Robert Johns
CSCI 7637: Knowledge-Based AI
Assignment 2
February 15, 2018

Designing New Recipes With Case-Based Reasoning

I. Introduction

All humans eat food. All humans solve problems based on problems they've already encountered. It is therefore natural to tie these two together and consider how we could design an AI agent to reason, based on previous cases, about the design of a new recipe. In this way, our agent will explore a fresh and exciting territory: computational creativity.

II. Case-Based Reasoning

In the case-based reasoning (CBR) problem-solving methodology, an agent, upon encountering a new problem, employs the following steps:

1. Search through a bank of solutions to problems in the same domain, finding and retrieving the most similar.
2. Modify the retrieved solution to better fit the current problem.
3. Evaluate the effectiveness of the solution.
4. Store the solution, to be retrieved later. (Goel, Joyner & Thaker, 2016, p. 102)

In addition to its abstract simplicity, this problem-solving methodology becomes more effective as it encounters more cases. As our agent designs and stores news solutions, it will need to expend less effort to find solutions to new problems.

III. Designing Recipes

In a way, all of culinary history could be described as case-based reasoning. Cultures throughout history have developed their cuisine over time, making modifications to fit the availability of ingredients, changing cultural tastes, and newly available cooking technologies. In developing their cuisines, world cultures have, in a way, been solving the same problem (with evolving parameters) over and over again: that of creating recipes within constraints to satisfy needs. Rather than creating these dishes from scratch, chefs will take an existing recipe and modify an ingredient, or a step, to satisfy a constraint or simply to try something new.

To design an AI agent which can design recipes, we need to define the problem explicitly. As a disclaimer, I am neither a culinary historian nor a competent cook, but I've broken down the structure of recipes as follows:

- A **recipe** is a set of instructions to create a **dish** for eating.
- A dish consists of at least one **component**. As an example, macaroni and cheese consists of two components: the macaroni noodles and the cheese sauce.
- A component is comprised of **ingredients**, which are the atomic units of a recipe.

- To combine the ingredients into components and the components into a dish, a recipe contains **steps**, for example "bake at 300 degrees for 30 minutes" or "fry in olive oil until golden brown"

Once cooked, a final dish consists of several qualities, which we will use as the parameters for our CBR algorithm. The qualities of a recipe our algorithm will consider are:

- **Style** of recipe. Instances include "slab of meat" (steak), "sandwich" (hamburger, Reuben, NOT a hot dog) and "salad", among many others.
- **Flavor profile** of the recipe, consisting of values like "spicy" and "sweet"
- **Nutrient makeup** of the recipe.
- **Ingredient makeup** of the recipe. This quality will serve to allow for specific ingredient requests, like "I want something with Gouda cheese", and will inform the flavor palate and nutrient makeup.
- **Miscellaneous data**. Much like the community value in the BGP routing protocol, this will be a catch-all for any relevant information that does not fit the above qualities. For instance, if the user wants a breakfast dish, "breakfast" would be added to the miscellaneous data.

We assume that the problem of designing a recipe is stated in this way: a user will input parameters into the agent, for example "I want a spicy slab of meat with 20 grams of protein, with a sauce that contains cayenne pepper." We will then input those parameters into the agent, which will process the request and determine a recipe, and will then output a list of ingredients and a set of steps for creating the meal.

IV. Problem-Solving Method

Our algorithm will be responsible for all steps in the CBR methodology except for the third: evaluation. Evaluation should be done by the end user: only they can determine for themselves if the recipe tastes good and satisfies their needs. After our agent produces a recipe, the user will evaluate the recipe returned and give feedback to the agent indicating whether or not the recipe was cooked, and if so, how successfully it came out. Broad pseudo-code for our algorithm follows:

```
parameters ← get requirements from users
retrieved_recipe ← retrieve_recipe(parameters)
recipe ← retrieved_recipe
while distance(recipe, parameters) < goal:
      recipe ← modify_recipe(recipe, parameters)
      if distance(recipe, parameters) > limit_1:
            parameters ← relax_constraints(recipe, parameters)
            retrieved_recipe ← retrieve_recipe(parameters)
            recipe = retrieved_recipe
if distance(recipe, retrieved_recipe) > limit_2:
      store recipe in database
return recipe
```

From here, four functions need to be discussed: `retrieve_recipe()`, `distance()`, `modify_recipe()` and `relax_constraints()`.

Beginning with the most straightforward function, `relax_constraints()` will eliminate the parameter the current recipe is furthest from satisfying. For example, if a user asks for a sweet bowl of pudding with chocolate and ghost peppers, in all likelihood no recipe in existence will combine anything sweet with a ghost pepper. Therefore, `relax_constraints()` would drop the ghost pepper parameter, allowing the algorithm to break the infinite loop.

All the remaining functions depend, in a way, on `distance()`. Because of the qualitative nature of the parameters, this function can also be tricky. Some of the qualitative measures can be made quantitative (a continuum of spiciness, for example). Others will require a slightly different tactic. For example, our agent could, instead of having a binary on-off continuum for every possible ingredient, simply compare the list of required ingredients for a recipe against the list provided by the user. Once all of the individual similarity metrics are computed, they will be aggregated into a total similarity score (by using k-nearest neighbor or a similar distance metric) and returned to the user.

The function `retrieve_recipe()` will function similarly to `distance()`, except returning a recipe instead of a similarity score. This will require a careful indexing scheme to efficiently move through a large volume of recipes. Our agent can easily cut down on its options by only considering the style of recipe given in the parameters. Depending on the number of styles our agent can consider, this could potentially cut out a huge majority of possible recipes, leaving a manageable number for `retrieve_recipe()` to search through. Once the options are trimmed, our agent could potentially employ an n-ary discrimination tree to more quicky find well-matching options.

The last and most important function, `modify_recipe()`, is where the reasoning comes in. Odds are that no stored recipe will exactly match the parameters provided by the user. The agent then encounters its heaviest lifting: modifying the existing recipe while remaining within the bounds of the parameters. To do this effectively, our agent will require a separate database from which to pull information, containing ingredients, their qualities and nutritional values, and ways to cook them. The function `modify_recipe()` will use this database to consider modifications: if a ghost pepper is too spicy, perhaps a habanero pepper will do. If baking a component doesn't result in the right texture, perhaps frying it will more closely match what the user desires.

The final piece of the algorithm is the storage scheme. To keep retrieval manageable, it's critical not to store every successful recipe in the database. Therefore, our agent will only store a recipe if it is at least a minimum distance away from its nearest neighbor, according to the `distance()` function.

V. Conclusion

The designers of IBM's recipe-concocting system have stated that while "humans are good at reasoning about two ingredients… pretty much no one can reason about four ingredients." (IBM Research). Harnessing the power of computers can help to mitigate some of these combinatorial woes: while a computer has difficulty *creating* or truly *thinking* like a human, computers can *process information* much faster than humans can. Using machines to design new recipes is an exciting application of artificial intelligence design. Given CBR's relation to how humans have solved the recipe-creation problem throughout history, our algorithm forms a solid foundation for designing an agent to better feed all of us.

Works Cited

IBM Research. (2013, November 22). *Computational Creativity.* Retrieved from https://www.youtube.com/watch?v=mr-1JAnairs

Goel, Ashok; Joyner, David; Thaker, Bhavin (2016). *KBAI Ebook: Knowledge-Based Artificial Intelligence*. Publisher, city and state not given.