Flavor Bible “Flavor” Graph Visualization Final Report

Rae Corrigan, Tina Liu, and Michael Rendleman

May 8, 2019

Project Description

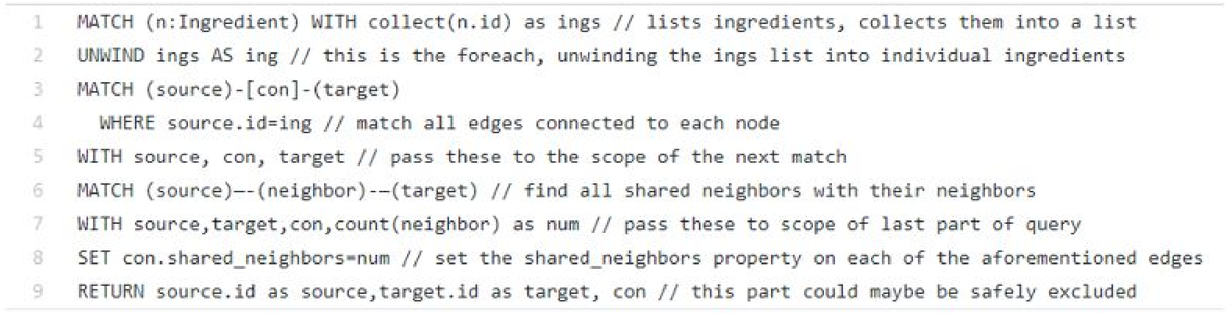
Our group used data from a cookbook that promotes culinary creativity called “The Flavor Bible” to create a web-based interface. Users can search for any food, food category, condiment, or spice, and generate its 3-D connected flavor subgraph. This 3-D subgraph consists of all nodes connected to the food, food category, condiment, or spice queried and are displayed based on edge weight. The better two foods go together and the more shared neighbors they have, the greater the weight of the edge between them and the closer a neighboring node will be displayed. Our graph displays both the food name for the nodes and the edge weight for relationships between nodes. This generated map of complimentary flavors can be used as a visualization tool for a user who is seeking to find inspiration for a new recipe.

Another feature our team implemented is a query we performed in the back end to find the most common and influential foods, condiments, or spices. This top 10 list is displayed to make it easy to explore different kinds of foods in the database. Because the most influential foods have the largest number of neighboring nodes, it is easy to find many flavors that surprisingly go well together. This tool can be useful for a user who wants to find a creative way to cook a common food like chicken.

Our final feature identifies communities in the data based on connection density. The idea is to find ingredients for recipes, and the community consisting of cliques of nodes with highly connected subgraphs would represent a recipe. To implement this, we researched the various algorithms already in Neo4j and chose the Louvain algorithm.

Architectural Overview

Neo4J was chosen for our graph database because it best fits our application’s dataset. Cypher is the query language used in Neo4J. Figure 1 shows a Cypher query that is used as part of our application, to conduct shared neighbor queries.



```
MATCH (n:Ingredient) WITH collect(n.id) as ings // lists ingredients, collects them into a list
UNWIND ings AS ing // this is the foreach, unwinding the ings list into individual ingredients
MATCH (source)-[con]-(target)
  WHERE source.id=ing // match all edges connected to each node
WITH source, con, target // pass these to the scope of the next match
MATCH (source)--(neighbor)--(target) // find all shared neighbors with their neighbors
WITH source,target,con,count(neighbor) as num // pass these to scope of last part of query
SET con.shared_neighbors=num // set the shared_neighbors property on each of the aforementioned edges
RETURN source.id as source,target.id as target, con // this part could maybe be safely excluded
```

Figure 1 The Cypher query used to find shared neighbors for a node of interest. Cypher is the Neo4j query language

Ruby on Rails allowed us to create an interactive, web-based application to serve on a localhost - an overview of the Rails architecture is shown in Figure 2. A user can use our autocomplete text input to select a food, condiment, or spice, and a map will be displayed, allowing the user to visualize flavors from other foods, condiments, or spices that complement their selection. The strength of compatibility between foods is based on our visualization distance metric.

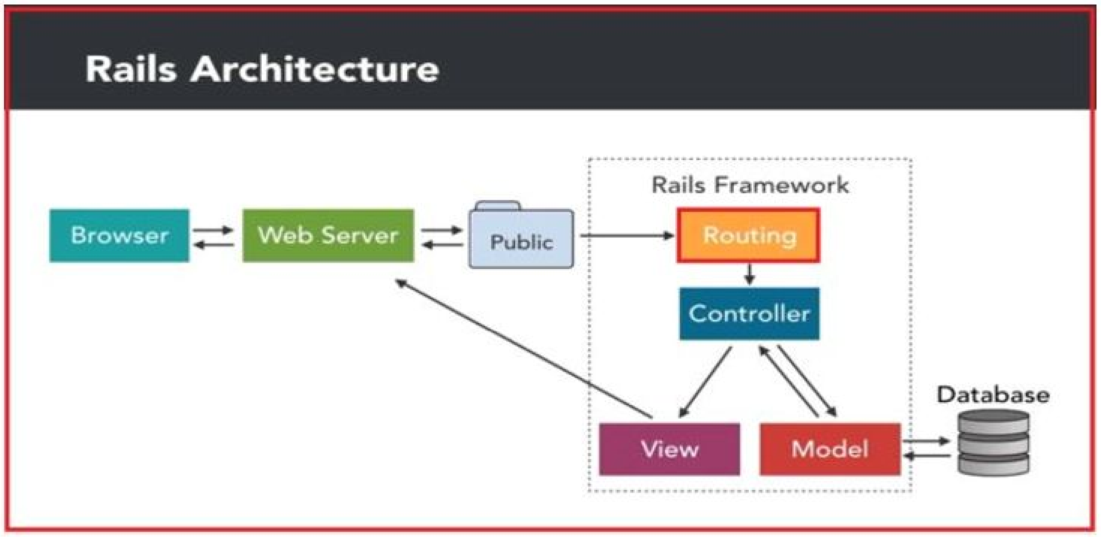


Figure 2 The basic architecture for a Ruby on Rails application. The Flavor Bible app adhered closely to this framework. Image source: https://www.c-sharpcorner.com/article/generate-a-controller-and-view-in-ruby-on-rails/

We defined a visualization distance metric for edges as 100 divided by the sum of the weight between the two nodes and the number of shared neighbor nodes because we wanted to incorporate both edge weight and common neighbor nodes in the display. The scalar 100 was chosen as a constant after trial and error to see what ranges of numbers would show up best. Because 3d-force-graph plots smaller edge distances closer to the source node and larger edge distances farther from the source node, we modified our metric by inverting it, so the edge weights and shared neighbor numbers are in the denominator. With our application, it is intuitive to link a flavor with strongly associated flavors.

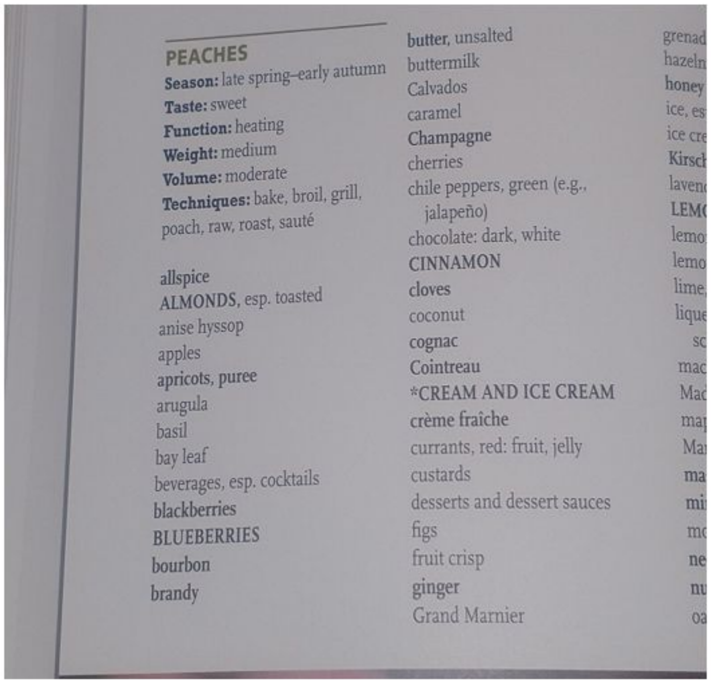
SCSS was used to style our web application. We integrated the Bootstrap framework as a plug-in to assist in creating a responsive application. HAML and Ruby were used to implement front-end and back-end, respectively.

A GitHub repository was set up as version control for our project. This made it easy for us collaborate, as we could individually work on different aspects of the project separately and then share code to combine our solutions. We were able to track our progress and issues in GitHub, making project management easy to organize.

Database Schema

Our data comes from the book "The Flavor Bible", and it originally consisted of 1,863 nodes and 20,316 edges to represent 1,863 different types of foods, food groups, seasonings, or condiments and 20,316 undirected, weighted edges to represent associations between the nodes. These weighted edges indicate the relative strength of the flavor association of how well the foods go well together, which can be weighted 1 (weakest association), 5, 20, or 60 (strongest association). These edge weights are based on notation in the Flavor Bible book.

For example, Figure 3 shows the entry in "The Flavor Bible" for peaches. In the list, neighboring nodes are listed below a short description of the food. Printed in normal text, for example apples, are the neighboring nodes with edge weight 1. If the text is in bold, for example **blackberries**, it means the edge weight is 5. Neighboring nodes listed in bold and all capitalized, for example **CINNAMON**, represents an edge weight of 20. Finally, ingredients listed in bold, capitalized, and an asterisk, for example ***CREAM AND ICE CREAM**, are neighbors to peaches with edge weight 60.



PEACHES
Season: late spring–early autumn
Taste: sweet
Function: heating
Weight: medium
Volume: moderate
Techniques: bake, broil, grill, poach, raw, roast, sauté

allspice
ALMONDS, esp. toasted
anise hyssop
apples
apricots, puree
arugula
basil
bay leaf
beverages, esp. cocktails
blackberries
BLUEBERRIES
bourbon
brandy

butter, unsalted
buttermilk
Calvados
caramel
Champagne
cherries
chile peppers, green (e.g., jalapeño)
chocolate: dark, white
CINNAMON
cloves
coconut
cognac
Cointreau
*CREAM AND ICE CREAM
crème fraîche
currants, red: fruit, jelly
custards
desserts and dessert sauces
figs
fruit crisp
ginger
Grand Marnier

grenad
hazeln
honey
ice, es
ice cre
Kirsch
laven
LEM
lemo
lemo
lime,
lique
sc
mac
Mac
ma
Ma
ma
mi
mc
ne
nu
oa

Figure 3 An exert from the Flavor Bible showing the description of and connections for peaches. Different text signals different connectivity weights, with plain text equally a weight of 1, bold text equaling a weight of 5, bold capitalized text equaling a weight of 20, and bold capitalized text with an asterisk equaling a weight of 60.

During the preprocessing stage, we cleaned up the data. The final dataset contains 1,749 nodes and 19,892 edges. Duplicate edges and duplicate nodes were deleted. Names of foods that included accents or tildes were renamed because the website does not display special characters correctly. For example, "crème fraîche" is now "creme fraiche" and "jalapeno" was changed from "jalapeño". In addition to accents and tildes on letters, we discovered that some special characters were not properly displayed on the web application. For example, all double dashes, -- , were replaced with a comma. The original database also included random bits of text from the original book, so it was necessary to manually go through and delete nodes that did not represent a valid food, condiment, or spice. For example, "toast coriander seeds to release their flavor", "even sweet desserts should be in balance", and "add at the end of the cooking process" were names of nodes we deleted. However, we decided to keep certain, short descriptions like "bitterness", "sourness", and "sweetness" to allow a user to, for example, find all bitter foods. Additionally, disconnected or poorly connected nodes (0-2 edges) were removed as they provided no useful flavor relationship information.

Data Statistics and Indexes

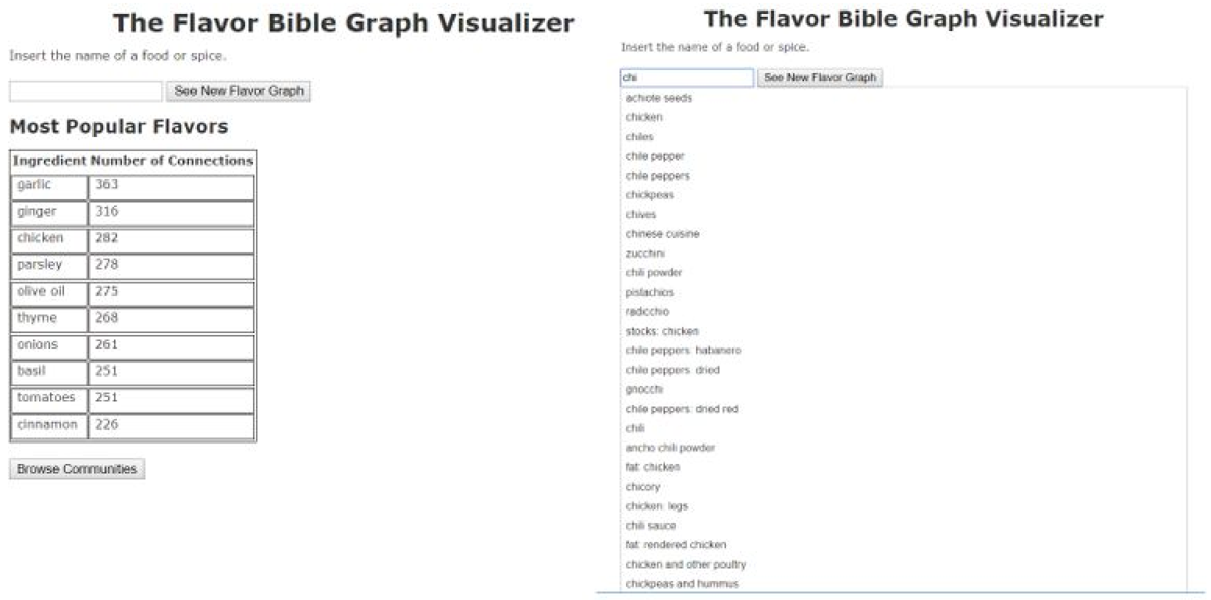
- 1,749 nodes
- 19,892 edges
    - 16,377 with an edge weight of 1
    - 2,824 with an edge weight of 5
    - 626 with edge weight 20
    - 45 with edge weight 60
- Index on node ID (ingredient names)

Tools

- Neo4j - graph database [1]
- Ruby on Rails - web application framework [2]
- HAML - markup language [3]
- SCSS - style sheet language [4]
- GitHub - version control [5]
- Cypher - Neo4j query language [6]
- Bootstrap - front-end framework [7]
- 3d-force-graph (open source) - visualize graph structures in web browser [8]

Features Designed and Implemented

First, we defined a distance metric for visualizing associating ingredients. It was important to consider both the edge weight between the 2 ingredients and the number of shared neighbors between them. So, our visualization distance metric is defined as the sum of the edge weight and number of shared nodes. The first feature we designed was the ability to visualize any node's local graph. On our homepage, a text box allows the user to search all possible ingredients with autocomplete, as shown in Figure 4.



The Flavor Bible Graph Visualizer

Insert the name of a food or spice.

Most Popular Flavors

Ingredient	Number of Connections
garlic	363
ginger	316
chicken	282
parsley	278
olive oil	275
thyme	268
onions	261
basil	251
tomatoes	251
cinnamon	226

The Flavor Bible Graph Visualizer

Insert the name of a food or spice.

chi

achiote seeds
chicken
chiles
chile pepper
chile peppers
chickpeas
chives
chinese cuisine
zucchini
chili powder
pistachios
radicchio
stocks: chicken
chile peppers: habanero
chile peppers: dried
gnocchi
chile peppers: dried red
chili
ancho chili powder
fat: chicken
chicory
chicken: legs
chili sauce
fat: rendered chicken
chicken and other poultry
chickpeas and hummus

Figure 4 Web application home page before (left) and after (right) starting a search. Autocomplete updates dynamically.

Because nodes can have anywhere from 3 to 363 neighbors, an analysis was performed for bottlenecks in queries. To obtain a local subgraph for a particular node, it is necessary to query for all its neighbors, which is done in three separate queries through the Neo4j Ruby driver. The first query finds the source node. As expected, this query time is approximately the same, or at least on the same order of magnitude, for all nodes. The second query found all relationships (neighboring nodes) associated with the node. This query took longer for ingredients for a large number of nodes, but not significantly longer. The final query finds all the non-source nodes associated with the relationships, meaning the number of shared neighbors between the two nodes. Results are shown in Table 1, with the node name, number of neighbors, three individual query times, and total retrieval times for 10 ingredients.

Test Nodes for Query Timing	Number of Neighbors	Source node query (s)	Relationships query (s)	Neighboring nodes query (s)	Total retrieval time (ms)
garlic	363	0.002182	0.028719	0.024545	55.446
ginger	316	0.001357	0.016335	0.006917	24.609
chicken	282	0.00192	0.018588	0.007263	27.771
parsley	278	0.00241	0.030658	0.009868	42.936
olive oil	275	0.006102	0.044496	0.007596	58.194
avocado	20	0.001978	0.018809	0.0023	23.087
hungarian cuisine	20	0.002118	0.029653	0.002134	33.905
eggs, frittata	20	0.002166	0.024875	0.001733	28.774
oil, hazelnut	20	0.002735	0.02461	0.003182	30.527
oil, walnut	20	0.002687	0.027884	0.001627	32.198

Table 1: To compare results, 5 ingredients were chosen with over 250 neighboring nodes and 5 ingredients were chosen with only 20 neighboring nodes.

Originally, the goal was to add indices to optimize this query. However, Neo4j Ruby Gem does not support relationship index querying because Neo4j adds an index on nodes by default. From the table, we concluded that the total retrieval time of all three queries ranged from 23 to 58 milliseconds and this difference was not significant when loading the local subgraph for a particular ingredient.

A 3D subgraph is rendered after the search is performed, and it shows more strongly associated flavors displayed closer to the target node, as shown in Figure 5. To do this properly in 3d-force-graph, we inverted and weighted the visualization distance metric to be 100/(EdgeWeight + Number of Shared Neighbors). The names of nodes and weight of edges are displayed on mouse-over.

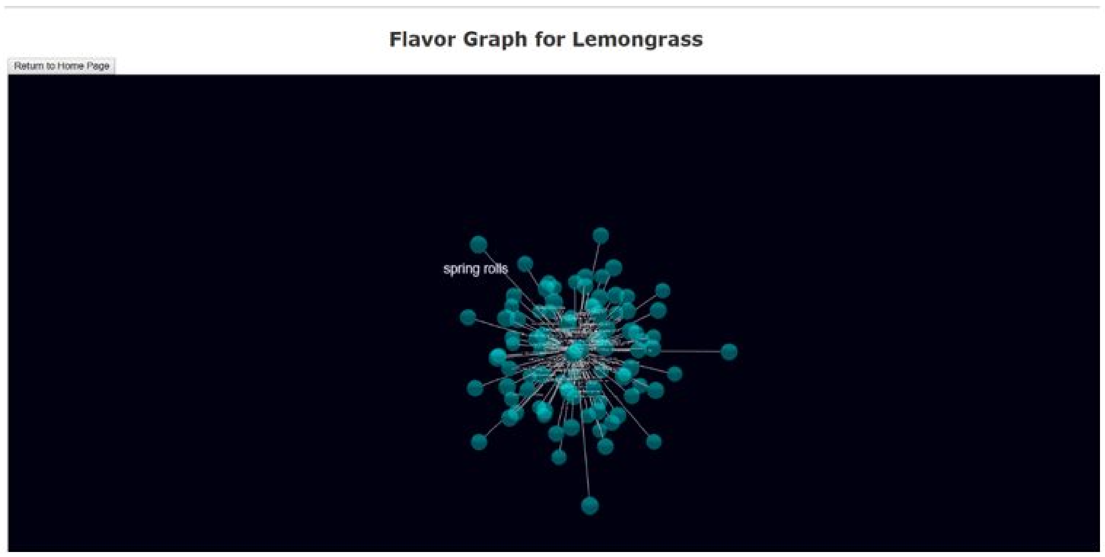
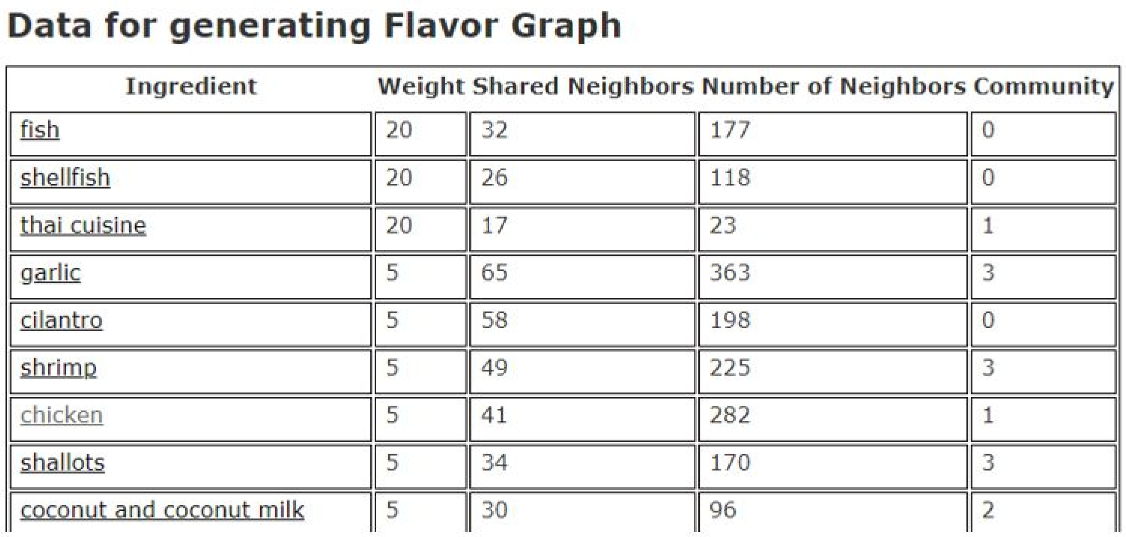


Figure 5 3D rendering of the flavors connected to lemongrass. Nodes closer to the central, lemongrass node are more complimentary flavors (i.e. they have a higher relationship weight), and nodes further from the central node are less complimentary flavors.

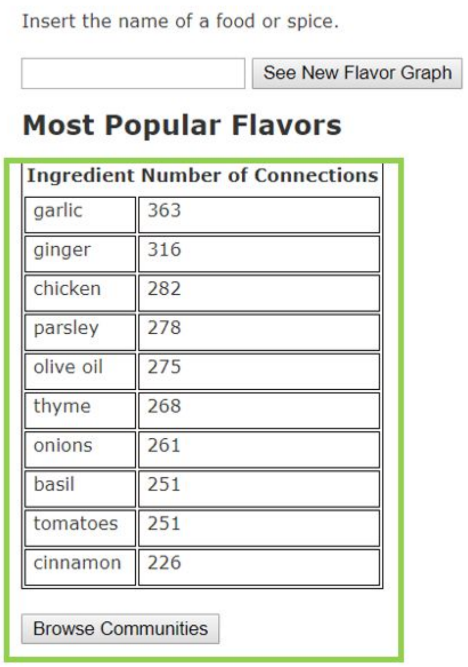
A table of all the neighbors, edge weights, and number of shared neighbors is displayed below the 3D graph, as shown in Figure 6. The user is able to jump from the queried ingredient's local graph to any of its immediate neighbors by clicking on the neighboring ingredient's name.



Data for generating Flavor Graph

Ingredient	Weight	Shared Neighbors	Number of Neighbors	Community
fish	20	32	177	0
shellfish	20	26	118	0
thai cuisine	20	17	23	1
garlic	5	65	363	3
cilantro	5	58	198	0
shrimp	5	49	225	3
chicken	5	41	282	1
shallots	5	34	170	3
coconut and coconut milk	5	30	96	2

Figure 6 Part of the table of connected nodes (complimentary flavors) for lemongrass including connection weight, how many neighbors each node shares with lemongrass, how many neighbors each node has total, and what community it belongs to, if any. Each flavor name is a link to that flavor's 3D graph.



Insert the name of a food or spice.

Most Popular Flavors

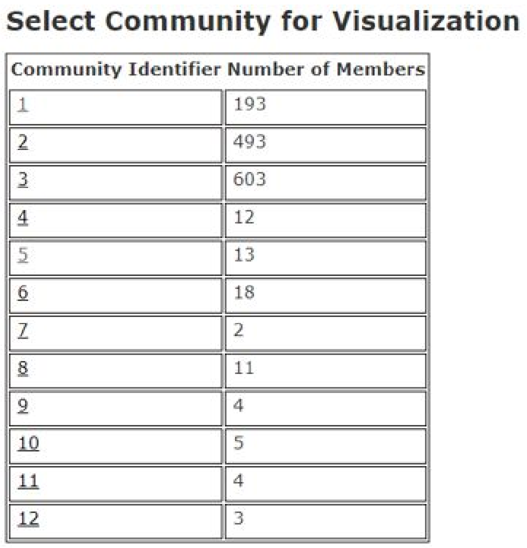
Ingredient	Number of Connections
garlic	363
ginger	316
chicken	282
parsley	278
olive oil	275
thyme	268
onions	261
basil	251
tomatoes	251
cinnamon	226

Figure 7 The website homepage includes a list of most popular flavors (highlighted in green) to help encourage culinary creativity. These flavors have the most complimentary flavors, which suggests that they can be used in a wide variety of recipes.

Our homepage has a table of the most popular ingredients, as shown in Figure 7. A query written in Cypher obtained the most popular ingredients by the number of neighboring nodes. The table displays the top 10 ingredients in order of decreasing number of neighbors. This encourages culinary creativity, as it gives the user inspiration for a common ingredient like “garlic” or “chicken”.

We identified “communities” in the data based on connection density, shown in Figure 8, via the Louvain algorithm [9], an algorithm already implemented in Neo4j. It creates clusters of ingredients composed of cliques of nodes with highly connected subgraphs. The idea behind this is to find ingredients for recipes, however, this is not realistic because some communities consist of over 600 nodes.

Cypher, the querying language used in Neo4j, was used to calculate our visualization distance metric, find the top 10 most common ingredients, and identify communities. Figure 1 shows the Cypher queries used to calculated shared neighbors.



Select Community for Visualization

Community Identifier	Number of Members
1	193
2	493
3	603
4	12
5	13
6	18
7	2
8	11
9	4
10	5
11	4
12	3

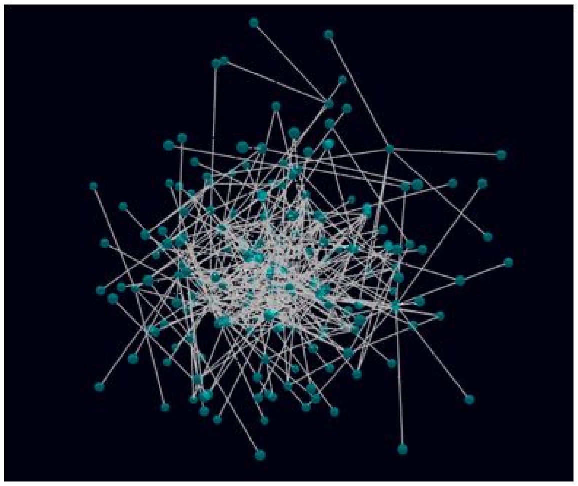


Figure 8 Community visualization is available for each of the 12 communities identified with Neo4j’s Louvain community detection algorithm. These communities represent highly connected subgraphs of flavor nodes. Community 1 is shown to the right of this figure.

Lessons Learned

While graph databases are becoming more popular and gaining traction in the web app development world, their relative newness means that some of the programs and plugins designed to work with them are not well documented or widely used. Because of this, baseline tasks, such as setting up the Rubymine IDE to interface with Neo4j via a third party plugin, proved more complicated with Neo4j than with other, more established database systems.

In addition, existing plugins do not always implement the same variety of features present in the original software. For example, it's simple to create an index on a particular relationship in Neo4j but querying those indices doesn't happen by default. This means that unless the driver software linking Neo4j and Rails' ActiveRecord supports those queries (which it doesn't) the index is unusable. This is unfortunate, seeing as the majority of the time associated with database accesses for displaying local flavor graphs is spent retrieving relationships.

The order of gems installed in a Gemfile on a Ruby project matters. Rails uses Bundle, which requires Gems to be in the order they appear in the Gemfile. Integrating Bootstrap in our project required the source priority to depend on the gems loaded, so order mattered significantly.

Open Issues and Future Work

Two additional features we tried to implement were node highlighting on mouse-over and redirection to a connected node's page on click. These features were partially implemented in the 3d-force-graphs source code, which we used as our model. However, javascript integration issues stalled the integration of these features, so we were not able to include them (although some base code remains in the "show.html.haml" script section).

Another potential future feature could traverse a random path in the graph, generating a small list of ingredients to suggest to use together in a recipe. We did some research and found this is possible in Neo4j, but we did not have time to implement this feature.

System Requirements

To install and run the project…

1. Install Ruby 2.4.4 and Rails 5.2.3.
    a. Visit https://www.tutorialspoint.com/ruby-on-rails/rails-installation.htm for installation instructions and more details.
    b. If using Windows, you'll also need Bash on Ubuntu on Windows: https://www.windowscentral.com/how-install-bash-shell-command-line-windows-10 All Ruby and Rails commands will be run out of this terminal.
2. Install Neo4j Community Edition.
    a. Go to https://neo4j.com/download/.
3. Clone our Git repository: https://github.com/rcorrigan/flavorbible.git
    a. This contains all code we used for the project as well as the data for our database in xml format.
    b. The graph data to use is called "tuesdayImport.xml".

4. Copy the “tuesdayImport.xml” file to the neo4j-community-3.5.5/import directory.
5. Run a Neo4j console.
    a. On Windows, this is ‘./neo4j-community-3.5.5/bin/neo4j.bat console’ in Git Bash.
    b. On Macs, use ./bin/neo4j console in the Terminal. Make sure you are in the neo4j-community-3.5.5 folder.
6. Start up a Rails Server within the root directory of our git repository.
    a. Use the command ‘rails server’.
7. Go to localhost:3000 in a web browser to interact with the website.

Feel free to contact our team via ICON with any questions as to set up or use of the app

References

[1] https://neo4j.com/developer/ruby-course/
[2] https://rubyonrails.org/
[3] http://haml.info/
[4] https://sass-lang.com/documentation/syntax
[5] https://github.com/rcorrigan/flavorbible
[6] https://neo4j.com/developer/cypher-query-language/
[7] https://getbootstrap.com/
[8] https://github.com/vasturiano/3d-force-graph
[9] https://neo4j.com/docs/graph-algorithms/current/algorithms/louvain/