CS 180: Lecture 4
Tuesday, Week 2
10/10/17

Complexity

- When we consider time complexity, we want to assume n to be very large.
- If we compare algorithms with $O(n), O(2n), O(\frac{n}{2})$, we say they run about the same speed
    - Same with something like $O(2n + 500 + log(n))$, etc...
- We only really care about the **highest order of complexity**

Order: a function $T(n) = O(f(n))$ if ∃ constants n_0, c such that ∀ $n \geq n_0$, we have $T(n) \leq cf(n)$

Can think of order as an **upper bound**.

Examples

- $n^2 = O(18n^2 + 10) \implies$ True
- $18n^2 + 10 = O(n^2) \implies$ True
- $n^2 = O(4n^2 + log(n) + \frac{n^3}{10}) \implies$ True
- $n^3 = O(n^2) \implies$ False

Omega Notation: a funtion $T(n) = \Omega(f(n))$ if ∃ constants n_0, c such that $\forall n \geq n_0$, we have $T(n) \geq cf(n)$

Can think of Omega notation as a **lower bound**

What is the **difference in application** between Order and Omega notation

- Order primarily used to describe **algorithms**
- Omega primarily used to describe **problems**

If a problem/algorithm is $O(n_1)$ and $\Omega(n_2)$ such that n_1 = n_2, then we say the problem is $\Theta(n)$

Graphs/Graph Algorithms

Graphs and their Properties

Graph - a graph is defined by a set of verticies (nodes) and edges (links). we write this formally by saying $G = (V, E)$ with, for example, $V = \{a, b, c, d\}$, $E = \{(b, c), (a, b), (a, c)(a, d)\}$

Graphs can have direction.

- A **directed** graph travels in a certain direction along its edges
- An **undirected** graph has no directions between its verticies

Graphs can have weighted or unweighted edges

- A **weighted** graph means that edges have different weights, one edge has a larger value than another
- An **unweighted** means that all edges mean the same thing

Graphs can be connected/disconnected

- If we can start at one vertex and travel along edges to touch all other vertex, the graph is **connected**
- If, when we start at one vertex, we cannot touch every other vertex by traversing edges, the graph is **disconnected**
    - Make sure to consider wether or not the graph is directed in the context of the problem

Eulerian Graphs

Problem: We want to see wether or not we can draw a given shape without picking up our pencil.

Rules:

- We must go over every edge and touch every vertex
- We cannot go over an edge twice

We are able to do this if and only if $\exists$ 2 or less vertexes with an odd number of edges attatched to it. Graphs that meet this quality are called **Eulerian graphs**.

Graph Traversal

There are different methods used to traverse graphs to search for something.

- **Breadth First Search** - search within neighborhood of starting point
    - Starting from a vertex, v_0
    - Examine all of the vertecies connected to v_0 first.
    - If we don't find what we need, then consider each of v_0's neighbors, lets call each one v_n
    - For each neighbor v_n look at their immediate neighbors that haven't been checked
    - etc...
- Properties of BFS
    - We can make a BFS tree starting with our initial point at the top, and traveling down one level that represents each of the neighbors we searched at that iteration
    - *Property*: When a vertex is on level i of the BFS tree, then the shortest path between our original vertex v and this new point is i.
- **Depth First Search** - search completely down a path from a starting point
    - Start from vertex v_0

- Pick a vertex connected to it, lets say v_1, and test that
- Make v_1 our new temporary vertex
- If v_1 has any unexplored neighbors, lets say v_2
  - Test that vertex, make v_2 new temporary vertex
- else v_1 has all neighbors explored
  - backtrack to the edge that we were at before v_1, in this case v_0, and test it again.

General Graph Properties

Assuming our graph is connected, and undirected, we have

- $|V| = n$
- $n - 1 \leq |E| \leq \frac{n(n-1)}{2}$
  - There cannot exist two edges between vertices in an undirected graph.
  - Therefore we have $(n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$

CS 180: Discussion 2
Friday, Week 2
10/13/17

Graphs

Storage of Graphs

- Matrix
  - Query : $O(1)$
  - Add Value: $O(n)$
  - Add edge $O(1)$
- Linked List
  - Query: $O(n)$
  - Add value: $O(1)$
  - Add edge: $O(1)$

Searching Graphs

- **DFS** - depth first search
  - Search as far down levels as you can
  - When you can't go down further, backtrack to the last one and check any other paths.
  - If out of paths for that node, backtrack again until we find one an unchecked path

- DFS == **STACK**

- **BFS** - breadth first search
  - Search in neighborhood of the original point
  - When all points nearby are searched, move to one of the neighbors and check all of his neighbors
  - Works level by level.
  - BFS == **QUEUE**

Implementing Each Searching Algorighm

DFS

```
def DFS(root)
    check root
    if root.child == None:
        return
    else for child in root.children:
        if child has already been checked
            do nothing
        else
            return DFS(child)
```

- How can we know if we have a cycle or not?
  - Implement this using a stack
  - If the number you have is already in the stack, we have a cycle

```
def DFSStack(root)
    check root
    if root.children== None:
        stack.pop(root)
        if stack.isempty
            return
    else:
        for child in root.children:
            check child
            add child to stack
            return DFSStack(child)


```

BFS:

Which is better to use when?

- If we want a search graph, we should use BFS.
    - Can search all nearby nodes, and most often graphs wont be too deep
    - For DFS, we may get stuck going down a really long path and doing a lot of unnecessary searches.

Trees

Different kinds of Trees

- Trees
    - Acyclic
- Binary Trees
    - Each parent node has at most 2 children
- Binary Search Trees
    - Looking at a node, everything will a value smaller than that node will be down the left path, and everything with a larger value than that node will be down the right path.

Tree Traversal

- In order
    - Go down the left edge and call In Order on that
    - Print node we are on
    - Go down the right edge and call In Order on that.
- Pre-order
    - Print the node we are on
    - Go down the left edge and call In Order on that
    - Go down the right edge and call In Order on that.
- Post-order
    - Go down the left edge and call In Order on that
    - Go down the right edge and call In Order on that.
    - Print the node we are on

Each of these traversals is $\mathbf{O(n)}$

Binary Search Trees

Let us consider the different operations that we can do

Insertion:

We cannot assume all BSTs are balanced, so this has $O(log(n))$ on average, but $O(n)$ worst case

```
def insert(num, root):
    if num > root.data:
        if root.right == NULL:
            root.right = new node(num)
            return
        else
            return insert(num,root.right)
    else
        if root.left == NULL:
            root.left = new node(num)
            return
        else
            return insert(num,root.left)
```

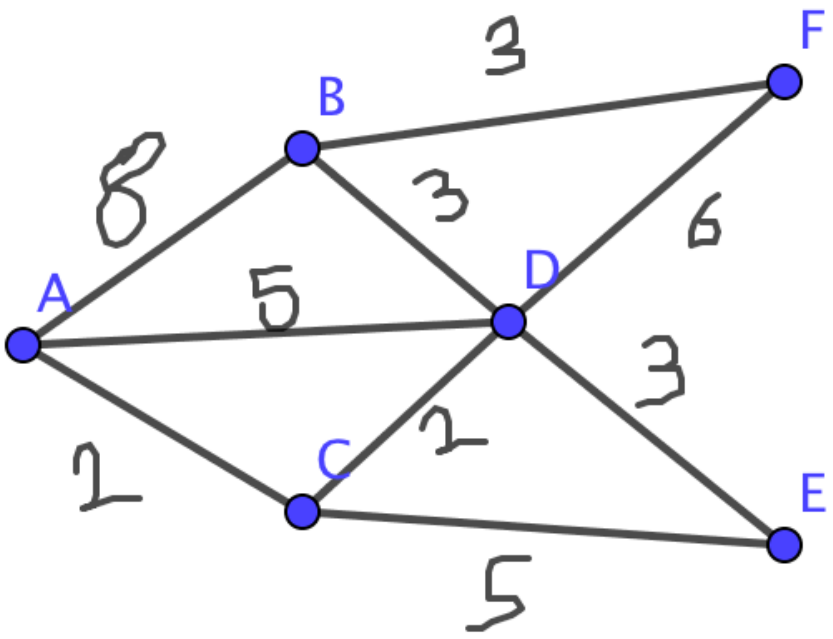
Shortest Path Problems

The problems we deal with are single source shortest path problem

- Find the distance from one point to every other point.

If graph is unweighted, then we can just use BFS, but most problems will be concerned with graphs with weighted edges. The famous algorithm that defines how to solve this problem is called Dijkstra's Algorithm

Let us consider the following graph



We want to find the shortest path from A to F. Here's how we do this

```
#DIKJSTRA'S ALGORITHM (for this problem)

* give each node a value. our starting node A = 0, every other node is infinity
* update each neighbor node of A based on size of edge
    - C = 2, D = 5, B = 8
* choose the smallest node to consider next (C), and mark it as checked
* update all neighbors of C with a new value V = value(C) + size(edge)
    - If a value is already there, keep the smallest value
    - B = 8, D = 4, E = 7
* choose D to consider since it is smallest
* update again, this time from D = 4
    - E = 7, F = 10, B = 7
* now consider E, but it has no neighbors to consider, so move on to B
* update again from B = 7
    - F = 10 (doesn't change)
* Since F is the smallest value left, our algorithm is done

CONCLDE: Shortest path to F is 8 units. It is from A -> C -> D -> F


```

In general we can generalize Dikjstra's Algorithm like so

```
#DIKJSTRA'S ALGORITH FOR SHORTEST PATH

1) Give starting vertex a value of 0, all other vertexes a value of infinity
2) Set initial node as current, all other nodes as unvisited
3) Consider each neighbor of initial node and calculate the tentative distance. The tentative distance is denoted as value(current_vertex) + value(edge_to_new_vertex). If the new vertex already has a value, keep the smaller of the two
4) Once we are done, mark the current node as visited.
5) Consider stopping conditions
   a) If unvisited node with smallest tentative distance is desired node, we are
   finished
   b) If smallest unvisited node has tentative distance infinity, then no connection
   between initial node and desired node, return failure
6) Otherwise, set the node with smallest tentative distance as current and return to step 3.

```

CS 180: Lecture 6
Tuesday, Week 3
10/17/17

Bredth First Search

Time Complexity

Not all graphs are connected, and wether or not a graph is connected can change the time complexity of BFS. Assume $G(e, n)$

- If the graph is connected, then we can just say that the search is $O(e + n)$
- If the graph is disconnected however, then we must visit different components as well, so the complexity becomes $O(e + n)$

Directed Graphs

The problem is essentially the same for directed graphs, but the most important thing that changes is the definition of **reachability**.

Problems with Directed Graphs

Topological Sort

If we have a directed graph that doesn't have a cycle, we call that a **Directed Acyclic Graph (DAG)**. If we are looking to find a cycle in a directed graph, we can do this using a DFS algorithm.

Given these graphs, the problem of trying to sort these graphs is called **Topological Sorting**. TS has the unique propery that it does not necesarrily have a unique solution , there can be more than one solution ot the problem. We want to find an algorithm for this, but first we need to define a couple of things

in-degrees - the number of directed edges coming in to that vertex, $deg^{-}(v)$

out-degrees - the number of directed edges leaving a vertex. Denoted $deg^{+}(v)$

source - a vertex v such that $deg^{-}(v) = 0$

The algorithm, known as **Kahn's Algorithm** is as follows

```
#Kahn's Algorithm for topological sort

1) Choose a source
2) Output the source
3) Update in-degrees of all of the vertecies connected to the source
   a) If any of these become a source, add them to the source list
4) Move back to step 1 and perform this recursively.

```

Now we will investigate the time complexity of this algorithm. Evaluate the inner loop of the algorithm

- Initial the degrees of each vertex = $O(e)$
- Find a source such that the in-degrees of the vertex is 0 $O(n)$
- Updating the in-degrees of each vertex = $O(n)$
    - Maximum, we need to check $n-1$ verticies

The inner loop runs at $O(n)$, and we need to run this loop for n points at most, so we get a final complexity of $\boxed{O(n^2)}$.

However we can do a different analysis of the algorithm

- Initialize the degrees of each vertex = $O(e)$
- Find a source such that in-degrees of vertix is 0 = $O(n)$
- There are e edges, and we process each edge once, so updating in-degrees = $O(e)$

Therefore, we have gotten rid of the loop and we have a runtime of $\boxed{O(e+n)}$

How did analyzing the algorithm in two different ways give us two different runtimes

- One method analyzed the algorithm as a loop and considered the number of times the loop could be run
- The other method analyzed entire algorithm in full

(For further resources on Kahn's Algorithm: http://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/)

Two-Color Graphs

Given a graph G, how can we make it so that no two adjacent vertecies have the same color?

- Easy solution: Give every vertex a different color.

The more interesting problem arises when we attempt to limit on the number of colors. To help understand this problem, we define a different term that will lead us to the solution.

Bipartite Graph - a bipartite graph is a graph such that the verticies can be separated into two disjoint sets such that no two verticies within the same graphs are adjacent

From this we can get the following conclusions about 2-colorability of a grpah

- A graph is 2-colorable if and only if it is bipartite
- If a graph contains any cycles with odd length, it is not 2-colorable

To figure out wether a graph is bipartite/2-colorable, we can use a BFS algorithm.

Creative Problem

Given

- G is a directed, acyclic graph
- K is the maximal number of edges in any path

Goal: Partition V into $K+1$ groups such that there are no edges between verticies in a group

How can we approach this problem? Look at givens for hints.

Solve for Thursday lecture.