<u>DS210 Final Project Writeup</u>

For my project, I chose to analyze a directed citation network and determine the average number of citations across all nodes as well as the shortest distances between each node. I did this through modules main, bfs, and degree, and I used the BFS algorithm and calculated averages to determine my results.

The main.rs module houses the main function, the file reading function, and the adjacency list creation function. As the code traverses each module, it first reads the graph's text file with the read_file function. This function iterates through the graph and collects all of the values into one vector. One of the most critical pieces I encountered in this function was the .split_whitespace() method, which omits any spacing when reading the graph, thus ensuring an accurate read. When I tested this function throughout my project, it at first could not read the file because of the white spaces, so I had to implement this method in the function. Once the white spaces have been cleared, map() and collect() create a new iterator of the graph values. The loop below the main body ensures that if one line is faulty (ex. Contains "47389" instead of "47389 28292"), it will skip that line and omit it from the graph. At the end, we determine the maximum node count by retrieving the value of the largest vertex. This value will be the value that the graph iterates over, ensuring that every node is covered when we run the remaining algorithms. The enum Result ensures that if the file reading was successful, we can proceed with the rest of the code. If the read encountered an error (ex. Missing values, white space, invalid characters), the process would return an Err result instead.

The match at the start of main ensures that the read_file function produced an Ok result, thus allowing the program to proceed. The next step creates the adjacency list with adj_list. The list is first initialized as an empty vector with a length of the amount of nodes in the graph. After that, the function iterates over the graph (the resulting tuples from the read_file function) and adds the tuples to the list and their neighbors. If a tuple's contents are out of bounds of the number of nodes, then the list accommodates by extending if necessary. When I ran the adjacency list function, I saw that some additions were blank brackets, meaning that those nodes did not have any neighbors. This could mean that either the nodes did not have neighbors or they did not exist. Here is a sample of what part of the adjacency list looked like during one of the code runs:

Node 10275: [1087, 1292, 3111, 3122, 9212235, 9403219, 9404201, 9405424, 9508379, 9603208, 9604320, 9605235, 9605266, 9605283, 9610247, 9612291, 9612346, 9612364, 9703212, 9705201, 9707415, 9710234, 9710401, 9710538, 9804255, 9805264, 9806354, 9806403, 9806519, 9809291, 9809506, 9811369, 9902220, 9903227, 9903513, 9905441, 9907365, 9909364, 9912278]

Node 10276: [9207268, 9403257, 9505340, 9906272]
Node 10277: [2050, 9205247, 9302207, 9304267, 9308315, 9604320, 9612364, 9710401, 9801272, 9804255, 9806354, 9810499, 9902220, 9902371, 9903276, 9905441]
Node 10278: []
Node 10279: [7020, 9207273, 9308205, 9410242, 9612292, 9803315, 9803466, 9903407, 9912343]
Node 10280: [9502417, 9612239, 9809583, 9904409]
Node 10281: []

As we can see here, the adjacency list of each node presents the neighbors to that node, which are then used in BFS to determine distance and degree of separation. Node 10278 and Node 10281 have blank lists, meaning that they may have no neighbors or do not exist within the ~41k nodes in the data set.

The adjacency list is then passed into the distances_bfs function, which uses it to figure out the distances between each node. We begin by initializing a vector of distances with None values, which will be filled in with Some(x) values as the function iterates through the vector. The initial vertex is set to Some(0) to begin the traversal. Initializing VecDeque afterwards creates a queue of vectors to be traversed over, thus preventing a vertex from being iterated over multiple times and skewing the distances. If a vertex is represented by a None value in the initial vector, it means it has not been iterated over yet. The iterator thus updates the distance accordingly and moves to the next vertex in the queue. For the example output snippet I provided above to show the adjacency list, for node 10276, distances_bfs would return the shortest distances from 10276 to each of the nodes in the adjacency list, so the distances between nodes 9207268, 9403257, 9505340, and 9906272. Once the distances are determined, they are printed in an Option vector, so the output may look something like [Some(5), Some(10)...]. In the same module, once we have the distances from distance_bfs, we use collect_distances to collect them into a hash map of the node and distances for the next function to iterate through.

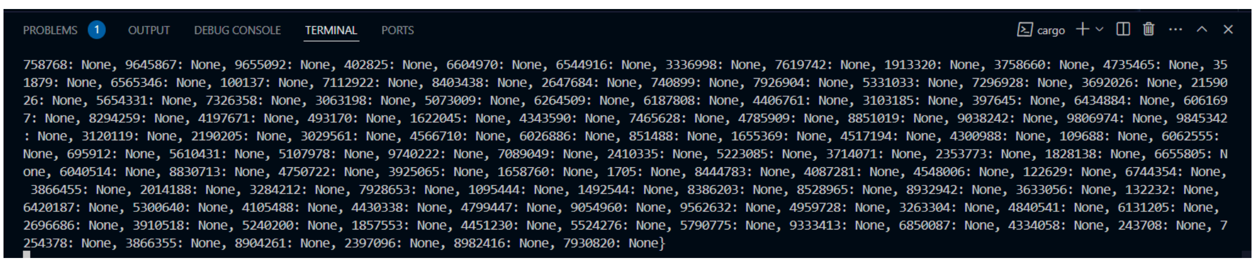
Once that function call has been made, the program moves to the degree of separation function, where it calculates the shortest distances between the vertices while adhering to a certain threshold. The function first calls collect_distances for the hash map of distances. It then initializes a new vector to collect the shortest distances, which would be the distances that fall above or equal to the specified threshold. If a distance falls within that threshold, then the distance and associated vertex are pushed into the new hash map. To test what this iteration would look like under different thresholds, the threshold range is set from 1 to 5. With this range, we could expect to see the amount of nodes in the shortest distance list increase or decrease over time depending on how many nodes fall within the threshold. However, because the graph consists of so many nodes, none of the nodes may fall within the threshold, thus resulting in a blank hash map.

Once the map of shortest distances has been created for each threshold, we then calculate the average number of citations per paper with the average_number_citations function. This function initializes a hash map for the citation count and then iterates over the edges in the graph, since the edges will determine the connections among each of the nodes. The loop then uses a closure to count the number of edges per node at certain degrees. For example, at a degree of 1, a node may have 2 direct connections; however, if the degree were changed to 2, the amount of connections may increase to 12 because it now takes neighbors that are 2 steps away into account. This is what we test in our main function too, where degree is set to 2, so we can see the direct neighbors and neighbors that are off by 1 degree of separation. Once we have the number of citations counted among the papers in the graph, we can sum the amount and then divide it by the total number of nodes, thus giving us the average number of citations across the network.

<u>Output</u>

Because of an unknown error not picked up by the compiler or unexpected circumstance, the code compiled and produced a result, but the result was not what I had expected. I ran various debug statements and tried to pinpoint the source of the unexpected output, but everything I tried continued to produce the same output. Thus, I had to leave the output as is. Here are some screenshots of what the code produced:

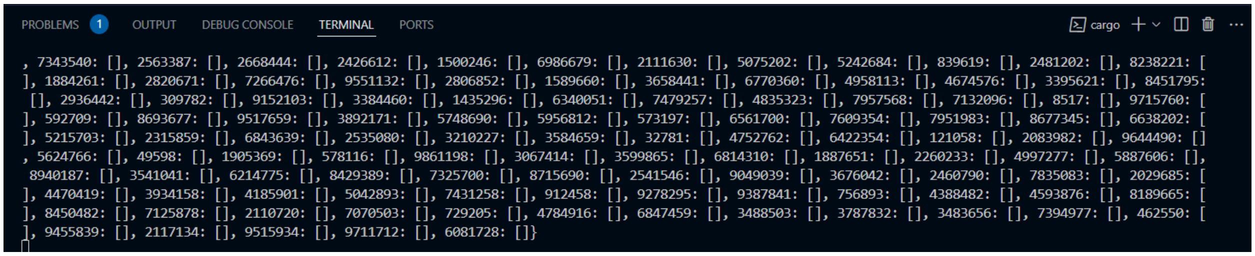
BFS_graph:




```
758768: None, 9645867: None, 9655092: None, 402825: None, 6604970: None, 6544916: None, 3336998: None, 7619742: None, 1913320: None, 3758660: None, 4735465: None, 35
1879: None, 6565346: None, 100137: None, 7112922: None, 8403438: None, 2647684: None, 740899: None, 7926904: None, 5331033: None, 7296928: None, 3692026: None, 21590
26: None, 5654331: None, 7326358: None, 3063198: None, 5073009: None, 6264509: None, 6187808: None, 4406761: None, 3103185: None, 397645: None, 6434884: None, 606169
7: None, 8294259: None, 4197671: None, 493170: None, 1622045: None, 4343590: None, 7465628: None, 4785909: None, 8851019: None, 9038242: None, 9806974: None, 9845342
: None, 3120119: None, 2190205: None, 3029561: None, 4566710: None, 6026886: None, 851488: None, 1655369: None, 4517194: None, 4300988: None, 109688: None, 6062555:
None, 695912: None, 5610431: None, 5107978: None, 9740222: None, 7089049: None, 2410335: None, 5223085: None, 3714071: None, 2353773: None, 1828138: None, 6655805: N
one, 6040514: None, 8830713: None, 4750722: None, 3925065: None, 1658760: None, 1705: None, 8444783: None, 4087281: None, 4548006: None, 122629: None, 6744354: None,
 3866455: None, 2014188: None, 3284212: None, 7928653: None, 1095444: None, 1492544: None, 8386203: None, 8528965: None, 8932942: None, 3633056: None, 132232: None,
6420187: None, 5300640: None, 4105488: None, 4430338: None, 4799447: None, 9054960: None, 9562632: None, 4959728: None, 3263304: None, 4840541: None, 6131205: None,
2696686: None, 3910518: None, 5240200: None, 1857553: None, 4451230: None, 5524276: None, 5790775: None, 9333413: None, 6850087: None, 4334058: None, 243708: None, 7
254378: None, 3866355: None, 8904261: None, 2397096: None, 8982416: None, 7930820: None}
```

Instead of producing a Some output, the code returned None for all of the distances. I did what I could to debug it, but I had no luck in finding a solution to this issue. However, I can say that the adjacency list, which I supplied a portion of in my explanation, compiled correctly and returned the neighbors of each node, and some of the nodes had no neighbors. I theorize that because many nodes did not have neighbors or did not exist because only 41k of about 9 million possible numbers did within the graph, many of the distances were bound to present as None values. The code was compiling very fast and was difficult to read also, so I easily could have missed a rare Some output on the few nodes that had neighbors.

degree_result:



This screenshot is from the first loop with threshold = 1. The same result from BFS_graph carries over into degree_result, where because of the massive number that the graph iterates over, a majority of the distances and shortest paths will amount to 0. Although this may mean the results seem skewed, the large range was necessary because of the amount of large numbers in the data set. When I checked the set myself, many nodes ranged from as low as the 400s to as high as 9600000. Thus, there is a large possibility for large distances between data.

As a result of these results, the average number of citations for each threshold iteration stayed at 0.0. When I first saw these outputs, I was confused because I did not know how the final number could be 0 citations, but after closer inspection, I know why this output resulted. Although it could have been an error, which would have also caused the results in the two pictures above, the most likely answer is because of the large data set. Because we are dividing the number of citations from the small set of nodes that had neighbors and citations by the total number of nodes, the number will be offset by the amount of nodes without any citations. In Rust, f64 division often results in rounding to prevent clutter, so what is actually 0.0000000000000056 may be rounded to 0.0.

<u>Test Cases</u>

I decided to run tests on my distances_bfs function and degree_of_separation functions because they were some of the most used within the program.

To test distances_bfs, I used a small mock data set from ChatGPT that would be manageable in determining distances by hand. Here is the following prompt I input:

You

make a small sample data set to perform a bfs function test on

ChatGPT

Sure, let's create a small graph represented as an adjacency list and perform a BFS function test on it. Here's a simple example:

rust Copy code

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

    #[test]
    fn test_bfs_small_graph() {
        // Define a small graph (adjacency list representation)
        let graph = vec![
            vec![1, 2],  // Vertex 0 is connected to vertices 1 and 2
            vec![0, 3],  // Vertex 1 is connected to vertices 0 and 3
            vec![0, 4],  // Vertex 2 is connected to vertices 0 and 4
            vec![1],     // Vertex 3 is connected to vertex 1
            vec![2],     // Vertex 4 is connected to vertex 2
        ];

        // Choose a starting vertex
        let start_vertex = 0;

        // Call the BFS function
        let distances = distances_bfs(start_vertex, &graph, graph.len());

        // Verify the results
        assert_eq!(distances[0], Some(0));  // Expected distance from vertex 0 to its
        assert_eq!(distances[1], Some(1));  // Expected distance from vertex 0 to ver
        assert_eq!(distances[2], Some(1));  // Expected distance from vertex 0 to ver
        assert_eq!(distances[3], Some(2));  // Expected distance from vertex 0 to ver
        assert_eq!(distances[4], Some(2));  // Expected distance from vertex 0 to ver
    }
}
```

In the test case, we define a vector of vectors- the adjacency list- of all of the neighbors to each node and then define the beginning vertex as Vertex 0. Then, we check each distance and assert that the actual distance calculated by the function is equal to the expected distance. For instance, the distance from Vertex 0 to Vertex 0 is 0 because it's going to itself, while the distance from Vertex 0 to Vertex 3 is 2 since 3 is not a direct neighbor to 0, thus requiring 2 steps to arrive at Vertex 3. As an extra step of verification, we should assert that the adjacency list and graph hold the same lengths, ensuring that no vertex was missed in building.

Here is the result of this test:

```
    Finished test [unoptimized + debuginfo] target(s) in 1.92s
     Running unittests src\main.rs (target\debug\deps\finalproj-73962302c840aaf6.exe)

running 1 test
test test_bfs ... ok

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

The second test tests the degree_of_separation function, which I also turned to ChatGPT for a sample data set since I would not be able to analyze a test of 41k vertices by hand in a reasonable amount of time. Here is the screenshot of my input and the result:

You

how to test with a sample data set
```
pub fn degree_of_separation(graph: Vec<Vec<usize>>, node_count: usize, threshold: Option<u32>) -> HashMap<usize, Vec<Option<u32>>> {
  let initial_vertex = 0;
  let distance_map: HashMap<usize, Option<u32>> = collect_distances(initial_vertex, graph.clone(), node_count);
  //println!("Distance Map: {:?}", distance_map); // Debugging print statement
  let mut shortest_distances: HashMap<usize, Vec<Option<u32>>> = HashMap::new();
  for node in 0..node_count {
    shortest_distances.insert(node, Vec::new());
  }
  println!("Threshold: {:?}", threshold); // Debugging print statement
  for (node, distance) in distance_map.iter() {
    if let Some(dist) = distance {
      if let Some(threshold_value) = threshold {
        if *dist <= threshold_value {
          println!("Adding distance {} for node {}", dist, node); // Debugging print statement
          shortest_distances.get_mut(node).unwrap().push(Some(*dist));
        }
      }
    }
  }
  println!("Shortest Distances: {:?}", shortest_distances); // Debugging print statement
  shortest_distances
}
```

ChatGPT

To test the `degree_of_separation` function with a sample dataset, we need to define a test case where we have a known graph structure and can verify the expected output based on different threshold values.

Here's an example test case:

rust — Copy code

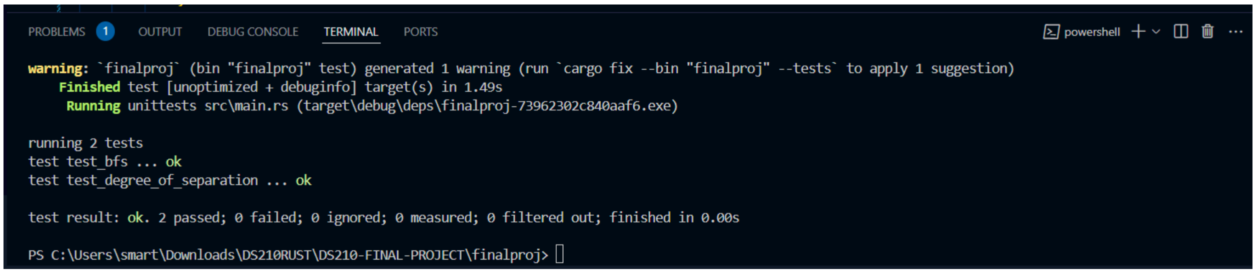
```rust
#[test]
fn test_degree_of_separation() {
    // Define a sample graph
    let graph = vec![
        vec![1, 2],
        vec![0, 3],
        vec![0, 4],
        vec![1],
        vec![2],
    ];
```

Here's an example test case:

rust

```rust
#[test]
fn test_degree_of_separation() {
    // Define a sample graph
    let graph = vec![
        vec![1, 2],
        vec![0, 3],
        vec![0, 4],
        vec![1],
        vec![2],
    ];

    // Define the node count
    let node_count = graph.len();

    // Test with different threshold values
    let thresholds = vec![Some(1), Some(2), Some(3)];

    for threshold in thresholds {
        // Call the degree_of_separation function
        let result = degree_of_separation(graph.clone(), node_count, threshold);

        // Verify the result based on the threshold
        match threshold {
            Some(t) => {
                // For each node in the graph
                for node in 0..node_count {
                    // Check if the shortest distances are within the threshold
                    for &dist in &result[&node] {
                        if let Some(d) = dist {
                            assert!(d <= t, "Distance {} for node {} exceeds threshol
                        }
                    }
                }
            }
            None => {
                // If threshold is None, all distances should be Some(0)
                for node in 0..node_count {
                    assert_eq!(result[&node], vec![Some(0)], "Incorrect distance for
                }
            }
        }
    }
}
```

The code then defines the node count with the graph length and a sample set of thresholds. Like the degree_of_separation function in the degrees module, the code should expect to return the distances that fall under the thresholds. In this case, the thresholds are 1, 2, and 3, so any distance under or equal to the thresholds will be returned. Node_count is then used to iterate over all of the nodes and retrieve the distances. From there, the distances are compared to the thresholds. If any distance is greater than or equal to the threshold, it will be printed as such. If not, then it will be compared to the index of the result. The final piece then asserts if the expected value matches the result. Here are the results of the test:



Cited Sources

I ran into several problems when debugging and thus had to use ChatGPT for much of the debug process. Many of the issues I ran into involved incorrect types and needing to rewrite portions of code to match those types or fix a faulty output. I have explained each of the code blocks in detail to solidify my understanding. As per the university's policy on AI, here are screenshots of my debugging process for the different modules:

In summary, the degree of separation in a directed graph represents the shortest path or minimum number of edges required to travel from one node to another using breadth-first search traversal.

You

how would i code the citations per degree in a situation with this possible output: this node has three direct citations but 12 at degree 2

ChatGPT

To implement a function that calculates the number of citations per degree in a directed graph and generates output similar to "this node has three direct citations but 12 at degree 2," you can follow these steps:

1. Iterate over each node in the graph.
2. For each node, count the number of outgoing edges (direct citations).
3. Iterate over each neighbor of each node to count the number of citations at degree 2.
4. Print the results for each node.

Here's how you can implement this in Rust:

rust Copy code

```rust
use std::collections::HashMap;

type Vertex = usize;
type ListOfEdges = Vec<Vec<Vertex>>;

fn citations_per_degree(graph: &ListOfEdges) {
    // HashMap to store the number of citations per degree
    let mut citations_per_degree: HashMap<usize, usize> = HashMap::new();

    // Iterate over each node in the graph
    for (node_index, node_edges) in graph.iter().enumerate() {
        // Count the number of outgoing edges (direct citations) for the current node
        let direct_citations = node_edges.len();
        println!("Node {}: {} direct citations", node_index, direct_citations);

        // Iterate over each neighbor of the current node
        for neighbor_index in node_edges.iter() {
            // Check if the neighbor index is within the bounds of the graph
            if let Some(neighbor_edges) = graph.get(*neighbor_index) {
                // Count the number of edges (citations) at degree 2
                let degree_2_citations = neighbor_edges.len();
                println!(
                    "    Node {}: {} citations at degree 2",
                    neighbor_index, degree_2_citations
                );

                // Update the count in the citations_per_degree HashMap
                *citations_per_degree.entry(degree_2_citations).or_insert(0) += 1;
            }
        }
```

Message ChatGPT

any further questions or issues, feel free to ask!

You

still getting errors here: for threshold in 1..=5 {
let mut degree_result = degree_of_separation(&adjacency_list, node_count, threshold);
println!("{:?}", degree_result);
let mut average_citations = average_number_citations(&adjacency_list);
//println!("{:?}",)
}
let mut shortest_distance_pairs: HashMap<usize,usize> = degree_of_separation(bfs_graph, node_count, threshold);

ChatGPT

The error you're encountering is because you're attempting to use the `threshold` variable outside the loop where it's defined. The `threshold` variable is scoped within the loop, and it's not accessible outside of it.

To fix this, you need to move the declaration of `threshold` outside the loop so that it's accessible when you use it to call the `degree_of_separation` function after the loop.

Here's the corrected version:

```rust
// Define threshold variable outside the loop
let threshold = 5;

// Calculate degree of separation and average citations for different thresholds
for threshold in 1..=5 {
    let degree_result = degree_of_separation(&adjacency_list, node_count, threshold);
    println!("Degree result for threshold {}: {:?}", threshold, degree_result);

    let average_citations = average_number_citations(&adjacency_list);
    println!("Average citations: {}", average_citations);
}

// Call degree_of_separation function after the loop
let shortest_distance_pairs: HashMap<usize, Vec<usize>> = degree_of_separation(&bfs_g
```

In this corrected version:

- I moved the declaration of `threshold` outside the loop, ensuring it's accessible when calling the `degree_of_separation` function after the loop.
- I removed the `mut` qualifier from `degree_result` and `average_citations` variables since they are not being mutated within the loop.
- I corrected the parameter type for `shortest_distance_pairs` to match the return type of the `degree_of_separation` function.

With these changes, the code should compile without errors. If you have any further questions or issues, feel free to ask!

SM **You**

debug the shortest distances function- errors at output initialization, call of collect_distances function, and insides of the for loop at the end: pub fn degree_of_separation(graph: Vec<Vec<usize>>, node_count: usize, threshold: usize) -> HashMap<usize, Vec<usize>> {

```
    let mut distance_map: HashMap<usize,usize> = collect_distances(graph, node_count);
    let mut shortest_distances: HashMap<usize, Vec<usize>> = HashMap::new();
    for (node, distance) in distance_map.iter() {
        if *distance <= threshold {
            shortest_distances[node].push(distance);
        }
    }
    shortest_distances;
}
```

ChatGPT

The function `degree_of_separation` has a few issues that need to be addressed:

1. **Output Initialization:** The `shortest_distances` hashmap needs to be properly initialized with an empty vector for each key before pushing distances into it.
2. **Calling `collect_distances` Function:** Ensure that you are calling the `collect_distances` function correctly and passing the correct parameters.
3. **Pushing Distances into `shortest_distances`:** You should push the `distance` into the vector associated with the `node` key, not the distance itself.

Here's the corrected version of the function:

rust Copy code

```rust
use std::collections::HashMap;

pub fn degree_of_separation(graph: Vec<Vec<usize>>, node_count: usize, threshold: usi
    // Call collect_distances function and collect the distances
    let distance_map: HashMap<usize, usize> = collect_distances(graph.clone(), node_c

    // Initialize shortest_distances hashmap with an empty vector for each key
    let mut shortest_distances: HashMap<usize, Vec<usize>> = HashMap::new();
    for node in 0..node_count {
        shortest_distances.insert(node, Vec::new());
    }

    // Iterate over distance_map and populate shortest_distances
    for (node, distance) in distance_map.iter() {
        if *distance <= threshold {
            shortest_distances.get_mut(node).unwrap().push(*distance);
        }
    }

    shortest_distances
}
```

Message ChatGPT

You
heres what distances_bfs looks rn- how to fix but keep it somewhat the same: pub fn distances_bfs(initial_vertex: Vertex, graph: Vec<Vec<usize>>, node_count: usize) -> Vec<Option<u32>> {

ChatGPT
If you need to keep the signature of the `distances_bfs` function the same, but the implementation returns a vector of vectors of `usize`, you can make adjustments to the implementation to convert the distances to `Option<u32>` before returning them.

Here's a potential approach to adjust the implementation while keeping the function signature the same:

rust Copy code

```rust
pub fn distances_bfs(initial_vertex: Vertex, graph: Vec<Vec<usize>>, node_count: usiz
    let mut distance: Vec<Option<u32>> = vec![None; node_count];
    let mut queue: VecDeque<Vertex> = VecDeque::new();
    let mut visited: Vec<bool> = vec![false; node_count];

    // Initialize the queue with the initial vertex
    queue.push_back(initial_vertex);
    visited[initial_vertex] = true;
    distance[initial_vertex] = Some(0);

    // Perform BFS
    while let Some(v) = queue.pop_front() {
        for &u in &graph[v] {
            if !visited[u] {
                visited[u] = true;
                distance[u] = distance[v].map(|dist| dist + 1); // Convert distance t
                queue.push_back(u);
            }
        }
    }

    distance
}
```

In this implementation, we're still returning `Vec<Option<u32>>`, but we're calculating the distances as `usize` and then converting them to `Option<u32>` before assigning them to the `distance` vector. This way, the function signature remains the same, but the distances are calculated and returned correctly.

You
errors with distances bfs and last print statement: fn main() {
let data = read_file("Cit-HepPh.txt");
let num_nodes = data.0;
let list_edges = data.1;
let adjacency_list = adj_list(&list_edges, num_nodes);

Message ChatGPT

You

Error parsing node count: invalid digit found in string
Error reading file: invalid digit found in string this happened

ChatGPT

The error message "invalid digit found in string" indicates that the code attempted to parse a string into an integer, but the string contained characters that are not valid digits. This can happen if the input file contains unexpected characters, such as non-numeric characters or whitespace.

To troubleshoot this issue, you should inspect the input file to ensure it is formatted correctly and that each line contains valid numeric data. Here are some steps you can take:

1. **Check the Input File**: Open the input file (`Cit-HepPh.txt`) and verify that each line contains only numeric values separated by whitespace. Ensure there are no extra characters, such as letters or special symbols, that might cause parsing errors.
2. **Handle Unexpected Input**: Update your code to handle unexpected input gracefully. For example, you can print a warning message when encountering invalid lines or skip them altogether.

Here's an updated version of the `read_file` function that handles invalid lines by printing a warning message and skipping them:

rust Copy code

```rust
fn read_file(path: &str) -> Result<(usize, Vec<(usize, usize)>), Box<dyn std::error::
    let mut node_count = 0;
    let mut edge_list: Vec<(usize, usize)> = Vec::new();
    let file = File::open(path)?;
    let buf_reader = BufReader::new(file);
    for (index, line) in buf_reader.lines().enumerate() {
        let line_str = line?;
        let v: Vec<&str> = line_str.trim().split(' ').collect();
        if index == 0 {
            node_count = match v[0].parse::<usize>() {
                Ok(value) => value,
                Err(err) => {
                    eprintln!("Error parsing node count: {}", err);
                    return Err(Box::new(err));
                }
            };
        } else if v.len() >= 2 {
            if let (Ok(x), Ok(y)) = (v[0].parse::<usize>(), v[1].parse::<usize>()) {
                edge_list.push((x, y));
            } else {
                eprintln!("Warning: Skipping invalid line {}", index + 1);
            }
        } else {
            eprintln!("Warning: Skipping incomplete line {}", index + 1);
        }
    }
    Ok((node_count, edge_list))
}
```

Message ChatGPT

This updated implementation of the `read_file` function should correctly parse the sample data and construct the edge list. Make sure to replace the existing `read_file` function with this updated version and rerun your program to see if it resolves the out-of-bounds errors.

SM

You

okay none of the edges were out of bounds but this started getting outputted: Node 65572: []
Node 65573: []
Node 65574: []
Node 65575: []
Node 65576: []
Node 65577: []
Node 65578: []
Node 65579: []

ChatGPT

It seems like the size of your adjacency list is not matching the number of nodes in your graph. This can lead to uninitialized or empty lists for nodes that don't have any edges.

The issue likely lies in how you determine the size of the adjacency list in your `adj_list` function. Here's a revised version of the function that ensures the adjacency list has the correct size:

rust Copy code

```rust
fn adj_list(my_vec: &Vec<(usize, usize)>, node_count: usize) -> Vec<Vec<usize>> {
    let mut adjacency_list: Vec<Vec<usize>> = vec![Vec::new(); node_count];

    for &(i, j) in my_vec.iter() {
        if i < node_count && j < node_count {
            // Extend the adjacency list if necessary
            while adjacency_list.len() <= i || adjacency_list.len() <= j {
                adjacency_list.push(Vec::new());
            }
            adjacency_list[i].push(j);
        } else {
            eprintln!("Warning: Edge ({}, {}) is out of bounds", i, j);
        }
    }

    // Print adjacency list for debugging
    for (i, row) in adjacency_list.iter().enumerate() {
        println!("Node {}: {:?}", i, row);
    }

    adjacency_list
}
```

In this version, we iterate through the edge list `(i, j)` and ensure that both `i` and `j` are within the range of `node_count`. Additionally, we extend the adjacency list to accommodate any indices larger than the current size. Finally, we print the adjacency list for debugging purposes.

Replace your current `adj_list` function wit... is updated version and rerun your program to see if it resolves the issue.

Message ChatGPT

iterating over different thresholds. Let me know if you need further clarification or assistance!

SM **You**

here is another module which i am trying to debug and write a piece of code for based on the instructions in comments. this is the degrees module: // find length of each list of connections from bfs

```
use std::collections::HashMap;

mod bfs;
use crate::distances_bfs::{};

pub fn degree_of_separation(distance_map: &HashMap<usize, usize>, threshold: usize) ->
HashMap<usize,Vec<usize>> {
  let mut distance_map: HashMap<usize,usize> = collect_distances(graph: Vec<Vec<usize>>,
node_count);
  let mut shortest_distances: HashMap<usize,usize> = HashMap::new();
  for (x,y) in distance_map {
    if y <= threshold {
      shortest_distances[x].push(y);
    }
  }
  return shortest_distances;
}

fn average_num_citations
// find number of citations per node at each degree
// this node has three direct citations but 12 at 2

// take avg over all nodes

// find length of vector for shortest_distances
//do for loop of different thresholds
```

ChatGPT

Let's break down the instructions and debug the `degree_of_separation` function accordingly:

1. **Find length of each list of connections from BFS**: This indicates that we need to use the `distances_bfs` function to get a map of distances from a starting node to all other nodes. We'll then collect these distances into a HashMap.
2. **Take average over all nodes**: This suggests calculating the average number of citations per node at each degree. This requires some additional logic to process the obtained distances.
3. **Find length of vector for shortest_distances**: This asks to find the length of the vector containing the shortest distances for different thresholds.

Here's the revised `degree_of_separation` function with explanations:

```rust
use std::collections::HashMap;

// Importing the distances_bfs funct[...] from the bfs module
mod bfs;
```

ChatGPT

Let's break down the instructions and debug the `degree_of_separation` function accordingly:

1. **Find length of each list of connections from BFS**: This indicates that we need to use the `distances_bfs` function to get a map of distances from a starting node to all other nodes. We'll then collect these distances into a HashMap.
2. **Take average over all nodes**: This suggests calculating the average number of citations per node at each degree. This requires some additional logic to process the obtained distances.
3. **Find length of vector for shortest_distances**: This asks to find the length of the vector containing the shortest distances for different thresholds.

Here's the revised `degree_of_separation` function with explanations:

rust   Copy code

```rust
use std::collections::HashMap;

// Importing the distances_bfs function from the bfs module
mod bfs;
use crate::bfs::distances_bfs;

pub fn degree_of_separation(graph: Vec<Vec<usize>>, node_count: usize, threshold: usi
    // Getting the distances map using BFS
    let distance_map = collect_distances(&graph, node_count);

    // HashMap to store nodes within threshold distance
    let mut shortest_distances: HashMap<usize, Vec<usize>> = HashMap::new();

    // Iterating over the distance map to collect nodes within threshold distance
    for (node, distance) in distance_map.iter() {
        if *distance <= threshold {
            // If the distance is within threshold, insert it into shortest_distances
            shortest_distances.entry(*distance).or_insert_with(Vec::new).push(*node);
        }
    }

    // Calculating average number of citations per node at each degree
    let mut avg_citations_per_degree: HashMap<usize, f64> = HashMap::new();
    for distances in shortest_distances.values() {
        // Counting the number of citations at each degree
        let citations_at_degree: usize = distances.iter().map(|&x| graph[x].len()).su
        // Calculating average citations per degree
        let avg_citations = citations_at_degree as f64 / distances.len() as f64;
        avg_citations_per_degree.insert(distances.len(), avg_citations);
    }

    // Find the length of the vector for shortest_distances
    let shortest_distances_length = shortest_distances.len();

    // Debugging print statements to print average citations per degree
    println!("Average citations per degree:");
    for (degree, avg_citations) in a citations_per_degree.iter() {
        println!("Degree {}: {}", degree, avg_citations);
```

Message ChatGPT