Food science and nutrition using graph algorithms for recipe recommendation / CSCI4999-AKJ1

(2023-2024 T2) Final Year Project – Final Report

Students: Hyun Jun Yoo (1155100531)

Supervisor: Michael Ruisi Yu

Date: 17/04/2024

1. Introduction

Last semester, our project centered on the extraction and analysis of data from Food.com. Initially, we conducted web crawling to gather extensive food-related data and performed cleaning operations. We then modeled a sophisticated food graph model using the extracted data. We created two key components: a Recipe Information graph and Recipe Nutrition graph. Furthermore, we devised a scenario illustrating how to recommend food recipes to users based on these graphs.

Work Distribution

In this term, we divided the process into some parts. In my part, I was in charge of improving Cosine similarity which I did last semester and implementing web application construction (building user interactive functions), applying various graph layouts in the large dataset version of Recipe Information graph, and implementing main recommendation according to scenario we built. In Tae whei part, he took charge of the overall data preprocessing process, I-model, graph mining function (K-clique, Maximal clique) and side recommendation by using meta path sim.

Additions this last semester

This semester, to solve the memory usage and computing time problems that arose while calculating similarity last semester, we tried to improve the problem by calculating similarity using only the ingredients used in the two recipes. By adopting this strategy, we aim to mitigate memory usage and computation time, thus rendering our graph more robust and scalable. In this report, we will discuss the efficiency of this solution in constructing the graph, the extent of memory conservation achieved, and the impact on the structure of the graph.

Additionally, we tried to effectively apply the graph model to web applications. Through this, we have developed various functionalities aimed at enhancing user access to our graph. We implemented a recipe recommendation system based on the scenarios previously devised, supplemented by additional recommendation from graph mining tasks. Finally, we upload our project product to GitHub [1]. Our current works are available in GitHub [1]. User guideline is written in Read.me file.

Report Structure

The rest of the report is organized as follows. Section 2 -1 demonstrates what improvements were made to potential problems that arose when calculating Cosine similarity last semester.

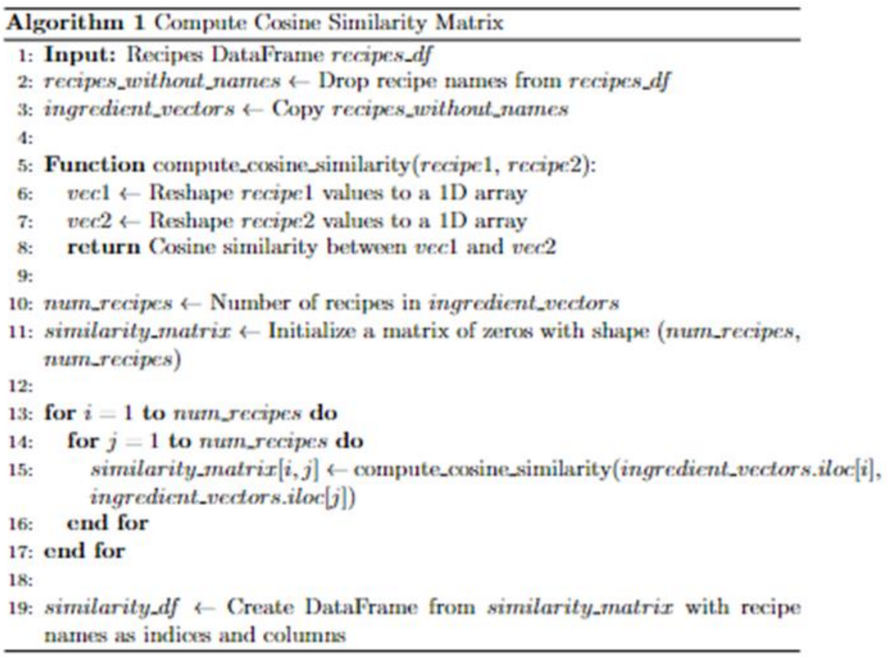
Section 2 - 2 explains the process of integrating our graph model into a web application. Section 3 offers a comprehensive demonstration of the functionality and efficacy of our recommendation system within the envisioned scenarios. Finally, Section 4 is the conclusion of our report.

2. Improvements to the System

1) Improved Cosine Similarity

The provided pseudo code in Algorithm1 of Fig. 1 calculates cosine similarity between recipes using the *scikit-learn library* and *NumPy*. Line 2 to 3 drop the *recipe_names* column and copy *recipes_without_names* with *ingredient_vectors*. Line 5 to 8 is a function for calculating cosine similarity between *recipe1* and *recipe 2* vectors. It takes two recipe vectors as input, reshape them into one-dimensional arrays. Line 13 to 17 use nested loops iterate through each pair of recipes, calling the *compute_cosine_similarity* function for each pair. Line 19 is the resulting similarity values are stored in the similarity matrix.

The previous method of calculating cosine similarity between recipes faced several limitations, primarily concerning memory efficiency and computational speed. By constructing a matrix encompassing all recipes and all ingredients, the approach incurred significant memory overhead, particularly as the data size increased. For instance, with 1000 recipes and 1266 unique ingredients, the memory requirements become prohibitive. Additionally, the computational time required for calculating cosine similarity for each pair of recipes was substantial, leading to performance bottlenecks. When calculating the cosine similarity for each pair of recipes using the previous method, the algorithm had to iterate through every combination of two recipes in the dataset. For each pair of recipes, the algorithm constructed their corresponding ingredient vectors and then computed the cosine similarity between these vectors. This process involved nested loops iterating over the entire dataset, resulting in a quadratic time complexity with respect to the number of recipes. The new approach addresses these limitations by focusing solely on the ingredients used in each pair of recipes, thereby significantly reducing memory consumption and computational time. This optimization not only simplifies the calculation process but also ensures that only relevant information directly contributing to recipe similarity is considered, resulting in a more efficient and meaningful analysis.



```
Algorithm 1 Compute Cosine Similarity Matrix
1: Input: Recipes DataFrame recipes_df
2: recipes_without_names ← Drop recipe names from recipes_df
3: ingredient_vectors ← Copy recipes_without_names
4:
5: Function compute_cosine_similarity(recipe1, recipe2):
6:    vec1 ← Reshape recipe1 values to a 1D array
7:    vec2 ← Reshape recipe2 values to a 1D array
8:    return Cosine similarity between vec1 and vec2
9:
10: num_recipes ← Number of recipes in ingredient_vectors
11: similarity_matrix ← Initialize a matrix of zeros with shape (num_recipes,
    num_recipes)
12:
13: for i = 1 to num_recipes do
14:    for j = 1 to num_recipes do
15:       similarity_matrix[i, j] ← compute_cosine_similarity(ingredient_vectors.iloc[i],
          ingredient_vectors.iloc[j])
16:    end for
17: end for
18:
19: similarity_df ← Create DataFrame from similarity_matrix with recipe
    names as indices and columns
```

Figure 1. Previous cosine calculation pseudo code

New Approach

In current method, we calculate the cosine similarity between pairs of recipes based on their ingredients. In line 8 to 9, for each pair of recipes, we extract the ingredient names and amounts from *ingredients_dict_1* and *ingredients_dict_2*. Then, we combine the ingredients lists and removes duplicates to get a list of all unique ingredients present in both recipes using *all_ingredients*. For each recipe, we construct a numerical vector (*vector1, vector2*) where the elements correspond to the amount of each ingredient in the recipe. The cosine similarity between the two vector is calculated using the cosine similarity function from *scikit-learn*.

Previously, the method involved constructing a matrix encompassing all recipes and ingredients, which consumed a large size of matrix, especially as the dataset scaled up in terms of recipe and ingredient count. Consequently, this approach led to memory issues, particularly with large datasets.

On the other hand, second method which is using only the ingredients used in the two recipes, was much more efficient in terms of memory and time. First, we showed much more efficiency in terms of memory by constructing a matrix consisting only of the recipe being compared and the ingredients used in the tow recipes, rather than having to construct a matrix for all recipes and all ingredients. In terms of time, this method was able to complete the comparison between all recipes in 213 seconds.

This method not only simplifies the calculation process, but also maintains the essence of the analysis by prioritizing ingredients that directly contribute to recipe similarity. In addition, we will explain the direct measurement results of how much effect we were able to achieve in terms of memory and time using this method, and what kind of graph shape was ultimately formed.

```
**Algorithm 2** Compute Cosine Similarity and Create Graph
Input: Recipes Dictionary $recipes\_dict$
Initialize empty list $cosine\_similarity\_scores$
for each $recipe\_name\_1, ingredients\_dict\_1$ in $recipes\_dict$ do
  for each $recipe\_name\_2, ingredients\_dict\_2$ in $recipes\_dict$ do
    if $recipe\_name\_1 \neq recipe\_name\_2$ then
      Initialize empty list $all\_ingredients$
      Get ingredient names and amounts for both recipes
      $ingredients\_1 \leftarrow$ list of keys of $ingredients\_dict\_1$
      $ingredients\_2 \leftarrow$ list of keys of $ingredients\_dict\_2$
      $all\_ingredients \leftarrow$ list of unique elements in $ingredients\_1$ and
      $ingredients\_2$
      if $all\_ingredients$ is not empty then
        Initialize empty lists $vector\_1$ and $vector\_2$
        Construct vectors $vector\_1$ and $vector\_2$ with ingredient amounts
        Compute cosine similarity between $vector\_1$ and $vector\_2$
        Append similarity score to $cosine\_similarity\_scores$
      end if
    end if
  end for
end for
```

Figure 2. Current cosine calculation pseudo code

2) Web Application Implementation

In this semester, we worked on a project to develop a web application for a food recommendation system, building upon the food graph model developed in the previous semester. We investigated together what functions to use to apply graph to web applications, and Taewhei investigated how to apply and run graph to web applications, and I implemented the functions that users needed afterward.

Preliminaries: Sigma.js

Throughout this process, we primarily utilized Sigma.js for visually presenting information to the user. While we explored other techniques mentioned in an article on useful JavaScript techniques for network graph visualization, we found Sigma.js to be the most suitable and practical for our project needs, thus adopting its technique.

Sigma.js is an open-source JavaScript library developed explicitly for graph drawing. It leverages modern web technologies like HTML5 and WebGL to produce high-quality, interactive, and scalable graph visualization within web browsers. We opted for Sigma.js for two main reasons. First, it can effectively handle large numbers of nodes and edges without sacrificing performance. This is achieved through its utilization of WebGL for rendering, enabling faster drawing of larger graphs compared to solutions based on Canvas or SVG. Additionally, Sigma.js offers customizable rendering options, allowing our food graph model to provide users with a diverse range of visualizations.

Cytoscape.js is a powerful JavaScript library for visualizing and analyzing networs or graphs on the web. While Cytoscape.js also offers specialized tools for network and graph visualization, we chose Sigma.js due to its focus on fast rendering of large graphs, which aligns with the scale of graph data we are dealing with this term. Secondly, we anticipated an easy transition of the graph environment and the creation of various interactive functions. Last semester, we constructed a graph using the networkx library in Python and visualized it using the Gephi tool to examine its structure and explore various insights. Similarly, Hudan Studiawan et al. [3] demonstrated importing a graph from Gephi and exporting it to Sigma.js, validating its suitability for our project. Moreover, Sigma.js utilizes the graphology library for building and manipulating graph data, facilitating easy reading of our graph file format (GEXF) and implementation of various graph interaction functions using diverse graph APIs.
Here are some key points about the GEXF file format:

1) XML-based Structure: GEXF files are based on XML (eXtensible Markup Language), which is a popular markup language for encoding structured data. XML uses tags to define elements and attributes, making it easy to parse and manipulate using programming language.
2) Graph Structure: As its core, a GEXF file represents a graph consisting of nodes and edges. Nodes typically represent entities, while edges represent relationships or connections between entities.

3) Node and Edge Attributes: Nodes and edges in GEXF file can have various attributes associated with them. These attributes provide additional information about nodes and edges such as labels, colors, sizes.

Large-Scale Graph visualization

In this part, we will explain how we connected our graph model to Sigma.js for our web application and elaborate on the various graph layouts presented to users based on the graph's data size. We expanded it further this semester to create Recipe Information graph by creating 1,000 recipe nodes and about 1,200 ingredient nodes using the 1,000 recipes dataset. Additionally, we'll discuss the limitations we encountered with Sigma.js as our data expanded.

Layout Selection and Refinement

As we investigated the layout of the graph that grows by increasing data, we considered three graph layout methods (Random Layout, Fruchterman-Reingold (FR), ForceAtlas2 (FA2).
Fig 3. Shows the graph when a random layout is applied. Random layout has the advantage of being very simple and speed-wise, as it does not require algorithmic computation, generating a random layout is very fast. However, it does not contain any meaningful structural information about the graph. Nodes are positioned arbitrarily, making it difficult to discern any underlying patterns or relationships. Additionally, as can be seen in Figure 3, random layout often results in overlapping nodes or edges, which can make the visualization cluttered and difficult to interpret.

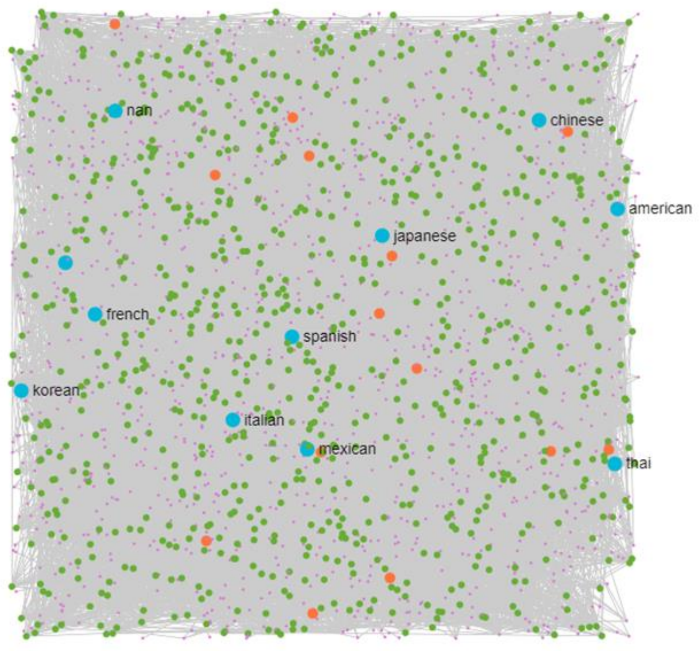


Figure 3. Recipe information graph (1000 recipes) with random layout.

The Fruchterman-Reingold (FR) and ForceAtlas2 (FA2) layouts are two popular algorithms used for graph layout in network visualization. The FR layout algorithm simulates a physical system where nodes repel each other like charged particles and edges act as springs pulling connected nodes together. The algorithm iteratively adjusts the positions of nodes based on forces acting upon them until the system reaches equilibrium, resulting in an aesthetically pleasing arrangement where nodes with stronger connections tend to be closer together. ForceAtlas2 is an improved version of the original ForceAtlas layout algorithm, introduced by Mathieu Jacomy et al. It employs various optimizations to handle large-scale graphs efficiently, such as adaptive scaling of forces and multilevel modularity optimization. FA2 is particularly well-suited for visualizing large networks with thousands or even millions of nodes and edges, making it a popular choice for interactive graph exploration and analysis in applications like network science, social network analysis, and data visualization.

As a result, applying the Fruchterman-Reingold layout (FR) in Figure 4, nodes are distributed more evenly in space than random layout, providing a balanced layout. Jacomy M et al. [5] studied the performance of various layouts and found that FR had very slow performance in large data sets. (Figure 5) On the other hand, FA2 has performed much better than FR. While the FR layout algorithm produces visually appealing layouts, it can be computationally expensive for very large graphs. As the number of nodes and edges increases, the algorithm's runtime may become impractical interactive visualizations. They also said that FR is suitable for small datasets, and FA2 has a better measured quality for large datasets.

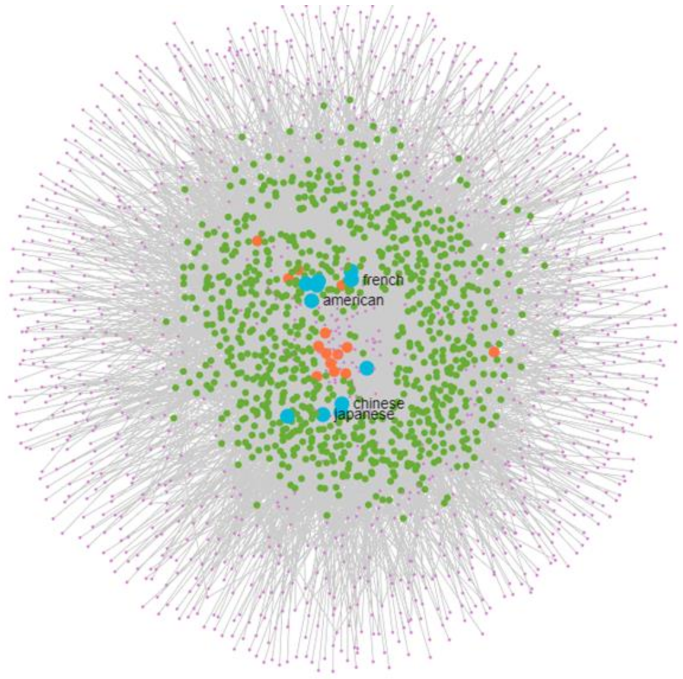


Figure 4. Recipe information graph (1000 recipes) with FR layout

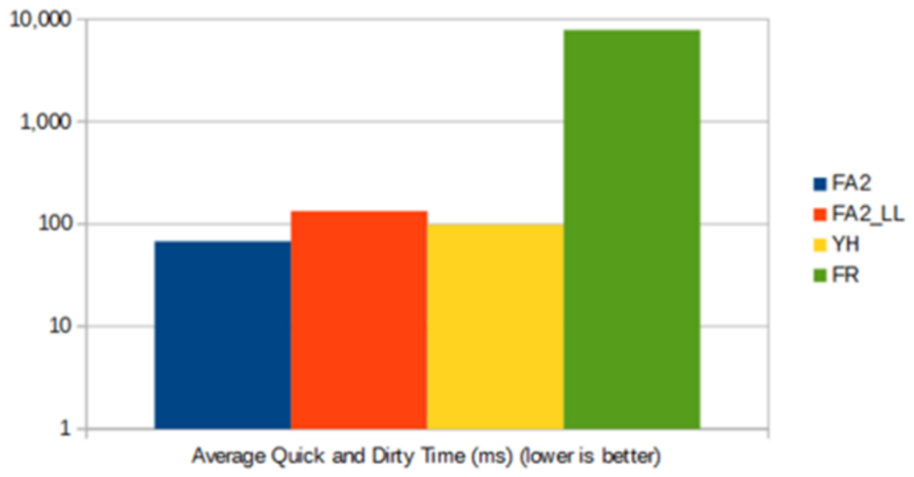


Figure 5. Result of performance of four layout algorithm [5]

In the case of FA2, we thought it was suitable for our large-scale food datasets because it is designed to handle large-scale networks efficiently. FA2 dynamically adjusts the strength of forces acting on nodes based on their local neighborhood density. This feature helps in preventing nodes from overlapping and ensures a more aesthetically pleasing layout. It can also help us identify clusters or groups within our graph through its Linlog mode.

As illustrated in Figure 6, the graph generated from 100 recipes appears densely packed, presenting a challenge in visual clarity. To solve challenge in visual clarity, I experimented with adjusting the scaling parameters during the creation of the gexf file using Gephi. I experimented with various layout algorithms supported by Sigma.js, including ForceAtlas2, Random layout, and Circular layout, in an attempt to address the density issue. Despite these efforts, I have encountered challenges and have yet to find a satisfactory solution. The reason I think so far is that there is a limit to the canvas size when opening a web application through Sigma.js. The size of the canvas or container in which I am rendering the graph could be limiting how far apart the nodes can spread. This seems to occur because the container holding the graph does not have enough space to accommodate the desired spread.

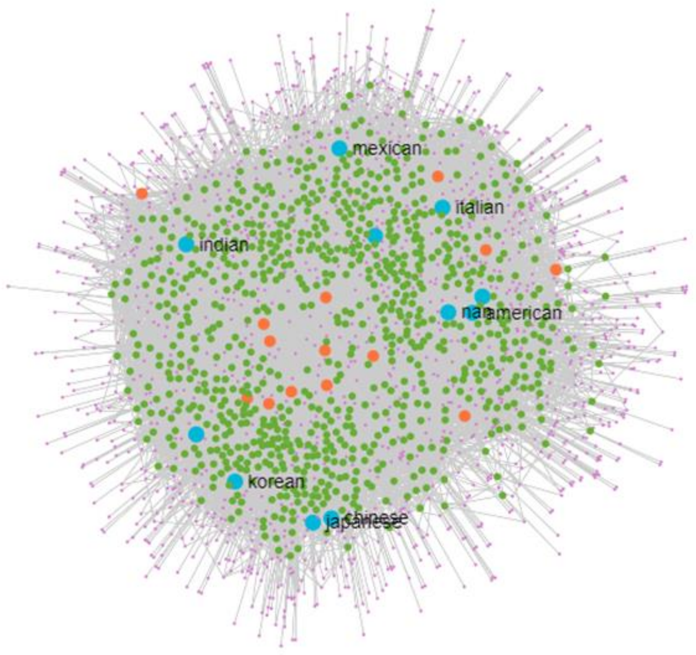


Figure 6. Recipe information graph with FA2 layout

Implementation of Interactive graph functions

In this part, I have created a suite of interactive graph functions tailored to meet the diverse needs of users. With understanding of the challenges posed by large-scale graph visualization, our focus has been twofold: Easily provide users with the concise, comprehensive information users want for data of any size and give users the ability to delve deeper into the complexity of the graph as they choose. Each feature has been developed with this purpose in mind, and we will explain exactly what the features are and what effect each feature can have on users.

User-friendly food graph navigation features

a. Searching specific ingredient with neighbors

This functionality enables users to focus more intently on exploring recipes and ingredients related to their searched ingredient. Users can check all the ingredient nodes in our graph, and if the user click specific ingredient, the searched ingredient and its surrounding neighboring nodes are visually highlighted. This allows users to easily identify related recipes. Additionally, with other nodes hidden, users can concentrate solely on their primary area of interest. This feature streamlines recipe exploration, helping users quickly find information related to the ingredients they're interested in. In Figure 7, we brought in how the function works in our graph. User can identify the ingredients in the graph and when the user selects the flour ingredient, the camera zooms in to the flour ingredient and at the same time, the corresponding neighbors are also shown hovering.

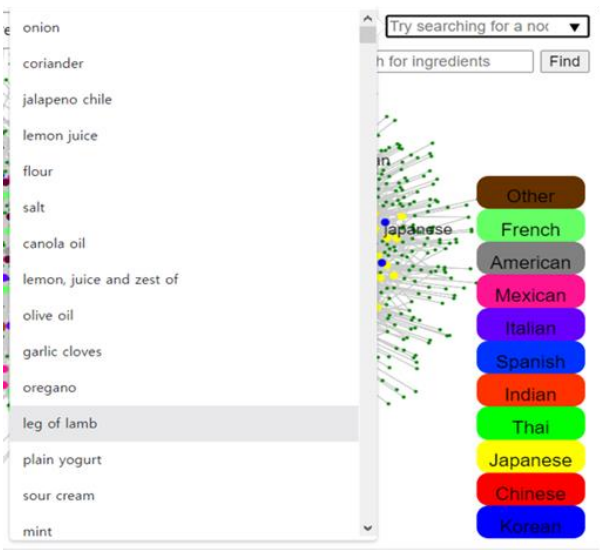


Fig 7(a) ingredient list for selection

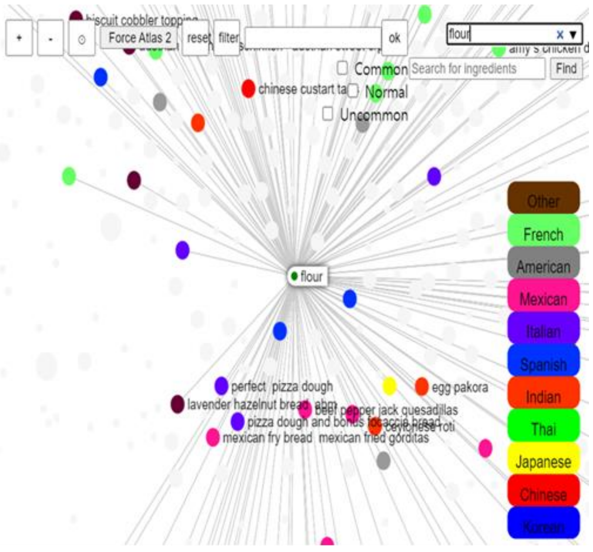
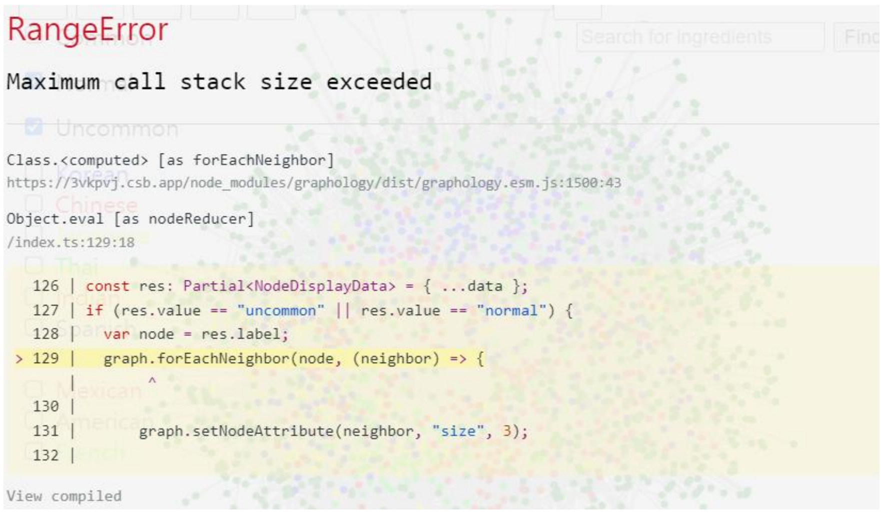


Fig7(b) updated screen reflecting selection

b. Filter ingredient function

We've implemented a filtering feature based on ingredient classification. This functionality allows users to streamline their recipe exploration process by categorizing ingredients into common, uncommon, and normal categories and then filtering recipes based on these classifications. When users select a checkbox corresponding to a specific ingredient classification, the graph dynamically adjusts to display relevant recipes and ingredients. For example, if a user chooses the "common" checkbox in Figure 10, the graph will show recipes connected to ingredients classified as common. This enables users to focus their exploration on ingredients that are commonly used in recipes.

In this function, it ran without problems when running on 100 recipe datasets. But when using a dataset of 1000 recipes, an error occurred: Maximum call stack size exceeded. This error is now an error that occurs when too many functions are accumulated in the call stack in JavaScript. In the case of my function, when the user checks uncommon, graph.setNodeAttribute (to make neighbors highlight) function was executed on neighbors connected to uncommon ingredient. However, in this case, a neighbor node where the setNodeAttribute function was already executed in another uncommon ingredient node duplicated the function in another uncommon ingredient node, causing the call stack size to overflow.



```
RangeError
Maximum call stack size exceeded

Class.<computed> [as forEachNeighbor]
https://3vkpvj.csb.app/node_modules/graphology/dist/graphology.esm.js:1500:43

Object.eval [as nodeReducer]
/index.ts:129:18

  const res: Partial<NodeDisplayData> = { ...data };
  if (res.value == "uncommon" || res.value == "normal") {
    var node = res.label;
>   graph.forEachNeighbor(node, (neighbor) => {
          ^

        graph.setNodeAttribute(neighbor, "size", 3);


View compiled
```

Figure 8. Screen of maximum call stack size error

To mitigate the density issue, I implemented a solution involving dynamic node resizing when highlighting neighbors connected to an uncommon node. Specifically, I incorporated this functionality into a conditional statement to bypass nodes whose sizes had already been adjusted. By line 5 conditional statement in Figure 8, instead of accessing all neighbors, duplicate neighbors are omitted. By doing so, I aimed to reduce the number of functions accumulating in the call stack during the problem-solving process, thus optimizing performance.

```
Algorithm 3 Ingredient Node Selection
 1: Function Check Ingredient Type()
 2: $renderer.setSetting("nodeReducer", (\text{node}) \rightarrow$
 3: if $node.type == "uncommon"$ then
 4:       $graph.forEachNeighbor(node, (\text{neighbor}) \rightarrow$
 5:       if $neighbor.size \neq 3$ and $neighbor.nodedef \neq "ingredient")\{$
 6:          $neighbor.size = 3;$
 7:        $\}$
 8:        $res.color \leftarrow red;$
 9:       $res.size \leftarrow 3;$
10: end if
11: if $res.size \neq 3$ then
12:     $res.hidden = \text{true};$
13: end if
14: return $res;$
```

Figure 9. Filtering uncommon ingredients node pseudo code

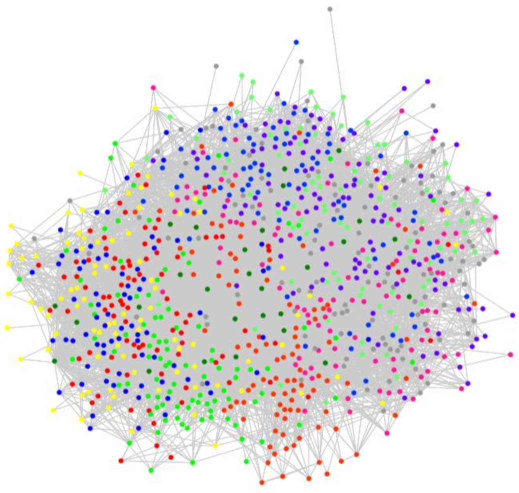


Figure 10 (a). Subgraph when the user clicks the common check box

(Dark green node: common ingredients, other color nodes: recipe node)

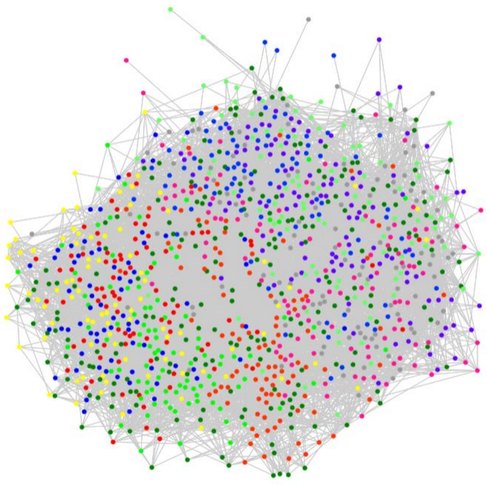


Figure 10 (b). Subgraph when the user clicks the normal check box

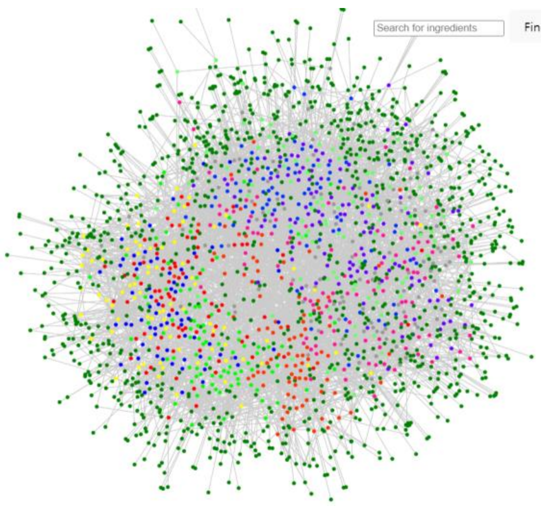


Figure 10 (c). Subgraph when the user clicks the uncommon check box

c. Cuisine filtering

I have implemented a feature aimed at enhancing user experience and facilitating culinary exploration. This functionality allows users to streamline their recipe browsing experience by narrowing down their focus to specific cuisines of interest. Upon accessing the recipe graph interface, users will notice a convenient checkbox system designed to cater to various culinary preferences. By simply selecting the checkbox corresponding to their desired cuisine, users can seamlessly filter out recipes and ingredients unrelated to their chosen culinary theme. This empowers users to delve deeper into the culinary traditions they are most interested in exploring, whether it’s Chinese, Korean, Japanese or any other cuisine available in our extensive graph.

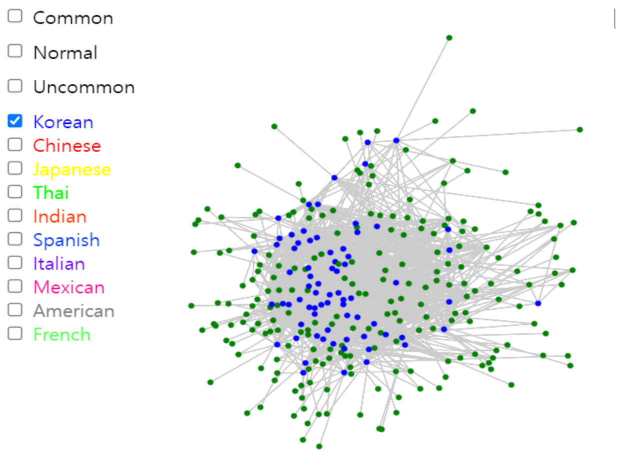


Figure 11. Subgraph that appears when the user clicks the Korean check box

(Blue node : Korean recipe node, green node : ingredient node)

d. Recipe similarity notification function

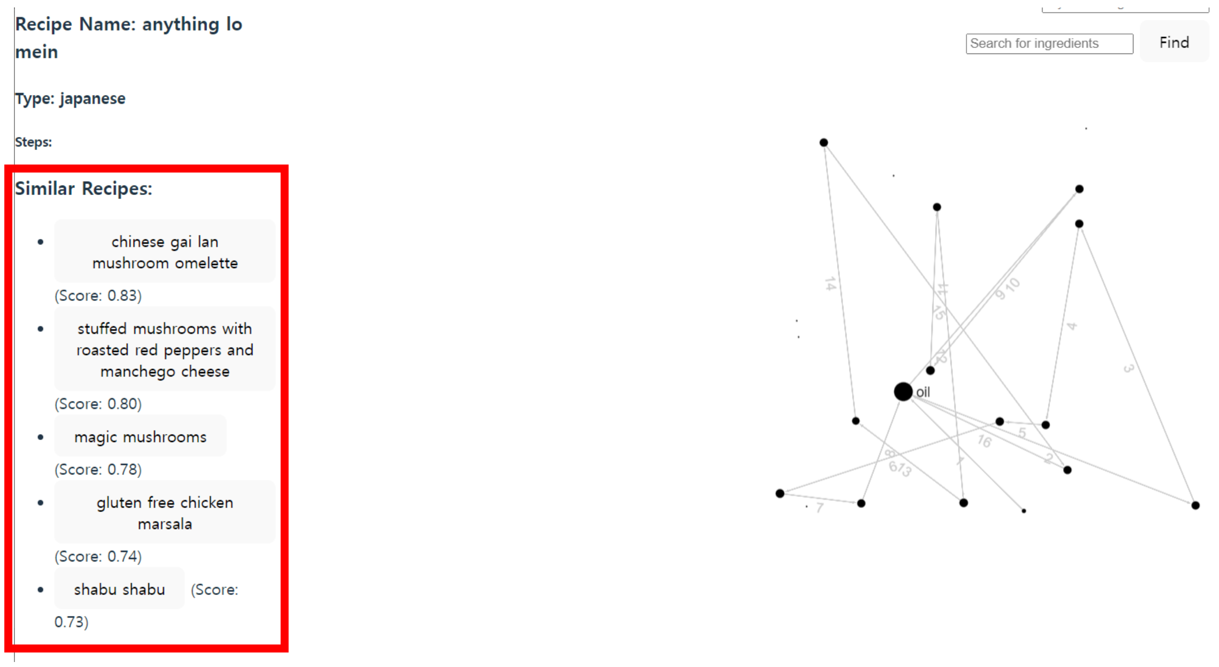
In our recipe graph project, we build a function that leverages advanced techniques such as Cosine similarity to notify users of the five most similar recipes when they click on a specific recipe. When users interact with a particular recipe in the graph, our system lists recipes in order of highest similarity score among the recipes linked to the recipe, then finds the top 5 and provides information to the user. This provides users with valuable suggestions for exploring variations of the selected recipe or discovering similar recipe that aligns with their culinary preferences. Whether users are seeking alternatives versions of their favorite dishes or exploring new culinary inspirations, this feature offers a personalized and insightful tool to enrich their recipe exploration journey.

In figure 12, line 1 to 2 set *neighbors* of user selected node in *res*. In line 5, we get the similarity value by using *graph.getEdgeAttribute* which is providing in *graphology* library. Then in line 6 to 10, we build *recipeObj* for storing recipe name and similarity value. After pushing neighbors object in *recipeList*, we sorted in descending order to extract top 5 similar recipe.

```
Algorithm 4 Extracting Recipe Neighbors and Similarity Scores
 1: $res \leftarrow$ neighbors($node$)
 2: $simvalue \leftarrow 0$
 3: $recipeList \leftarrow []$
 4: for $neighbor$ in $res$ do
 5:     $simvalue \leftarrow$ graph.getEdgeAttribute(node, neighbor, "value");
 6:     $recipeObj \leftarrow$ {
 7:         name : $res[i]$,
 8:         score : simvalue,
 9:     };
10:     $recipeList.push(recipeObj)$;
11: end for
12: $recipeList$.sort($descending$)
13: top5Recipes $\leftarrow$ recipeList.slice(0, 5)
```

Figure 12. Pseudo code to find top 5 similar recipes

Through this function, we were able to obtain the recipe most like each recipe as shown in the figure, but there were cases where no information could be obtained. The reason this problem occurs is because we set the similarity score threshold to 0.7 to prevent too many edges from being created when calculating the similarity score. With this threshold setting, edges were not formed for recipes with low similarity scores to other recipes, so there was a limitation in providing information about recipes similar to the recipe in figure 13 (b).


Recipe Name: anything lo mein

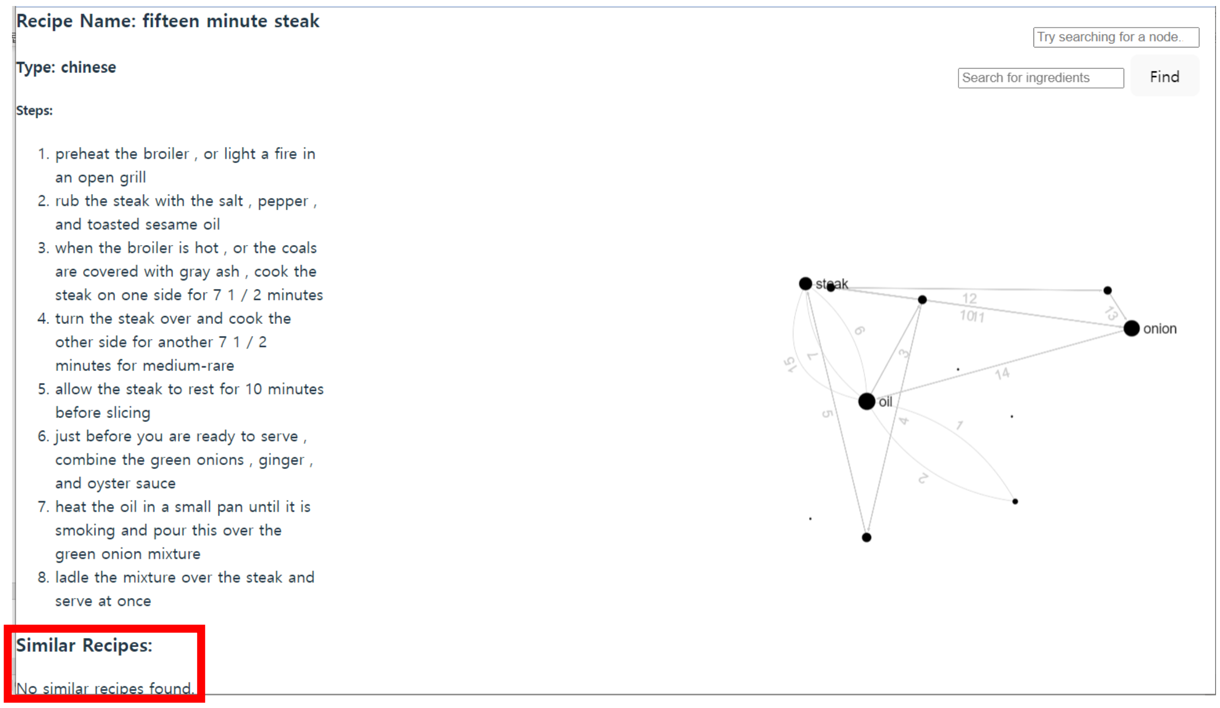
Type: japanese

Steps:

Similar Recipes:

- chinese gai lan mushroom omelette (Score: 0.83)
- stuffed mushrooms with roasted red peppers and manchego cheese (Score: 0.80)
- magic mushrooms (Score: 0.78)
- gluten free chicken marsala (Score: 0.74)
- shabu shabu (Score: 0.73)

Figure 13 (a). The red box section provides information on the top 5 recipes like the recipe


Recipe Name: fifteen minute steak

Type: chinese

Steps:

1. preheat the broiler , or light a fire in an open grill
2. rub the steak with the salt , pepper , and toasted sesame oil
3. when the broiler is hot , or the coals are covered with gray ash , cook the steak on one side for 7 1 / 2 minutes
4. turn the steak over and cook the other side for another 7 1 / 2 minutes for medium-rare
5. allow the steak to rest for 10 minutes before slicing
6. just before you are ready to serve , combine the green onions , ginger , and oyster sauce
7. heat the oil in a small pan until it is smoking and pour this over the green onion mixture
8. ladle the mixture over the steak and serve at once

Similar Recipes:

No similar recipes found.

Figure 13 (b). Case of no top 5 similar recipes similar in the red box

e. K-Core Graph mining function

We've introduced a graph mining feature known as the K-core functionality. K-core of a graph is a maximal subgraph in which every vertex has degree at least K within that subgraph. In simpler terms, it's like peeling off layers of a graph until you're left with the densest core where each node is connected to at least K other nodes within that core. This tool utilizes graph theory techniques to extract and display subgraphs that adhere to the k-core property. When users input a specific value for k. the K-Core functionality generates a subgraph from the existing recipe graph, adhering to the k-core criteria. The k-core of a graph refers to the maximal subgraph in which every vertex is connected to at least k other vertices within the subgraph. By leveraging this function, users can explore the underlying structure properties of the recipe graph by focusing on densely connected regions represented by the k-core. This provides users with insights into the core components of the graph and highlights significant relationships between recipe and ingredients. Whether users are interested in analyzing the robustness of recipe connections or identifying key ingredients that contribute to the overall graph connectivity, this function offers a versatile and insightful tool for graph exploration and analysis within the recipe domain. As a result of executing the K-core function on a graph consisting of 1000 recipes, subgraphs of K up to 15 were formed, and it looked like Figure 15.

Algorithm 5 K-core Function

```
$k \leftarrow$ k-value
Flag $\leftarrow$ false
for $i \leftarrow 0$ to $k$ do
  while true do
    $c \leftarrow 0$
    $graph.forEachNode(node) \rightarrow$
      $degree =$ graph.degree(node)
      if $(degree < i)\{$
        graph.dropNode($node$)
        $c \leftarrow c + 1$
      $\}$
    $\})$
    if $c == 0$ then
      break
    end if
  end while
end for
```

Figure 14. K-core function pseudo code

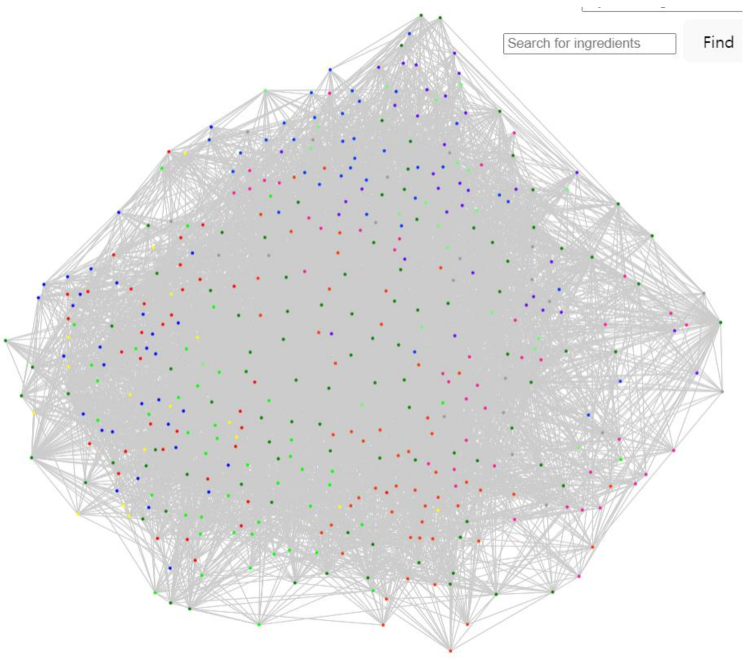


Figure 15. Subgraph of applying K-core function (K = 15)

f. Providing steps of recipe using directed graph

In our recipe information, we've implemented a feature that enables the creation of a directed graph based on recipe order and ingredient usage. Directed graph is a type of graph in which edges have a direction associated with them. In other words, the connections between nodes have a specific directionality, indicating that they go from one node (the source) to another node (the target). This directionality can be represented by arrows or directed edges. We applied this directed graph to a graph that connects ingredients according to recipe order. When users select a recipe node, nodes are created using the ingredients used in the recipe and connected according to the order in which the ingredients are used to create a directed graph.

In Figure 16, when a user clicks on a recipe node, we provide a directed graph for that recipe. Currently, each ingredient is connected in order and the edge indicates the order in which ingredient is used. The labels of edges in a directed graph indicate the order in which the ingredient is used. I initially encountered cases where parallel edges occurred when the same materials were used in different orders, resulting in overlapping edges. To solve this problem, I created parallel-Index and changed the edge type to curve if the parallel index exists between edges and changed it to straight arrow if not. Currently, when there are parallel edges, we are expressed using curved edges without arrow, but we are looking into whether a curved arrow can be created to become a directed graph.

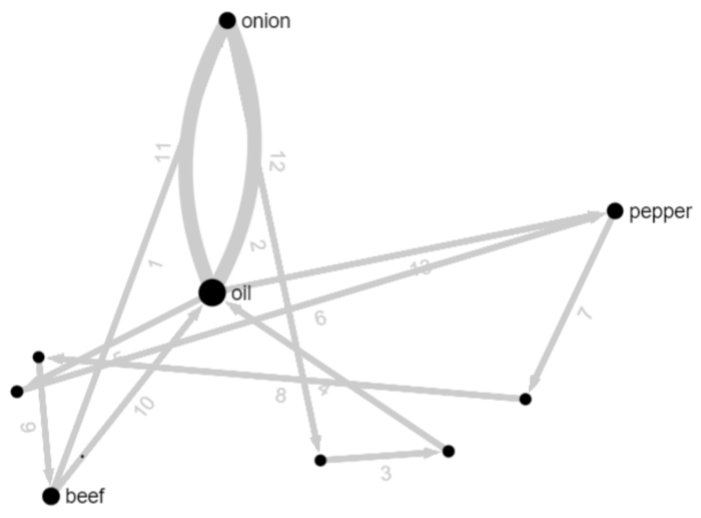


Figure 16. Screen of directed order graph for "beef ribs bbq sokalbi koo"

Another problem is that, as shown in the figure 17, ingredient nodes are formed but no edges are created. The reason why this happens is that although the ingredients are used in the recipe, the ingredients are not directly mentioned in the recipe order we brought from the dataset or are mentioned in other words, so an edge is not created, and a directed graph is not created. To solve this problem, we may analyze the context of the recipe order to infer missing ingredients. For example, if a recipe mentions “vegetables” without specifying the types of vegetables, we can infer common ingredients such as onions, carrots based on the vegetable ingredients used in this recipe. By using this strategy, we can mitigate the challenge of establishing connections between nodes in the graph when ingredients are not explicitly mentioned in the recipe order.

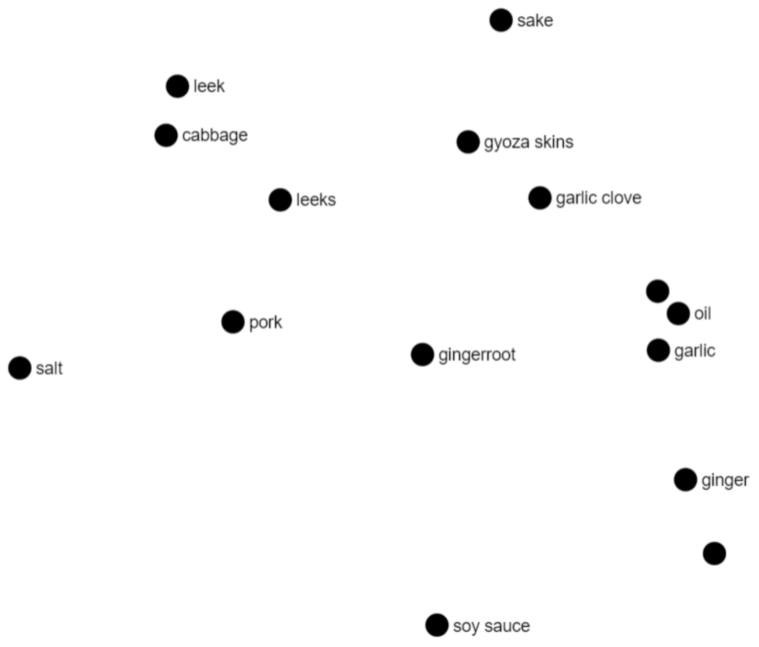


Figure 17. Recipe for cases where edges are not formed

When determining the size of nodes, random walk path algorithm was used to determine the size of each node. The random walk path algorithm is an algorithm that starts at a given node in a graph or network and, at each step, randomly chooses one of its neighboring nodes to move to. This algorithm is used to explore the structure of a graph or network by traversing it in a random manner. This algorithm leverages the structure of the directed graph to compute node size, providing users with visual cues that reflect the importance and connectivity of each node within the graph. By integrating these features, we believe that users can not only understand the ingredients and order used in a recipe immediately, but also gain better insight into which ingredients are important in the recipe.

Next, we will explain how to apply the random walk path algorithm to express the importance of ingredient nodes and explain what the advantages are. This function performs random walks on the graph and calculates the visit count for each ingredient node to determine its importance. In line 2 to 3, it initializes a dictionary called *visitCount* to store the visit count for each node and set all counts to zero. In line 4, it iterates over the specified number of walks. In line 6 to 7, for each walk, it randomly traverses the graph from a starting node and increments the visit count of each node. In line 12 to 13, we find the importance of each node, *visitCount* of each node is divided by *maxVisit*. Finally, it returns the importance of each node based on its normalized visit count. Through this function. Resizing nodes based on importance helps to visually highlight the most significant ingredient in the graph. User can quickly identify key ingredients that play a crucial role in the recipe order dataset.

Algorithm 6 Node Importance using random walk path

```
1: Function Check Ingredient Type(graph, walkLength, numWalks)
2: $visitCount \leftarrow$ dic
3: $graph.forEachNode((node) => (visitCount[node] = 0))$
4: for $i = 1$ to $numWalks$ do
5:       $graph.forEachNode((node) => \{$
6:          $walkpath = random_walk(graph, node, walkLength);$
7:          $walk.forEach((n) => visitCount[n]++);$
8:    $\})$
9: end for
10: $maxVisits = Max(Object.values(visitCount))$
11: $nodeImportance = \{\}$
12: $graph.forEachNode((node) =>$
13:    $nodeImportance[node] = visitCount[node]/maxVisits$
14: return $nodeImportance$
```

Figure 18. Random walk path function pseudo code

3. Recommendation System

In this section, we will explain the overall UI and sequentially show how the web application we developed works according to the recommendation system scenario we planned last semester.

Overall UI configuration

When user accesses our site, UI will look like the figure 19 and each function is as follows:

① From the left, camera zoom in, zoom out, and camera reset buttons

"+" Zoom In Button: This button zooms in on the graph, allowing users to inspect finer details or delve deeper into specific sections

"- "Zoom Out Button: This button zooms out the graph, providing an overview of the overall structure. Users can grasp the general shape of the graph or discern large-scale patterns.

"⊙" Camera Reset Button: This button resets the graph's camera to its initial position.

② R button : Graph reset button, F button : Ingredient filter button

"R" button: This button resets the graph to its initial state. When clicked, it restores the graph to its original layout. Users can utilize this button to revert the graph to its starting configuration, providing a clean slate for further exploration or analysis.

"F" button: This button hides all ingredient nodes in the graph when clicked. It enables users to temporarily remove ingredient node from the visualization, facilitating a clearer view of other graph elements

③ K – core function

The user enters a number into the input box. This number represents the minimum degree required for vertices to be included in the K-core subgraph. When the user clicks the "OK" button, it triggers the K-core function to execute.

④ Common, uncommon, normal ingredient filter function

This checkbox allows users to filter the graph based on the commonality of ingredients, dividing them into common, normal, uncommon categories. Users can adjust which ingredients are displayed in the graph based on their frequency of occurrence.

⑤ Cuisine filter function

Like the ingredient filter function described earlier, the cuisine filter function allows users to filter recipes based on their cuisine type. When a user selects a specific cuisine checkbox, the graph is filtered to display only recipes corresponding to that cuisine.

⑥ Top part : List of all ingredients, Bottom part : Input the ingredients with find button

In the top part of the interface, users can view a comprehensive list of all ingredients present in the graph. Users can scroll through the list to explore all the ingredients visually.
In the bottom part of the interface, users can input specific ingredients into an input field. After entering the desired ingreidents, users can click the “Find” button to search for recipes that match the entered ingredients.

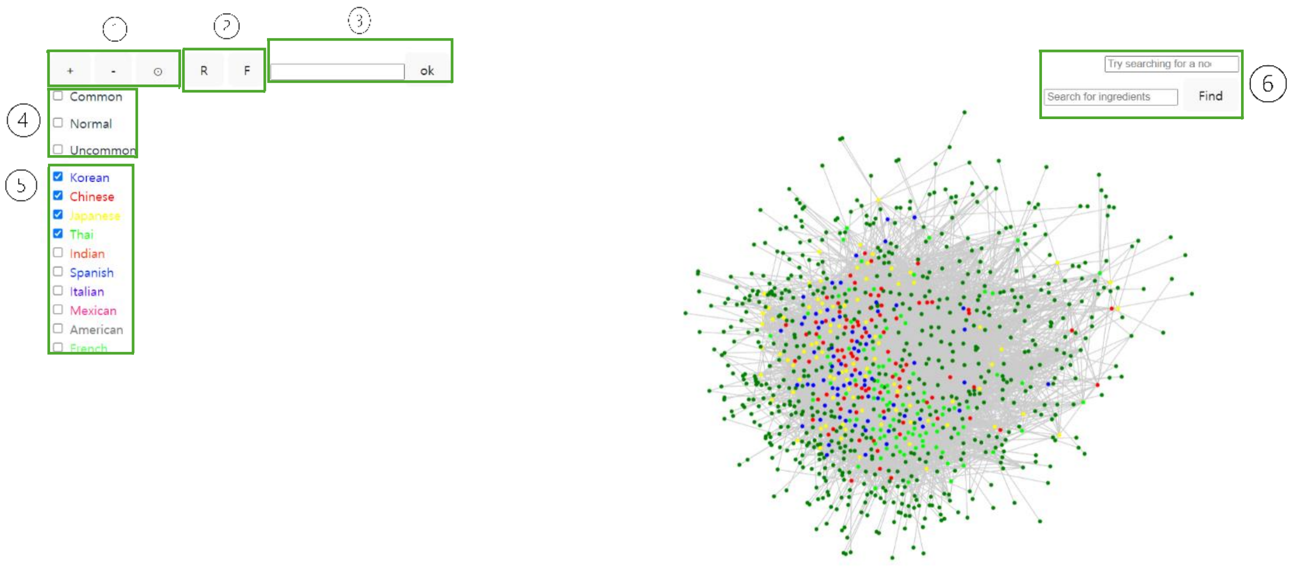


Figure 19. Overall UI of our graph web application

Recipe recommend system

Last semester, we envisioned this scenario:

1) Users select ingredients
2) N- listed recipe nodes will be matched from Recipe Information graph
3) Parse N – listed recipes to Recipe Nutrition graph via recipe name
4) Sort the n -listed recipes using rating and Health Star Rating score
5) Recommend Healthy recipe towards users

The major change we made to the scenario this semester was to first integrate the Recipe Nutrition graph into Recipe Information graph. The Recipe Nutrition graph was a graph that showed the nutritional information of the recipe, and we decided that it would be better to include the data in the recipe node rather than displaying this information through the graph. This integration resulted in the following modifications to the scenario:

1) User select ingredients
2) N- listed recipe nodes will be matched from Recipe Information graph
3) Recommend Healthy recipe towards users

In step 2, we provide a subgraph of recipes that match according to ingredients in Figure 20. When providing a subgraph consisting of recipe nodes in this step, we plan to provide different node sizes depending on the recipe's HSR score and user average rating score calculated last semester. We think this has the advantage of helping users determine which recipe is better among the matching recipes at a glance. Additionally, the edge shown indicate when a similarity score is formed between recipes, showing that similar recipes are connected to each other.

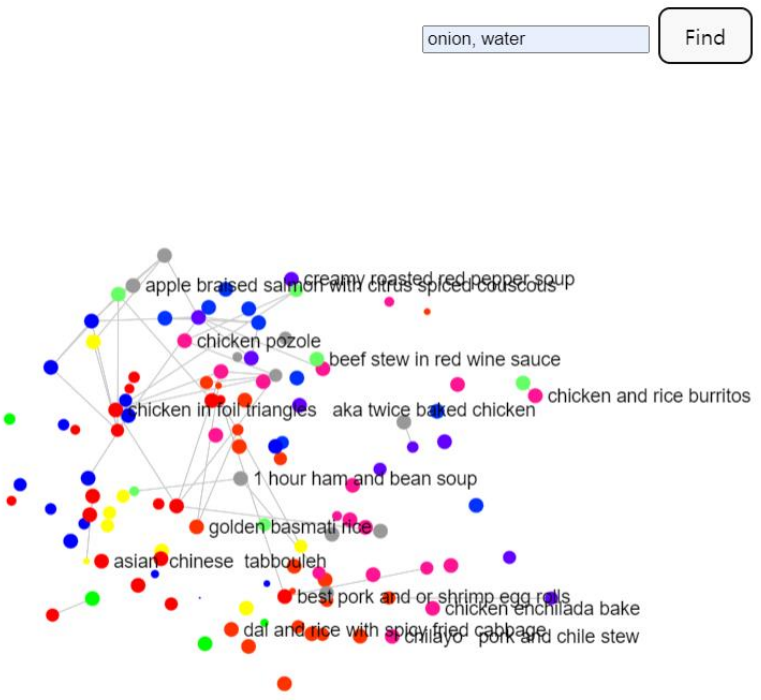
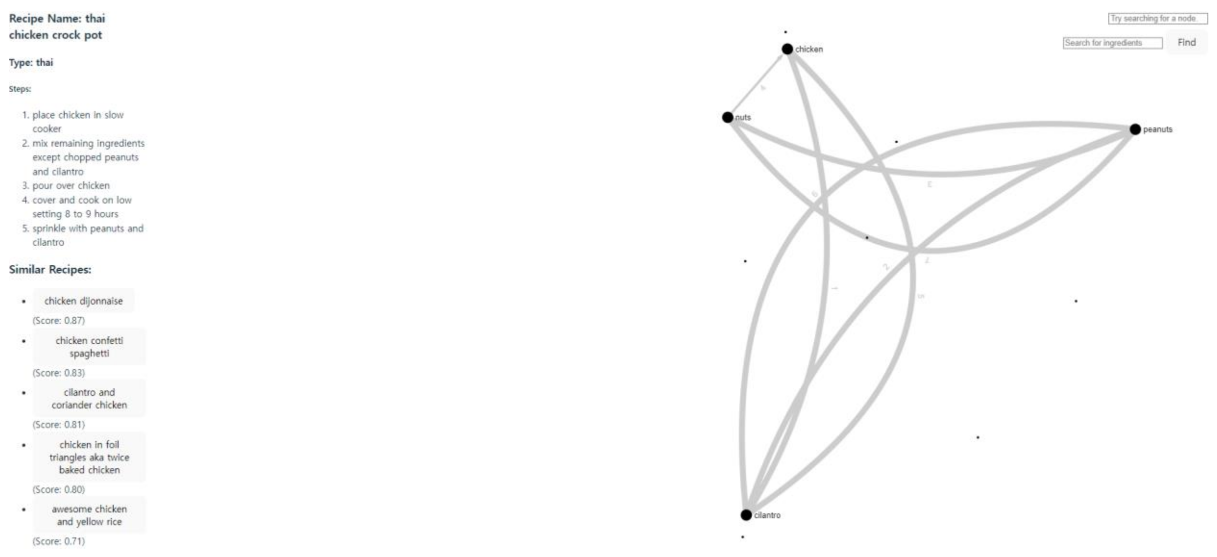

onion, water
Find
apple braised salmon with citrus spiced couscous
creamy roasted red pepper soup
chicken pozole
beef stew in red wine sauce
chicken and rice burritos
chicken in foil triangles
aka twice baked chicken
1 hour ham and bean soup
golden basmati rice
asian chinese tabbouleh
best pork and or shrimp egg rolls
chicken enchilada bake
dal and rice with spicy fried cabbage
chilayo pork and chile stew

Figure 20. Recipe nodes matching onion and water

If the user selects one of the matching recipes, a directed graph of the recipe and additional information are provided. At this time, we will provide cooking instructions for the recipe, and we will also provide other options by providing recipes similar to the recipe.


Recipe Name: thai chicken crock pot

Type: thai

Steps:

1. place chicken in slow cooker
2. mix remaining ingredients except chopped peanuts and cilantro
3. pour over chicken
4. cover and cook on low setting 8 to 9 hours
5. sprinkle with peanuts and cilantro

Similar Recipes:

- chicken dijonnaise
  (Score: 0.87)
- chicken confetti spaghetti
  (Score: 0.83)
- cilantro and coriander chicken
  (Score: 0.81)
- chicken in foil triangles aka twice baked chicken
  (Score: 0.80)
- awesome chicken and yellow rice
  (Score: 0.71)

Figure 21. UI of thai chicken crock pot recipe information

Discovery and Reflection during project

The most interesting part I found while working on this project was the process of creating a graph by combining the direction graph with ingredients and recipe steps. At this time, there were cases where the graph was created well as I designed, but there were also cases where the graph was not formed properly due to the problems I explained earlier. However, I think this directed graph is an interesting study because I think it will be very useful for users to understand recipe steps at a glance.

The most difficult part to solve was the graph layout part. The graph displayed on the screen appears overly dense, making it challenging to discern individual nodes and their connections. The abundance of nodes and edges creates a cluttered visual representation, hindering clarity and making it difficult for users to interpret the underlying structure of the graph. As a result, navigating and extracting meaningful insights from the graph becomes cumbersome and less intuitive. When implementing graph functions, I usually experimented with 100 sample datasets, so I didn't think it would be a big problem when the graph layout was expanded to 1000. I discovered this problem late and tried to solve it, but *Sigma.js* has a limit on Canvas size, so I have not been able to solve this problem yet.

4. Conclusion

In conclusion, this semester’s endeavors have been marked by significant progress and innovation in our project. Building upon the foundation laid in the previous semester, we successfully addressed the cosine similarity memory issue through a novel approach. Moreover, we expanded the scope of our recipe information graph by incorporating an additional 1000 recipes, thereby enriching its depth and breadth. We implemented functions based on a user-centered approach beyond simple graph analysis functions by integrating the recipe information graph into a web application and utilizing Sigma.js to provide users with intuitive and diverse functions.

During building graph in web application, the calling stack size error that occurred when expanding the graph was resolved with 1000 recipe data, but I think it is a potential problem that may occur again if the data is further expanded in the future. Additionally, the inability to provide a perfect directed order graph due to the inaccuracy of order data remains to be improved. In the future, we might use NLP (Natural Language Processing) technology to analyze recipe order text and identify keywords related to ingredients. This allows us to more accurately extract the ingredients used in recipes.

Through this semester, we successfully implemented the recipe recommendation system scenario we created, and implemented functions that can provide various information to users and functions that allows users to check only certain types of recipes, making it easier for users to check. We’ve tried to provide a simpler layout. If I can do this project further in the future, I’d like to expand the data from 1000 to 10000 and even solve the graph layout problem that I was unable to solve this semester. So far, we think that to solve the problem of graph layout, rather than using Sigma.js. we should investigate other tools and try using tools that have many functions for large graph layout. Therefore, next time, we would like to implement a more complete and systematic graph layout and various graph functions that are comparable to the users.

References

1. Our Recipe recommendation product - https://github.com/F-los/FYP
2. 9 Best JavaScript techniques for Network Graph Visualization – https://rapidops.medium.com/9-best-javascript-techniques-for-network-graph-visualization-f5f377419b1a
3. 7 Helpful Sigma.js Examples to Master Graph Visualization https://rapidops.medium.com/7-helpful-sigma-js-examples-to-master-graph-visualization-a8cadf9e9b14
4. graph-based forensic analysis of web honeypot https://www.researchgate.net/publication/305374359_Graph-based_forensic_analysis_of_web_honeypot

5. Jacomy M, Venturini T, Heymann S, Bastian M. ForceAtlas2, a continuous graph layout algorithm for handy network visualization designed for the Gephi software. PLoS One. 2014 Jun 10;9(6):e98679. doi: 10.1371/journal.pone.0098679. PMID: 24914678; PMCID: PMC4051631.
6. Graphology library - https://graphology.github.io/
7. Sigma.js - https://www.sigmajs.org/
8. Random path walk algorithm - https://neo4j.com/docs/graph-data-science/current/algorithms/random-walk/