I chose the facebook dataset as I find social media interesting and it plays an increasingly larger role in our lives as time goes on. I also was curious to analyze the social connectivity of those on facebook. Especially concerning the theory of six degrees of separation which I have always found especially fascinating, which is the idea that all people are connected through six social connections.[1] This idea can be extended quite easily to social networks such as facebook as well. I created a module that was able to read in the graph data and make it into an adjacency list and included the function get_neighbors which gets the neighbors of a specific vertex. The data I used was an undirected graph given the nature of facebook where you have friends and there are no one way connections. Each line in the text file consists of one edge, with a node and a node it is connected to. Since it is an unweighted and undirected graph, breadth-first search will work quite well for finding the shortest path between every node, as long as you do breadth-first search starting at every node.[2] I made a function that counted the unique nodes as originally the adjacency list created by the Graph struct was not considering target vertices listed in the graph dataset as keys themselves. This caused my results to be inaccurate. The adjacency list thus had a length of about 3,600 which included only the source vectors in the dataset. However the unique nodes function listed the correct amount of nodes at 4,039 which is consistent with the data I was using. However since my dataset is undirected, both the source and target vectors should be included as keys in the adjacency list for the rest of my functions to work, since they operate on the assumption that every vector will have its own key in the adjacency list. The total unique nodes function is quite simple and works by iterating through the adjacency list and adding each

---

1 https://en.wikipedia.org/wiki/Six_degrees_of_separation#:~:text=Six%20degrees%20of%20separation%20is,as%20the%20six%20handshakes%20rule.

2 ChatGPT, Professor Leonidas Kontothanassis

key and value which all represent the vectors and their neighbors to a hash set. Using a hash set avoids duplicate entries since a hash set will only accept unique entries.

```
fn compute_distances_from_vertex(start: i32, graph: &Graph) -> HashMap<i32, u32> {
   let mut distances = HashMap::new();
   let mut queue = VecDeque::new();

   distances.insert(start, 0);
   queue.push_back(start);
/*  The below uses a while loop that assigns the variable current to whatever vertex
is popped from
the front */
   while let Some(current) = queue.pop_front() {
       let current_distance = distances[&current];
       if let Some(neighbors) = graph.get_neighbors(current) {
           for &neighbor in neighbors {
               if !distances.contains_key(&neighbor) {
                   distances.insert(neighbor, current_distance + 1);
                   queue.push_back(neighbor);
               }
           }
       }
   }

   distances
}
```

The breadth first search algorithm used in the code works by making a VecDeque and starting with the first node, which is an argument given to the function, adding each of the neighbors and their distance; Which is calculated by recursively adding 1 after each level of neighbors is surpassed. The function goes through all of the neighbors adding them to the back of the queue so they are examined (going through their neighbors in the same way) on a first in first out basis.

The compute all distances function simply goes through every key in the adjacency list and starts a BFS search from that key.

```
/// Calculates the number of connections each vertex has.
fn calculate_connection_counts(distances: &HashMap<i32, HashMap<i32, u32>>) ->
HashMap<i32, usize> {
    distances.iter().map(|(&vertex, dist_map)| (vertex, dist_map.len())).collect()
}
```

Calculate connection counts uses an iterable and maps over it with a closure that iterates through every key value pair in the distances hashmap it then calculates the length of the distances vector in the hashmap and sets that number as the value representing the number of connections of the key which represents the node in the new hashmap.

```
fn find_largest_web_within_six_degrees(distances: &HashMap<i32, HashMap<i32, u32>>) ->
(Option<i32>, usize) {
    let mut max_web_size = 0;
    let mut vertex_with_max_web = None;

    for (&vertex, dist_map) in distances {
        let web_size = dist_map.values().filter(|&&d| d <= 6).count();
        if web_size > max_web_size {
            max_web_size = web_size;
            vertex_with_max_web = Some(vertex);
        }
    }

    (vertex_with_max_web, max_web_size)
}
```

The find_largest_web_within_six_degrees function takes the distances from the breadth first search function defined earlier and iterates through the nested hashmap of the distances using a filter to count the connections that are within 6 degrees. When it encounters a web that is larger than the current largest one it assigns it to the variable max_web and records the variable. Thus by the end of the for loop we will have found the max web and the vertex connected to it.

```rust
/// Identifies the mode of the connection counts.
fn find_mode_vertex(connection_counts: &HashMap<i32, usize>) -> Option<usize> {
   let mut frequency = HashMap::new();
   let mut max_freq = 0;
   let mut mode_vertex = None;

   for &count in connection_counts.values() {
       let count_freq = frequency.entry(count).or_insert(0);
       *count_freq += 1;

       if *count_freq > max_freq {
           max_freq = *count_freq;
           mode_vertex = Some(count);
       }
   }

   mode_vertex
}
```

Find mode vertex uses a for loop to iterate through connection_counts and figures out which connection count appears the most often,  and assigns that connection amount as the mode vertex, short for mode vertex connections.

```rust
pub fn analyze_graph(graph: &Graph) {
   let all_distances = compute_all_distances(graph);
   let connection_counts = calculate_connection_counts(&all_distances);
   let (vertex_with_max_web, max_web_size) =
find_largest_web_within_six_degrees(&all_distances);
   let mode_vertex = find_mode_vertex(&connection_counts);

   println!("Vertex with the largest web within 6 degrees: {:?}",
vertex_with_max_web);
   println!("Size of the largest web within 6 degrees: {}", max_web_size);
   println!("Mode vertex by number of connections: {:?}", mode_vertex);
}
```

Analyze graph simply uses the functions defined before and prints the results. This function is useful as it allows the user to get multiple different analyses of the graph without having to call multiple functions.

```
pub fn print_top_vertices_by_neighbors<W: Write>(graph: &Graph, top_n: usize, writer:
&mut W) -> io::Result<()> {
   let mut neighbor_counts: Vec<(i32, usize)> = graph.adjacency_list.iter()
       .map(|(vertex, neighbors)| (*vertex, neighbors.len()))
       .collect();

   neighbor_counts.sort_by(|a, b| b.1.cmp(&a.1));
   let top_vertices = neighbor_counts.iter().take(top_n);

   writeln!(writer, "Top {} vertices by number of neighbors:", top_n)?;
   for (vertex, count) in top_vertices {
       writeln!(writer, "Vertex {}: {} neighbors", vertex, count)?;
   }
   Ok(())
}
```

This function uses map on an iterable like before this time it uses sort by which is a function that sorts neighbor_counts. The function is passed a writer so that it can print directly to the console and be tested more easily . The sort function is sorted in descending order as shown by `(|a, b| b.1.cmp(&a.1))`. Then the function uses the writer to print the top n vertices.

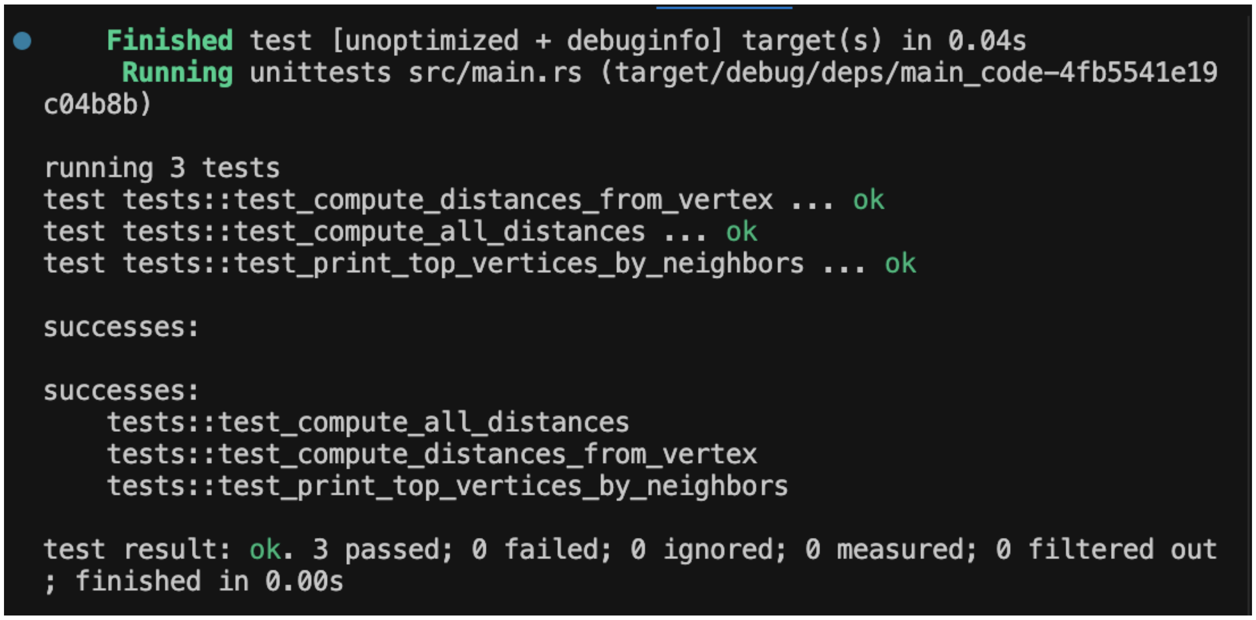
```
fn analyze_six_degrees(all_distances: &HashMap<i32, HashMap<i32, u32>>) ->
(HashMap<i32, usize>, f64) {
   let num_vertices = all_distances.len() as f64;
   let mut within_six_degrees = HashMap::new();
   let mut total_within_six = 0;
   for (vertex, distances) in all_distances {
       let count = distances.values().filter(|&&d| d <= 6 && d > 0).count();
       within_six_degrees.insert(*vertex, count);
       total_within_six += count;
   }
```

```
    let average_percentage = total_within_six as f64 / num_vertices / num_vertices *
100.0;
    (within_six_degrees, average_percentage)
}
```

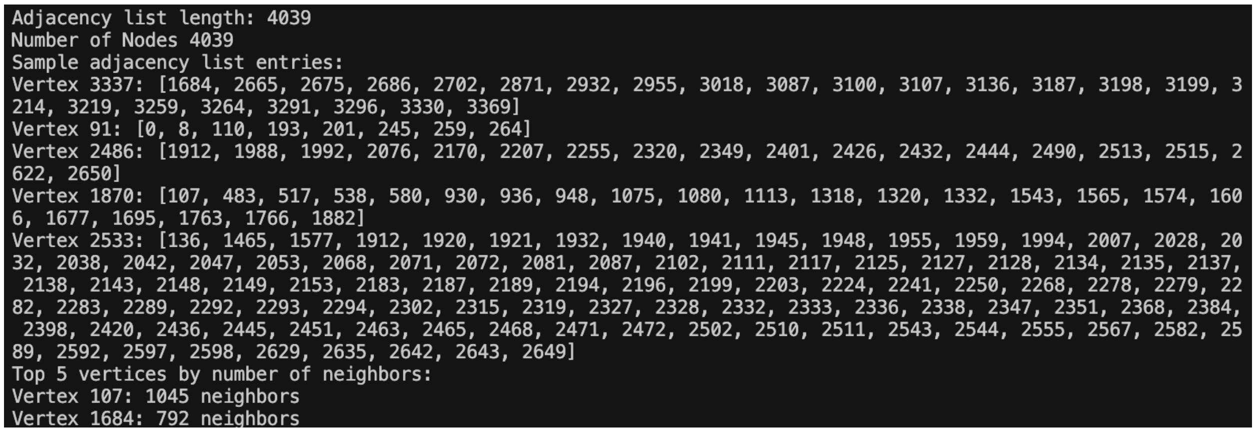
This function iterates through the all_distances hashmap and counts only the vectors within 6 degrees. It uses that information to create the new hashmap that the function will return which contains each vector and the count for how many vectors within 6 degrees it contains. It also returns an average percentage of the total nodes in the graph that our within each nodes web of 6 degrees.

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

    #[test]
    fn test_print_top_vertices_by_neighbors() {
        let mut graph = Graph {
            adjacency_list: HashMap::new(),
        };

        // Setup a simple graph
        graph.adjacency_list.insert(0, vec![1, 2, 3]);
        graph.adjacency_list.insert(1, vec![0, 2]);
        graph.adjacency_list.insert(2, vec![0, 1]);
        graph.adjacency_list.insert(3, vec![0]);

        let mut output = vec![];
        print_top_vertices_by_neighbors(&graph, 2, &mut output).unwrap();

        let output_str = str::from_utf8(&output).unwrap();
        assert!(output_str.contains("Top 2 vertices by number of neighbors:"));
        assert!(output_str.contains("Vertex 0: 3 neighbors"));
        assert!(output_str.contains("Vertex 1: 2 neighbors"));
    }
    #[test]
    fn test_compute_all_distances() {
        let mut graph = Graph {
            adjacency_list: HashMap::new(),
        };
        // Creating an undirected graph: 0 - 1 - 2 (both directions)
        graph.adjacency_list.insert(0, vec![1]);
        graph.adjacency_list.insert(1, vec![0, 2]);
        graph.adjacency_list.insert(2, vec![1]);

        let distances = compute_all_distances(&graph);

        assert_eq!(distances[&0][&1], 1); // Check distance from 0 to 1
        assert_eq!(distances[&1][&2], 1); // Check distance from 1 to 2
        assert_eq!(distances[&0][&2], 2); // Check distance from 0 to 2 through 1
        assert_eq!(distances[&2][&0], 2); // Check distance from 2 to 0 through 1
    }

    #[test]
    fn test_compute_distances_from_vertex() {
        let mut graph = Graph {
            adjacency_list: HashMap::new(),
        };
        // Creating an undirected graph: 0 - 1 - 2 - 3 (chain)
        graph.adjacency_list.insert(0, vec![1]);
        graph.adjacency_list.insert(1, vec![0, 2]);
        graph.adjacency_list.insert(2, vec![1, 3]);
        graph.adjacency_list.insert(3, vec![2]);

        let distances = compute_distances_from_vertex(0, &graph);

        assert_eq!(distances[&1], 1); // Direct neighbor
        assert_eq!(distances[&2], 2); // Two hops away
        assert_eq!(distances[&3], 3); // Three hops away
    }
}
```

These tests simply use sample graphs that we already know the correct output for and compare the correct output with what our functions will output.

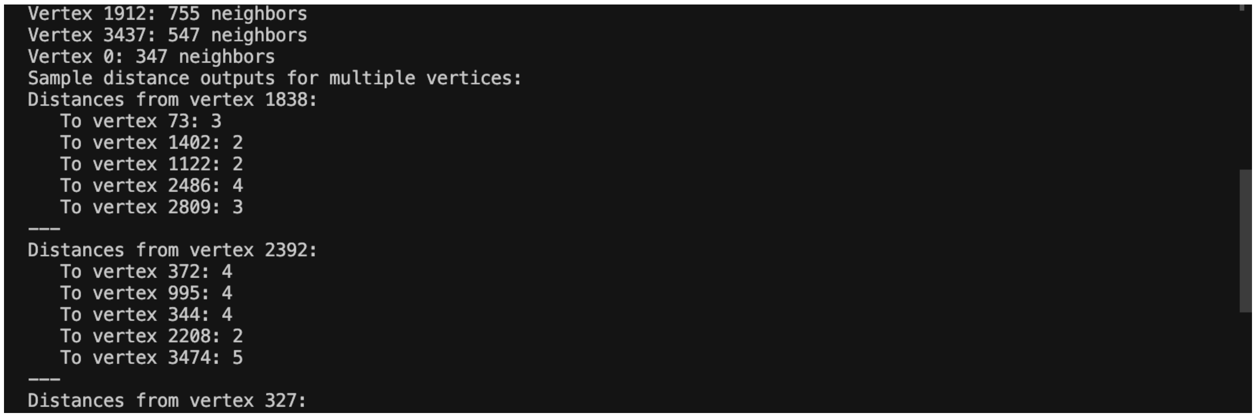


```
●     Finished test [unoptimized + debuginfo] target(s) in 0.04s
       Running unittests src/main.rs (target/debug/deps/main_code-4fb5541e19
  c04b8b)

  running 3 tests
  test tests::test_compute_distances_from_vertex ... ok
  test tests::test_compute_all_distances ... ok
  test tests::test_print_top_vertices_by_neighbors ... ok

  successes:

  successes:
      tests::test_compute_all_distances
      tests::test_compute_distances_from_vertex
      tests::test_print_top_vertices_by_neighbors

  test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
  ; finished in 0.00s
```



```
Adjacency list length: 4039
Number of Nodes 4039
Sample adjacency list entries:
Vertex 3337: [1684, 2665, 2675, 2686, 2702, 2871, 2932, 2955, 3018, 3087, 3100, 3107, 3136, 3187, 3198, 3199, 3
214, 3219, 3259, 3264, 3291, 3296, 3330, 3369]
Vertex 91: [0, 8, 110, 193, 201, 245, 259, 264]
Vertex 2486: [1912, 1988, 1992, 2076, 2170, 2207, 2255, 2320, 2349, 2401, 2426, 2432, 2444, 2490, 2513, 2515, 2
622, 2650]
Vertex 1870: [107, 483, 517, 538, 580, 930, 936, 948, 1075, 1080, 1113, 1318, 1320, 1332, 1543, 1565, 1574, 160
6, 1677, 1695, 1763, 1766, 1882]
Vertex 2533: [136, 1465, 1577, 1912, 1920, 1921, 1932, 1940, 1941, 1945, 1948, 1955, 1959, 1994, 2007, 2028, 20
32, 2038, 2042, 2047, 2053, 2068, 2071, 2072, 2081, 2087, 2102, 2111, 2117, 2125, 2127, 2128, 2134, 2135, 2137,
 2138, 2143, 2148, 2149, 2153, 2183, 2187, 2189, 2194, 2196, 2199, 2203, 2224, 2241, 2250, 2268, 2278, 2279, 22
82, 2283, 2289, 2292, 2293, 2294, 2302, 2315, 2319, 2327, 2328, 2332, 2333, 2336, 2338, 2347, 2351, 2368, 2384,
 2398, 2420, 2436, 2445, 2451, 2463, 2465, 2468, 2471, 2472, 2502, 2510, 2511, 2543, 2544, 2555, 2567, 2582, 25
89, 2592, 2597, 2598, 2629, 2635, 2642, 2643, 2649]
Top 5 vertices by number of neighbors:
Vertex 107: 1045 neighbors
Vertex 1684: 792 neighbors
```

The main function prints the adjacency list length and number of nodes which should be the same if my code is running correctly. I had it print some sample adjacency list entries to get a visual confirmation that the graph module was correctly reading in the data. It printed the top 5 vertices by number of neighbors, this data was not too surprising although having 1,000 friends is impressive for the top vertex. The individual vertex connectivity within 6 degrees works as expected based on the sample output The sample distance output shows what is expected. The average vertex can reach almost the entire graph with 6 degrees, this makes sense given the interconnectivity of social networks. Starting from some vectors the entire graph can be reached by 6 degrees, however this does not necessarily mean that the entire graph is within 6 degrees of each other. For example if vertex 1 was 6 degrees in one direction from vertex 2 and 6 degrees in a completely different direction from vertex 3, the function would show all of them being in the web of 6 degrees but vertex 3 and 2 would be shown as 12 degrees apart. Most vertices were connected to the whole graph based on the mode number of connections being 4039 (the amount of available nodes). It is a bit surprising that none of the nodes formed islands that were not connected from the rest of this graph, further analysis would have to be done to know why this is.[3] My code demonstrates the interconnectivity of the facebook data given through the graph.txt. Further analysis could be done on the connectivity for example how much of the graph can be covered in 5, 4 and so on degrees starting from one vector, or what is the minimum amount of degrees required to interconnect the whole graph.



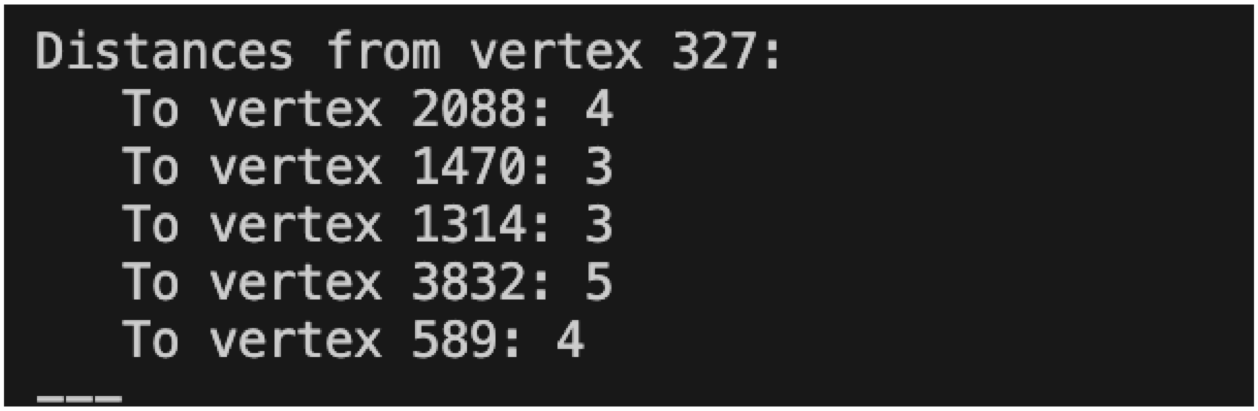
```
Vertex 1912: 755 neighbors
Vertex 3437: 547 neighbors
Vertex 0: 347 neighbors
Sample distance outputs for multiple vertices:
Distances from vertex 1838:
   To vertex 73: 3
   To vertex 1402: 2
   To vertex 1122: 2
   To vertex 2486: 4
   To vertex 2809: 3
---
Distances from vertex 2392:
   To vertex 372: 4
   To vertex 995: 4
   To vertex 344: 4
   To vertex 2208: 2
   To vertex 3474: 5
---
Distances from vertex 327:
```

Chatgpt was used to write the code and help my understanding of the code:

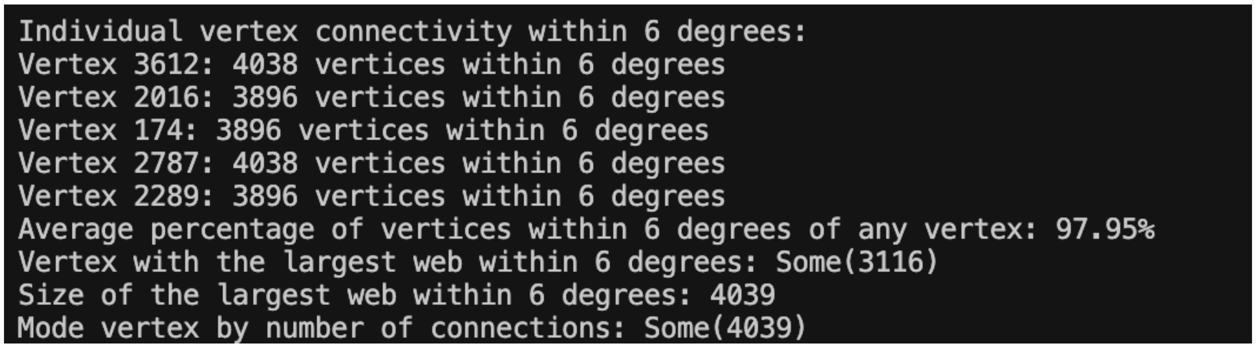
https://chat.openai.com/share/d74a1282-a7eb-4c85-881d-449e7fa9c975

https://chat.openai.com/share/fc5fa955-b4ea-444d-8da9-c36ad4de5944n

[3] ChatGPT https://chat.openai.com/share/d74a1282-a7eb-4c85-881d-449e7fa9c975



```
Distances from vertex 327:
   To vertex 2088: 4
   To vertex 1470: 3
   To vertex 1314: 3
   To vertex 3832: 5
   To vertex 589: 4
---
```



```
Individual vertex connectivity within 6 degrees:
Vertex 3612: 4038 vertices within 6 degrees
Vertex 2016: 3896 vertices within 6 degrees
Vertex 174: 3896 vertices within 6 degrees
Vertex 2787: 4038 vertices within 6 degrees
Vertex 2289: 3896 vertices within 6 degrees
Average percentage of vertices within 6 degrees of any vertex: 97.95%
Vertex with the largest web within 6 degrees: Some(3116)
Size of the largest web within 6 degrees: 4039
Mode vertex by number of connections: Some(4039)
```