there are two paths connecting node 5 to node 4.
Example

• Are these graphs trees?
Example

• Are these graphs trees?

Yes, this is a tree. Any two vertices are connected by exactly one path.

No, this is not a tree. For example, there is no path connecting node 7 to node 4.
Root of the Tree

• A rooted tree is a tree where one node is designated as the root.
• Given a tree, ANY node can be the root.
Terminology

• A rooted tree is a tree where one node is explicitly designated as the root.
  – From now on, as is typical in computer science, all trees will be rooted trees
  – We will typically draw trees with the root placed at the top.

• Each node has exactly one node directly above it, which is called a parent.

• If Y is the parent of X, then Y is the node right after X on the path from X to the root.
Terminology

• If Y is the parent of X, then X is called a child of Y.
  – The root has no parents.
  – Every other node, except for the root, has exactly one parent.

• A node can have 0, 1, or more children.

• Nodes that have children are called internal nodes or non-terminal nodes.

• Nodes that have no children are called leaves or terminal nodes, or external nodes.
Terminology

- The **level** of the root is defined to be 0.
- The **level** of each node is defined to be $1 + \text{level}$ of its parent.
- The **height** of a tree is the maximum of the levels of all nodes in the tree.
M-ary Trees

• An **M-ary tree** is a tree where every node is either a leaf or it has **exactly** $M$ children.

• Example: **binary** trees, **ternary** trees, ...

Is this a binary tree?

Is this a binary tree?
M-ary Trees

• An **M-ary tree** is a tree where every node is either a leaf or it has **exactly** $M$ children.

• Example: **binary** trees, **ternary** trees, ...

This is **not** a binary tree, node 3 has 1 child.

This is a binary tree.
Ordered Trees

• A rooted tree is called **ordered** if the order in which we list the children of each node is significant.

• For example, if we have a binary ordered tree, we will refer to the left child and the right child of each node.

• If the tree is not ordered, then it does not make sense to talk of a left child and a right child.
Properties of Binary Trees

- A binary tree with $N$ internal nodes has $N+1$ external nodes.
- A binary tree with $N$ internal nodes has $2N$ edges (links).
- The height of a binary tree with $N$ internal nodes is at least $\lg N$ and at most $N$.
  - Height = $\lg N$ if all leaves are at the same level.
  - Height = $N$ if each internal node has one leaf child.
Defining Nodes for Binary Trees

typedef struct node *link;
struct node
{
  Item item;
  link left;
  link right;
};
Traversing a Binary Tree

- **Traversing** is the process of going through each node of a tree, and doing something with that node. Examples:
  - We can print the contents of the node.
  - We can change the contents of the node.
  - We can otherwise use the contents of the node in computing something.

- We have three choices about the order in which we visit nodes when we traverse a binary tree.
  - **Preorder**: we visit the node, then its left subtree, then its right subtree.
  - **Inorder**: we visit the left subtree, then the node, then the right subtree.
  - **Postorder**: we visit the left subtree, then the right subtree, then the node.
Examples

• In what order will the values of the nodes be printed if we print the tree by traversing it:
  – Preorder?
  – Inorder?
  – Postorder?
Examples

• In what order will the values of the nodes be printed if we print the tree by traversing it:
  – Preorder? 0, 1, 2, 6, 7.
  – Inorder? 1, 0, 6, 2, 7.
  – Postorder? 1, 6, 7, 2, 0.
Recursive Tree Traversal

```c
void traverse_preorder(link h) {
    if (h == NULL) return;
    do_something_with(h);
    traverse_preorder (h->l);
    traverse_preorder (h->r);
}

void traverse_inorder(link h) {
    if (h == NULL) return;
    traverse_inorder (h->l);
    do_something_with(h);
    traverse_inorder (h->r);
}

void traverse_postorder(link h) {
    if (h == NULL) return;
    traverse_postorder (h->l);
    traverse_postorder (h->r);
    do_something_with(h);
}
```
Recursive Examples

Counting the number of nodes in the tree:

```c
int count(link h)
{
    if (h == NULL) return 0;
    int c1 = count(h->left);
    int c2 = count(h->right);
    return c1 + c2 + 1;
}
```

Computing the height of the tree:

```c
int height(link h)
{
    if (h == NULL) return -1;
    int u = height(h->left);
    int v = height(h->right);
    if (u > v) return u+1;
    else return v+1;
}
```
Recursive Examples

```c
void printnode(char c, int h) {
    int i;
    for (i = 0; i < h; i++) printf(" ");
    printf("%c\n", c);
}

void show(link x, int h) {
    if (x == NULL) { printnode("*", h); return; }
    printnode(x->item, h);
    show(x->l, h+1);
    show(x->r, h+1);
}
```

Printing the contents of each node:
(assuming that the items in the nodes are characters)
Recursive functions are also frequently used to traverse graphs.

When traversing a tree, it is natural to start at the root.

When traversing a graph, we must specify the node with start from.

In the following examples we will assume that we represent graphs using adjacency lists.
typedef struct struct_graph * graph;

struct struct_graph
{
    int number_of_vertices;
    list * adjacencies;
};
Graph Traversal - Graph Search

• Overall, we will use the terms "graph traversal" and "graph search" almost interchangeably.

• However, there is a small difference:
  – "Traversal" implies we visit every node in the graph.
  – "Search" implies we visit nodes until we find something we are looking for.

• For example:
  – A node labeled "New York".
  – A node containing integer 2014.
Graph Search in General

• GraphSearch(graph, starting_node)
  – Initialize list to_visit to a list with starting_node as its only element.
  – While(to_visit is not empty):
    • Remove a node N from list to_visit.
    • "Visit" that node.
    • If that node was what we were looking for, break.
    • Add the children of that node to the end of list to_visit.

• The pseudocode is really a template.
• It does not specify what we really want to do.
• To fully specify an algorithm, we need to better define what each of the red lines.
Graph Search in General

• GraphSearch(graph, starting_node)
  – Initialize list to_visit to a list with starting_node as its only element.
  – While(to_visit is not empty):
    • Remove a node N from list to_visit.
    • "Visit" that node.
    • If that node was what we were looking for, break.
    • Add the children of that node to the end of list to_visit.

• Depending on what we specify in those lines, this template can produce a wide variety of applications:
  – Printing each node of the graph.
  – Driving directions.
  – The best move for a board game like chess.
  – A solution to a mathematical problem...
Specifying Graph Search Behavior

• **GraphSearch**(graph, starting_node)
  – Initialize list to_visit to a list with starting_node as its only element.
  – While(to_visit is not empty):
    • Remove a node N from list to_visit.
    • "Visit" that node.
    • If that node was what we were looking for, break.
    • Add the children of that node to the end of list to_visit.

• What do we do when visiting a node?
• Whatever we want. For example:
  – Print the contents of the node.
  – Use the contents in some computation (min, max, sum, ...).
  – See if the node has a value we care about ("New York", 2014, ...).
  – These are all reasonable topics for assignments/exams.
Specifying Graph Search Behavior

• GraphSearch(graph, starting_node)
  – Initialize list to_visit to a list with starting_node as its only element.
  – While(to_visit is not empty):
    • Remove a node N from list to_visit.
    • "Visit" that node.
    • If that node was what we were looking for, break.
    • Add the children of that node to the end of list to_visit.

• Inserting children of a node to the to_visit list:
• We have a choice: insert a child even if it already is included in that list, or not?
  – In some cases we should not. Example: ???
  – In some cases we should, but we may not see such cases in this course.
Specifying Graph Search Behavior

• **GraphSearch**, starting_node
  – Initialize list `to_visit` to a list with `starting_node` as its only element.
  – While `to_visit` is not empty:
    • Remove a node N from list `to_visit`.
    • "Visit" that node.
    • If that node was what we were looking for, break.
    • Add the children of that node to the end of list `to_visit`.

• Inserting children of a node to the `to_visit` list:

• We have a choice: insert a child even if it already is included in that list, or not?
  – In some cases we should not. Example: printing each node.
  – In some cases we should, but we may not see such cases in this course.
Specifying Graph Search Behavior

• GraphSearch(graph, starting_node)
  – Initialize list to_visit to a list with starting_node as its only element.
  – While(to_visit is not empty):
    • Remove a node N from list to_visit.
    • "Visit" that node.
    • If that node was what we were looking for, break.
    • Add the children of that node to the end of list to_visit.

• Most important question (for the purposes of this course):
  – Removing a node from list to_visit: **Which node**? The first, the last, some other one?

• The answer has profound implications for time complexity, space complexity, other issues you may see later or in other courses...
Depth-First Search

- **DepthFirstSearch**\( (\text{graph}, \text{starting\_node}) \)
  - Initialize list \textit{to\_visit} to a list with \textit{starting\_node} as its only element.
  - While (\textit{to\_visit} is not empty):
    - Remove the last node \textit{N} from list \textit{to\_visit}.
    - "Visit" that node.
    - If that node was what we were looking for, break.
    - Add the children of that node to the end of list \textit{to\_visit}.

- In depth-first search, the list of nodes to visit is treated as a LIFO (last-in, first-out) queue.

- **DepthFirstSearch**\( (\text{graph}, 5) \):
  - In what order does it visit nodes?
Depth-First Search

- **DepthFirstSearch(graph, starting_node)**
  - Initialize list `to_visit` to a list with `starting_node` as its only element.
  - While(`to_visit` is not empty):
    - Remove the last node `N` from list `to_visit`.
    - "Visit" that node.
    - If that node was what we were looking for, break.
    - Add the children of that node to the end of list `to_visit`.

- **DepthFirstSearch(graph, 5):**
- In what order does it visit nodes?
- The answer is not unique.
  - One possibility: 5, 4, 3, 7, 0, 1, 2, 6.
  - Another possibility: 5, 3, 4, 7, 0, 6, 1, 2.
  - Another possibility: 5, 0, 6, 4, 3, 7, 1, 2.
Depth-First Search

void depth_first(Graph g, int start)
{
    int * visited = malloc(sizeof(int) * g->number_of_vertices);
    int i;
    for (i = 0; i < g->number_of_vertices; i++) visited[i] = 0;
    depth_first_helper(g, start, visited);
}

void depth_first_helper (Graph g, int k, int * visited)
{  link t;
    do_something_with(k); // This is just a placeholder.
    visited[k] = 1;
    for (t = listFirst(g->adjacencies[k]); t != NULL; t = t->next)
        if (!visited[linkItem(t)]) depth_first_helper(g, linkItem(t), visited);
}
Breadth-First Search

- **BreadthFirstSearch***(graph, starting_node)*
  - Initialize list to_visit to a list with starting_node as its only element.
  - While(to_visit is not empty):
    - Remove the first node N from list to_visit.
    - "Visit" that node.
    - If that node was what we were looking for, break.
    - Add the children of that node to the end of list to_visit.

- In breadth-first search, the list of nodes to visit is treated as a LIFO (last-in, first-out) queue.
- **BreadthFirstSearch***(graph, 5):*  
  - In what order does it visit nodes?
Breadth-First Search

• BreadthFirstSearch(graph, starting_node)
  – Initialize list to_visit to a list with starting_node as its only element.
  – While(to_visit is not empty):
    • Remove the first node N from list to_visit.
    • "Visit" that node.
    • If that node was what we were looking for, break.
    • Add the children of that node to the end of list to_visit.

• BreadthFirstSearch(graph, 5):

• In what order does it visit nodes?
• The answer is not unique.
  – One possibility: 5, 4, 3, 0, 7, 1, 2, 6.
  – Another possibility: 5, 3, 4, 0, 7, 6, 1, 2.
  – Another possibility: 5, 0, 4, 3, 6, 1, 2, 7.
void breadth_first(Graph g, int k) {
    int i;  link t;
    int * visited = malloc(sizeof(int) * g->number_of_vertices);
    for (i = 0; i < g->number_of_vertices; i++) visited[i] = 0;
    QUEUEInit(V);  QUEUEput(k);
    while (!QUEUEempty())
        if (visited[k = QUEUEget()] == 0)
            {  
                doSomethingWith(k);  // This is just a placeholder.
                visited[k] = 1;
                for (t = g->adjacencies[k]; t != NULL; t = t->next)
                    if (visited[linkItem(t)] == 0) QUEUEput(linkItem(t));
            }
}
Note

• The previous examples should be treated as very detailed C-like pseudocode, not as ready-to-run code.

• We have seen several different implementations of graphs, lists, queues.

• To make the code actually work, you will need to make sure it complies with specific implementations.