

PSET 4
PROJECT REPORT
CAPTAN BREAKFAST (THE FOOD SOLDIER)

1. Introduction

This paper introduces a grounded scenario for the use of Fuzzy CSP techniques considered in PSET 3, a working system that makes use of these reasoning techniques, and an evaluation on the performance of the system and techniques on the given scenario.

2. Scenario and Intelligent Behavior

2.1 Scenario

Captain Breakfast is our robot that works for a busy (or lazy) master. Because the master does not have time or does not want to take care of breakfast, the robot will do that for him/her. Captain Breakfast's job is to keep food items in the fridge, keep a check on the supplies and take them out when the master orders it to do so. It is smart enough to deduce master's breakfast preferences based on master's binary preferences for food items in the fridge.

Captain Breakfast also learns about the consumption patterns for food items and based on that, it presents choices to the master about type of food item he or she wants (if there are more types of the same items). Over time, Captain Breakfast would have mapped the preferences of food items individually and in pairs. So, when the owner asks it to take out eggs for breakfast for example, it would know the bread is more preferred over doughnuts and depending on the preference value, it would take bread out along with eggs. Based on the food availability, master's preferences, or both, the robot recommends breakfast options for the day. If the breakfast contains items that are low on supply (availability) or over, it will warn the master, so that (maybe) the master can go to a store and replenish the food supplies.

In the morning, as soon as master walks into the kitchen, the robot will greet him and offer a breakfast suggestion for the day. The master can agree to have this breakfast, or he can object to the robot's suggestion. The master might not feel like having eggs today, or he wants to eat his favorite breakfast over and over again. Given the master's new conditions, Captain Breakfast is able to update the breakfast suggestion. Overall, the robot is able to have a continuous interaction with the master, in which the robots suggests breakfast options, and the master objects and adds new conditions. Ultimately, the interaction will lead to a point, where the robot will suggest a breakfast that the master is fine with having for that day.

2.2 Inputs to the system

The system takes 3 text files as inputs:

- *breakfast_log*: this file lists the breakfasts the master has had over a number of days. Each breakfast has a main item, compliment item, and drinks item
- *time_to_last*: this file includes the time to last for each item, in terms of number of servings

- *preferences*: this file contains master's binary preferences for items. Pair of items from any category are mapped to a preference value between 0 and 1

The figure shows three Notepad windows representing input files for a system. The first window, 'breakfast_log', shows a table with three columns: 'main', 'complement', and 'drinks'. The second window, 'time_to_last', shows a list of items and their corresponding serving counts. The third window, 'preferences', shows a table with four columns: 'main', 'complement', 'drinks', and 'preference'.

main	complement	drinks
eggs	bread	milk
bread	jam	juice
oatmeal	apple	milk
eggs	bread	coffee
bread	jam	coffee
eggs	ham	coffee
bread	ham	juice
pancakes	syrup	milk
oatmeal	sugar	milk

item	: servings
eggs	: 4
bread	: 5
milk	: 8
jam	: 15
juice	: 8
oatmeal	: 6
apple	: 1
coffee	: 10
ham	: 7
pancakes	: 10
syrup	: 16
sugar	: 17

main	complement	drinks	preference
eggs	bread		0.5
eggs		milk	0.6
	bread	milk	0.4
bread	jam		0.1
bread		juice	0.2
	jam	juice	0.6
oatmeal	apple		0.4
oatmeal		milk	0.3
	apple	milk	0.7
eggs		coffee	0.6
	bread	coffee	0.4
bread		coffee	0.5
	jam	coffee	0.6
eggs	ham		0.2
eggs		coffee	0.5
	ham	coffee	0.5
bread	ham		0.9
	ham	juice	0.6
pancakes	syrup		0.8
pancakes		milk	0.8
	syrup	milk	0.4
oatmeal	sugar		0.6
	sugar	milk	0.5

Fig: Representation of the input files for the system

2.3 Demonstration

In this section, we'll demonstrate some cases where the system demonstrates intelligent behavior. We start by showing how the system's interface looks like:

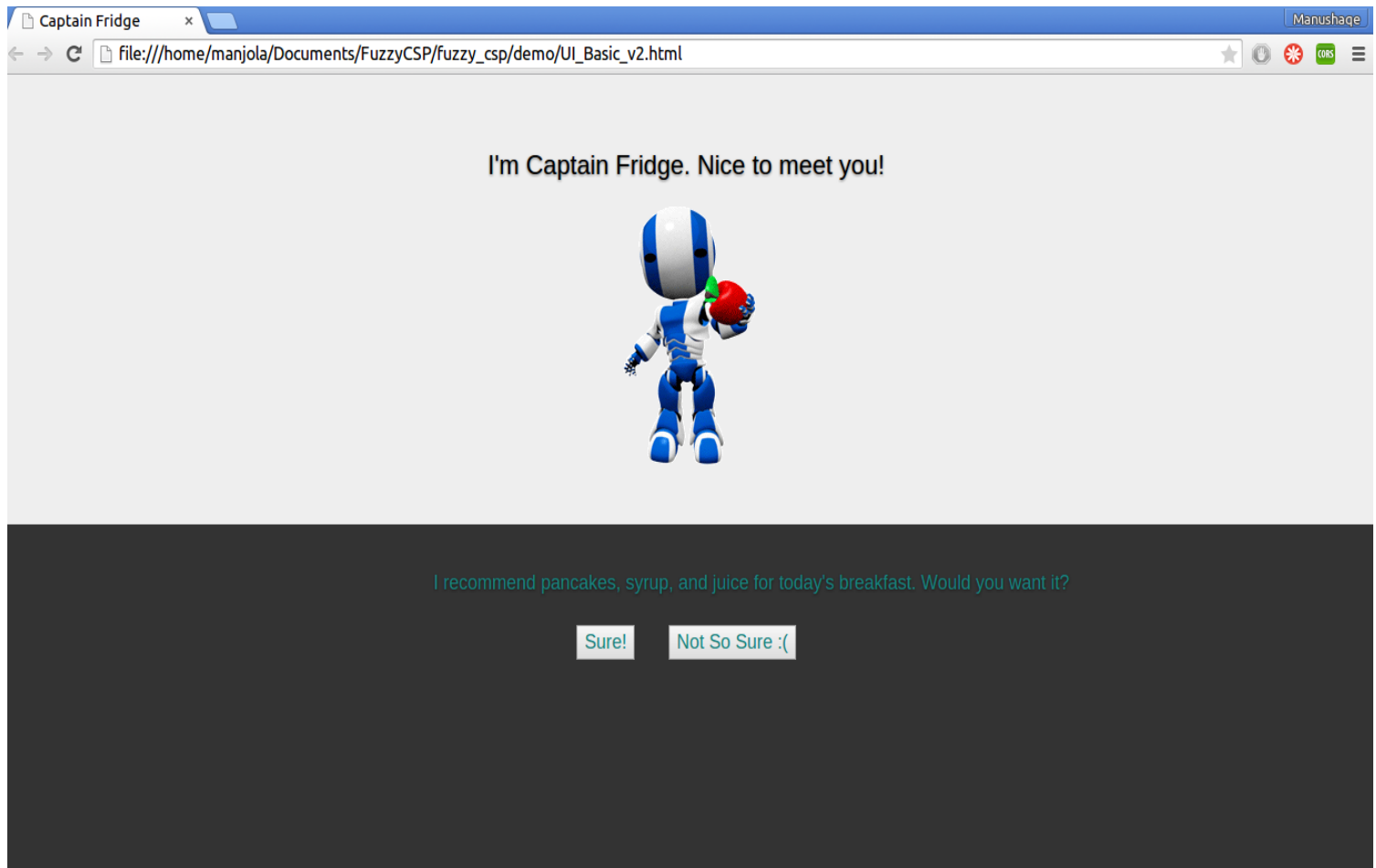



Fig: A first look at the system's interface

The first breakfast suggestion of Captain Breakfast is a combined solution, taking into account the availability of the food items in the fridge, and master's preference. If the master wants to have this breakfast, he/she can click on the button "Sure!". If the master wants other options, he will click on the button "Not So Sure :(". The interface will change, giving the master more options:

I'm Captain Fridge. Nice to meet you!



Rejected:
["pancakes","syrup","juice"]
["pancakes","sugar","juice"]

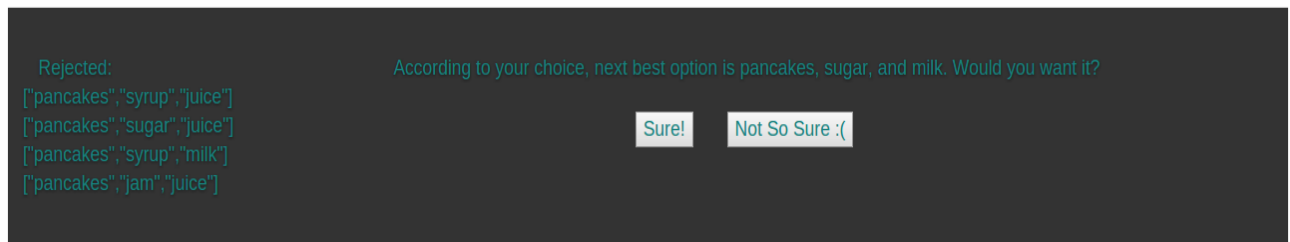
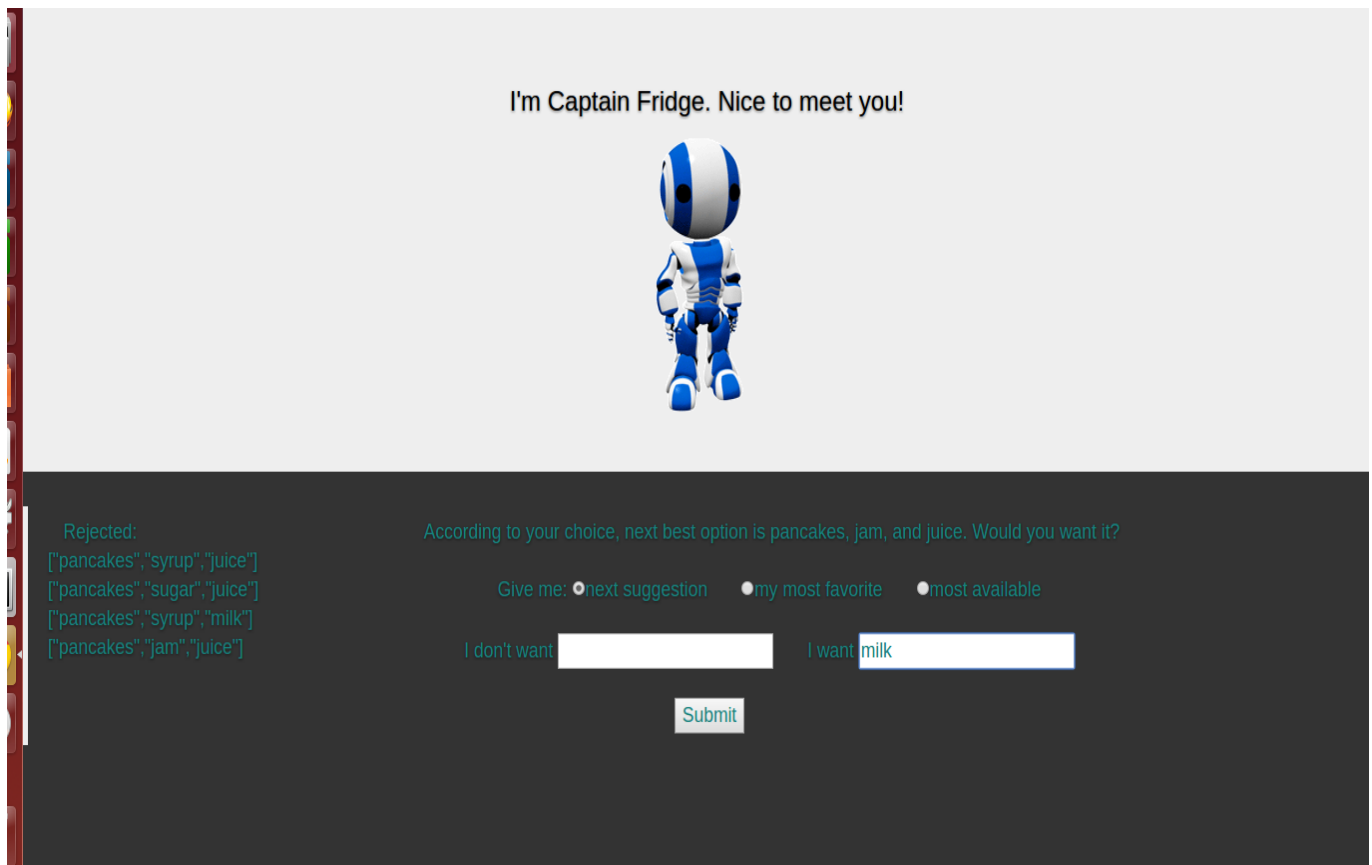
According to your choice, next best option is pancakes, sugar, and juice. Would you want it?

Give me: ☒ next suggestion ☐ my most favorite ☐ most available

I don't want I want

Fig: More options for the user

Next suggestion will give the next best combined breakfast. *My most favorite* will give the most favorite breakfasts for master. *Most available* will give the most available breakfast choices. The master can also give text input for an item he doesn't want: *I don't want* _____, and/or for an item he wants: *I want* _____. The robot can handle any combination between the first set (next suggestion, my most favorite, most available) and the text inputs. After the master inputs the new conditions, he can *Submit* his choice. The robot will make another breakfast suggestion. The reject breakfasts are listed on the left side of the screen. The following screenshots cover some of these possible combinations.



**Fig: The master wants the next suggestion that contains milk.
The second screenshot shows the new suggested breakfast, which contains milk.**

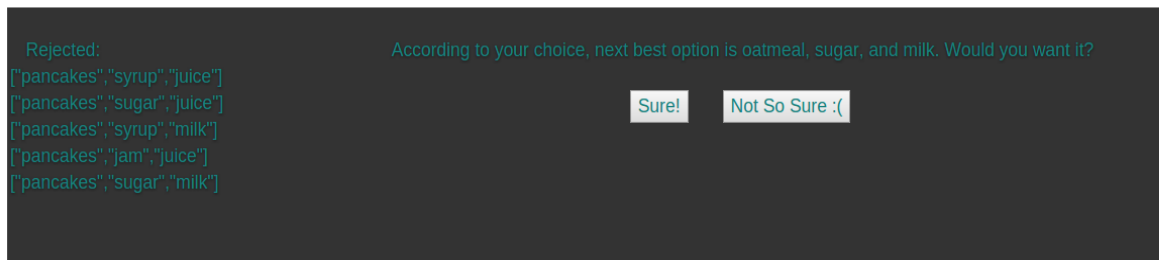
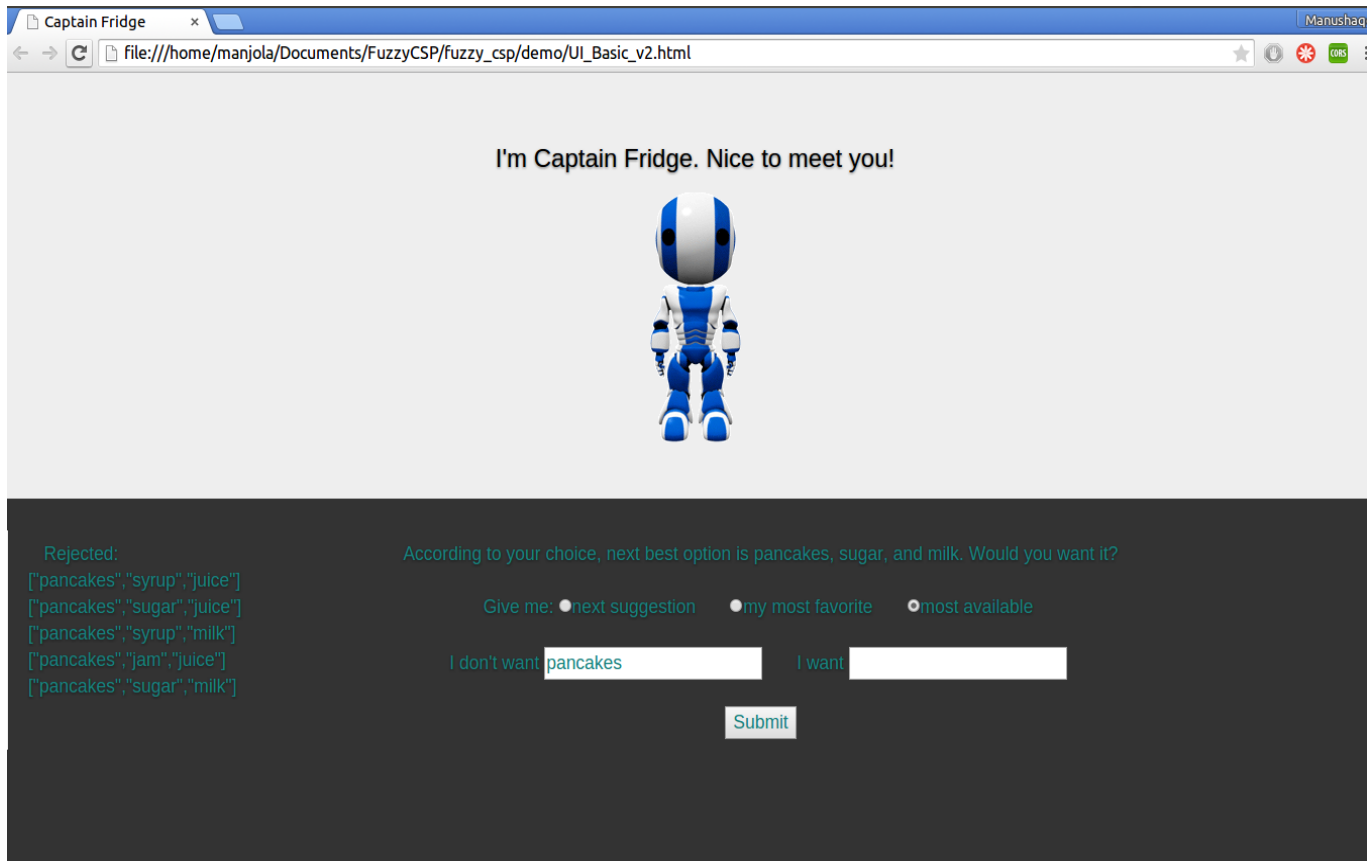


Fig: The master wants the most available breakfast that does not contain pancakes. The second screenshot shows the new breakfast, with oatmeal instead of pancakes.

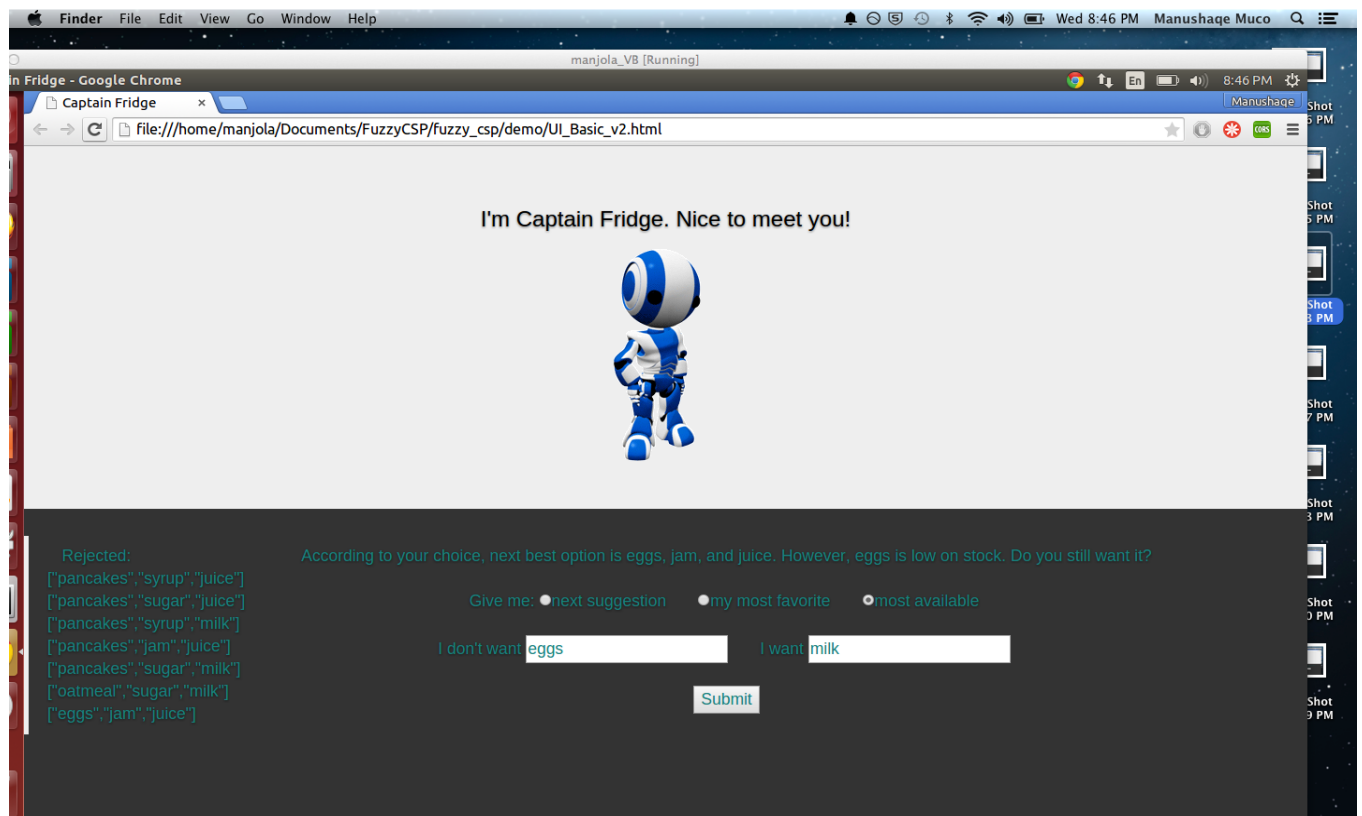


Fig: The robot has warned the master that the current breakfast, contains an item low in availability (eggs is low on stock). The master, now, wants the most available suggestion that does not contain eggs, and that contains milk.

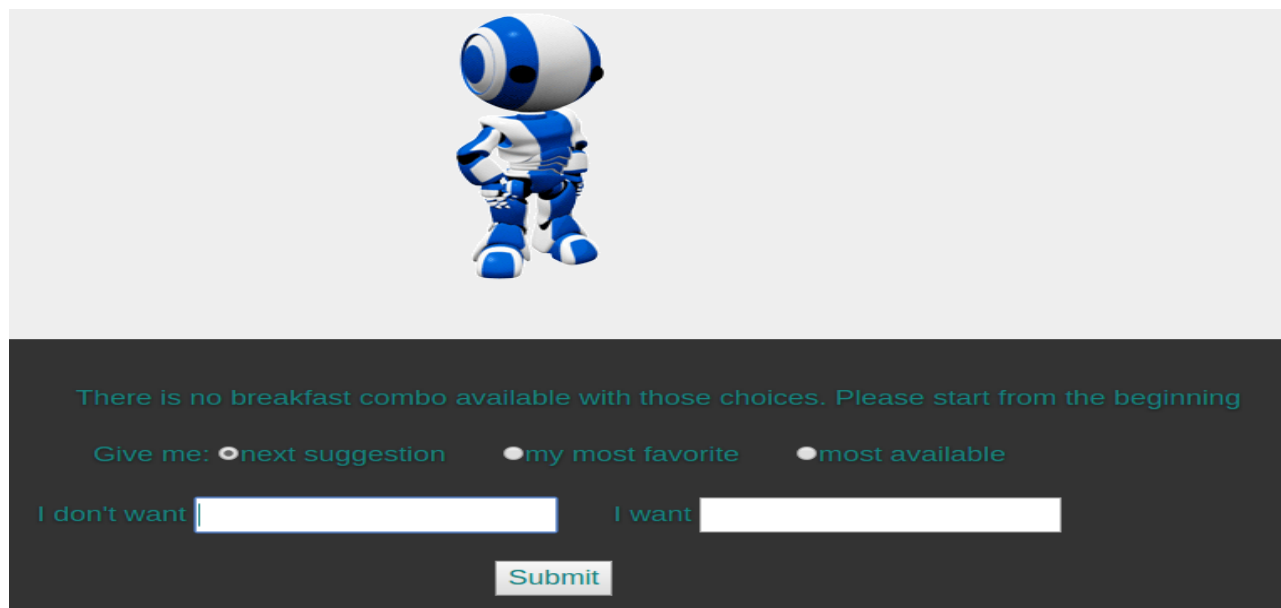



Fig: The robot cannot find a breakfast suggestion with the conditions the master gave. The robot asks the master to start over.



Rejected: ["pancakes", "syrup", "milk"] According to your choice, next best option is pancakes, syrup, and milk. Would you want it?

Give me: ☒ next suggestion ☐ my most favorite ☐ most available

I don't want I want bread

Rejected: ["pancakes", "syrup", "milk"] According to your choice, next best option is bread, ham, and coffee. However, bread is out of stock. Please choose another option.

Give me: ☐ next suggestion ☒ my most favorite ☐ most available

I don't want I want

Fig: The master wants one of his most favorite breakfasts that contains bread. The robot warns the master that bread is out of stock, and that the master needs to choose another option.

3. Implementation

In this section, we discuss the implementation of the grounded scenario, extensions from PS3, and some creative approaches taken in design and implementation.

3.1 System architecture

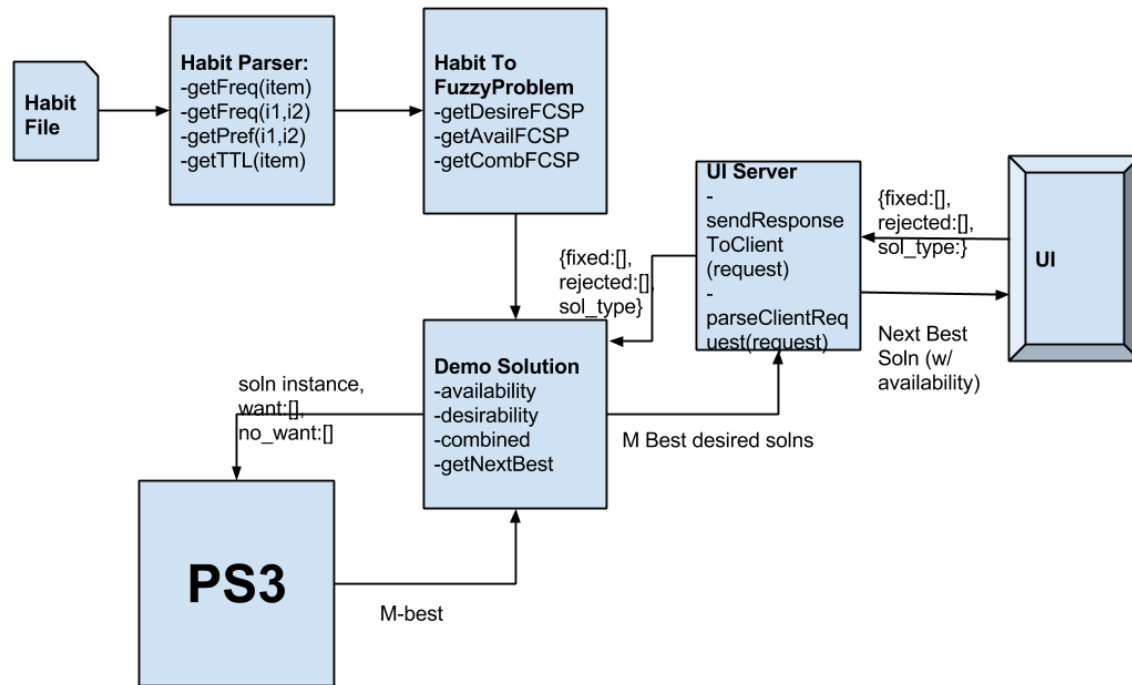


Fig: System diagram for the grounded scenario

The figure above shows various modules working together to produce the entire system. To describe briefly, the “UI” module takes the input on what kind of breakfast the user desires, e.g. “most available”, “most preferred”, or “combined”. Using AJAX, we pass the inputs to “UI server”, created with a client-server architecture provided by Python libraries/modules called “Flask” and “Flask-RESTful”. The “UI Server” parses the user input, and sends the information about the solutions desired to the module “Demo Solution”. At the same time, various kinds of input files, collectively denoted by “Habit file” above, are parsed by “Habit Parser” module, and converted into Fuzzy Constraint Satisfaction Problems by “HabitToFuzzyProblem” module. Now that “Demo Solution” has the Fuzzy CSP and the types of solutions to return, it passes the information to PS3 architecture, which returns M best solutions to the “Demo Solution”. It passes the next best breakfast combo among the M best, disregarding the previously rejected solutions to the “UI Server”, and then eventually to the “UI”.

Now we go one by one to each module and describe their functionality, inputs, and outputs in more detail.

3.1.1 User Interface

The User Interface we developed for the demonstration of our FCSP technique presents to the user several types of preference choices to select the breakfast:

1. Default option

The as the user interface is loaded, the interface makes a connection with the server to determine the best combination of main item, complimentary item and drinks according to the best joint satisfaction degree for preference as well as availability. If this option is rejected by the user, the user interface sends a new request to the server to get a new list of options that includes the new constraints that are added by the user include - wanted items, unwanted items, best option based on preferences alone and best option based on availability alone. We will discuss about each of these additional choices that the user can request from the robot in the list below.

2. Most favourite selection of items (main dish, complimentary dish and drink)

The most favourite set of items are learned by the robot based on master's preference of food items in binary pairs. In our demonstration, we use a predefined list of binary preferences of our three variables. The user can select this option using the radio button on the web interface.

3. Most available set of items

The user can select this option using the radio button on the web interface to ask for the most available combination of items. We use the file `breakfast_log` as explained in section 2.2 to record and maintain the pattern of consumption of items to calculate the availability of the individual items.

4. Next (combined) preference

The user can select this option to get the combined best solution with joint satisfaction of preference and availability for the set of items. We will cover the approach we have taken to combine the preferences based on availability and desirability in section 3.3.2 and 3.3.3.

5. Include specific item

This option is a text based input on the web interface to allow the user to select only the solutions that contain a specific item. However, it is not always guaranteed that the user will get a feasible solution with the item if the availability of the item is zero. In the case of low availability, the user is warned by the user interface about the decreasing availability of the item.

6. Exclude specific item

Similar to the include option above, the user can input via text what he/she doesn't want to appear in the solution returned by the server.

The UI engine is implemented using javascript. The main functions that our user interface implements to incorporate the inputs from the user for communicating the user commands and receiving the results from our algorithm are:

- `getNextPreference(send_data)` : This function creates a set of arguments containing wanted items, unwanted items and type of solution (availability, desirability and combined) requested by the user.
- `get_string_response(json_response, is_init)`: This function receives the response from the UI server described in the next section. The function parses the response from the server to look for solutions in response to a wanted or unwanted item. The server responds with a value of 0, 1 or 2 for each solution which indicate the level of stock available for each item. The value of 0 indicated that the item is no longer available in this case, the UI prompts the user to make another selection. The value of 1 indicates a low availability, the UI warns the user of this situation, but allows the user to go for this choice if the user chooses. The value of 2 maps to high availability and does not elicit any extra action on part of the UI.

3.1.2 UI Server

This module parses the input from the UI, and responds with the next best solution with the availability of each of the items. The availability returned is {0, 1, 2} corresponding to {"finished", "low", "ok"}.

The input coming from the client contains following: "sol_type", "want", "no_want", and "rejected_sols". After those variables are parsed, the following function is called: `demo_sol.get_mbest_fix_solutions(sol_type, want_items, no_want_items)`. The description of this function is in section 3.1.6 below, but it gives a list of m best solutions arranged in ascending order of satisfaction degrees.

These are the main functions in the UI Server:

1. `get_next_solution(candidate_solns, attempt_num, rejected_sols)`

This is a recursive function that calls itself by increasing the "attempt_num" until it finds the best solution that is not one of the rejected solutions. It returns the solution.

2. `send_response(sol)` :

This function takes a solution instance, e.g. ("bread", "jam", "juice"), and sends a response to the client with the availability of each of the item. An example response string is:

```
"{"sol":["bread","jam","milk"], "stock":[0,2,1]}"
```

3.1.3 Habit File

The Habit File module consists of the three input files described in section 2.2

3.1.4 Habit Parser

The Habit Parser module takes the three text files mentioned in section 2.2 (breakfast_log, time_to_last, preferences) as inputs, and by reading and parsing these files it populates the following fields:

- **domains:** domains is initiated as a list, whose sublists are the domains of the FCSP. The module populates this list, by parsing the breakfast_log file, and creating a sublist for each category of item. In our grounded scenario, the domains will look like:

```
domains = [ [domain for main items], [domain for complement items], [domain for drinks items]]  
= [['eggs', 'bread_m', 'oatmeal' ...] , ['bread_c', 'jam', 'apple', ... ] , ['milk', 'juice' ...]
```

So that the domains are different, for items that are used in both main and complement category, we add ‘_m’ for main category, ‘_c’ for complement category to differentiate. Such an example is ‘bread’.

- **breakfast_file_toList:** The module will read the breakfast_log file, and populate this list. Each line (starting from 3rd line) in breakfast_log represent a breakfast the master had. Each line will be converted to a sublist [main item, complement item, drinks item]. In our grounded scenario, the list will look like:

```
breakfast_file_toList= [ ['eggs', 'bread', 'milk'], ['bread', 'jam', 'juice'] ... ]
```

- **time_to_last_dict:** The module will read the time_to_last file, and populate this dictionary. The keys will be the items, and the value will be the number of servings for the item. One possible pair key:value is ‘eggs’: 4.
- **preferences_dict:** The module will read the preferences file, and populate this dictionary. The keys will be a tuple of items (main, complement, drinks), and the value will be the preference value. Because the preferences file contains binary preferences, the third item will be coded as an empty string. For example:
 - The binary preference for eggs as the main item, bread as a complement item, will be added to the dictionary like the following key-value pair:
(‘eggs’, ‘bread’, ‘’): 0.5
 - The binary preference for eggs as main, milk as drinks, will be added as:
(‘eggs’, ‘’, ‘milk’): 0.6

The Habit Parser module provides the following functionalities:

- `get_variables()`: returns the variables of the FCSP under consideration, as a list. For our grounded scenario it returns a list `["main", "complement", "drinks"]`.
- `get_domain(variable)`: returns the domain for the variable `variable`, as a list. The possible variables in our scenario are "main", "complement", "drinks". For example, if we want `get_domain("main")`, the function will return the domain by getting `domains[0]`
- `get_domains()`: returns all the domains of the FCSP. Each domain is a sublist of the list. In fact, this function returns `domains`.
- `get_all_items()`: returns a list of all the elements of each domain. In our scenario, it will return as list `['eggs', 'bread_m', ... 'bread_c', ... , 'milk']`
- `get_time_to_last_item(item)`: returns the time to last for item `item`, as an integer number of servings. The function will return the value for the key `item` in the `time_to_last_dict`.
- `get_frequency(item_list)`: returns the frequency of use for the items in the `item_list`. `item_list` can be a single item list `[item]`, a list with two items `[item1, item2]`, and so on. The function does this by counting how often the item(s) in the `item_list` appear in `breakfast_file_toList`.

If `item_list` is a single item list, the function will return how many times this item has been had by the master, i.e, how often this item appears in `breakfast_log`. For example, `get_frequency(['eggs'])` will return 4.

If `item_list` has two elements, the function returns how many times this items had been eaten by master together, i.e, how often the pair of items appears in `breakfast_log`. For example, `get_frequency(['eggs', 'bread'])` will return 1.

- `get_preferences(items)`: returns the master's preference for the food items in the `items` list. In our scenario, we only have binary preferences, so the third item will have to be given as an empty string. To see how the function works, let's consider `get_preferences(['eggs', 'bread', ''])`. The function will return the value of the key `('eggs', 'bread', '')` in the `preferences_dict`.

3.1.5 HabitToFuzzyProblem

This module converts the parsed files into three different Fuzzy Constraint Satisfaction Problems (FCSPs) -- availability FCSP, desirability FCSP, and combined FCSP.

To recap briefly, A FCSP contains variables, their domains, and the constraints between the variables. The variables and domains in our grounded scenario are the same for all three kinds of FCSPs, but the constraint satisfaction scores between various variables are different. For our scenario, we are considering the binary constraints, i.e. each constraint is a function of 2 variables. Eg. there is one constraint between “main” and “drinks”. However, there is a fuzzy set of satisfaction scores for different values that these variables take, e.g. let’s say “main” = “eggs” and “drinks” = “milk” has a score of 0.6, while “main” = “bread” and “drinks” = “juice” has a score of 0.2.

The main functions it provides are:

1. `get_availability_fcsp()`

For availability FCSP, we create binary constraints with scores derived from the availability of each items. If any two items are more available, they have higher scores in the constraints. I describe how we can come up with a score for each pair of items in next section(“creative approaches/design decisions”).

2. `get_desirability_fcsp()`

Constraints for desirability fcsp directly come from the HabitParser’s `get_preference(items)` function. We get the constraints for each pair of items if the Master provided it.

3. `get_combined_fcsp()`

Constraint scores for combined FCSP come from a combination of constraints from availability and desirability constraint scores. The combination formula we used is described in the next section.

3.1.6 Demo Solution

This module is the wrapper around PS3’s FuzzyCSSolution class. Its main task is to receive the user input from the UI Server, call a function in a particular instance (availability, desirability, or combined) of PS3’s FuzzyCSSolution class, and return the M best solutions according to the input, back to the UI Server. There is just one main function in this class:

- `get_mbest_fix_solutions(sol_type, want_items, no_want_items)`

The parameter “sol_type” is a string {“availability”, “desirability”, “combined”}. According to that parameter, the function decides which FuzzyCSSolution instance to use. Then, it calls the function

`sol_instance.get_m_best_fixed_solutions_branch_n_bound(m, want_items, no_want_items)`. It then returns whatever it gets from that function to the UI Server.

3.1.7 PS3 and extensions

We added one function to PS3 to return the m best solutions, considering the items wanted and items not wanted in the solution:

```
1. get_m_best_fixed_solutions_branch_n_bound(m, want_items,  
    no_want_items)
```

This function uses the branch and bound algorithm introduced in PS3 to get m -best solutions. While getting m -best solutions, it prunes the branch if the current value chosen in the algorithm is in the `no_want_items` list, or if it is not in the `want_items_list` when `want_items_list` contains items in it. This is done before the usual branch and bound operation.

Here, one decision we had to make was whether to only accept a solution if it contains all of the wanted items, or it is okay if any of the item in the `want_items_list` is in the solution. We went with the latter option, and we discuss it why in the next section.

3.2 Some creative approaches/design decisions

In this section, we discuss some of the decisions we had to make while creating the system.

3.2.1 Client-Server vs. I/O

We had two choices of how to communicate the user input from web client (Javascript) to the main system (Python) and vice versa, i.e. how to deliver the solutions back to the web client. One option was to have a producer-consumer architecture using I/O, i.e. Javascript collects all the inputs and publishes/writes/outputs to a file, which Python continuously reads from. When anything changes in that file, Python knows the input has changed, and runs the backend engine with the inputs parsed from that file.

Another option was the client-server architecture, where Javascript can send AJAX requests to a Python server listening to a port.

We went with the client-server approach, because it seemed more natural, given we are using website in front end, and it is very easy to send AJAX requests from websites to a server. Javascript has an in-built way of doing that. Also, it is equally easy for Python server to send the responses back, with the same provided architecture. We found really easy-to-install modules called “Flask” and “Flask-RESTful” which made our work very easy.

3.2.2 Individual Availability Score and Pair Availability Score

One requirement of our scenario was that we needed to derive a availability scores between the values of each pair of variables. Eg, we need to have a score between (“main” = “eggs” and “drinks” = “juice”), (“main” = “bread” and “drinks” = “juice”), and so on. This is because we want to come up with binary constraints for the availability problem described above.

First, we need individual item’s availability score. We get that with the following formula:

$$Availability(item) = 1 - \frac{frequency(item)}{ttl(item)},$$

where $frequency(item)$ is how many times the item has been used in breakfast, and $ttl(item)$ is time to last, i.e. total number of servings for the item. Here, if $frequency(item) == ttl(item)$, then $availability(item) = 0$, i.e. if all servings have been used up, availability is 0. On the other hand, if $frequency(item) == 0$, i.e. it hasn't been used at all, the availability is 1.

Now, the availability score for pair of item is given by the formula:

$$Availability(item1, item2) = \sqrt{Availability(item1) * Availability(item2)}$$

This is the geometric mean of availabilities of two items. This was chosen, because if one of the items is not available, we want the total score to be 0. Similarly, this score makes sure to give higher penalty (= lower score) if one of the item's availability is low, compared to taking average, for example.

3.2.3 Combined Binary Preferences from Availability and Preference scores

The combined scores from availability and preference scores was also chosen to be the geometric mean of two scores, for similar reasons. If something is not available at all, we don't want it in the feasible solution, and same applies if the master doesn't want the combination at all.

There is one tricky part, however. The master gives preferences for only a few combinations, and similar thing applies in real life. We cannot expect a master to enumerate preferences for each possible pair of items. However, we need a score to combine with the availability score. For this condition, we have assumed the preference score to be 0.5, which implies that the master is 50/50 on his preference between two items, if he hasn't explicitly stated.

3.2.4 Or vs. And of Want Items

As stated before, we keep track of what items the user has said that he/she wants or doesn't want over the course of the interaction in the UI. While looking at m-best solutions, we consider those items. For `no_want_list`, it is clear that we should discard a solution if it contains any of the items in that list. However, for `want_list`, it is not clear whether we should accept a solution only if it contains all the wanted items or any of them.

We went with the latter, i.e. we accept a solution if it contains any of the wanted items. It was done because we assume that when a user says he wants a certain item, and later says he wants another item, they shouldn't mean that he wants both of those items. Also, if the user says he wants two values of the same variable, e.g. the `want_list` is ["juice", "milk"], we have no feasible solution if we assume the user wants both, because there is no solution possible for ("main", "complement", "drink") combination in such a case. In most of these cases, we found "either of the items" to be more plausible than "none of the items". This still could be debated otherwise, but we chose one that we felt was a better choice.

4. Performance of the scenario

According to our benchmarks tests in PS3, branch and bound performed significantly better than heuristics search as the size and complexity of the problem increased. We used ‘tightness’ and ‘connectivity’ as a measure of problem complexity in our tests. The connectivity of the problem is represented by the connectivity of the constraint graph that represents the problem. The connectivity of the graph is the proportion of variables that are related by a constraint. Another measure we used was tightness - which refers to the proportion of variable assignments disallowed by the constraints. Tightness in a more intuitive sense is the difficulty of finding a solution to the problem.

Since branch and bound performed consistently well in our tests as we varied these problem characteristics, we opted to use this algorithm for our implementation of m-best solutions for the grounded scenario.

Since our grounded scenario had three variables and fewer than 20 constraints, the performance of our technique was very good. However, as we know, since fuzzy constraint satisfaction converts constraint satisfaction problem into an optimization problem, the runtime as well as the memory requirements are expected to grow exponentially with the number of variables. The runtime of branch and bound algorithm was seen to increase beyond 1 milli-sec with variables exceeding 6 in number. For 7 variables, it took branch and bound 7 milli-sec to compute the best solution.

The performance for our branch and bound based m-best solution ranged from 3-5 millisec. There was additional delay in request and response due to the client and server interaction that we are not accounting for in this measurement.

Though our implementation is capable of solving several generic problem with varying number of variables and constraints, we chose the scenario based on our original proposal. The algorithm performed well on time and memory consumption in our demo.

