Outline

Continue Week 05 stuffs (Graph DS Applications)

Two algorithms to traverse a graph

- Depth First Search (DFS) and Breadth First Search (BFS)
- Plus some of their interesting applications

visualgo.net/dfsbf.html

Reference: Mostly from CP3 Section 4.2
- Not all sections in CP3 chapter 4 are used in CS2010!
  - Some are quite advanced :O
SOME GRAPH DATA STRUCTURE APPLICATIONS
So, what can we do so far? (1)

With just graph DS, not much that we can do...
But here are some:

• Counting \( V \) (the number of vertices)
  – Very trivial for both AdjMatrix and AdjList: \( V = \text{number of rows}! \)
  – Sometimes this number is stored in separate variable so that we do not have to re-compute every time, that is, \( O(1) \), especially if the graph never changes
So, what can we do so far? (2)

- Enumerating neighbors of a vertex $v$
  - $O(V)$ for AdjMatrix: scan $\text{AdjMatrix}[v][j], \forall j \in [0..V-1]$
  - $O(k)$ for AdjList, scan $\text{AdjList}[v]$
    - $k$ is the number of neighbors of vertex $v$ (output-sensitive algorithm)
  - This is an important difference between AdjMatrix versus AdjList
    - It affects the performance of many graph algorithms. Remember this!

\begin{itemize}
  \item 
\end{itemize}
So, what can we do so far? (3)

- Counting $E$ (the number of edges)
  - $O(V^2)$ for AdjMatrix: count non zero entries in AdjMatrix
  - $O(V + E)$ for AdjList: sum the length of all V lists
  - Sometimes this number is stored in separate variable so that we do not have to re-compute every time, i.e. $O(1)$, especially if the graph never changes
So, what can we do so far? (4)

- Checking the existence of edge($u, v$)
  - $O(1)$ for AdjMatrix: see if $\text{AdjMatrix}[u][v]$ is non zero
  - $O(k)$ for AdjList: see if $\text{AdjList}[u]$ contains $v$

- There are a few others, but let’s reserve them for PSes or even for test questions 😊
Trade-Off

**Adjacency Matrix**

**Pros:**
- Existence of edge i-j can be found in $O(1)$
- Good for dense graph/
  Floyd Warshall’s (Lecture 12)

**Cons:**
- $O(V)$ to enumerate neighbors of a vertex
- $O(V^2)$ space

**Adjacency List**

**Pros:**
- $O(k)$ to enumerate k neighbors of a vertex
- Good for sparse graph/Dijkstra’s/
  DFS/BFS, $O(V+E)$ space

**Cons:**
- $O(k)$ to check the existence of edge i-j
- A small overhead in maintaining the list (for sparse graph)
VisuAlgo Graph DS Exploration (1)

Click each of the sample graphs one by one and verify the content of the corresponding Adjacency Matrix, Adjacency List, and Edge List.
VisuAlgo Graph DS Exploration (2)

Now, use your mouse over the currently displayed graph and start drawing some new vertices and/or edges and see the updates in AdjMatrix/AdjList/EdgeList structures.
GRAPH TRAVERSAL ALGORITHMS
Review – Binary Tree Traversal

In a binary tree, there are three standard traversal:

- **Preorder**

  \[
  \text{pre(u)} \quad \text{visit(u)}; \quad \text{pre(u->left)}; \quad \text{pre(u->right)};
  \]

- **Inorder**

  \[
  \text{in(u)} \quad \text{in(u->left)}; \quad \text{visit(u)}; \quad \text{in(u->right)};
  \]

- **Postorder**

  \[
  \text{post(u)} \quad \text{post(u->left)}; \quad \text{post(u->right)}; \quad \text{visit(u)};
  \]

  (Note: “level order” is just BFS which we will see next)

We start binary tree traversal from root:

- \(\text{pre(root)}/\text{in(root)}/\text{post(root)}\)
  - \(\text{pre} = 0, 1, 2, 3, 4\)
  - \(\text{in} = 1, 0, 3, 2, 4\)
  - \(\text{post} = 1, 3, 4, 2, 0\)
What is the **Post**Order Traversal of this Binary Tree?

1. 0 1 2 3 4  
2. 0 1 3 2 4  
3. 3 4 1 2 0  
4. 3 1 4 2 0
Traversing a Graph (1)

Two ingredients are needed for a traversal:
1. The start
2. The movement

Defining the start (“source”)

• In tree, we *normally* start from root
  – Note: Not all tree are rooted though!
    • In that case, we have to select one vertex as the “source”, see below

• In general graph, we do not have the notion of root
  – Instead, we start from a distinguished vertex
    • We call this vertex as the “source”
Traversing a Graph (2)

Defining the movement:

• In (binary) tree, we only have (at most) two choices:
  – Go to the left subtree or to the right subtree

• In general graph, we can have more choices:
  – If vertex \( u \) and vertex \( v \) are adjacent/connected with edge \((u, v)\);
    and we are now in vertex \( u \);
    then we can also go to vertex \( v \) by traversing that edge \((u, v)\)

• In (binary) tree, there is no cycle

• In general graph, we may have (trivial/non trivial) cycles
  – We need a way to avoid revisiting \( u \rightarrow v \rightarrow u \rightarrow u \rightarrow \ldots \) indefinitely

Solution: BFS and DFS 😊
Breadth First Search (BFS) – Ideas

• Start from s

• If a vertex v is reachable from s, then all neighbors of v will also be reachable from s (recursive definition)

• BFS visits vertices of G in breadth-first manner (when viewed from source vertex s)
  
  – Q: How to maintain such order?
  
  • A: Use queue Q, initially, it contains only s

  – Q: How to differentiate visited vs unvisited vertices (to avoid cycle)?

  • A: 1D array/Vector visited of size V, 
    visited[v] = 0 initially, and visited[v] = 1 when v is visited

  – Q: How to memorize the path?

  • A: 1D array/Vector p of size V, 
    p[v] denotes the predecessor (or parent) of v
BFS Pseudo Code

for all $v$ in $V$
    $\text{visited}[v] \leftarrow 0$
    $p[v] \leftarrow -1$
$Q \leftarrow \{s\}$  // start from $s$
$\text{visited}[s] \leftarrow 1$

while $Q$ is not empty
    $u \leftarrow Q$.dequeue()
    for all $v$ adjacent to $u$  // order of neighbor
        if $\text{visited}[v] = 0$  // influences BFS
            $\text{visited}[v] \leftarrow \text{true}$  // visitation sequence
            $p[v] \leftarrow u$
            $Q$.enqueue($v$)

// after BFS stops, we can use info stored in $\text{visited}/p$
Graph Traversal: BFS(s)

Ask VisuAlgo to perform various Breadth-First Search operations on the sample Graph (CP3 4.3, Undirected)

In the screen shot below, we show the start of **BFS(5)**
BFS Analysis

for all v in V
    visited[v] ← 0
    p[v] ← -1
Q ← {s} // start from s
visited[s] ← 1

while Q is not empty
    u ← Q.dequeue()
    for all v adjacent to u // order of neighbor
        if visited[v] = 0 // influences BFS
            visited[v] ← true // visitation sequence
            p[v] ← u
            Q.enqueue(v)

// we can then use information stored in visited/p

Time Complexity: O(V+E)
• Each vertex is only in the queue once ~ O(V)
• Every time a vertex is dequeued, all its k neighbors are scanned; After all vertices are dequeued, all E edges are examined ~ O(E)
→ assuming that we use Adjacency List!
• Overall: O(V+E)
Depth First Search (DFS) – Ideas

• Start from \( s \)
• If a vertex \( v \) is reachable from \( s \), then all neighbors of \( v \) will also be reachable from \( s \) (recursive definition)
• DFS visits vertices of \( G \) in *depth-first* manner (when viewed from source vertex \( s \))
  – Q: How to maintain such order?
    • A: Stack \( S \), but we will simply use recursion (an implicit stack)
  – Q: How to differentiate visited vs unvisited vertices (to avoid cycle)?
    • A: 1D array/Vector \( \text{visited} \) of size \( V \),
      \( \text{visited}[v] = 0 \) initially, and \( \text{visited}[v] = 1 \) when \( v \) is visited
  – Q: How to memorize the path?
    • A: 1D array/Vector \( p \) of size \( V \),
      \( p[v] \) denotes the predecessor (or parent) of \( v \)
DFS Pseudo Code

DFSrec(u)

visited[u] ← 1 // to avoid cycle
for all v adjacent to u // order of neighbor
    if visited[v] = 0 // influences DFS
        p[v] ← u // visitation sequence
        DFSrec(v) // recursive ( implicit stack)

// in the main method
for all v in V
    visited[v] ← 0
    p[v] ← -1
DFSrec(s) // start the
recursive call from s
Graph Traversal: DFS(s)

Ask VisuAlgo to perform various Breadth-First Search operations on the sample Graph (CP3 4.1, Undirected)

In the screen shot below, we show the start of $\text{DFS}(0)$.
DFS Analysis

DFSrec(u)
    visited[u] ← 1 // to avoid cycle
    for all v adjacent to u // order of neighbor
        if visited[v] = 0 // influences DFS
            p[v] ← u // visitation sequence
            DFSrec(v) // recursive (implicit stack)

// in the main method
for all v in V
    visited[v] ← 0
    p[v] ← -1
DFSrec(s) // start the recursive call from s

Time Complexity: O(V+E)
• Each vertex is only visited once O(V), then it is flagged to avoid cycle
• Every time a vertex is visited, all its k neighbors are scanned; Thus after all vertices are visited, we have examined all E edges ~ O(E) → assuming that we use Adjacency List!
• Overall: O(V+E)
// iterative version (will produce reversed output)
Output "(Reversed) Path:"
i ← t // start from end of path: suppose vertex t
while i != s
    Output i
    i ← p[i] // go back to predecessor of i
Output s

// try it on this array p, t = 4
// p = {-1, 0, 1, 2, 3, -1, -1, -1}
Path Reconstruction Algorithm (2)

```plaintext
void backtrack(u)
    if (u == -1) // recall: predecessor of s is -1
        stop
    backtrack(p[u]) // go back to predecessor of u
    Output u // recursion like this reverses the order

// in main method
// recursive version (normal path)
Output "Path:'
backtrack(t); // start from end of path (vertex t)
// try it on this array p, t = 4
// p = {-1, 0, 1, 2, 3, -1, -1, -1}
```
SOME GRAPH TRAVERSAL APPLICATIONS
What can we do with BFS/DFS? (1)

Several stuffs, let’s see **some of them:**

- **Reachability test**
  - Test whether vertex \( v \) is reachable from vertex \( u \)?
  - Start BFS/DFS from \( s = u \)
  - If \( \text{visited}[v] = 1 \) after BFS/DFS terminates,
    then \( v \) is **reachable** from \( u \); otherwise, \( v \) is **not reachable** from \( u \)

\[
\text{BFS}(u) \quad // \quad \text{DFSrec}(u)
\]

```plaintext
if visited[v] == 1
    Output "Yes"
else
    Output "No"
```

![Graph Diagram](image)
Reachability Test

Ask VisuAlgo to perform various DFS (or BFS) operations on the sample Graph (CP3 4.1, Undirected)

Below, we show vertices that are reachable from vertex 0
What can we do with BFS/DFS? (2)

- Identifying component(s)
  - Component is sub graph in which any 2 vertices are connected to each other by at least one path, and is connected to no additional vertices
  - With BFS/DFS, we can identify/label/count components in graph G
  - Solution:

\[
\text{CC} \leftarrow 0 \\
\text{for all } v \text{ in } V \\
\text{visited}[v] \leftarrow 0 \\
\text{for all } v \text{ in } V // O(V)\
\text{if visited}[v] == 0 \\
\text{CC} \leftarrow \text{CC} + 1 \\
\text{DFSrec}(v) // O(V+E)\
// BFS from v
// is also OK
\]
Reachability Test

Ask VisuAlgo to perform various DFS (or BFS) operations on the sample Graph (CP3 4.1, Undirected)

Call **DFS(0)/BFS(0), DFS(5)/BFS(5), then DFS(6)/BFS(6)**
What is the time complexity for “counting connected component”?

1. Hm... you can call $O(V+E)$
   DFS/BFS up to $V$ times...
   I think it is $O(V^2(V+E)) = O(V^2 + VE)$

2. It is $O(V+E)$...

3. Maybe some other time complexity, it is $O(______)$
What can we do with BFS/DFS? (3)

- Topological Sort
  - Topological sort of a DAG is a linear ordering of its vertices in which each vertex comes before all vertices to which it has outbound edges
  - Every DAG has one *or more* topological sorts
  - One of the main purpose of finding topological sort: for Dynamic Programming (DP) on DAG (will be discussed a few weeks later...)

Reminder to myself: slow down here
What can we do with BFS/DFS? (4)

• Topological Sort
  – If the graph is a DAG, then simply running **DFS** on it (and at the same time record the vertices in “post-order” manner) will give us one valid topological order
    • “Post-order” = process vertex \( u \) after all children of \( u \) have been visited
  – See pseudo code in the next slide
DFS for TopoSort – Pseudo Code

Simply look at the codes in red/underlined

DFSrec(u)

visited[u] ← 1 // to avoid cycle
for all v adjacent to u // order of neighbor
    if visited[v] = 0 // influences DFS
        p[v] ← u // visitation sequence
        DFSrec(v) // recursive (implicit stack)
append u to the back of toposort // "post-order"

// in the main method
for all v in V
    visited[v] ← 0
    p[v] ← -1
    clear toposort
for all v in V
    if visited[v] == 0
        DFSrec(s) // start the recursive call from s
reverser toposort and output it
What can we do with BFS/DFS? (5)

- Topological Sort
  - Suppose we have visited all neighbors of 0 recursively with DFS
  - toposort list = [list of vertices reachable from 0] - vertex 0
    - Suppose we have visited all neighbors of 1 recursively with DFS
      - toposort list = [[list of vertices reachable from 1] - vertex 1] - vertex 0
      - and so on...
  - We will eventually have = [4, 3, 5, 2, 1, 0, 6, 7]
  - Reversing it, we will have = [7, 6, 0, 1, 2, 5, 3, 4]
Topological Sort

Ask VisuAlgo to perform Topo Sort (DFS) operation on the sample Graph (CP3 4.4, Directed)

Below, we show partial execution of the DFS variant
Trade-Off

**O(V+E) DFS**

- **Pros:**
  - Slightly easier? to code (this one depends)
  - Use less memory
  - Has some extra features (not in CS2010 syllabus but useful for your PS3)

- **Cons:**
  - Cannot solve SSSP on unweighted graphs

**O(V+E) BFS**

- **Pros:**
  - Can solve SSSP on unweighted graphs (revisited in latter lectures)

- **Cons:**
  - Slightly longer? to code (this one depends)
  - Use more memory (especially for the queue)
Hospital Tour Problem (PS3)

Given a layout of a hospital...

- Determine which room(s) is/are the ‘important room(s)’
- Among those room(s), pick one with the lowest rating score
Online Quiz 1 (Tomorrow)
(Thu, 17 Sep 2015, during your lab session)

Try OQ1 Preview (test ID: 31) if you have not done so

http://visualgo.net/test.html

You can always challenge yourself more with this:

http://visualgo.net/training.html?diff=Hard&n=20&tl=40&module=heap,bst,avl,ufds,bitmask,graphds
Written Quiz 1 (This Saturday)
(Sat, 19 Sep 2015, LT19, SR@LT19, TR9)

3 Sections only, 90 minutes:

- Most basic questions about Binary Heap/BST/AVL/UFDS/
  Bitmask/Graph Data Structures have been automated in the
  Online Quiz 1
- So this one is definitely (much) harder than Online Quiz 1…
  – Disclaimer: Doing well in OQ1 may not correlate with doing well in WQ1

Material:
- Lecture 1-2-3-4-5, Tutorial 1-2-3-4, Lab Demos 1-2-3-4, PS1-2
  – IMPORTANT: UFDS, bitmask, and Graph DSes are included
  – Lecture 06 (DFS/BFS) is excluded
- CP3: page 36-54 😊
Summary

In this lecture, we have looked at:

• Some applications of Graph Data Structures
  – Continuation from Lecture 05

• Graph Traversal Algorithms: Start + Movement
  – Breadth-First Search: uses queue, breadth-first
  – Depth-First Search: uses stack/recursion, depth-first
  – Both BFS/DFS uses “flag” technique to avoid cycling
  – Both BFS/DFS generates BFS/DFS “Spanning Tree”
  – Some applications: Reachability, CC, Toposort