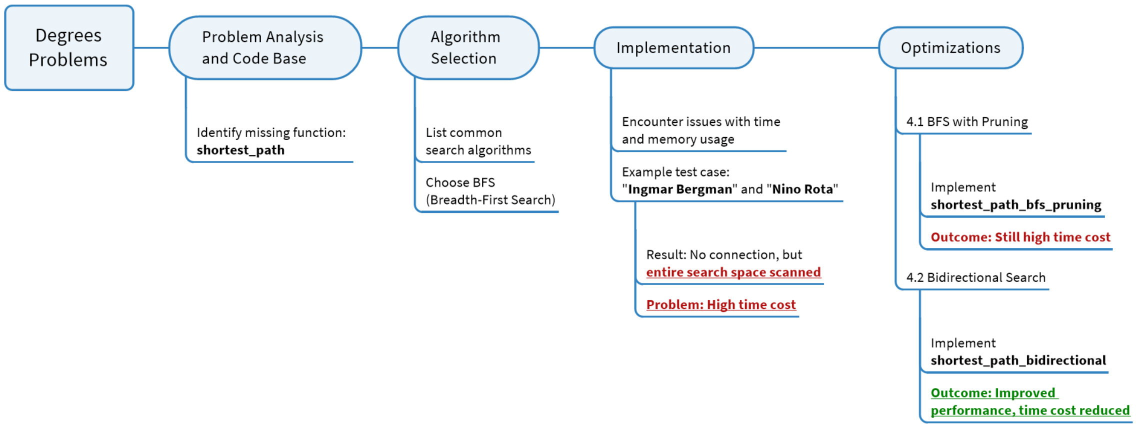
Report on Approach to Solving the Degrees of Separation Problem



Member Name	Student ID
Nguyen Thanh Vinh	2101140081
Tran Duc Viet	2101140080
Pham Tuan Phuong	2201140068

1. Problem Analysis and Code Base

1.1 Problem

- Requirement: Find the shortest path between two actors through shared movies.
- Missing function in `degree.py` : `shortest_path` to calculate the shortest connection.
- Goal: Identify intermediary steps between two actors through movie connections.

1.2 Code Base

1.2.1 File `util.py`

- **Purpose**: Provides essential data structures and helper functions for managing the search process in finding the
shortest connection between two actors through shared movies.
- **Components**:

    - `Node`: Represents a single search node with key information:
        - `state`: Stores the `person_id` of the actor at this point in the search.
        - `parent`: References the previous `Node` that led to this actor, allowing path reconstruction from the target
back to the source.
        - `action`: Holds the `movie_id` that connects this actor to their `parent` actor, representing the shared
movie.
    - `StackFrontier`: Manages nodes in a Last-In-First-Out (LIFO) structure, suitable for depth-first search where
recently added nodes are explored first.
    - `QueueFrontier`: Manages nodes in a First-In-First-Out (FIFO) structure, ideal for breadth-first search.
Ensures that nodes are explored in the order they were added, making it suitable for finding the shortest path in
an unweighted graph, like connecting actors through movies.

1.2.2 File `degrees.py`

- **Purpose**: Executes the main functionality of the program.
- **Functions and roles**:
    - `load_data`: Loads actor, movie, and connection data.
    - `person_id_for_name`: Finds actor ID from the name.
    - `neighbors_for_person`: Retrieves a list of connected actors through shared movies.
    - **Required function**: `shortest_path` – this function will be responsible for finding the shortest path between
two actors using data from the helper functions.

2. Algorithm Selection

2.1 Common Search Algorithms

Algorithm	Description	Limitations
Depth-First Search (DFS)	Explores each branch to its deepest level before backtracking.	Not suitable for shortest path due to potential depth issues.

Breadth-First Search (BFS)	Explores nodes level by level, guaranteeing shortest path in unweighted graphs.	Can be memory-intensive for large graphs.
Greedy Best-First Search	Chooses the path that appears closest to the target using a heuristic.	Does not guarantee shortest path in unweighted graphs.
A*	Uses a heuristic to estimate cost to target, suitable for weighted graphs.	More effective in cases with defined heuristic costs, not ideal here.
Minimax	Used in adversarial games to minimize the possible loss, assuming optimal opponent behavior.	Not suitable for shortest path; primarily for decision-making in games.
Alpha-Beta Pruning	Optimizes Minimax by pruning unneeded branches, reducing computation in game trees.	Only applicable in adversarial search, not pathfinding.

2.2 Reason for Choosing BFS

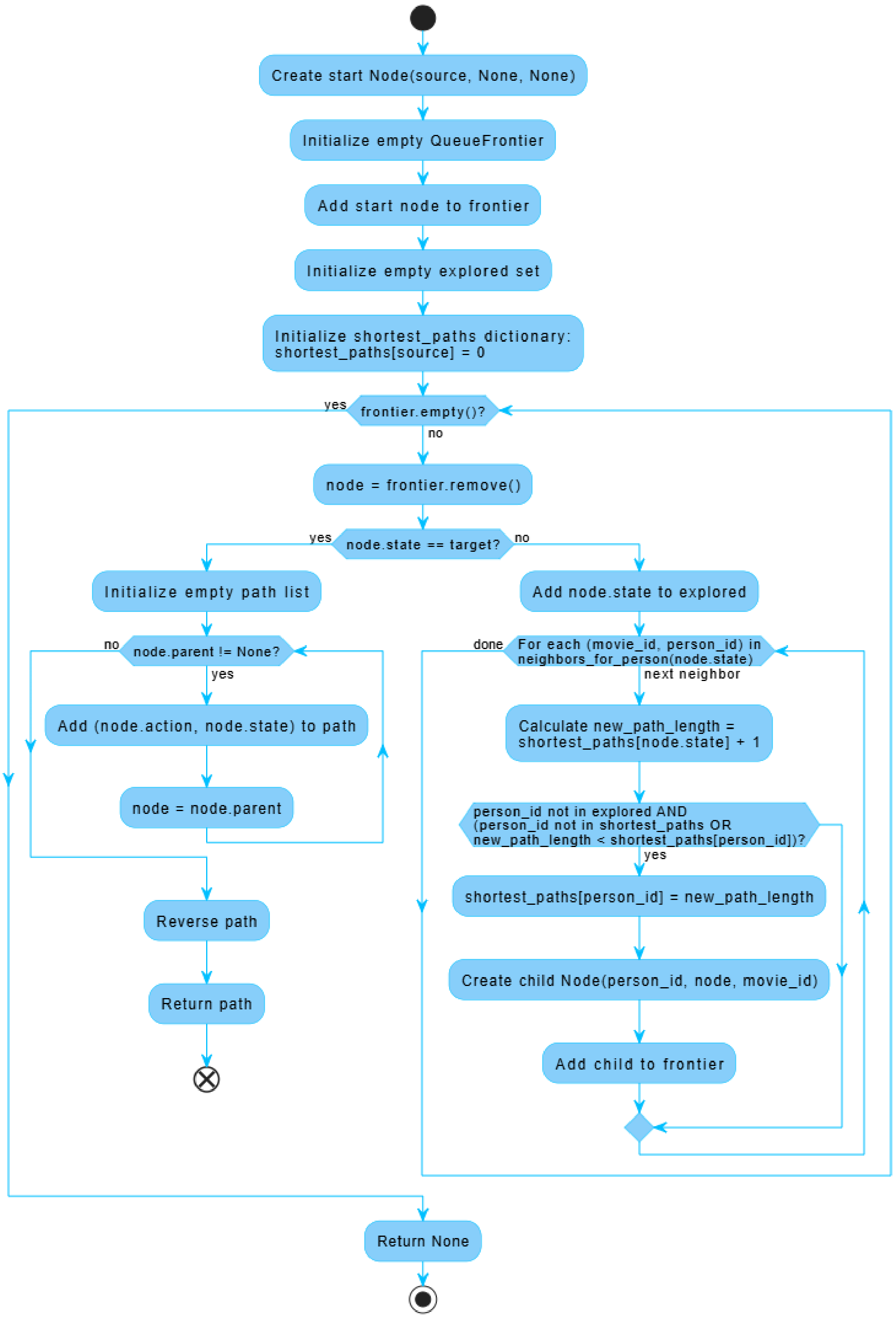
Factor	Explanation
Shortest Path Guarantee	BFS ensures that the first path to reach the target is the shortest in unweighted graphs.
Level-by-Level Processing	Nodes are explored level by level, preventing deeper, longer paths from being selected first.
Simplicity	Straightforward to implement without additional heuristics or complex structures.

3. Initial Implementation

3.1 Approach

The `shortest_path` function was implemented using **Breadth-First Search (BFS)**. BFS explores nodes level-by-level to
ensure the shortest path is found first. Nodes are added to a queue and processed until the target node is reached.

- Flow Chart



3.2 Implementation

```
def shortest_path_bfs_simple(source, target)
```

Explanation:

- **Initialize**: Start with the `source` node in the frontier and an empty `explored` set to track visited nodes.
- **Loop until target**: Remove the first node from the queue, check if it's the target.
- **Path reconstruction**: If the target is reached, build the path by following the `parent` pointers from the target
back to the source.
- **Explore neighbors**: For each neighbor (connected actor), add it to the frontier if it hasn't been explored.
- **No path case**: If the queue is empty and the target hasn't been reached, return `None`.

3.3 Issues Encountered

During testing, high time and memory usage were observed in some cases:

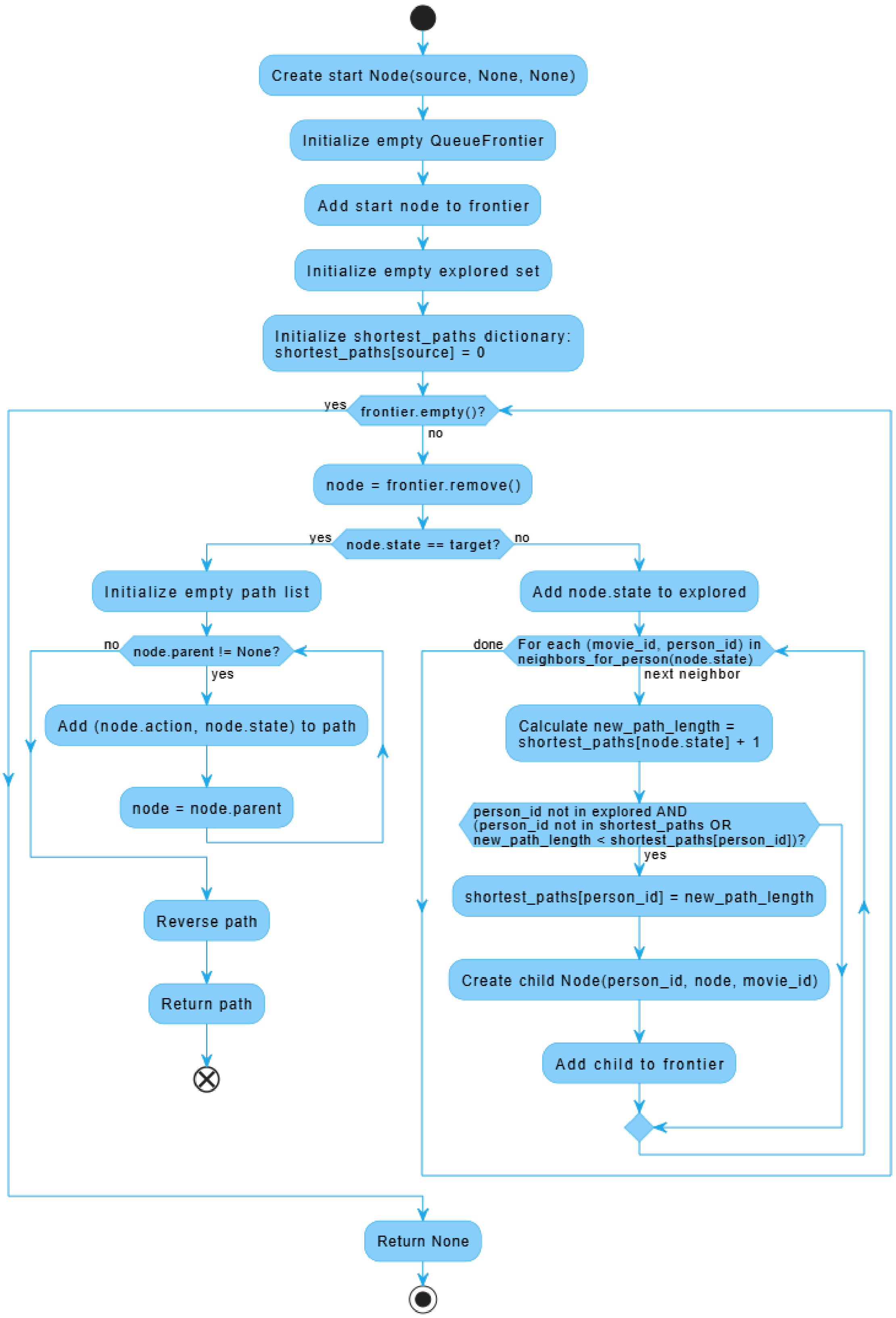
- **Test Case**: Finding the connection between "**Ingmar Bergman**" and "**Nino Rota**".
  - **Problem**: No direct or indirect connection exists between these two actors, but BFS exhaustively scans the
entire search space.
  - **Outcome**: Excessive time was required to conclude the absence of a path, highlighting a need for optimization
in cases where no path exists.
  - **Time Consume**: **6 hours** to find no connection between "**Ingmar Bergman**" and "**Nino Rota**".

4. Optimizations

Due to the inefficiencies observed in the initial BFS implementation, we applied two main optimization strategies to
improve performance.

4.1 BFS with Pruning

- **Idea**: Use pruning to avoid redundant paths and reduce the search space by only expanding nodes that can lead to a
shorter or unique path.
- Flow Chart:



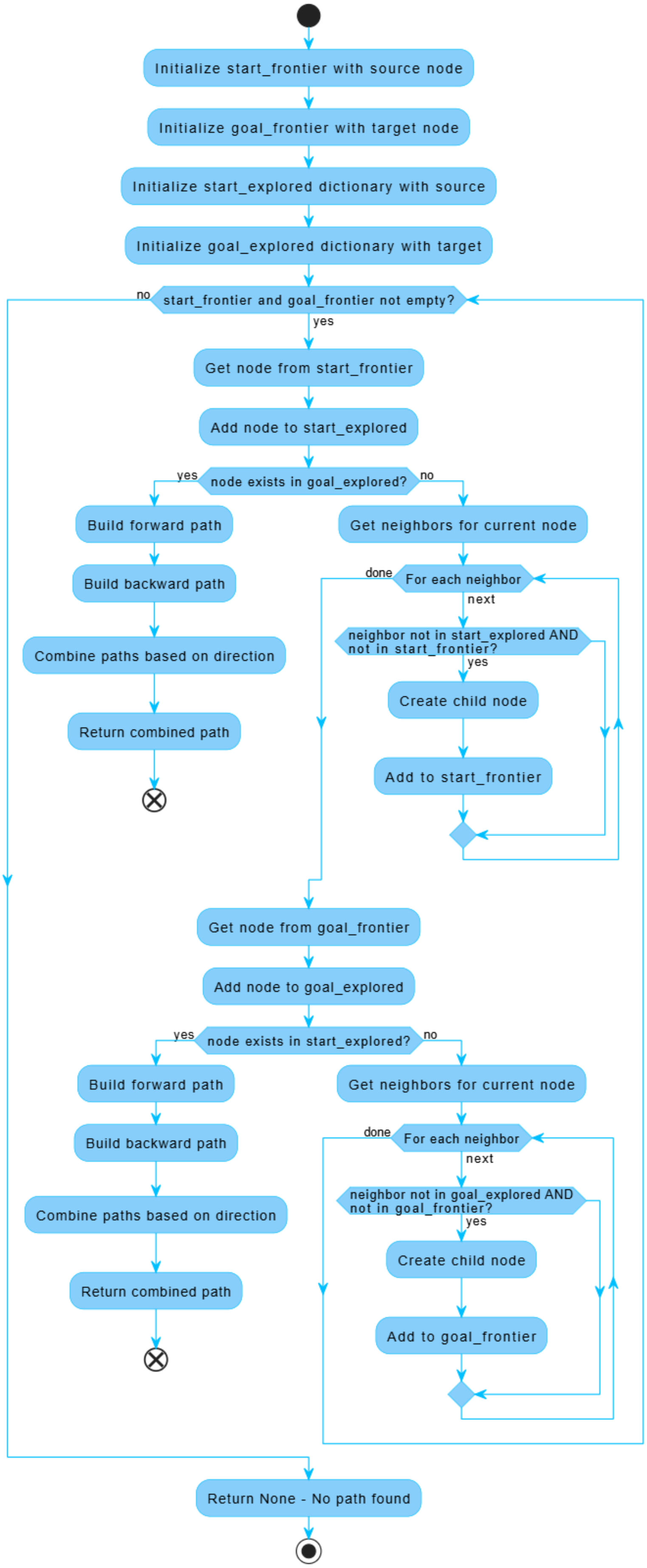
- **Implement**:
  - A `shortest_paths` dictionary is used to track the minimum known path to each node.

  - Each neighbor is only added to the frontier if it leads to a shorter or new path. This effectively "prunes"
redundant paths and reduces the number of nodes expanded.

- **Outcome**:
  - Pruning reduced some redundant searches, but for larger networks with no connections, the time cost remained high.

4.2 Bidirectional Search

- **Idea**: Perform BFS from both the source and target nodes simultaneously, reducing the search space by half and
meeting at an intersection point.
- Flow Chart:


Initialize start_frontier with source node
Initialize goal_frontier with target node
Initialize start_explored dictionary with source
Initialize goal_explored dictionary with target
start_frontier and goal_frontier not empty?
no
yes
Get node from start_frontier
Add node to start_explored
node exists in goal_explored?
yes
no
Build forward path
Build backward path
Combine paths based on direction
Return combined path
Get neighbors for current node
For each neighbor
done
next
neighbor not in start_explored AND
not in start_frontier?
yes
Create child node
Add to start_frontier
Get node from goal_frontier
Add node to goal_explored
node exists in start_explored?
yes
no
Build forward path
Build backward path
Combine paths based on direction
Return combined path
Get neighbors for current node
For each neighbor
done
next
neighbor not in goal_explored AND
not in goal_frontier?
yes
Create child node
Add to goal_frontier
Return None - No path found

- Implementation Details
    - `initialize_frontiers`
        - Sets up two search frontiers: one for the source (`start_frontier`), one for the target (`goal_frontier`).
        - Allows bidirectional search from both ends.
    - `initialize_explored_dicts`
        - Creates two dictionaries to track visited nodes: `start_explored` (from source) and `goal_explored` (from target).
        - Each dictionary begins with its respective start node.
    - `search_step`
        - Expands nodes in the specified direction (either "forward" or "backward").
        - Removes a node from the frontier, adds it to `explored`.
        - Checks for intersection with `other_explored` from the opposite direction.
        - If intersection found, calls `build_path` to construct the path.
        - Adds unexplored neighbors to the frontier for further expansion.
    - `build_path`
        - Merges paths from forward and backward searches at the intersection.
        - Follows parent pointers to reconstruct paths from each side.
        - Combines `path_forward` and `path_backward` to form a complete path from source to target.
- **Outcome**: Bidirectional Search effectively halved the search space, significantly improving time and memory
  efficiency, especially in large networks or cases with no connection.

5. Conclusion

5.1 Comparison of Implementation Approaches

Search Method	Description	Optimization Technique	Pros	Cons

Simple BFS	Level-by-level exploration	None	Guarantees shortest path	High memory and time cost for large graphs
BFS with Pruning	Avoids redundant paths	Shortest path tracking	Reduces redundant searches, improving efficiency	Increased memory overhead for path tracking
Bidirectional BFS	Explores from both start and end	Two-directional search	Significant reduction in search space and steps	More complex implementation, requires dual tracking

Table 2: Test Case Comparison

- Case 1: Emma Watson and Jennifer Lawrence
- Case 2: Ingmar Bergman and Nino Rota

Search Method	Test Case	Connection Found	Execution Time
Simple BFS	Case 1	Yes	**730 seconds**
	Case 2	No	**21 600 seconds**
BFS with Pruning	Case 1	Yes	**1.56 seconds**
	Case 2	No	**600 seconds**
Bidirectional BFS	Case 1	Yes	**0.048 seconds**
	Case 2	No	**0.0 seconds**

Part 2: Best Algorithm Conclusion

- **Bidirectional BFS** is the most suitable algorithm due to:
  - Efficient reduction in search space, making it highly effective in large or sparse networks.
  - Faster execution time with fewer steps when a path exists.
  - Ability to handle cases with no connection more gracefully compared to Simple BFS and BFS with Pruning.