Matthew Morris
Professor Kontothanassis
DS210
2 April 2025

DS210 HW7 Journal

Question 1:

In this question, the plan is to compute an approximation to *PageRank*. PageRank was developed as a method for ranking web pages.

```
use std::fs::File;
use std::io::{self, BufRead};
use std::path::Path;
use rand::Rng;

struct Graph {
   vertices: usize,
   adjacency_list: Vec<Vec<usize>>,
}
```

This code imports the necessary packages to accomplish the pagerank task. Firstly, I import file system management so that the code can take in a .txt file. Then, using the io package, I use a buffer read input. Then I use path, which creates paths for the filesystem. Then I also important rand::rng which allows me to randomize values.

From there, I construct a struct of a graph which is necessary because we are creating a directed graph as per the instructions. The vertices are stored in usize, along with the adjacency list. This is stored as a vector within a vector because we want to have them be pairs.

```
fn read_lines<P>(filename: P) -> io::Result<io::Lines<io::BufReader<File>>>
where
   P: AsRef<Path>,
{
   let file = File::open(filename)?;
   Ok(io::BufReader::new(file).lines())
}
```

This function reads the lines of the respective filename I've assigned (I assign in main function) so that we can take the data and read them in individual lines. This has buffer reading for efficiency. (Source: https://doc.rust-lang.org/std/io/trait.BufRead.html)

```
impl Graph {
```

```
    fn new(vertices: usize) -> Self {
        Graph {
            vertices,
            adjacency_list: vec![Vec::new(); vertices],
        }
    }
```

Next, we create an implementation of a graph, taking both vertices and an adjacency list. This creates a new vector for the adjacency list. I start by initializing “new” which takes a new graph with empty adjacent lists for each vertex. From there it adds a directed edge from one vertex to another by appending to the adjacency list.

```
fn walk_simulation(&self, start: usize, steps: usize) -> usize {
        let mut current = start;
        let mut rng = rand::thread_rng();

        for _ in 0..steps {
            if self.adjacency_list[current].is_empty() {
                current = rng.gen_range(0..self.vertices);
            } else {
                let random = rng.gen::<f64>();
                if random < 0.8 {
                    let edges = &self.adjacency_list[current];
                    current = edges[rng.gen_range(0..edges.len())];
                } else {
                    current = rng.gen_range(0..self.vertices);
                }
            }
        }
```

Then, we simulate walking based on the criteria mentioned in the prompt. For each step it goes through the following criteria: first, if current vertex has no outgoing edges, jump to random vertex. Otherwise, with 80% probability it will follow a random outgoing edge or 20% of the time it will jump to a random vertex. It then returns the final vertex after all steps (90 steps).

```
fn compute_pagerank(&self, num_walks: usize, steps_per_walk: usize) -> Vec<(usize,
f64)> {
        let mut visit_counts = vec![0; self.vertices];

        for start in 0..self.vertices {
            for _ in 0..num_walks {
                let end = self.walk_simulation(start, steps_per_walk);
```

```
                visit_counts[end] += 1;
            }
        }

        let total_walks = num_walks as f64 * self.vertices as f64;
        let mut pageranks: Vec<(usize, f64)> = visit_counts
            .iter()
            .enumerate()
            .map(|(vertex, &count)| (vertex, count as f64 / total_walks))
            .collect();
        pageranks.sort_by(|a, b| b.1.partial_cmp(&a.1).unwrap());
        pageranks
    }
```

This function computes pagerank. It takes in the number of walks, the steps of each walk (around the graph). From there it computes page rank scores from random walks. For each vertex taken in, it performs num_walks random walks. Then it counts how often each vertex is visited. Then it normalizes the counts by the total number of walks. This is because we want to have everything as a percentage, rather than a raw value, as described by the instructions.

```
fn main() -> io::Result<()> {
    let mut graph = None;
    let mut line_count = 0;

    for line in read_lines("pagerank_data.txt")? {
        let line = line?;
        let parts: Vec<&str> = line.split_whitespace().collect();

    Ok(())
}
```

In the main function first, we read the graph data from pagerank_data.txt. This is also where we assign the specific file to read from (I used data.txt initially, which had the sample test for quicker testing). When I first initialize main, it takes io (input/output as the file function). It also intializes a new graph (mutable) = none. It also sets the first line for the .txt file reader to 0 so that the function knows where to start. From there, it reads line by line of the .txt file, splitting at whitespace.

```
        if line_count == 0 {
            let vertices = parts[0].parse::<usize>().unwrap();
            graph = Some(Graph::new(vertices));
```

```
        } else {
            let from = parts[0].parse::<usize>().unwrap();
            let to = parts[1].parse::<usize>().unwrap();
            if let Some(ref mut g) = graph {
                g.add_edge(from, to);
            }
        }
        line_count += 1;
    }
```

In this part of the function, we create conditionals: firstly, if the line count is 0, intializes vertices to equal the first location in parts, then it unwraps it. Then it sets the graph to be the new vertices. This function essentially serves to map what is read in the .txt file to the graph, after being read line by line.

If the line count isn't 0, it will do the same part as the other one, but it will create edges if the from and to are connected.
It then adds 1 to line count which allows the line reader to move onto the next line of the .txt file and repeat the process over again.

```
  if let Some(graph) = graph {
        let pageranks = graph.compute_pagerank(90, 90);

        for (vertex, rank) in pageranks.iter().take(5) {
            println!("vertex {}: approximate PageRank {:.3}", vertex, rank);
        }
    }
```

This computes page ranks as 90 walks per vertex and 90 steps per walk.

From there, it prints a function that takes the top 5 page ranks, and prints them out.

```
#[cfg(test)]
mod tests {
   use super::*;

   #[test]
   fn test_sum_of_pagerank_is_1() {
       let mut g = Graph::new(3);
       g.add_edge(0, 1);
       g.add_edge(1, 2);
       g.add_edge(2, 0);
```

```
        let pageranks = g.compute_pagerank(10, 10);
        assert_eq!(pageranks.len(), 3);
        let total: f64 = pageranks.iter().map(|(_, pr)| pr).sum();
        assert!((total - 1.0).abs() < 0.01); //need to check if approximate pagerank is
close to 1
    }
}
```

As a test, the sum of pageranks across vertices should equal 1. This test, given 10 walks per vertex and 10 steps per walk with a set 3 edges, should equal one. As a result, I add up the pageranks, and since they are decimals they potentially won't perfectly add up to one, but they will be close. I set 0.01 as a margin for error, saying that if the total-1 is +/- 0.01 (done by the absolute value, then the page rank function is done correctly.

Evaluating the complexity of this function, we get O(90*90*V) , however it is more likely O(W*V*S). W and S are fixed (walk and step count) and vertices is different.

Output of question 1:

vertex 603: approximate PageRank 0.002
vertex 596: approximate PageRank 0.002
vertex 948: approximate PageRank 0.001
vertex 94: approximate PageRank 0.001
vertex 801: approximate PageRank 0.001

running 1 test
test tests::test_sum_of_pagerank_is_1 ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

Question 2:

```
use std::collections::VecDeque;
use rand::Rng;
```

Import necessary packages, rng and vecdeque (learned in lecture)

```
#[derive(Debug)]
struct Node {
   value: i32,
   left: Option<Box<Node>>,
   right: Option<Box<Node>>,
```

```
}
```

Assigning left and right nodes, as well as value. Left and right are for the left and right parts of the tree, and are navegable as a result.

```
impl Node {
   fn new(value: i32) -> Self {
       Node {
           value,
           left: None,
           right: None,
       }
   }
```

Creates a new node given a value and no children.

```
fn add_child(&mut self, value: i32) -> bool {
       if self.left.is_none() {
           self.left = Some(Box::new(Node::new(value)));
           true
       } else if self.right.is_none() {
           self.right = Some(Box::new(Node::new(value)));
           true
       } else {
           false
       }
   }
}
```

Tries to attach a child note with the given value. First it checks if left is empty. If it is empty, it will add a new node there. Otherwise it will try the right side of the tree. If both children exist, it will return false. It will return true if a child is successfully added.

```
struct BinaryTree {
   root: Option<Box<Node>>,
}
```

Initializes the struct of the binary tree which we can navigate. Intializes the root as well.

```
impl BinaryTree {
   fn new() -> Self {
       BinaryTree { root: None }
```

```
    }
```
Creates an empty binary tree, with root = none.

```
    fn build_random_tree(&mut self, node_count: usize) {
        if node_count == 0 {
            return;
        }
```

If node count is 0, returns the result. This function takes node_count (unknown size).

```
let mut rng = rand::thread_rng();
        let mut nodes_to_process = VecDeque::new();
        let mut current_count = 0;

        self.root = Some(Box::new(Node::new(current_count as i32)));
        nodes_to_process.push_back(self.root.as_mut().unwrap());
        current_count += 1;
```

From there, this function helps create the random tree, creating random values for the nodes. Creates a queue (vecdeque) for breadth first search. It then creates a counter for how many nodes have been created.

```
        while current_count < node_count && !nodes_to_process.is_empty() {
            if let Some(current_node) = nodes_to_process.pop_front() {
                let children_count = rng.gen_range(1..=2);
```

This goes on to pop the front, and generate a range. This allows the nodes to be processed correctly, where the current count is always less than node count. Continues as long as note count hasn't been hit, and there are still nodes to process.

```
                for _ in 0..children_count {
                    if current_count >= node_count {
                        break;
                    }

                    if current_node.add_child(current_count as i32) {
                        current_count += 1;
                    }
                }
```

Obtains the current node from queue, and randomly decides to add 1 or 2 children, as defined by instructions.

```
        if let Some(left) = current_node.left.as_mut() {
                    nodes_to_process.push_back(left);
                }
                if let Some(right) = current_node.right.as_mut() {
                    nodes_to_process.push_back(right);
                }
            }
        }
```

If there are enough nodes, it breaks the function, and then adds child using add_child. If successful, it adds one to current_count.

```
    fn find_diameter(&self) -> i32 {
        if let Some(root) = &self.root {
            let (_, diameter) = self.dfs_diameter(root);
            diameter
        } else {
            0
        }
    }

    fn dfs_diameter(&self, node: &Box<Node>) -> (i32, i32) {
        let mut max_diameter = 0;
        let mut depths = Vec::new();

        if let Some(left) = &node.left {
            let (depth, diameter) = self.dfs_diameter(left);
            depths.push(depth);
            max_diameter = max_diameter.max(diameter);
        }

        if let Some(right) = &node.right {
            let (depth, diameter) = self.dfs_diameter(right);
            depths.push(depth);
            max_diameter = max_diameter.max(diameter);
        }

        depths.sort_unstable();
        let current_diameter = depths.iter().sum::<i32>();
        max_diameter = max_diameter.max(current_diameter);
```

```
        let max_depth = depths.last().unwrap_or(&0) + 1;

        (max_depth, max_diameter)
    }
}
```

This function allows for double breadth first search. First computes the diameter of the entire tree. It then calls double breadth first search on the root, returning the diameter part of the tuple, ignoring depth. Returns 0 for an empty tree.

Then it uses recursion to obtain the max depth and diameter. It sets up a max diameter and alsit to store depths from left/right.

If there is a left child: recursively get its depth and diameter. Push depth into depths. Updates max diameter if the childs diameter is larger.

```
fn main() {
    let mut tree = BinaryTree::new();
    tree.build_random_tree(120);
    let diameter = tree.find_diameter();
    println!("Tree diameter: {}", diameter);

    if let Some(root) = &tree.root {
    println!("Root node value: {}", root.value);
    }
}
```

Main function simply prints the main root value and the diameter as defined by the function. It also assigns 120 for the number of notes.

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_tree_diameter() {
        let mut tree = BinaryTree::new();
        tree.build_random_tree(120);
        let diameter = tree.find_diameter();
        assert!(diameter > 0, "Diameter should be positive for a tree with 120 nodes");
        assert!(diameter <= 119, "Diameter cannot exceed n-1 for a tree with n nodes");
    }
```

```rust
    #[test]
    fn test_small_tree() {
        let mut tree = BinaryTree::new();
        tree.build_random_tree(5);
        let diameter = tree.find_diameter();
        assert!(diameter >= 0 && diameter <= 4, "Diameter should be between 0 and 4 for
a 5-node tree");
    }

    #[test]
    fn test_empty_tree() {
        let tree = BinaryTree::new();
        let diameter = tree.find_diameter();
        assert_eq!(diameter, 0, "Empty tree should have diameter 0");
    }

    #[test]
    fn test_single_node_tree() {
        let mut tree = BinaryTree::new();
        tree.build_random_tree(1);
        let diameter = tree.find_diameter();
        assert_eq!(diameter, 0, "Single node tree should have diameter 0");
    }
}
```

Creates tests for a variety of things. Used chatgpt to find tests.

Output:
Tree diameter: 16
Root node value: 0

Source:
DFS
https://efficientcodeblog.wordpress.com/2017/12/13/bidirectional-search-two-end-bfs/

✅ 4. Structural Invariants

No Node Has More Than 2 Children

You can write a recursive check:

rust

```rust
fn verify_binary_property(node: &Box<Node>) {
    if node.left.is_some() { verify_binary_property(node.left.as_ref().unwra
    if node.right.is_some() { verify_binary_property(node.right.as_ref().unw
}
```

Run this after `build_random_tree()` to ensure valid binary tree.

✅ 5. Statistical Test for Tree Breadth

Run 100 trials of `build_random_tree(120)` and assert that the **average depth** and **diameter** are within some statistical bounds:

- Mean diameter is not too small (suggests undergrowth).
- Max diameter stays under n−1.

✅ 6. Negative/Invalid Input Handling

You could enhance `build_random_tree()` to handle invalid cases (e.g., `node_count =`

✅ 2. Small Fixed Trees (Deterministic Behavior)

Even though your tree is random, you can build **manual trees** using `Node::add_child` directly to test known structures.

Manually Constructed Tree with Known Diameter

rust

```rust
let mut root = Box::new(Node::new(0));
root.add_child(1); // Left
root.left.as_mut().unwrap().add_child(2); // Left of Left
let tree = BinaryTree { root: Some(root) };
assert_eq!(tree.find_diameter(), 2); // path: 2 → 1 → 0
```

This tests:

- Depth propagation
- Diameter via a specific subtree

✅ 3. Random Trees (Probabilistic, Property-Based)

For larger trees built randomly:

General Validity Tests

rust

```rust
let mut tree = BinaryTree::new();
tree.build_random_tree(100);
let d = tree.find_diameter();
assert!(d > 0 && d <= 99); // max diameter in 100-node tree is 99
```

Repeat with different sizes (10, 50, 1000) to test scaling what tests are ideal for this program?

Great question — testing tree structures like this requires verifying **both structure and behavior**, especially since you're building **random binary trees**. Here's a breakdown of **ideal tests** for this program:

✅ 1. Base Cases (Edge Conditions)

Empty Tree

```rust
let tree = BinaryTree::new();
assert_eq!(tree.find_diameter(), 0);
```

- Verifies correct handling of None root.

Single Node Tree

```rust
let mut tree = BinaryTree::new();
tree.build_random_tree(1);
assert_eq!(tree.find_diameter(), 0);
```

- Tree has no edges, so diameter is 0.

✅ 2. Small Fixed Trees (Deterministic Behavior)

Even though your tree is random, you can build **manual trees** using `Node::add_child` directly