Dating App: Interaction-Based Matching (MBTI + Likes/Friendlikes)

Group members: Aysegul Kula, Taha, Veronica

This project implements a dating-and-friendship matching system. Users are stored in a custom hash table keyed by email. The system loads user profiles and interaction history from CSV files, updates each user's learned preferences based on who they like, detects mutual matches, and can suggest new matches using a simple statistical compatibility rule. To improve usability, we also support popularity tracking, autocomplete aided by name, mbti and popularity considerations.

Public GitHub repo: https://github.com/taha6767/Cs62FinalTahaAyseVeronica

Course deliverable: Part II.C project write-up

Date: December 12, 2025

Needfinding Results

Our initial idea was to make an all encompassing search tool to be able to check if your feelings go both ways but we have decided having everyone on campus publicly searchable would be against privacy rules and we should only list people who have opted in and created an account. We have interviewed many people and one thing that was common in almost every interview was having the option to see if the person you want to be friends with or like is also interested in you would increase the amount of connections people would make and increase their likely hood of using the app. The problem best not solved by software is making two people actually want each other.

Dataset Format

To put shortly we have generated a csv file with 1000 entries called users.csv (this has email name gender etc.) and the related relationshipsNew.csv file (this one has who likes who data)

We use CSV files to store our data because they are simple to read and write. The format organizes data into rows and columns which makes it easy to parse using standard Java tools. We generated the data ourselves to ensure it worked well for testing. We used a list of first names and a separate list of last names. We randomly matched these names together to create unique users. We used random chance to determine the gender for each person. We also made sure to include every case so that men and women and non binary people are all represented in the final list.

Datastructures used and reasoning

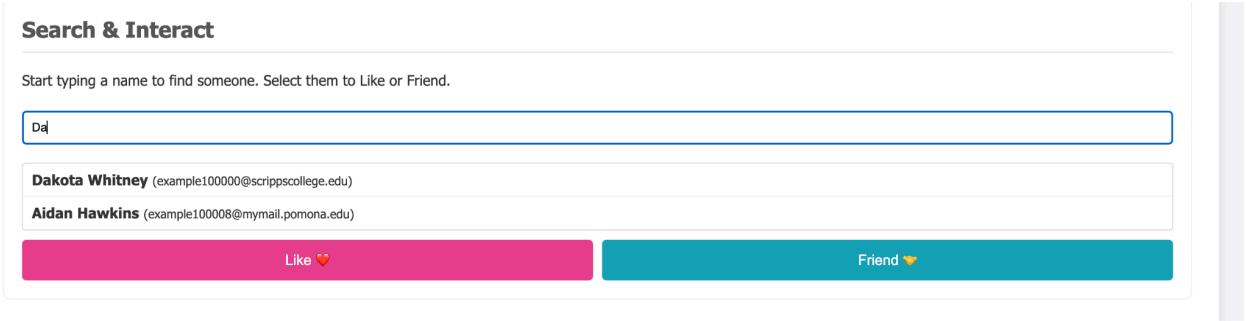
Our project uses a hash table as the core data structure to store users, indexed by unique email addresses. This allows fast insertion, lookup, and deletion, which are essential for frequent operations such as login, liking users, and retrieving profiles. Relationships like likes and friendships are stored as adjacency lists inside each user object, effectively modeling the system as a directed social graph. This approach is memory-efficient and makes it easy to check for mutual interactions.

For autocomplete and search, we iterate over active users in the hash table and apply token-based name matching, then sort results by popularity or compatibility. This design was chosen over more complex structures (e.g., tries) to keep the implementation simple, readable, and sufficient for the scale of our dataset while still meeting performance needs.

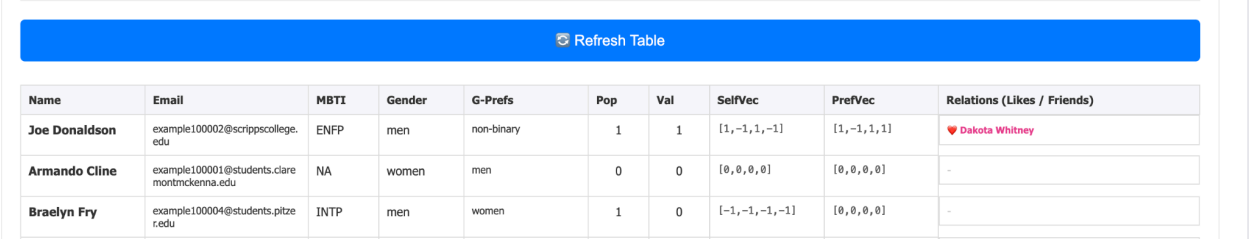
Results

A “results” section showing screenshots of your code execution (e.g., print outs in main()) that demonstrate each and every one of your features

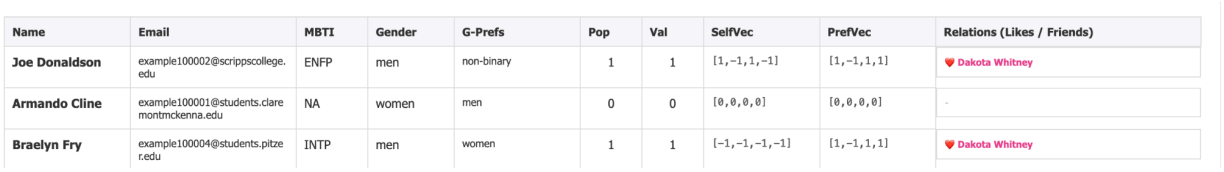
1) Autocomplete: as I wrote “Da” the autocomplete has shown me 2 possible matches so that I don’t need to write the whole name I can just click to choose.



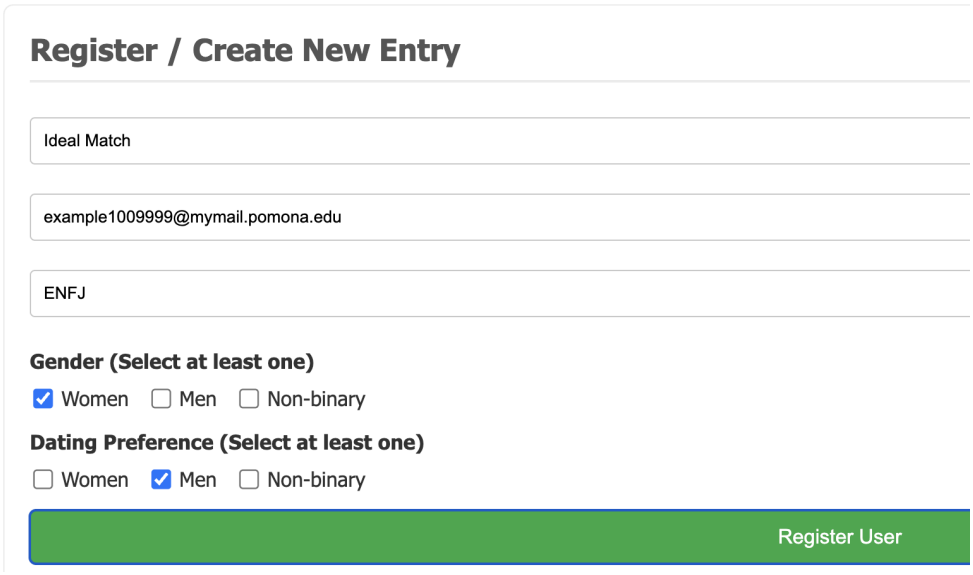
2) Like or friend option. After I click Like the related fields in the hash table get updated as exemplified below. (I am logged in as Braelyn Fry
At first my Relations vector is empty but after I press like on Dakota Whitney



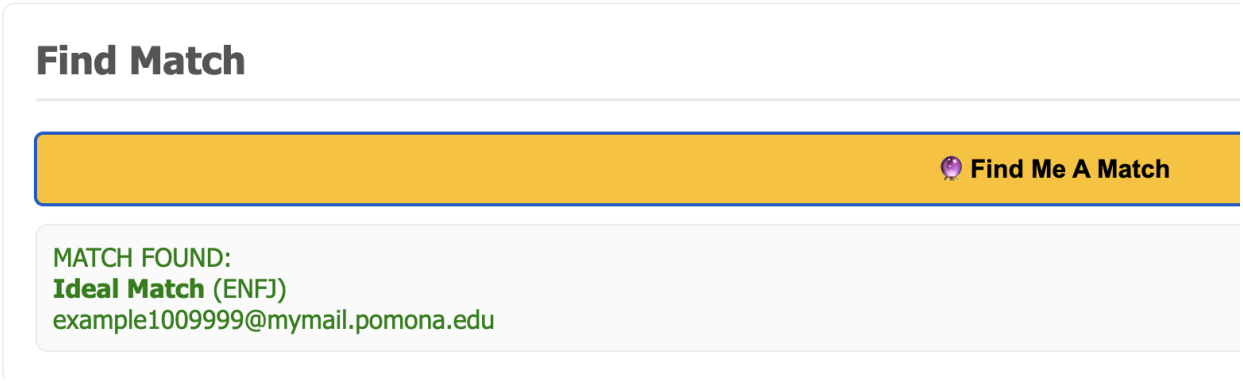
The hash table gets updated. As Dakot Whitney’s personality type is ENFJ my Preference Vector gets updated to reflect ENFJ in an int array format which is [1,-1,1,1]



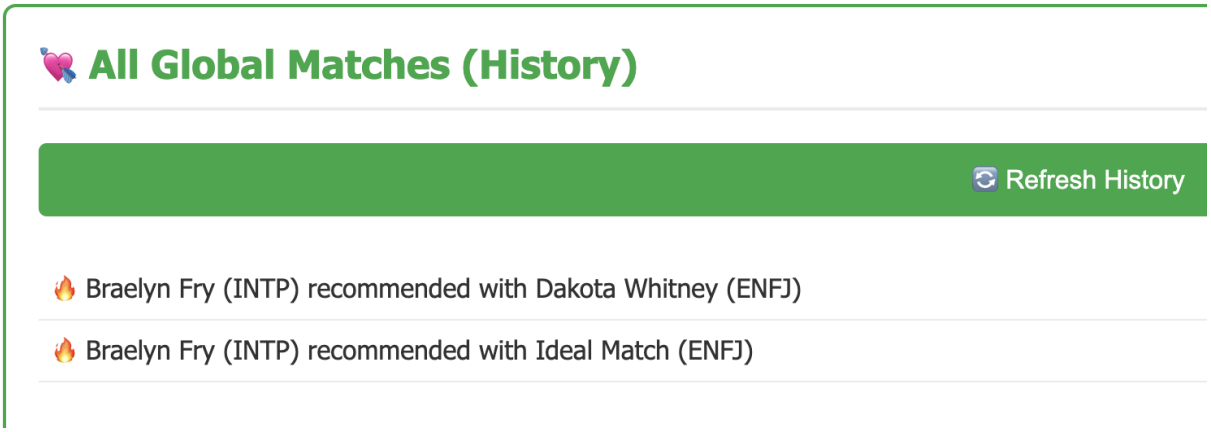
3) Match feature. First we create a user that is compatible with us. Since our gender is chosen as Men and we are interested in Women the new match has to be a Women who is interested in Men and their personality type is [1,-1,1,1] or ENFJ for short.



Now we find the match:



And all matches are not down at the bottom so both parties can be sent a message letting them know of the mutual connection:



Analysis

1. Mutual-Like Friend Matching (Interactions)

This feature handles users sending "romantic likes" or "friend likes" to others and detecting if a mutual match occurs.

Pre-conditions:

- Both the "source" (liker) and "target" (likee) users must exist in the hash table.
- The source user must not interact with themselves.
- Valid relationship types ("like" or "friend") must be provided.

Edge Cases:

- Self-Interaction: A user tries to like themselves. The WebController explicitly blocks this with an error message.
- Duplicate Likes: If a user likes the same person multiple times, the ArrayList in People will store duplicate entries unless manually checked, potentially skewing MBTI statistics. But this is intended as one person might not be able to express their love with just one like or want to skew their MBTI statistics in hopes of finding someone like “the one” they like.
- One-sided Match: A user likes someone who has not liked them back. No match is created, and the target is simply added to the likedEmails list.

Time Complexity: O(L) (Average Case)

- To determine if a match occurs, the algorithm scans the target user's list of likes (likedEmails) to see if they have already liked the source user back.
- If L is the number of people the target user has liked, the scan takes O(L) time.
- Updating MBTI stats and adding the email to the list are O(1) operations.

Space Complexity: O(1) (Auxiliary)

- The operation modifies existing lists in memory and does not require significant extra storage relative to the input size.

2. Recommendation Algorithm (Find Match)

This feature iterates through the database to find a user who is compatible based on MBTI statistics (the 0.33 proportional preference rule) and mutual gender preferences.

Pre-conditions:

- No preconditions if no match is found we return no match found.

Edge Cases:

- Empty Preferences: If a user has never liked anyone (validLikes == 0), the compatibility check returns true (neutral), effectively matching them with anyone who satisfies gender requirements.
- "NA" MBTI: If a candidate has "NA" (empty) MBTI data, they will be treated as having all traits at 0. They will fail to match with anyone who has a "Strong Preference" (requires > 0.33 score example chose E 2 times I one time so they have a strong preference for E as ⅔ valid choices is 0.33).
- Full Scan with No Match: If no compatible user exists, the loop iterates through the entire hash table and returns null.

Time Complexity: O(N)

- The algorithm iterates through the hash table array of size M (where M approx 2N to maintain load factor). It visits every slot once in the worst case.
- Inside the loop, checks for gender and MBTI compatibility which are constant time O(1) operations.
- Therefore, the complexity is linear with respect to the number of users N.

Space Complexity: O(1)

- The algorithm uses a single pointer (candidate) and a few boolean variables to track compatibility during the search. It does not create new data structures.

3. Autocomplete Search

This feature filters users by a name prefix and sorts the results based on their popularity (likedByCount).

Pre-conditions:

- The search prefix must not be null (returns empty list if null).
- Users must have names stored in the database.

Edge Cases:

- Case Sensitivity: The search is case-insensitive; the prefix and user names are both converted to lowercase before comparison.
- No Matches: If no names contain the token, an empty list is returned.

Time Complexity: O(N log N)

- The method iterates over all N active users in the table to find matches (O(N)).
- It then sorts the matching candidates. In the worst case (e.g., searching for "a"), all N users match. Sorting takes O(N log N) using Java's built-in sort.

Space Complexity: O(N)

- The method creates a new ArrayList (candidates) to store the matching users. In the worst case, this list holds all N users.

Affordence Analysis

This software mainly benefits users who are comfortable with structured matching systems such as personality-based recommendations and algorithmic ranking. It provides a clear and consistent way to discover and interact with other users. However, users who do not fit well into predefined categories (e.g., MBTI types or gender labels) may feel less represented by the system.

The project reflects common norms in modern dating platforms by quantifying social interactions and preferences, but it also challenges more opaque matching systems by making its logic explicit. Potential misuse includes gaming popularity rankings or targeting specific users through search and autocomplete features. More broadly, this project highlights how technical design choices can influence social behavior by shaping what interactions are easy or visible.

Reflection

How long did this project take the group, overall? Did it align with your expectations?
How was this process? Did you stick with your original idea and proposed data structures, or did you make changes along the way? If you made changes, why?
Any feedback on the final project assignment itself

The project took us quite a long time. It took us a lot more iterations than we have expected. Each time we thought we were done, a new problem came up. It was a bit frustrating but rewarding at the end.

We did not completely stick to the data structures we proposed. Our main data structure was a hashmap as planned, but to model the relationships, we changed plans. In our minds, graphs made sense to show likes/matches. But in reality, we realized that it is a lot slower and harder to use graphs, as finding nodes can get quite hard. Instead, we used arraylist embedded the hashmap to model liksed and matches.
As feedback on the final project, this is more on our planning then the structure itself. But i think if the checkpoints required us to have more of the actual content done, we would do better with time management. But that is on us. Overall this project was fun and rewarding.