GRAPHS

cs2420 | Introduction to Algorithms and Data Structures | Spring 2016
administrivia...
-assignment 8 due Wednesday

-midterm 2 is in two weeks
  -will cover material from linked lists through next week’s topics
  -no practice exam this time
  -format will be similar to midterm 1
    -concepts from lectures, labs, and assignments are all fair game!

-Prof. Meyer’s office hours for this week are moved to Friday 10am-12pm
Why Big O
Complexity Analysis, Big O – what’s it for?

- What do we do with data structures/algorithms: Insert, Delete, Find

- Myth: Computers are so fast, it doesn’t matter.

- Intuition: What happens if we DOUBLE the amount of data?

- What is effect on find?
  - Bad Data Structures: Quadruple time
  - Regular Data Structures: Double time
  - Good Data Structures: Add constant to time
  - Great Data Structures: No change
last time...
binary search trees (BSTs)
Why Binary Trees?

-CS is about the why.
  -Are you internalizing this, or are you just memorizing and regurgitating?

-Why BST?
  -Search time as fast as an ordered array
  -Insertion time as fast as a linked list
- a binary search tree is a binary tree with a restriction on the ordering of nodes
  - all items in the left subtree of a node are less than the item in the node
  - all items in the right subtree of a node are greater than or equal to the item in the node

- BSTs allow for fast searching of nodes
Is this a BST?
insertion

Goto: https://goo.gl/forms/gsaUZBmJkr7yDDYG2

Insert unid, answer Question 2: BST Questions, Submit
insertion on balanced tree

-Bonus Question: Define balanced tree

-worst case: $O(\log N)$

-best case: $O(\log N)$
insertion on unbalanced tree

- worst case: \(O(N)\)

- best case: \(O(1)\)
searching a balanced tree

- average case: $O(\log N)$
  - inserted in random order

- worst case: $O(N)$
  - inserted in ascending or descending order

- best case: constant
deletion
since we must maintain the properties of a tree structure, deletion is more complicated than with an array or linked-list

there are three different cases:

1. deleting a leaf node
2. deleting a node with one child subtree
3. deleting a node with two children subtrees

first step of deletion is to find the node to delete

-just a regular BST search
-BUT, stop at the parent of the node to be deleted
How to Delete:
boar
cat
dog
How to Delete:
boar
cat
dog

- alligator
- bison
- cat
- dog
- elephant
- monkey
- newt
- snake
How to Delete:

- boar
- cat
- dog
- alligator
- bison
- monkey
- elephant
- snake
- newt
How to Delete:
- boar
- cat
- dog
- alligator
- bison
- elephant
- monkey
- snake
- newt
deletion performance

what is the cost of deleting a node from a BST?

- first, find the node we want to delete: $O(\log N)$

- cost of:

  - case 1 (delete leaf):
    - set a single reference to null: $O(c)$
  - case 2 (delete node with 1 child):
    - bypass a reference: $O(c)$
  - case 3 (delete node with 2 children):
    - find the successor: $O(\log N)$
    - delete the duplicate successor: $O(c)$

- [http://www.algolist.net/Data_structures/Binary_search_tree/Removal](http://www.algolist.net/Data_structures/Binary_search_tree/Removal)
Answer Question 3 on Google
today... graphs
Why Trees? Why Graphs?

- Why do we use trees in computer science?

- What information does the following Graphs make explicit?
nodes are cities, edges are flights
-graphs
-paths
-depth-first search
-breadth-first search
graphs
-you already know what graphs are, they are trees
-Okay, Trees are a *subset* of graphs

-a graph is a set of nodes connected by edges
- an edge is just a link between two nodes
- nodes don’t have a parent-child relationship
- links can be bi-directional

-graphs are used *EXTENSIVELY* throughout CS
some definitions

- **undirected graph**
- **directed graph**
- **edge weighted**
- **unconnected**
- **node degrees**

- **cycle**
- **acyclic graph**
- **connected acyclic graph, a.k.a. a tree**
- **rooted tree or hierarchy**
- **node depths**

- **0.1**
- **5.0**
- **0.3**
- **2**
- **3**
- **2**
- **1**
- **2**
- **2**
- **2**
- **1**
- **1**
- **0**
what makes this a graph and not a tree?
Graph Implementation Strategy

- graphs have no root; must store all nodes

```
class Graph<E> {
    List<Node> nodes;
    ...
}
```

- implementation is more general than a tree

```
class Node{
    E Data;
    List<Node> neighbors;
    ...
}
```

- the order in which neighbors appear in the list is unspecified
  - a different order still make the same graph!
paths
- A path is a sequence of nodes with a start-point and an end-point such that the end-point can be reached through a series of nodes from the start-point.

- In this example, there is a path from SLC to DFW: SLC → IAD → ATL → DFW.

- There is not a path from DFW to SLC.
Shortest Path from Las Vegas to El Paso? Google Question
Do you fly the shortest distance?

- Of course not.

- What do you minimize?
pathfinding

- there may be more than one path from one node to another

- we are often interested in the *path length*

- finding the shortest (or cheapest) path between two nodes is a common graph operation
  - Simplifying Assumption for “toy” examples → all edges cost one
cycles

-a cycle in a graph is a path from a node back to itself
  -B — E — D — B

-while traversing a graph, special care must be taken to avoid cycles, otherwise what?

-Bonus Question: can trees have cycles?
-any problem with a starting state, a goal state, and options as to which direction to take for each step can be represented with a graph

-and solved with pathfinding!

-Sample Problems?
  -Google Map Directions
  -7 degrees of Kevin Bacon
  -Maze path finder
  -Assigning peer reviews
  -Chess playing computer
example

-in games, moving a character around a space

-character finds the shortest path from its current location to the destination
  -not always a straight line

-terrain is represented as a graph
  -every non-obstacle spot on the terrain is a node
  -nodes are connected to adjacent nodes

-navigating a maze...
- depth-first search (just like a tree) — DFS

- breadth-first search — BFS

- if there exists a path from one node to another these algorithms will find it
  - the nodes on this path are the steps to take to get from point A to point B

- if multiple such paths exist, the algorithms may find different ones
we want to find a path from A to C
depth-first search
- look at the first edge going out of the start node
- recursively search from the new node
- upon returning, take the next edge
- if no more edges, return

-bonus question: why is the above incomplete?

-when visiting a node, mark it as visited so we don’t get stuck in a cycle
  - skip already visited nodes during traversal

-for each node visited, save a reference to the node where we came from to reconstruct the path
we want to find a path from A to C

SO... start from A, traverse its first edge, save where we came from, and recurse

A.visited = true
B.cameFrom = A
traverse the first unvisited node in the edge list recursively, save where we came from

B.

\begin{itemize}
  \item B.visited = true
  \item E.cameFrom = B
\end{itemize}
traverse the first unvisited node in the edge list recursively, save where we came from

E.visited = true

look at the first edge; node A has already been visited, so skip
Look at next edge; C has not been visited yet

C.cameFrom = E
node C is our goal. we are done!

C.visited = true

follow each node’s cameFrom to reconstruct the path

C.cameFrom = E, E.cameFrom = B, B.cameFrom = A

path: A — B — E — C
is there a better (shorter) path from A to C?

what determines which path DFS finds?

DFS is not guaranteed to find the shortest path, just a path.
DFS(Node curr, Node goal) {
    curr.visited = true

    if(curr.equals(goal))
        return

    for(Node next : curr.neighbors)
        if(!next.visited)
            {
            next.cameFrom = curr
            DFS(next, goal)
            }
}

// path is now saved in nodes’ .cameFrom
breadth-first search
-instead of visiting deeper nodes first, visit shallower nodes first
  -visit nodes closest to the start point first, gradually get further away

-create an empty queue
-put the starting node in the queue
-while the queue is not empty
  -dequeue the current node
  -for each unvisited neighbor of the current node
    -mark the neighbor as visited
    -put the neighbor into the queue

-notice it is not recursive… it just runs until the queue is empty!
we want to find a path from $A$ to $C$

mark and enqueue the start node $A$

$A$.visited = true
Dequeue the first node in the queue (**A**)

mark and enqueue **A**’s unvisited neighbors

\[
\begin{align*}
\text{B} & . \text{cameFrom} = \text{A} \\
\text{D} & . \text{cameFrom} = \text{A} \\
\text{B} & . \text{visited} = \text{true} \\
\text{D} & . \text{visited} = \text{true}
\end{align*}
\]

**queue:** [B, D]
Dequeue the first node in the queue (B)

mark and enqueue B’s unvisited neighbors

E.cameFrom = B
E.visited = true
dequeue the first node in the queue (D)

mark and enqueue D’s unvisited neighbors

C.cameFrom = D
C.visited = true
Deque the first node in the queue (E)

mark and enqueue E’s unvisited neighbors

(no unvisited neighbors!)
dequeue the first node in the queue (C)

C is the goal! reconstruct the path with `cameFrom` references

C.cameFrom = D,  
D.cameFrom = A

path: A — D — C
is this the shortest path?  
path: A → D → C

BFS visits nodes closest to the start-point first

therefore, the first path found is the shortest path (closest to the start node)
https://visualgo.net/dfs}\textsc{bfs}
BFS(Node start, Node goal)
{
    start.visited = true
    Q.enqueue(start)

    while(!Q.empty())
    {
        Node curr = Q.dequeue()
        if(curr.equals(goal))
            return

        for(Node next : curr.neighbors)
        {
            if(!next.visited)
            {
                next.visited = true
                next.cameFrom = curr
                Q.enqueue(next)
            }
        }
    }
}
Google Question:
what path will BFS find from B to C?

A) B E C
B) B E A D C
C) B E D C
D) none
what path will DFS find from A to D?

A) A B E D
B) A D
C) none
D) this is a trick question
what path will DFS find from A to C?
A) A B E C
B) A D C
C) A E C
D) A C
E) none
F) this is a trick question
what is true of DFS, searching from a start node to a goal node?

A) if a path exists, it will find it
B) it is guaranteed to find the shortest path
C) it is guaranteed to not find the shortest path
D) it must be careful about cycles
E) a, b, and d
F) a, c, and d
G) a and d
what is true of BFS, searching from a start node to a goal node?
A) if a path exists, it will find it
B) it is guaranteed to find the shortest path
C) it is guaranteed to not find the shortest path
D) it must be careful about cycles
E) a, b, and d
F) a, c, and d
G) a and d
topological sort

“loose” sorting based on dependencies
What order must courses be taken in?

CS1410

CS2100  CS4150  CS3500  CS3810  CS3200

CS2420

CS3100  CS3500  CS3505  CS4400

MATH2250

CS3810

CS4500
Topological Sort $\rightarrow$ find order
CS courses must be taken in

- $1410, 2420, 3500, 4150, 2100, 3100$
- or
- $1410, 2100, 2420, 3100, 4150, 3500$
- or
- $1410, 2100, 3100, 4150, 2420, 3500$
- or
- $1410, 2100, 3500, 4150, 3100, 2420$
- Topological Sort requires keeping track of node “indegree”
  - the indegree of a node is the number of edges it has incoming
  - this can be saved as part of the Node class, and can be easily computed as the graph is constructed
  - any time a node adds another node as a neighbor, increase the neighbor’s indegree
<table>
<thead>
<tr>
<th>Course Code</th>
<th>Course Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS1410</td>
<td></td>
</tr>
<tr>
<td>CS2100</td>
<td></td>
</tr>
<tr>
<td>CS2420</td>
<td></td>
</tr>
<tr>
<td>CS3100</td>
<td></td>
</tr>
<tr>
<td>CS3200</td>
<td></td>
</tr>
<tr>
<td>CS4150</td>
<td></td>
</tr>
<tr>
<td>CS3500</td>
<td></td>
</tr>
<tr>
<td>CS3505</td>
<td></td>
</tr>
<tr>
<td>CS3810</td>
<td></td>
</tr>
<tr>
<td>CS4400</td>
<td></td>
</tr>
<tr>
<td>CS4500</td>
<td></td>
</tr>
<tr>
<td>MATH2250</td>
<td></td>
</tr>
</tbody>
</table>
topological sort

- consider a graph with no cycles

- a topological sort orders nodes such that...
  - if there is a path from node A to node B, then A appears before B in the sorted order

example: scheduling tasks
- represent the tasks in a graph
- if task A must be completed before task B, then A has an edge to B
One Algorithm

1. **step through each node in the graph**
   - if any node has indegree 0, add it to a queue

2. **while the queue is not empty**
   - dequeue the first node in the queue and add to the sorted list
   - visit that node’s neighbors and decrease their indegree by 1
   - if a neighbor’s new indegree is 0, add it to the queue
queue:
sorted list:
enque any nodes with indegree 0

queue: [CS1410, MATH2250]

sorted list:
while queue not empty:
1. dequeue node, add it to the sorted list

queue: MATH2250
sorted list: CS1410
while queue not empty:
1. dequeue node, add it to the sorted list
2. decrease neighbors’ indegree

queue: MATH2250
sorted list: CS1410
while queue not empty:
1. dequeue node, add it to the sorted list
2. decrease neighbors’ indegree
3. if neighbors’ new indegree is 0, add it to the queue

queue: MATH2250, CS2100, CS2420

sorted list: CS1410
while queue not empty:
1. dequeue node, add it to the sorted list
2. decrease neighbors’ indegree
3. if neighbors’ new indegree is 0, add it to the queue

queue: CS2100, CS2420
sorted list: CS1410, MATH2250
while queue not empty:
1. dequeue node, add it to the sorted list
2. decrease neighbors’ indegree
3. if neighbors’ new indegree is 0, add it to the queue

queue: CS2420

sorted list: CS1410 MATH2250 CS2100
while queue not empty:
1. dequeue node, add it to the sorted list
2. decrease neighbors’ indegree
3. if neighbors’ new indegree is 0, add it to the queue

queue:

sorted list: [CS1410, MATH2250, CS2100, CS2420]
while queue not empty:
1. dequeue node, add it to the sorted list
2. decrease neighbors’ indegree
3. if neighbors’ new indegree is 0, add it to the queue

queue:

sorted list: [CS1410, MATH2250, CS2100, CS2420]
while queue not empty:
1. dequeue node, add it to the sorted list
2. decrease neighbors’ indegree
3. if neighbors’ new indegree is 0, add it to the queue

queue:  
CS3100  CS4150  CS3500  CS3810  CS3200

sorted list:  
CS1410  MATH2250  CS2100  CS2420
while queue not empty:
1. dequeue node, add it to the sorted list
2. decrease neighbors’ indegree
3. if neighbors’ new indegree is 0, add it to the queue

queue: CS4150 CS3500 CS3810 CS3200
sorted list: CS1410 MATH2250 CS2100 CS2420 CS3100
while queue not empty:
1. dequeue node, add it to the sorted list
2. decrease neighbors’ indegree
3. if neighbors’ new indegree is 0, add it to the queue

queue: CS3500 CS3810 CS3200

sorted list: CS1410 MATH2250 CS2100 CS2420 CS3100 CS4150
while queue not empty:
1. dequeue node, add it to the sorted list
2. decrease neighbors' indegree
3. if neighbors' new indegree is 0, add it to the queue

queue: CS3810 CS3200 CS3505
sorted list: CS1410 ... CS4150 CS3500
while queue not empty:
1. dequeue node, add it to the sorted list
2. decrease neighbors’ indegree
3. if neighbors’ new indegree is 0, add it to the queue

queue: [CS3505, CS4400]
sorted list: [CS1410, ... , CS4150, CS3500, CS3810, CS3200]
while queue not empty:
1. dequeue node, add it to the sorted list
2. decrease neighbors’ indegree
3. if neighbors’ new indegree is 0, add it to the queue

queue: CS4400, CS4500

sorted list: CS1410, CS4150, CS3500, CS3810, CS3200, CS3505
while queue not empty:
1. dequeue node, add it to the sorted list
2. decrease neighbors’ indegree
3. if neighbors’ new indegree is 0, add it to the queue
while queue not empty:
1. dequeue node, add it to the sorted list
2. decrease neighbors’ indegree
3. if neighbors’ new indegree is 0, add it to the queue
sorted list: CS1410, MATH2250, CS2100, CS2420, CS3100, CS4150, CS3500, CS3810, CS3200, CS3505, CS4400, CS4500
which of the following is a valid topological ordering?

A) A D C E B
B) C D E B A
C) A B E D C
D) A B C D E
next time...
-reading
  - chapter 14 in book

-homework
  - assignment 8 due Wednesday