CSC111 Project 2:
"Destiny" - Find Your Next Soulmate Here

Hsin Kuang (Joseph) Cheung
Tsz Kan Charlotte Wong
Yan Lam
Chun Yin Liang

March 31, 2025

1 Introduction

The University of Toronto is a community with a diverse student population, with lots of international students (University of Toronto, 2024). As we enter early adulthood, the struggle between isolation and intimacy becomes significant. Many students feel lonely and stressed when making new connections and managing their academic burden, especially those from abroad who experience homesickness. **How can students in UofT meet new people and build a supportive circle beyond lectures with less effort? This is a challenge that is faced by most of our group members since we are new to the city.**

To address this problem, we wanted to introduce a new application called **Destiny**. It is designed to make socializing easier and more interesting, enriching students' university lives. This is not just another "dating app" but a platform where students can seek both friends and potential romantic partners. By offering a wide pool of potential matches, the application helps users connect effortlessly - whether you are looking for a study partner, a coffee companion, or even a date.

With a user-friendly interface, students can create profiles showcasing their interests, personality traits, and preferred connections. The algorithm helps suggest matches based on shared hobbies and values, making it easier to meet like-minded people while reducing the insecurity of approaching someone out of fear of rejection. To ensure more compatible connections, the application provides a personalized environment where students can select their preferred qualities.

By encouraging friendships and romantic bonds, we aim to create a welcoming platform where students can develop personally, and build meaningful, enduring relationships. Four of our group members are international students themselves, and thus, we understand firsthand the importance of a strong supportive network, either as friends or something more, in enhancing mental health and student engagement throughout university life.

This application is not yet fully completed, but rather more like a prototype, since we lack user base. With some modification to the program code to make it real-time, we can make the app a real application.

2 Dataset

Ideally, we should collect data from current UofT students to build a realistic database of actual users. However, due to time constraints and privacy concerns, we chose to generate a synthetic dataset of 200 users for this project. These users have different attributes and preferences, and we used the Faker Python library to generate these data. The data generation is done with `generate_users_with_class()` function in `user_network.py`, which creates a list of `User` objects. We also added the four of us into this list, so we have a total of 204 users in our dataset.

3 Computational Overview

In this project, we are using trees as a part of our user recommendation system, and graphs for the visualization of user connections in our app. Apart from `main.py`, this project contains five other Python programs: `user_network.py`, `graph.py`, `ui.py`, `tree.py`, and `common.py`. Below is a breakdown of the main components and responsibilities of each.

`user_network.py`

This file contains the core logic of the user system. It contains the `User` class and `Characteristic` class. Every instance of the `User` class is a user in the dating app. We will make use of the instance attributes of the `User` class in both the recommendation system and graph visualization, such as the name, age, dating goal, and current friends of the user. One of these instance attributes is `characteristics`, which is an instance of the `Characteristic` class. Every instance of this class represents the characteristics of a user that influence their connection preferences. We will make use of this class's attributes for each user when recommending matching users to them. Some instance attributes of the `Characteristic` class include MBTI, academic major, and primary spoken language of the user.

The `generate_users_with_class()` function simulates user generation. It takes in `list_size` and `seed` as the arguments. The `add_fixed_users()` function adds predefined users (e.g., project creators) to the user list. This function mutates the list passed in. The `add_user()` is an interactive function that prompts for user input (like a registration form) and appends a new `User` object to the user list. It serves as an alternative to using the graphical user interface (as the user uses the terminal to input information).

Tree-Based Recommendation System — `tree.py` and `common.py`

These two files contains the tree methods and the recommendation system of the app. Since these methods and functions will be repeatedly called when our app operates, we used two files to store the classes and functions to avoid circular imports. We are implementing a Binary Tree to match users to generate a list of matches based on the users' preference for attributes. Overall, attributes are ranked as decision nodes according to user priorities (e.g., if a user values hobbies the most, hobbies will be the first level of the tree). Users who meet a given attribute are assigned a 1, while those who do not are assigned a 0. As a result, different users follow different branches, and a recommendation list is generated by prioritizing those who fulfill the highest number of a user's preferred attributes.

Before using the app, each user is asked to select a dating goal: meeting new friends, situationship, short-term relationship, or long-term relationship. Then, the user is asked to enter their answer for each attribute, and is stored in our dataset.

To build the preference tree, we use `data_wrangling()` function to convert users' information and the matching status of their attributes into a CSV file named `data.csv`. We first call the `add_priority()` function to allow users to order their characteristics by preference, storing the result in a list called `heading`. Then, we filter the dataset to create a `potential_users` list, containing only those users whose dating goals match that of the current user. In order to build the CSV file, we initialize a dictionary with keys corresponding to the column headers in the file. By default, the dictionary contains a `name` key, to which we add the names of the users from `potential_users`. The other keys in the dictionary correspond to each attribute in `heading`, and their values are either 1 (indicating a match) or 0 (indicating no match), based on comparing the attribute value of each potential user with those of the current user. Lastly, a pandas `DataFrame` is created from the dictionary and exported to the CSV file.

The CSV file is then processed by the `build_preference_tree()` function, which generate a preference tree with a format:

```
[name, ethnicity, interests, mbti, communication_type,
political_interests, religion, major, year, language,
likes_pets, likes_outdoor_activities, enjoys_watching_movies]
```

with indicators 0 and 1 (Note: the order of the attributes changes based on the `data_wrangling()` function).

As the CSV function processes each row, `match` is created as a list containing the preference indicators, with the user's name appended at the end. This list is then passed as an argument to `insert_sequence`, which constructs

the tree accordingly.

The `insert_sequence()` function inserts a sequence of items into a binary tree, where the first item, either 0 or 1, determines whether to recurse into the left or right subtree. Each subsequent item in the sequence continues branching deeper into the tree. The final item, which represents the user's name, is stored in a list at the corresponding leaf node.

The preference tree is then run by the `run_preference_tree()` function. It is a recursive function that returns a list of ranked recommended users by traversing the binary tree in the preference-based manner, which prioritizes the left subtree first (closest match) before processing the right subtree (lower match). The base case occurs when `self._root` is a list, which stores user names (indicating a leaf node). The function is then able to extend the `recommendation_list` with the user names, returns, and stops recursion. The recursive step happens when the current node is not a leaf. The function extends the `recommendation_list` by recursively traversing the left subtree first (higher preference) then the right subtree (lower preference).

Graph-Based Visualization — `graph.py`

After the users are matched through our app, we use network graphs to visualize the connections between all users. All graph generations and updates are handled in graph.py, with the help of NetworkX, Plotly, and Dash Python libraries. NetworkX is used for generating the graph structure, such as nodes and edges. Plotly is used to render visual elements of the graph, such as node colors and sizes. Dash is used to implement the interactive web interface for the graphs.

There are two graphs, one displaying social connections (social graph) and the other displaying romantic connections (dating graph). In these graphs, each node represents one user, and each edge represents a connection between two users. In the social graph, the size of each node is directly proportional to its degree, that is, a user with more friends will appear as a larger node in the graph. The nodes also have different colors according to their degree, where a lighter color represents that the user has more friends. In the dating graph, all nodes have the same size and color, since the number of romantic partners is limited to one. Now we will go through how we achieve this.

These two graphs are created using the function `create_app()`, which returns a Dash application. Three lists of `User` objects are passed into the function, which are `user_list`, `user_looking_for_friends`, and `user_looking_for_love`. `user_list` stores all the users in the app, and `user_looking_for_friends` stores all the users looking for social connections, while `user_looking_for_love` stores all the users looking for romantic connections. The last two lists are sublists of `user_list`.

To create the nodes and edges of the graphs, `user_looking_for_friends` is passed into the function `plot_social_connections()`, and similarly, `user_looking_for_love` is passed into `plot_romantic_connections()`. These two functions utilize the NetworkX library and construct the underlying graph structures for our two graphs. We do not have to implement a class representing the Graph Abstract Data Type ourselves, since NetworkX already has all of the Graph attributes and methods built in, such as `add_node()` and `add_edge()` functions. So, `plot_social_connections()` simply goes through all the `User` objects in the input list, and creates user nodes by calling `add_node()` on each user. In `plot_social_connections()`, we make use of each user's `social_current` and `social_degree`, which are obtained from the input list of `User` objects. NetworkX also allows us to specify the size of the node easily, so we can modify each node's size according to the user's number of friends. We create edges between user nodes using the NetworkX function `add_edge()`, after referring to each user's `social_current` (a list of `User` objects which are friends of that user). The dynamic coloring of user nodes is done using Plotly (specifically, the class `plotly.graph_objects.Scatter`). Similarly, `plot_romantic_connections()` creates nodes and edges according to its input `user_looking_for_love`, but with a constant node size and color. We also utilized NetworkX's `spring_layout()` function to position user nodes naturally on the graph based on their connections.

Additionally, we implemented two interactive features for the graphs, allowing developers to easily retrieve information about the app's users. First, the developer can search for a specific user on the graph simply by typing their full name in the input field at the top of the graph. Then, the graph will automatically zoom in and "focus" on the searched user, and highlight the user node in red. The searched user's edges will also be highlighted in red, visualizing their connections on the graph. On top of the graph, the searched user's number of social connections and romantic partners will be shown. This way, the developer can have accurate information about that user, without manually

counting the number of edges attached to the user node, or referring to the color scale. Also, the same effect is achieved when the developer clicks on a user node on the graph. Afterwards, the developer can click "reset" to zoom out and view the entire graph. Both features are implemented using the `update_graphs()` callback function, which utilizes the Dash library, and checks for inputs in the web interface from the developer..

As we are not familiar with using these three Python libraries, some parts of the graph visualization were done with the assistance of Generative AI, such as the formatting of some displayed texts. Once we obtained a template of how to format different visual elements (e.g., which attributes to modify and some sample values), we could easily duplicate and modify them to achieve our desired layout and formatting of the graphs' web interface.

Graphical User Interface (GUI) — `ui.py`

`ui.py` generates the GUI for the app, and links user input to methods and functions from other files. The tkinter library is primarily used here, as it includes lots of tools and functions to create a GUI in Python. Widgets like Tk, Frame, Label, Entry, and Button are used to build the app's layout and handle user interactions. We also used the Python Imaging Library (PIL) module to load our home page's image into the GUI.

The GUI also provides a pathway for the developers of the app to access the graph visualization, since the graph should not be shown to normal users. We will provide details on accessing the graph in the next section.

Similar to `graph.py`, we used Generative AI for some parts of `ui.py` when formatting visual elements. Since this GUI is not a core part of the computations done in this project, but more of a quality of life feature for our users, we will not go into detail to discuss the GUI implementation.

4 Instructions for running the program

Sample demonstration (we will be creating two users, one shown on the social graph, and one on the dating graph):

- Run `main.py`. The GUI show be shown as a pop-up window.
- Enter any name, say, "test" to continue.
- Your screen should look like Figure 1 now.

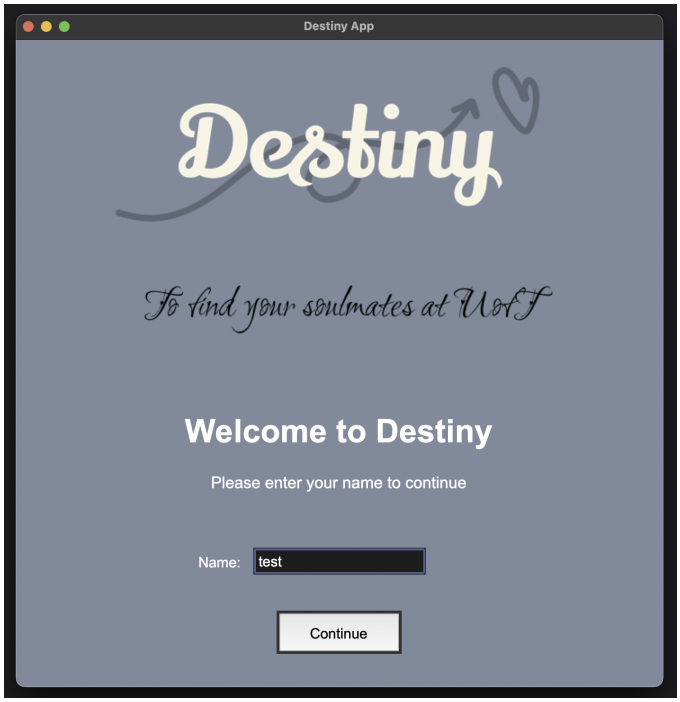

Figure 1

- Then, click "Continue" to proceed.
- Now, fill in all the attributes as you desire, except for the Dating Goal.
- Please choose "Meeting new friends" for the Dating Goal. This ensures the first user is added to our social graph. Rank the attributes you value the most at the bottom of the page by selecting an attribute and clicking the arrows.
- Your screen should mostly look like Figure 2, apart from any different attributes you chose.

Figure 2

- Now scroll down and click “Create Profile”. Then, click “Find Matches”.
- Here, you can either choose “Match” or “Pass”. We recommend matching with 3 users for testing purposes.
- After you have finished matching, press “Exit”. Now you will be re-directed back to the main menu. We have added the first user into the app.
- Now, you can create another profile for another user, say, “test2”.
- Your screen should look like Figure 3.

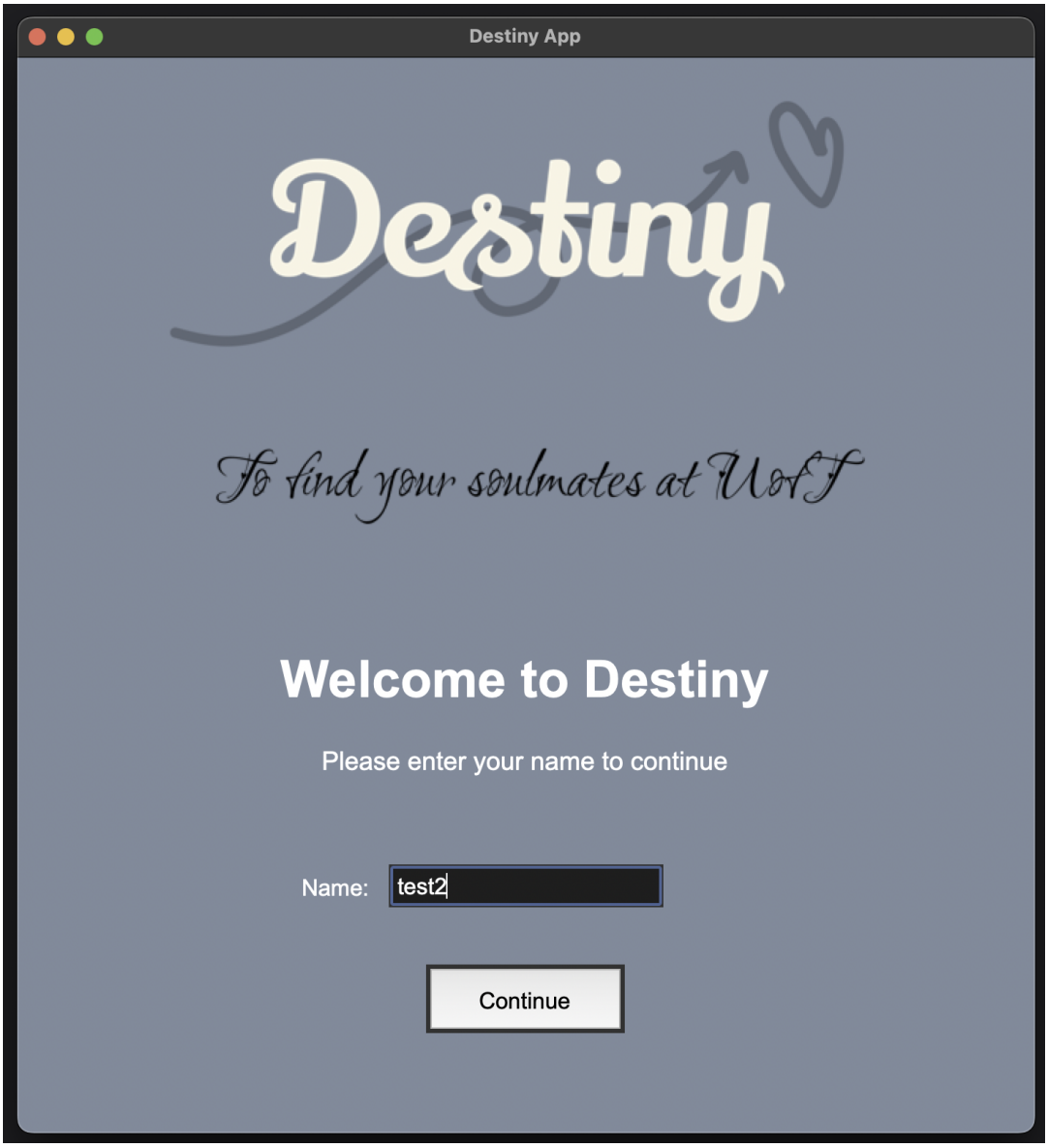

Figure 3

- Click "Continue" to proceed.
- Similarly, fill in all the attributes as you desire, except for the Dating Goal.
- Please choose any of the three options other than "Meeting new friends" for the Dating Goal. For example, "Long-term relationship". This ensures this user is added to our dating graph. Rank the attributes you value the most at the bottom of the page by selecting an attribute and clicking the arrows.
- Your screen should mostly look like Figure 4, apart from any different attributes you chose.

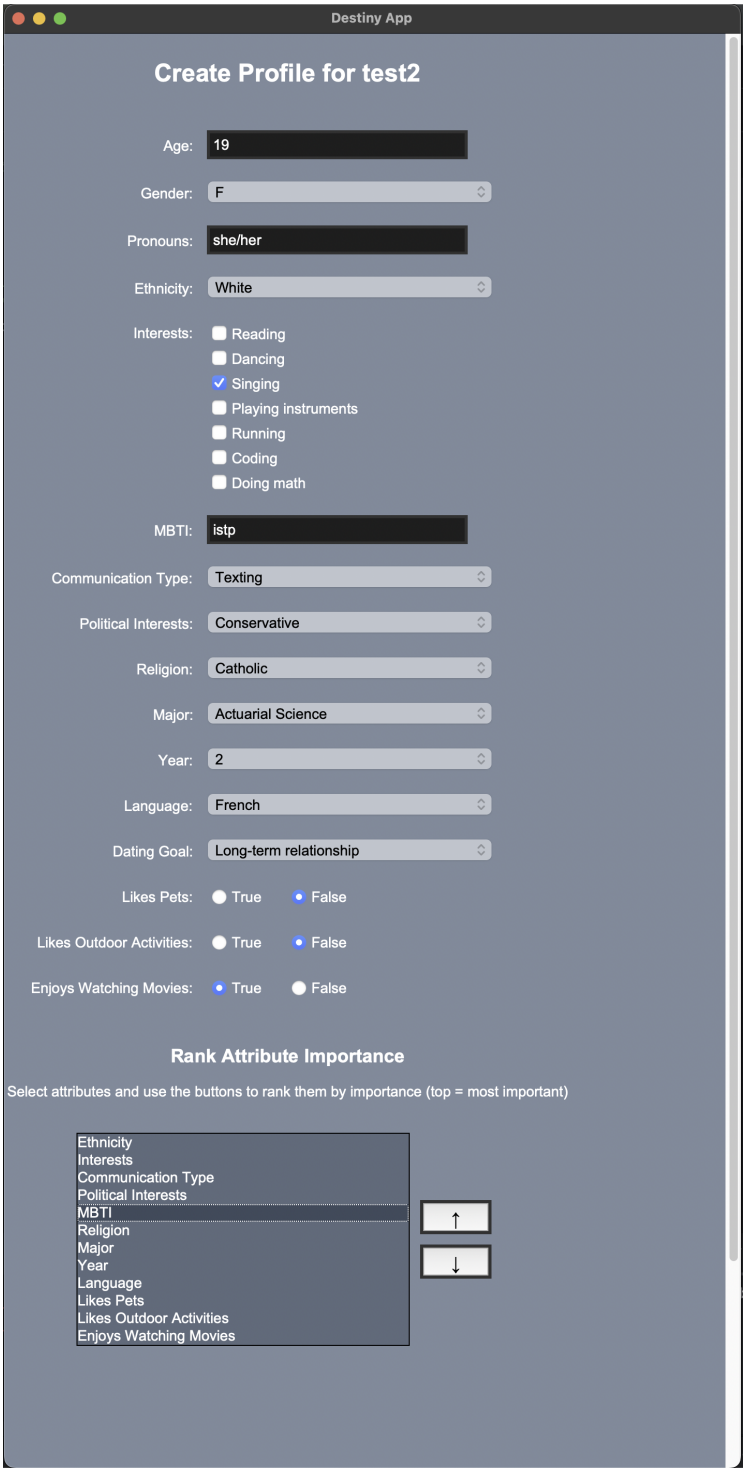

Figure 4

- Now scroll down and click “Create Profile”.
- Instead of matching friends, you will be brought to a page that helps you match with someone who is also looking for a relationship.
- Now, click “Match” until you successfully match with a romantic partner.
- Then, you can press “Return to Home”.
- Now, input “admin” as the name, as shown in Figure 5.

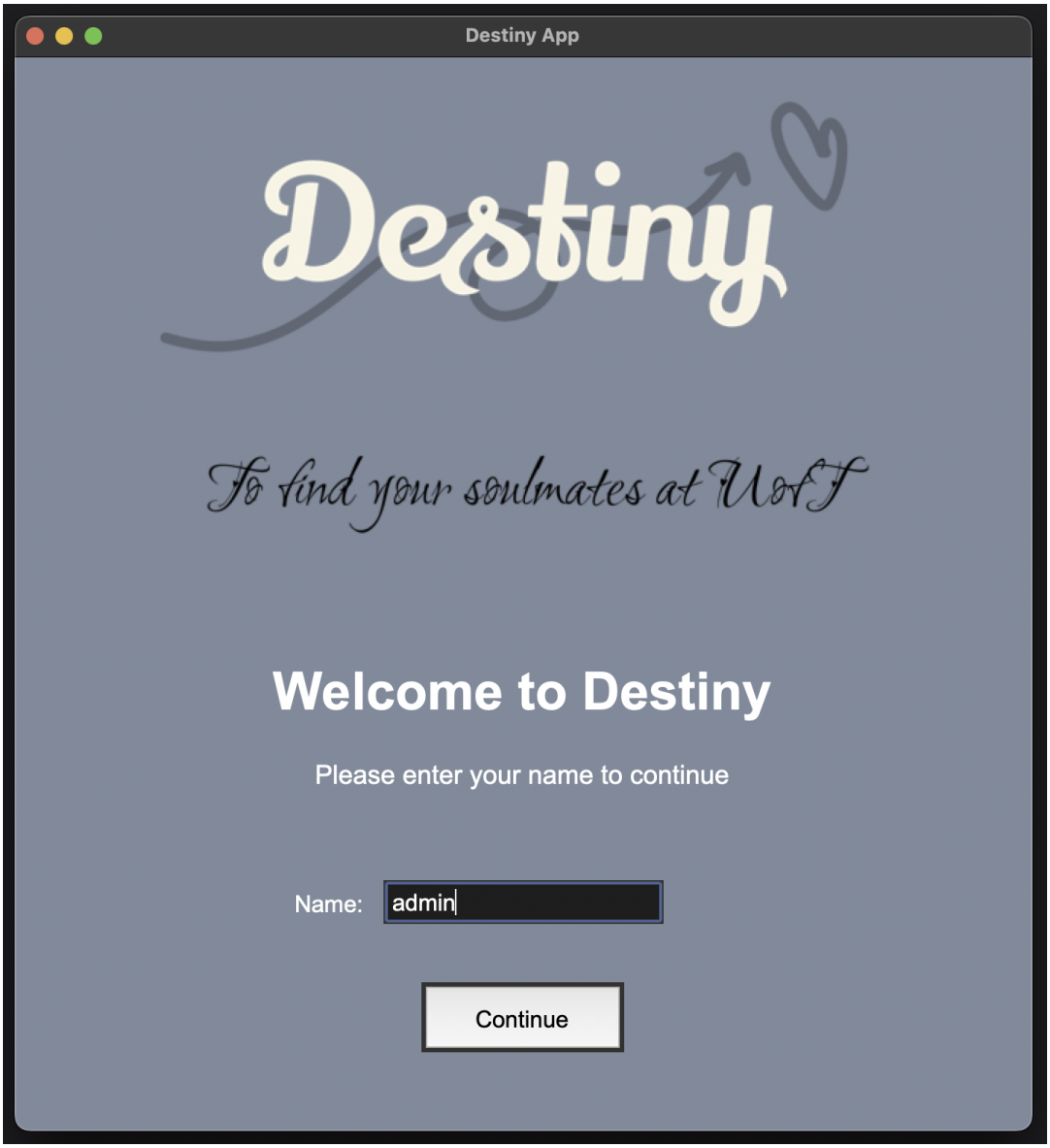

Figure 5

- Click “Continue” to proceed.
- You should be logged into the admin console now.
- For debugging purposes, you can “Print User List to Console”.
- Now, press “View Network Graph” to view the visualization of the connections of the users in the application.
- A local web application would pop up in your default browser.
- Now, your screen should look similar to Figure 6.

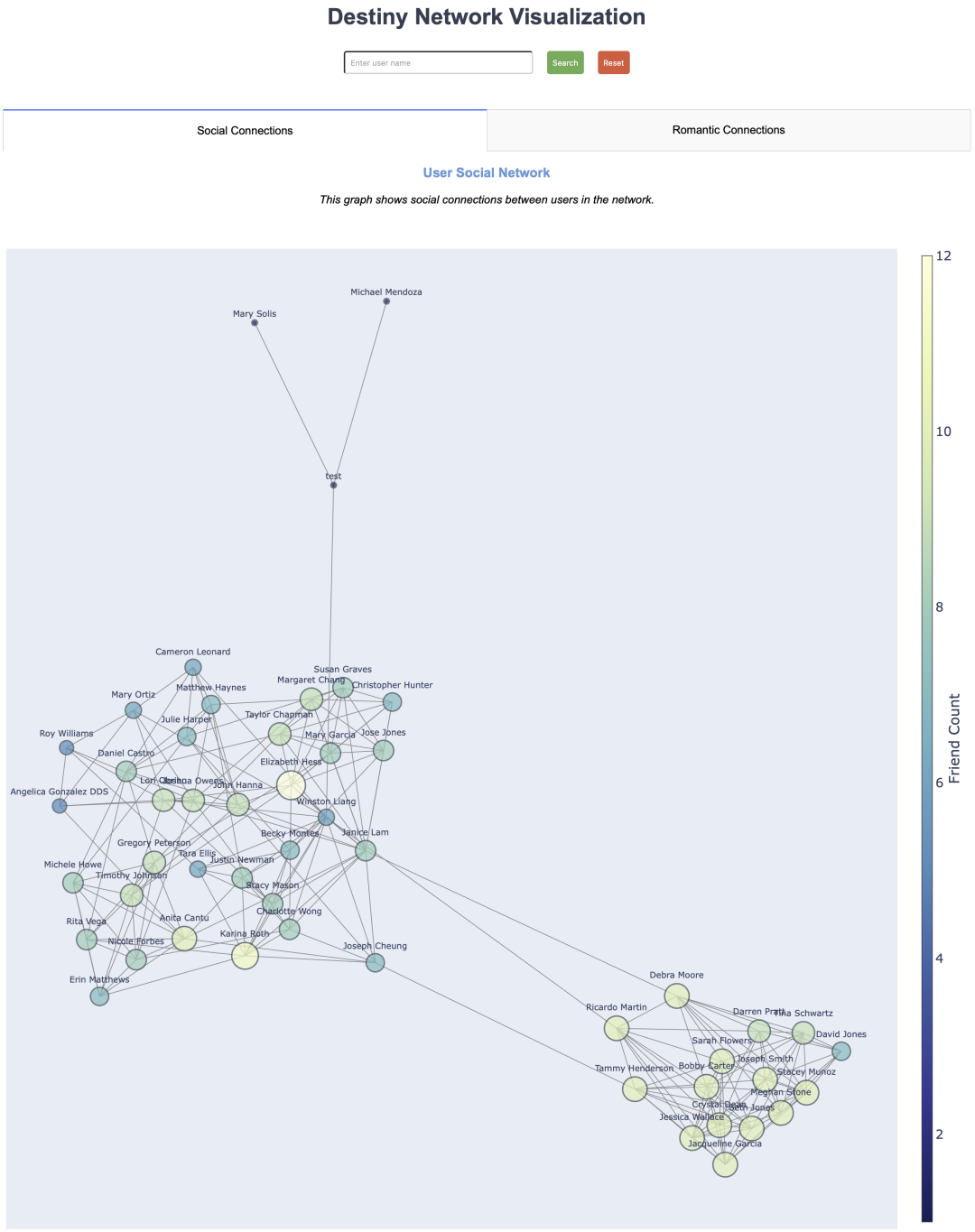

Figure 6

- There are two connection maps: Social connections (friends) and Romantic connections (relationships).
- You can navigate the maps freely to see the connections between users.
- Now, click on the "Romantic Connections" tab to view the dating graph.
- Your screen should look similar to Figure 7 now.

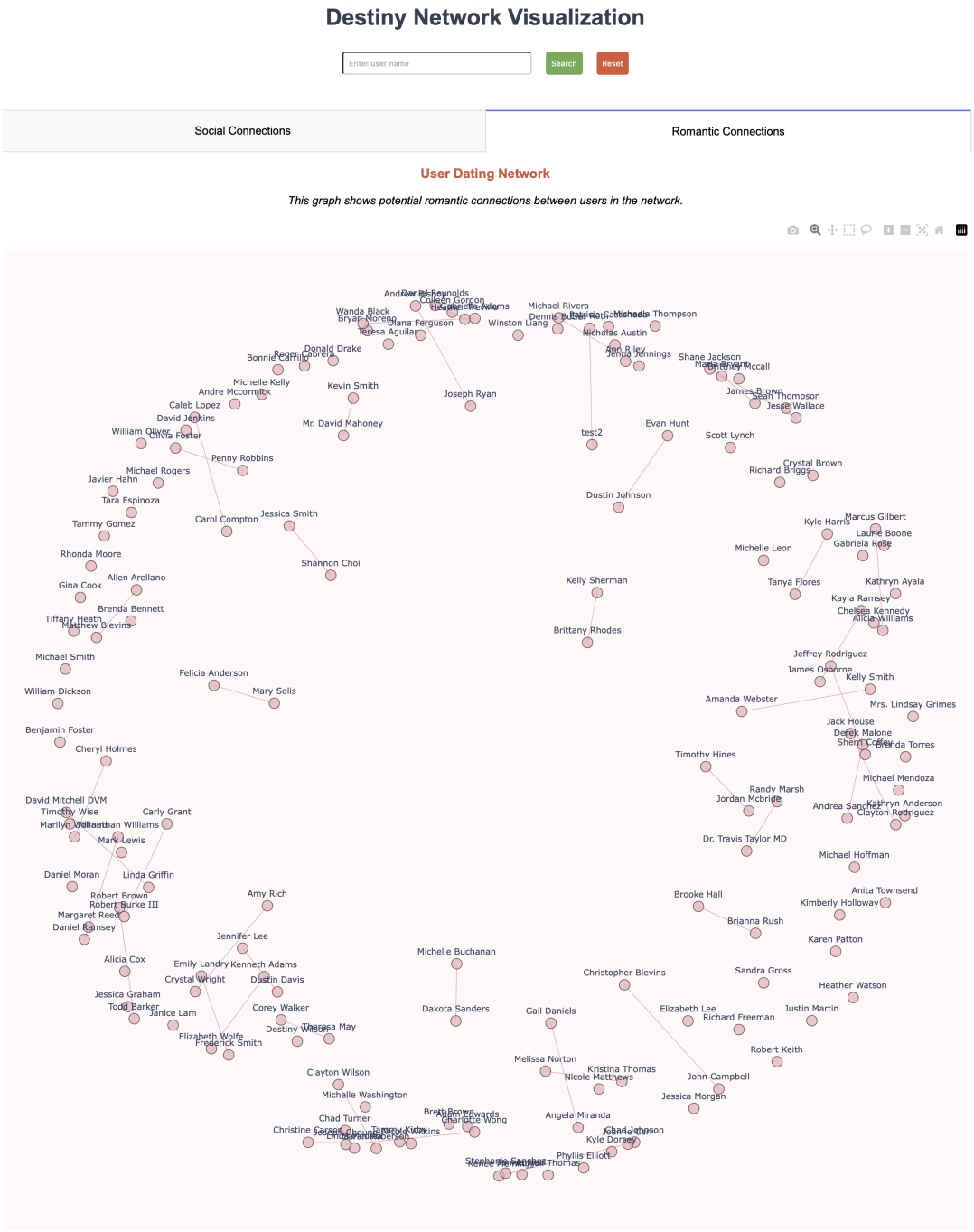

Figure 7

- Now, click on the "Social Connections" tab to return to the social graph.
- Try out the searching function by inputting "test" in the search box.
- The graph will zoom into the neighborhood that "test" is in, highlighting the user node and its connections in red.
- You can also view their number of connections on the top of the graph.
- Your screen should look similar to Figure 8 now.

Figure 8

- Now, click on the "Romantic Connections" tab again to return to the dating graph.
- Try out the searching function by inputting "test2" in the search box.
- The graph will zoom into the neighborhood that "test2" is in, highlighting the user node and its partner in red.
- You can also view their number of connections on the top of the graph.
- Your screen should look similar to Figure 9 now.

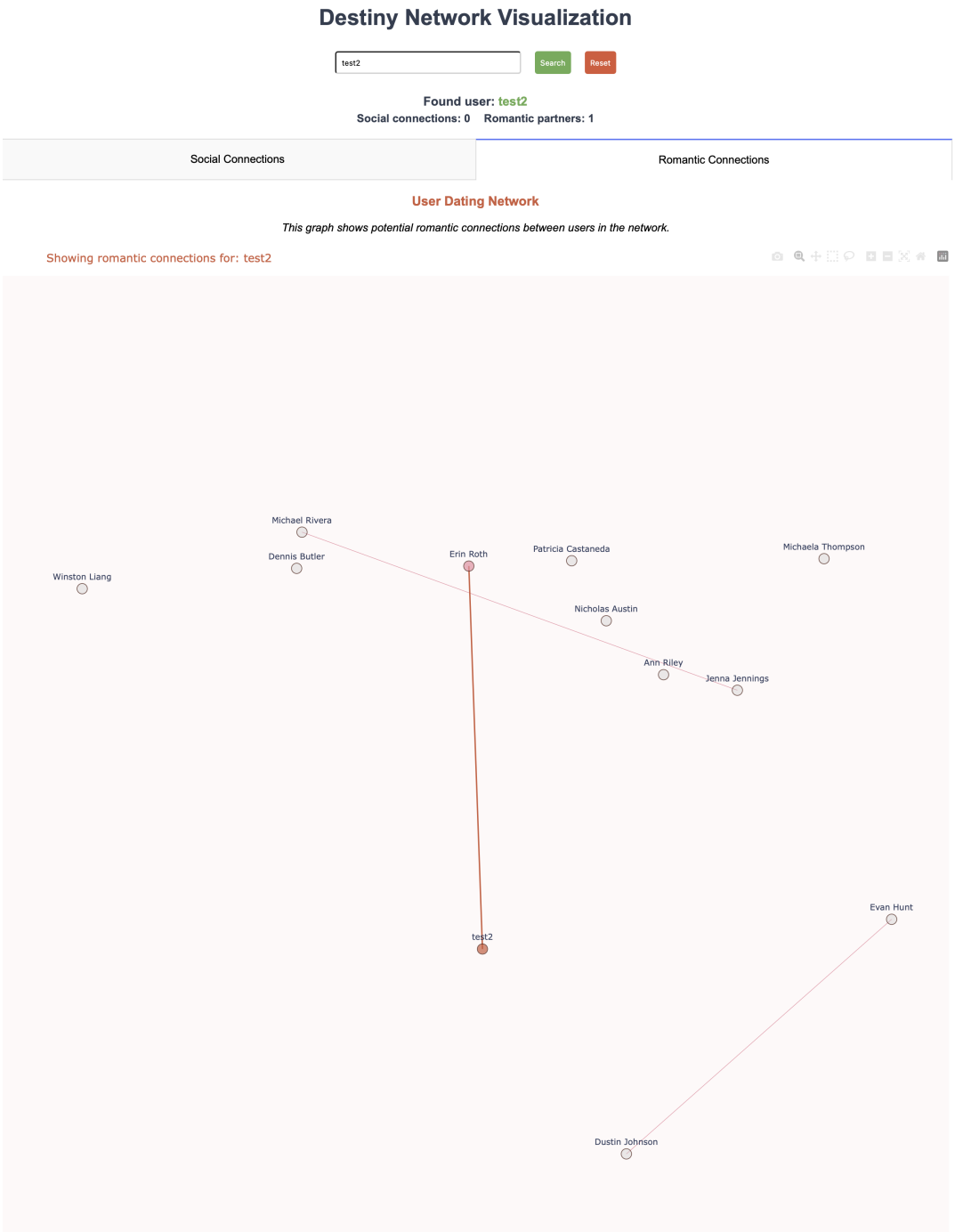

Figure 9

Great! Our newly added users are reflected correctly on the interactive graphs.

5 Changes to the project plan

Initially, we implemented the tree using the basic <self._root> and <self._subtree> structure to align with the format of Exercise 2's animal tree. However, we found that using a binary tree with self._root, self._left, and self._right was more suitable for our recommendation system. This approach is particularly effective because all the 1s in the tree, representing users most relevant to the target user, are positioned along the leftmost branch. By using a binary tree, we can simplify the process of generating a recommendation list by recursively traversing only the leftmost branch.

6 Discussion

Does the product meet our original goal? At this stage, this application does not really help UofT Students to match their soulmates, since the app right now has no user base and is more like a prototype. This is mainly due to time and ability constraints – we do not have the ability to build a real working app with an interactive UI, since it is out-of-the-scope for our current CSC110/111 curriculum. Yet, we wanted to make the application as real as possible. That is why we did seek help from AI for helping us on unfamilar libraries by generating code involving visual elements (`ui.py`) and (`graph.py`). This is mainly because we do not want to spend most of the time exploring how to use new libraries to format visual elements, but instead we want to focus on building our own matching algorithms, graph data structures and data generating functions, which is closer to the aim of our course project. Although the application is not really a published, usable app, we have built a project that can easily be transformed into a real-time working application. The prototype itself demonstrates the potential of the app and how it leverages tree/graph knowledge we learnt in class for creating this matching algorithm based on user features. The tree structure sets priorities at different levels, enabling the creation of a customized tree and a corresponding recommendation list. The graph effectively visualizes connections between users through edges, allowing us, as editors, to review potential relationships. This successfully fulfills our goal of developing a connection-based application.

What are some limitations?

There are quite a few limitations we encountered when making our project. The main limitation is the deployment of the project as a real application. The project is not a real-time application due to various reasons like the lack of a server host, user base, and experience in making applications. Apart from that, originally we planned to implement a function that records the past romantic relationships and friendships of all the users. However, it would also imply that we have to generate users with broken friendships and relationships, which can be a bit complicated. We also discussed that this function is not that important since the app shouldn't really be recording past relationships of people, as we mainly just want to help users to know more people. In some sense, recording a person's ex feels like violating their own privacy rights, and such data isn't really needed. Another limitation would be our use of AI. As mentioned above, we did seek the help of AI to generate code for out-of-scope libraries and rarely used functions due to time constraints. Data privacy would also be a great concern. Currently, we are not saving any user data into a file when the program is terminated. However, in a real-world setting, the data should be encrypted safely in a server, of course not in our local machine. Some functions might not be used directly for the UI application, but is rather used in the console, since we did not plan to create a user interface back then during our coding process. For example, `add_user()` is actually never implemented in the UI application. Another version of it exists as `add_user_to_network` in `ui.py`. However, we kept the function since working with the console is essential for creating test cases for new functions and debugging.

7 References

University of Toronto. (2024). *Quick facts.* https://www.utoronto.ca/about-u-of-t/quick-facts