CS 180: Lecture 1
Thursday, Week 0
9/28/17

Course Introduction

When we create algorithms, we want to use a **Universal Method of Computing**. It needs to be applicable across all OS/Systems an able to be used in the distant future.

When we design and analyze algorithms, there some things we need to consider

- How well the algorithm performs (resources)
- We want to try to find a Lower Bound Algorithm
- Analyze the time complexity of our algorithm

Random vs. Arbitrary

Many people think of random and arbitrary as the same things, but in terms of designing our algorithms, they are quite different

Random - based on finite/set probability function

- ex: Normal Distribution/ Binomial Distribution
- use this predetermined probability function to decide what we want
- Disadvantage: Can be very time/resource consuming to go through this random process

Arbitrary - determined by something categorical, or just chosen on a whim

- ex: can be chosen by height, weight, alphabeitcal, or just chosen on a whim
- Advantage: Arbitrary choice makes proofs simpler

As we can see, having to rely on random choice probably will not be best for our algorithms, so we want to construct algorithms with **arbitrary chioce** when applicable.

Egg Problem

Our goal is that we want to find the highest floor of an n story building such that when when you drop an egg from that floor, the egg won't break. However, our constraints on the problem are that we only have two eggs to test with, and we only are allowed 10 drops. Given our constraints, what is the maximum amount of floors that we can test on to guarantee we find a solution?

Break this problem down.

- What if the egg breaks on the first drop?

    - Then we have nine more tries with our last egg to find the floor.
    - Implies we should start at floor 10.
- What if the first egg doesn't break but the second egg breaks
    - Then we have eight more tries with our last egg
    - We know that floors 1-10 don't need to be tested because the first egg survived
    - Therefore, we need to test floors 11-18 with our 8 tries, so we should move up 9 floors to floor 19.

Following this pattern, we see that the amount of floors we can test on is

$$10 + 9 + \ldots + 1 = \sum_{k=1}^{10} k = \boxed{55}$$

CS 180: Lecture 2
Tuesday, Week 1
10/3/17

Problem Set 1: Due Next Thursday, 10/12/17

Serial Computation

When we talk about a **serial model of computation**, we are saying that one action is executed at a time in the computer. There are several components of a computer to consider when using this model.

ALU (Arithmetic Logic Unit) - performs basic arithmetic operations

RAM (Random Access Memory) - for our purposes, we have infinite RAM to solve problems

I/O (Input/Output) - devices to relay information to the computer/for the computer to relay information back to the user.

This model is very basic/very universal - applies to most/all systems, which is what we desire when creating algorithms.

Example Problem: Adding Numers in Series

Lets consider a possible algorithm for adding numbers in a series, i.e.

$$x_1 + x_2 + \ldots + x_n = S$$

How would we write this down according to our serial method of computation?

- Read x_1, put it in ALU

- Read x_2, add it to x_1
- $...$
- Read x_n add it to x_1 + $...$ + x_{n-1} stored in ALU
- Output Answer

Analysis of the algorithm

- Total Operations: 1 + 2(n-1)
  - x_1 iteration doesn't have an add operation
- $2n$ for input, and 1 for output
- Total: $2n + 1 + 2n - 2 + 1 = \boxed{4n}$

Matching Problem

Consider a group of Men and women that have interest in each other.

Defining the problem

- We have n men and n women (same of each)
- Each man and women has an order of preference for matching with a partner
- If we have two people that aren't matched that prefer each other over their partners, then we refer to this as an UNSTABLE matching
  - i.e. m matched with w and m' matched with w' but m and w' prefer each other.
- If we do not have this solution anywhere, the solution is STABLE

Find an algorithm that guarantees a stable solution every time.

Algorithm

While ∃ an unmatched man...

- Pick a man that hasn't been matched yet (arbitrary), called m
- Take the highest woman left in his priority list, called w and attempt to temporarily match them
  - If the woman is unmatched, she automatically accepts
  - If the woman is matched with another man m', compare in her priority list
    - If m is higher than m' on her list, she leaves m' who is now unmatched again, and matches with m.
    - Else, m' is higher than m on her list, she keeps m' and m remains unmatched
  - In either case, remove w from the priority list of m, since she has been considered either way

Note: Write solutions in bullet form for clarity

Analysis

Does this solution leave everybody matched at the end?

- Because each man will have every woman in his list, each woman will be matched with somebody eventually, and since ever woman is matched, then every man will be matched also

Now we need to prove the stability of our solution. Let's assume the unstable case occured, and prove by contradiction that it isn't possible.

- Unstable case: m matched with w and m' matched with w' but m and w' prefer each other.
- Case 1: m proposed to w' before
    - If they were matched previously, then w' will never match with m' in our algorithm since he is lower on her priority list than m
    - If they were not matched previously, then her match, called m'', must be higher than m on her priority list, and thus m'' is also higher than m' on her priority list.
- Case 2: m did not propose to w' before
    - This case is not possible. The unstable case has m matched with w, who is lower on his priority list than w', but according to our algorithm, that means he must have proposed to w' at some point before considering w.

Therefore, we have shown our solution is valid and always produces a stable matching.

CS 180: Lecture 3 Thursday, Week 1 10/5/17

Task Problem

Assume you are given a set of daily tasks. Each task is represented by an interval on the time axis (ex: 10-12, 9am - 4pm, etc...). Given the entire set of tasks, find a subset of non-overlapping tasks.

Simple Solution - just pick one task arbitrarily

- When given a question, read it thoroughly to see exactly what it asks
- Don't overcomplicate it and try to solve something it isn't asking
- Give the simplest possible answer

Clarification: We want the solution that maximizes the number of tasks in this subset.

Algorithm 1: Pick Smallest First

A possible algorithm to find the solution is given as follows

- Pick the smallest interval, and add it to the timeline
- Eliminate all other tasks that overlap it

- Repeat until all tasks are eliminated

Does the algorithm solve our problem? NO. Proof by counter-example:

|————————————||————————————————————|

|—————————|

- According to our algorithm, we would take the smallest one (the one on the bottom) and eliminate all overlaps (ones on top)
- However, we see that we only have one task, where we could possibly have two, so this algorithm is bad.

Algorithm 2: Pick First Event

- Pick the first event point and add it to the timeline
- Eliminate all overlaps
- Repeat

Does this algorithm solve our problem? NO. Proof by counter-example:

|—————————————————————————————————————|

|———||———————||———————||——————||—————————|

- According to our algorithm, we take the first task (top one) and eliminate the overlaps (whole bottom row)
- It is clear that the bottom row is the far more optimal solution

Algorithm 3: Pick the Event that Ends First

- Pick task that ends first
- Eliminate overlaps
- Repeat

Does this algorithm solve the problem? YES. Let's prove it by contradiction

- Assume our solution isn't optimal. Then $\exists$ an optimal solution that is different than ours.
- Assume the first i intervals, $0 < i < k$, are the same
- Compare the next interval, the $i + 1$ interval.
- Because of the construction of our algorithm, the $i + 1$ interval of our solution will end before the $i + 1$ interval of the other solution. Replace the $i + 1$ interval in this proposed solution with the one from our solution.
- Now the first $i + 1$ intervals between our solution and proposed optimal solution match
- Continue this process through the k_{th} interval
- Therefore, proposed optimal solution is now the same as our solution, and we have proved that our algorithm produces an optimal solution by contradiction.

CS 180: Discussion 1
Friday, Week 1
10/6/17

Problem 1: Combine Sorted Arrays

We are given two separate sorted arrays of numbers. Combine them into one sorted array

- Compare the first two numbers in each array
- Take the smaller one, put into first element of sorted array and advance index of array it came from
- Repeat.

Time complexity: If array1 has size n, and array2 has size m, then this is $O(m+n)$

Problem 2: Up-Down Array Sorting

Given a random array, ex: 4 1 2 6 5 3. Return an array sorted such that the relationship between each value is up, down, up, down.... ex: 1 3 2 5 4 6. ($1 \to 3 =$ Up, $3 \to 2 =$ down, etc...)

Naieve approach

- Sort the array
    - $O(nlog(n))$
- Take the first value, put in new array
- Take the last value, put it in next element of new array
- Repeat until finished.

Better approach

- Compare the first two values
    - If the relation is correct, ignore and move to next pair
    - If the relation is not correct, switch the two values
- Repeat until done.

This method has $O(n)$ time which is better than $O(nlog(n))$.

Problem 3: Proper Parenthesis

We are given a string of parenthesis. How do we tall if it is a valid string (i.e. all openings are closed and all openings and closings are proper)

Examples

- "(() (()))" is valid
- ") () () (())" is invalidL closing parenthesis before open parenthesis
- "( ))" is bad - extra closing parenthesis
- "( ))" is bad - extra closing parenthesis

My Solution

```
bool solve(string parenthesis_string)
{
  n = 0;
  counter = 0;
  while (n < parenthesis_string.length())
  {
    //Check the individual character and change the counter
      if (parenthesis_string[n] == "(")
      {
        counter++;
      }
      else if (parenthesis_string[n] == ")")
      {
          counter --;
      }
      else
      {
          cout << "Error: Invalid string given.";
          return false;
      }

    //Check counter. If it's negative at any point, we have too many closing parenthesis
      if (counter < 0)
      {
          cout << "Error: Extra closing parenthesis or closing parenthesis before open parenthesis."
          return false;
      }

     //Increment n
      n++
  }

  //End of loop over string, check the counter
  if (counter == 0)
  {
      cout << "Valid parenthesis string given";
      return true;
  }

  else if (counter > 0)
  {
      cout << "Error: Too many open parenthesis"
      return false;
  }

  else
  {
      cout << "Error: Too many closing parenthesis";
      return false;
  }
}
```

- Keep a counter for the parenthesis string
    - Increment the counter if we have an opening parenthesis
    - Decrement the counter if we have a closing parenthesis
- If the counter is ever negative, then we have a closing parenthesis before an opening parenthesis , so this is not valid.
- If we get to the end and the counter is not 0, that means something hasn't been closed or there's an extra closing paren
- If the counter is 0, then we return true.

Another Solution: The Stack!

- If open parenthesis, push onto the stack
- If closed parenthesis, pop from the stack
    - If we can't pop the stack, too many closed paren, return false
- At the end, check stack
    - If stack is empty, the parenthesis string is valid, return true
    - If the stack is nonempty, some open parenthesis didn't get closed, return false.

Dont forget about using Stacks/Queues in solutions to problems.

Problem 4: Greatest Numer after Each Element

We have the following problem:

- Given an array of numbers, we want to find the greatest number after each element
    - Put -1 if such a number does not exist.
- Example array and solution below: 1 2 1 5 20 10 15

  20 20 20 20 -1 15 -1

Think about this problem and come back next week with a solution.

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 $\exists$ constants n_0, c such that $\forall\ n \geq n_0$, 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 $\exists$ 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 $\underline{\Theta(n)}$

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: Lecture 5 Thursday, Week 2 10/12/17

SICK FOR THIS LECTURE AND MISSED NOTES. IF YOU WANT NOTES HERE PLEASE SEND TO M

- **FACEBOOK (Connor Jennison)**
- **EMAIL (connorjennison@me.com)**

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)
```

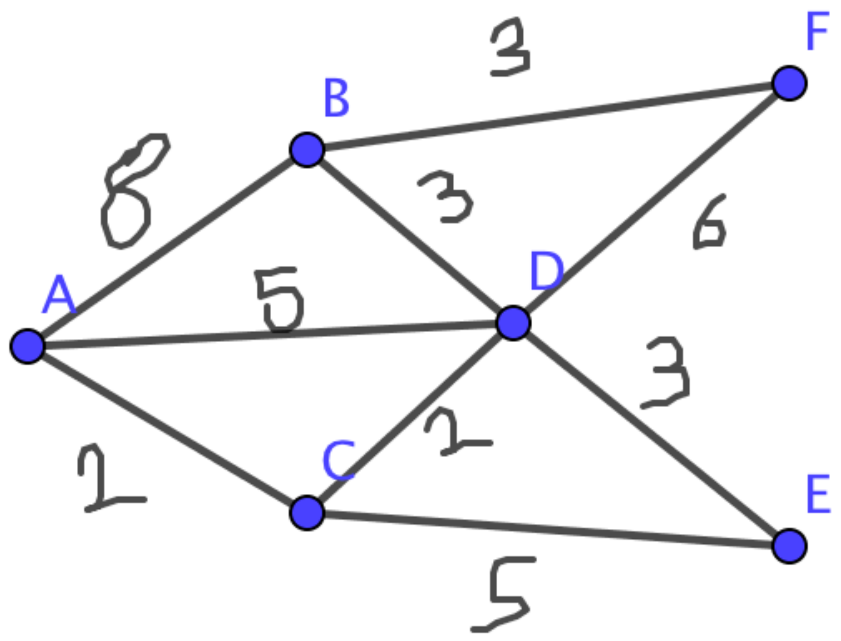
Shortest Path Problems

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.