Peter Emero

Professor Chator

DS 210

1 May 2024

Final Project Writeup

The first aspect of my project that is important to discuss is my data and how I formatted it. I used data from https://www.kaggle.com/datasets/davidcariboo/player-scores/data. Specifically, the dataset came in multiple parts. I merged them to create a CSV with the 5 metrics I needed to find similarities: **total goals, home goals, away goals, minute of event, and description**. Additionally, I included game_id to make sure I could tell which game is which, especially for reading the data. I did the reading in the following function:

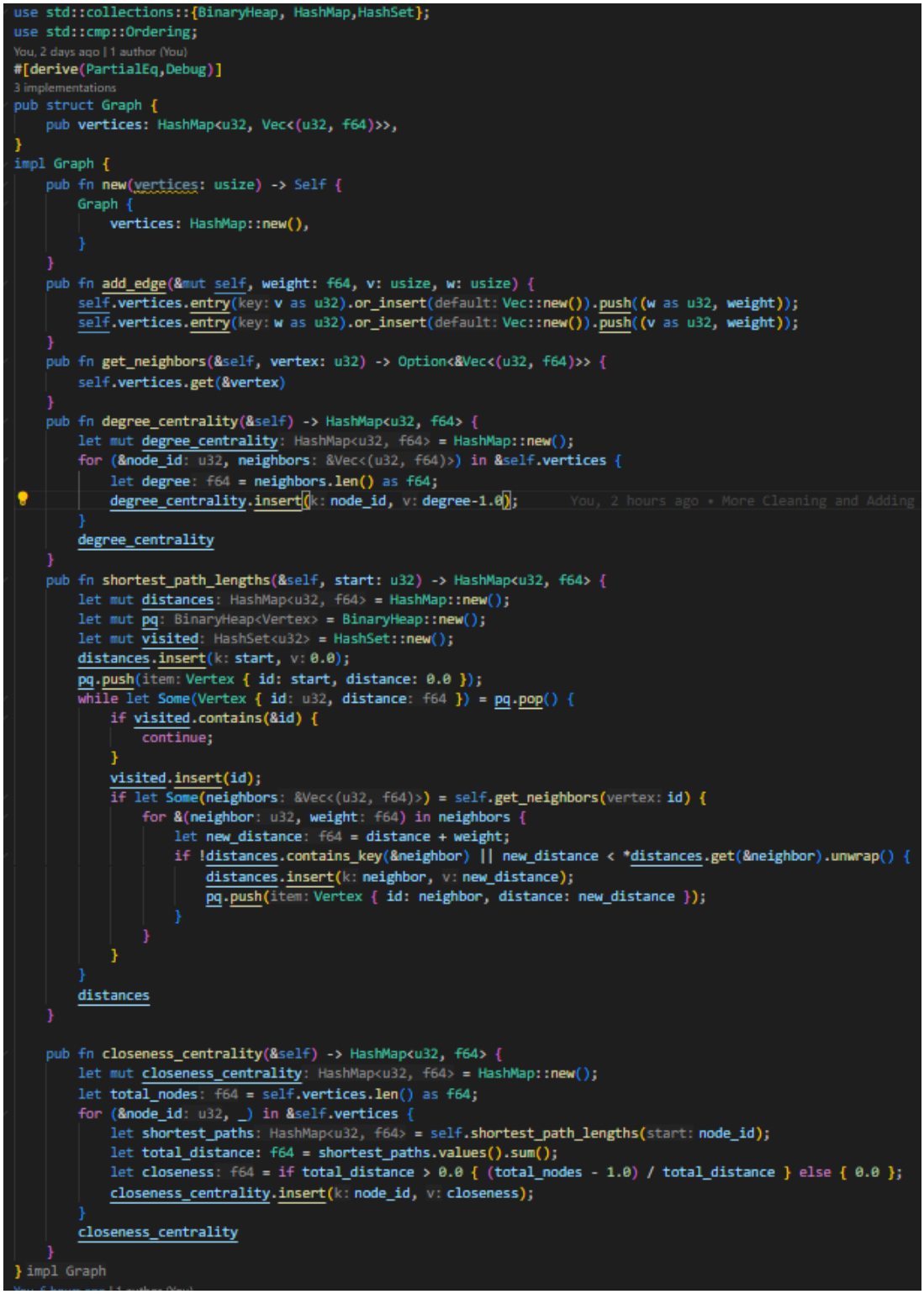
```rust
fn read_data(path: &str) -> HashMap<u64, Vec<GameEntry>> {
    let mut result: HashMap<u64, Vec<GameEntry>> = HashMap::new();
    let file: File = File::open(path).expect(msg: "Couldn't Open");
    let mut buf_reader: Lines<BufReader<File>> = std::io::BufReader::new(inner: file).lines();
    buf_reader.next();
    let mut prev_game_id: u64 = 100000000;
    let mut node_id: u64 = 0;
    for line: Result<String, Error> in buf_reader {
        let line_str: String = line.expect(msg: "Error Reading");
        let events: Vec<String> = line_str.split(",").map(|s: &str| s.to_string()).collect();
        let game_id: u64 = u64::from_str(&events[0]).unwrap();
        if prev_game_id != game_id {
            node_id += 1;
        }
        prev_game_id=game_id;
        let game_entry: GameEntry = GameEntry {
            node_id,
            game_id: u64::from_str(&events[0]).unwrap(),
            minute: f64::from_str(&events[1]).unwrap(),
            description: events[5].clone(),
            home_club_goals: u64::from_str(&events[13]).unwrap(),
            away_club_goals: u64::from_str(&events[14]).unwrap(),
            total_goals: u64::from_str(&events[13]).unwrap() + u64::from_str(&events[14]).unwrap()
        };
        result.entry(key: game_entry.node_id).or_insert(default: Vec::new()).push(game_entry.clone());
        if node_id == 1000{
            break
        }
    }
    result
```

Before explaining the function, it is important to understand the output. I am returning a HashMap of every event (the CSV file had one event per line) in a vector associated with a node_id. I decided to use node_id instead to make it easier to deal with missing values or rows that had not been added, which prevented future errors in other functions. Additionally, I created a **GameEntry** struct see below:

```
#[derive(Debug,Clone)]
2 implementations
struct GameEntry {
    node_id: u64,
    game_id: u64,
    minute: f64,
    description: String,
    home_club_goals: u64,
    away_club_goals: u64,
    total_goals: u64,
}
```

In the above screenshot of my read_data function, you can see that I parsed each line by line separating the values into vectors and only storing the important information needed for the similarity function. After storing the game entries in a vector based on the game_id and node_id I check to see if I have reached the required 1000 nodes. The dataset has tens of thousands of events, however because of the extreme size of the file it is not possible to run the entire file on my laptop. Therefore, I decided to cut the number of games to the minimum to gain valuable insight into games as a whole and what makes a game “the most similar” without struggling through long processing times. Especially considering there is not a huge amount to be gained by adding so many more games.

The next important part of my code is how I created the graph. I created a separate module for my Graph struct and its associated methods as seen below:

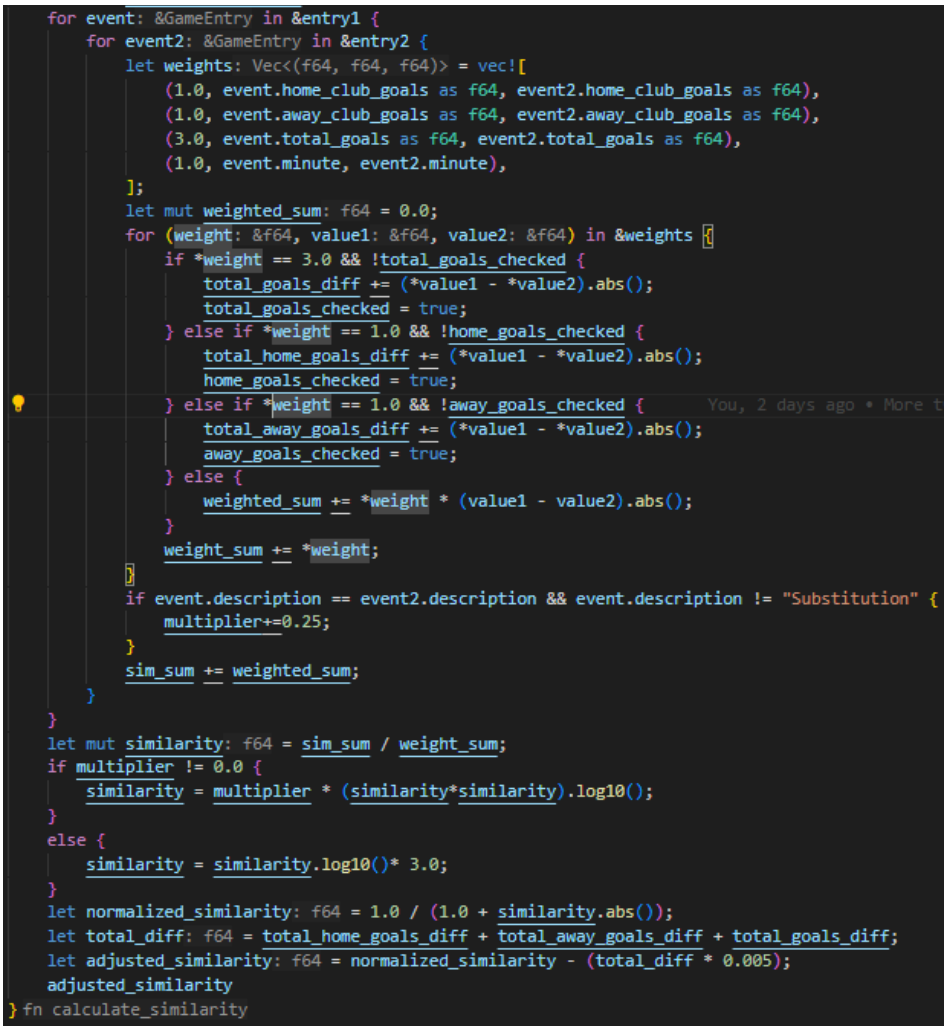


```rust
use std::collections::{BinaryHeap, HashMap,HashSet};
use std::cmp::Ordering;
You, 2 days ago | 1 author (You)
#[derive(PartialEq,Debug)]
3 implementations
pub struct Graph {
    pub vertices: HashMap<u32, Vec<(u32, f64)>>,
}
impl Graph {
    pub fn new(vertices: usize) -> Self {
        Graph {
            vertices: HashMap::new(),
        }
    }
    pub fn add_edge(&mut self, weight: f64, v: usize, w: usize) {
        self.vertices.entry(key: v as u32).or_insert(default: Vec::new()).push((w as u32, weight));
        self.vertices.entry(key: w as u32).or_insert(default: Vec::new()).push((v as u32, weight));
    }
    pub fn get_neighbors(&self, vertex: u32) -> Option<&Vec<(u32, f64)>> {
        self.vertices.get(&vertex)
    }
    pub fn degree_centrality(&self) -> HashMap<u32, f64> {
        let mut degree_centrality: HashMap<u32, f64> = HashMap::new();
        for (&node_id: u32, neighbors: &Vec<(u32, f64)>) in &self.vertices {
            let degree: f64 = neighbors.len() as f64;
            degree_centrality.insert(k: node_id, v: degree-1.0);        You, 2 hours ago • More Cleaning and Adding
        }
        degree_centrality
    }
    pub fn shortest_path_lengths(&self, start: u32) -> HashMap<u32, f64> {
        let mut distances: HashMap<u32, f64> = HashMap::new();
        let mut pq: BinaryHeap<Vertex> = BinaryHeap::new();
        let mut visited: HashSet<u32> = HashSet::new();
        distances.insert(k: start, v: 0.0);
        pq.push(item: Vertex { id: start, distance: 0.0 });
        while let Some(Vertex { id: u32, distance: f64 }) = pq.pop() {
            if visited.contains(&id) {
                continue;
            }
            visited.insert(id);
            if let Some(neighbors: &Vec<(u32, f64)>) = self.get_neighbors(vertex: id) {
                for &(neighbor: u32, weight: f64) in neighbors {
                    let new_distance: f64 = distance + weight;
                    if !distances.contains_key(&neighbor) || new_distance < *distances.get(&neighbor).unwrap() {
                        distances.insert(k: neighbor, v: new_distance);
                        pq.push(item: Vertex { id: neighbor, distance: new_distance });
                    }
                }
            }
        }
        distances
    }

    pub fn closeness_centrality(&self) -> HashMap<u32, f64> {
        let mut closeness_centrality: HashMap<u32, f64> = HashMap::new();
        let total_nodes: f64 = self.vertices.len() as f64;
        for (&node_id: u32, _) in &self.vertices {
            let shortest_paths: HashMap<u32, f64> = self.shortest_path_lengths(start: node_id);
            let total_distance: f64 = shortest_paths.values().sum();
            let closeness: f64 = if total_distance > 0.0 { (total_nodes - 1.0) / total_distance } else { 0.0 };
            closeness_centrality.insert(k: node_id, v: closeness);
        }
        closeness_centrality
    }
} impl Graph
```

We can see that the main aspect of the Graph struct is its adjacency list which is created by using the add.edge function of the Graph struct which is used when I iterate through an edge list based on the similarity score of two distinct games. My central question is **"What matches have been the most similar since 2012?"** Therefore, I wanted to make the graph exclusively of games with a similarity of above .75. So, when I calculated degree centrality in the future, there was meaningful insight as to what the maximum degree would mean. You can also see how I calculated the closeness and degree centrality of each vertex.

I will get into how I am finding my two largest takeaways later. However, it is more important to introduce the formula for which I am calculating similarity. Pictured here is the function:



```
    for event: &GameEntry in &entry1 {
        for event2: &GameEntry in &entry2 {
            let weights: Vec<(f64, f64, f64)> = vec![
                (1.0, event.home_club_goals as f64, event2.home_club_goals as f64),
                (1.0, event.away_club_goals as f64, event2.away_club_goals as f64),
                (3.0, event.total_goals as f64, event2.total_goals as f64),
                (1.0, event.minute, event2.minute),
            ];
            let mut weighted_sum: f64 = 0.0;
            for (weight: &f64, value1: &f64, value2: &f64) in &weights {
                if *weight == 3.0 && !total_goals_checked {
                    total_goals_diff += (*value1 - *value2).abs();
                    total_goals_checked = true;
                } else if *weight == 1.0 && !home_goals_checked {
                    total_home_goals_diff += (*value1 - *value2).abs();
                    home_goals_checked = true;
                } else if *weight == 1.0 && !away_goals_checked {      You, 2 days ago • More t
                    total_away_goals_diff += (*value1 - *value2).abs();
                    away_goals_checked = true;
                } else {
                    weighted_sum += *weight * (value1 - value2).abs();
                }
                weight_sum += *weight;
            }
            if event.description == event2.description && event.description != "Substitution" {
                multiplier+=0.25;
            }
            sim_sum += weighted_sum;
        }
    }
    let mut similarity: f64 = sim_sum / weight_sum;
    if multiplier != 0.0 {
        similarity = multiplier * (similarity*similarity).log10();
    }
    else {
        similarity = similarity.log10()* 3.0;
    }
    let normalized_similarity: f64 = 1.0 / (1.0 + similarity.abs());
    let total_diff: f64 = total_home_goals_diff + total_away_goals_diff + total_goals_diff;
    let adjusted_similarity: f64 = normalized_similarity - (total_diff * 0.005);
    adjusted_similarity
}fn calculate_similarity
```

The biggest aspect of this formula is **the addition of a multiplier for shared events.** Total goals, as well as home and away goals, are important, but having the same description i.e. “Right-footed shot” for one of their goals should be weighted heavier in similarity. This is because it is more unlikely to occur, so events with the description “Substitution” were not considered. Since each team is allowed 3 subs in a game, substitutions are much more common than a shared goal type. I went through many iterations of this scorecard and felt that using a log scale for the pre-adjusted and normalized was the best solution. It allowed me to get a range of distributions that made sense and showed how certain events impacted similarity. To finish the rest of the formula, I normalized and adjusted the similarity based on the differences of other factors before returning the final similarity score. Based on this metric, we can see from the output:

```
The highest similarity score was: 0.9552107034415706

The two games were: 2223915 and 2224109
 Game 1 has the following events:

GameEntry { node_id: 597, game_id: 2223915, minute: 76.0, description: "Substitution", home_club_goals: 1, away_club_goals: 0, total_goals: 1 }
GameEntry { node_id: 597, game_id: 2223915, minute: 83.0, description: "Substitution", home_club_goals: 1, away_club_goals: 0, total_goals: 1 }
GameEntry { node_id: 597, game_id: 2223915, minute: 79.0, description: "Substitution", home_club_goals: 1, away_club_goals: 0, total_goals: 1 }
GameEntry { node_id: 597, game_id: 2223915, minute: 87.0, description: "Right-footed shot", home_club_goals: 1, away_club_goals: 0, total_goals: 1 }

 Game 2 has the following events:
GameEntry { node_id: 791, game_id: 2224109, minute: 80.0, description: "Right-footed shot", home_club_goals: 0, away_club_goals: 1, total_goals: 1 }
GameEntry { node_id: 791, game_id: 2224109, minute: 74.0, description: "Substitution", home_club_goals: 0, away_club_goals: 1, total_goals: 1 }
GameEntry { node_id: 791, game_id: 2224109, minute: 86.0, description: "Substitution", home_club_goals: 0, away_club_goals: 1, total_goals: 1 }
GameEntry { node_id: 791, game_id: 2224109, minute: 76.0, description: "Substitution", home_club_goals: 0, away_club_goals: 1, total_goals: 1 }
GameEntry { node_id: 791, game_id: 2224109, minute: 86.0, description: "Substitution", home_club_goals: 0, away_club_goals: 1, total_goals: 1 }
GameEntry { node_id: 791, game_id: 2224109, minute: 84.0, description: "Substitution", home_club_goals: 0, away_club_goals: 1, total_goals: 1 }
```

The highest similarity score was approximately .95. This is because the differences in minutes between these assorted events are so minimal (the total range is only 13 minutes) so when combined with a boost because of the shared “Right-footed shot” goal, they had an extremely

high score. To round out the rest of the top 10 most similar games:

```
The top 10 games are:
The similarity score was 0.9552107034415706, with their game_ids being 2223915 and 2224109 and having a total of 1 goals scored.
The similarity score was 0.9439080916905759, with their game_ids being 2222796 and 2224657 and having a total of 0 goals scored.
The similarity score was 0.9439080916905759, with their game_ids being 2223979 and 2224071 and having a total of 0 goals scored.
The similarity score was 0.9402256739400452, with their game_ids being 2223948 and 2224061 and having a total of 1 goals scored.
The similarity score was 0.9365948831668456, with their game_ids being 2223870 and 2223979 and having a total of 0 goals scored.
The similarity score was 0.9285625739577797, with their game_ids being 2224160 and 2224591 and having a total of 1 goals scored.
The similarity score was 0.9284633106920003, with their game_ids being 2222647 and 2222820 and having a total of 3 goals scored.
The similarity score was 0.926566765129454, with their game_ids being 2222623 and 2223092 and having a total of 2 goals scored.
The similarity score was 0.9227698515903493, with their game_ids being 2223915 and 2223948 and having a total of 1 goals scored.
The similarity score was 0.9213633740951411, with their game_ids being 2223074 and 2223094 and having a total of 1 goals scored.
```

This is an interesting distribution of outcomes, especially because of the many low-scoring games. I theorize that because soccer games are naturally lower scoring, there is a better chance of having multiple games with similar event distributions because of the larger sample size with that amount of goals.

To return to the degree and closeness centrality mentioned as parts of the Graph struct, I felt these concepts to be important because degree centrality can show what type of game has many similar neighbors (since the edges are all at least .75 similarity score). Closeness centrality can show which types of events make games the “most similar” (highest weights).

To calculate closeness centrality, I am finding the shortest path to each other vertex, summing the distances, then dividing it by the total nodes. This gives me the closeness, but to find the shortest path, I used a Binary Heap called pq. The node with the smallest distance is popped from the pq heap. If I had already visited that node,l then I skipped it, but if not then I marked it as seen. Then, you iterate through the neighbors of the nodes and it calculates the new distance by adding the weight of the edge between the current node and the neighbor to the current node's distance. If the new distance is smaller than the current distance of the neighbor

(or if the neighbor hasn't been visited yet), the algorithm updates the neighbor's distance in the distances map and pushes the neighbor onto the pq heap. I then wrote code to iterate through the results and find the node with the highest closeness centrality. Here are the results:

```
The node with the highest degree centrality is node 526 with a degree of 227. The assocaited game id is 2223844
This game has the following events:
GameEntry { node_id: 526, game_id: 2223844, minute: 61.0, description: "Substitution", home_club_goals: 1, away_club_goals: 1, total_goals: 2 }
GameEntry { node_id: 526, game_id: 2223844, minute: 56.0, description: "Substitution", home_club_goals: 1, away_club_goals: 1, total_goals: 2 }
GameEntry { node_id: 526, game_id: 2223844, minute: 71.0, description: "Substitution", home_club_goals: 1, away_club_goals: 1, total_goals: 2 }
GameEntry { node_id: 526, game_id: 2223844, minute: 45.0, description: "Right-footed shot", home_club_goals: 1, away_club_goals: 1, total_goals: 2 }
GameEntry { node_id: 526, game_id: 2223844, minute: 70.0, description: "Left-footed shot", home_club_goals: 1, away_club_goals: 1, total_goals: 2 }
GameEntry { node_id: 526, game_id: 2223844, minute: 74.0, description: "Substitution", home_club_goals: 1, away_club_goals: 1, total_goals: 2 }
GameEntry { node_id: 526, game_id: 2223844, minute: 83.0, description: "Substitution", home_club_goals: 1, away_club_goals: 1, total_goals: 2 }
GameEntry { node_id: 526, game_id: 2223844, minute: 71.0, description: "Substitution", home_club_goals: 1, away_club_goals: 1, total_goals: 2 }
```

This result is not surprising because according to [footystats], the most common scoreline of soccer games is 1-1. The two most common types of goals are left and right-footed shots, meaning this game has events that are extremely likely to appear in other games in the dataset. Therefore, we can garner that when determining what makes a game “similar” the scoreline is typically 1-1.

To find degree centrality, I am storing results in a HashMap called degree_centrality after iterating over each node in the graph, using the vertices (adjacency list).
For each node, it counts the neighbors using .len() and then calculates the centrality by subtracting 1 to avoid counting itself. The degree centrality value is inserted into the HashMap with the node ID as the key. The result of this can be seen below:

```
The node with the highest closeness centrality is 673 with a centrality of 950.3830921902558. The assocaited game id is 2223991
This game has the following events:
GameEntry { node_id: 673, game_id: 2223991, minute: 2.0, description: "Left-footed shot", home_club_goals: 2, away_club_goals: 4, total_goals: 6 }
GameEntry { node_id: 673, game_id: 2223991, minute: 44.0, description: "Substitution", home_club_goals: 2, away_club_goals: 4, total_goals: 6 }
GameEntry { node_id: 673, game_id: 2223991, minute: 60.0, description: "Header", home_club_goals: 2, away_club_goals: 4, total_goals: 6 }
GameEntry { node_id: 673, game_id: 2223991, minute: 69.0, description: "Substitution", home_club_goals: 2, away_club_goals: 4, total_goals: 6 }
GameEntry { node_id: 673, game_id: 2223991, minute: 74.0, description: "Substitution", home_club_goals: 2, away_club_goals: 4, total_goals: 6 }
GameEntry { node_id: 673, game_id: 2223991, minute: 17.0, description: "Penalty", home_club_goals: 2, away_club_goals: 4, total_goals: 6 }
GameEntry { node_id: 673, game_id: 2223991, minute: 82.0, description: "Substitution", home_club_goals: 2, away_club_goals: 4, total_goals: 6 }
GameEntry { node_id: 673, game_id: 2223991, minute: 42.0, description: "Header", home_club_goals: 2, away_club_goals: 4, total_goals: 6 }
GameEntry { node_id: 673, game_id: 2223991, minute: 65.0, description: "Substitution", home_club_goals: 2, away_club_goals: 4, total_goals: 6 }
GameEntry { node_id: 673, game_id: 2223991, minute: 76.0, description: "Substitution", home_club_goals: 2, away_club_goals: 4, total_goals: 6 }
GameEntry { node_id: 673, game_id: 2223991, minute: 72.0, description: "Substitution", home_club_goals: 2, away_club_goals: 4, total_goals: 6 }
```

It is interesting to see that a node with such a high amount of goals scored would be the most connected node. I reason that the 6 goals scored in this game are of different types. This

allows it to have short distances between other nodes, because of the multiplier aspect of the similarity score formula. Therefore, we can discern that a game being similar is largely dependent on the type of goals scored within the game.

In conclusion, to directly answer the question “What matches have been the most similar since 2012?” We found that games 2223915 and 2224109 were the most similar out of the 1000 analyzed games. Game 2223915 was between FC Girondins Bordeaux and FC Toulouse in the French League (Ligue 1). Game 2224109 was between Montpellier HSC and FC Évian Thonon Gaillard, who also play in Ligue 1. Therefore, I am inclined to believe that the style of play within a league impacts similarity. Furthermore, as seen by the results of the degree centrality, similar games are more likely to be low-scoring and have a scoreline of 1-1 because it is the most common scoreline in the sport. Additionally, the analysis of the closeness centrality shows that similar games depend on the type of goals scored within them. This is consistent with the two most similar games which both had a “Right-footed shot” goal. Although analyzing additional games could bring new information to light, it is very interesting to see the factors that influence the similarity between games and allowed me to conclude the two most similar games since 2012.