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
2. **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 (2) to add a “type” 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 21, 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 less than 20 items.

2. 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. {‘cheese’:{‘water’:1, ‘energy’:1, ‘protein’:1}. ‘Milk’:{‘water’:1, ‘energy’:1, ‘protein’:1}, ‘Bread’:{‘water’:1, ‘energy’:1, ‘protein’:1}}

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.

3. Encoding variables, constraints and the CSP

Each variable represented a meal. We encoded variable domains in two different ways. (1) The variable domain included everything from the database. (2) Its domain included all food items in the database with the correct “type” label. This means, if a food had a “breakfast” label in the database, then it should only appear in breakfast variables’ domain. These two encodings were set up this way to investigate memory space issues as described in later sections.

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.

4. Developing ordering methods

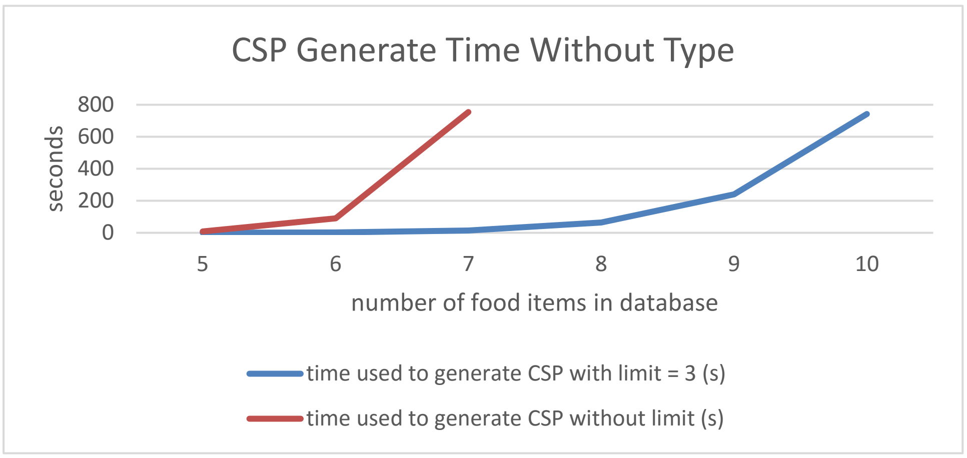
In order to speed up the search process when going through variables, and to increase variety across meals, we used and compared two value ordering methods – val_arbitrary and val_ordering_max.

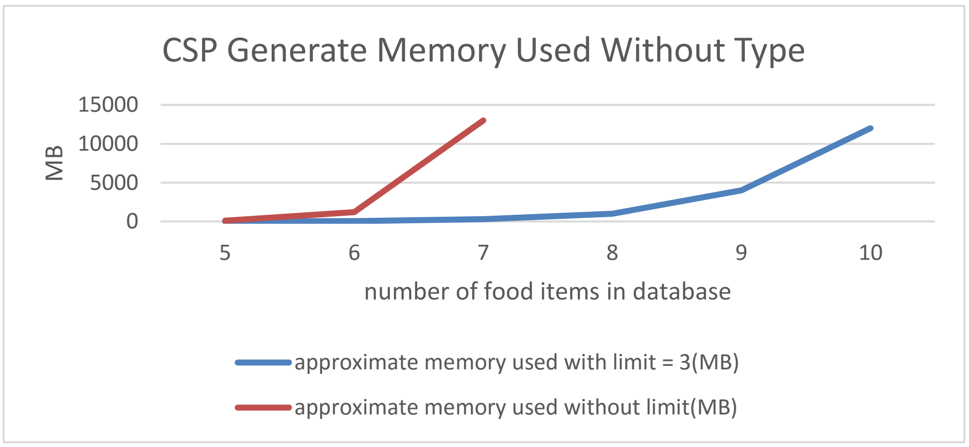
Evaluation and Results

First we evaluated the space and time required to generate the CSP model. Recall that there were two different ways that we set up variables while working (the source code used the second method). This was because we noticed a significant memory space challenge following the first option, in which each variable would have all database foot items in its domain. The challenge was more severe when the user wanted to "eat as much as possible". We concluded that this was due to an exponential explosion when generating all possible tuples. We tried to alleviate this issue by introducing the second method, which added a "type" tag to the database to reduce the number of tuples generated.

Below are the table and graphs comparing time and memory with no "type" tag. An exponential increase is evident in both cases, and limiting the number of food for each meal significantly decreased the size of power set.

number of items in db	time used to generate CSP with limit = 3 (s)	approximate memory used with limit = 3(MB)	time used to generate CSP without limit (s)	approximate memory used without limit(MB)
5	0.34	10	8.44	100
6	2.35	40	89.42	1200
7	13.76	280	754.28	13000
8	63.3	1000	NA	Memory Error
9	240.14	4000	NA	Memory Error
10	742.23	12000	NA	Memory Error





Then we added the “type” tag and ran the same tests. We were able to increase the max amount of food items in database from 10 to more than 30 before memory error occurred, while keeping a low search time. The table below shows a quick comparison

Number of items in db	Generate CSP time with limit without type	Generate CSP time with limit with type
8	63.3s	0.029s
9	240.14s	0.034s
10	742.73	0.071s
28	NA	26.74s
30	NA	35.49s
35	NA	111.65s

Apparently, adding more restriction to variable domains would help alleviate space issues.

Lastly, we compared efficiencies of different value ordering methods – val_arbitrary and the custom val_ordering_max. The performance in terms of search speed was comparable in our model, and both could finish search very fast. This was because for this question, space complexity was much bigger than run time complexity, so it was difficult to notice the performance difference between the two ordering methods within the memory limit. However, one big non-performance related difference was that val_ordering_max took the special requests into account and was able to produce variety across different meals. The val_arbitrary ordering would produce the same meal every time.

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 were 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 “type” 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 to make search more efficient.