1. Problem Description (10%)

Parsing the actual file as the information inside is considered malformed by many compilers making a work around required. Also accounted for names that have letters with accents.

2. Solution Description (40%)

Due to the less than functional form that the given project file's information was, using JSON files in any capacity was impossible. Using a delimiter that is set to find "," divides the different columns, then due to the cast column always being the third, the code would set itself to that column.Then , the selected cast section would be sent to a function where the term searched for "name:" as that would have the proper name that would be needed. After getting the raw name, to account for names that use unique letters a decoder that located the \u starter and numbers was made and it was pushed to a function to decode and insert the proper letter into the name. When the names for a row of a particular cast column was extracted and decoded it is stored inside of a hash table. After going through the entire file, the hash table is sent to an adjacency list function where they are connected in accordance to Milestone 1. For Milestone 2, degree_centrality is checked for each actor, then sorted into ascending order and the top 5 is displayed. For Milestone 3, BFS was employed for this task as the hash was far too large for DFS, after the going through the table, a check for connectivity is made and if the value is not the correct one, the number of actual connected components is displayed along with stating that the graph is not connected. For Milestone 4, first the input for the Actors is checked inside the table to assure that it is there, if the same name is inputted twice 0 is returned as there are no separation between the two, after those two checks the code moves through the entire list to search for the other actor incrementing the distance counter as it searches. For Milestone 5 it would run the same checks as Milestone 4, but in addition, if the input has no neighbors, which may happen, it will give a message declaring that. After going through the checks it will return the list from the function which will print itself out showing the direct path.

3. Initial Non-AI Attempt to Code the Solution (10%)

By orders of magnitude , the majority of the time spent was figuring out how to get the information out of "tmdb_5000_credits" and into a workable function. As can be seen in the code there is no use of libraries of any kind, this is because every one I tried all came back saying that the JSON form that was inside the tmdb_5000_credits file is malformed or incorrect and refused to parse them in any capacity. After not getting multiple to work I turned to chatGPT which told me to use libraries I already used as well as telling me to change every line of the 4804 csv file to get them in the right format. Needless to say something had to change. So I looked at the code and realized that everything is perfectly aligned so all I needed to do was find a way to isolate the cast column and then I would be able to manually parse without relying on a library. After that I saw names that had \u900 and realized that I had to parse unicode which took some youtube guides since I didn't trust chatgpt at this point. Everything after that was quite easy and finished in around 5 days.

4. AI Prompts Used (10%)

“What are the best JSON libraries that can be used to parse csv files”

“What does it mean when the code states that JSON format is incorrect”

“Is there any way to fix JSON format inside of code?”

“Is there a way to get the text of the file without putting it into a JSON format?”

5. Code Testing Description (10%)

The test was done with the code itself. For the Milestone 4 and 5 I used the results of Milestone 2 as well as, to stress test, an incorrect input on the third one to show that it doesn’t stop the code in its tracks along with a message to tell the user about the error.

6. Code Including Tests (20%)

Below is a screenshot of the code, including a set of tests for each function.

```
1   #include <iostream>
2   #include <fstream>
3   #include <string>
4   #include <vector>
5   #include <sstream>
6   #include <unordered_map>
7   #include <set>
8   #include <algorithm>
9   #include <queue>
10  #include <unordered_set>
11
12  using namespace std;
```

```cpp
string decode_unicode(const string& input) {
    string decoded;
    size_t pos = 0;
    string remaining = input;

    while ((pos = remaining.find("\\u", pos)) != string::npos) {
        decoded += remaining.substr(0, pos); // Append text before the Unicode sequence

        if (pos + 6 <= remaining.size()) {
            string hex_code = remaining.substr(pos + 2, 4);
            char32_t code_point = static_cast<char32_t>(stoi(hex_code, nullptr, 16));

            if (code_point <= 0x7F) {
                decoded += static_cast<char>(code_point);
            } else if (code_point <= 0x7FF) {
                decoded += static_cast<char>((code_point >> 6) | 0xC0);
                decoded += static_cast<char>((code_point & 0x3F) | 0x80);
            } else if (code_point <= 0xFFFF) {
                decoded += static_cast<char>((code_point >> 12) | 0xE0);
                decoded += static_cast<char>(((code_point >> 6) & 0x3F) | 0x80);
                decoded += static_cast<char>((code_point & 0x3F) | 0x80);
            }

            pos += 6;
        } else {
            cerr << "Warning: Malformed Unicode sequence in input: " << remaining << endl;
            break;
        }

        remaining = remaining.substr(pos);
        pos = 0;
    }

    decoded += remaining; // Append remaining text
    return decoded;
```

```cpp
vector<string> split_line(const string& line, char delimiter) {
    vector<string> columns;
    stringstream ss(line);
    string column;
    bool in_quotes = false;

    for (char c : line) {
        if (c == '"' && (column.empty() || column.back() != '\\')) {
            in_quotes = !in_quotes;
        } else if (c == delimiter && !in_quotes) {
            columns.push_back(column);
            column.clear();
        } else {
            column += c;
        }
    }
    if (!column.empty()) {
        columns.push_back(column);
    }
    return columns;
}
```

```cpp
vector<string> extract_names_from_cast(const string& cast_column) {
    vector<string> names;
    stringstream ss(cast_column);
    string part;

    while (getline(ss, part, ',')) {
        size_t name_pos = part.find("name:");
        if (name_pos != string::npos) {
            size_t name_start = part.find_first_not_of(" ", name_pos + 5);
            size_t name_end = part.find_first_of(",}", name_start);
            if (name_end == string::npos) name_end = part.size();

            string raw_name = part.substr(name_start, name_end - name_start);
            if (!raw_name.empty() && raw_name.front() == '\"') raw_name.erase(0, 1);
            if (!raw_name.empty() && raw_name.back() == '\"') raw_name.pop_back();

            if (raw_name.find("\\u") != string::npos) {
                raw_name = decode_unicode(raw_name);
            }

            names.push_back(raw_name);
        }
    }
    return names;
}
```

```cpp
// adjacency list
void build_adjacency_list(const unordered_map<string, vector<string>>& movie_actors,
                          unordered_map<string, set<string>>& adjacency_list) {
    for (const auto& [movie_id, actors] : movie_actors) {
        for (size_t i = 0; i < actors.size(); ++i) {
            for (size_t j = i + 1; j < actors.size(); ++j) {
                adjacency_list[actors[i]].insert(actors[j]);
                adjacency_list[actors[j]].insert(actors[i]);
            }
        }
    }
}
```

```cpp
void find_top_5_actors(const unordered_map<string, set<string>>& adjacency_list) {
    vector<pair<string, int>> degree_centrality;

    // Calculate degree centrality
    for (const auto& [actor, co_actors] : adjacency_list) {
        degree_centrality.emplace_back(actor, co_actors.size());
    }

    // Sort
    sort(degree_centrality.begin(), degree_centrality.end(),
        [](const pair<string, int>& a, const pair<string, int>& b) {
            return b.second < a.second; // Sort by second element (degree) descending
        });

    // Display the top 5
    cout << "Top 5 actors by degree centrality:" << endl;
    for (size_t i = 0; i < 5 && i < degree_centrality.size(); ++i) {
        cout << degree_centrality[i].first << " - Degree: " << degree_centrality[i].second << endl;
    }
}
```

```cpp
//  BFS and mark all connected nodes
void bfs(const string& start_node, const unordered_map<string, set<string>>& adjacency_list,
         unordered_set<string>& visited) {
    queue<string> to_visit;
    to_visit.push(start_node);
    visited.insert(start_node);

    while (!to_visit.empty()) {
        string current = to_visit.front();
        to_visit.pop();

        for (const auto& neighbor : adjacency_list.at(current)) {
            if (visited.find(neighbor) == visited.end()) {
                visited.insert(neighbor);
                to_visit.push(neighbor);
            }
        }
    }
}
```

```cpp
//determine if the graph is connected and count connected components
void check_graph_connectivity(const unordered_map<string, set<string>>& adjacency_list) {
    unordered_set<string> visited;
    int connected_components = 0;

    for (const auto& [actor, _] : adjacency_list) {
        if (visited.find(actor) == visited.end()) {
            // Start a new BFS
            ++connected_components;
            bfs(actor, adjacency_list, visited);
        }
    }

    if (connected_components == 1) {
        cout << "The graph is connected." << endl;
    } else {
        cout << "The graph is not connected." << endl;
        cout << "Number of connected components: " << connected_components << endl;
    }
}
```

```cpp
int shortest_degree_of_separation(const string& actor_a, const string& actor_b,
                                  const unordered_map<string, set<string>>& adjacency_list) {
    // Check if both actors exist
    if (adjacency_list.find(actor_a) == adjacency_list.end()) {
        cerr << "Error: Actor \"" << actor_a << "\" not found in the graph." << endl;
        return -1;
    }
    if (adjacency_list.find(actor_b) == adjacency_list.end()) {
        cerr << "Error: Actor \"" << actor_b << "\" not found in the graph." << endl;
        return -1;
    }

    if (actor_a == actor_b) return 0;

    unordered_set<string> visited;
    queue<pair<string, int>> to_visit; // Pair of actor and current distance
    to_visit.push({actor_a, 0});
    visited.insert(actor_a);

    while (!to_visit.empty()) {
        auto [current_actor, distance] = to_visit.front();
        to_visit.pop();

        for (const auto& neighbor : adjacency_list.at(current_actor)) {
            if (neighbor == actor_b) return distance + 1;

            if (visited.find(neighbor) == visited.end()) {
                visited.insert(neighbor);
                to_visit.push({neighbor, distance + 1});
            }
        }
    }
```

```
208
209         return -1; // Actors are in different connected components
210     }
```

```
212     // find the shortest path between two actors
213     vector<string> shortest_actor_chain(const string& actor_a, const string& actor_b,
214                                         const unordered_map<string, set<string>>& adjacency_list) {
215         // Check if both actors exist
216         if (adjacency_list.find(actor_a) == adjacency_list.end()) {
217             cerr << "Error: Actor \"" << actor_a << "\" not found in the graph." << endl;
218             return {};
219         }
220         if (adjacency_list.find(actor_b) == adjacency_list.end()) {
221             cerr << "Error: Actor \"" << actor_b << "\" not found in the graph." << endl;
222             return {};
223         }
224 
225         if (actor_a == actor_b) return {actor_a};
226 
227         unordered_map<string, string> predecessors;
228         unordered_set<string> visited;
229         queue<string> to_visit;
230 
231         to_visit.push(actor_a);
232         visited.insert(actor_a);
233 
234         while (!to_visit.empty()) {
235             string current_actor = to_visit.front();
236             to_visit.pop();
237 
238             // Check if the current actor has neighbors
239             if (adjacency_list.find(current_actor) == adjacency_list.end()) {
240                 cerr << "Warning: Actor \"" << current_actor << "\" has no connections." << endl;
241                 continue;
242             }
243 
```

```
244             for (const auto& neighbor : adjacency_list.at(current_actor)) {
245                 if (visited.find(neighbor) == visited.end()) {
246                     visited.insert(neighbor);
247                     predecessors[neighbor] = current_actor;
248 
249                     if (neighbor == actor_b) {
250 
251                         vector<string> path;
252                         string step = actor_b;
253                         while (step != actor_a) {
254                             path.push_back(step);
255                             step = predecessors[step];
256                         }
257                         path.push_back(actor_a);
258                         reverse(path.begin(), path.end());
259                         return path;
260                     }
261 
262                     to_visit.push(neighbor);
263                 }
264             }
265         }
266 
267         return {}; // No path
268     }
```

```
270  int main() {
271      string file_path = "tmdb_5000_credits.csv";
272      ifstream file(file_path);
273
274      if (!file.is_open()) {
275          cerr << "Error: Could not open file " << file_path << endl;
276          return 1;
277      }
278
279      unordered_map<string, vector<string>> movie_actors;
280      unordered_map<string, set<string>> adjacency_list;
281
282      string line;
283      bool is_header = true;
284
285      while (getline(file, line)) {
286          if (is_header) {
287              is_header = false;
288              continue;
289          }
290
291          vector<string> columns = split_line(line, ',');
292
293          if (columns.size() >= 3) {
294              string movie_id = columns[0];
295              string cast_column = columns[2];
296
297              vector<string> actors = extract_names_from_cast(cast_column);
298              movie_actors[movie_id] = actors;
299          }
300      }
301
302      file.close();
```

```
304         build_adjacency_list(movie_actors, adjacency_list);
305
306
307         cout << "Adjacency List:" << endl;
308         for (const auto& [actor, co_actors] : adjacency_list) {
309             cout << actor << ": ";
310             for (const auto& co_actor : co_actors) {
311                 cout << co_actor << ", ";
312             }
313             cout << endl;
314         }
315
316
317         find_top_5_actors(adjacency_list);
318
319
320         check_graph_connectivity(adjacency_list);
```

```
323         cout << "\nShortest Degree of Separation Examples:" << endl;
324     vector<pair<string, string>> test_pairs = {
325         {"Samuel L. Jackson", "Morgan Freeman"},
326         {"Stan Lee", "Anne Fletcher"},
327         {"Nonexistent Actor", "Robert De Niro"} // Test case for missing actor
328     };
329
330     for (const auto& [actor_a, actor_b] : test_pairs) {
331         int distance = shortest_degree_of_separation(actor_a, actor_b, adjacency_list);
332         if (distance != -1) {
333             cout << "Degree of separation between " << actor_a << " and " << actor_b << ": " << distance << endl;
334         } else {
335             cout << "Could not determine degree of separation between " << actor_a << " and " << actor_b << "." << endl;
336         }
337     }
338
339     for (const auto& [actor_a, actor_b] : test_pairs) {
340             vector<string> path = shortest_actor_chain(actor_a, actor_b, adjacency_list);
341             if (!path.empty()) {
342                 cout << "Shortest path between " << actor_a << " and " << actor_b << ": ";
343                 for (size_t i = 0; i < path.size(); ++i) {
344                     cout << path[i];
345                     if (i < path.size() - 1) cout << " -> ";
346                 }
347                 cout << endl;
348             } else {
349                 cout << actor_a << " and " << actor_b << " are in different connected components or not in the graph." << endl;
350             }
351         }
352
353         return 0;
354     }
```

example of Milestone one:

```
Naomi Gaskin: Anthony Anderson, Bill Duke, Bruce McGill, Christopher Lawford, DMX, Daniel Kash, David Vadim, Dean Mckenzie, Drag-On, Eva Mendes, Gregory Vitale, Isaiah
ington, Jennifer Irwin, Jill Hennessy, John Ralston, Matthew G. Taylor, Michael Jai White, Noah Danby, Paolo Mastropietro, Quancetia Hamilton, Rick Demas, Rothaford Gra
hane Daly, Shawn Lawrence, Steven Seagal, Tom Arnold,
Jiří Krytinář: Alexander Oliver, Anne Howells, Barbara Bryne, Brian Pettifer, Charles Kay, Christine Ebersole, Cynthia Nixon, Dana Vávrová, Douglas Seale, Elizabeth Ber
e, F. Murray Abraham, Felicity Lott, Hana Brejchová, Herman Meckler, Isobel Buchanan, Ivan Pokorný, Jan Kuželka, Jan Pohan, Jeffrey Jones, Jitka Molavcová, Jiří Lír, Jo
arrafa, John Strauss, John Tomlinson, Jonathan Moore, June Anderson, Karel Effa, Karel Fiala, Karel Gult, Karel Hábl, Karl-Heinz Teuber, Kenny Baker, Ladislav Krečmer,
eth Bartlett, Martin Cavina, Michele Esposito, Milan Demjanenko, Milan Riehs, Miriam Chytilová, Miro Grisa, Nicholas Kepros, Patrick Hines, Pavel Nový, Peter DiGesu, Ph
 Lenkowsky, René Gabzdyl, Richard Frank, Richard Stilwell, Robin Leggate, Roderick Cook, Roy Dotrice, Samuel Ramey, Simon Callow, Therese Herz, Tom Hulce, Vincent Schia
```

Example of Milestone 2 and 3:

```
Top 5 actors by degree centrality:
Samuel L. Jackson - Degree: 1899
Morgan Freeman - Degree: 1596
Stan Lee - Degree: 1511
Anne Fletcher - Degree: 1501
Robert De Niro - Degree: 1467
The graph is not connected.
Number of connected components: 82
```

Example of Milestone 4 and 5:

```
Shortest Degree of Separation Examples:
Degree of separation between Samuel L. Jackson and Morgan Freeman: 2
Degree of separation between Stan Lee and Anne Fletcher: 2
Error: Actor "Nonexistent Actor" not found in the graph.
Could not determine degree of separation between Nonexistent Actor and Robert De Niro.
Shortest path between Samuel L. Jackson and Morgan Freeman: Samuel L. Jackson -> Alan North -> Morgan Freeman
Shortest path between Stan Lee and Anne Fletcher: Stan Lee -> Aasif Mandvi -> Anne Fletcher
Error: Actor "Nonexistent Actor" not found in the graph.
Nonexistent Actor and Robert De Niro are in different connected components or not in the graph.
```

(Provide a readable screenshot from your IDE showing the implementation and tests)

Code:

```
    #include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>
#include <unordered_map>
#include <set>
#include <algorithm>
#include <queue>
#include <unordered_set>

using namespace std;



string decode_unicode(const string& input) {
    string decoded;
    size_t pos = 0;
```

```
    string remaining = input;

    while ((pos = remaining.find("\\u", pos)) != string::npos) {
        decoded += remaining.substr(0, pos); // Append text before the Unicode sequence

        if (pos + 6 <= remaining.size()) {
            string hex_code = remaining.substr(pos + 2, 4);
            char32_t code_point = static_cast<char32_t>(stoi(hex_code, nullptr, 16));

            if (code_point <= 0x7F) {
                decoded += static_cast<char>(code_point);
            } else if (code_point <= 0x7FF) {
                decoded += static_cast<char>((code_point >> 6) | 0xC0);
                decoded += static_cast<char>((code_point & 0x3F) | 0x80);
            } else if (code_point <= 0xFFFF) {
                decoded += static_cast<char>((code_point >> 12) | 0xE0);
                decoded += static_cast<char>(((code_point >> 6) & 0x3F) | 0x80);
                decoded += static_cast<char>((code_point & 0x3F) | 0x80);
            }

            pos += 6;
        } else {
            cerr << "Warning: Malformed Unicode sequence in input: " << remaining << endl;
            break;
        }

        remaining = remaining.substr(pos);
        pos = 0;
    }

    decoded += remaining; // Append remaining text
    return decoded;
}

vector<string> split_line(const string& line, char delimiter) {
    vector<string> columns;
    stringstream ss(line);
    string column;
    bool in_quotes = false;

    for (char c : line) {
        if (c == '"' && (column.empty() || column.back() != '\\')) {
            in_quotes = !in_quotes;
        } else if (c == delimiter && !in_quotes) {
```

```
            columns.push_back(column);
            column.clear();
        } else {
            column += c;
        }
    }
    if (!column.empty()) {
        columns.push_back(column);
    }
    return columns;
}

vector<string> extract_names_from_cast(const string& cast_column) {
    vector<string> names;
    stringstream ss(cast_column);
    string part;

    while (getline(ss, part, ',')) {
        size_t name_pos = part.find("name:");
        if (name_pos != string::npos) {
            size_t name_start = part.find_first_not_of(" ", name_pos + 5);
            size_t name_end = part.find_first_of(",}", name_start);
            if (name_end == string::npos) name_end = part.size();

            string raw_name = part.substr(name_start, name_end - name_start);
            if (!raw_name.empty() && raw_name.front() == '\"') raw_name.erase(0, 1);
            if (!raw_name.empty() && raw_name.back() == '\"') raw_name.pop_back();

            if (raw_name.find("\\u") != string::npos) {
                raw_name = decode_unicode(raw_name);
            }

            names.push_back(raw_name);
        }
    }
    return names;
}

// adjacency list
void build_adjacency_list(const unordered_map<string, vector<string>>& movie_actors,
                          unordered_map<string, set<string>>& adjacency_list) {
    for (const auto& [movie_id, actors] : movie_actors) {
        for (size_t i = 0; i < actors.size(); ++i) {
            for (size_t j = i + 1; j < actors.size(); ++j) {
```

```
                adjacency_list[actors[i]].insert(actors[j]);
                adjacency_list[actors[j]].insert(actors[i]);
            }
        }
    }
}

void find_top_5_actors(const unordered_map<string, set<string>>& adjacency_list) {
    vector<pair<string, int>> degree_centrality;

    // Calculate degree centrality
    for (const auto& [actor, co_actors] : adjacency_list) {
        degree_centrality.emplace_back(actor, co_actors.size());
    }

    // Sort
    sort(degree_centrality.begin(), degree_centrality.end(),
        [](const pair<string, int>& a, const pair<string, int>& b) {
            return b.second < a.second; // Sort by second element (degree) descending
        });

    // Display the top 5
    cout << "Top 5 actors by degree centrality:" << endl;
    for (size_t i = 0; i < 5 && i < degree_centrality.size(); ++i) {
        cout << degree_centrality[i].first << " - Degree: " << degree_centrality[i].second << endl;
    }
}

//  BFS and mark all connected nodes
void bfs(const string& start_node, const unordered_map<string, set<string>>&
adjacency_list,
        unordered_set<string>& visited) {
    queue<string> to_visit;
    to_visit.push(start_node);
    visited.insert(start_node);

    while (!to_visit.empty()) {
        string current = to_visit.front();
        to_visit.pop();

        for (const auto& neighbor : adjacency_list.at(current)) {
            if (visited.find(neighbor) == visited.end()) {
                visited.insert(neighbor);
                to_visit.push(neighbor);
```

```
            }
        }
    }
}

//determine if the graph is connected and count connected components
void check_graph_connectivity(const unordered_map<string, set<string>>& adjacency_list) {
    unordered_set<string> visited;
    int connected_components = 0;

    for (const auto& [actor, _] : adjacency_list) {
        if (visited.find(actor) == visited.end()) {
            // Start a new BFS
            ++connected_components;
            bfs(actor, adjacency_list, visited);
        }
    }

    if (connected_components == 1) {
        cout << "The graph is connected." << endl;
    } else {
        cout << "The graph is not connected." << endl;
        cout << "Number of connected components: " << connected_components << endl;
    }
}

int shortest_degree_of_separation(const string& actor_a, const string& actor_b,
                                  const unordered_map<string, set<string>>& adjacency_list) {
    // Check if both actors exist
    if (adjacency_list.find(actor_a) == adjacency_list.end()) {
        cerr << "Error: Actor \"" << actor_a << "\" not found in the graph." << endl;
        return -1;
    }
    if (adjacency_list.find(actor_b) == adjacency_list.end()) {
        cerr << "Error: Actor \"" << actor_b << "\" not found in the graph." << endl;
        return -1;
    }

    if (actor_a == actor_b) return 0;

    unordered_set<string> visited;
    queue<pair<string, int>> to_visit; // Pair of actor and current distance
    to_visit.push({actor_a, 0});
    visited.insert(actor_a);
```

```
    while (!to_visit.empty()) {
        auto [current_actor, distance] = to_visit.front();
        to_visit.pop();

        for (const auto& neighbor : adjacency_list.at(current_actor)) {
            if (neighbor == actor_b) return distance + 1;

            if (visited.find(neighbor) == visited.end()) {
                visited.insert(neighbor);
                to_visit.push({neighbor, distance + 1});
            }
        }
    }

    return -1; // Actors are in different connected components
}

// find the shortest path between two actors
vector<string> shortest_actor_chain(const string& actor_a, const string& actor_b,
                                    const unordered_map<string, set<string>>& adjacency_list) {
    // Check if both actors exist
    if (adjacency_list.find(actor_a) == adjacency_list.end()) {
        cerr << "Error: Actor \"" << actor_a << "\" not found in the graph." << endl;
        return {};
    }
    if (adjacency_list.find(actor_b) == adjacency_list.end()) {
        cerr << "Error: Actor \"" << actor_b << "\" not found in the graph." << endl;
        return {};
    }

    if (actor_a == actor_b) return {actor_a};

    unordered_map<string, string> predecessors;
    unordered_set<string> visited;
    queue<string> to_visit;

    to_visit.push(actor_a);
    visited.insert(actor_a);

    while (!to_visit.empty()) {
        string current_actor = to_visit.front();
        to_visit.pop();
```

```
        // Check if the current actor has neighbors
        if (adjacency_list.find(current_actor) == adjacency_list.end()) {
            cerr << "Warning: Actor \"" << current_actor << "\" has no connections." << endl;
            continue;
        }

        for (const auto& neighbor : adjacency_list.at(current_actor)) {
            if (visited.find(neighbor) == visited.end()) {
                visited.insert(neighbor);
                predecessors[neighbor] = current_actor;

                if (neighbor == actor_b) {

                    vector<string> path;
                    string step = actor_b;
                    while (step != actor_a) {
                        path.push_back(step);
                        step = predecessors[step];
                    }
                    path.push_back(actor_a);
                    reverse(path.begin(), path.end());
                    return path;
                }

                to_visit.push(neighbor);
            }
        }
    }

    return {}; // No path
}

int main() {
    string file_path = "tmdb_5000_credits.csv";
    ifstream file(file_path);

    if (!file.is_open()) {
        cerr << "Error: Could not open file " << file_path << endl;
        return 1;
    }

    unordered_map<string, vector<string>> movie_actors;
    unordered_map<string, set<string>> adjacency_list;
```

```
    string line;
    bool is_header = true;

    while (getline(file, line)) {
        if (is_header) {
            is_header = false;
            continue;
        }

        vector<string> columns = split_line(line, ',');

        if (columns.size() >= 3) {
            string movie_id = columns[0];
            string cast_column = columns[2];

            vector<string> actors = extract_names_from_cast(cast_column);
            movie_actors[movie_id] = actors;
        }
    }

    file.close();

    build_adjacency_list(movie_actors, adjacency_list);


    cout << "Adjacency List:" << endl;
    for (const auto& [actor, co_actors] : adjacency_list) {
        cout << actor << ": ";
        for (const auto& co_actor : co_actors) {
            cout << co_actor << ", ";
        }
        cout << endl;
    }


    find_top_5_actors(adjacency_list);


    check_graph_connectivity(adjacency_list);


    cout << "\nShortest Degree of Separation Examples:" << endl;
vector<pair<string, string>> test_pairs = {
    {"Samuel L. Jackson", "Morgan Freeman"},
```

```
        {"Stan Lee", "Anne Fletcher"},
        {"Nonexistent Actor", "Robert De Niro"} // Test case for missing actor
    };

    for (const auto& [actor_a, actor_b] : test_pairs) {
        int distance = shortest_degree_of_separation(actor_a, actor_b, adjacency_list);
        if (distance != -1) {
            cout << "Degree of separation between " << actor_a << " and " << actor_b << ": " <<
distance << endl;
        } else {
            cout << "Could not determine degree of separation between " << actor_a << " and " <<
actor_b << "." << endl;
        }
    }

    for (const auto& [actor_a, actor_b] : test_pairs) {
            vector<string> path = shortest_actor_chain(actor_a, actor_b, adjacency_list);
            if (!path.empty()) {
                cout << "Shortest path between " << actor_a << " and " << actor_b << ": ";
                for (size_t i = 0; i < path.size(); ++i) {
                    cout << path[i];
                    if (i < path.size() - 1) cout << " -> ";
                }
                cout << endl;
            } else {
                cout << actor_a << " and " << actor_b << " are in different connected components or
not in the graph." << endl;
            }
        }

        return 0;
    }
```