Design Doc: Capstone - Predictive Search Feature

Author: Lucy Qu <lucyqu@google.com>
Reviewed by: Nico Rodriguez, Andrew Sweet

Last updated: July 30, 2020

Project Information

- Shopping Property Team
- People Involved:
    - Lucy Qu, lucyqu@google.com: Intern, feature owner
    - Himani Yadav, himaniyadav@google.com: Co-Intern, capstone project
    - Anika Bagga, anikabagga@google.com: Co-Intern, capstone project
    - Nico Rodriguez, nicorod@google.com: Host
    - Andrew Sweet, asweet@google.com: Co-Host

Objective

Create a predictive search feature for when the user searches for the name of the person to add to the group. The suggested names will be ranked in order of match with search string. Upon hitting enter, the list of users will be displayed, ranked in order of relevancy with the current user.

Background

In the context of the ways in which COVID-19 has impacted interpersonal relationships, this capstone project seeks to connect people through a social media platform via the idea of completing challenges with your closest social network. People form groups with their friends and complete challenges, for which they can create posts and write comments on each other's posts. See Figma prototype.

As part of the group page, the predictive search bar will help to enhance the user experience by allowing users to search for other users to add to the group. The search bar will make use of autocompletion, autocorrect, whitespace correction, and a ranking algorithm based on connection to the current user.

Overview

- User searches for name in search bar

- Suggested names will be listed based on auto-completion and auto-correction of the user input, ranked in order of best match with search string
- Returned users will be ranked in order of relevancy. Factors that determine this include whether they are already in a group with the current user, the number of shared groups between the current user and the target user, and whether they are in a group with someone else who is in a group with the current user and so on.
- Add user to group upon pressing add icon

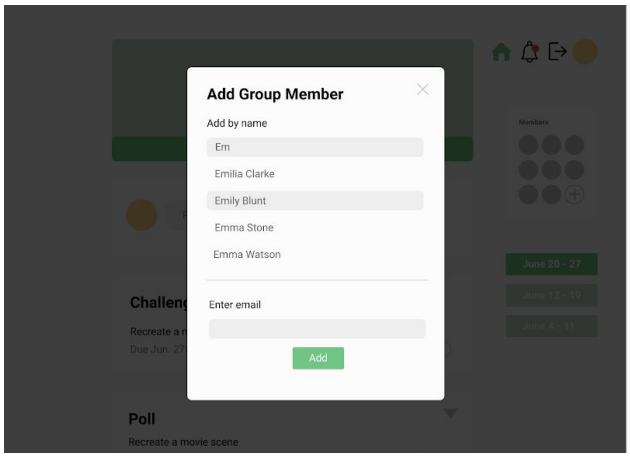
Infrastructure and Production overview

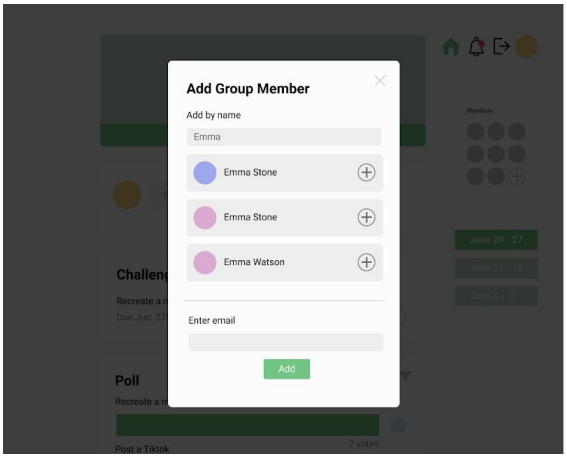
- Java Servlets
- Google App Engine
- Datastore

Detailed Design

Front End

- The number of name suggestions will be limited to 5.
- Upon pressing enter, bring up all user results relating to the search string
- Highlight name on hover and on click, fill in the search bar with the name that was clicked and bring up user results based on name that was clicked
- Cycle through suggestions upon pressing up or down arrow keys and highlight the current name in the cycle. Upon pressing enter, complete the name in the search bar and pull up users with matching the chosen name.
- Add user to group upon pressing add icon



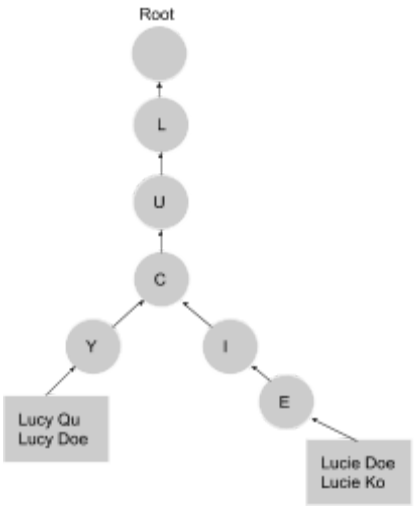


Trie Data Structure

A trie is an ordered tree data structure that uses strings or characters as keys. Each node of the tree contains multiple branches (children), and we can traverse down the branches of the tree to retrieve words.

For this implementation of using tries to find names based on a search string, create two trie instances, one containing first names and one containing last names. This way, suggestions will surface if the user types someone's last name into the search bar. Each trie will tie the first name or last name to the rest of the name. For example, in the trie for just first names, the end of the first name will then contain a set of full names, for which the first names match. Similarly, the last name trie will have a set of full names at the end of the last character in the last name.

The following image shows the trie data structure. It contains "Lucy" tied with full names "Lucy Qu" and "Lucy Doe" along with "Lucie" with full names "Lucie Doe" and "Lucie Ko." The last name trie follows a similar structure, whereby last names "Qu," "Doe," and "Ko" form the branches of the tree and the last letter holds a set of the full names.



Trie.java

1. Instance variables
    a. Map from Character to Trie object, indicating the node's children
    b. Set of Strings, containing the matching full names at the end of the last letter for a first or last name
2. Methods
    a. Insert - inserts either first or last name into trie
        i. Each character should be converted to uppercase to allow for case insensitive search on user input
        ii. Look at each character in the word, checking if each character is in the map of children. If it is not a child, add that letter in the map. Otherwise, recursively call the method on the next letter
    b. Autocomplete - find all words with given prefix

i. Find the ending node of the given prefix and from that node, look at its children and find all possible words from that ending node. Return a set of strings.

c. Whitespace - searches for whether the given word can be split by a space such that the newly formed full name exists in the trie
   i. Split the word at each index and find whether the first and last name exist in the set of full names in the trie
      1. Ie. “lucyqu” → “lucy qu” → add to set of full names
   ii. Check whether the reverse of the name exists in the trie
      1. Ie. “qulucy” → “qu lucy” → “Lucy Qu” is a name in the trie so add to set of full names

d. Autocorrect - Finds words within the defined maximum Levenshtein distance of 2.
   i. Levenshtein distance (AKA edit distance) between two words is the minimum number of insertions, deletions, or substitutions required to change one word to another.
      1. Ie. “Lucy” and “Lucie” have a Levenshtein distance of 2. The difference between the two names is that y is substituted with i, and the e is an insertion.
   ii. To find the edit distance between two words, we can use dynamic programming. In the following example comparing “Kate” and “Kat”, the edit distance (1) is found in the bottom right-most cell.

Table 1

		K	A	T	E
	0	1	2	3	4
K	1	0	1	2	3
A	2	1	0	1	2
T	3	2	1	0	**1**

   iii. Now, if we want to compare “Kate” and “Katt”, we have the following table.

Table 2

		K	A	T	E
	0	1	2	3	4
K	1	0	1	2	3
A	2	1	0	1	2

T	3	2	1	0	1
T	**4**	**3**	**2**	**1**	**1**

iv. The difference between Table 1 and Table 2 is merely the last row of Table 2. Thus, we can combine the trie data structure, recursion and dynamic programming to find words within the maximum edit instance of the search string while eliminating the need to recalculate the previous rows.

v. The worst case run time of this solution is $O(n * k)$, where n is the length of the search string and k is the number of nodes in the trie.

vi. For a trie with the names “Lucy,” “Lucie,” and “Lucinda,” the below table shows the matrix formed from comparing names with the search string (“Lucy”) while traversing down the tree. The resulting set of names that are within edit distance of 2 from the search string are “Lucy” and “Lucie.”

		L	U	C	Y
	0	1	2	3	4
L	1	0	1	2	3
U	2	1	0	1	2
C	3	2	1	0	1
Y(*)	**4**	**3**	**2**	**1**	**0**
I	4	3	2	1	1
E(*)	**5**	**4**	**3**	**2**	**2**
N	5	4	3	2	2
D	6	5	4	3	3
A(*)	7	6	5	4	4

Autocomplete and Autocorrect

1. Autocorrector.java - Class that calls the methods within the Trie class to return a set of name suggestions.
    a. Instance variables
        i. Trie firstNameTrie - trie containing first names of users, tied with their full names

        ii. Trie lastNameTrie - trie containing last names of users, tied with their full names
    b. Methods
        i. Suggest - returns set of suggestions
            1. Searches trie for all words with prefix, accounting for whitespace, and maximum edit distance of 2

Ranking Suggestions

1. Ranked in order of match
    a. Names with a complete first or last name match is ranked before other names
        i. Ie. If the search string is “Andrew”, suggestion “Andrew Sweet” will be ranked before “Andy Smith”
        ii. Ie. If the search string is “Leonard,” suggestion “Matthew Leonard” will be ranked before “Leonardo Vince”
    b. First name matches will be ranked before last name matches.
        i. Ie. If the search string is “Andrew”, suggestion “Andrew Sweet” will be ranked before “Mattew Andrew”
    c. Names ranked in order of Levenshtein distance from search string
        i. Ie. If the search string is “Lucy”, suggestion “Luce Chen” will be ranked before “Lucie Wang”

Querying Results

1. List results based on name inputted in search bar
    a. Query the database and pull up names where the given search matches the name
        i. Datastore does not allow for double filters--we cannot filter for matching first name and matching last name
        ii. Thus, add in a full name field in order to query the database. Full name field should be in all upper case because datastore does not allow for case insensitive querying. As such, when querying for the database, the name that needs to be queried can be made into uppercase, and then we can find matching names by filtering through the full name attribute.
2. If the user clicks on one of the suggested names or uses the arrow keys to select one of the suggested names, the results will be displayed with the users whose names *completely* match the selected name.
    a. This is done assuming that if the user specifically selects such a name, they want to find people only with that name.
3. Otherwise, results are pulled up containing users with related names.

Ranking Results

Results are ranked based on the current user’s relationship with suggested user

1. Classes
    a. Dijkstra.java

        i. Uses Java generics in order to work on any type of Edge or Vertex
    b. Edge.java (Interface)
    c. Vertex.java (Interface)
    d. UserEdge.java
        i. Implements Edge
    e. UserVertex.java
        i. Implements Vertex

Dynamically build weighted graph while searching for suggested user and note that user's distance from root node (current user)

1. To dynamically build the graph, we need a method "getOutgoingEdges," which queries the database for users that are in a shared group with the current user vertex
2. The weight of each edge is based on the number of groups that are shared between the two vertices (UserVertex object)
3. Maintain a HashMap mapping from user ID to UserVertex object. This is so that when finding outgoing edges and creating such edges, duplicate user vertices are not created, which will result in an infinite loop in Dijkstra's algorithm
4. Implement Dijkstra's algorithm to find the shortest path from the root to the target vertex
5. Create a HashMap, mapping User object to distance from root
    a. If there is a complete match between the search term and the name of the user, increment this mapped score by 10. This is so that users whose names are complete matches with the search string are ranked above related users (but not complete match) that may have a closer connection to the search user.
6. Order list of users by distance from root

Adding Member

1. Upon clicking the plus icon next to a particular user, add the user to the group.
    a. Update the datastore to include the user id as part of the group's list of member ids.
    b. Update the datastore to include the group id as part of the member's list of group ids.

Names Servlet

1. Gets all user names from the database. This is necessary for initially populating the tries and ensuring that the tries contain all of the names currently in the datastore.
2. Saving State
    a. Save Search Predictor object in file so that Tries do not have to be reconstructed and repopulated every time a server call is made. In Names Servlet, use init() and destroy() to write the Search Predictor object to file when server is closed and read from file during server calls.

Updating Tries

1. Every time a new user is added into the database, insert the name into the first and last name tries. Do this by reading the Search Predictor object from the file and then saving the updated object back into the file.

Testing Plan

Test User Data

1. Generate a list of first and last names stored in CSV file
2. Parse the CSV file in Java to return a list of full names by concatenating the first and last name
3. Insert test names into local database as a User Entity

Unit Tests

- Unit tests will be included to test all Java methods
    - Test full names are correctly returned on prefix
        - If user enters someone's last name, full name should show up
    - Users with the same first name and last name will show up
    - Test full names are correctly returned, taking into account white space
    - Test full names are correctly returned based on maximum edit distance
    - Insertion into trie
    - Test names are sorted in alphabetical order

Alternative Solutions Considered

- Have a helper method that uses dynamic programming to find the edit distance between two words. While the prefix length minus the search name length is less than the maximum edit distance, append the next character onto the prefix and recursively call the method on the new prefix. Once we reach the end of a name and the edit distance is less than or equal to the maximum edit distance, add name to the set of strings to be returned. For example, in a trie with "Lucy" and "Lucie," the helper method will be called with the following pairs of words: ("Lucy", "L"), ("Lucy", "Lu"), ("Lucy", "Luc"), ("Lucy", "Luci"), ("Lucy", "Lucie"). This means a 2D array is populated from the bottom up for each of these comparisons.
    - This solution has many recalculations that can be eliminated if we can just reuse the data instead of creating the edit distance matrix from scratch at every new node.
        - The run time is $O(n * m * k)$, where n is the length of the first string, m is the length of the second string, and k is the number of nodes in the trie.
    - The proposed solution eliminates these recalculations. The values in each line of the matrix are calculated using the previous row instead of having to recreate a new matrix. For example, when comparing "Lucy" and "L," we will calculate one point of edit distance values. Then, when we compare "Lucy" and "Lu," we use

that previously calculated row to create a new row for the edit distance between “Lucy” and “Lu.”
        - This allows for a worst case run time of O(n*k), where n is the length of the search string and k is the number of nodes in the trie
- Separate autocomplete and autocorrect, where autocomplete is suggested as user is typing in search bar and autocorrect is suggested on hitting enter
    - Combining autocomplete and autocorrect would allow for better real time feedback
- Have only one trie, containing the full name of the users
    - If user searches by last name, results will not show up since the trie contains only full names and search starts at the root
    - Although you would only need to recur down one trie to find matching names with the input, suggestions are not as robust because it assumes that the user will only be inputting names in order of first name followed by last name.
- Let the trie data structure have a map from String to Trie, where the string will be a character in the name, except when it reaches the end of the name for which the String will be the user’s full name in order to tie the first and last name together. There would then also be a boolean isEnd to indicate the end of the first or last name.
    - There is less clarity in this implementation in that sometimes the String represents one character in the name while other times it represents the full name.
    - The proposed solution offers a clearer distinction between keys in the trie and the set of full names. By having the HashMap be from Character to Trie and then containing a Set of Strings indicating full names, there is a distinction between keeping characters in the Trie as keys for tree traversal and storing a set of string representing full names.

Work Estimates

Code Timeline

- July 24 - Populate database with test data, complete trie insert, search with prefix, start frontend, and results
- July 31 - Add users to group upon pressing add icon, update tries when new users are added into database, implement white space and edit distance algorithm
- August 4 - Merge with master

Execution Timeline

- August 5: Finalize features, capstone presentation