Universitat Pompeu Fabra Barcelona

DATA STRUCTURES AND ALGORITHMS II, 2022-2023

MUNG BEAN: Where Vegans and Vegetarians Connect

Welcome to Mung Bean, where vegans and vegetarians connect.
Join us in embracing a compassionate lifestyle!

Feryjhon Abanilla, u213917
Jorge Camacho, u21395
Ramon Casas, u103898
DATE OF SUBMISSION: 11/06/2023

TABLE OF CONTENTS

INTRODUCTION

For this project, we aim to develop a console-based social network that allows users to create profiles and simulate the actions which those profiles can do. The problem we face is that we must create an efficient and lightweight social network platform. Our objective and proposals are based on creating an intuitive and interactive social network. Users will have the ability to create profiles, add friends, and make posts within the network. By implementing a linked list data structure, we can efficiently manage user connections and navigate through the network. In this report, we are going to analyze and explain how our network works. We will explain our data architecture, the main functionalities and the solutions that we implemented for each main objective.

PROJECT OBJECTIVES

This section should cover the objectives met by the project.
***Each objective** presented in the below subsections should cover the following items:*

- *Overview: Describe how it was implemented. Beyond what was given to you in the guidance document, please indicate the main characteristics of the data structures and algorithms implemented towards this objective and its behaviour. This should include:*
  - *Which variables were used and for which purpose?*
  - *What were the data structures chosen for this objective? What is their expected use?*
  - *What was the algorithm chosen for this objective? What is its expected behaviour? What about its performance in Big O?*
  - *What limitations does the algorithm have? What about its implementation?*
  - *What can be improved?*
- *Time: Time required for developing this objective*
- *Location: What line of code and of which file this implementation belongs to.*

Mandatory objectives met

In Mung Bean, we strongly believe in persistence, so we have implemented all mandatory objectives possible.

Mandatory Objective 1: Implement at least one List, a Stack, and a Queue functional as part of some of the functionalities of the project.

We have implemented various lists, one stack and one queue.
The stack has been implemented in order to be one of the options that all users have. That function is the option of getting three different users as options to send them requests to be friends. We coded the main functions that describe the stack as we can see in *Figure 1*.

```
63      //STACK FUNCTIONS
64      void init_stack(Stack* stack);
65      int is_full(Stack stack); //??????
66      int is_empty(Stack stack);
67      User top(Stack stack);
68      void pop(Stack* stack);
69      void push(Stack* stack, User user);
```

Figure 1. All functions used to implement the stack, screenshot from the file "user.h".

These functions are used when any user chooses the option of adding random friends. When that happens, we initialize the stack. Stack is a data structure as follows in *figure 2.*

```
51  //stack of users to add unknown friends
52  typedef struct{
53      User* users;
54      int top;
55  }Stack;
```

Figure 2. Data structure of the Stack. Screenshot from file "user.h".

It contains the index of the top and a dynamic array of type "User" which will be explained later. After initializing it, we fill it with three random users. We ask the acting user whether he/she/they wants to send a friend request to that user. That is done by iterating in the stack following the LIFO principle, and each time we show a possibility and the user decides, we update the top index until the stack is empty.

The data structure of the stack is implemented in *user.h* in the lines 51-55. The functions in *figure 1* can be seen how they are used in the lines 126-164 in *menu.c.*

A queue has been implemented in order to manage the friend requests that the users get. Once again we coded the main functions that describe a Queue in order to follow the FIFO principle, as we can see in *Figure 3.*

```
71    //QUEUE AND REQUESTS FUNCTIONS
72    void init_queue(Requests* requests);
73    void dequeue(Requests* requests);
74    void enqueue(Requests* requests, Node_request* request);
75    int is_empty_queue(Requests requests);
76    char* first_queue(Requests requests);
```

Figure 3. Functions used to implement the queue. Screenshot from "user.h"

These functions will act over the data structure used for managing the friend requests, which is the one on Figure 4. We can basically see that it is a linked list, where each node has the name of the user requesting a friendship and the pointer to the next node.

```
14    typedef struct node_request{
15        char name[MAX_LENGTH];
16        struct node_request *next_request; // Pointer to next user
17    }Node_request;
18
19    typedef struct{
20        Node_request* first_request;
21        Node_request* last_request;
22        int num_requests;
23    }Requests;
```

Figure 4. Data structure for managing the friend requests, implemented as a linked list. Screenshot from user.h file.

These functions are used when the user wants to manage his requests of friends. Following the FIFO principle, first, we show him/her/them the first person added to the queue, giving the option of accepting the friendship or not or quitting that menu to do something else. Each time that the user accepts or declines a request queue, we delete the first person on the queue, and we update the first person or node of the link. On the other hand, when a user gets friend requests, we add that request as the last element of the list or as the first if the queue is empty.

The implementation of the linked list where the queue acts is in the lines shown by *figure 4,* in the file user.h and the same for the functions of the queue. To see how we use the functions of the queue, we can go to the *menu.c* file to the lines 124-125, where we call the function *send_requests,* found in *community.c* starting in line 362.

About the lists, the main structure to manage how we save our users is a List, as we can see in *Figure 5.* The list, called community, contains nodes called *User_list.* The declaration of this linked list can be found in community.h at the start of that file. This dynamic list saves the first node of the list and the last one, and also all the posts made by the users. Moreover, it contains the data of how many users there are and how many posts there are. On the other hand, each node saves all the data of one user, and the pointer to the previous and next node. The previous node is not used, but it has been added just in case we needed to use any algorithm that demands it.

```
16  //Linked List of users
17  typedef struct user_list{
18      User user;
19      struct user_list *next_user; // Pointer to next user
20      struct user_list *prev_user; // Pointer to previous use
21      //struct user_list *request_list; //friend request list
22  }User_list;
23
24  typedef struct{
25      User_list* first_node;
26      User_list* last_node;
27      int current_size;
28      int historic_size;
29      int posts_number;
30      int posts_capacity;
31      Post* posts;
32  }Community;
```

Figure 5. Screenshot of the implementation of the data structures that manage the users of our social network.

Objective 2: Search Algorithm.

We needed to implement a way to find an exact user in our community (a linked list). Therefore, we decide that implementing a Linear search will be the easiest way to do that. The next algorithm has a complexity of O(n). This function can be found on *community.h* and starts on the line 298. It gets as parameters the whole community by copy and the name of the user that we want to find. This function is used a lot of times. For example, when we want to send friend requests and manage them, to act as a specific user, when validating that the username is available or even when reading from the CSV files. The function works by navigating through the linked list going node by node and comparing their username with the one given a parameter, when they coincide we return the pointer to a node. That allows us to act as a specific user, send requests and manage requests properly.

```
298  User_list* linear_search(char* user_name, Community community){
299      User_list* temp = community.first_node;
300      while(temp != NULL){
301          if(strcmp(user_name,temp->user.username) == 0) {
302              return temp;
303          }
304          temp = temp->next_user;
305      }
306      return NULL;
307  }
```

Figure 5. Implementation of the algorithm of linear search

Objective 3: Sort Algorithm.

We coded the Quicksort sort algorithm. This algorithm is used in order to sort the hash table in decreasing order and has an O(n log(n)) complexity. Our Quicksort basically picks a pivot in the array and puts it in the correct position by letting the smaller numbers at the **right** and the bigger values at the **left**. We used Quicksort to order in a descending way because it makes more sense to do it like that regarding the purpose for what we made it. We used three functions in order to implement the algorithm, they can be found in *community.c* at lines 563-590. The function Quicksort gets as parameters a lower boundary and an upper one and also an array. In our case, that array is in fact a hash table that contains the items as char values, as we can see in *figure 6.* If the stop condition is not reached (the lower boundary being bigger than the upper limit) we pick the pivot by calling the function *partition*, and the function calls itself two times, creating two different arrays. Those arrays are divided by the pivot.

```
584  void Quicksort(hash_table_item **array, int inf, int sup) {
585      if (inf < sup) {
586          int pivot = partition(array, low: inf, high: sup);
587          Quicksort(array, inf, sup: pivot - 1);
588          Quicksort(array, inf: pivot + 1, sup);
589      }
590  }
```

Figure 6. Implementation of Quicksort function

The partition function purpose is to divide the array that it gets as a parameter into subarrays which are sorted respectively to the pivot. This function rearranges the elements in such a way that all elements smaller than the pivot are placed before it, and all elements larger than the pivot are placed after it. It returns the index of the next pivot.

```
569  int partition(hash_table_item **array, int low, int high) {
570      char* pivot = array[high]->value;
571      int i = low - 1;
572
573      for (int j = low; j <= high - 1; j++) {
574          if (strcmp(array[j]->value, pivot) >= 0) { // Modified condition for descending order based on value
575              i++;
576              swap( a: &array[i],  b: &array[j]);
577          }
578      }
579      swap( a: &array[i + 1],  b: &array[high]);
580      return (i + 1);
581  }
```

Figure 7. Partition function implementation.

A good way to improve the algorithm is to make a way to choose the pivot the most efficient way, searching for the median.

Objective 4: Hash table

For this project, we have imported a Hash Table made by github user maw101 (https://github.com/maw101/C-Hash-Table). This C hash table implementation uses open addressing (also known as closed hashing) and double hashing for collision resolution. For each collision we use an expression to determine the new index:

*index = (hash_a(key) + (i * (hash_b(key) + 1))) % hash_table_size*

In *hash_table.h* we can see the following data structures:

A *hash_table_item* contains a key-value pair of type *char **.
A *hash_table_table* contains in

```
// item struct for key-value pairs
typedef struct {
    char * key;
    char * value;
} hash_table_item;

// hash table struct
typedef struct {
    int size;
    int count;
    hash_table_item ** items;
} hash_table_table;
```

Figure 8. Hash table data structures

The main functions of the hash table are:

Creation

```C/C++
//function to create a hash_table, returns a hash_table_table*
hash_table_new()
//function to create an item, returns a hash_table_item*
hash_table_new_item(const char * k, const char * v);
```

Insertion

```C/C++
//function to insert a key and its value to hash table
hash_table_insert(hash_table_table * hash_table, const char * key, const
char * value)
```

Search

```C/C++
//function to search in hash table, return a char* with the key's
value, NULL if key is not found
hash_table_search(hash_table_table * hash_table, const char * key)
```

Deletion

```C/C++
//function to an item in hash table by key
hash_table_delete_key(hash_table_table * hash_table, const char *
key)
//function to delete the hash table
hash_table_delete_table(hash_table_table * hash_table)
```

Desirable objectives met

We have implemented functions in order to import data from a CSV file with data of the users in the social network so that we can load some users to test the code. At first, we were only able to load the main data for the users, that means that posts, friends and other things were not loaded. But we ended up also loading posts, friends and friend requests. We used csv files because we can divide the data into different fields and make it easier to organize our code. In *community.c* (lines 132 and 281) two functions for reading the files can be found. One is to create a user while loading their information. The other is to load the posts into the user that made it.

```
35  void read_csv(char* filename,Community *community);
36  void read_posts_csv(char* filename,Community *community, hash_table_table *hash_table, hash_table_table *trendingtopics, hash_table_table *stopwords);
```

Figure 9. Declaration of the reading functions.

The function *read_csv* navigates through a file with the next fields: “username", "birthyear", "email", "location", "tastes". Each field is one piece of data that all users must have defined. The function opens the file in reading mode and creates an auxiliary variable of type User where we will assign the information read. We use the c function *strtok* to delimit where fields end and start. Each field that we read is assigned to the corresponding variable of the user. Finally, that user is added to the Community of users by calling our function *add_new_user*. In the end, we close and open again the same file in order to load the friends of each user and their pending friend requests.

The function read_posts_csv is called after the other one. Once we have added all the existing users, we call this function to assign them their posts. We read from files with the following fields: "username", "datetime", "content". It works similarly.

We have also made the social network has as topic to be a forum for vegan people that wants to share opinions, recipes among other things with other people.

Exploratory objectives met

We believe that our most interesting exploratory objective is the implementation of a trending topic detector. Through the use of English stop words[1], we managed to develop a function that not only gives the most frequent words among the posts in our community, but that does it in a "smart" way, i.e. it omits frequent words and it only shows the really relevant words.

Observe the difference between a non-smart frequency word explorer and a smart one:

Non-smart	Smart
Word: the, Times mentioned: 9 Word: with, Times mentioned: 9 Word: vegan, Times mentioned: 7 Word: in, Times mentioned: 7 Word: and, Times mentioned: 7 Word: just, Times mentioned: 6 Word: plant-based, Times mentioned: 5 Word: new, Times mentioned: 5 Word: for, Times mentioned: 5 Word: my, Times mentioned: 4	Word: vegan, Times mentioned: 7 Word: plant-based, Times mentioned: 6 Word: life, Times mentioned: 2 Word: green, Times mentioned: 2 Word: chilling, Times mentioned: 2 Word: food, Times mentioned: 2 Word: enjoying, Times mentioned: 2 Word: nature, Times mentioned: 2 Word: exploring, Times mentioned: 2 Word: balanced, Times mentioned: 2

Another very important exploratory objective that we want to be very considered is the writing of data. We have various functions to add users and posts created in the running of the program so that the "progress" of our social network can be saved.

We have also implemented a more visual console display and a themed social network.

SOLUTION

In this section, we will explain how we reached a working solution. We will explain the main data structure that allows us to create users and how the main loop works, explaining all the options that you can choose to do actions.

We decided to implement a Linked List in order to manage the users that are signed in our social network. That linked list is called *Community.* It contains two pointers. The first is pointing to the first node of the list, the other to the last node of the list. Moreover, it contains the number of nodes contained in the structure and a dynamic array that saves "globally" the posts of the users next to the capacity and number of elements in that array.

Each node is called *User_list.* Each node contains a pointer to the next node and to the previous node and a variable of type User that will be explained later. That means, each node represents one specific user.

[1] https://en.wikipedia.org/wiki/Stop_word

The declaration of the data structures *Community and User_list*can be seen in *figure 5.*

Then we have the structure User. This structure is used to store all the necessary information of the user that allows us to do actions and differentiate it. It contains his username, it can not be repeated by other users, his birth year, location, email address and 5 tastes. Then it has some structures that store data such as the friends of the user, the posts made by the user or the pending friend requests he gets. The friends are stored in a dynamic array of type char, where each position is a username, the posts are saved in a dynamic array of type Post (a data typedef with the content, the username of the one who published the post and the datetime) and lastly, the requests are stored in a linked list that acts following the FIFO principle, meaning that is a queue (see *figure 4)*.

```c
typedef struct {
    char username[MAX_LENGTH];
    char datetime[DATE_TIME_LENGTH];
    char* content;
} Post;
```

```c
typedef struct{
    char username[MAX_LENGTH];
    int birth_year;
    char email[MAX_LENGTH];
    char location[MAX_LENGTH];
    int user_id;
    char tastes[MAX_SIZE][MAX_LENGTH];
    Requests requests;
    //USER FRIENDS
    int friends_number;
    int friends_capacity;
    char** friends_of_user;
    //USER POSTS
    int posts_number;
    int posts_capacity;
    //char** user_posts;
    Post* user_posts;
}User;
```

Figure 10. User and post data structures definitions.

Now we are going to explain how the execution works. When running the program it tries to load a list of users from a csv, adding them to a Linked list of type Community.The program creates a node, User_list, by allocating memory and filling it with all the needed data with the user readed in the file. Then it reads another file that contains the posts and assigns each post to the specific user that posted it and also to the array of posts in the community.

Various options are displayed in the console. This is done in the file *menu.c.* In this file are the most important functions that allow us to act in the social network. The options are in the function *menu():*

1. Create user. This option allows us to create from scratch a new member of the social network and add it to the linked list (community from now on).
2. List all existing users. It prints all the users that exist by iterating through the nodes and printing the data in each node.
3. Log in option. By entering a username we can make some actions simulating that we are that user. In order to do that we use the linear search algorithm explained before. That gives us a pointer to a node of the community, which is a user. Having the pointer allows us to modify its properties as we will explain later. It calls the function *user_menu().*
4. Frequent words. This option lists the most used words on posts. It can include connectors so it is not that useful. This is done by the hash tables functions explained before. We apply quicksort in order to print them in descending order.
5. Frequent topics. The same as option 4 but it uses a list of words to differentiate between important words and words with no important meaning. We apply quicksort in order to print them in descending order.

When we access as a specific user we get again a new menu (*user_menu()):*

1. Send requests. When selecting this option the user is asked for the username of the wanted friend. We check if the request is valid, meaning that it is send to someone existing or verifying that the request is not already sent to that same person or if the user is not sending the request to himself. Once verified that it is correct we call the function *send_requests*
2. Random Requests. This option calls the function *random_requests.* It basically uses the stack structure, filling it with random users. The usernames that are in the stack are moved into a temporal queue of friend requests that is displayed in the console and the user can accept to send the request or not.
3. Manage requests. This option calls the function *manage_requests*. This function basically accesses the linked lists of type requests inside each user and dequeues the first element while the user accepts or declines the request.
4. Make a post. This option calls the function *write_new_post.* This function gets the input of the user and checks if the length of the post is acceptable or not. If it is, it calls the hash functions in order to update the hash table with the new data, and then calls the function *add_post*. This function adds to the dynamic array of type posts that is inside each user. It checks if there is enough space. Then it also adds it to the dynamic array of posts in Community.
5. List posts. Prints the posts of the user acting.
6. See friends. It iterates in the array of friends of the user printing the usernames of all his friends.
7. Feed. This option calls the function *user_feed*. This function iterates through the global feed (the one of the community) and checks if the user that made each post is a friend of the user acting by calling the function *are_friends*. If they are friends we display the post and its info to the user.

Now we are going to analyze the most important functions. Functions that are used to check boolean values (for example if two users are friends) will be skipped. Also, some functions

as the sort and searching algorithms or the functions of the hash table as they will be explained or are already explained.

This section should dive deep into the solution developed for this project.

System Architecture

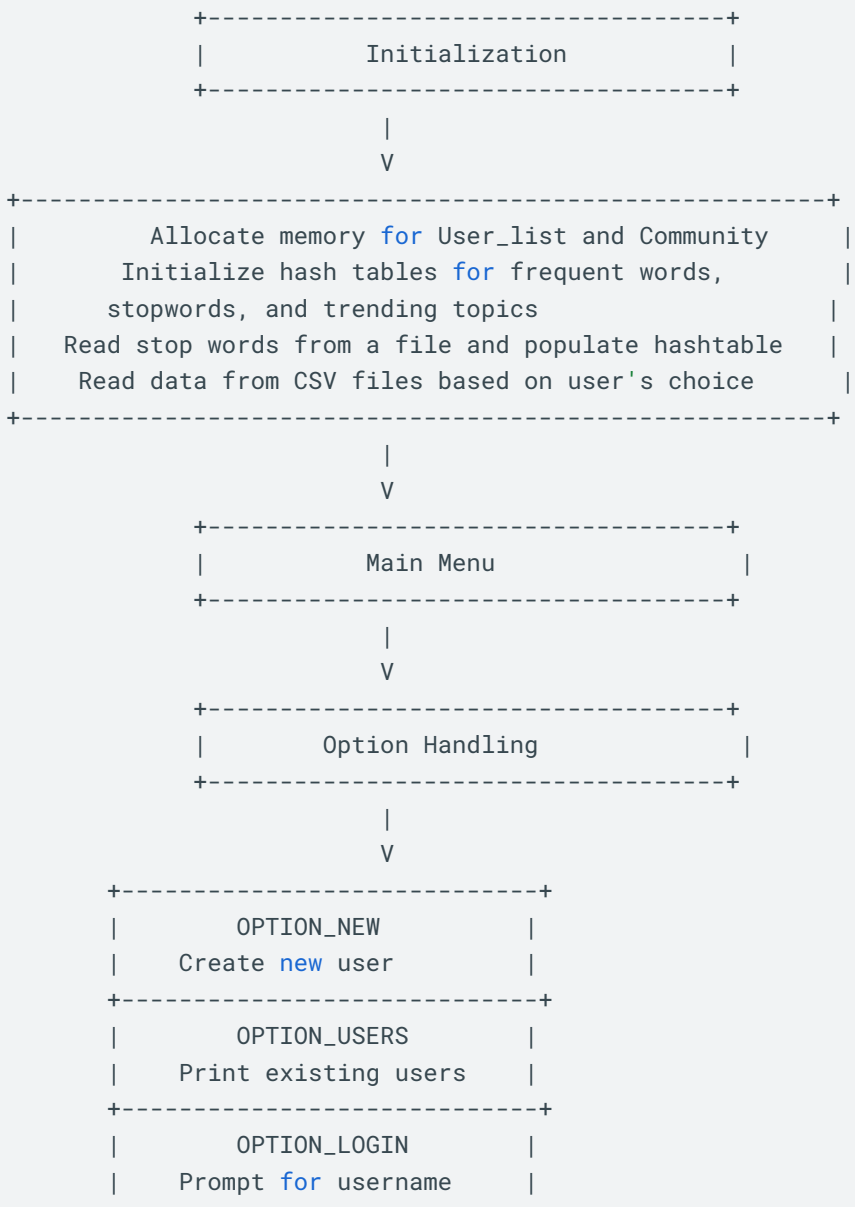
We describe the system architecture of our social network using the menu functions as a starting point, as we believe they are very descriptive.

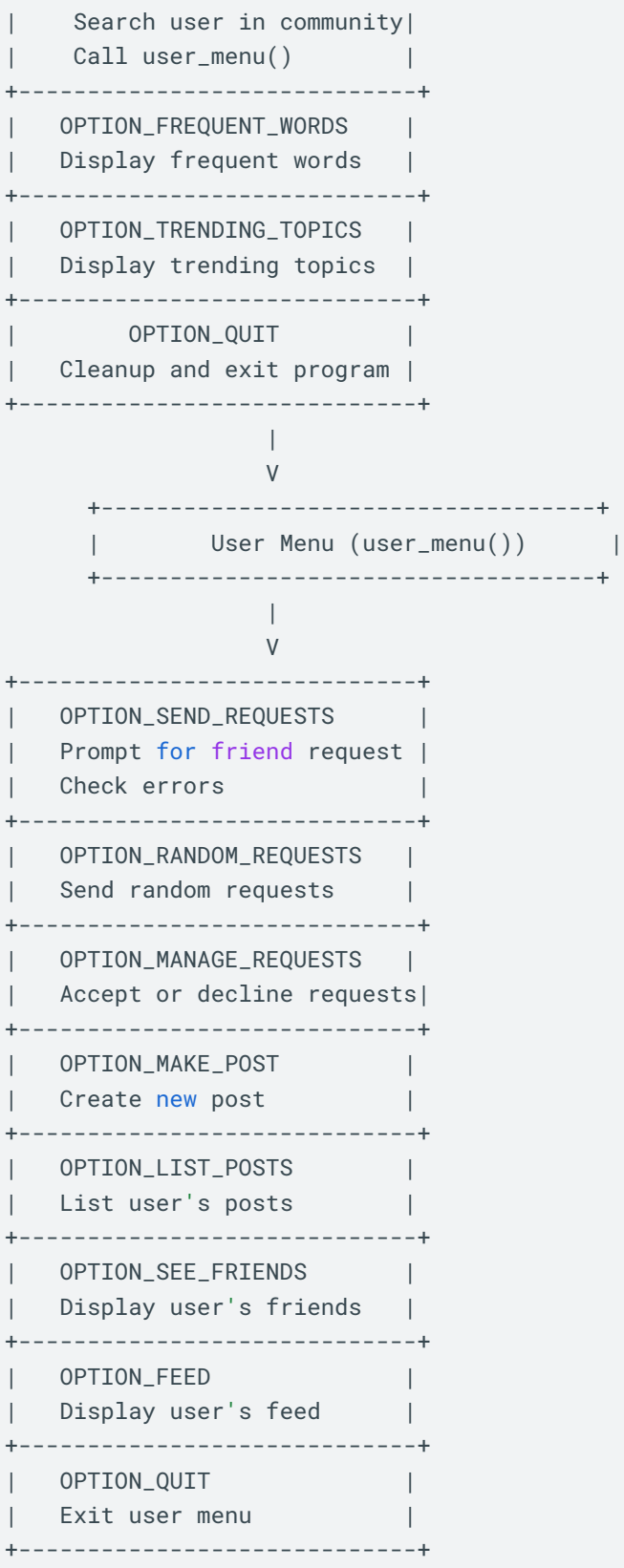
1. **Initialization**
- Allocate memory for the User_list and Community structures.
- Initialize hash tables for frequent words, stopwords, and trending topics.
- Read stop words from a text file and populate the stopwords hash table.
- Initialize the community by reading data from CSV files (memory_saving.csv, posts.csv, memory_saving_aux.csv) depending on the user's choice.
2. **Main Menu:**
- Display a menu of options to the user.
- Read the user's chosen option.
- If the option is to start from scratch, load data from memory_saving_aux.csv. Otherwise, load data from memory_saving.csv and posts.csv.
- Continue displaying the menu until the user chooses the "Quit" option.
3. **Option Handling**
- Based on the user's selected option, perform the following actions:
  OPTION_NEW: Create a new user and add them to the community.
  OPTION_USERS: Print the list of existing users in the community.
  OPTION_LOGIN: Prompt the user for their username, search for the user in the community, and if found, call the user_menu() function to display the user-specific menu.
  OPTION_FREQUENT_WORDS and *OPTION_TRENDING_TOPICS*: Display the top ten most frequent words or trending topics, respectively.
  OPTION_QUIT: Clean up resources, write data back to CSV files, and delete the community.
4. **User Menu (user_menu()):**
- Display a menu of options for the logged-in user.
- Read the user's chosen option.
- Perform actions based on the selected option:
  OPTION_SEND_REQUESTS: Prompt the user to enter the username of a person to whom they want to send a friend request. Check for errors such as sending requests to oneself or sending multiple requests to the same person.
  OPTION_RANDOM_REQUESTS: Send friend requests to three random users in the community.

OPTION_MANAGE_REQUESTS: Manage pending friend requests (accept or decline requests).
OPTION_MAKE_POST: Create a new post by providing the necessary details.
OPTION_LIST_POSTS: List all the posts made by the logged-in user.
OPTION_SEE_FRIENDS: Display the list of friends for the logged-in user.
OPTION_FEED: Display the user's feed, which includes posts from their friends.
OPTION_QUIT: Exit the user menu.

Using ChatGPT, a tool we used also in the 4th seminar of the subject, we have done the following diagram:

C/C++




Search user in community
Call user_menu()
OPTION_FREQUENT_WORDS
Display frequent words
OPTION_TRENDING_TOPICS
Display trending topics
OPTION_QUIT
Cleanup and exit program
User Menu (user_menu())
OPTION_SEND_REQUESTS
Prompt for friend request
Check errors
OPTION_RANDOM_REQUESTS
Send random requests
OPTION_MANAGE_REQUESTS
Accept or decline requests
OPTION_MAKE_POST
Create new post
OPTION_LIST_POSTS
List user's posts
OPTION_SEE_FRIENDS
Display user's friends
OPTION_FEED
Display user's feed
OPTION_QUIT
Exit user menu

Error Handling

We implemented error handling in various parts of our code. As we have a space limitation in the report, we will focus in the error handling implemented in the menu functions:

Error Handling in the show_menu() Function

The show_menu() function is responsible for displaying the main menu and handling user input. It is crucial to incorporate error handling to ensure that the program can handle unexpected inputs or errors gracefully. Here are some possible error scenarios and their corresponding error handling strategies:

Action	Possible cases	Handling
Invalid Option	When the user enters an invalid option, such as a non-numeric or out-of-range input.	After displaying the "Invalid option!" message, prompt the user to press enter to continue, and then clear the screen for the menu to be displayed again.
File loading errors	If there are issues with loading the required files, such as missing or inaccessible files.	Display an error message indicating the failure to load the files and provide instructions for resolving the issue. Terminate the program or offer the user the option to retry or quit.
Quitting the Program	When the user chooses to quit the program, it is essential to handle the cleanup and data saving operations properly.	Before quitting, delete the hash tables, dump data into respective CSV files, and delete the community structure. Display a success message if the cleanup and saving process completes successfully. If any errors occur during cleanup or saving, display an error message indicating the failure.

Error Handling in the user_menu() Function

The user_menu() function handles the menu options for a specific user. Similar to the previous function, error handling is crucial to handle unexpected inputs or errors gracefully. Here are some possible error scenarios and their corresponding error handling strategies:

Action	Possible cases	Handling
Sending Friend Requests	If the user tries to send a friend request to an invalid or non-existent username.	Check if the entered username is valid and not the user's own username. If the username is invalid or already a friend, display an appropriate error message and prompt the user to enter a different username.
Managing Pending Requests	If the user tries to manage pending requests, but there are no pending requests.	Check if there are any pending friend requests for the user. If there are no pending requests, display a message indicating that there are no pending requests to manage.
Making a New Post	If there are errors while creating a new post, such as exceeding character limits or memory allocation failures.	Check for errors during the process of creating a new post. If any errors occur, display an appropriate error message and prompt the user to try again or take corrective actions.
Viewing Friends or Feed	If the user does not have any friends yet or if the feed is empty.	Check if the user has any friends or if the feed is empty. If there are no friends or the feed is empty, display a message indicating the absence of friends or feed items.

Universitat Pompeu Fabra Barcelona

Data model design

C/C++

Dataset description and processing

Data handling is a crucial aspect of any social network, including Mung Bean. In Mung Bean, data is primarily stored and managed using CSV files. The code includes functions to read and write data from and to these CSV files.

On the one hand, we have the reading functions. The "read_csv" function is responsible for reading user data from a CSV file. It opens the file, reads each line, and extracts the necessary information such as username, birth year, email, location, and tastes. It then adds the user to the community using the "add_new_user" function. Additionally, it loads information about friendships and friend requests between users. The "read_posts_csv" function reads post data from a CSV file. It extracts information such as the username, datetime, and content of each post and adds it to the respective user's post list using the "add_post" function.

On the other hand, we have writing functions. The "write_csv" function writes the user data back to a CSV file. It iterates through the users in the community and formats their information into a single line. It appends the user's tastes, friends, and friend requests into a single string before writing it to the file. The "write_posts_csv" function writes the post data back to a CSV file. It iterates through the posts in the community and writes each post's username, datetime, and content into the file.

These functions ensure efficient handling of data in Mung Bean by reading and writing user and post information from and to CSV files. This approach allows for easy storage, retrieval, and modification of data, providing a seamless user experience within the social network.

REFERENCES

Quicksort useful information
https://www.geeksforgeeks.org/quick-sort/

Hash table implementation
https://github.com/maw101/C-Hash-Table

Tool used to convert text into ASCII Art titles
https://patorjk.com/software/taag/#p=display&f=Graffiti&t=Type%20Something%20

Source of stopwords dataset used:
https://www.perseus.tufts.edu/hopper/stopwords

Dataset creation and miscellaneous assistance:
https://openai.com/blog/chatgpt