Case-based Cooking: A semantic similarity based approach for retrieval and adaptation of recipes for lists of ingredients

Bachelorarbeit

im Studiengang Angewandte Informatik
der Fakultät Wirtschaftsinformatik
und Angewandte Informatik
der Otto-Friedrich-Universität Bamberg

Verfasser: Robert Terbach

Gutachter: Prof. Dr. Ute Schmid

Abstract

After a short introduction to case-based reasoning this work builds a simple system centring around creating adaptations for recipes. In the domain of cooking a data base of recipes serving as cases and an ontology of foods as background knowledge build ground for the application presented here. The ontology as well as recipe base are lent from the research project WikiTaaable, which hence is also introduced here. The goal is creating adaptations for recipes when they can not satisfy a query. These adaptations are created when needed from similarity measures operating on the ontology. Focus of this work is on methods to automatically create adaptations. The strategies of creating are evaluated as well as the mentioned similarity measures. Further on a measurement is introduced which tries to evaluate the fitness of a substitution to a recipe. Additionally a Java program is provided which allows to explore the ontology and recipe data base. Finally the problems with the ontology and the algorithms working on it are discussed and suggestions for improvement given.

Contents

1 Introduction **1**

2 Case-Based Reasoning and the Computer Cooking Contest **3**
2.1 Case-Based Reasoning 3
2.1.1 The Case-Based Reasoning Cycle 4
2.1.2 Current Developments in Case-Based Reasoning 8
2.2 The Computer Cooking Contest 10
2.2.1 Participants of the Computer Cooking Contest 2012 . . . 10

3 Approach to Creating an Own Case-Based Reasoner **14**
3.1 Motivation 14
3.2 System Outline 15
3.3 WikiTaaable Data as Case Base and Background Knowledge . . 16
3.3.1 Recipes 16
3.3.2 Foods 17
3.3.3 Dish Type Ontology 17
3.3.4 Other Features 17
3.4 Recipe Retrieval—Queries 19
3.5 Adaptations 20
3.5.1 Finding Matching Adaptations 20
3.5.2 Manually Created Adaptations 21
3.5.3 Generating Adaptations from Food Classes 21
3.5.4 Fitness Criterion 24
3.5.5 Generalised Substitutions 26
3.5.6 Problems 27
3.6 Revision 29
3.7 Storage 29
3.8 Conclusion 30

4 Realisation and Evaluation **31**
4.1 Data Structures 31
4.1.1 Datastore 31
4.1.2 TreeItem 32
4.1.3 Recipe 32
4.1.4 Query 33
4.1.5 Substitution 33
4.2 Metrics 34
4.2.1 Food Similarity 34

4.2.2 Recipe Similarity 34
4.2.3 Substitution Fitness 36
4.3 Evaluation 37
4.3.1 Test Data 37
4.3.2 Finding Possible Substitutions With the Ingredient-Ingredient Similarity 38
4.3.3 Evaluation of Substitution-Recipe Fitness 43
4.4 The GUI-based Program 45

5 Conclusion **47**

Bibliography **51**

A WikiTaaable XML Dump Examples **52**

B Java Application **56**

List of Tables

3.1 An example ingredient line showing possible properties of an ingredient line in a recipe. 16

4.1 Found substitutions for different missing kinds of oil. 40
4.2 Found substitutions for *cayenne pepper* and *cocoa*. 42
4.3 The four variations of the substitution fitness algorithm. 43
4.4 44

List of Figures

2.1 The case-based reasoning 4R-cycle. Figure from Aamodt and Plaza (1994) 5
2.2 The task-method decomposition of case-based reasoning from Aamodt and Plaza (1994) 6

3.1 An excerpt of the food ontology in WikiTaaable also used in this work. We can see the deeply elaborated structure, but also can see problems, e.g. the veal breast and the lamb breast are not part of the hierarchy corresponding to their animal. The arrows show the *subClassOf* relation. 18
3.2 Local ontology around *Balsamic vinegar*. 24
3.3 Local ontology around *Chicken breast*. 24
3.4 Excerpt of the local ontology below *Cheese*. 28

4.1 Number of possible substitutions found for missing foods by different algorithms. 39
4.2 Excerpt of local ontology around *oil* and *fat*. 41

B.1 A screenshot of the application which is part of this work. It shows an example manual adaptation process. 56

Chapter 1

Introduction

Digital devices are more and more invading our daily business. Long gone are the times that computers were big bulky blocks with even bigger cubes as displays. Today we are able to put all this together into devices not even a centimeter of thickness with thousand times the computational power and storage. Computers have gone mobile and from being professional tools to cheap common household items and permanent companions. A lot of information can be stored on spaces so tiny they cannot be seen. Warehousing led to the RFID[1] technology which allows for identification and localisation of objects in a large scale. With the currently emerging NFC[2] capable phones and credit cards this technology finally arrived the consumer market. In fact many items purchased today are marked with RFID tags and more and more will. Specialised computers are built in many objects without even being noticed. Digital toasters sending an e-mail when the toast is ready are the future. More and more devices in a household become connected—the internet of things. And so are refrigerators. Combine these technologies and the fridge will know what it contains as all foods are marked with RFID or recognised with image processing technology and send notifications in case something important is missing or expired. A user could plan his meal on the way back from work and the computers in the fridge and in the pantry can tell what and probably even where to buy missing goods. In a step further robots could even cook a complete meal[3] from the ingredients put into it.
The work presented here does not go that far, it leaves cooking to humans, but embraces the idea of mobile computers supporting in the kitchen. The high amount of internet sites specialised in cooking and availability of tablets, some of which are specifically built for this purpose, made big steps to digitalise cooking. This work takes the idea of a fridge having an overview of available foods and aims to provide recipes that can be cooked from these. For this the system built here has access to an ontology of foods and knowledge about recipes and their characteristics. Of course cooking is not that simple, and tastes are different and often not all ingredients needed for a recipe are available. Therefore the system should learn and adapt, not only adapting to the users preferences or diets, which is work of recommender systems, but also adapting recipes. A case-

[1] Radio-Frequency IDentification
[2] Near Field Communication
[3] `http://everycook.org/`

based reasoning system is presented trying to find and adapt recipes so they can be cooked from available foods. Adapting has two interesting points at which it is useful: First some recipes, especially when they are searched by ingredients available, most likely are not be built up from the exact foods available, say a recipe uses red pepper, but unfortunately only black one is available. For cooking the recipes it should not be a problem. This point is about recipes being specified slightly different than the data provided for searching. Point number two is about making replacements with higher impact—unfortunately the vinegar is empty, which was needed for the salad dressing, how can it be replaced (without asking the neighbours)? The juice of a lemon might do the job as well. These two ingredients do not directly seem to be equivalent. This might be taken further. A Viennese schnitzel is originally made with escalopes of veal, but it can also be made from pork (which is thought of being barbaric by some people, but due to the costs very common).
The presented work focuses on creating these adaptations by using knowledge of relations (in the sense of being akin) between foods and is trying to evaluate if a substitute fits to the original recipe. The following chapters introduce concepts of case-based reasoning and its recent history before the system being built is presented. In doing so the strategies for creating adaptations as well as evaluating them are illustrated. Finally the implementation and its results are being discussed.

Chapter 2

Case-Based Reasoning and the Computer Cooking Contest

2.1 Case-Based Reasoning

The expression *Case-Based Reasoning*(CBR) includes the word *reasoning* so first we remind that concept: Reasoning, simply said, is transforming and combining a set of information into new meaning. From existing facts new ones are created. From this, case-based reasoning is an approach to problem solving *based* on *cases*—previous problems. Its goal is to solve new problems remembering and reusing knowledge from previously solved cases. This leads us to the useful "byproduct" of this problem solving procedure, which makes case-based reasoning a matter of learning. Past experiences are stored and therefore available for reuse. This kind of learning is not only of theoretical use, it is inherent in human nature. Research in the field of cognitive psychology approve the validity of learning from past examples in human problem solving. Studies also show that this strategy is used not only in early learning, but experts use it as well(Gentner (1983). When transferring the mentioned technique to computing we nevertheless need to relax the abilities in comparison to the human mind. On the one hand we cannot change the actual structure of the computer's memory so easily (not to say that this was simple in the human mind, quite the contrary it is an extremely complex process, but at least hidden to the learner), and on the other hand we must focus on *intra-domain analogy* in contrast to cross-domain analogies that can humans can actually use. The common example of cross-domain analogy is the symmetry of solar systems and the usual Bohr model of atoms(Aamodt and Plaza (1994); Carbonell (1985, 1983)). The understanding of case-based reasoning used in this work must be simplified to solve problems within one domain, e.g. reasoning from known checkers positions and their ongoing courses of the games to a move to be made in a new previously unknown position, or creating a new recipe from existing ones.

Learning
We can learn in many ways from actions and their outcomes: The simplest is learning a new solution. In the case of checkers this would mean that a draw at any time of the game led to a better position, or even a win. The next step is learning from errors, implying a single draw that led to a worse position is to be learned as bad move to prevent repeating the mistake. It is also possible, though hard, to learn strategies. This goes further than making one move in the game, it is a multi-step operation, say we want to build a certain pattern on the field of checkers, which can be achieved by a sequence of draws. We remember a game in which we were in that position, but started from a different initial situation. We can take some of the steps from the previous games, and adapt the pattern where needed. By this we learned a new pattern. A computer system using the described strategy is PRODIGY/ANALOGY which learned writing computer programs (Veloso (1994)).

Regarding case-based reasoning the paper *Case-Based Reasoning: Foundational Issues, Methodolical Variations, and System Approaches*, which, with its 19 years of age, is even an adult by now, written by Agnar Aamodt and Enric Plaza (Aamodt and Plaza (1994)) still is the best roundup and historical outline of case-based reasoning. The following is trying to condense the most important parts and give an overview of developments of the most recent years.

2.1.1 The Case-Based Reasoning Cycle

The Process Model
In principle a CBR systems work consists of the following steps: Understanding the latest problem, finding matching one or more past experiences, adapting the knowledge to the current situation, assessing the suggested solution and finally adding the new insight to the system's knowledge base. Aamodt and Plaza split CBR-systems into different categories and build a framework to describe different approaches and methods of case-based reasoning. The framework is divided into two orthogonal parts, the first describes the general process of CBR, and the second presents different tasks within steps of the process. In the process model they make clear that case-based reasoning is a cyclic process. We can confirm this view with our previous description saying that we learned a new example from a solved problem further improving our knowledge. Every learned case can later be retrieved for solving a new problem. The frameworks CBR cycle is quartered into the following sub-processes:

1. Retrieve
2. Reuse
3. Revise
4. Retain

When tackling a new problem, it is solved by being processed from each of these steps. Initially one or more similar cases are *retrieved* from the case-base, then these are *reused* to suggest a solution. Next step is to *revise* and see if the solution is satisfying. Finally the new solution is *retained* for later use. As of the *R*s in the processes names the CBR cycle is also called the *4R Cycle*. The cycle is illustrated in figure 2.1.

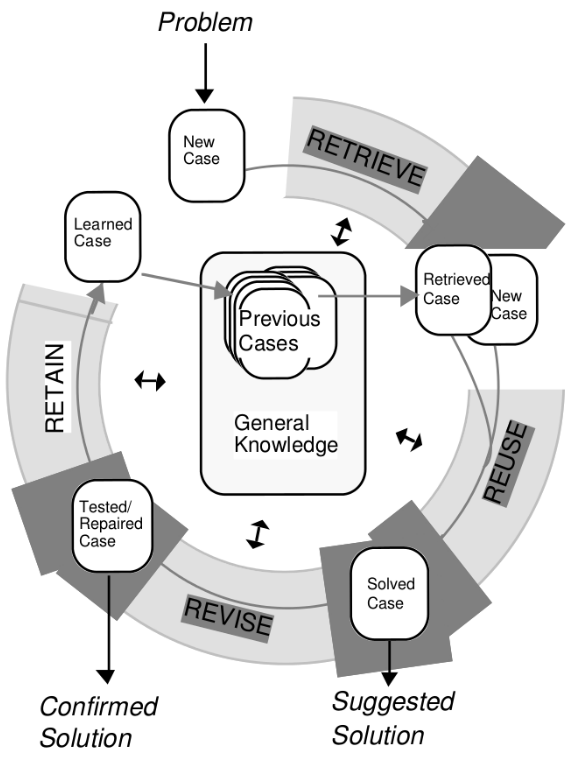


Figure 2.1: The case-based reasoning 4R-cycle. Figure from Aamodt and Plaza (1994)

The Task-Method Model

On the other hand the *task-method view* decomposes each of the four processes into tasks which describe the internal mechanism of the CBR system. Aamodt and Plaza break these tasks down even further by naming methods that can be used to complete the tasks. The hierarchy is shown in figure 2.2. The plain lines mark part-of relations. We can see that the top task *problem solving* is divided into the four processes we saw earlier. These in turn are decomposed into multiple subtasks. The items that are connected to the subtasks by the stippled lines are different methods suitable for accomplishing the tasks. While the tasks are meant to be complete, say sufficient to fulfil the parent task, the methods are just suggestions. Other methods or combinations might work as well. The methods are a matter of choice and determine the outcomes of performing a task, therefore methods have to be chosen suitable for the target domain.

Knowledge Representation

A fifth component not only completes the cycle, but lays the foundation of the CBR process: *Knowledge representation.* The importance of this is visible in figure 2.1—the knowledge base is the centre of the circle and interaction with

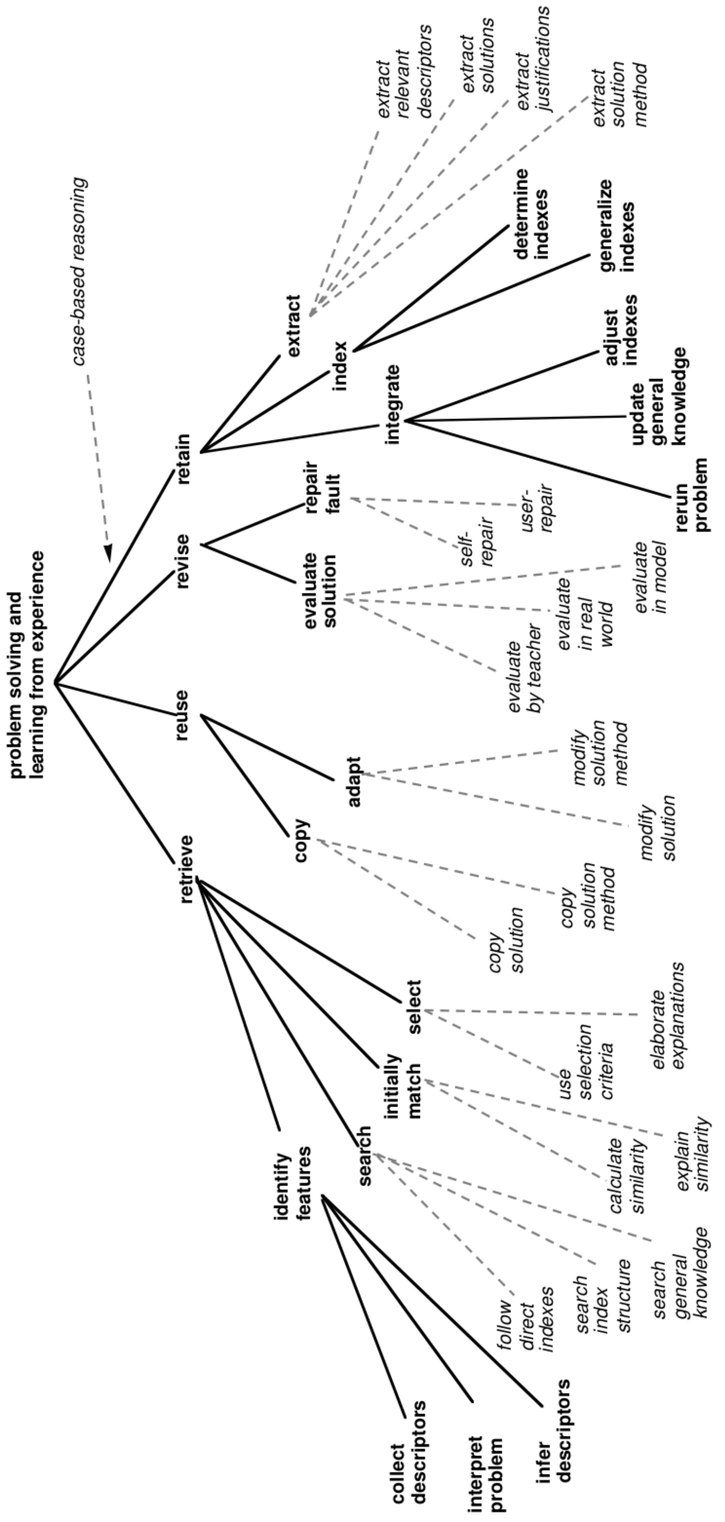


Figure 2.2: The task-method decomposition of case-based reasoning from Aamodt and Plaza (1994)

it may be an important factor in each step of case-based reasoning. Knowledge can be split into two parts: Representation of cases and general knowledge. Obviously representation of cases is a very important aspect in a CBR system as the system must be able to efficiently retrieve known sequences and the structure may have meaning as well e.g. by relationships between multiple cases. It is also important to be able to add new cases as result of a successful learning experience. In their paper Aamodt and Plaza revise two influential memory models: The dynamic memory model and the category-exemplar model.
The first one focuses on building a hierarchic structure where the elements are called *episodes*. In this model specific cases that share similar properties are grouped to a more general episode. When adding new cases it is possible to regroup episodes dynamically to a new generalised episode. This way the cases can be navigated by a tree structure, and generalised knowledge can be used for adaptation (substitution and adaptation will be used synonymously from here on) more easily.
The category-exemplar model is organised into a three-tier structure. On the bottom are the cases, in this model also called *exemplars*. If possible, say a group of exemplars share a set of features, they are grouped to *categories*. Cases are connected by links, if they differ only in few features. Within one category the exemplars are sorted by similarity to their prototypical category. On the highest level links are connected from the features, also called problem descriptors, to these cases and categories.

Retrieval

Retrieval is the process that leads from a possibly even incomplete problem description to one or more best matching cases. The process starts with identifying the relevant features from the input and, if needed, preprocessing the problem. In the next step the case base is searched for one or more matching cases. From this set of possible matches a similarity measure is taken to explain why a case is an analogue. With this measure decisions can be made which case fits the best. Selection of the best match can best be made with more background knowledge. For this task it is possible to weigh features providing more information for decision making. Knowledge for the selection task can be knowledge in the system or help of a human expert choosing the best match.

Reuse

In the simplest case a simple copy of the known result can be a correct answer for the problem, i.e. in classification problems. In other problems it might be necessary to adapt the existing solution. Aamodt and Plaza describe two main ways for adaptation: (1) Transforming the original solution to better match the new case, and (2) transforming the method that constructed the known solution, so called derivational reuse. While the first one simply operates with some transformations made available from a domain describing model, the latter one reconstructs the creational process of a solution. This is a planning process in itself. The plan can be modified i.e. by changing operators, creating new subgoals, or reevaluating alternative paths. *Prodigy/Analogy*Veloso (1994); Carbonell (1985, 1983) is an example of derivational reuse.

Revise

Before a solution can be marked as correct (or wrong) and be learned as such, it needs to be revised. Usually revising happens outside the CBR system. As it may take a long time, e.g. medical treatments, to be sure if a solution was correct, so the possibility to learn a case without certain correctness is needed as well, this can be achieved by marking the it as non-evaluated. In some cases an expert can evaluate quickly if the solution was correct. The recipe planner CHEF(e.g. Hammond (1986)) is able to simulate its resulting recipe to an internal model deciding for itself if the proposed solution needed repair.
If repair the adaptation was not enough to create a working solution the case might be repaired. CHEF can repair recipes thanks to a strong causal model that can identify errors in the plan and add steps to it which ensure that the error does not occur. Of course error repair can also be accomplished by an expert. After the repair step the case can either be retained, or the revision can be repeated until no error can be found.

Retain

Retaining is the learning step in a case-based reasoner. It is important to decide which parts of a solution should be retained, be it the complete case, or as in the Prodigy/Analogy system the methods that create a case. In general the first subtask is to identify these features, if needed justifications, descriptors and the like. It is system dependent if a completely new case is stored, or just the changes made in adaptation. When storing changes, this step can be taken further by generalising multiple cases, as it is described in the memory model paragraph. Finally the new knowledge needs to be integrated into the data model and the model itself be updated to reflect the learned episodes.

2.1.2 Current Developments in Case-Based Reasoning

Since the recapitulation of Aamodt and Plaza has been published case-based reasoning has evolved a lot. New methods have been developed, the existing were further elaborated. CBR also split up into different fields of application. One very big area that researches capabilities of CBR is in the field of health sciences.

Health Sciences

It is not for nothing that on the International Conference on Case-based Reasoning(ICCBR) of 2012 a complete block was reserved for this field. CBR not only has a part in diagnoses, it is also used in supporting therapy. In the health sciences CBR systems often take the role of *clinical decision support systems*(CDSS). CDSS connect knowledge with actual observations with the aim of improving health care. Marling et al. (2012) describe a system that attempts to help type 1 diabetes patients on insulin pump therapy. Their outcomes show promising results, nevertheless need more improvement. In problem classifying tasks accuracies of 77.5% to 97.9% were possible. Case retrieval was also quite successful, with retrieved cases being rated ad beneficial 82% of the time by experts. An interesting evaluation concerns plan adaptations where adaptations were rated fine 47% of the time and minor adjustments were needed in 47% of the test cases. The numbers show promising results, but also make clear that more work needs to be done, as error rates in that height are not acceptable for medical use. Elvidge (2012) evaluates a program that tries to help nurses in end-of-life cancer care improving the patients quality of life.López et al. (2012) aim to give predictive prognoses of breast cancer risks based on 1199 attributes describing peoples habits (smoking, diets, sports, etc.), clinical history, genetic information, and more. The system has already proven to be better than non-CBR competitors, that failed in combining all information available.This only holds for test data as the real data they tested with is missing most values for the vast amount of available features, though. Ahmed et al. (2012) introduce a system helping diagnose stress related disorders. This one does not only use CBR as method of choice, but also shows another trend in case-based reasoning: Combination and integration with other techniques of artificial intelligence. In this case fuzzy logic, rule-based reasoning and information retrieval techniques are used. Their evaluation shows that CBR in combination with the other methods is able to support even experienced clinicians and enables individualised diagnoses.

Computer Games

Another field that shows interest in CBR is artificial intelligence for computer games Ontañón et al. (2007, 2008); Palma et al. (2011)). After being successful with an AI playing real-time strategy games, the authors of Silva et al. (2013) changed to the domain of role playing games. Different to the previously presented CBR application this area of research takes differences over time into account and directly manipulates the environment. Their system learns directly from the player in a distinct learning phase of the game. The player takes the role of the character the AI is supposed to play and solves given tasks in the way he wants the computer to act. This approach is also used in other works, learning from demonstrations is becoming a useful technique to generating cases. It was also a focus topic in ICCBR2012 under the title *Traces for Reusing Users' Experiences*, where traces (or stories) represent episodes of user interaction that build the cases. Thevenet et al. (2012) uses traces to learn new gestures on-the-fly for a motion controlled interface. We have seen in these examples that a focus is put on setting up a case base, in the examples these are drawn directly from user interaction—the behaviour of the user will later on determine actions the reasoner can take.

Case-Based Reasoning Processes

Research on the process view is putting screwdrivers on all four processes, as well as the knowledge base. On retrieval work is, for example, put in optimizing retrieval times. The CBR systems often struggle when retrieving cases from a large case base. The similarity measures usually do not scale well. It is easy to imagine that a naive k-nearest neighbour algorithm that works on more than 1000 features cannot run efficiently on thousands of cases. Both, Bergmann et al. (2012) and Kendall-Morwick and Leake (2012) are using two-phase retrieval methods to improve performance in those situations. Generally speaking the first phase filters the case base for probably relevant cases so that the second phase, can operate on less cases. This is done to highly reduce the number of costly comparisons if possible. Nevertheless these optimizations come with the risk of losing the optimal case, when it was filtered out. Other approaches to

improving retrieval lie within the similarity measures.
On the process of retrieval and revision a lot of research is done on weighting feature importance e.g.: Jagannathan and Petrovic (2012); Bergmann et al. (2012); Gunawardena and Weber (2012); Kar et al. (2012). Confidence is another interesting measurement taken into account when evaluating how good a match is. It is also used to describe the fitness of adaptations (Minor et al. (2012); Kar et al. (2012)).
Regarding storage of cases the work is done in organizing the case base and structuring cases. The *Utility Problem* describes the dilemma of learning cases. When a new case is learned, it may only differ little from the case it was derived from. Therefore, if it is stored directly we have two nearly identical cases. If this happens often the case base grows rapidly without adding novel information making case retrieval slower. Hence the database size increases, but the overall performance decreases (e.g. Houeland and Aamodt (2010)). This may be avoided with smart generalisation strategies as stated earlier or effective indexing strategies. Also randomly *forgetting* cases is shown to be an effective strategy to keep the case base small (Markovitch and Scott (1988)). CBR more and more becomes integrated into or empowered with other methods of machine learning. Be it fuzzy logic or information retrieval as stated above, or other such as genetic algorithms (López et al. (2009)).

2.2 The Computer Cooking Contest

The Computer Cooking Contest(CCC)[1] is an event taking place at the International Conference on Case-Based Reasoning since 2008. It was established to attract young and new people to the field of case-based reasoning and artificial intelligence, as well as to provide an application for research projects that can draw public attention to it, like the roboleague is able to for robotics. The contest is open to everything related to food and computers and gives a platform to very different approaches some of which we will see in the next chapter. The common target is creating new recipes or menus which will be evaluated by professional cooks and rated by international scientists.

2.2.1 Participants of the Computer Cooking Contest 2012

The following section will introduce several participants of the CCC 2012 and their different approaches to digitalising food.

Yummly
Yummly[2] basically is a webservice that collects high quality recipes from various professional recipe pages and lots of food blogs, from general cooking to some serving specific diets. Currently they have about 60 sources and more than 1.000.000 recipes, still expanding. Very special about Yummly's service is the search engine. It starts with simple ingredient-based search, where single ingredients can be included or excluded, but allows for specific searching by season of the year, preparation time, diets, allergies, cooking techniques, recipe origin, and the course. Furthermore they have an interesting ordinal scale for

[1]http://www.computercookingcontest.net/
[2]http://www.yummly.com

tastes like *salty*, *savory*, *sweet*, or *spicy*, where likes and dislikes can be set up, as well as an absolute scale for nutrition values per serving. Yummly can also organize favorite recipes of a user and recommend new recipes based on these. From the case-based reasoning point of view Yummly states that it can determine substitutions in recipes by background knowledge or comparison with other recipes[3].

Foodpairing

Foodpairing[4] works completely different from other recipe services. Instead of recipes they only store foods. With this foods new recipes can be created by *pairing* them based on their flavour. The company analyses foods, right now they have analysed more than 1000 ingredients, for their major flavor components and structures the results in so called *trees*. The trees are clustered by categories like meat or herbs and subcategories. The currently chosen food is visualised in the centre, possible combinations are arranged around it. The better a new ingredient fits to the currently chosen, the closer it is to the centre one. It can only create the trees from one centred ingredient, so if a third food should be added to a pairing the user must ensure it fits both previous foods himself. During a combination task the centred ingredient can be changed, though. Foodpairing can only provide pairings of ingredients, it can not create full recipes with quantities or even instructions, so Foodparing is just an inspirational source for creating new recipes for chefs or bartenders.
For cooking with case-based reasoning this could be a powerful background knowledge source, very usable when trying to find substitutions since it allows finding close matches by taste, independent from categories.

Siansplan

A more integrated look into cooking is the background of Siansplan[5]. Sian is an English Mum who wants to organise her family meals. With the website `siansplan.com` she and the team aim to help planning meals for an overall week. The main idea is that the user decides on which days he can cook normally and plan the recipes for these days. Siansplan then creates printable shopping lists to reduce the amount of not needed food bought. Up to two days in a week can be marked as recycling days, on which recipes are used that can be made with leftovers from the other days further reducing garbage food. It provides some 100's of tested recipes and the possibility to add one's own recipes. Siansplan embraces its users with weekly new handmade plans sent via email fully packed with links to the recipes used and a printable shopping list even with reminders what you should check your stock for. The mails also contain tips for replacing items, e.g. in case one does not want to buy expensive beef that chicken is a good alternative. This of course is not done by a case based reasoning system, but handmade input and Siansplans approach of using computers for organising food is an interesting contestant in the contest though.

Taaable and WikiTaaable

Taaable is a long-term contestant at the CCC. During the years they partic-

[3]`http://help.yummly.com/entries/355269-How-are-substitutions-determined-`
[4]`https://www.foodpairing.com/`
[5]`http://www.siansplan.com`

ipate they enriched the CCCs default data set, put it into ontologies and are constantly attempting to improve these with new data. Taaable is the CBR system and the ontologies have been transferred to a Semantic MediaWiki[6] called WikiTaaable. From the CCCs recipe base they created six hierarchies and annotated these with a lot more data. The most basic of these is the *food* ontology containing more than 2000 ingredients. The other ontologies contain *dish types*, *dish roles*, *origins*, *diets or allergies*, and *culinary actions* that describe the preparation of ingredients and the cooking itself. Most important for this work is the food ontology, the others, except *culinary actions* are used as well, therefore they are introduced in a more detailed manner later on. The wiki also contains known substitutions, they are distinguished by two types—context independent generic substitutions and specific substitutions. The latter ones are only applicable for one recipe they were created with. The wiki does only contain one generic substitution and also only few specific substitutions, though. Substitutions are parts of learned episodes.
Recipes contained in WikiTaaable consist of a list of ingredients, annotated with the amount and its unit. The recipe also has a text that describes the preparation, as well as a dish type. Diets it is suited or is not suited for are given implicitly by the foods used. The wiki also contains a formal preparation for the recipes, created with the culinary actions mentioned before, but they are not available in the XML-dump. Also they are often error-prone, as they are generated from text by a program.
In Taaable queries are executed by searching for recipes that match the query. If none can be found the query is generalised with help of the ontology until a recipe that fulfils the constraints is found. This so called *smooth-search* leads to the most specific generalisation of the query for which a result is known. A query may consist of a set of foods and entities of the other ontologies both, as excluded or included. Taaable then looks for recipes that contain or, respectively, do not contain these foods. For example a query may be: "A Chinese dessert with fruit but without ginger". It is queried for few ingredients wished. When a result is found by using a generalisation (e.g. *blueberry* has been generalised to *berry*) a recipe that fits the generalisation (*strawberry cake* contains berries) is first abstracted to the matching level and afterwards specialised to the ingredient that was actually queried.
Taaable is an active project of research and the involved groups are trying to improve the system from different angles. In Gaillard et al. (2012) they explore expert driven knowledge discovery for generic adaptations. Another try to improve results is improving the food ontology which is, in most areas, quite flat—for example garlic is on one level with chocolate. (Cojan et al. (2011); Cordier et al. (2009); Gaillard et al. (2012); Badra et al. (2008); Dufour-Lussier et al. (2011); Gaillard et al. (2011)).

CookIIS

CookIIS is another regular contestant in the CCC. They are a participant from the first time on. In the early years they built up their own ontologies and ran the CBR process on the professional tool *Information Access Suite (e:IAS).*

[6]`http://semantic-mediawiki.org/`

In 2012 They participated with an Android[7] based mobile app connected to a server handling the CBR cycle. The server now uses the open source tool myCBR[8] by the DFKI[9] and the School of Computing and Technology at University of West London. The tool has two parts, on the one hand it has a workbench for modelling and retrieval of knowledge-intensive similarity measures. On the other side it offers an SDK to integrate these into ones own applications. For substitutions CookIIS uses the *JBoss Drools*[10] rule engine. Note that substitutions are built from rules, not from cases. CookIIS has a very special data basis for the adaptation process: A German cooking portal was analysed and adaptation knowledge extracted from comments belonging to recipes. The system supports the full 4R cycle by using user feedback for revision and is also able to retain new cases. (Ihle et al. (2009); Hanft et al. (2009); Bach et al. (2012))

[7]Googles mobile phone operating system: `http://www.android.com/`, developer information & SDK: `http://developer.android.com`

[8]`http://www.mycbr-project.net/`

[9]Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, `http://www.dfki.de/web/welcome?set_language=en&cl=en`

[10]JBoss Drools: `http://www.jboss.org/drools/`

Chapter 3

Approach to Creating an Own Case-Based Reasoner

3.1 Motivation

In section 2.2.1 we have seen multiple contestants in searching for recipes by ingredients. All of these share the same entry point to retrieval: Search for single (or few) ingredients. In Yummly ingredients can be included or excluded, diets, allergies, recipe origin, and other constraints may be specified and the result will be a recipe which satisfies all these limitations. The same holds for Taaable and CookIIS. These approaches allow for great freedom in their results (e.g. any main dish that contains tomatoes and garlic is found) and take into account that one or more explicitly stated ingredients are desired and other likewise specifically stated ingredients are excluded. Therefore the motivation using these systems is knowing which ingredients a user wants to eat (maybe because an ingredient cannot be stored any longer, e.g. fresh meat), but not knowing a specific recipe. Contrasting the previous this work aims to provide recipes that are actually cookable with ingredients available in the shelf, fridge or anywhere else foods can be stored. The difference is that in the existing systems it might be necessary to go the supermarket after choosing the recipe, in this works approach this is not needed. From this approach a different constraint emerges: Every ingredient from a results recipe must be included in the query. This is an inversion of the existing systems constraint where every recipe must contain all ingredients from the query. Here no food not contained in the query must be used.

Consequently the results change: On the one hand the results seem to be bound to more restrictions—not being allowed to contain foods out of the queries scope. Nevertheless, on the other side very small recipes (e.g. egg dumplings consist only of salt, flour, water, and eggs) will be satisfied by nearly every query. One strategy to avoid these overly easy recipes is using the other constraints as well, a user who wants a main dish should specify this. As of the structure of queries described the focus of this work lies on adaptations to give the user more options and creative new variations to choose the definite recipe from.

In this chapter the overall system structure will be introduced and be filed into

Aamodt and Plazas CBR framework. Then the focus will lie on strategies and methods to create adaptations.

3.2 System Outline

The main part of this work is creating a Java[1] application that operates on WikiTaaables data and implements the case-based reasoning cycle of Aamodt and Plaza's framework. As case base and general knowledge the previously described contents of WikiTaaable are used and extended during program usage. A new case is defined by a so called *Query*, which contains available foods[2] and other constraints. The features of this case, most essential the foods, will be created from user input. Details about queries will be discussed in section 3.4. For retrieval all known recipes will be compared to the query, this is feasible as long as the data base does not get too big, otherwise indexing mechanisms were needed. A matching case then will be an existing recipe fulfilling the query as far as possible. If it needs adaptation possible substitutions will be created, which need evaluation by the user. Evaluation will be discussed in more depth in the next chapter. Once the user has found a recipe that satisfies his constraints and likings this new recipe as well as newly created substitutions will be added to the data base. Note the distinction between the problem description and the case created in the end. While the original 4R cycle visualises the problem as a new case and takes all its constraints into account during solving the problem and learning, the system built here will operate a little different from this. In the motivation we learned that the goal of the system is to create a recipe that is fully cookable with the ingredients provided. This in reverse means that not all foods from the problem description need to be contained in the solution. Therefore following holds: $foods(recipe) \subseteq foods(query)$. The ingredients of the solution form a subset of the foods in the query. The query itself will not be stored, it does not contain any useful information which is not contained in the (new) recipe. Nevertheless, all other constraints must be met by the result. From the previous formula and recipe retrieval techniques view storing and searching would unnecessarily restrict future queries. In other words the recipe is more general with respect to the retrieval mechanism.
From the simplicity of the data base integration of new knowledge is simple. New recipes will be added to the set of recipes. New substitutions will be added to the set of substitutions. The ontologies will not change assuming that no new foods will be entered. Adding a new food into its corresponding place in the ontology needs to be done before querying since otherwise the program can not take it into account without any knowledge about it. Therefore the general knowledge will not change, and no indexing needs to be done.

[1] `http://www.java.com/`

[2] When referring to a food by itself, it is called food. When used in context of a recipe it is called ingredient

3.3 WikiTaaable Data as Case Base and Background Knowledge

As of the vast amount of work that has been put into WikiTaaable, their data has grown from the simple XML-representation of the CCC to a highly annotated Wiki with interconnected items. Therefore it makes sense to use these resources to build an application upon (other contestants in the CCC profit from this as well). The data of WikiTaaable is stored in a Semantic MediaWiki[3] and available for download as an XML or RDF dump[4]. For this work the XML dump is used. Note that the dump is not complete, the texts of the recipes are not contained in the dump hence they are not used in this work. However, adaptation of the text is a more or less trivial process of replacing the old ingredients names with the those of the substitutes assuming the ingredients preparation does not change. Examples of the data of WikiTaaable can be seen in appendix A

3.3.1 Recipes

WikiTaaable has annotated the recipes of the CCC with a whole lot of information. Most importantly the ingredients, they are all linked to food objects in the ontology, their amounts are measured with the corresponding unit, and in some cases their condition is also stated. The ingredients are stored in so called “ingredient lines”, each of those may have properties as shown in table 3.1.

has_Ingredient_line	
Ingredient	Tomato (link to the food)
Ingredient_qualifiers	whole, finely chopped
Ingredient_quantity	1
Ingredient_unit	lb
Ingredient_text	1 lb Whole tomatoes, Finely chopped

Table 3.1: An example ingredient line showing possible properties of an ingredient line in a recipe.

Recipes may also explicitly define the following three properties:

- Has_origin (e.g. Italy, India.)
- Produces_dish_type (e.g. Baked_good_dish, Cookie_dish. These are links in the ontology. More than one are allowed.)
- Can_be_eaten_as (e.g. Side_dish, Main_dish. These are also links in the ontology. More than one are allowed.)

Other properties are given implicitly by the used ingredients. For these see the section about the foods (section 3.3.2). Recipes are not interconnected other than by sharing properties. A sample recipe in its XML form is shown in appendix A.1

[3]http://semantic-mediawiki.org/
[4]http://wikitaaable.loria.fr/index.php/Main_Page#Dumps

3.3.2 Foods

Besides knowing recipes, the most important part of WikiTaaables data is the food ontology. Describing 2154[5] nodes it is noticeably extensive. The root is called *Food*, it has 19 subcategories each of which has subcategories on its own. A node may be subcategory of one or more categories. An extract of the tree is shown in figure 3.1. The graph only shows a small fraction of the foods actually contained in it, but we get a good impression on the graph growing in both directions breadth and depth, and containing many different foods specified in detail. What is not obvious in the graph but indicated is that some relations are missing in the actual WikiTaaable data. Flank steak, Lamb breast, or Veal breast are some examples which *subClassOf* relations are not modelled exhaustively. They are not connected to the animal they belong to.
Foods are also marked with dietary restrictions such as *veganism* or *gluten-free*. These restrictions can be of two kinds, either the food is compatible with a diet, or it is not. This does not mean that a food has one of these properties for each diet, though. For example *Chicken breast* is marked as incompatible with *Cholesterol diet*, *Veganism*, and *Vegetarian*, but it is not stated that it is suitable for a *Nut free* diet. Therefore these diets are not necessarily safe for use. Recipes can infer dietary restrictions by its ingredients. If one ingredient is not suitable for a diet, the recipe will not be either. If an ingredient is suitable for a diet and no other ingredient of the recipe states it is not suitable for that same diet then the recipe is suitable for the diet.

3.3.3 Dish Type Ontology

The dish type ontology features 14 subcategories on the first level, 79 types are defined overall. Some are final and unambiguous from the first level, e.g. *Crepe dish*, *Mousse dish*, *Pancake dish*, or *Waffle dish*. While others, say *Baked good dish*, *Salted dish*, and *Sweet dish* have up to 16 subcategories. In the same manner as in the food ontology one category in the hierarchy may have multiple parent categories. A *Jam dish* has the superclasses *Preserve dish* and *Sweet dish*.

3.3.4 Other Features

The *dish role* ontology has seven categories, of which only one, namely *Snack and appetizer dish*, has two subclasses—*Snack dish* and *Appetizer dish*. As of this flat hierarchy this work will not take it into account as hierarchy, but only use it as a list.
Diets contains seven categories, each final. Foods are marked with these, see also section 3.3.2.
Localisation is another hierarchy available enabling recipes to specify their origin. It is sorted by continent and country. Unfortunately the countries are not specified more precisely to model local differences. Nevertheless, this ontology is not used in this work.
A very interesting hierarchy is the *Culinary actions* ontology describing actions that can be applied to foods and combinations of multiple foods within a recipe. Many recipes are enriched with these modelling their textual representation of

[5] A few of these are only for testing purposes

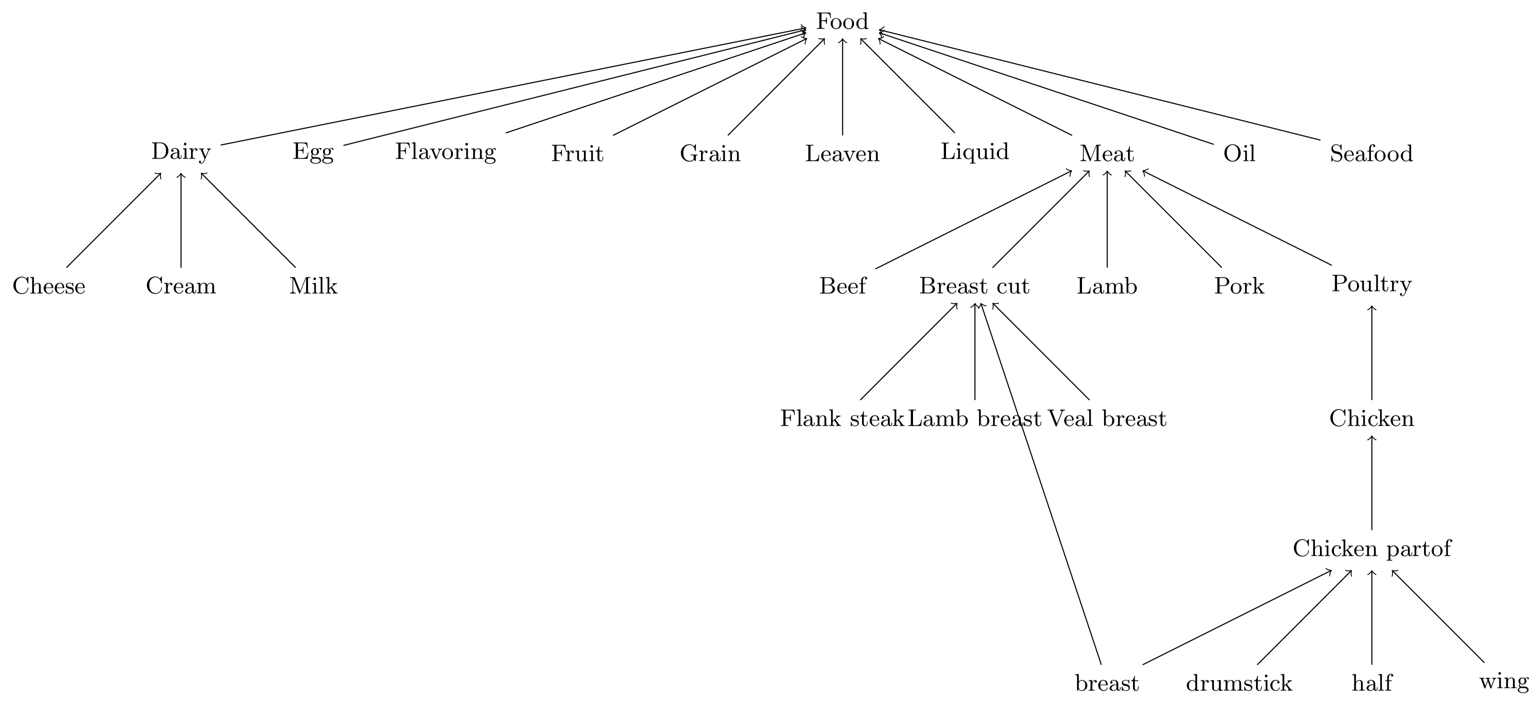


Figure 3.1: An excerpt of the food ontology in WikiTaaable also used in this work. We can see the deeply elaborated structure, but also can see problems, e.g. the veal breast and the lamb breast are not part of the hierarchy corresponding to their animal. The arrows show the *subClassOf* relation.

the cooking instructions in a computer readable format. As of high complexity this is not used in this work either.
The WikiTaaable XML-dump contains 53 substitutions, they are distinguished between generic and specific substitutions. While the generic only contains one of these, all others are specific substitutions. They are bound to the recipe they were used in. As they do not seem that well elaborated and the focus of this work is adaptations, these are not used in this work.

3.4 Recipe Retrieval—Queries

The first step in the 4R-cycle is retrieving cases. As stated before cases are resembled by recipes. The following introduces the queries and explains the process of finding a match. When querying the database for a recipe, a query contains the following information:

- Available foods
- Desired diets
- Forbidden diets
- Desired dish role
- Desired dish types

Where all but *Available foods* are optional. *Desired dish role* is of cardinality 0..1 while all others may be of arbitrary size. Obviously *Desired diets* must be disjoint from *forbidden diets.*
Of course **available foods** are the most important of these. The system will try to find a recipe that can be cooked with the given ingredients, at best all ingredients from the recipe are available—the recipe can be cooked without any adjustments (which of course is not desirable for demonstration purposes of the program). While the other items are strong constraints, say they must be satisfied when specified, the available foods are weaker constraints. It is not needed that all of them are used, but it is not allowed to use a food which is not specified. This will not be the case for most recipes and hence is the reason for the case based reasoning approach of this work. It is likely that some basic ingredients are available in the cupboard and fridge and maybe something special and a whole bunch of spices, but the things do not really seem to fit. Does this situation happen on a Sunday when the stores are closed, or unexpected guests are visiting, a new recipe is needed. The program then will search for recipes that can be fulfilled by their maximum part with the available foods. When querying, the recipes can be sorted by their fitness for the query which simple version is defined as follows:

$$fitness(recipe, query) = \frac{|foods(recipe) \cap foods(query)|}{|foods(recipe)|}$$

The formula returns the percentage of the recipe that can be fulfilled with the queries foods by dividing the number of foods contained in both recipe and query by the number of foods in the recipe. To reduce the output a threshold value is used that cuts out all recipes below the value. This measurement has its weaknesses, a recipe with only four ingredients and three of these available in the query—only one ingredient needs to be replaced—is rated 0.75 and therefore worse than a recipe with ten ingredients and three items missing(0.7). It might be harder to find three good substitutions, that even suit each other, than just one. But for now it is assumed that most recipes consist of more than four ingredients, as for example salt and pepper are very common and would already provide half of the ingredients. All other constraints (desired diets, forbidden diets, desired dish role, and desired dish types) are conditional filters that allow the user to specify his wish. In contrast to other contestants the system described here does not take a field for forbidden ingredients into account. It is not needed, as an unwanted ingredient simply would not be put into the set of available foods.

3.5 Adaptations

If $fitness(recipe, query) = 1$, say

$$\forall food \quad isIngredient(food, recipe) \rightarrow isAvailable(food, query)$$

a recipe can be used directly. In every other case adaptations need to be done. This section will introduce methods for creating new adaptations. Starting with the basics, adaptations replace a recipe's ingredients with different ones. This is done when an ingredient is not available in the query. Using other foods contained in the query these can hopefully be replaced appropriately. This also is the mechanism creating new recipes in this system (problems that arise from this will be discussed later). At the simplest level adaptations are unary. One ingredient can be replaced with another. For example “basmati rice” can directly be replaced with “parboiled rice”. Nevertheless often this is not enough. Ingredients sometimes have a more connected relationship. Say for example lamb is often accompanied by rosemary. So if lamb is not available it is likely that you might want to change the used herb as well. In some cases it might even be interesting to leave ingredients or parts of those combinations out or on the other side add one more. So we do not only have $1 : 1$ substitutions, but $2 : 2$, $n : n$, and even $n : m$ as well.
From this complexity also comes the difficulty in finding good adaptations. As of the difficulty the higher order substitutions will only be available for manually creating adaptations. The other adaptation generation strategies will only be able to create 1:1 substitutions. Finally a metric is introduced trying to measure how good a substitutions fits to a recipe.

3.5.1 Finding Matching Adaptations

When a recipe is not completely satisfiable with the available foods it has to be adapted. The first step in the recipe modification process is to find possible adaptations. An adaptation has two sides, the foods from the recipe that are to replace and their substitutions. We need both of these to find potential repairings for a chosen recipe. As we have seen before it is tried to find a recipe that can be cooked to the greatest extend possible, or, vice versa, it is desirable to replace as few ingredients as possible. So from the set of all known substitutions we need to find those that are applicable. A substitution fits if it satisfies the following conditions:

$$\begin{aligned}
&\forall substitution, query, recipe \quad matches(substitution, recipe, query) \Leftrightarrow \\
&\forall food_0 \quad toReplace(food_0, substitution) \rightarrow (isIngredient(food_0, recipe) \wedge \\
&\forall food_1 \quad replacement(food_1, substitution) \rightarrow isAvailable(food_1, query) \wedge \\
&\quad \exists food_2 \quad toReplace(food_2, substitution) \rightarrow \neg isAvailable(food_2, query)
\end{aligned}$$

Or in words, a substitution matches the recipe-query-combination if all foods on the substitution's toReplace-side are parts of the recipe but any of these is not available in the query. Also all substitute foods need to be available in the query.
There are two sources to get these adaptations from. The first is maintaining a list of known substitutions. They may either just be given from the background knowledge, or even linked to recipes they have been successfully used with. If no useful one can be found new adaptations need to be created.

3.5.2 Manually Created Adaptations

The most trivial way of creating new adaptation is creating them by an experts hand. In the same way a solid basis of recipes is given it is possible to provide the program with a number useful substitutions. These substitutions have one enormous advantage: They are already checked by the expert and therefore more likely to be of good quality. Also they can easily be created in $n : m$ arity. Expanding the lamb example from earlier, substitutions like

$$\{\text{lamb, rosemary, red wine}\} \leftarrow \{\text{beef, bay leaves, dark beer}\}$$

can be added by a human. On the other hand manual adaptations also have a disadvantage: They are not bound to cases where they have been successful. This is an information that could be useful for deciding how good a substitution fits to a recipe (see also section 3.5.4).
A general problem of adaptations is their reference to concrete ingredients. The following example will point this out: Say we want to cook a lasagna, the recipe states that it should be gratinated with parmigiano-reggiano cheese. Unfortunately we do not have parmigiano-reggiano in our fridge. There only is some mozzarella. Now, this is a pity, we cannot use the mozzarella, because the recipe says parmigiano-reggiano—of course we can use it: Just apply the adaptation $parmigiano - reggiano \leftarrow mozzarella$. Two weeks later we again want to cook lasagna (it was delicious, wasn't it?), but today we neither have parmigiano-reggiano, nor mozzarella. Just some slices of gouda—OK we can use that as well. The new adaptation is $parmigiano - reggiano \leftarrow gouda$. We could continue this example to a nearly arbitrary amount of different sorts of cheese and come up with a whole lot of adaptations. But we can not add such an amount of adaptations for every ingredient by hand! Therefore we now explore the possibility of creating adaptations automatically.

3.5.3 Generating Adaptations from Food Classes

In the last section we learned that we cannot create all possible substitutions by hand but need to be able to create them when needed. Therefore we now need strategies allowing the computer to generate the needed substitution during a running CBR-process. First lets recapitulate the information sources available:

- Recipe to be adapted
- Query and all its data
- Food ontology
- Other ontologies
- Other recipes

The most promising source is the food ontology. From WikiTaaable's ontology of more than 2000 hierarchically structured items we can design a method to create substitutions. The actual ontology's structure will give us the possibility to find substitutions, however simultaneously impose its problems to our algorithm. We have seen before that the ontology introduces abstract foods as for example *berry*, which may be part of a recipe, though. *Fruit* is part of seven recipes. There are even *semi-abstract* foods used more frequently than their more differentiated children. A prominent example is *oil*, which is used in 86 recipes, while e.g. *olive oil* and its even more differentiated children count 43. In most cases *oil* is just enough for the cook, with his knowledge he can imply something like *neutral in taste* and choose an appropriate oil by himself. If a special taste is proposed by the recipe it states the explicit kind of oil to use. From the cheese example in section 3.5.2 we can tell the need to somehow traverse the food ontology to see connections between different foods. Obviously we do not need to traverse the full tree, an apple can not replace a cheese at all. To decide if one food can replace another we need to introduce a measurement.

Ingredient-Ingredient Similarity

Knowing if two ingredients are quite similar is key to finding new substitutions. To measure the similarity of ingredients we can use the structure given from the food ontology. At first it seems to be a good idea to measure the distance between the two nodes in the graph. Algorithms like the Dijkstra algorithm can find the shortest path between two nodes. This comes with high costs, as the database contains thousands of foods. Searching this tree is not feasible especially as foods that are far away from each other hardly can be good substitutions for each other. The search needs to be bound to feasible distances, reasonable for the cooking domain. Also direct distance measures are not meaningful, both the distances from *Red onion* to *Sugar* and from *Red onion* to *Shallots* are the same, namely 2. Therefore a measurement that takes the structure of the ontology into account is needed. Two approaches to this will be introduced and compared next.

HasDescendant

`HasDescendant` is a property describing a relationship between two nodes. A food has another food as descendant if the other one is a child of the first, or descendant of a child. The following equation describes this recursive predicate more formulaic:

$$\forall food_p, food_d \quad hasDescendant(food_p, food_d) \Leftrightarrow hasChild(food_p, food_d) \vee \\ (hasChild(food_p, o) \wedge hasDescendant(o, food_d))$$

UpwardsDistance

We have seen the structure of the food ontology (see figure 3.1), we need to work on this hierarchy to create meaningful measures. The idea of `upwards Distance` is derived from the iterative deepening search (for example described by Russell and Norvig (2010)). In iterative deepening a depth-first search is bound to search only down certain levels in the problem tree. The search is split into iterations of depth n, starting with $n = 0$. When in one iteration of searching the target node cannot be found the boundary is increased so that the tree will be searched one level deeper. This is repeated until the goal node is found.
The upwards distance measure introduced here has two parts. The first iteratively checks the parent classes of a food node. The other checks if the target node is contained in the subtree of a node. Therefore at $n = 0$ it is checked if $food_b$ is a descendant of $food_a$. Since we are interested into the shortest distance this step is also done inverse—is $food_a$ descendant of $food_b$? If not repeat the descendant check for all parents of both nodes. The algorithm needs to stop when the iteration hits the root node of the hierarchy as all foods are descendants of it, a distance measurement does not give any information in this case. Instead the two items to be tested are too distinct from each other. The algorithm returns the number iterations needed to find the path between the nodes, only steps up in the hierarchy are counted hence the name *upwards distance*. When downward steps are also taken into account the following problem appears: The recipe requests *Balsamic vinegar*, it is not available, but *Red wine vinegar* is, as well as *Water*. As the red wine vinegar is child of *Wine vinegar*, which in return is, the same as balsamic vinegar, child of *Vinegar*. One level above this is *Liquid*, which also contains the water. The distance to water then is `3`

$$\text{Balsamic vinegar} \Rightarrow \text{Vinegar} \Rightarrow \text{Liquid} \rightarrow \text{Water}^6$$

and the distance to red wine vinegar is also `3`:

$$\text{Balsamic vinegar} \Rightarrow \text{Vinegar} \rightarrow \text{Wine vinegar} \rightarrow \text{Red wine vinegar}$$

This part of the food ontology is shown in figure 3.2 When not counting downward steps the steps upwards are actually weighted more heavy, at last a child actually *is* an instance of an upper level as of transitive inheritance (this also implies that any descendant of a node is a good substitution for this node).

Same Tribe

The same tribe relation is a simplification of the upwards distance measurement. Instead of increasing the step width in iterations it is bound to do one step in the same manner. The algorithm checks whether two foods are related by any of their parents. $Food_a$ hence is a descendant of any of $food_b$s parents, or vice versa. This greatly simplifies the computation and gives a restriction on relationship. It is supposed to stop the algorithm from searching too deep quickly losing context of the original food. Its usefulness will by analysed in chapter 4. This restrictions nevertheless bear drawbacks when operating in deep structures. It will not be able to find substitutions for *Chicken breast fillet* other than *Chicken breast half* and *Chicken breast quarter*. Figure 3.3 shows

[6] $\Rightarrow$ indicates a step upwards in the hierarchy while $\rightarrow$ moves to a child.

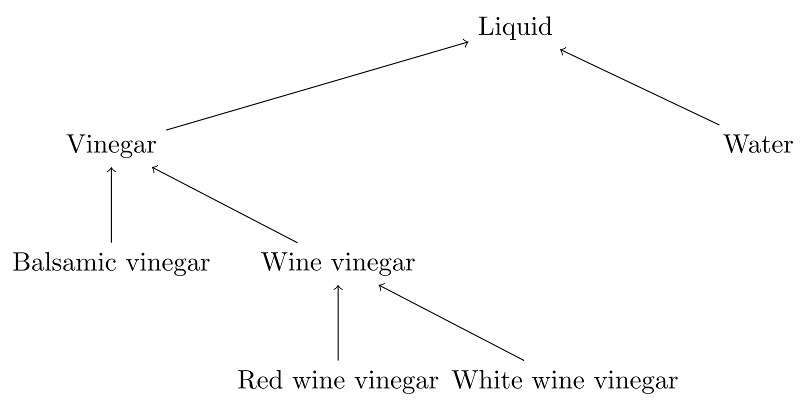


Figure 3.2: Local ontology around *Balsamic vinegar*.

the local structure of the ontology. The same tribe relation will not visit the nodes *Chicken partof* and *Breast and flank cut* the upwards distance does. The algorithm has advantages and drawbacks, as we have seen, the evaluation will show how satisfying the results are and how quick similarity of foods degrades when moving the hierarchy upwards. Another modification of this is also under discussion namely an asymmetric variant only comparing the substitute with descendants of the original ingredients parents.
An interesting extension to this could use the knowledge of Foodpairing to measure the relations by taste of the ingredients instead of the "subclassOf" relations used in the WikiTaaable ontology. Substitutions like *vinegar* ← *lemonjuice* could be found easily with knowledge about taste, which is far outside of the capabilities of the used ontology.

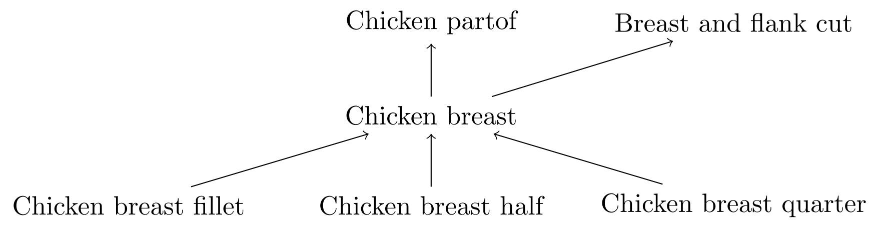


Figure 3.3: Local ontology around *Chicken breast*.

3.5.4 Fitness Criterion

In the previous section a method to find substitutions was introduced. But it only focused on finding substitutions that fit the ingredient to be replaced. Since it is the recipe that needs to be adapted a new measurement to decide how good the adaptation fits the overall recipe may be useful. While we used the food ontology to create the adaptation the validation of it can take the other existing recipes into account adding more data to the decision process.

Direct Recipe Similarity
The first step is to find other recipes that we can use for comparison. One way is finding recipes that contain both the ingredient to be replaced and its replacement. Though, this is just a validation that two ingredients match together and not necessarily a sign that one is a good substitution for the other. The more elaborate way is finding recipes that are to a certain amount similar to the one to adapt and check if the replacement is contained in these. This is an indicator that the surrogate fits to the recipe. In this case the other recipe does not necessarily contain the food to be replaced. The similarity of two recipes can be measured with the *Jaccard similarity coefficient*:

$$similarity(recipe_1, recipe_2) = \frac{|foods(recipe_1) \cap foods(recipe_2)|}{|foods(recipe_1) \cup foods(recipe_2)|}$$

The main drawback of this method is not taking any of the given ontology into account. This is a problem especially due to the ingredients stated in the recipes, many of which are not stated as leaves in the hierarchy thus not unambiguous to the algorithm. A human knows which one to take when the recipe states *Onion* and will adapt automatically if no other than *Red Onion*s are available—which are perfectly fine. Therefore another measurement is proposed utilising the ontology's knowledge.

Ontology Supported Recipe Similarity
Advancing to solve the above issue the food ontology is used to improve the similarity measure. The basic principle of the Jaccard similarity coefficient is retained, only the fractions numerator is changed, hence the decision whether an ingredient is available or not. As stated earlier the ontology is hierarchic and therefore every food is also an instance of its parents by inheritance. Instead of the simple intersection in Jaccard's formula, the set in the numerator is constructed having the inheritance in mind by using the `hasDescendant` relation. Hence recipes are considered similar if both share the same ingredients or at least ingredients with certain similarity as defined earlier.

Evaluating the Substitutions Fitness With both formulas we can sort all recipes respective to the one to adapt and compare the substitution to the nearest neighbours. In the next step, to check if the substitution under review fits the original recipe, we can check if the surrogate is contained in any of the nearest neighbours. If this is the case, then the food is suitable in combination with other ingredients contained in the original recipe.
When comparing multiple possible substitutions in order to use the best different strategies may be used. The first is checking which replacement is used first when traversing the sorted list of nearest neighbours. Another strategy takes a range of those nearest neighbours (say a specific number of neighbours, or recipes to a certain degree of similarity) and chooses the one substitute which is used more often in those. In the case that these strategies do not state an outright winner, that is either both substitutions are equally good or, in the worst case, none of these are used in any nearest neighbours, choose one of multiple arbitrarily (at least taking the distance measure of section 3.5.3 into

account).
As a step further, when adaptations are used regularly, they can hold references to the recipes they are used in. These references give us more information on how good a substitution is. If it is used more often or to more similar recipes, then it is a better substitution. On the other hand we can reduce the number of recipes the one to adapt is being compared to by only comparing the referenced. For the latter an adaptation needs multiple references so that the metric does not get trapped in these recipes.

3.5.5 Generalised Substitutions

When manually creating adaptations the biggest difficulty is reaching completeness, say there exists a substitution a human knows of which may be applied, this one is not necessarily contained in the data base. This arises from the complexity of foods and the reference constraints of adaptations introduced in 3.5.1. Lets take another look onto the lasagna—cheese problem from section 3.5.2. We had two possible variants there as replacements of parmigiano-reggiano: mozzarella and gouda. Emmentaler, cheddar and many other kinds of cheese may be used as well. Now from these five kinds of cheese 20 substitutions can be created when assuming the substitutions are invertible. Writing these by hand, say for each ingredient may have the advantage that all of them are "checked" by an expert, but they are both, not connected to any recipe in the beginning, and thousands of possible adaptations to write. The idea to cope with the latter issue was creating adaptations that use the ingredients parents. In WikiTaaable cheese is partitioned into 14 subcategories like *blue cheese*, *firm cheese*, *fresh cheese*, *semi-firm cheese*, *soft cheese*, or *soft-ripened cheese*. For example *gouda* is within *semi-soft cheese* and a category called *cooked pressed cheese*, while *gouda* itself is split up into four subcategories. Mozzarella is filed under *semi-soft cheese* and then subcategory *pasta filata*, *emmental* and *cheddar* are *semi-firm cheeses parmigiano-reggiano* is filed under *firm cheese*. An extract of the cheese structure can be seen in figure 3.4. *Cheese*, though being a very abstract description, there is no cheese that would just be distinctively called "cheese", is used in many recipes directly as well. Now we create the following substitutions:

- parmigiano-reggiano ← aged gouda (assume here that the actual queries gouda is specified in that exact manner)
- parmigiano-reggiano ← mozzarella
- parmigiano-reggiano ← emmental
- parmigiano-reggiano ← cheddar

Concluding these substitutions taking the structure from figure 3.4 into account, straightforward we may come up with the following:

- parmigiano-reggiano ← semi-soft cheese[7]
- parmigiano-reggiano ← semi-firm cheese
- parmigiano-reggiano ← cooked pressed cheese

The semi-soft cheese substitution would allow *gouda* and *mozzarella* to substitute the *parmigiano-reggiano*, while *semi-firm* allows *emmental* and *cheddar*.

[7]Note that for this step we already abstracted one layer of the mozzarella's hierarchy

The last one, the *cooked pressed cheese* substitution, is very interesting as it allows for *emmental* and *gouda* to be surrogates. The structure shown in figure 3.4 is a good example for foods being categorised under multiple different concepts. While *semi-firm* and *semi-soft* are descriptions of the cheeses condition, *cooked pressed* is a hint on their production process. This multi-categorisation allows for better differentiated, though still generalised, adaptations. It is out of scope of this work, but generalisations like this are an opportunity to establish a case-based reasoning cycle only for managing substitution purposes. Other machine learning principles could also tackle this point, for example concept learning. Nevertheless, for other learning techniques to work the background knowledge needs to be more detailed, other predicates for the foods are also a topic for further investigation. Within the cheese domain one predicate may be *can be used to gratinate*, for vegetables it may be interesting if they are suited for a soup. Maybe these predicates should even be linked to other concepts like *origin*, the understanding of ingredients suitable for a Chinese soup might be different to that of those suitable for a German one.
General adaptations in the other direction will appear quite often in the system of this work. These will specialise foods that are stated in a general form to those actually available. Say the recipe lists *cheese*, then the adaptation will substitute *cheese* by *aged gouda*, as the query should never contain any general food description.
When working with higher arity substitutions generalisation needs to be done carefully, otherwise the connection between the stated ingredients gets lost. Once again remind the lamb and rosemary substitution. Say a recipe states it uses a *lamb tenderloin* as well as *rosemary.* A substitution replaces both with *pork tenderloin* and *tarragon.* Another substitution used in a different recipes replaces *lamb spareribs* and *rosemary* with *pork spareribs* and *tarragon.* The generalisation results in $\{lamb, rosemary\} \leftarrow \{pork, tarragon\}$. Up to this point generalisation works fine. Now there exists another substitution $\{beef, parsley\} \leftarrow \{pork, sage\}$ and these two might be generalised to

$$\{meat, herb\} \leftarrow \{pork, herb\}$$

In this case the information of a connection between the ingredients is lost.

3.5.6 Problems

Creating substitutions is a very complex task. In this chapter some strategies were introduced, while always trying to mention their problems as well. This section will try to raise awareness for more problems regarding the previous approaches.
We have seen unary adaptations working well. We can also create adaptations on higher hierarchy levels. When coping with $n : m$ adaptations different problems arise. On the one hand growing complexity when creating adaptations leads to too many possible adaptations. The complexity shows in both cases, when manually creating adaptations before in need and automatically creating in place. In the first case we have the advantage of knowing combinations that belong together, nevertheless there are still too many of them to efficiently add all possible adaptations, yet even know all of these. In the latter the advantage of background knowledge is not present, the algorithm can hardly figure out

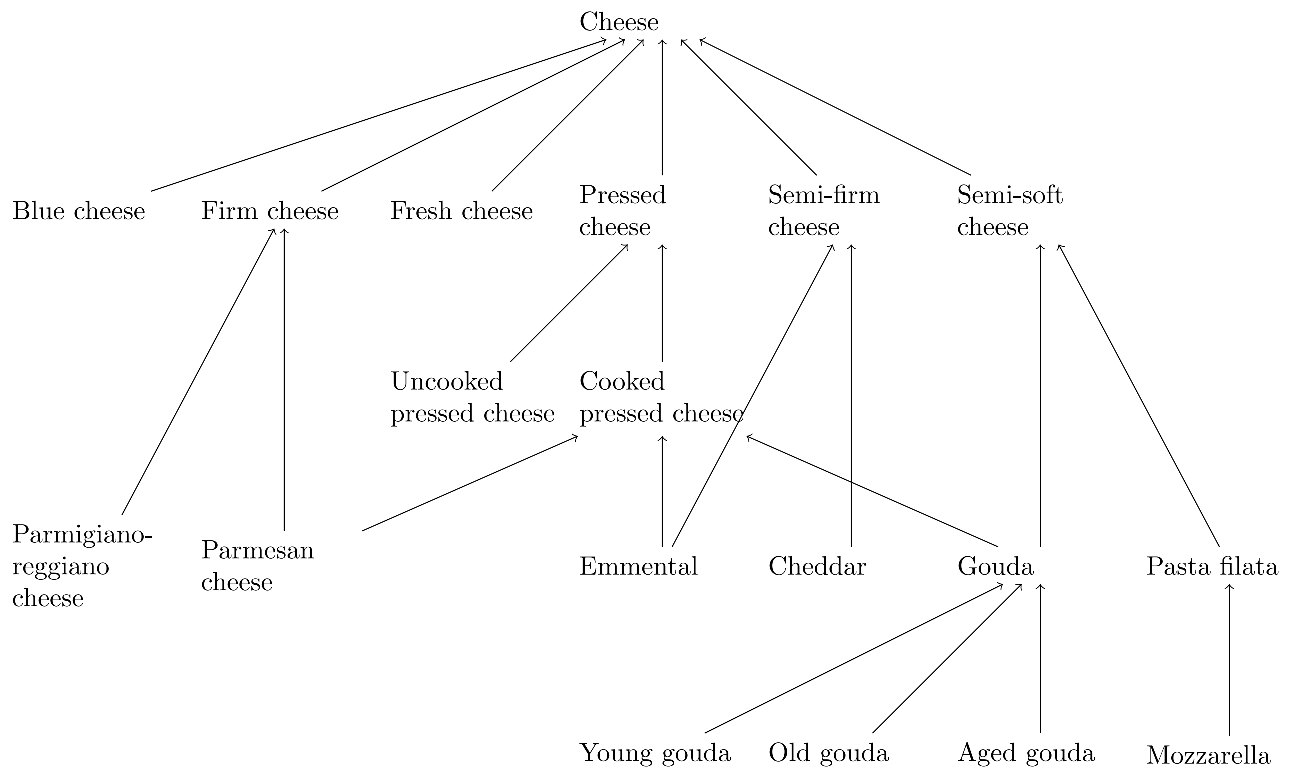


Figure 3.4: Excerpt of the local ontology below *Cheese*.

two components that belong together and should be replaced together. This is even more the case when one of those two is available in the query and therefore replacement is not inherently necessary. We can observe two tendencies for higher arity adaptations from this: Higher arity allows more different possible combinations on both sides: let p be the number of foods in the ontology, in $n : m$ adaptations then $\frac{p!}{(p-n)!} * \frac{(p-n)!}{(p-n-m)!}$ adaptations can be created. For 50 foods 5527200 distinct 2 : 2 adaptations are possible. Of course most of these do not make any sense, which leads to the other tendency. Only very few adaptations of higher arity are useful. There may be no use for any 6 : 6 adaptation at all, unless it is supposed to create a thoroughly new recipe. In that case we will struggle with the preparation steps. Note that for adapting the preparation text when using $n : m$ adaptations links are needed that exactly describe which foods replace which missing ingredients.
Another problem is that the food ontology is not equally well elaborated in all its subcategories. We have seen an example in the meat ontology where many items are defined, for example *Turkey breast* is defined in *Breast and flank cut*, unfortunately not linked to *Turkey* within the *Poultry* domain, though. More fine grained hierarchies are another important step in improving the performance of adaptations, as these heavily rely on the modelled knowledge. In a first step expanding the existing ontologies such as *Breast and flank cut* and therefore building up more interconnected nodes with multiple parent nodes is desirable. Then other categories, for example *Usable for roast* or *Usable in soup* may be of interest.
It should also be pointed out that WikiTaaable's food ontology is not complete (of course it is most likely that it never will be), for example *Pea* is not included though used in some recipes. Likewise some recipes state ingredients that are linked to other foods, some are just linked to the wrong one[8], some foods do not exist specifically in the ontology hence their would-be parent is linked[9], and for some not even all stated ingredients are differentiated[10]

3.6 Revision

The system built in this work does not have a model it could test its results on. The software introduced in the next chapter will enable the user to evaluate the results and let him choose which recipes and adaptations are being used. Therefore new recipes and adaptations may be treated as validated by an expert.

3.7 Storage

Currently storage of cases is very basic. New recipes are simply added to the set of known recipes. This also holds for substitutions. The latter ones store more information, though. Whenever a substitution is used, be it a known one or a newly created during adaptation, it adds the original recipe it was used on as a reference. Those references can later be used to take a measurement of how

[8]Antipasto potato salad uses *purple onions* but these refer to *Onion*.

[9]Broiled bluefish sauce states *Pepper sauce, hot, liquid*, but *Liquid* is referenced. *Liquid* is far too general, but no better parent is available.

[10]In beef cheese ball only one ingredient line is stated, the text states more. There seems to be an error in some automatic processing.

good a substitution fits to another recipe.
Since the other ontologies will not change there need not be any modification here. Furthermore changing the ontologies, especially the food ontology is not a trivial task, it should only be done with lot of consideration. To be able to do this in an easy, human understandable way the Taaable project introduced the Semantic MediaWiki as storage for its ontologies.

3.8 Conclusion

This chapter described in detail the contents of a WikiTaaable XML dump used as data base in this work and introduced the methods used for creating a CBR cycle. The emphasis was on adaptations where strategies for both, creating and validating substitutions, were presented. An outlook to more advanced methods on adaptations especially regarding generalisation was given and finally problems discussed. From the foundation of this chapter the next one will describe the Java program built and evaluate the presented adaptation strategies.

Chapter 4

Realisation and Evaluation

The practical part of this work is a Java implementation of the case-based reasoning cycle in the cooking domain based on data from the WikiTaaable project. This chapter will describe the mechanics used in the implementation in detail. The data structures will be introduced and algorithms explained.

4.1 Data Structures

The algorithms used in the program rely heavily on the characteristics of sets, say a set never contains more than one instance of an element. Therefore in most cases the data structures are constructed of numerous sets. Of course the data base should not contain a single food more than once, neither a recipe or any other of the unique concepts.

4.1.1 Datastore

An instance of `Datastore` contains all knowledge of the CBR system. It is the central access point to get references to the basic types such as `Diet`, `DishRole`, `DishType`, and `Food`, as well as the types consisting of composition of these, namely `Recipe` and `Substitution`. WikiTaaables XML dump uses URIs to reference unique objects within the ontology. The data has been parsed into Java objects resembling the interconnected items. An object describing e.g. one food should therefore be unique, where ever it is used references to the original object must be used. This is crucial especially for foods, hence a method to retrieve a `Food` object from the datastore based on its name is provided. For the learning process new knowledge must be added later on. For this purpose methods exist to add new recipes and substitutions. The method `knowsRecipe` is a convenience for creating new substitutions preventing a name from being used twice. `Datastore` is also used to serialise the data to the hard disc. Every class used in this thus must implement the `Serializable` interface of Java. The interface offers methods to access all sets of data:

- `getDiets():Set<Duet>`
- `getDishRoles():Set<DishRole>`
- `getFoods():Set<Food>`
- `getRecipes():Set<Recipe>`

- `getSubstitutions():Set<Substitution>`
- `knowsRecipe(String):boolean`
- `addRecipe(Recipe):boolean`
- `addSubstitution(Substitution):void`

4.1.2 TreeItem

`TreeItem` is an abstract class modelling the hierarchic structures of WikiTaaables ontologies. It allows descendants to use `parent` and `children` relations enabling the algorithms to traverse the ontologies. As each element of the ontologies has a name (in WikiTaaable called `label`, compare appendix A as well as lexical variants and names in other languages it offers the fields `name` and `lexicalVariants`. While `name` is mainly used for displaying an object consistently with the same name every time the program runs, other algorithms may also work on `lexicalVariants` (`findFood()` does that). As of this, when creating a new food, the name should also be put into the field `lexicalVariants`. A recursive search strategy is also implemented in the class. The method *hasDescendant(TreeItem)* checks whether or not another TreeItem is in the hierarchy below the current one by recursively checking its children for equality. The implementation is shown in listing 4.1
The interface offers the following methods:

- `getChildren():Set<TreeItem>`
- `getParents():Set<TreeItem>`
- `getName():String`
- `getLexicalVariants():Set<String>`
- `setLexicalVariants(Set<String>):void`
- `hasDescendant(TreeItem):boolean`

`TreeItem` is extended by `Food` and `DishType`. While the latter one adds no functionality `Food` offers two additional fields, namely `compatibleDiets` and `incompatibleDiets`. As stated in section 3.3.4 the dish roles and the diets are not used as hierarchic ontology and therefore not extensions of `TreeItem`.

```java
public boolean hasDescendant(TreeItem other) {
  if (getChildren().contains(other))
    return true;
  for (TreeItem item : getChildren()) {
    if(item.hasDescendant(other))
      return true;
  }
  return false;
}
```

Listing 4.1: The hasDescendant(TreeItem):boolean method of `TreeItem`.

4.1.3 Recipe

A `Recipe` is a composition of its ingredients and other attributes. As of this it holds fields providing access to the instances of `Food` used, the `DishTypes`, and

`DishRoles`. Another utility method is `percentageCookableWith( Collection <Food> )` calculating to what percentage the instance's `ingredients` field is similar to the parameter's collection. The method `satisfiesConstraints(...)` returns true if the objects `dishRoles`, `dishTypes` and its `ingredients` `Diets` fulfil the given constraints. The method takes two `Collection<Diet>` as parameter as this constraint may be used in both directions, *must fulfil* and *must not fulfil*. The `Recipe` interface has the following methods:

- `getName() : String`
- `getDishTypes() : Set <DishType>`
- `getDishRoles() : Set <DishRole>`
- `getIngredients() : Set<Food>`
- `percentageCookableWith(Collection <Food>) : double`
- `satisfiesConstraints(Collection <Diet>, Collection <Diet>, DishRole, Collection <DishType>) : boolean`

4.1.4 Query

As we learned from section 3.4 a query has the following fields: Available foods, desired diets, forbidden diets, dish role, and dish types. In the Java implementation the `Query` class presents `get-` and `set`methods for all these:

- `getDishTypes():Set<DishType>`
- `setDishTypes(Set<DishType>):void`
- `getDishRole():DishRole`
- `setDishRole(DishRole):void`
- `getAvailableFoods():Set<Food>`
- `setAvailableFoods(Set<Food>):void`
- `getWhishedDiets():Set<Diet>`
- `setWhishedDiets(Set<Diet>):void`
- `getForbiddenDiets():Set<Diet>`
- `setForbiddenDiets(Set<Diet>):void`

4.1.5 Substitution

Substitutions need to store the foods they are to replace and the corresponding surrogates. Therefore any instance of `Substitution` should fill the fields `toReplace` and `replaceWith`. When they have been successfully used on a recipe the adapted recipe is also stored, to do so `Substitution` has the field `recipesUsedIn`. The following methods are offered:

- `getToReplace():Set<Food>`
- `setToReplace(Set<Food>):void`
- `getReplaceWith():Set<Food>`
- `setReplaceWith(Set<Food):void`
- `getRecipesUsedIn():Set<Recipe>`
- `setRecipesUsedIn(Set<Recipe>):void`

4.2 Metrics

From the data structures displayed in the previous section a case-based reasoning system is built operating on these. At different steps of the 4R-cycle measurements need to be taken to make decisions. This section discusses the measurements for food similarity, which is needed to find new substitutions and as basis for higher calculations, and recipe similarity, which acts as foundation for the substitution fitness calculation. The latter ones aim is deciding if a substitute fits into the original recipe.

4.2.1 Food Similarity

When creating new substitutions it needs to be decided which food can replace a missing ingredient. As usually multiple foods are available in the query, it needs to be tested, if one can replace another appropriately. Two measurements have been introduced in section 3.5.3. Implementations of these are described here and their performances evaluated and compared.
Listing 4.2 shows the implementation of the **upwards Distance** measurement. As stated in section 3.5.3 the **same tribe** property is a more constrained version of this algorithm. Therefore the same implementation is used for testing. In the listing we can see the constant `MAXDEPTH` which defines how many steps upwards the hierarchy may be taken. For the latter variant this is set to `1`, other values are tested against it. The formerly mentioned asymmetrical version needs just slight modification of the algorithm, it is therefore not stated here explicitly. The sole difference is using `upDistanceHelp` in one direction only.
The algorithm has a procedural entry point starting with `0` depth as first step and increasing step width after each iteration that did not lead to a clear result. An iteration consists of using the function `upDistanceHelp`, which will recursively check all parents up to the given depth for the specified `target` as descendant. `ROOT` is a constant representing the root of the food ontology which should not be crossed as any other food is a descendant of it.

4.2.2 Recipe Similarity

This section will introduce two algorithms aiming to measure the similarity of recipes and compare their outcomes. Measuring the similarity of two recipes is essential to this works approach on evaluating the fitness of substitutions related to the recipe being adapted. While the first one is the naive straightforward similarity measure the second takes the food ontology into account. The algorithms are implementations of the formulas introduced in section 3.5.4 The comparison shows the result of both algorithms on some example recipes and qualitatively evaluates the benefit of using the ontology. The similarity measure should take two recipes to measure their similarity as input and return a value between 0 and 1 as result, where 0 is no similarity and 1 is a perfect match. Not that a perfect match does not imply equality of recipes, just that the same ingredients are used.
The method signature therefore is

```
similarity(Recipe r1, Recipe r2):double.
```

```java
/**
* Returns the minimum number of steps in upward direction
* needed to go from arg0 to arg1, or vice versa.
* Returns -1 if no connection between the nodes can be
* found within MAXDEPTH.
*/
public static int upDistance(Food food0, Food food1) {
  int updepth = 0;
  int result1, result2;
  do {
    result0 = upDistanceHelp(food0, food1, updepth);
    result1 = upDistanceHelp(food1, food0, updepth);
    if (result0 == 1 || result1 == 1)
      return updepth;
    updepth++;
    // max steps reached without solution
    if (updepth == MAXDEPTH)
      return -1;
  } while (result0 != -1 && result1 != -1);
  // while stopped as of reaching ROOT
  return -1;
}

// Checks if target is descendant of any ascendant
// of item updepth levels above.
// return 1 for true, 0 for false, -1 for root
private static int upDistanceHelp(Food item, Food target, int updepth) {
  if (item.equals(ROOT)){
    // do not process the root, every food is child of it.
    return -1;
  } else if (updepth == 0) {
    // current level reached, now search for descendant
    return item.hasDescendant(target) ? 1 : 0;
  } else {
    int result = -1;
    for (Food parent : item.getParents()) {
      // recursive step for each parent
      result = Math.max(result,
         upDistanceHelp(parent, target, updepth - 1));
      if (result == 1) {
        // if any of those steps returned a match, return it further.
        return 1;
      }

    }
    // will be 0 if the node is not found, -1 when ROOT was hit.
    return result;
  }
}
```

Listing 4.2: The upwards distance algorithm.

Direct Similarity based

The direct similarity simply uses union and intersection operations[1] on the sets of ingredients from the recipes to be compared. The implementation is shown in listing 4.3.

```java
public static double recipeSimilarity(Recipe r1, Recipe r2) {
  double union = CollectionUtils.union(r1.getIngredients(),
      r2.getIngredients()).size();
  double intersection =
      CollectionUtils.intersection(r1.getIngredients(),
      r2.getIngredients()).size();
  // special case, recipes do not share at least one ingredient.
  // Prevent division by zero. (3 Recipes do not contain ingredients)
  if (union == 0 || intersection == 0)
    return 0;
  else
    return intersection / union;
}
```

Listing 4.3: The direct recipe similarity algorithm.

Descendant based

As described in section 3.5.4 the descendant based similarity puts more effort into the enumerator of the fraction. This is done with usage of the recursive hence transitive `hasDescendant`-method of `TreeItem`, which can be seen in listing 4.1. The algorithm is stated in listing 4.4. It will still return `1` when both recipes contain exactly the same ingredients, but slightly penalise usage of descendants, independent of depth of inheritance, though. This happens as of the variable `count` being increased by one if `r2` only contains a descendant of an ingredient in `r1`. At the same time the union of both sets of ingredients contains both distinct ingredients.

4.2.3 Substitution Fitness

Based on the previous two measurements on deciding if two recipes are similar to each other the following algorithms will try to determine if a substitution fits to the recipe to adapt. To achieve this recipes with a similarity above a certain threshold (in the evaluation the constant `THRESHOLD` is set to `0.4` say 40% of the two recipes ingredients match) are checked to contain the substitute under investigation. The same two strategies that led to different recipe similarities is applied to checking the adaptation. Either the new food is contained in the other recipe (see listing 4.5 for the implementation) or the substitute is a descendant of any ingredients parent. The implementation is analogue, just the `containsAny` condition is replaced by the similarity measures of section 4.2.1. In both algorithms the number of recipes that contain matches are returned. If for one ingredient multiple possible substitutions exist the one with most correspondence in other recipes is asserted to be the superior one. From the

[1]The Apache commons collections is used in version 3.2.1, `http://commons.apache.org/proper/commons-collections/`

```
public static double recipeSimilarity(Recipe r1, Recipe r2) {
  int count = 0;
  for (Food f1 : r1.getIngredients()) {
    for (Food f2 : r2.getIngredients()) {
      if (f1.equals(f2) || f2.hasDescendant(f1) || f1.hasDescendant(f2)) {
        count++;
        break;
      }
    }
  }
  double union = CollectionUtils.union(r1.getIngredients(),
     r2.getIngredients()).size();
  // special case, recipes do not share at least one ingredient. Prevent
  // division by zero. (3 Recipes do not contain ingredients)
  if (union == 0 || count == 0)
    return 0;
  else
    return count / union;
}
```

Listing 4.4: The descendant based similarity algorithm.

two algorithm variants and two variants for recipe similarity four competing variations are evaluated.
An instance of this algorithm is created by providing the `RECIPE` the substitution should be matching.

```
public double fitness(Substitution s) {
  double fitness = 0;
  for (Recipe other : datastore.getRecipes()) {
    if (similarity(RECIPE, other) > THRESHOLD) {
      if (CollectionUtils.containsAny(other.getIngredients(),
          s.getReplaceWith())) {
        fitness++;
      }
    }
  }
  return fitness;
}
```

Listing 4.5: The substitution fitness comparing ingredients directly.

4.3 Evaluation

4.3.1 Test Data

The test query provides the following ingredients, providing a basic set of very common ingredients:

- apple
- basil
- basmati rice
- bay leaf
- beef stock
- black pepper
- butter
- caraway
- carrot
- chili powder
- corn salad
- cream
- curry powder
- dark chocolate
- dry white wine
- egg
- extra virgin olive oil
- flour
- gouda
- ground pork
- honey
- iceberg lettuce
- kiwi fruit
- milk
- mustard
- oregano
- paprika (powder)
- parmigiano-reggiano cheese
- pine nut
- potato
- red kidney bean
- red onion
- rosemary
- salami
- salt
- spaghetti
- sugar
- sunflower oil
- sunflower seed
- tomato
- tomato paste
- vanilla sugar
- vegetable stock
- water

These ingredients allow to investigate the behaviour of the algorithms on different parts of the ontology with varying possibilities to find substitutions. For example the domain of *oil* is investigated with help of *extra virgin olive oil* and *sunflower oil*. Also the high amount of herbs and spices will show up some interesting results. On purpose no kind of vinegar was added to examine the results when replacing this common item.

4.3.2 Finding Possible Substitutions With the Ingredient-Ingredient Similarity

General Results

As of the identity of upwards distance with maximum depth `1` and the same tribe measurement the comparison is implemented with the same algorithm. As additional contestant the algorithm was modified to search the depth of `1` in only one direction breaking the commutativity of the similarity measure. It examines if a food is a descendant of the ingredent that needs replacement or one of its parents. The algorithms are used to create new substitutions for the foods missing in the query. In the first step of evaluation the amounts of created possible substitutions are opposed for different maximum steps in upwards distance and the previously introduced asymmetric measurement. The above query led to 90 recipes with sufficient similarity to be cookable from available foods, which were tested for needed substitutions. All in all substitutions for 90 ingredients were found. For ten foods no possible substitution could be found. Figure 4.1 shows an excerpt of the results. For each tested algorithm one bar visualises how many substitutions for a food could be found. For example with maximum depth of `0` for 70% of the missing ingredients no substitution could be found. For about 25% (only) one possible substitution is found. As it was expected for higher maximum steps for the upwards distance algorithm the number of possible substitutions grows rapidly. With a value of one the number of ingredients without replacements drops below 20%, while the number of ingredients replaceable by one other remains constant. Interestingly the latter percentage does not drop abruptly, even with maximum depth `3`. For levels `1` to `3` 30% - 50% of the foods offer a choice of two to five foods for replacement. The asymmetric algorithm is best compared against upwards distance with maximum depth `1`, as it allows for one step to the parents of the food to be replaced. As observable in the figure nearly 60% of the foods find no associate. While nearly the same amount of unary choice substitutions are sustained, the number of possible substitutions which allow for a small choice (group 2-5) drops massively.

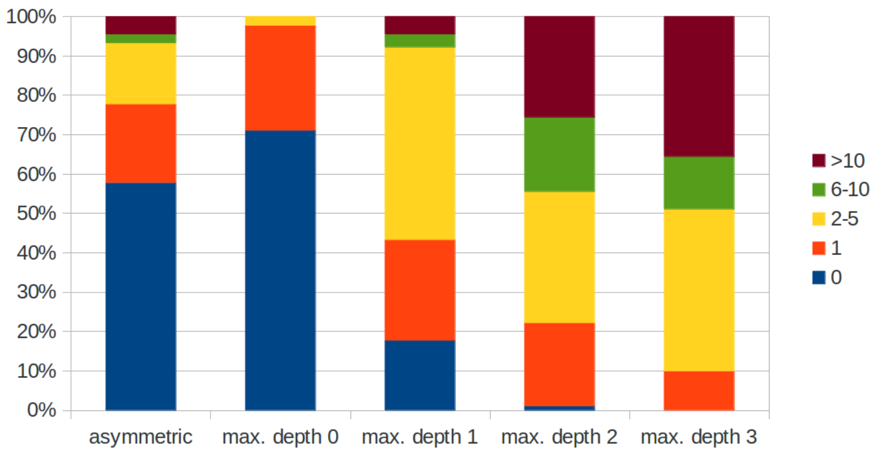


Figure 4.1: Number of possible substitutions found for missing foods by different algorithms.

Deeper Analysis of Common Cases

To further investigate especially those substitutions that offer no choice at all, and those which provide many alternatives a qualitative view on the results need to be taken. Analysis is needed on those that gain little more possible substitutions, as well as those that may suddenly be replaced by more than 10 other ingredients. We need to find out, if more options lead to the possibility of making better choices. Previously we have seen the problems when we wanted to replace *chicken breast fillet* (see section 3.5.3). We will see if we run into similar problems with the supplied query, if higher step widths lead to results, or if we end up with too many irrelevant substitutions.

Successful Examples

At first we will take a look at some successful searches for substitutions and understand why they received good results. Some recipes stated *rice* as needed ingredient. This was not specified directly in the query, but the available *basmati rice* should do the job quite well. Indeed the asymmetric algorithm, as well as the upwards distance algorithm from its lowest level are able to find the substitution. This is easily explained as basmati rice is a direct child of rice hence found in the first step of `hasDescendant`. Also none of the algorithms finds any other food. This is due to the ingredients provided and the local structure of the ontology. *Rice* is a subclass of *grain* which covers all the basic kinds of grain such as barley, corn, oat, wheat or rice. The only parent of *grain*

Algorithm	Olive oil	Corn oil	Oil
asymmetric	Sunflower oil Extra virgin olive oil	Sunflower oil Extra virgin olive oil	Sunflower oil Extra virgin olive oil
max. depth 0	Extra virgin olive oil	—	Sunflower oil Extra virgin olive oil
max. depth 1	Sunflower oil Extra virgin olive oil Butter	Sunflower oil Extra virgin olive oil	Sunflower oil Extra virgin olive oil
max. depth 2	Sunflower oil Extra virgin olive oil Butter	Sunflower oil Extra virgin olive oil	Sunflower oil Extra virgin olive oil
max. depth 3	Sunflower oil Extra virgin olive oil Butter	Sunflower oil Extra virgin olive oil	Sunflower oil Extra virgin olive oil

Table 4.1: Found substitutions for different missing kinds of oil.

if *food*, which is not crossed by the algorithm. Had the query *corn* specified, or more specifically *popcorn*, the upwards distance would have found these from step width 1 on. Obviously corn is quite a bad substitution for rice let alone popcorn. This problem is common for standard ingredients, even short distances in the tree may lead to useless results. We will see that later on in detail.
As the next example three kinds of oil are analysed, one missing ingredient is *olive oil*, one *corn oil* and the last one their parent *oil*. The results are shown in table 4.1. We can see here that all algorithms perform quite well, but depth 0 upwards depth does not find any substitution for *corn oil*, as it will only search within its descendants of which none even exist. Compared to that the case of *olive oil*, which has multiple descendants, one of which is *extra virgin olive oil*, allows the algorithm to find the latter one. Increasing the maximum depth leads to results for *corn oil* by taking the step to the superclass *oil*. From there all oils can be accessed. We can see the usefulness of increasing the step width one by one. *Olive oil* surprisingly allows for one more possible substitution: *Butter*, which indeed is a good substitution in some cases say for frying. Actually *for frying fat* is a modelled concept within the ontology and led to exactly this substitution as *olive oil* is a child of it. *For frying fat* in return is a subclass of *fat*, which also holds *butter*. Now the algorithm leads to the following path between both foods:

$$\text{Butter} \Rightarrow \text{Fat} \rightarrow \text{For frying fat} \rightarrow \text{Olive oil}$$

With one step upwards from *butter*, *olive oil* is reachable with `hasDescendant`. This reveals a weakness in the ontology—corn oil can also be used as frying fat, but is not marked as such (as most other oils). Also *vegetable oil* is a class within *for frying fat* but not contained in the local *oil* ontology (see figure 4.2 for a visualisation)!

Advantage of Asymmetric Algorithm

An interesting pattern shows up when trying to replace the cheese *cheddar*. The unbalanced measurement as well as upwards distance with distance 0 are

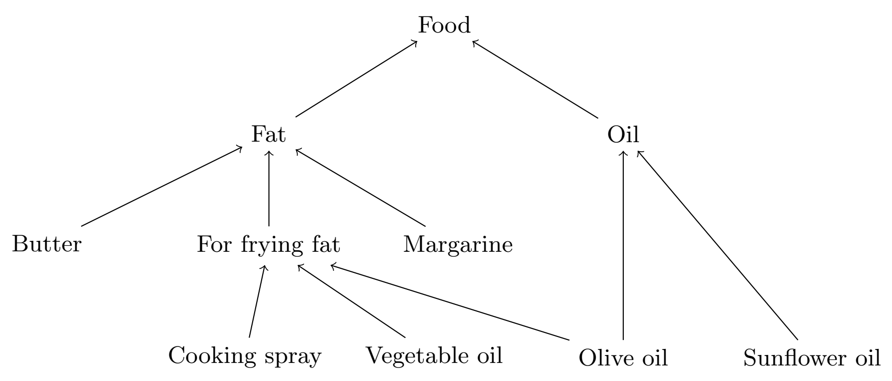


Figure 4.2: Excerpt of local ontology around *oil* and *fat*.

not able to find any substitutions. Note that the query contains *gouda* and *parmigiano-reggiano cheese*. Remembering the local structure of cheese shown in figure 3.4 the explanation is easily found. The first concept covering cheddar and any of the other two is *cheese*. Had *Parmesan* been missing *gouda* and *parmigiano-reggiano cheese* would have been found easily by checking one level of inheritance. But the figure shows that two levels of abstraction are needed, in both directions from the possible substitutions as well as *cheddar* itself. Nevertheless the upwards distance algorithm found two possible substitutions for one level of depth: *Cream* and *milk*. These are dairies, the same way *cheese* is a *dairy*. Therefore the paths

$$\text{cream} \Rightarrow \text{dairy} \rightarrow \text{cheese} \rightarrow \text{semi-firm cheese} \rightarrow \text{cheddar}$$

and

$$\text{milk} \Rightarrow \text{dairy} \rightarrow \text{cheese} \rightarrow \text{semi-firm cheese} \rightarrow \text{cheddar}$$

are found. At the next level the other kinds of cheese are found as well. Obviously improvements need be made on the algorithms. One possible way is only using the asymmetric algorithm, allowing more depth levels. This would have led to a very good performance on this problem—*gouda* and *parmigiano-reggiano cheese* would have been found at depth 2, the other contestants at depth 3. Another approach could be measuring the distance down the hierarchy as well. Counting steps downwards as well as upwards in the hierarchy results in path lengts 4 for the previously mentioned path starting from *milk* and

$$\text{cheddar} \Rightarrow \text{semi-firm cheese} \Rightarrow \text{cheese} \rightarrow \text{semi-soft cheese} \rightarrow \text{gouda}$$

also has length of 4. Suppose the query stated *young gouda* it would even be one more step.

More Problematical Cases

In the next example we will again see data that renders the asymmetric algorithm as better solution before showing situations where it quickly runs into the same problem as its commutative brothers. Two quite different ingredients

Algorithm	Cocoa	Cayenne pepper
asymmetric	Dark chocolate	Chili powder
max. depth 0	—	—
max. depth 1	Dark chocolate Salt, Sugar	Chili powder, Tomato Salt, Sugar
max. depth 2	above & Red onion, Honey, Tomato, Mustard, Carrot, Black pepper, Corn salad,. . .	above & Red onion, Honey, Tomato, Mustard, Carrot, Black pepper, Corn salad,. . .
max. depth 3	as above	as above

Table 4.2: Found substitutions for *cayenne pepper* and *cocoa*.

will lead to quite similar results, namely *cocoa* and *cayenne pepper*. Asymmetric search leads to two perfect substitutions in the first step. As of the hierarchy these steps can only be taken with one step to the parents, maximum depth 0 `upwardsDistance` does not find any results. Nevertheless with maximum distance 1 it also finds the best results. But it also finds *salt* and *sugar*, and for *chili powder* even *tomato*. While *sugar* may be a working substitution for *cocoa*, salt surely is not neither for *chili powder*. In the next steps the number of found possible surrogates explodes. As *chili powder* can be found two levels below *vegetable* as well as *spice* any child of these is suggested by the algorithm. In the case of *chocolate* this even happens on the first level of the asymmetric algorithm. Only for search restricted to descendants of it *dark chocolate* is solely found. The same applies to other examples such as *vinegar*. This behaviour is owed to the ontology which bears problems when one node has many children, and the recipes expects a direct child one level below the node. This can be observed very good in the case of *flavouring*. Hundreds of other foods are descendants of *flavouring*, reaching from *chocolate*, *garlic*, *herb*, over *liquor*, and *onion* to *salt*, *spice*, *sugar*, *vinegar*, and *wine*. This makes it hard to find good substitutions from elsewhere but descendants of the food to replace, as typically many of the ingredients are always available in a typical kitchen. As a last short successful example let a substitution of *orange* be mentioned. As of the deep structure within the *fruit* scope substitutions are found first at depth level two. In this case it may be replaced with *apple* or *kiwi*, both of which may give a recipe an unusual twist. Were more fruits contained in the query they of course would also be found, for example *lemon* which is of course more nearly related to *orange*.

Conclusion

We may conclude that structures within the ontology with nodes having few outgoing links nevertheless being precisely modelled (such as seen in the *cheese* ontology) enable the algorithms to find good substitutions, while nodes of a high degree clutter the results with useless substitutions. Erdős-Rényi-Graph like structures found a better basis for the algorithm than Barabási–Albert-Graph like ones. The latter ones pretend similarity in foods which does not exist as seen in *flavouring*. On the side of the algorithms we saw that finding more substitution does not necessarily mean increasing the chance of finding good substitutions. Quite the contrary when operating on flat structures as recently described way too many alleged neighbours are found making it hard to decide which substitution is fit. The results also show that the iterative searching procedure often allows for good results. A higher level strategy may be to even switch between asymmetric and symmetric measurements to find few but good results and stop if a certain amount or at least one good match is found. As seen in the examples moving just one step up in the hierarchy often leads to good results.

4.3.3 Evaluation of Substitution-Recipe Fitness

Although we found problems in finding good substitutions we are still interested in testing if a substitution is a good match for a recipe. This measurement based on the recipes hence different data than when the substitutions were created might also give a hint of the quality of a substitution. To evaluate the algorithms described in section 4.2.2 the possible substitutions for some recipes will be presented excerpt-wise with the values for the four possible variations which are shown in table 4.3.

	Recipe Similarity	
Food Similarity	Direct Food Similarity	Descendant-wise
Direct Food Similarity	(1)	(2)
Asymmetric Tribe-wise	(3)	(4)

Table 4.3: The four variations of the substitution fitness algorithm.

Results
Table 4.4 shows some interesting parts of the results. The ingredients from the previous section which had matching substitutions as well as those that rendered to be a hard problem are revisited. The table reveals problems within all manifestations of the algorithm. In the first example *vanilla extract* needs a replacement. Some of the proposed substitutions are *vanilla sugar* which is quite a good match especially in a cake, *black pepper* which really does not fit into cake, *salt* and *sugar*. Promising are the values 7 and 18 in the fields for tribe-wise recipe similarity. They show that some similar recipes use this ingredient, the match would be quite good. Also the result 0 for *black pepper* is very acceptable. Nevertheless *salt* and *sugar* dominate the results. The reason for this is the frequent use of *sugar* and *salt* in many recipes. Another result, which is used less often but rated nearly as well matching as *vanilla sugar* is *sunflower seed*. The same pattern shows for replacing *chocolate syrup*. As the measurement of fitness is only based on the substitute the results for *salt* and *sugar* are the same as above, but the values of *dark chocolate*, *vanilla sugar*, and *honey* are promising. It is even hard from a human point of view which of the last three was the best replacement, maybe a combination of these would be the best—mixing honey with melt chocolate seems to make up quite a good chocolate syrup.
In the next example, *Swedish cabbage rolls*, *rice* is being replaced with *basmati rice*, which luckily is the only substitution at discussion, the algorithms can not bring any more information about the quality of the match in here. Yet again

we see that *sugar* and *salt* are rated as good matches for the recipe, both of which outrun *black pepper*.
Curtis's pizza dough gives an example of replacing *olive oil* which we saw earlier on. Unfortunately the *extra virgin olive oil* is not recognised as a good match, while butter is. The same problems already discussed appear in the last given data set.

Replace	With	(1)	(2)	(3)	(4)
Brownie cake					
Vanilla extract	Vanilla sugar	0	7	0	18
Vanilla extract	Black pepper	0	0	0	0
Vanilla extract	Salt	22	88	45	190
Vanilla extract	Sugar	25	88	45	190
Vanilla extract	Sunflower seed	0	6	0	17
Chocolate syrup	Vanilla sugar	0	7	0	18
Chocolate syrup	Dark chocolate	0	4	0	6
Chocolate syrup	Salt	22	88	45	190
Chocolate syrup	Sugar	25	88	45	190
Chocolate syrup	Honey	1	2	1	3
Swedish cabbage rolls					
Rice	Basmati rice	0	0	0	0
White pepper	Black pepper	0	1	0	1
White pepper	Salt	1	2	2	6
White pepper	Sugar	0	2	0	6
Curtis's pizza dough					
Olive oil	Butter	2	6	10	27
Olive oil	Sunflower oil	1	3	1	5
Olive oil	Extra virgin olive oil	0	0	0	0
Cotton cake					
Cocoa	Vanilla sugar	0	16	0	25
Cocoa	Pine nut	0	10	0	17
Cocoa	Sugar	39	175	55	269
Cocoa	Dark chocolate	0	0	0	0
Cocoa	Salt	0	0	0	0

Table 4.4: .

Conclusion

Different problems make measuring the fitness of a substitution for a recipe hard. We have seen that *salt* and *sugar* dominate other possible substitutions because they are very common in recipes. Especially the second example shows two issues *Swedish cabbage rolls* does not have many near neighbours otherwise we again would have seen way higher values for *salt* and *sugar*. And even worse not a single one of these uses *basmati rice*. This problem can be seen in the result quite often, a potentially good match is not confirmed by this measurement because the substitute is an ingredient not used often in recipes, e.g. *dark chocolate* is used in only one recipe of the whole WikiTaable data base. At least the results show that using the ontology based recipe similarity leads to more distinct values instead of zeros.

4.4 The GUI-based Program

In addition to simple test application cycles for the previously discussed algorithm evaluation an application with a graphical user interface (GUI) based on JavaFX is provided. It allows for easy evaluation of queries and proposed adaptations. The program provides an interface allowing to specify a query with all possible constraints. When the user added all foods available and if desired more constraints a query can be executed. The results are shown in a list, the user can choose one of the presented recipes. The pane below is split into three list views. The rightmost shows the ingredients of the chosen recipe. Missing ingredients are marked red. On the left of this list two list views allow adaptation of the recipe. The left one of these shows all the missing ingredients. When one of those is marked the right list offers other (available) foods as replacements. When the user decides to use one substitution the surrogate replaces the missing ingredient in the recipes ingredient list. The item is now marked green. The user can repeat these steps for each missing ingredient. This way the adaptations are evaluated by the user. When adapting the recipe is finished it may be stored for future use. If this is done the substitutions used for creating the new recipe are stored as well, including a link to the original recipe. The full application with example recipes and adaptations can be seen in appendix B.1.
The workflow of the app is as follows:

1. Set constraints
   - Choose arbitrary number of diets (3 way switch, states: `desired, forbidden, indifferent`)
   - Choose one or no dish role
   - Choose arbitrary number of desired dish types
   - Add one or more available foods to the list
2. Run query
3. Choose recipe from the list `Possible Recipes` (choice is not final)
4. Check for missing ingredients
5. Choose possible replacements
6. Save the adapted recipe

The application offers two modes, operation of the first one, the manual mode, has just been described. To further explore the capabilities of the system a fully automatic mode is implemented which makes use of the algorithms presented in this work. Instead of searching for possible best recipes first, it adapts each recipe before making a choice. By now it does not adapt to fulfil other than food constraints, though. Therefore no recipe that initially does not fit a vegetarian diet is taken into account for adaptation when vegetarian was chosen, even if it was possible to adapt it to become compatible to the diet. The automode's purpose is finding recipes and automatically adapting them with the best possible substitutions. The application is just a demo how a program like this might look like. Other contestants put a lot more of effort in making the computational logic comfortably usable such as Yummly or other web services.

Chapter 5

Conclusion

This work presented a simple case-based reasoning system in the cooking domain. The program uses ontologies by a related system, WikiTaaable, which model foods, dish types, and other concepts, as well as its recipes as data base. From a given query which consists of a set of available foods as well as other constraints a recipe should be found which is cookable from these foods, and none other, as much as possible. Depending on these inputs it might not be possible to fully cook a recipe with its original ingredients. Therefore most promising recipes should be adapted, missing ingredients be replaced by other available ones most similar to these. Finding and evaluating these possible substitutions was focus of this work. It was shown that replacement based on semantic similarity given by the hierarchical food ontology was possible in principle. This was tested on algorithms that could search the ontology based on an iterative deepening strategy. One variant did so searching for replacement, say an ingredient contained in the query, in one direction from the food to replace, the other one searched symmetrical outgoing from both the one to replace and possible replacements. Nevertheless issues with the similarity measures were revealed. Due to different structures that showed up in the ontology, say some subhierarchies were rather flat with nodes with a high downwards outdegree, while other local structures showed the opposite. Especially when in the first case the neighbourhood of foods contradicts actual similarity of ingredients the pretended similarity leads to useless results. Hence the partition of the ontology should be improved so that algorithms can effectively work on it. Another solution could be using a group of algorithms each specialised on local subdomains. Another very interesting step that could be done is adding more different concepts to the hierarchy. For example adding concepts for taste like *sour* would allow algorithms to find *lime juice* as replacement for *vinegar* which is not possible by now due to the high distance between the nodes. With more interconnected nodes other similarity measures may also be tested, for example comparing foods by the number of concepts they share. Another approach described in this work was trying to evaluate the fitness for a substitution for a recipe. In the case that a number of different foods are available as substitution one might fit better into a recipe than others. To evaluate this a measurement was introduced searching for recipes similar to the one to adapt and counting if a substitute is contained in these. If this is the case the ingredient was successfully used in other known similar recipes and therefore suitable for the current recipe. In few cases this approach showed working results, but in most cases it had several different issues. As of the low number of recipes within the data base only few recipes really are quite similar to each other. Therefore the threshold needed to be lower which led to many recipes being similar as of sharing basic ingredients such as *salt*, *sugar*, *flour*, or *pepper*. These ingredients also showed up to be the best match to almost any recipe as they are used that often. For many other ingredients very few recipes using them exist, if any at all. Therefore the measurement rendered to be relatively useless on the given data. It may be interesting to test it on a much bigger data base without the stated drawbacks. If similar recipes can be found to the one to adapt a k-nearest neighbours approach can be tried to prevent *salt* and the like from dominating the results. By now it can not really be used as often no more than 5 recipes sharing half of the ingredients can be found.

Bibliography

Aamodt, A. and Plaza, E. (1994). Case-based reasoning; foundational issues, methodological variations, and system approaches. *AI COMMUNICATIONS*, 7(1):39–59.

Ahmed, M. U., Begum, S., and Funk, P. (2012). System overview on the clinical decision support system for stress management. In *Proceedings of the ICCBR 2012 Workshops*, pages 111–116.

Bach, K., Althoff, K.-D., Satzky, J., and Kroehl, J. (2012). Cookiis mobile: A case-based reasoning recipe customizer for android phones. In Petridis, M., Roth-Berghofer, T., and Wiratunga, N., editors, *Proceeding of the 17th UK Workshop on Case-Based Reasoning*, pages 15–26. School of Computing, Engineering and Mathematics, University of Brighton, UK.

Badra, F., Bendaoud, R., Bentebibel, R., Champin, P.-A., Cojan, J., Cordier, A., Després, S., Jean-Daubias, S., Lieber, J., Meilender, T., et al. (2008). Taaable: Text mining, ontology engineering, and hierarchical classification for textual case-based cooking. In *9th European Conference on Case-Based Reasoning-ECCBR 2008, Workshop Proceedings*, pages 219–228.

Bergmann, R., Minor, M., Islam, S., Schumacher, P., and Stromer, A. (2012). Scaling similarity-based retrieval of semantic workflows. In Lamontagne and Recip-García (2012), pages 15–24.

Carbonell, J. G. (1983). Derivational analogy and its role in problem solving. In *AAAI*, pages 64–69.

Carbonell, J. G. (1985). Derivational analogy: A theory of reconstructive problem solving and expertise acquisition.

Cojan, J., Dufour-Lussier, V., Gaillard, E., Lieber, J., Nauer, E., and Toussaint, Y. (2011). Knowledge extraction for improving case retrieval and recipe adaptation. In *Computer Cooking Contest Workshop*, London, Royaume-Uni.

Cordier, A., Lieber, J., Molli, P., Nauer, E., Skaf-Molli, H., and Toussaint, Y. (2009). WIKITAAABLE: A semantic wiki as a blackboard for a textual case-based reasoning system. In *SemWiki 2009 - 4rd Semantic Wiki Workshop at the 6th European Semantic Web Conference - ESWC 2009*, Heraklion, Grèce.

Dufour-Lussier, V., Lieber, J., Nauer, E., and Toussaint, Y. (2011). Improving case retrieval by enrichment of the domain ontology. In *Case-Based Reasoning Research and Development*, pages 62–76. Springer.

Elvidge, K. (2012). A system overview of a case-based reasoning system for care planning of end-of-life cancer care. In *Proceedings of the ICCBR 2012 Workshops*, pages 99–104.

Gaillard, E., Lieber, J., Nauer, E., et al. (2011). Adaptation knowledge discovery for cooking using closed itemset extraction. In *The Eighth International Conference on Concept Lattices and their Applications-CLA 2011*.

Gaillard, E., Nauer, E., Lefevre, M., and Cordier, A. (2012). Extracting Generic Cooking Adaptation Knowledge for the TAAABLE Case-Based Reasoning System. In *Cooking with Computers workshop @ ECAI 2012*, Montpellier, France.

Gentner, D. (1983). Structure-mapping: A theoretical framework for analogy. *Cognitive science*, 7(2):155–170.

Gunawardena, S. and Weber, R. O. (2012). Reasoning with organizational case bases in the absence negative exemplars. In Lamontagne and Recip-García (2012), pages 35–44.

Hammond, K. J. (1986). Chef: A model of case-based planning. In *AAAI*, pages 267–271.

Hanft, A., Ihle, N., and Newo, R. (2009). Refinements for retrieval and adaptation of the cookiis application. P-145:139–148.

Houeland, T. G. and Aamodt, A. (2010). The utility problem for lazy learners-towards a non-eager approach. In *Case-Based Reasoning. Research and Development*, pages 141–155. Springer.

Ihle, N., Newo, R., Hanft, A., Bach, K., and Reichle, M. (2009). Cookiis–a case-based recipe advisor. In *Workshop Proceedings of the 8th International Conference on Case-Based Reasoning*, pages 269–278.

Jagannathan, R. and Petrovic, S. (2012). A local rule-based attribute weighting scheme for a case-based reasoning system for radiotherapy treatment planning. In *Case-Based Reasoning Research and Development*, pages 167–181. Springer.

Kar, D., Chakraborti, S., and Ravindran, B. (2012). Feature weighting and confidence based prediction for case based reasoning systems. In *Case-Based Reasoning Research and Development*, pages 211–225. Springer.

Kendall-Morwick, J. and Leake, D. (2012). On tuning two-phase retrieval for structured cases. In Lamontagne and Recip-García (2012), pages 25–34.

Lamontagne, L. and Recip-García, J. A., editors (2012). *Proceedings of the ICCBR 2012 Workshops*.

López, B., Pous, C., Pla, A., and Gay, P. (2009). Boosting cbr agents with genetic algorithms. In *Case-Based Reasoning Research and Development*, pages 195–209. Springer.

López, B., Plà, A., Pous, C., Gay, P., and Brunet, J. (2012). System overview: Breast cancer prognosis through cbr. In *Proceedings of the ICCBR 2012 Workshops*, pages 105–110.

Markovitch, S. and Scott, P. D. (1988). The role of forgetting in learning. In *ML*, pages 459–465.

Marling, C., Bunescu, R. C., Shubrook, J., and Schwartz, F. (2012). System overview: The 4 diabetes support system. In *Proceedings of the ICCBR 2012 Workshops*, pages 81–86.

Minor, M., Islam, M. S., and Schumacher, P. (2012). Confidence in workflow adaptation. In *Case-Based Reasoning Research and Development*, pages 255–268. Springer.

Ontañón, S., Mishra, K., Sugandh, N., and Ram, A. (2007). Case-based planning and execution for real-time strategy games. In Weber, R. and Richter, M., editors, *Case-Based Reasoning Research and Development*, volume 4626 of *Lecture Notes in Computer Science*, pages 164–178. Springer Berlin Heidelberg.

Ontañón, S., Mishra, K., Sugandh, N., and Ram, A. (2008). Learning from demonstration and case-based planning for real-time strategy games. In Prasad, B., editor, *Soft Computing Applications in Industry*, volume 226 of *Studies in Fuzziness and Soft Computing*, pages 293–310. Springer Berlin Heidelberg.

Palma, R., Sánchez-Ruiz, A. A., Gómez-Martín, M. A., Gómez-Martín, P. P., and González-Calero, P. A. (2011). Combining expert knowledge and learning from demonstration in real-time strategy games. In Ram, A. and Wiratunga, N., editors, *ICCBR2011*, volume 6880 of *Lecture Notes in Computer Science*, pages 181–195. Springer.

Russell, S. J. and Norvig, P. (2010). *Artificial intelligence: a modern approach*. Prentice Hall, 3rd edition.

Silva, M., McCroskey, S., Rubin, J., Youngblood, G. M., and Ram, A. (2013). Learning from demonstration to be a good team member in a role playing game. In *The Twenty-Sixth International FLAIRS Conference*.

Thevenet, Q., Lefevre, M., Cordier, A., and Barnachon, M. (2012). Intelligent interactions: Artificial intelligence and motion capture for negotiation of gestural interactions. In Lamontagne and Recip-García (2012).

Veloso, M. M. (1994). prodigy/analogy: Analogical reasoning in general problem solving.

Appendix A

WikiTaaable XML Dump Examples

```
<owl:Class rdf:about="http://wikitaaable.loria.fr/index.php/
    Special:URIResolver/Category:Sunflower_oil">
  <rdfs:label>Sunflower oil</rdfs:label>
  <swivt:page rdf:resource="http://wikitaaable.loria.fr/index.php/
      Category:Sunflower_oil"/>
  <rdfs:isDefinedBy rdf:resource="http://wikitaaable.loria.fr/index.php/
      Special:ExportRDF/Category:Sunflower_oil"/>
  <swivt:wikiNamespace rdf:datatype="http://www.w3.org/2001/XMLSchema#
      integer">14</swivt:wikiNamespace>
  <property:Compatible_with_diet rdf:resource="&wiki;Category:Veganism"/>
  <property:Compatible_with_diet rdf:resource="&wiki;Category:Vegetarian"
      />
  <property:Compatible_with_diet rdf:resource="&wiki;Category:Gluten-2
      Dfree_diet"/>
  <property:Has_description rdf:datatype="http://www.w3.org/2001/
      XMLSchema#string">Sunflower oil is the non-volatile oil expressed
      from sunflower (Helianthus annuus) seeds. Sunflower oil is
      commonly used in food as a frying oil, and in cosmetic
      formulations
 as an emollient....</property:Has_description>
  <property:Has_lexical_variant>
    <swivt:Subject rdf:about="http://wikitaaable.loria.fr/index.php/
        Special:URIResolver/Category:Sunflower_oil-23
        _480e81b2f690f6d943cc903a979cd48e">
      <swivt:masterPage rdf:resource="&wiki;Category:Sunflower_oil"/>
      <property:Language rdf:datatype="http://www.w3.org/2001/XMLSchema#
          string">English</property:Language>
      <property:Plural rdf:datatype="http://www.w3.org/2001/XMLSchema#
          string">Plural not available</property:Plural>
      <property:Singular rdf:datatype="http://www.w3.org/2001/XMLSchema#
          string">sunflower oil</property:Singular>
    </swivt:Subject>
  </property:Has_lexical_variant>
  <property:Has_lexical_variant>
```

```
    <swivt:Subject rdf:about="http://wikitaaable.loria.fr/index.php/
        Special:URIResolver/Category:Sunflower_oil-23
        _cdbfad806ecb5a5a72b162a82ae37f0a">
      <swivt:masterPage rdf:resource="&wiki;Category:Sunflower_oil"/>
      <property:Language rdf:datatype="http://www.w3.org/2001/XMLSchema#
          string">Francais</property:Language>
      <property:Plural rdf:datatype="http://www.w3.org/2001/XMLSchema#
          string">huiles de tournesol</property:Plural>
      <property:Singular rdf:datatype="http://www.w3.org/2001/XMLSchema#
          string">huile de tournesol</property:Singular>
    </swivt:Subject>
  </property:Has_lexical_variant>
  <property:Has_lexical_variant>
    <swivt:Subject rdf:about="http://wikitaaable.loria.fr/index.php/
        Special:URIResolver/Category:Sunflower_oil-23
        _06a05c9fa5a5569d8b8ebe9557d3ab2f">
      <swivt:masterPage rdf:resource="&wiki;Category:Sunflower_oil"/>
      <property:Language rdf:datatype="http://www.w3.org/2001/XMLSchema#
          string">Deutsch</property:Language>
      <property:Plural rdf:datatype="http://www.w3.org/2001/XMLSchema#
          string">Plural not available</property:Plural>
      <property:Singular rdf:datatype="http://www.w3.org/2001/XMLSchema#
          string">Sonnenblumenoel</property:Singular>
    </swivt:Subject>
  </property:Has_lexical_variant>
  <swivt:wikiPageModificationDate rdf:datatype="http://www.w3.org/2001/
      XMLSchema#dateTime">2012-11-20T00:41:14Z</
      swivt:wikiPageModificationDate>
  <swivt:wikiPageSortKey rdf:datatype="http://www.w3.org/2001/XMLSchema#
      string">Sunflower oil</swivt:wikiPageSortKey>
  <rdfs:subClassOf rdf:resource="&wiki;Category:Oil"/>
</owl:Class>
```

Listing A.1: Excerpt of the XML dump of the food *sunflower oil*.

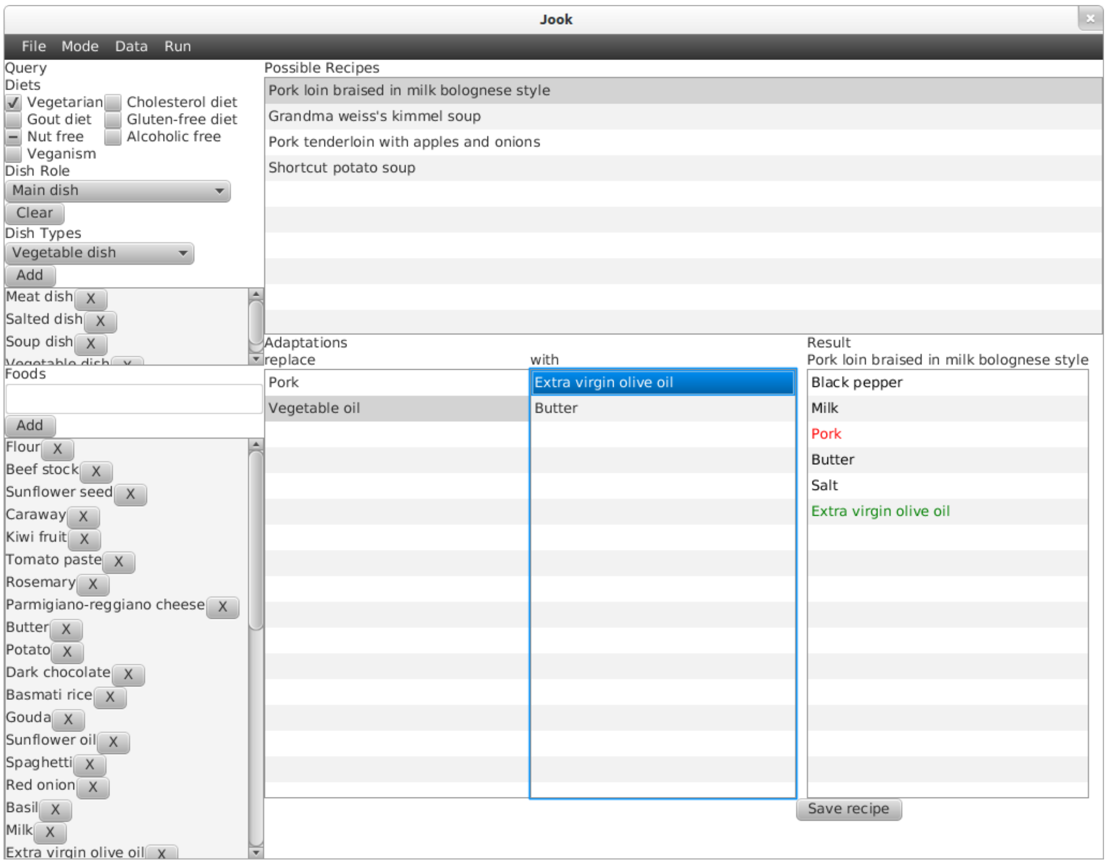
```

<swivt:Subject rdf:about="http://wikitaaable.loria.fr/index.php/
    Special:URIResolver/Curtis-27s_pizza_dough">
  <rdf:type rdf:resource="&wiki;Category:Unachievable_formal_preparation"
      />
  <rdf:type rdf:resource="&wiki;Category:Recipe"/>
  <rdf:type rdf:resource="&wiki;Category:RecipeCompulsary"/>
  <rdfs:label>Curtiss pizza dough</rdfs:label>
  <swivt:page rdf:resource="http://wikitaaable.loria.fr/index.php/Curtis
      %27s_pizza_dough"/>
  <rdfs:isDefinedBy rdf:resource="http://wikitaaable.loria.fr/index.php/
      Special:ExportRDF/Curtis%27s_pizza_dough"/>
  <swivt:wikiNamespace rdf:datatype="http://www.w3.org/2001/XMLSchema#
      integer">0</swivt:wikiNamespace>
  <property:Has_ingredient_line>
    <swivt:Subject rdf:about="http://wikitaaable.loria.fr/index.php/
        Special:URIResolver/Curtis-27s_pizza_dough-23
        _3a23606c0eaab1d8138842a82b3dc34a">
      <swivt:masterPage rdf:resource="&wiki;Curtis-27s_pizza_dough"/>
      <property:Ingredient rdf:resource="&wiki;Category:Flour"/>
      <property:Ingredient_quantity rdf:datatype="http://www.w3.org/2001/
          XMLSchema#double">2</property:Ingredient_quantity>
      <property:Ingredient_text rdf:datatype="http://www.w3.org/2001/
          XMLSchema#string">2 c Flour</property:Ingredient_text>
      <property:Ingredient_unit rdf:datatype="http://www.w3.org/2001/
          XMLSchema#string">c</property:Ingredient_unit>
    </swivt:Subject>
  </property:Has_ingredient_line>
  <property:Has_ingredient_line>
    <swivt:Subject rdf:about="http://wikitaaable.loria.fr/index.php/
        Special:URIResolver/Curtis-27s_pizza_dough-23
        _148e5f7bff8fb40b7d095d6a047f02ef">
      <swivt:masterPage rdf:resource="&wiki;Curtis-27s_pizza_dough"/>
      <property:Ingredient rdf:resource="&wiki;Category:Olive_oil"/>
      <property:Ingredient_quantity rdf:datatype="http://www.w3.org/2001/
          XMLSchema#double">2</property:Ingredient_quantity>
      <property:Ingredient_text rdf:datatype="http://www.w3.org/2001/
          XMLSchema#string">2 tb Olive oil</property:Ingredient_text>
      <property:Ingredient_unit rdf:datatype="http://www.w3.org/2001/
          XMLSchema#string">tblsp</property:Ingredient_unit>
    </swivt:Subject>
  </property:Has_ingredient_line>
  <property:Has_ingredient_line>
    <swivt:Subject rdf:about="http://wikitaaable.loria.fr/index.php/
        Special:URIResolver/Curtis-27s_pizza_dough-23
        _7e6115101a57afc63f27b45ffa76a1ce">
      <swivt:masterPage rdf:resource="&wiki;Curtis-27s_pizza_dough"/>
      <property:Ingredient rdf:resource="&wiki;Category:Sugar"/>
      <property:Ingredient_quantity rdf:datatype="http://www.w3.org/2001/
          XMLSchema#double">0.5</property:Ingredient_quantity>
      <property:Ingredient_text rdf:datatype="http://www.w3.org/2001/
          XMLSchema#string">1/2 ts Sugar</property:Ingredient_text>
      <property:Ingredient_unit rdf:datatype="http://www.w3.org/2001/
          XMLSchema#string">tsp</property:Ingredient_unit>
    </swivt:Subject>
  </property:Has_ingredient_line>
  <property:Has_ingredient_line>
    <swivt:Subject rdf:about="http://wikitaaable.loria.fr/index.php/
        Special:URIResolver/Curtis-27s_pizza_dough-23
        _2eb47e83a7b64b06e9323feae2a5bb14">
      <swivt:masterPage rdf:resource="&wiki;Curtis-27s_pizza_dough"/>
      <property:Ingredient rdf:resource="&wiki;Category:Yeast"/>
      <property:Ingredient_quantity rdf:datatype="http://www.w3.org/2001/
          XMLSchema#double">1</property:Ingredient_quantity>
      <property:Ingredient_text rdf:datatype="http://www.w3.org/2001/
          XMLSchema#string">1 pk Yeast</property:Ingredient_text>
      <property:Ingredient_unit rdf:datatype="http://www.w3.org/2001/
          XMLSchema#string">pk</property:Ingredient_unit>
    </swivt:Subject>
  </property:Has_ingredient_line>
  <property:Has_ingredient_line>
```

```
    <swivt:Subject rdf:about="http://wikitaaable.loria.fr/index.php/
        Special:URIResolver/Curtis-27s_pizza_dough-23
        _0ca5fc47fe29f244c92a54d3ab38e13b">
      <swivt:masterPage rdf:resource="&wiki;Curtis-27s_pizza_dough"/>
      <property:Ingredient rdf:resource="&wiki;Category:Salt"/>
      <property:Ingredient_text rdf:datatype="http://www.w3.org/2001/
          XMLSchema#string">Dash of salt</property:Ingredient_text>
    </swivt:Subject>
 </property:Has_ingredient_line>
   <property:Has_ingredient_line>
     <swivt:Subject rdf:about="http://wikitaaable.loria.fr/index.php/
         Special:URIResolver/Curtis-27s_pizza_dough-23
         _a80832fda78be9b72748dd586b75c24b">
       <swivt:masterPage rdf:resource="&wiki;Curtis-27s_pizza_dough"/>
       <property:Ingredient rdf:resource="&wiki;Category:Water"/>
       <property:Ingredient_qualifiers rdf:datatype="http://www.w3.org
           /2001/XMLSchema#string">warm</property:Ingredient_qualifiers>
       <property:Ingredient_quantity rdf:datatype="http://www.w3.org
           /2001/XMLSchema#double">0.75</property:Ingredient_quantity>
       <property:Ingredient_text rdf:datatype="http://www.w3.org/2001/
           XMLSchema#string">3/4 cup warm water</property:Ingredient_text
           >
       <property:Ingredient_unit rdf:datatype="http://www.w3.org/2001/
           XMLSchema#string">c</property:Ingredient_unit>
     </swivt:Subject>
   </property:Has_ingredient_line>
   <property:Produces_dish_type rdf:resource="&wiki;Category:Pizza_dish"/
       >
   <swivt:wikiPageModificationDate rdf:datatype="http://www.w3.org/2001/
       XMLSchema#dateTime">2012-03-20T17:08:28Z</
       swivt:wikiPageModificationDate>
   <swivt:wikiPageSortKey rdf:datatype="http://www.w3.org/2001/XMLSchema#
       string">Curtiss pizza dough</swivt:wikiPageSortKey>
</swivt:Subject>
```

Listing A.2: XML dump of the recipe *Curtis's pizza dough*.

Appendix B

Java Application



Query

Diets

[✓ Vegetarian] Cholesterol diet
Gout diet Gluten-free diet
[– Nut free] Alcoholic free
Veganism

Dish Role: Main dish

Dish Types: Vegetable dish

Meat dish X
Salted dish X
Soup dish X
Vegetable dish X

Foods

Flour X
Beef stock X
Sunflower seed X
Caraway X
Kiwi fruit X
Tomato paste X
Rosemary X
Parmigiano-reggiano cheese X
Butter X
Potato X
Dark chocolate X
Basmati rice X
Gouda X
Sunflower oil X
Spaghetti X
Red onion X
Basil X
Milk X
Extra virgin olive oil X

Possible Recipes

- Pork loin braised in milk bolognese style
- Grandma weiss's kimmel soup
- Pork tenderloin with apples and onions
- Shortcut potato soup

Adaptations

replace	with
Pork	Extra virgin olive oil
Vegetable oil	Butter

Result

Pork loin braised in milk bolognese style

- Black pepper
- Milk
- Pork
- Butter
- Salt
- Extra virgin olive oil

Figure B.1: A screenshot of the application which is part of this work. It shows an example manual adaptation process.

Ich erkläre hiermit gemäß § 17 Abs. 2 APO, dass ich die vorstehende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

(Datum) (Unterschrift)