**System Requirements Specification Index**

**For**

**Social Network Graph Analysis System**

**Version 1.0**

**IIHT Pvt. Ltd.**

**fullstack@iiht.com**

**Table of Contents**

1 Project Abstract

2 Business Requirements

3 Constraints

4 Template Code Structure

5 Execution Steps to Follow

**Social Network Graph Analysis**

**System Requirements Specification**

**1** **Project Abstract**

NetConnect Analytics requires a specialized set operations system to analyze social network connections, community structures, and information flow patterns. The system will identify friend circles, analyze mutual connection networks, detect community clusters, and track information propagation through social graphs. This tool will enable researchers to identify influential nodes, understand connection patterns, and predict information spread through complex social networks.

**2** **Business Requirements:**

| Screen Name | Console input screen |
| --- | --- |
| Problem Statement | 1. System needs to store and analyze different types of network data (users, connections, communities, interaction metrics)  2. System must support filtering connections by relationship type, interaction frequency, or community membership  3. Console should handle different set operations like Basic set operations (friend lists, mutual connections), Set comprehensions (for filtering specific user patterns), Set methods (add, remove, intersection, union), Set relationship tests (analyzing network overlaps), Set-based network metrics calculation |

**3** **Constraints**

**3.1** **Input Requirements**

1. User Sets:

o Must be stored as sets of strings representing user IDs

o Example: `network\_users = {"user1", "user2", "user3", "user4", "user5"}`

2. Connection Sets:

o Must be stored as sets of connections between users

o Example: `user1\_connections = {"user2", "user3", "user4"}`

3. Community Sets:

* + Must be stored as sets representing community memberships
  + Example: `tech\_community = {"user1", "user3", "user5", "user7"}`

4. Predefined Network Sets:

* + Must use these exact predefined sets:
    - `network\_a = {"user1", "user2", "user3", "user4", "user5", "user6", "user7"}`
    - `network\_b = {"user5", "user6", "user7", "user8", "user9", "user10"}`
    - `tech\_group = {"user1", "user3", "user5", "user8", "user10"}`
    - `gaming\_group = {"user2", "user4", "user6", "user8", "user9"}`
    - `arts\_group = {"user3", "user5", "user7", "user10"}`

5. Connection Maps:

* + Must use these exact connection mappings:
    - `user1\_connections = {"user2", "user3", "user5"}`
    - `user2\_connections = {"user1", "user4", "user6"}`
    - `user3\_connections = {"user1", "user5", "user7"}`
    - `user4\_connections = {"user2", "user6"}`
    - `user5\_connections = {"user1", "user3", "user7", "user8"}`

6. New Network Sets:

* + Must use these exact predefined items:
    - `new\_users = {"user11", "user12", "user13"}`
    - `influencers = {"user3", "user5", "user8", "user11"}`

**3.2** **Operations Constraints**

**1.** Set Creation:

○ Must use proper set creation syntax

○ Example: `{"user1", "user2", "user3"}` or `set(["user1", "user2", "user3"])`

**2.** Set Access:

○ Must use proper set membership testing

○ Example: `if user in user\_set:` or `for user in user\_set:`

**3.** Set Operations:

○ Must use standard set operations

○ Example: Union: `A | B` or `A.union(B)`, Intersection: `A & B` or `A.intersection(B)`

**4.** Set Comprehensions:

○ Must use set comprehensions where appropriate

○ Example: `{user for user in network\_a if user in tech\_group}`

**5.** Set Methods:

○ Must use appropriate set methods

○ Example: `user\_set.add(new\_user)`, `user\_set.remove(user)`, `user\_set.update(new\_users)`

**6.** Set Relationships:

○ Must test set relationships correctly

○ Example: `network\_a.issuperset(tech\_group)`, `user1\_connections.isdisjoint(user4\_connections)`

**7.** Set-Based Analytics:

○ Must use sets for network analysis

○ Example: `len(user1\_connections & user2\_connections)` to count mutual friends

**8.** Error Handling:

○ Must handle set operation errors appropriately

○ Example: Using `.discard()` instead of `.remove()` to avoid KeyError

**9.** Immutability:

○ Must respect set operation principles

○ Example: Creating new sets for results rather than modifying inputs

**10.** Set Theory Applications:

○ Must apply set theory to network problems

○ Example: Using symmetric difference to find exclusive connections

**3.3** **Output Constraints**

1. Display Format:

o Show network analysis results with clear formatting

o Format large user sets with appropriate separators

o Each analysis result must be displayed on a new line

**2.** Output Format:

o Must show in this order:

§ Show "== SOCIAL NETWORK GRAPH ANALYSIS =="

§ Show "Total Users: {count}"

§ Show "Available Networks: {networks}"

§ Show "Current Network Data:"

§ Show sets with format: "{network\_name}: {users}"

§ Show "Analysis Result:" when displaying operation results

**4. Template code Structure:**

**1.** Data Management Functions:

o `initialize\_data()` - creates the initial network sets and connection data

**2.** Dictionary Operation Functions:

o `find\_mutual\_connections(user\_a\_connections, user\_b\_connections)` - finds mutual connections between users

o `find\_exclusive\_connections(user\_a\_connections, user\_b\_connections)` - finds connections unique to each user

o `find\_all\_connections(connections\_list)` - merges all connections from a list

o `find\_common\_groups(user, group\_dict)` - finds all groups a user belongs to

o `is\_direct\_connection(user\_a, user\_b, connections\_dict)` - checks if users are directly connected

o `is\_second\_degree\_connection(user\_a, user\_b, connections\_dict)` - checks for connections through a mutual friend

o `calculate\_connection\_strength(user\_a, user\_b, interaction\_data)` - calculates connection strength

o `find\_community\_overlaps(group\_a, group\_b)` - finds users in multiple communities

o `identify\_bridge\_users(communities\_list)` - identifies users that connect multiple communities

o `calculate\_network\_density(users, connections\_dict)` - calculates network density

o `find\_isolated\_users(users, connections\_dict)` - finds users with no connections

o `recommend\_connections(user, all\_users, connections\_dict)` - recommends new connections

**3.** Display Functions:

o `format\_users\_for\_display(group\_name, users)` - formats users for display

o `display\_analysis\_result(operation, set\_a, set\_b, result)` - displays analysis results

o `display\_data(data, data\_type)` - displays sets or other data types

**4.** Program Control Functions:

o `main()` - main program function

# **5. DETAILED FUNCTION IMPLEMENTATION GUIDE**

## **5.1 Data Initialization Functions**

### **1. Write a Python function to initialize the network data with predefined sets.**

**Define:** initialize\_data()  
 The function should:

* Create a set named network\_a containing exactly these users: {"user1", "user2", "user3", "user4", "user5", "user6", "user7"}
* Create a set named network\_b containing exactly these users: {"user5", "user6", "user7", "user8", "user9", "user10"}
* Create a set named tech\_group containing exactly these users: {"user1", "user3", "user5", "user8", "user10"}
* Create a set named gaming\_group containing exactly these users: {"user2", "user4", "user6", "user8", "user9"}
* Create a set named arts\_group containing exactly these users: {"user3", "user5", "user7", "user10"}
* Create a dictionary named connections with these exact mappings:
  + "user1": {"user2", "user3", "user5"}
  + "user2": {"user1", "user4", "user6"}
  + "user3": {"user1", "user5", "user7"}
  + "user4": {"user2", "user6"}
  + "user5": {"user1", "user3", "user7", "user8"}
  + "user6": {"user2", "user4"}
  + "user7": {"user3", "user5"}
  + "user8": {"user5", "user9", "user10"}
  + "user9": {"user8"}
  + "user10": {"user8"}
* Create a set named new\_users containing: {"user11", "user12", "user13"}
* Create a set named influencers containing: {"user3", "user5", "user8", "user11"}
* Return a tuple containing (network\_a, network\_b, tech\_group, gaming\_group, arts\_group, connections, new\_users, influencers)
* All data must use proper set structures with curly braces {}

## **5.2 Set Intersection Operations**

### **2. Write a Python function to find mutual connections between two users.**

**Define:** find\_mutual\_connections(user\_a, user\_b, connections)  
 The function should:

* Validate that connections is not None, raise ValueError if None
* Validate that user\_a exists in connections dictionary, raise ValueError with message f"User {user\_a} not found in connections"
* Validate that user\_b exists in connections dictionary, raise ValueError with message f"User {user\_b} not found in connections"
* Use set intersection operator & or .intersection() method to find common connections
* Access user connections using dictionary indexing: connections[user\_a] and connections[user\_b]
* Return the intersection of both users' connection sets: connections[user\_a] & connections[user\_b]
* Use variable name mutual\_connections for intermediate results
* Handle empty intersection by returning empty set
* Example: if user1 connects to {user2, user3, user5} and user3 connects to {user1, user5, user7}, mutual is {user5}

### **3. Write a Python function to find users that are members of both groups.**

**Define:** find\_common\_group\_members(group\_a, group\_b)  
 The function should:

* Validate that group\_a is not None, raise ValueError if None
* Validate that group\_b is not None, raise ValueError if None
* Use set intersection operator & or .intersection() method to find common members
* Return the intersection of both group sets: group\_a & group\_b
* Use variable name common\_members for the result
* Handle empty groups by returning empty set
* Preserve immutability by not modifying input sets
* Example: tech\_group ∩ gaming\_group = users in both groups

## **5.3 Set Union Operations**

### **4. Write a Python function to find users that are members of either group.**

**Define:** find\_users\_in\_any\_group(group\_a, group\_b)  
 The function should:

* Validate that group\_a is not None, raise ValueError if None
* Validate that group\_b is not None, raise ValueError if None
* Use set union operator | or .union() method to combine groups
* Return the union of both group sets: group\_a | group\_b
* Use variable name all\_users for the result
* Handle empty groups correctly (union with empty set returns the other set)
* Preserve immutability by not modifying input sets
* Example: tech\_group ∪ gaming\_group = all users in either group

## **5.4 Set Symmetric Difference Operations**

### **5. Write a Python function to find connections exclusive to each user.**

**Define:** find\_exclusive\_connections(user\_a, user\_b, connections)  
 The function should:

* Validate that connections is not None, raise ValueError if None
* Validate that user\_a exists in connections dictionary, raise ValueError with message f"User {user\_a} not found in connections"
* Validate that user\_b exists in connections dictionary, raise ValueError with message f"User {user\_b} not found in connections"
* Use symmetric difference operator ^ or .symmetric\_difference() method
* Access user connections using dictionary indexing: connections[user\_a] and connections[user\_b]
* Return connections that are unique to either user: connections[user\_a] ^ connections[user\_b]
* Use variable name exclusive\_connections for the result
* Handle cases where users have no connections by returning the other user's connections
* Example: if user1 has {user2, user3, user5} and user2 has {user1, user4, user6}, exclusive is {user3, user4, user5, user6}

## **5.5 Set Membership Testing**

### **6. Write a Python function to check if two users are directly connected.**

**Define:** is\_direct\_connection(user\_a, user\_b, connections)  
 The function should:

* Validate that connections is not None, raise ValueError if None
* Validate that user\_a exists in connections dictionary, raise ValueError with message f"User {user\_a} not found in connections"
* Use the in operator to check membership: user\_b in connections[user\_a]
* Return boolean True if directly connected, False otherwise
* Use variable name is\_connected for intermediate boolean results
* Handle cases where user has no connections (empty set) by returning False
* Consider connections as bidirectional (if A connects to B, then B connects to A)
* Example: is\_direct\_connection("user1", "user2", connections) returns True if user2 is in user1's connection set

### **7. Write a Python function to check if two users are connected through a mutual friend.**

**Define:** is\_second\_degree\_connection(user\_a, user\_b, connections)  
 The function should:

* Validate that connections is not None, raise ValueError if None
* Validate that user\_a exists in connections dictionary, raise ValueError with message f"User {user\_a} not found in connections"
* Validate that user\_b exists in connections dictionary, raise ValueError with message f"User {user\_b} not found in connections"
* First check if users are directly connected using is\_direct\_connection(), return False if they are
* Iterate through user\_a's connections to find mutual friends
* For each friend in connections[user\_a], check if user\_b is in connections[friend]
* Return True if any mutual friend connects both users, False otherwise
* Use variable name has\_mutual\_friend for boolean checks
* Handle cases where intermediate users don't exist in connections
* Example: user1 → user2 → user4 means user1 and user4 are second-degree connected

## **5.6 Advanced Set Operations**

### **8. Write a Python function to find all connections up to a certain depth.**

**Define:** find\_all\_connections(user, connections, depth=1)  
 The function should:

* Validate that connections is not None, raise ValueError if None
* Validate that user exists in connections dictionary, raise ValueError with message f"User {user} not found in connections"
* Validate that depth is at least 1, raise ValueError if less than 1
* Initialize all\_connections set to store results
* For depth=1, return direct connections: connections[user]
* For depth>1, use iterative approach with frontier expansion
* Track current frontier and expand to next level connections
* Use variable names current\_depth, frontier, new\_frontier
* Exclude the original user and previously found connections to avoid cycles
* Return set of all unique connections up to specified depth
* Handle cases where connections don't exist by checking if connection in connections

### **9. Write a Python function to identify users that connect multiple communities.**

**Define:** identify\_bridge\_users(communities)  
 The function should:

* Validate that communities is not None, raise ValueError if None
* Create dictionary user\_communities to track which communities each user belongs to
* Iterate through each community in the communities dictionary
* For each user in each community, add the community name to their tracking set
* Use set operations to efficiently track community memberships
* Filter results to only users belonging to more than one community: len(comms) > 1
* Return dictionary with users as keys and set of communities as values
* Use variable names user\_communities and bridge\_users
* Handle empty communities by returning empty dictionary
* Example: if user3 is in both tech\_group and arts\_group, they are a bridge user

### **10. Write a Python function to calculate network density.**

**Define:** calculate\_network\_density(users, connections)  
 The function should:

* Validate that users is not None, raise ValueError if None
* Validate that connections is not None, raise ValueError if None
* Validate that users set is not empty, raise ValueError if empty
* Calculate maximum possible connections using formula: n \* (n-1) / 2 where n is number of users
* Handle single user case by returning 0.0
* Count actual connections by iterating through user pairs
* Use variable name max\_possible\_connections for theoretical maximum
* Use variable name actual\_connections for counted connections
* Create set counted\_pairs to avoid double-counting in undirected graph
* For each connection, create order-independent pair identifier: tuple(sorted([user, connection]))
* Return density ratio: actual\_connections / max\_possible\_connections
* Result should be float between 0.0 and 1.0

## **5.7 Set Filtering and Analysis**

### **11. Write a Python function to find users with no connections.**

**Define:** find\_isolated\_users(users, connections)  
 The function should:

* Validate that users is not None, raise ValueError if None
* Validate that connections is not None, raise ValueError if None
* Initialize empty set isolated to store results
* Check each user for isolation in two ways:
  1. User not in connections dictionary OR user has empty connection set
  2. No other user connects to this user (check incoming connections)
* Use set comprehension or iteration to find users with no outgoing connections
* Use all() function to check if no other users connect to this user
* Return set of users that have no incoming or outgoing connections
* Use variable name isolated for the result set
* Handle cases where connections dictionary is incomplete
* Example: if user10 appears in users but has no connections, they are isolated

### **12. Write a Python function to recommend new connections based on friends of friends.**

**Define:** recommend\_connections(user, connections, depth=2)  
 The function should:

* Validate that connections is not None, raise ValueError if None
* Validate that user exists in connections dictionary, raise ValueError with message f"User {user} not found in connections"
* Get all connections up to specified depth using find\_all\_connections()
* Get direct connections: connections[user]
* Calculate recommendations by removing direct connections and the user themselves
* Use set difference operations: all\_connections - direct\_connections - {user}
* Use variable names all\_connections, direct\_connections, recommendations
* Return set of recommended users (friends of friends who aren't direct friends)
* Handle cases where no recommendations exist by returning empty set
* Example: if user1 connects to user2, and user2 connects to user4, recommend user4 to user1

## **5.8 Display and Formatting Functions**

### **13. Write a Python function to format a user set for display.**

**Define:** format\_users\_for\_display(group\_name, users)  
 The function should:

* Validate that users is not None, raise ValueError if None
* Sort users alphabetically using sorted(users)
* Join users with commas and spaces: ", ".join(sorted(users))
* Format with curly braces to show set notation: f"{group\_name}: {{{formatted\_users}}}"
* Use variable name formatted\_users for the joined string
* Handle empty sets by showing empty braces: "Group: {}"
* Return formatted string ready for display
* Example: "Tech Group: {user1, user3, user5, user8, user10}"

### **14. Write a Python function to display set operation results.**

**Define:** display\_analysis\_result(operation, set\_a, set\_b, result)  
 The function should:

* Print operation description with newline: f"\nAnalysis Result: {operation}"
* Display first set: f"Set A: {set\_a}"
* Display second set: f"Set B: {set\_b}"
* Display result set: f"Result: {result}"
* Use consistent formatting for all set displays
* Handle empty sets appropriately in display
* Ensure proper spacing between sections
* Example output format for intersection operation

### **15. Write a Python function to display formatted data based on data type.**

**Define:** display\_data(data, data\_type)  
 The function should:

* Validate that data is not None, print "No data to display." and return if None
* Handle different data\_type values using if/elif statements:
  + "networks": Display network dictionary with format\_users\_for\_display()
  + "connections": Display user connections as "user → connections"
  + "mutual\_connections": Display mutual connections between two users
  + "bridge\_users": Display users connecting multiple communities
  + "recommendations": Display connection recommendations for a user
* Use appropriate headers for each data type
* Sort users and connections for consistent display
* Handle empty data by showing appropriate messages
* Use variable names like network\_name, user, connections for iteration
* Print each item on a separate line with proper formatting

## **5.9 Main Program Function**

### **16. Write a Python function to demonstrate the social network graph analysis system.**

**Define:** main()  
 The function should:

* Call initialize\_data() to get all network data sets and connections
* Unpack the tuple: network\_a, network\_b, tech\_group, gaming\_group, arts\_group, connections, new\_users, influencers = initialize\_data()
* Create networks dictionary for easy access to all groups
* Calculate total users by creating union of all network sets: all\_users = set()
* Display system header: "===== SOCIAL NETWORK GRAPH ANALYSIS ====="
* Show statistics: total users count and available networks
* Display menu options numbered 0-5:
  + "1. View Network Data"
  + "2. Analyze Connections"
  + "3. Find Community Patterns"
  + "4. Calculate Network Metrics"
  + "5. Generate Recommendations"
  + "0. Exit"
* Implement comprehensive menu handling with nested submenus
* For each option, demonstrate corresponding set operations functions
* Handle user input validation using try/except blocks with ValueError
* Use appropriate variable names: choice, user\_a, user\_b, group\_name, etc.
* Loop until user selects exit option (0)
* Show results using display\_data() and display\_analysis\_result() functions
* Print "Thank you for using the Social Network Graph Analysis System!" on exit

**6. Execution Steps to follow:**

1. Run the program

2. View the main menu

3. Select operations:

- Option 1: View Network Data

- Option 2: Analyze Connections

- Option 3: Find Community Patterns

- Option 4: Calculate Network Metrics

- Option 5: Generate Recommendations

- Option 0: Exit

4. Perform operations on the network data

5. View results after each operation

6. Exit program when finished

Execution Steps to Follow:

● All actions like build, compile, running application, running test cases will be through Command Terminal.

● To open the command terminal the test takers, need to go to Application menu (Three horizontal lines at left top) -> Terminal -> New Terminal

● This editor Auto Saves the code

● If you want to exit(logout) and continue the coding later anytime (using Save & Exit option on Assessment Landing Page) then you need to use CTRL+Shift+B -command compulsorily on code IDE. This will push or save the updated contents in the internal git/repository. Else the code will not be available in the next login.

● These are time bound assessments the timer would stop if you logout and while logging in back using the same credentials the timer would resume from the same time it was stopped from the previous logout.

● To launch application: python3 filename.py

● To run Test cases: python3 -m unittest

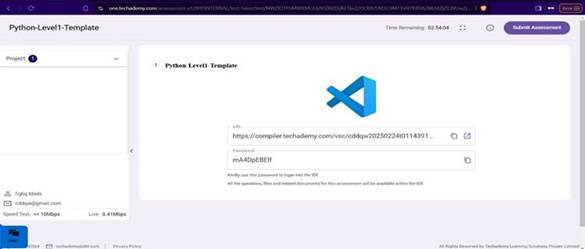
● Before Final Submission also, you need to use CTRL+Shift+B - command compulsorily on code IDE, before final submission as well. This will push or save the updated contents in the internal git/repository, and will be used to evaluate the code quality.

Screen shot to run the program

To run the application

Python3 filename.py

To run the testcase python -m unittest



● Once you are done with development and ready with submission, you may navigate to the previous tab and submit the workspace. It is mandatory to click on “Submit Assessment” after you are done with code.