Smart Meal Plan Generator:

IMeal

-- A CSP Project for CSC384

***Team Members***:

1. **Shuo Feng (fengshuo)**

Major responsibilities: problem encoding, experimental assessment, heuristic designs

Minor responsibilities: manuscript writing

1. **Yihui Li (liyihui)**

Major responsibilities: standard database building, file I/O method implementation, manuscript writing

Minor responsibilities: problem encoding, experimental assessment

# Project Motivation and Background

The application that we are trying to build, a meal plan generator is tasked with taking the input of a food nutrition database and generate a satisfying meal plan given the user input of specific requirements. This application, along with a comprehensive and accurate food database, will help users make informed, and fast decisions about what to eat.

The obvious approach in the situation would be designing a CSP. Users of this application will have different requirements, including the number of meals, amount of nutrition, variety and etc. CSP is the best approach that we have learned so far in class that handles these dynamic requests.

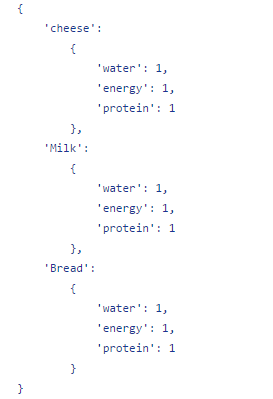
# Methods

The following steps are taken to implement the application.

1. Building a standard database

As described in the project proposal, we built upon an existing food database from (<https://www.ars.usda.gov/northeast-area/beltsville-md/beltsville-human-nutrition-research-center/nutrient-data-laboratory/docs/usda-national-nutrient-database-for-standard-reference/>), which included a collection of food and their corresponding nutrition values. The only additional parts of building the database were (1) to add a price tag to each type of food (This was not initially implemented. Will discuss more in the obstacles section) (2) to add a “time” tag which indicated whether the food was for breakfast, lunch or dinner. Since retail prices for specific food could be hard to find, for the purpose of this project, we took a simplistic approach and randomized a price under $10 for each food. We reduced the number of nutrients to ##, for two reasons. (1) People tend to care about only a few nutrients. (2) More nutrients will produce more input queries, which defeats the purpose that we want the application to generate fast and simple results. Also, for certain memory reason, which will be discussed in later sections, the large initial database was reduced to ## items.

1. Extracting data from database

The original database was in xlsx format. The desired output format was a dictionary of dictionaries of food items. Each food item has its name as the key of the outer dictionary, and each of its nutrient value as the value of the inner dictionary, with nutrient name as the corresponding key. E.g. 

The reason for picking this data structure is to preserve relationships among food names, nutrient names and nutrient values as well as allow easy access when creating variables. To make the conversion, the xlsx file is firstly converted to csv and then python read the csv file line by line using the built-in csv library.

1. Encoding variables, constraints and the CSP

Each variable represented a meal. Its domain included all food items in the database with the correct “time” label. This means, if a food had a “breakfast” label in the database, then it should only appear in breakfast variables’ domain.

There will be 5 different constraints placed over budget and the amount of nutrients that the user wants to consume on a per day basis – protein constraint, sugar constraint, calcium constraint, energy constraint and budget constraint. Satisfying tuples are extracted and added to each constraint, and that very constraint was then added to the CSP.

1. Developing ordering methods

In order to speed up the search process when going through variables, several ordering methods were implemented and results were compared. These ordering methods included the ordinary mrv, dh, lcv and another val\_ordering\_max that sorted food variables based on their total nutrient amount.

# Evaluation and Results

The following criteria were examined and evaluated.

1. Amount of time used to search for each ordering method
2. Number of values assigned for each ordering method
3. Maximum varieties achievable before incurring memory error when restricting number of food vs not restricting.

# Limitations and Obstacles

The biggest obstacles was memory space management. Given that the method involved generating all possible tuples using power sets, that each meal variable contained several food items, and that there are a collection of food items available in the database, the number of possible tuples increased exponentially as the number of meals we wish to generate. This problem limited the number of database items significantly, especially when the “time” tag was not added, as every food item will appear in every meal’s domain and increased the size of its collection of possible tuples. After the “time” tag was implemented, the results got better, as we were able to add more items, but the varieties were still less than ideal.

# Conclusions and Next Steps

First of all, we learned how to apply CSP concepts to work on a very open-ended project. Secondly, it was also a reality check that the methods we learned in class, or developed in the assignments, namely the plain backtracking propagator and the generate-all-possible-tuples approach were still rudimentary and subject to adjustments when applied on more complicated problems.

If we were to work on a similar CSP model in the future, we would definitely spend more time trying to overcome the biggest obstacles that we found by exploring methods that do not generate, or generate fewer possible tuples. We would also like to modify the propagator function to include forward checking or GAC.