Examensarbete

Design and implementation of a constraint satisfaction algorithm for meal planning

av

Niclas Sundmark

LITH-IDA-EX--05/094--SE

2005-12-16

Linköpings universitet
Institutionen för datavetenskap

Examensarbete

Design and implementation of a constraint satisfaction algorithm for meal planning

av

Niclas Sundmark

LITH-IDA- EX--05/094--SE

2005-12-16

Handledare: Johan Åberg
Examinator: Johan Åberg

Abstract

The world’s population is ageing. Due to societal improvements in healthcare, living standards, and socio-economic status, more and more people are living to old age. The proportion of the world's population aged 65 or over is expected to increase from 11% in 1998 to 16% in 2025. This causes a major public health issue, because with increased age there is an increased risk of developing a number of age-related diseases. However, there is increasing scientific evidence that many of the biological changes and risks for chronic disease, which have traditionally been attributed to ageing, are in fact caused by malnutrition (sub-optimal diets and nutrient intakes).

This report presents a constraint satisfaction approach to planning meals while taking into account amongst other things nutritional and economic factors. Two models for generating meal plans are presented and their respective strengths and weaknesses discussed. System design, implementation and the main algorithms used are described in more detail. These algorithms include Depth First Branch and Bound and its various improvements for meal plan generation as well as Item-based Collaborative Filtering for user preferences. Our test runs show that the system works well for smaller applications but runs into problems when the number of available recipes grows or a larger number of meals are planned. The tests also show that the two modelling approaches both have useful applications. Based on the test results some suggestions for further improvement of the system conclude the report.

1 Introduction

This chapter gives a brief introduction to the project detailed in this report.

1.1 Background

The meal planning system is part of a larger research project at Linköping University aiming to improve quality of life for the elderly population. The motivation for such a system is that in recent years quality of life has increased and is likely to continue doing so, this leads to people living to older ages. The proportion of the world's population aged 65+ is expected to increase from 11% in 1998 to 16% in 2025.

Recent research has shown that diseases previously believed to be caused by the aging process are actually caused by malnutrition and can be prevented. There can be many reasons for malnutrition such as economic reasons, social reasons (for example losing someone close to you leading to depression) or lack of variation in food intake.

In this research project an intelligent meal support system has been proposed, providing suggestions for what to eat over a specified period of time. While this system is targeted at the elderly population it can also have uses amongst the younger population such as families with children who want help deciding what to eat that is suitable for the entire family.

1.2 Aim of the project

The purpose of this project is to develop a constraint satisfaction algorithm and surrounding data structures for planning meals. The need for such a system is motivated by the growing problem of malnutrition amongst the elderly that is amongst other things caused by lack of variation in food intake. For this reason it is interesting to develop a tool that can be used for planning a series of meals taking into account variation requirements, nutritional values, cost of the meals etc. To this date constraint satisfaction has not been applied to the meal-planning problem but has been shown to be successful in other planning areas such as travel planning and for this reason good results were expected from the study.

1.3 Directives

The starting point of this project is a PhD thesis written by Mark Torrens [13] where constraint satisfaction is discussed in detail and several algorithms for solving such problems are presented and compared. The system is to be implemented in Java for portability and reduced workload when doing further development such as web applications. The recipe data is

to be extracted from an existing collection of fifty recipes in XML format and complemented by computer generated data where needed for performance testing.

2 Literature study

This chapter describes the literature study performed prior to designing and implementing the system as well as the conclusions reached. Section 2.1 summarises the preliminary research while 2.2 presents the further research into the areas that were determined to be interesting after the initial study was completed.

2.1 Basic research

Three methods for generating meal plans were examined; local search, branch and bound and search methods for Pareto optimality [14].

2.1.1 Local search

This area was very loosely studied, mainly because a local search algorithm needs to examine the entire search space to know that it has found the best solution. Branch and bound algorithms[1] on the other hand can prune uninteresting branches in the search tree. Another reason for not using local search is that at any given point in the search a local search algorithm gives no indication of how far from an optimal solution the current one is, again in contrast to branch and bound algorithms. This is a desired property if the meal planning system should allow a user to stop the search before the entire search space has been examined.

2.1.2 Branch and bound

The biggest advantage of classic branch and bound is that there has been a lot of research done on the subject. However the area of soft constraints is relatively new in comparison and the biggest problem when using weighted CSP modelling is setting appropriate weights for different constraints[2]. Having all weights equal is one option but requires that the valuation functions are adapted to reflect the relative importance of constraints so the problem isn't solved it is simply moved and possibly made more complex. Another possibility would be using some form of learning technique that sets constraint weights based on user feedback.

2.1.3 Pareto optimality

Pareto optimality[3] is a relatively new research area and a few different methods for approximating the set of Pareto optimal solutions were studied. To find the complete set has been shown to be very time consuming so it is not interesting for a time critical application. This is especially true in the

1 For information on branch and bound algorithms see section 6.1.

2 For information on soft constraints and the different classes of constraint satisfaction problems see chapter 3.

3 For information on Pareto optimality see section 6.2.

meal-planning system where the set of Pareto optimal solutions will be large because of a large amount of soft constraints and the difference between two Pareto optimal solutions is bound to be small.

2.1.4 Conclusions from preliminary studies

Branch and bound was determined to be the best approach for finding solutions in large part because of the amount of research done on the subject. It was also decided to look deeper into the area of Pareto optimality to see if this could be combined with branch and bound in some way to produce higher quality solutions. Finally research was to be done in the area of learning or estimating user preferences to produce meal plans of high quality possibly containing meals that the user had not provided a rating for.

2.2 Further research

After the initial literature study further research was made into the areas of improvements to the depth first branch and bound algorithm, Pareto optimality and estimating user preferences.

2.2.1 Dynamic variable ordering

Bacchus has found [1, 2, 3] that advanced algorithms such as Forward Checking (FC) [13] with Conflict-directed Backjumping (CBJ) [8, 9] result in a minimal, max 5% on random problems, improvement over regular FC if the Minimal Remaining Values (MRV) heuristic is used for dynamic variable ordering (DVO)[4]. Because of this Bacchus suggests using regular FC with DVO over more advanced algorithms due to the relative simplicity of FC. Note that if DVO is not used CBJ outperforms FC by a large amount.

One must keep in mind that the concept of the MRV heuristic is to quickly find invalid solutions and discard them. However when the majority of constraints in the problem are soft constraints the set of invalid solutions is usually relatively small.

2.2.2 Binary vs. non-binary CSP

Bacchus [3] has investigated when one should convert from a non-binary CSP to a binary version. His conclusion is that a conversion should be made when the number of constraints is low in comparison to the number of variables and constraints are strict. In general a conversion is profitable when the solution space is small.

4 More details on dynamic variable ordering and forward checking can be found in sections 6.1.2 and 6.1.3.

Worth noting is that most research in the area has been made on binary CSPs and some algorithms may have to be adapted to work with non-binary problems and might not give results similar to the binary case.

2.2.3 Methods for approximating the Pareto optimal set

Torrens discusses Pareto[5] approximation in [14] and has looked at a few different methods based on the principle of finding solutions in two steps. The problem is solved using a standard branch and bound algorithm and the obtained solutions are then filtered for Pareto optimality. The method most thoroughly investigated is the weighted-sums method that solves the problem a number of times using different weight vectors for the constraints. It then adds up the results from the different runs and removes the ones that are not Pareto optimal. A few different methods for choosing weight vectors is investigated but when execution time is taken into account a single weight vector consisting of only ones is found to be the best alternative.

2.2.4 Estimating user preferences

Machine learning of preferences has been shown possible by amongst others [5] and the results have at times been better than when the user himself has guessed how he would like an item. Furthermore the method explored in this paper, gradient descent, was shown to work both when data was given in bulk and with incremental input.

Instead of learning the user's preferences one can approximate them from ratings for similar items. This is the idea behind item based collaborative filtering [10, 12], as opposed to regular collaborative filtering where similarities are found between user types instead of items.

2.2.5 Conclusions

Dynamic variable ordering is highly interesting for speeding up the solution but might not give as much improvement as expected due to the large solution set. It was decided that four algorithms would be implemented for the meal planning system to see what worked best and how big the difference was. The four algorithms chosen were standard depth first branch and bound, forward checking, forward checking with dynamic variable ordering, and partial forward checking (forward checking adapted for soft constraints). More on these algorithms can be found in chapter 6.

Given our large solution set we did not decide to stick to binary constraints, unfortunately this lead to a few problems later on - more on that matter in

5 For information on Pareto optimality see section 6.2.

section 12.2. Pareto approximation was to be done with a single weight vector since we wanted high quality solutions in a limited time.

Item-based collaborative filtering is included in the system to generate preferences for recipes that users had not yet rated. More information on collaborative filtering can be found in section 6.3.

3 Constraint satisfaction

This chapter is intended as an introduction to constraint satisfaction or a reminder for those who have studied the subject before. More detailed information and mathematical definitions can be found in [13] for those interested but should not be needed to understand the information contained in this report. Section 3.1 gives a brief introduction, explaining when and why problems are modelled as constraint satisfaction problems. In section 3.2 constraint satisfaction problems are discussed in more detail and in section 3.3 you can read about how the meal-planning problem was modelled as a constraint satisfaction problem.

3.1 Introduction

Constraint satisfaction can be used to model a wide range of problems and solve them using one of a number of backtracking-based algorithms. Implementing a constraint system consists of two main steps:

1. **Modelling** the problem as a Constraint Satisfaction Problem (CSP).
2. **Solving** the CSP using one of the well-known algorithms.

Step two is relatively straightforward; you can read about the branch and bound algorithm and its variants in section 6.1. How the meal-planning problem was modelled will be discussed in more detail in section 3.3.

In general when the number of possible solutions is small the list of solutions can simply be presented to the user for selection. Optionally the list can be filtered or sorted according to some criteria (e.g. user preferences). User preferences can be used as hard filtering constraints or soft optimisation constraints. However when the set of valid solutions is large a more advanced encoding technique is needed. Constraint satisfaction techniques have been shown to be highly efficient and suitable for modelling configuration problems but the classic definition of constraint satisfaction is not powerful enough to express user preferences or optimisation constraints. For this reason an extended model is presented in [13]. In the following section the main notions of constraint satisfaction and the extensions needed for user preferences and optimisation are presented.

3.2 Constraint satisfaction problems (CSPs)

CSPs are common in applications such as configuration, planning, resource allocation, scheduling and timetabling. A CSP is specified by a set of variables and a set of constraints among them. A solution to a CSP is an assignment to all variables such that all constraints are satisfied. There can be many, one or no solutions to a given problem.

3.2.1 Basic notions

A CSP consists of a set of *variables*, a set of *domains* for those variables and a set of *constraints* amongst them, which must be satisfied to obtain a valid solution to the problem.

Each constraint has a *scope* consisting of the set of variables involved in the constraint. Unary constraints affect one variable and binary constraints affect two variables. Constraints affecting more than two variables are generally referred to as n-ary constraints. Most research on constraint satisfaction has been aimed at binary CSPs – problems containing only unary and binary constraints.

Some definitions:

- The *size* of a CSP is the number of variables in the problem.
- A *solution* is an assignment to all variables in the problem, such that all the constraints are satisfied.
- A *partial solution* is an assignment to some, but not all, of the variables such that all constraints are satisfied.
- The *solution space* of a CSP is the set of all valid solutions.
- The *search space* is the set of all possible assignments to all variables.

3.2.2 Introducing soft constraints

The main weakness of classical constraint satisfaction is the fact that it only finds one or more valid solutions but gives no information about which of those solutions is the best (and by how much). In order to deal with this problem an extended framework was needed that would not only find valid solutions but also be able to order them in some way.

In the extended framework constraints are classified depending on two characteristics:

- **Level of necessity**
  - *Hard constraints*, which must be satisfied to obtain a valid solution.
  - *Soft constraints*, which may be violated if necessary.
- **Degree of satisfaction**
  - *Crisp constraints*, which can be either completely satisfied or completely violated, they do not accept a degree of violation.
  - *Flexible constraints*, which can be satisfied (or violated) to a certain degree.

From these characteristics three different types of constraints are identified:

- **Hard and Crisp** constraints are classical constraints as described in 2.2.1.
- **Soft and Crisp** constraints may be violated but are either completely satisfied or completely violated. An example of this would be if a user would like a specific meal to contain chicken.
- **Soft and Flexible** constraints can be violated to a degree and are usually modelled with a valuation function. Examples of this would be if a user desires his meals to be as cheap as possible or take approximately 30 minutes to prepare.

A few different frameworks dealing with soft constraints have been developed:

- **Partial CSP** mainly deals with over constrained problems, i.e. problems that do not accept any solution. Another use of partial constraint satisfaction is solving problems where finding a complete solution is too complex or costly. MAX-CSP, or maximal CSP, is a general framework for modelling and solving partial CSPs that finds the solution with the *lowest number of violated constraints.*
- **Soft CSP** is the generic term for CSPs that deal with constraints that are neither crisp nor hard. Many instances derive from soft CSPs:
    - In **Fuzzy CSPs** each tuple of values in the constraints has an associated preference level (valuation) from 0 to 1, where 1 is the best and 0 the worst. A solution to a fuzzy CSP is a total assignment that *maximizes the minimum constraint valuation* in the assignment.
    - **Weighted CSPs** associate a weight, cost or penalty to value tuples in constraints. A solution to a weighted CSP is a total assignment that *minimizes the sum of the constraints.* Weighted CSPs can be seen as a generalization of MAX-CSP.
    - **Possibilistic CSPs** associate a possibilistic distribution among value tuples to each constraint. A solution to a possibilistic CSP is an assignment that *minimizes the maximum valuation* among the constraints. Possibilistic CSPs can be seen as the dual problem of fuzzy CSPs.
    - **Lexicographic CSPs** deal with the problem of generating solutions in a too coarse way (drowning effect) when using min-max or max-min schemas. Lexicographic CSPs discriminates between solutions with the same min/max valuation by considering *vectors of valuations* ordered in a lexicographic manner. Two solutions to a fuzzy or possibilistic CSP having the same min/max valuation can be differentiated by considering the second minimum/maximum valuation, and so on.

The meal planning system uses an approach based on weighted CSPs. The reason for this is that we want some constraints to be more important than others and that is not supported in any of the other models unless you implement the valuations accordingly, which is a lot more complicated to balance.

3.2.3 Branch and Bound

The most popular way to solve a CSP is to use one of many branch and bound algorithms. The most basic algorithm is known as depth first branch and bound and combines a depth first search with pruning techniques for the search tree and constraint valuations to ensure that the current branch can lead to a valid solution. The pruning is done by keeping track of the valuation for the best solution found so far, known as the upper bound (UB), and comparing this to the valuation of the current assignment, known as the lower bound (LB). Pruning is done when LB > UB. More details on the various branch and bound algorithms can be found in section 6.1.

3.3 Modelling the meal-planning problem as a CSP

As described above a CSP consists of a set of variables, a set of domains for those variables and a set of constraints amongst them. Two different models were designed for the meal-planning system and will be described in the following sections but a common set of constraints was developed for both models.

A variation constraint was needed to ensure dietary diversity and an availability constraint would make sure that we made the best possible use of available ingredients. These two constraints would be constraints between meals. In addition to this, constraints would be needed for all parameters of a meal – time, cost, difficulty and nutritional values. These constraints would keep the solutions as close as possible to the specified values and affect a single meal. To account for dietary restrictions or allergies two hard constraints for ingredients and meal categories to avoid were included as well as constraints for nutritional values.

We didn’t want the system suggesting meals the user wouldn’t appreciate so a rating system for recipes was included in the meal-planner and a preference constraint devised to prefer meals that the user had given good ratings.

More detailed information about these constraints and their valuation functions can be found in section 4.1.

3.3.1 Model one – the parameter model

In this model the problem contains one variable for each of the parameters of a recipe in the database as follows:

- Time – the time required to prepare the meal
- Cost – the total cost of the meal
- Difficulty – the preparation difficulty of the meal
- Carbohydrates – the carbohydrate content of the meal
- Fat – the fat content of the meal
- Energy – the energy content of the meal
- Protein – the protein content of the meal
- Sodium – the sodium content of the meal
- Calcium – the calcium content of the meal
- Cholesterol – the cholesterol content of the meal

The domains of these variables contain all the different values present in the recipe database. For example if we had a tiny database consisting of only two recipes, one with a preparation time of 20 minutes and the other with a preparation time of 30 minutes the domain of the Time variable would be {20, 30}.

In addition to the constraints described above this model needed an additional type of hard constraint, known as the recipe constraint, ensuring that the current partial assignment of variables could eventually (in a leaf node) lead to a valid assignment that would be matched to a recipe in the database.

This model has a constant search depth but the branching factor varies between levels of the tree due to the varying domain sizes.

Each call to the algorithm in this approach will only produce a single meal so in order to plan several meals; several calls to the algorithm have to be made, updating or adding (removing) constraints between calls. If we had chosen to plan several meals in one call the search depth would quickly have grown out of hand, already at two meals planned we would have had a search depth of 20 (two of each variable).

3.3.2 Model two – the recipe model

In this model the problem contains one variable for each meal to be planned. The domains are all the same and contain the names of all recipes in the database. For example if we had a tiny database consisting of the recipes “Vegetable Soup” and “Cheeseburger” the domain of all variables would be {“Vegetable Soup”, “Cheeseburger”}.

This model has a constant branching factor but the search depth varies with the number of meals planned.

Only one call to the algorithm is needed regardless of the number of meals planned.

4 System design

This chapter will describe briefly how the meal planning system was designed and why. Section 4.1 describes the different constraints that are supported by the system and section 4.2 describes the planning context.

4.1 Required constraints

Three different types of constraints were to be included in the system: optimisation constraints, user profile constraints and temporary constraints.

Optimisation constraints were to be in effect if no user profile or temporary constraints were present for a given parameter for a given meal. These constraints aim to plan cheap, simple and quick meals with good variation and availability of ingredients.

User profile constraints were to be in effect for any meal where the user was one of the people planned for. These constraints regulate nutritional values in addition to the values regulated by the optimisation constraints and also include any ingredients or meal categories the user cannot eat.

Temporary constraints are specified for a given meal on a given day and override user profile and optimisation constraints. They regulate the same values as user profile constraints.

In the following sections the constraints available will be described along with their valuations. Valuations range from 0 to 1 with 0 being the best valuation and 1 the worst. Time and cost constraints consider three ranges of values instead of the exact values specified in the recipes.

4.1.1 Optimisation constraints

These constraints are in effect if no user profile or temporary constraints have been specified.

4.1.1.1 Time constraint

Minimizes preparation time.
Valuation: `short -> 0, medium -> 0.5, long -> 1`

4.1.1.2 Difficulty constraint

Minimizes preparation difficulty.
Valuation: `easy -> 0, medium -> 0.5, hard -> 1`

4.1.1.3 Cost constraint

Minimizes total cost of the meal plan.
Valuation: `low -> 0, medium -> 0.5, high -> 1`

4.1.1.4 Variation constraint

Maximizes variation between meals, taking into account ingredients and categories.

Valuation: `0.7*categoryPenalty + 0.3*ingredientPenalty where categoryPenalty = duplicateCategories/totalCategories/(numberOfMeals-1) and ingredientPenalty = duplicateIngredients/totalIngredients/(numberOfMeals-1)`

For example if we plan two meals, the two meals have two common categories and each belongs to an additional category the valuation would be `0.7*2/4/1 = 0.35` assuming that they don't share any ingredients (this would be somewhat unlikely since they share categories).

4.1.1.5 Availability constraint

Maximizes availability of required ingredients, taking into account cost, needed amount and expiration date of the ingredients.

Valuation:
`costOfMissingMealIngredients/costOfAllIngredientsInMeal`

The cost of missing ingredients takes into account the amount missing and if any ingredients are unusable due to expiration dates or if they have already been used in other meals.

4.1.1.6 User taste constraint

Prefers recipes with higher rating. This constraint is in effect when a user has not rated the current recipe and the rating is determined by item-based collaborative filtering.

Valuation: `(5-rating)/4`

4.1.2 User profile constraints

These constraints are stored in the user profiles and override the general optimisation constraints if they are in conflict.

4.1.2.1 Time constraint

Optimises preparation time according to user's preference. Short, medium and long preparation times are selectable.

Valuation: `no deviation -> 0, deviates by one step -> 0.5, deviates by two steps -> 1`

For example if we desire recipes with high preparation time, low preparation time recipes will have a valuation of 1 and medium time will have a valuation of 0.5

4.1.2.2 Difficulty constraint

Optimises preparation difficulty according to user's preference. Easy, medium and hard difficulties are selectable.

Valuation: `no deviation -> 0, deviates by one step -> 0.5, deviates by two steps -> 1`

4.1.2.3 Cost constraint

Optimises cost according to user's preference. Low, medium and high costs are selectable.

Valuation: `no deviation -> 0, deviates by one step -> 0.5, deviates by two steps -> 1`

4.1.2.4 Availability constraint

Optimises availability of ingredients according to user's preference. Low, medium and high availability is selectable.

Valuation: `This is simply the optimisation constraint with a higher weight according to the user's preference; if low availability is selected they are identical.`

4.1.2.5 Variation constraint

Optimises meal variation according to user's preference. Low, medium and high variation is selectable.

Valuation: `This is simply the optimisation constraint with a higher weight according to the user's preference; if low variation is selected they are identical.`

4.1.2.6 Ingredient constraint

Avoids ingredients specified by the user. This is a hard constraint.

Valuation: `1 if a specified ingredient is included in any of the planned meals, 0 otherwise.`

4.1.2.7 Category constraint

Avoids meal categories specified by the user. This is a hard constraint.

Valuation: `1 if any of the planned meals are included in the category, 0 otherwise.`

4.1.2.8 Nutrition constraint

Optimises nutritional value according to user's preference. Values are specified with maximum and minimum desired values, values outside this range are penalized. The supported nutritional values are carbohydrates, energy, fat, protein, sodium, calcium and cholesterol.

Valuation: `0 if the value is in the specified range, 1 if it deviates by more than 50, deviation/50 otherwise.`

4.1.2.9 User taste constraint

Prefers recipes with higher rating. This constraint is in effect only when a user has rated the current recipe.

Valuation: `(5-rating)/4`

4.1.3 Temporary constraints

These constraints are specified for one or more meals on one or more days and override optimisation and user profile constraints if they are in conflict.

4.1.3.1 Time constraint

Optimises preparation time as specified. Short, medium and long preparation times are selectable.

Valuation: `no deviation -> 0, deviates by one step -> 0.5, deviates by two steps -> 1`

4.1.3.2 Difficulty constraint

Optimises preparation difficulty as specified. Easy, medium and hard difficulties are selectable.

Valuation: `no deviation -> 0, deviates by one step -> 0.5, deviates by two steps -> 1`

4.1.3.3 Cost constraint

Optimises meal cost as specified. Low, medium and high difficulties are selectable.

Valuation: `no deviation -> 0, deviates by one step -> 0.5, deviates by two steps -> 1`

4.1.3.4 Availability constraint

Optimises ingredient availability as specified. Low, medium and high availability is selectable.

Valuation: `This is simply the optimisation constraint with a higher weight according to the specification; if low availability is selected they are identical.`

4.1.3.5 Variation constraint

Optimises meal variation as specified. Low, medium and high variation is selectable.

Valuation: `This is simply the optimisation constraint with a higher weight according to the specification; if low variation is selected they are identical.`

4.1.3.6 Ingredient constraint (include)

Penalizes meals not containing the specified ingredient.

Valuation: `1-matchedMeals/numberOfMeals`

For example, we plan 4 meals and we want the temporary constraint to apply to all of them. If 3 of them contain the ingredient the penalty will be 0.25.

4.1.3.7 Ingredient constraint (avoid)

Avoids meals containing the specified ingredient. This is a hard constraint.

Valuation: `1 if a specified ingredient is included in any of the planned meals, 0 otherwise.`

4.1.3.8 Category constraint (include)

Penalizes meals that are not in the specified meal category.

Valuation: `1-matchedMeals/numberOfMeals`

4.1.3.9 Category constraint (avoid)

Avoids meals that are in a specified meal category. This is a hard constraint.

Valuation: `1 if any of the planned meals are included in the category, 0 otherwise.`

4.1.3.10 Nutrition constraint

Penalizes meals with nutritional values outside the specified range. The supported nutritional values are carbohydrates, energy, fat, protein, sodium, calcium and cholesterol.

Valuation: `0 if the value is in the specified range, 1 if it deviates by more than 50, deviation/50 otherwise.`

4.2 Meal plan context

The context of the meals planned for include the people participating in the meals. This can vary between different meals within a larger plan. For example, including a guest for dinner on Friday implies that the guest's user profile constraints (if known) must be taken into account for that particular meal. Each user profile stores an importance factor used to weigh or choose between conflicting constraints.

5 Implementation

This chapter describes the software model used for the meal-planning system. For a description of key classes and code organization see Appendix B.

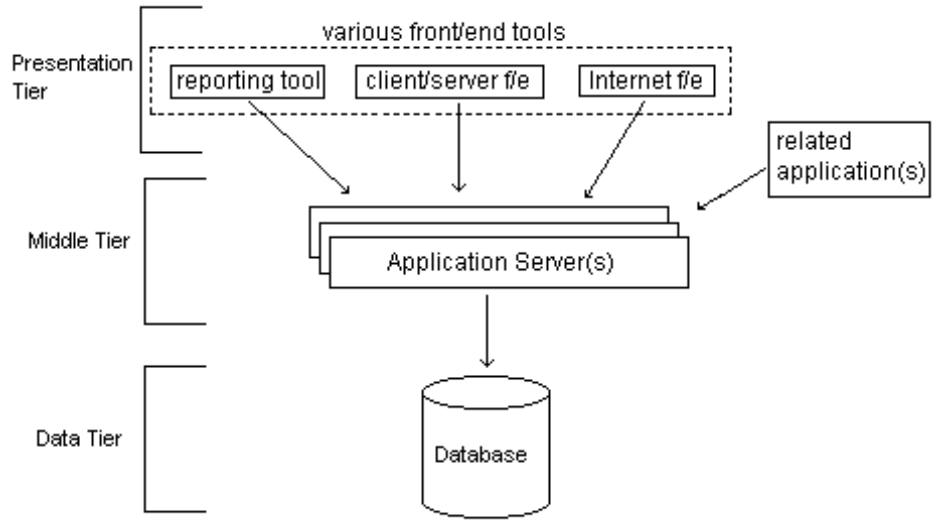
5.1 N-tier

The N-tier model [15] is a software model frequently used in distributed applications but it is also well suited for applications residing on a single system that need to be easily extendible or modified.

5.1.1 Pure 3-Tier / N-Tier Model

In the N-Tier model there are three formal tiers: the presentation tier, the middle tier and the data tier. The reason the model is denoted N-Tier is that the so-called middle tier can be implemented in one or more tiers.

Figure 1: Pure 3-Tier/N-Tier Model



5.1.1.1 Presentation tier

The presentation tier can consist of several types of clients and monitoring tools. These clients and tools rarely perform any computations of their own, all computation is done in the middle tier. The result of this is that if more than one client or tool needs to provide an identical or similar function a lot of development time is saved, as the work does not need to be duplicated. Apart from user interfaces and monitors the applications in the presentation tier may also be various applications that only need to exchange data with the N-Tier system.

5.1.1.2 Middle tier

As mentioned above the middle tier in itself can consist of one or more tiers, distributed over more than one machine if desired. This is where all the heavy computation takes place and the distributed approach allows for easy handling of increased workload - just add more servers. In small systems the middle tier does not need to be located on a separate machine, it may reside on the same machine as a client or database and simply be moved if a performance boost is needed.

5.1.1.3 Data tier

The data tier consists of the database and data access functionality; the database may be distributed over several machines as needed.

5.1.2 Tiers in the meal planning system

The meal planning system uses a 3-tier model with three tiers similar to those above: the User Interface tier, the Application Logic tier and the Data tier.

5.1.2.1 The User Interface tier

This tier has been developed in another project at the university and consists of a graphical user interface that was designed to be used by older people.

5.1.2.2 The Application Logic tier

This tier is the central one for this project. It contains all constraints and classes for managing plans, subjects and recipes as well as methods for generating plans.

5.1.2.3 The Data tier

This tier manages all hard drive access and saving/loading of all subject, plan and recipe data. Subject, plan and recipe data is currently stored in text files in a format similar to XML but the tiered model allows for an easy transition to a proper database system if desired.

6 Algorithms

This chapter describes the main algorithms used in the meal-planning system. Section 6.1 describes the solving algorithms used to generate meal-plans and section 6.3 describes the item-based collaborative filtering algorithm used for generating preferences when a subject has not yet rated a recipe. Section 6.2 describes the filter used for finding Pareto optimal solutions.

6.1 Depth first branch and bound

It has been shown that the solutions to a weighted constraint optimisation problem (WCOP) can be ordered by a better-than relation, in such a way that a solution with a lower constraint valuation is better than another solution with a higher valuation. The depth first branch and bound algorithm finds valid solutions (satisfying all hard constraints) that minimize the sum of the soft constraint valuations. In order for this approach to work the valuation functions for the soft constraints need to be normalized and the constraint weight vector must be numerically known.

The algorithm with various improvements as well as adaptations to find the k-best solutions is given in the following sections.

6.1.1 Description of the algorithm

Depth first branch and bound explores the search tree using a depth first strategy and evaluates constraints along the way. The algorithm starts from the empty assignment and gradually extends a partial assignment to eventually reach a complete assignment at a leaf node. At each level of the tree a different variable is instantiated so at level *k* the (partial) assignment contains *k* variables.

For each new variable assignment the relevant hard constraints are evaluated to make sure that the current assignment does not violate any of them. If no hard constraints are violated the soft constraints are evaluated and the resulting value, known as the *Lower Bound* (LB), is compared to the best one found so far, known as the *Upper Bound* (UB, initially set to ∞). Whenever a hard constraint is violated or LB > UB the current search path can not lead to a valid solution that is an improvement over the best solution found so far so the branch is pruned and the algorithm backtracks to the previous node. The *Lower Bound* is an underestimation of the future valuations of the current branch and is required to be non-decreasing along a path of the search tree. This is trivial if all constraint valuations are non-negative. When the algorithm reaches a leaf node and LB < UB a new best solution has been found, the current LB becomes the new UB and the algorithm backtracks.

Pseudo code for the basic DFBB algorithm is given in Algorithm 1. Assignments are expressed as sets of variable instantiations which are denoted {var <- val}. PreviousVariable returns nil if there are no more previous variables (the search is at the root node). When PreviousVariable returns nil and there are no more values to try for the current variable the whole search tree has been explored and the search terminates. The conditionals checking hard and soft constraints could be merged into one line however in the meal-planning system it makes sense to check if currentAssig is a valid assignment before computing LB. This is generally the case when the computation of soft constraints consumes a large amount of time compared to the hard constraints.

The goal for improving the efficiency of branch and bound algorithms is always to detect nodes that cannot lead to a valid and improved solution as soon as possible. Therefore if the problem is highly constrained with respect to the hard constraints it is also preferable to check them first. In general one could think about a mixing order of the tests but this has not been explored in the meal-planning system.

When solving a binary CSP the computation of the hard constraints is done by only checking the constraints that involve the current variable and any of the previously instantiated ones (the others have already been checked or will be checked for a future assignment). Since the meal-planning system uses a few n-ary hard constraints, most notably the recipe constraint[6], this part of the algorithm is only implicitly implemented through constraints immediately returning 0 if they do not apply to the current assignment. For example any hard ingredient constraint[7] used in the parameter model will immediately return 0 when the current assignment is not complete (since it can't be matched to a recipe to retrieve ingredients from). Whenever a hard constraint is violated the algorithm immediately backtracks without checking any further constraints.

The same thing applies to soft constraints but a lot of the soft constraints in the meal-planning system are n-ary constraints so the same approach is used as for hard constraints. While summarizing the constraint valuations to compute LB the algorithm backtracks without checking any further constraints if at any point LB > UB.

Algorithm 1 can easily be adapted to return the *k* best solutions to the problem instead of only the best one. In this case the first *k* valid solutions

[6] See section 3.3.1
[7] See section 4.1.2.6

are kept without computing LB or UB and when *k* solutions has been found UB is set to the worst valuation amongst them. From that point on LB is computed as in the original algorithm and when an improved solution is found it replaces the worst of the *k* solutions and UB is once again set to the worst valuation amongst them. The modifications needed for giving the *k* best solutions can be seen in Algorithm 2.

Algorithm 1: Depth First Branch and Bound (DFBB) algorithm to find the best solution to a WCOP.

```
function DFBB
Input: WCOP P = (X,D,HC,SC,W)
Output: bestSol: the best solution to P
  bestSol ← ” ”
  currentAssig ← ” ”
  LB ← 0
  UB ← ∞
  var ← FirstVariable()
  val ← FirstValue(var)
  end ← false
  while ¬end do
    currentAssig ← currentAssig{var←val}
       if HC(currentAssig) = 0 then
         LB ← SC(currentAssig)
         if LB < UB then
           if LastVariable?() then
             bestSol ← currentAssig
             UB ← LB
            NewBranch(var, val)
           else
            var ← NextVariable()
            val ← NextValue(var)
           endif
         else
           NewBranch(var, val)
         endif
       else
         NewBranch(var, val)
       endif
  end
  return bestSol
end

procedure NewBranch(var,val)
  while LastValue?(val) and ¬end do
    var ← PreviousVariable()
    if var ~= nil then
      val ← CurrentValue(var)
    else
      ; the whole search space has been explored!
```

```
      end ← true
    endif
  end
  if ¬end then
    val ← NextValue(var)
  endif
end

function HC(assign)
  foreach constraint C in HC
    if C.valuation(assign) > 0 then
      return 1
    endif
   end
  return 0
end

function SC(assign)
  valuation ← 0
  foreach constraint C in SC
    valuation = valuation + C.valuation(assign)
    if valuation > UB then
      return valuation
    endif
   end
  return valuation
end
```

Algorithm 2: Depth First Branch and Bound (DFBB) algorithm to find the *k* best solutions to a WCOP.

```
function DFBB for the k best solutions
Input: WCOP P = (X,D,HC,SC,W)
k : the number of best solutions to compute
Output: bestSolutions: the k best solutions to P
  bestSolutions ← " "
  worstSol ← " " ; the worst solution of the bestSolutions
  ...
  if |bestSolutions| ≥ k then
    ; LB is only computed if there are k solutions in
    ; bestSolutions
    LB ← SC(currentAssig)
  endif
  ...
  if |bestSolutions| ≥ k then
    worstSol ← WorstSolution(bestSolutions)
    bestSolutions ← ReplaceSolution(worstSol, currentAssig)
    worstSol ← WorstSolution(bestSolutions)
    UB ← SC(worstSol)
  else
    ; the first k valid solutions are always kept
```

```
    bestSolutions ← bestSolutions + currentAssig
  endif
  ...
  return bestSolutions
end
```

It’s worth noting that there could be more feasible solutions with the same valuation as the worst solution of the *k* best solutions found by Algorithm 2 although this is very unlikely in the meal-planning system with the number of soft constraints used and the large number of possible valuations for each constraint. This is more of an issue when using MAX-CSP where you only count the number of violated constraints.

The DFBB algorithm has an exponential worst-case complexity because in the worst case the algorithm visits all the nodes in the search tree and checks all the constraints involved in the problem. Therefore strategies to avoid achieving this bound have been proposed and the following sections review the main techniques used to improve the basic DFBB.

6.1.2 Forward Checking

Look-ahead techniques in constraint satisfaction work to ensure that the current assignment can be extended to a valid solution and they can be applied as pre-processing techniques or during runtime. The goal of these techniques is to prune branches that cannot lead to a valid solution at an early stage of the search. If used as pre-processing techniques they produce a smaller but equivalent problem due to decreased domain sizes when inconsistent values are removed. When used during the search inconsistent values can be removed from future variables and backtracking can be performed in a more intelligent manner.

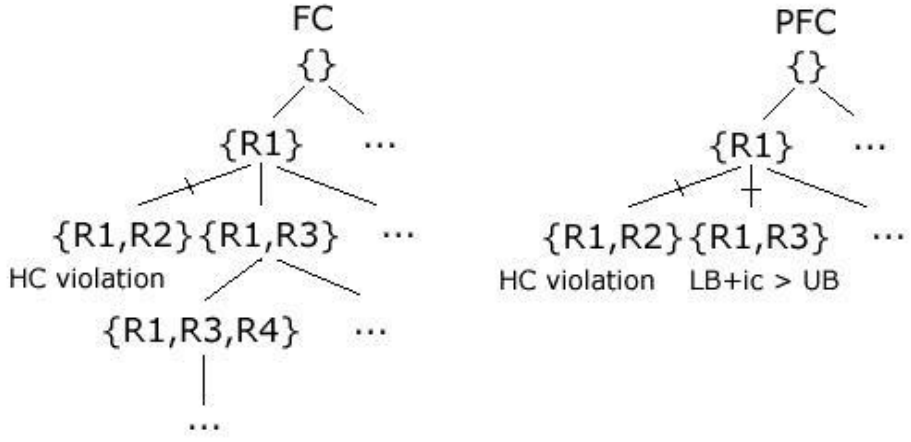
Classic *Forward Checking* (FC) evaluates constraints at each node that apply to the current variable and any future variables. If a future value leads to a constraint violation that value is removed from the domain of the variable in question. When there are no valid values (the domain is empty) for the next variable FC backtracks. The extra cost associated with FC compared to normal backtracking is that when FC backtracks it must restore the values that were removed in the last propagation. If one were to compare the two you could say that FC is the complement of backtracking because it checks forward to ensure that future values are consistent with the current assignment while backtracking ensures consistency in past assignments.

Because the definition of a *dead-end* node is different in a classic CSP and a CSP with soft constraints an adaptation of FC has been developed known as *Partial Forward Checking* (PFC). In classic CSP a dead-end node is a node

where all future values lead to a hard constraint violation. When dealing with soft constraints a dead-end node is, in addition to the previous definition, a node where all future values lead to a worse solution than the best one found so far. To deal with this fact PFC uses a value known as the *inconsistency count* (ic) that for each future value keeps track of how many soft constraint violations that value leads to. The ic value is used in the pruning process where the branch for variable *j* with value *k* is pruned if $LB + ic_{jk} \geq UB$. If pruning is not performed LB is then updated as $LB = LB + ic_{jk}$ and the search continues. In later revisions of the algorithm the pruning calculation has been improved to consider all future variables and pruning occurs if $LB + \sum \min(ic_j) \geq UB$. This means that if the best possible future assignment is worse than the best one found so far this branch is immediately pruned.

Figure 2 gives an example of the differences between FC and PFC. While FC only prunes branches that violate hard constraints PFC also prunes the ones that cannot lead to an improvement over the best solution found so far. The figure illustrates the recipe model and R1-R4 symbolizes names of recipes in the database.

Figure 2: Forward Checking versus Partial Forward Checking



6.1.3 Adaptations for weighted CSP

The PFC algorithm as described above solves MAX-CSP and only considers the number of violated constraints. To solve WCOP the algorithm must be adapted to consider the sum of the violations and adaptation is made by a change in the meaning of ic. Instead of the number of violated soft constraints ic_{jk} represents the sum of all constraint violations that arise from assigning value *k* to variable *j*. The adapted algorithm is shown in Algorithm 3.

The propagation part of the algorithm consists of two parts, hard and soft constraints. The hard constraint propagation is performed as in classic FC. The soft propagation is a two-step process, it first updates ic for all possible future assignments and then removes the values where LB + ic ≥ UB from the appropriate domains.

The main differences between Algorithm 1 and Algorithm 3 are the UpdateLowerBound, Propagation and UnPropagation functions. UpdateLowerBound computes ic and updates LB. Propagation updates ic and performs pruning of values that would lead to a worse (or invalid) solution than the best one found so far. UnPropagation restores any values removed in the last propagation. When the PFC algorithm reaches a leaf node it has found a new best solution. Note that no constraint valuations are needed in leaf nodes since if the current solution was worse than the best one found so far it could not have been reached due to the pruning process. The modifications needed for Algorithm 3 to find the k best solutions are similar to those needed to turn Algorithm 1 into Algorithm 2.

Algorithm 3: Partial Forward Checking (PFC) algorithm for finding the best solution to a WCOP.

```
function PFC
Input: WCOP P = (X,D,HC,SC,W)
Output: bestSol: the best solution to P
  bestSol ← ” ”
  currentAssig ← ” ”
  LB ← 0
  UB ← ∞
  var ← FirstVariable()
  val ← FirstValue(var)
  end ← false
  while ¬end do
    currentAssig ← currentAssig{var←val}
    if LastVariable?() then
      bestSol ← currentAssig
      UB ← LB
      NewBranch(var, val)
    else
      Propagation(UB)
      LB ← UpdateLowerBound(LB)
      var ← NextVariable()
      val ← NextValue(var)
      if val = nil then
        NewBranch(var, val)
      endif
    endif
  end
```

```
    return bestSol
  end

  procedure NewBranch(var,val)
    ...
    UnPropagation()
    var ← PreviousVariable()
    ...
  end

  procedure Propagation(UB)
    updates ic for any value in all unassigned variables
    removes future values which ic ≥ UB
    removes future values which violates any hard constraint
  end

  procedure UnPropagation()
    restores the values that were removed during the last
Propagation
  end
```

6.1.4 Variable and value ordering techniques

Branch and bound algorithms visit nodes in the order specified by heuristics that select the next variable and value, these are denoted `Next/PreviousVariable/Value` in the algorithm descriptions. Choosing a different heuristic can make a large difference for the performance of an algorithm depending on the problem to be solved.

Heuristics are divided into two main groups, static ordering and dynamic ordering. Static ordering determines the search order prior to the search and the search tree is fixed during the search process. Dynamic ordering selects variables and values during the search. For each node visited a new variable and value is selected so the search tree is not known in advance and it changes during the search process. Sometimes the two techniques are combined, usually in such a way that a static ordering is used to break ties in the dynamic process.

6.1.4.1 Variable ordering

The goal of all variable ordering techniques is to quickly increase LB and discover dead-end nodes as soon as possible. Many well-known heuristics exist and a few of them are presented below.

Static heuristics include:

- **Decreasing Backward Degree (BD)** that counts the number of constraints involving the current variable with previously selected ones and selects the most constrained variable. Variables chosen will normally

have a higher ic and thereby a higher LB. The drawback of this approach is the lack of information near the top of the tree.

- **Decreasing Forward Degree (FD)** that can be seen as the complement of BD. Instead of looking at previous variables it aims to propagate ic to future variables by selecting the variable most constrained with regard to future variables. The drawback is lack of information near the bottom of the tree.
- **Decreasing Degree (DG)** tries to overcome the drawbacks of BD and FD by using FD at the higher levels of the tree and switching to BD at the lower levels.

The most widely used heuristic for dynamic variable ordering selects the variable with the smallest (remaining) domain. This heuristic is known as *minimum domain* (DOM), or *minimum remaining values* (MRV) when forward checking is used, and has been shown [4] to be very effective when used with FC. This heuristic is only useful when domains are of different sizes.

6.1.4.2 Value ordering

There has not been a lot of research done in the area of value ordering for partial constraint satisfaction but the goal of such an approach would be to decrease UB as fast as possible.

6.2 Filtering for Pareto optimality

A solution is said to be Pareto optimal if it is not dominated by any other solution. A solution S_1 dominates S_2 if there are no constraint valuations that are worse for S_1 than S_2 and there is at least one constraint valuation that is better for S_1 than S_2. Hence the objective of a filter is simply to remove solutions that do not meet these criteria.

Algorithm 4: Pareto optimality filter.

```
function ParetoFilter
Input: Sol: a set of solutions to a WCOP
       SC: the set of soft constraints used to obtain Sol
Output: filteredSol: the Pareto optimal solutions in Sol
  foreach solution S1 in Sol
     if there exists another solution S2 in Sol such that
        for some constraint C in SC
        C.valuation(S1) > C.valuation(S2) then
      continue foreach
    elseif there is another solution S2 in Sol such that
          for some constraint C in SC
          C.valuation(S1) < C.valuation(S2) then
      filteredSol.add(S1)
    endif
```

```
  end
  return filteredSol
end
```

6.3 Collaborative filtering

Recommendation systems are widely used in e-commerce to give users suggestions on what to buy. These suggestions can be based on different kinds of data such as user demographics, best selling items or previously purchased items. Collaborative filtering (CF) is the most popular approach to date and uses the ratings of likeminded users to suggest items.

6.3.1 Introduction

The goal of collaborative filtering is to recommend a list of products for a user or to predict how a user would rate a certain item, it bases these recommendations or predictions on user ratings as stated above. In a typical system where CF is used you will find a set of *m* users and a set of *n* items. Each user has a list of items that he or she has rated explicitly or implicitly (through buying habits for example). When asked for a recommendation the algorithm will provide a list of the *top-N* recommended items based on this data, this approach is commonly known as *top-N recommendation*. Collaborative filtering systems come in two main flavours and these are described below.

6.3.1.1 User-based collaborative filtering

The user-based approach uses the entire user-item database to generate predictions. It begins by generating a neighbourhood of likeminded users using statistic methods. Likeminded users can for example be those who have purchased the same items in the past or have made similar item ratings. The ratings of the users in this neighbourhood are then used to generate the list of *top-N* items or a predicted rating for a given item. This approach is also known as *nearest neighbour* or *memory-based* collaborative filtering and is very widespread.

6.3.1.2 Item-based collaborative filtering

The item-based approach creates a model of ratings and takes a probabilistic approach to item recommendation by computing the expected value of a prediction given the user's previous ratings. The model building is performed by different machine learning algorithms such as Bayesian networks, clustering or rule-based systems. This approach is also known as *model-based* collaborative filtering.

6.3.2 Reasons to use item-based CF

The problem areas of user-based CF are those of sparse data and scalability. The sparse data problem arises from the fact that in systems with a lot of items a given user might have rated as little as 1% of the items, usually even

less. The result of this is that neighbourhood generation doesn't provide meaningful results and the nearest-neighbour approach produces poor quality recommendations. The large amount of items and users provide the scalability problem since execution time increases both with the number of users in the database and the number of items available.

Attempts have been made to solve the problem with sparse data by using semi-intelligent filtering agents that rate items using syntactic features. The higher density of ratings did improve the results but the problem of likeminded users who had rated a small set of items remained. This problem lead to an attempt using Latent Semantic Indexing but it did not improve on the problem as much as the next approach: model-based CF.
Model-based collaborative filtering turned out to be more successful than the previous attempts, especially concerning the scalability problem. The reason for this is that item-based CF looks for similarities between items instead of users, the intuition being that user would avoid items similar to those items he or she had previously disliked. Likewise a user would be more inclined to buy an item similar to other items he or she had rated highly. Since the algorithm compares items it does not need to compute the neighbourhood used in the user-based approach and this saves a lot of time. The similarity of items can be computed in a few different ways as will be described in the next section.

6.3.3 Item-based CF algorithm

The first step of the item-based collaborative filtering algorithm given a target item is computing the similarity of other items and selecting the *k* most similar. These items are then used to compute a rating prediction for the target item as a weighted average based on the user's ratings of the *k* items. The two steps are detailed below.

6.3.3.1 Similarity calculation

This step calculates the similarities between the target item and all other items; it then selects the *k* items that are most similar to the target item. The basic idea is to find all users who have rated the target item *i* as well as another item *j* that has been rated by the current user. When all such users have been found the similarity is computed based on user ratings using for example one of the following approaches: cosine-based similarity, correlation-based similarity or adjusted cosine similarity.

- Cosine-based similarity treats the pair of items as vectors in user-space and the similarity is computed as the cosine value of the angle between them.
- Correlation-based similarity computes similarity between item *i* and item *j* as the *Pearson-r* correlation between them. The set of users, *U*, who have

rated both items are isolated and the similarity is given by the user ratings of the two items, adjusted by the average rating for the items. $R_{u,i}$ is the rating user *u* has given item *i*.

$$sim(i,j) = corr_{i,j} = \frac{\sum_{u \in U}(R_{u,i} - \bar{R}_i)(R_{u,j} - \bar{R}_j)}{\sqrt{\sum_{u \in U}(R_{u,i} - \bar{R}_i)^2}\sqrt{\sum_{u \in U}(R_{u,j} - \bar{R}_j)^2}}.$$

- Adjusted cosine similarity takes note of the fact that the item ratings come from different users who might not use the same rating scale. For example a rating of 4 from a conservative user might have the same perceived value as a rating of 5 from another more generous user. To deal with this the adjusted cosine approach uses the same calculation as the correlation-based one but adjusts the ratings by subtracting the average rating of the rating user instead of the average rating for the item. The similarity calculation then becomes:

$$sim(i,j) = \frac{\sum_{u \in U}(R_{u,i} - \bar{R}_u)(R_{u,j} - \bar{R}_u)}{\sqrt{\sum_{u \in U}(R_{u,i} - \bar{R}_u)^2}\sqrt{\sum_{u \in U}(R_{u,j} - \bar{R}_u)^2}}.$$

6.3.3.2 Rating prediction

This is the most important step of the item-based CF algorithm. When we have found the set of most similar items it's time to look at the current user's ratings of these items and produce a predicted rating for the target item. There are two main techniques used to make this prediction: weighted sum and regression.

- The weighted sum approach does exactly what the name implies. It calculates the sum of all the ratings weighted by the similarities of the items. The sum is then scaled by the sum of the similarities to produce a rating in the predefined range. $P_{u,i}$ is the predicted rating of item *i* for user *u*, $s_{i,N}$ is the similarity between item *i* and item *N*.

$$P_{u,i} = \frac{\sum_{\text{all similar items, N}}(s_{i,N} * R_{u,N})}{\sum_{\text{all similar items, N}}(|s_{i,N}|)}$$

- The regression approach uses the same calculation as weighted sum but instead of the actual user ratings it adds up approximate ratings produced by regression.

6.3.4 Performance considerations for item-based CF

As noted earlier the item-based approach does not have the problem of neighbourhood-generation like the user-based approach but there are a few other things to consider. The item-based approach can be split into two parts, the similarity part and the prediction part, and this turns out to be a very good thing. The item-item similarity computation is the bottleneck of the

system but since it is separate from the prediction it can actually be pre-computed for the entire database. During runtime the similarity calculation step is then reduced to simple table lookups.

In the meal-planning system the similarity computation is currently done at runtime but it has been implemented in such a way that the computation can easily be replaced by a network call to a ratings server or a table lookup if pre-computation can be done locally.

6.4 Implementation in the meal planning system

For the meal-planning system three main constraint satisfaction algorithms have been implemented: basic DFBB, Forward Checking and Partial Forward Checking. The Forward Checking algorithm uses dynamic variable ordering with the MRV heuristic but is never used for the recipe model since all domains are of the same size. Dynamic value ordering has not been implemented in the meal-planning system but the recipe sorting described in section 7.2 performs approximately the same function.

The collaborative filtering algorithm used for recipe ratings is item-based and uses adjusted cosine similarity for the similarity computation. Weighted sum is used for the prediction since it requires less computation than the regression model and has also been shown [12] to perform better as data density increases.

7 Optimisation methods

This chapter will describe some of the steps taken to improve the quality of the generated plans, a value ordering heuristic that was tested to improve solving speed in the recipe model and the opportunity to halt the search before the entire search tree has been explored.

7.1 Suggestion variation

For each meal plan that is generated the system presents the user with a number of different suggestions. In a first approach the alternative assignments provided by the *k*-best DFBB algorithms were selected but these turned out to be much to similar to one another. This was especially true for the recipe model where the best alternative for the solution {Recipe1, Recipe2} would always be {Recipe2, Recipe1}. The problem had to be solved in different ways for the parameter model and the recipe model.

7.1.1 Parameter model

To produce varied suggestions in the parameter model we only allow a single recipe in common between two suggestions. To ensure this we can no longer use the alternatives provided by the *k*-best algorithms and have to

make a new call to the algorithm for each desired suggestion. Between each call we remove all but one recipe from the previous suggestion from the set of valid assignments in the recipe constraint.

7.1.2 Recipe model

Just like in the parameter model we only allow a single recipe in common between suggestions. We initially tried to simply filter the produced suggestions and only keep the ones that were varied enough but it turned out that we needed a much too large number of suggestions in order to present only a few plans to the user and the number increased rapidly with the number of meals planned.

Our second attempt was made using a modified version of the *k*-best PFC algorithm. Whenever a new solution was to be added to the *k*-best list we compared it to the solutions already in the list. If the new solution was too similar to one of the solutions that had a lower valuation, the new solution was discarded. If it was not too similar it was added to the list and we removed all the plans with higher valuations that were too similar to the new solution.

7.2 Recipe sorting (value ordering)

In an attempt to improve the solving speed for the recipe model we tried to sort the recipe list before the search, thus affecting the order in which values were selected from the variable domains. The list was sorted in such a way that we summarised the ratings of all users in the database for each recipe and the recipes with the highest values were selected first. This produced varied results for different problems and no clear conclusion was made.

7.3 Stop-and-go and threading

We utilized threading functionality to allow the user to get a result from the algorithm before it had explored the entire search tree. The system would run the algorithm in a separate thread and upon request return the best solution(s) found so far. However, there is currently no support for this functionality in the user interface. See Appendix B for details on how to use this functionality.

8 Generation of test data

This chapter describes how the test data in the form of randomised subjects and recipes was generated.

8.1 Randomisation methods

Two types of random distribution were used for data generation, linear and Gaussian ("normal") distribution. The linear distribution produces pseudorandom integer values from inclusive 0 to a specified exclusive max value with each value being just as probable as any other. In the descriptions below an interval will be presented which means we have simply added a fixed number to the random result, for example to provide values in the range 3-7 the formula used is `3+linearRand(5)`. The Gaussian distribution provides "normally" distributed values with mean 0 and a standard deviation of 1. In the descriptions below a *midpoint* and a *deviation* will be presented and the formula used is `midpoint+gaussianRand()*deviation` which provides us with a normal distribution with mean *midpoint* and standard deviation of *multiplier*. The result is rounded down to the nearest integer and will be normally distributed around the midpoint with approximately 68% of values ending up in the range `midpoint±deviation` and 95% within the range `midpoint±deviation*2`. Values outside the later range are discarded in our test data to provide us with a bounded interval.

8.2 Subject generation

The values randomised for subjects are desired nutritional values and desired meal parameters (time, cost, difficulty). The number of refused ingredients/categories varies between test runs. Recipe ratings are not randomised for the reason that in the current implementation they consume too much disk space, for the purpose of testing all ratings are considered to be 3.

8.2.1 Nutritional values

The Gaussian distribution was used to produce nutritional requirements. The selected ranges are based on recommendations of daily nutrition intake. Negative results are multiplied by –1.

Carbohydrates
Midpoint: 65
Deviation: 65

Energy content
Midpoint: 2100
Deviation: 300

Fat
Midpoint: 27
Deviation: 7

Protein
Midpoint: 55
Deviation: 8

Cholesterol
Midpoint: 100
Deviation: 100

Sodium
Midpoint: 60
Deviation: 60

Calcium
Midpoint: 1000
Deviation: 200

8.2.2 Meal parameters

The linear distribution was used for meal parameter generation.

Time
Interval: 1-3 (short-long)

Difficulty
Interval: 1-3 (easy-hard)

Cost
Interval: 1-3 (low-high)

8.3 Recipe generation

The values randomised for recipes are nutritional values, the meal parameters (time, cost and difficulty) and a number of ingredients and categories.

8.2.1 Nutritional values

The Gaussian distribution was used to produce nutritional values. The selected ranges are based on recommendations of daily nutrition intake. Negative results are multiplied by –1.

Carbohydrates
Midpoint: 65
Deviation: 65

Energy content
Midpoint: 2100
Deviation: 300

Fat
Midpoint: 27
Deviation: 7

Protein
Midpoint: 55
Deviation: 8

Cholesterol
Midpoint: 100
Deviation: 100

Sodium
Midpoint: 60
Deviation: 60

Calcium
Midpoint: 1000
Deviation: 200

8.2.2 Meal parameters

The Gaussian distribution was used for time and cost. Linear distribution was used for difficulty.

Time
Midpoint: 60
Deviation: 25

Difficulty
Interval: 1-3 (easy-hard)

Cost
Midpoint: 40
Deviation: 15

8.2.3 Ingredients and categories

The Gaussian distribution was used for these values.

Ingredients
Midpoint: 10
Deviation: 4

Categories
Midpoint: 3
Deviation: 1

9 Evaluation

To evaluate the different algorithms, plan variation schemes and the two problem models a number of tests were run on the test data and execution time measured. For each parameter combination three runs were performed and the average result recorded.

The parameters varied were:

- Number of recipes in the database
- Number of meals planned
- Number of plan suggestions
- Number of subjects planned for
- Number of meal parameters specified per subject
- Number of nutritional requirements specified per subject
- Number of refused ingredients/categories per subject

The default case uses 100 recipes, 2 meals, 1 suggestion, 1 subject and 0 of the other parameters. The parameters were varied one by one except for in the selected scenarios described in 9.2.

9.1 General parameter ranges

Number of recipes: 50, 100, 500, 1000, 3000, 5000
Number of meals: 1-6
Number of suggestions: 1-5
Number of subjects: 1-5, 10, 15, 20
Number of meal parameters: 0-3
Number of nutritional requirements: 0-7
Number of refused ingredients: 0, 5, 10, 15
Number of refused categories: 0-5, 10

9.2 Selected scenarios

We made up four typical scenarios to get a better idea of the performance of the system in real life situations. Results are presented in chapter 10.

9.2.1 The family

The family consists of two adults and two children; they have a rather small collection of recipes and prefer their meals to be cheap but do not have any demands on nutritional intake. The children refuse to eat fish so recipes from the fish category should not be planned.

How many meals can the family plan in a reasonable time?

Number of recipes: 100

Number of meals: varies
Number of suggestions: 2
Number of subjects: 4
Number of meal parameters: 1
Number of nutritional requirements: 0
Number of refused ingredients: 0
Number of refused categories: 1

9.2.2 The student

The student lives alone and wants his meals to be cheap, fast and easy to prepare. He has a very small collection of recipes and has some nutritional requirements concerning fat, protein and carbohydrates.

How many meals can the student plan in a reasonable time?

Number of recipes: 50
Number of meals: varies
Number of suggestions: 1
Number of subjects: 1
Number of meal parameters: 3
Number of nutritional requirements: 3
Number of refused ingredients: 0
Number of refused categories: 0

9.2.3 The senior citizen

The senior citizen has gathered a rather large collection of recipes and she prefers her food to be cheap. She has received some nutritional recommendations from her doctor and she would like to follow them. Since she doesn't quite trust the planner she wants to have many different plans to choose from. She doesn't appreciate some of the ingredients used in today's cooking so she doesn't want any meals containing them.

How many meals can she plan in a reasonable time?

Number of recipes: 500
Number of meals: varies
Number of suggestions: 6
Number of subjects: 1
Number of meal parameters: 1
Number of nutritional requirements: 7
Number of refused ingredients: 5
Number of refused categories: 0

9.2.4 The retirement home

The retirement home has access to a very large recipe database and prefers meals that are easy and fast to prepare. On average the residents have four nutritional requirements, five ingredients they cannot or will not eat and one category that they avoid.

How many people can the retirement home plan for at once?

Number of recipes: 3000
Number of meals: 5
Number of suggestions: 1
Number of subjects: varies
Number of meal parameters: 2
Number of nutritional requirements: 4
Number of refused ingredients: 5
Number of refused categories: 1

10 Results

Complete results for all parameter combinations can be found in Appendix A.

10.1 General results

Looking at forbidden ingredients we see that for the parameter model search times actually increase as we add ingredients, this is likely the result of these constraints only being in effect at the lowest level of the tree and thus not removing a lot of invalid assignments. For the recipe model the search times decrease slightly as we add more ingredients as expected. The results for forbidden categories are similar.

The number of meal parameters specified do not appear to have any predictable effect on search times, in the parameter model the time decreases from zero to one parameter, increases from one to two parameters and decreases again from two to three parameters. In the recipe model search times decrease up to two parameters but increase from two to three parameters.

Nutritional values seem to increase search times slightly for each value added up to four, more greatly so in the parameter model. Above four values the search times are fairly level for the recipe model and decreasing for the parameter model.

The number of subjects hardly affects the parameter model at all but increase search times rather linearly for the recipe model, again this is probably due to the fact that recipe rating constraints are evaluated only at the bottom of the search tree in the parameter model.

Size of the recipe database turns out to be one of the most important parameters, already at 300 recipes the parameter model fails to provide a two meal plan in a reasonable time. The recipe model scales better and provides two meal plans for database sizes of up to 3000 recipes when no restrictions are made. When we increase the restrictions, in form of nutritional requirements, meal parameters and number of subjects planned for, both models fail at 300 recipes but the recipe model does so to a far lesser degree.

The other important parameter is the number of meals planned. The parameter model scales rather linearly with the number of meals planned but the recipe model is awful when it comes to dealing with a large number of meals, already at four meals planned with no restrictions the search times are much too high. When increasing the restrictions the parameter model retains

its linear behaviour starting at a higher search time for one meal while the recipe model fails at three meals planned.

10.1.1 Model and algorithm comparison

Generally we see that the recipe model is faster than the parameter model but for a few parameters it is outclassed. When the number of subjects is large or when planning a larger number of meals the parameter model is superior because it scales a lot better.

Looking at all the different test runs we see that Partial Forward Checking has an advantage over the other algorithms. There are a few cases where it is not the fastest but with a few improvements to the system it is likely to outperform the competition regardless of parameters. For this reason PFC was used for all further testing.

10.1.2 Upper Bound development

The tests regarding the Upper Bound's decrease during search were performed to evaluate the possibility of terminating the search early. Either based on the number of complete assignments made or time spent searching.

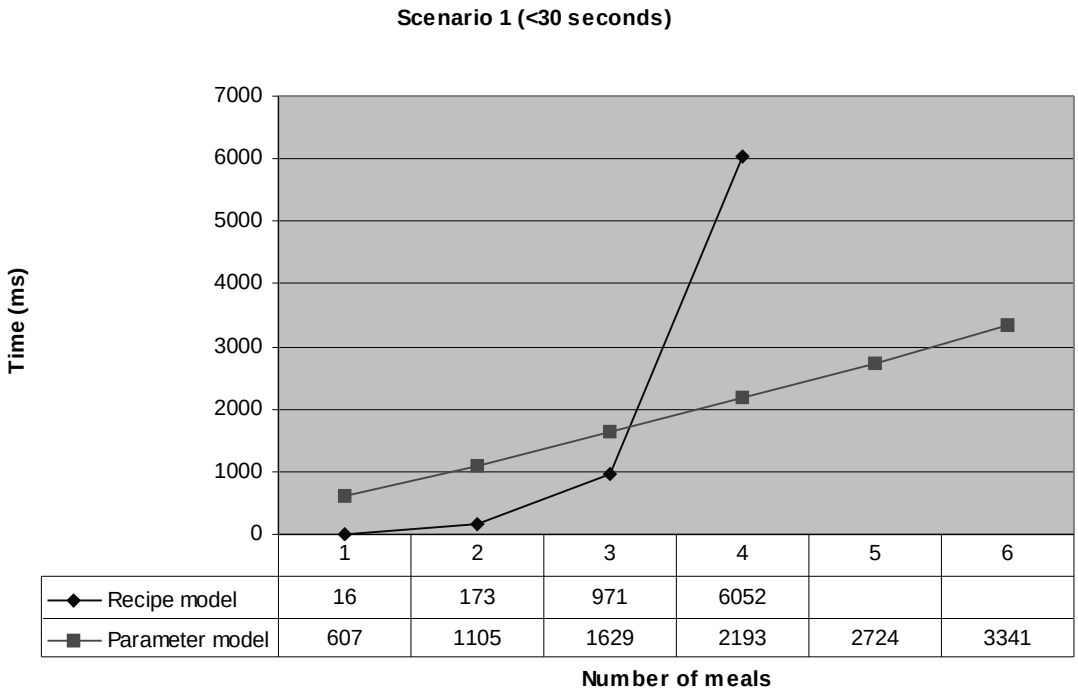
The tests show that the upper bound decreases more slowly towards the end of the search, much more so in terms of time spent than number of complete assignments. Looking at what would happen if we always halted the search when five seconds had passed we see that for more than 1000 recipes using the parameter model we would not get a result at all. For a smaller amount of recipes or using the recipe model however, our tests show that we would get a 2-23.5%[8] decrease in plan quality. Searches finished in less than 5 seconds not included.

10.2 Scenario results

All scenario tests were run using the Partial Forward Checking algorithm. Times above 30 seconds are not included.

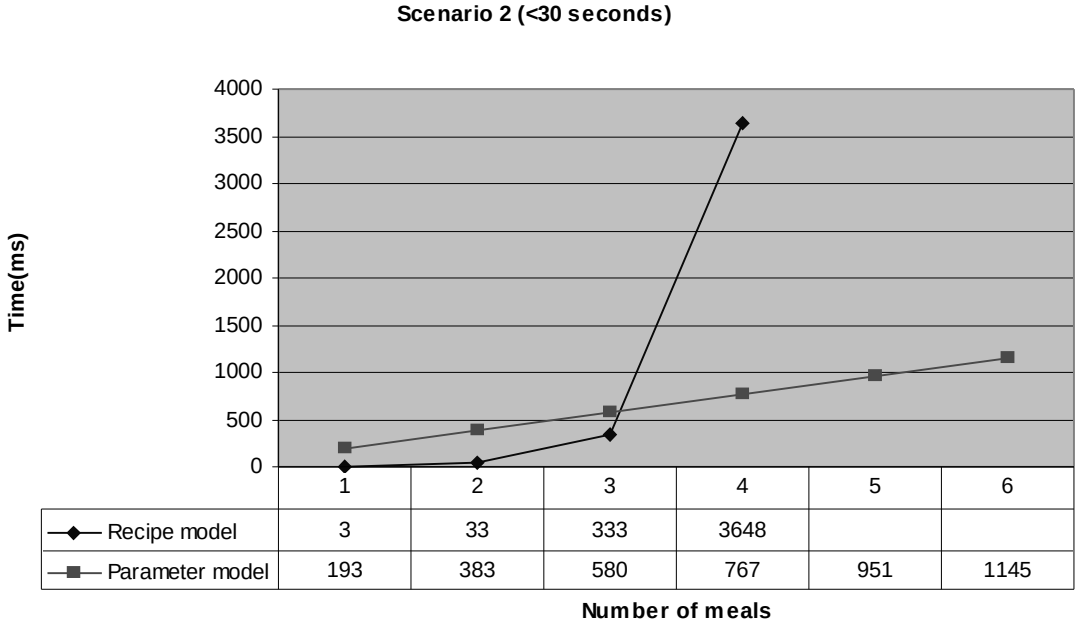
[8] See the last table in Appendix A for full results of a five second limit

10.2.1 The family



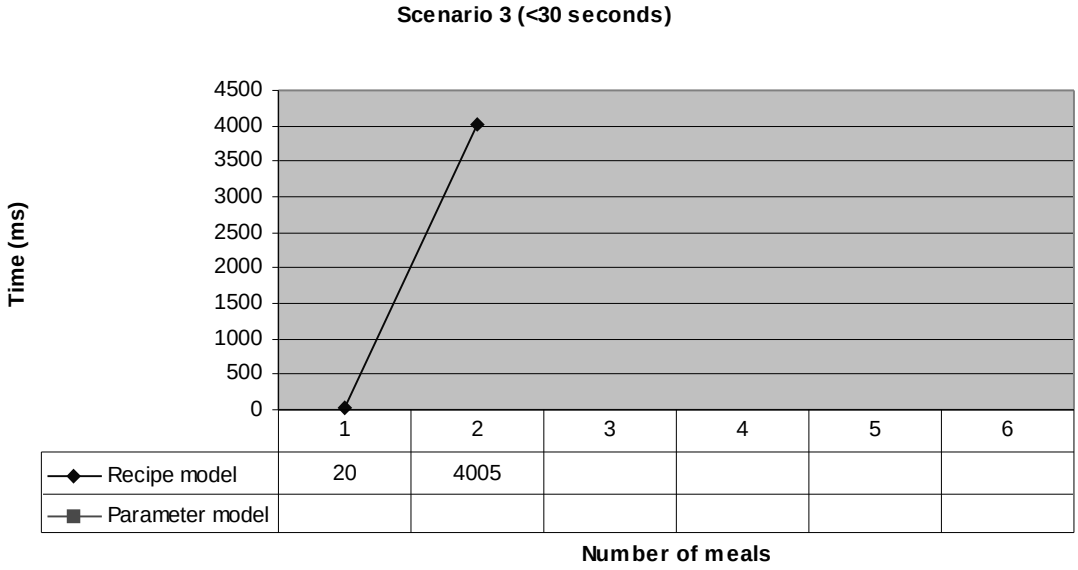
When planning three meals or less the recipe model is faster for these settings but as soon as we want to plan four or more meals we are much better off using the parameter model.

10.2.2 The student



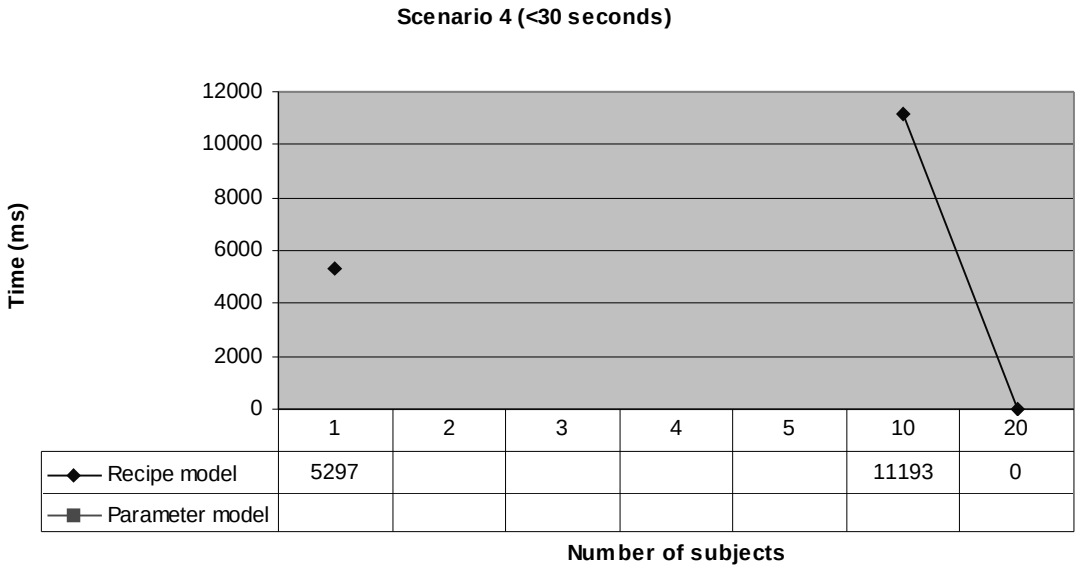
These results are very similar to the first scenario except that the times are lower as a result of there being only one subject and a very small recipe database.

10.2.3 The senior citizen



With a large number of nutritional constraints and suggestions as well as an increased number of recipes in the database we run into problems. The parameter model fails immediately and the recipe model can only handle two meals.

10.2.4 The retirement home



Again the parameter model fails for the smallest value due to the large number of constraints and size of the recipe database. As the number of constraints grow with each subject added we see the recipe model failing already for two subjects due to the large number of meals planned. Between five and ten subjects the hard constraints give us a very small set of valid assignments so search times start decreasing. At twenty subjects there are no valid assignments remaining.

11 Conclusions

For small or simple plans the meal planning system is usable in its current state but when we start adding restrictions, increasing the size of the recipe database or the number of meals planned the search times go out of hand. Looking at the general test results we see that the recipe model is faster than the parameter model for most settings but when the number of subjects, and thus recipe rating constraints, increase beyond ten the parameter model gains the upper hand. This is likely the result of those constraints not being evaluated as often as for the recipe model. The parameter model also scales a lot better with regard to the number of meals planned so both models have useful applications.

Comparing the various algorithms it seems that Partial Forward Checking is generally the fastest but there are a few exceptions. With a stronger constraint model and incremental calculation as described in the next chapter the advantage of PFC would probably be greater.

If we want to use the current system it would seem that a small recipe database should be used and if we need to use a lot of restrictions when planning for a large number of meals several shorter plans are preferable. This will result in poorer variation but otherwise provide us with a decent sized number of meals in a reasonable timeframe.

11.1 Realistic limits

In the current state of the system we would recommend using a small recipe database around 100 recipes. If planning just a few meals for few subjects the recipe model should be used but for larger plans the parameter model is superior. As long as the recipe database is kept small it seems that we can use the maximum number of restrictions without problems unless we start planning a large number of meals.

11.2 5 years from now - Moores law

Supporters of Moore's law believe that computer hardware capacity doubles in approximately 18 months and will continue to do so for some time. Assuming this is true, what would it mean for the meal planning system? With the exception of recipe database size for the parameter model and number of meals planned for the recipe model the search times increase rather linearly[9] with increased plan complexity so with a fixed database size and number of meals it seems likely that plan complexity could be at least eight (2^3) times as high.

[9] Can be seen in figures A-1—A-5 in Appendix A

Looking at the size of the recipe database or the number of meals planned though the increase in search times is alarming[10]. While an eightfold increase in computing power would certainly allow us to create more complex plans the search times in the above mentioned cases still grow a lot faster than hardware capacity. Any search that goes beyond 40 seconds today will likely be unacceptable in five years as well as expectations on computing speed increase. As can be seen in the tables following figures A-11 and A-12 in Appendix A search times above 40 seconds are common with the current implementation. It's obvious that these issues need to be addressed if the system is to be used for larger applications in the future.

[10] Can be seen in figures A-6—A-7 in Appendix A

12 Future work

In this chapter we present some ideas on how to improve performance that did not fit in the timeframe of the project.

12.1 Stronger constraint model

The current implementation requires a constraint to have a weight and a valuation function. In addition to this we suggest that a constraint keeps track of the set of variables involved in the constraint. The set should preferably be no larger than two variables as described in the next section or the improvements will be limited. This should save a lot of unnecessary method calls to valuation functions that will always return 0 and allow us to improve the various solving algorithms by only checking constraints that involve the current variable and a previously assigned one. It would also allow us to perform incremental calculation of LB in the case of PFC by adding the minimal ic value of each propagation round to LB.

The `UpdateLowerBound` function of Algorithm 3 would then look something like this:

```
procedure UpdateLowerBound(LB)
  LB = LB + lowestICFoundDuringPropagation
end
```

12.2 Binary constraints

Using n-ary constraints reduces the performance of (Partial) Forward Checking a lot. Since these constraints need to be evaluated for a lot of assignments the improvement over regular Depth First Branch and Bound is not as significant as it could be. We suggest that the current n-ary constraints are converted into several binary constraints or otherwise remodelled in order to take full advantage of FC and the incremental calculation of inconsistency counts in the PFC algorithm. This improvement in conjunction with the stronger constraint model suggested in 12.1 should show a significant improvement as the depth of the search tree increases.

12.3 Parallel computation

As an alternative to the multiple method calls used in the parameter model we had some thoughts about an algorithm that worked with several assignments in parallel. This would prevent the rapidly increasing search depth but present us with a few new problems.

Implemented as a single thread the approach would be equivalent to increasing domain sizes exponentially for each additional meal, which certainly might not be an improvement over a linear increase of search depth.

For example if variable X of assignment 1 had a domain of {a, b} and variable Y of assignment 2 had a domain of {c, d} what we essentially would be doing is creating a new variable XY with a domain of {{a, c}, {a, d}, {b, c}, {b, d}}.

What we would like to do is perform the assignments in separate threads to take advantage of multiprocessor systems, hyperthreading technology and the upcoming dual core processors. This approach would result in the standard threading problems of synchronization as constraints would need to be allowed between variables of different assignments but those should be solvable. The algorithm might quickly become very advanced as we try to avoid unnecessary constraint checks and use techniques similar to FC but it would be interesting to try.

12.4 Dynamic model selection

Since the recipe model and the parameter model suffer from different problems it would be preferable if the system could dynamically select the solving model that is likely to provide the fastest solution depending on plan parameters.

12.5 User feedback

We would like the system to warn the user when she has selected plan parameters that are likely to result in unacceptable search times. In conjunction with this warning a solution should be suggested. For example it could suggest planning fewer meals at once or using a smaller selection of recipes.

12.6 Automatic search termination

The system should be able to terminate the search early if it takes too long. For example setting a maximum search time of five seconds would result in a 2-23.5% decrease[11] in plan quality, measured in Upper Bound terms, but allow the user to create more complex plans.

12.7 Decreasing UB

Looking at figure A-9 in Appendix A we can see that the search for 3000 recipes that starts at a higher UB value than the others requires a lot more assignments before it reaches a solution. We also see that the final UB value is close to those of the other searches. This is something we might be able to take advantage of by running a fast search with a smaller recipe database to establish a good starting UB for the more complex search.

[11] See section 10.1.2

12.7 Taking advantage of recipe model assignment properties

Disregarding availability and temporary (meal-specific) constraints an assignment of {R1, R2} is equivalent to the assignment {R2, R1} as far as the search algorithms are concerned when using the recipe model. We might be able to take advantage of this by first running a modified version of the chosen algorithm, without the mentioned constraints, that never tries to assign a permutation of an earlier assignment. We then run a second search with the standard algorithm, using only the previously removed constraints and with the recipe database reduced to only include the recipes from the best simplified solution, to obtain the final result. Note that the constraints used for the first run do not need to be included for the second run as all possible assignments in the second run will have the same valuation for those constraints.

The smaller number of possible assignments and fewer constraints from the first run combined with the smaller domain sizes and fewer constraints of the second run might result in a faster solution, especially when the number of meals planned increases.

13 References

[1] Bacchus, Fahiem; Grove, Adam (1995). *On The Forward Checking Algorithm.* Proceedings of the First International Conference on Principles and Practice of Constraint Programming, pag. 292 – 308.

[2] Bacchus, Fahiem; Grove, Adam (1999). *Looking Forward in Constraint Satisfaction Algorithms.* Unpublished manuscript.

[3] Bacchus, Fahiem; van Beek, P (1998). *On the Conversion between Non-Binary and Binary Constraint Satisfaction Problems.* Proceedings of the 15th National Conference on Artificial Intelligence ({AAAI}-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence ({IAAI}-98), pag. 311 – 318.

[4] Bacchus, Fahiem; van Run, Paul (1995). *Dynamic Variable Ordering in CSPs.* In Proceedings the First International Conference on Principle and Practice of Constraint Programming, pag. 258 – 275.[5] Biso, Alessandro; Rossi, Francesca; Sperduti, alessandro (2000). *Eperimental results on Learning Soft Constraints.* Principles of Knowledge Representation and Reasoning, pag. 435 – 444.

[6] Cohen, David; Cooper, Martin; Jeavons, Peter; Korokhin, Andrei (2003). *Soft constraints: complexity and multimorphisms.* Principles and Practice of Constraint Programming 2003, pag. 244 – 258.

[7] Cohen, David; Cooper, Martin; Jeavons, Peter; Korokhin, Andrei (2003). *A Maximal Tractable Class of Soft Constraints.* Journal of Artificial Intelligence Research (JAIR). Volume 22, pag. 1 – 22.

[8] Dechter R.; Frost D. (1998). *Backtracking Algorithms for Constraint Satisfaction Problems.* Constraints, International Journal.

[9] Dechter R; Frost D. (2001). *Backjump-based Backtracking for Constraint Satisfaction Problems.* Artificial Intelligence. Volume 136, Issue 2 2002, pag. 147 – 188.

[10] Deshpande, Mukund; Karypkis, George (2004). *Item-Based Top-N Recommendation Algorithms.* ACM Transactions on Information Systems (TOIS). Volume 22 , Issue 1 (January 2004) , pag. 143 – 177.

[11] Khanna, Sanjeev; Sudan, Madhu; Trevisan, Luca; Williamson, David P. (2001). *The Approximability of Constraint Satisfaction Problems.* SIAM Journal on Computing. Volume 30, Issue 6 2001, pag. 1863 – 1920.

[12] Sarwar, Badrul; Karypkis, George; Konstan, Joseph; Riedl, John (2001). *Item-Based Collaborative Filtering Recommendation Algorithms.* In Proceedings of the 10th International World Wide Web Conference (WWW10), Hong Kong, May 2001, pag. 285 – 295.

[13] Torrens, Marc (2002). *Scalable Intelligent Electronic Catalogs.* Ph.D. Thesis, #2690, Swiss Federal Institute of Technology (EPFL).

[14] Torrens, Marc; Faltings, Boi (2002). *Using Soft CSPs for Approximating Pareto-Optimal Solution Sets.* Workshop on Preferences in AI and CP: Symbolic Approaches. Technical Report WS-02-13, pag. 99 – 106.

[15] Woodger Computing Inc. (2005). *Multi-Tier Architectures.* http://www.woodger.ca/archmult.htm, last accessed 2005-07-20.

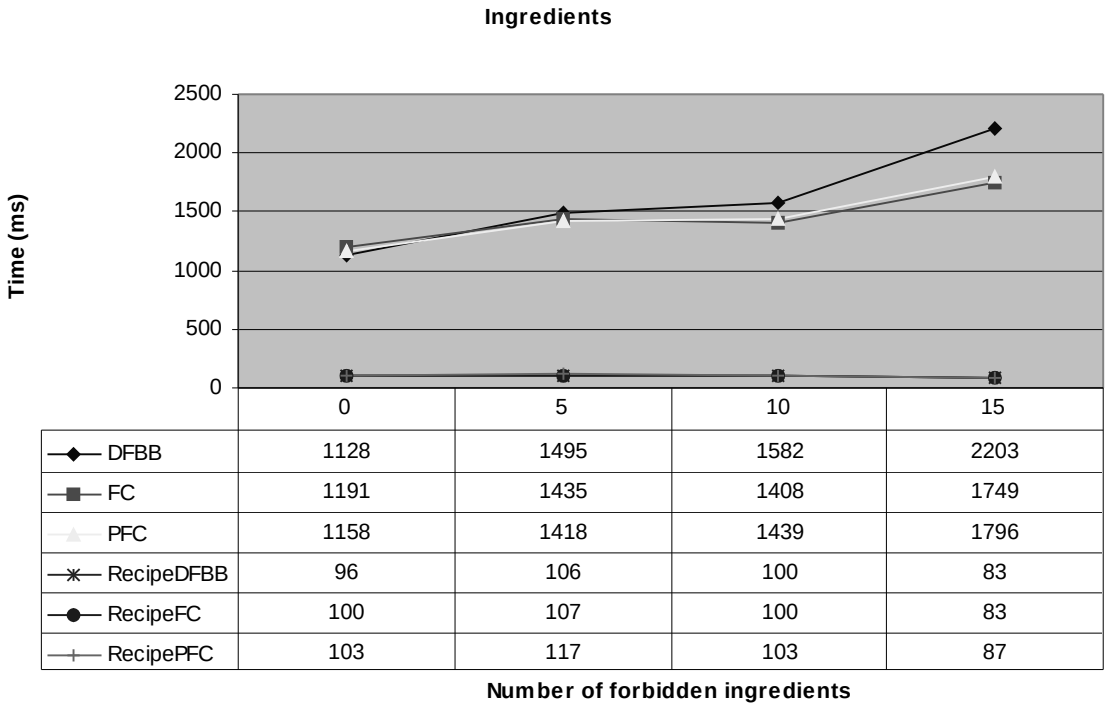
Appendix A – Test results

Tests were run under Windows XP on a 1.2 GHz AMD Athlon system with 768 MB of RAM. The parameters used for the tests were 100 recipes, 2 meals, 1 subject, 1 suggestion and 0 of the other parameters unless otherwise stated. The tests for recipes and meals were also run with increased parameters for comparison.

The parameter model had increased values of 20 subjects, 3 meal parameters and 7 nutritional values. For the recipe model they were 5 subjects, 3 meal parameters and 7 nutritional values. These increased values were also used for the UB measurements, where the objective was to see how early termination of a complex search would affect plan quality.

Execution times are measured in milliseconds and times above 30 seconds are not included except in the UB tests.

Figure A-1

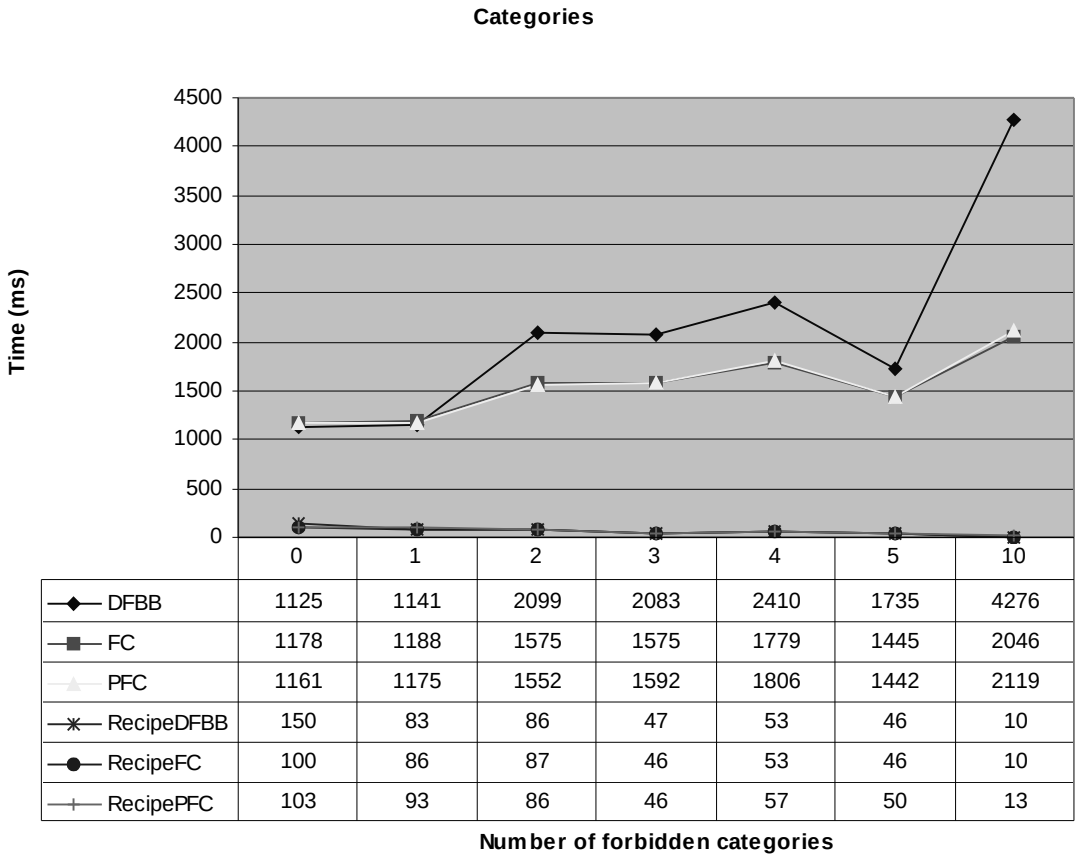


	0	5	10	15
DFBB	1128	1495	1582	2203
FC	1191	1435	1408	1749
PFC	1158	1418	1439	1796
RecipeDFBB	96	106	100	83
RecipeFC	100	107	100	83
RecipePFC	103	117	103	87

The parameters used for this test were 100 recipes, 2 meals, 1 subject, 1 suggestion, 0 meal parameters, 0 nutritional requirements and 0 forbidden categories. The number of forbidden ingredients was varied.

DFBB is the standard Depth First Branch and Bound algorithm, FC is the Forward Checking algorithm and PFC is the Partial Forward Checking algorithm. Results with the Recipe prefix use the recipe model[12] while those without the prefix use the parameter model[13].

[12] Section 3.3.2
[13] Section 3.3.1

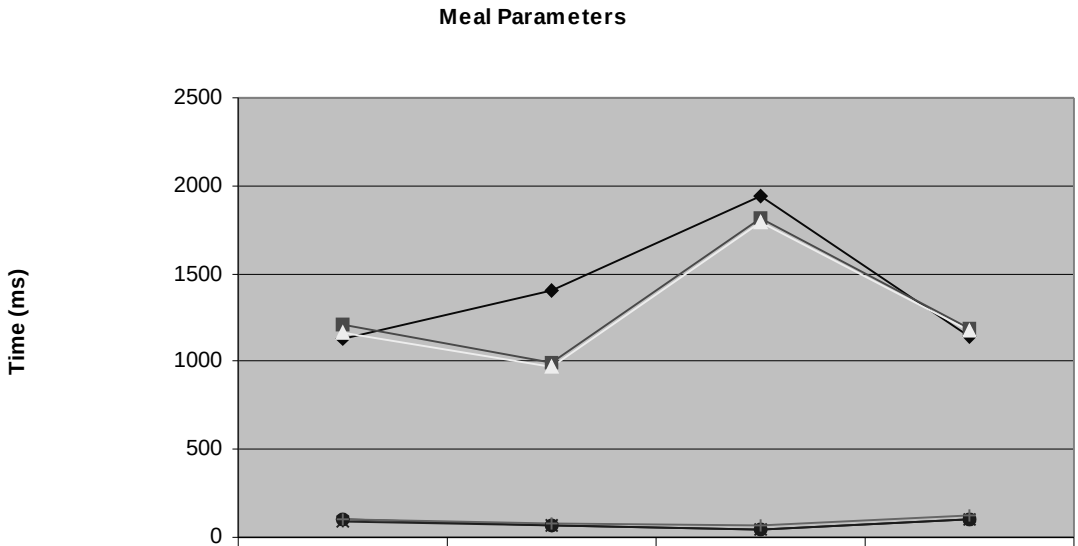


	0	1	2	3	4	5	10
DFBB	1125	1141	2099	2083	2410	1735	4276
FC	1178	1188	1575	1575	1779	1445	2046
PFC	1161	1175	1552	1592	1806	1442	2119
RecipeDFBB	150	83	86	47	53	46	10
RecipeFC	100	86	87	46	53	46	10
RecipePFC	103	93	86	46	57	50	13

The parameters used for this test were 100 recipes, 2 meals, 1 subject, 1 suggestion, 0 meal parameters, 0 nutritional requirements and 0 forbidden ingredients. The number of forbidden categories was varied.

DFBB is the standard Depth First Branch and Bound algorithm, FC is the Forward Checking algorithm and PFC is the Partial Forward Checking algorithm. Results with the Recipe prefix use the recipe model[14] while those without the prefix use the parameter model[15].

[14] Section 3.3.2
[15] Section 3.3.1

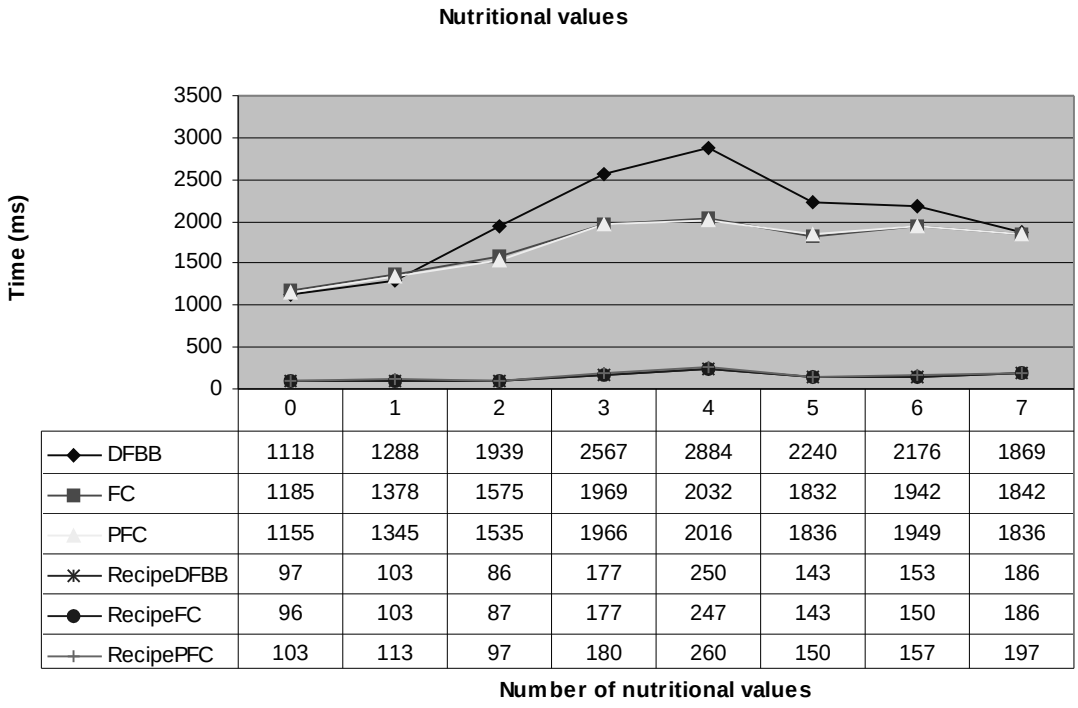


	0	1	2	3
DFBB	1128	1402	1939	1141
FC	1208	995	1812	1188
PFC	1165	971	1789	1181
RecipeDFBB	96	63	46	103
RecipeFC	100	63	47	103
RecipePFC	106	76	66	120

The parameters used for this test were 100 recipes, 2 meals, 1 subject, 1 suggestion, 0 nutritional requirements 0 forbidden ingredients and 0 forbidden categories. The number of meal parameters was varied.

DFBB is the standard Depth First Branch and Bound algorithm, FC is the Forward Checking algorithm and PFC is the Partial Forward Checking algorithm. Results with the Recipe prefix use the recipe model[16] while those without the prefix use the parameter model[17].

[16] Section 3.3.2
[17] Section 3.3.1

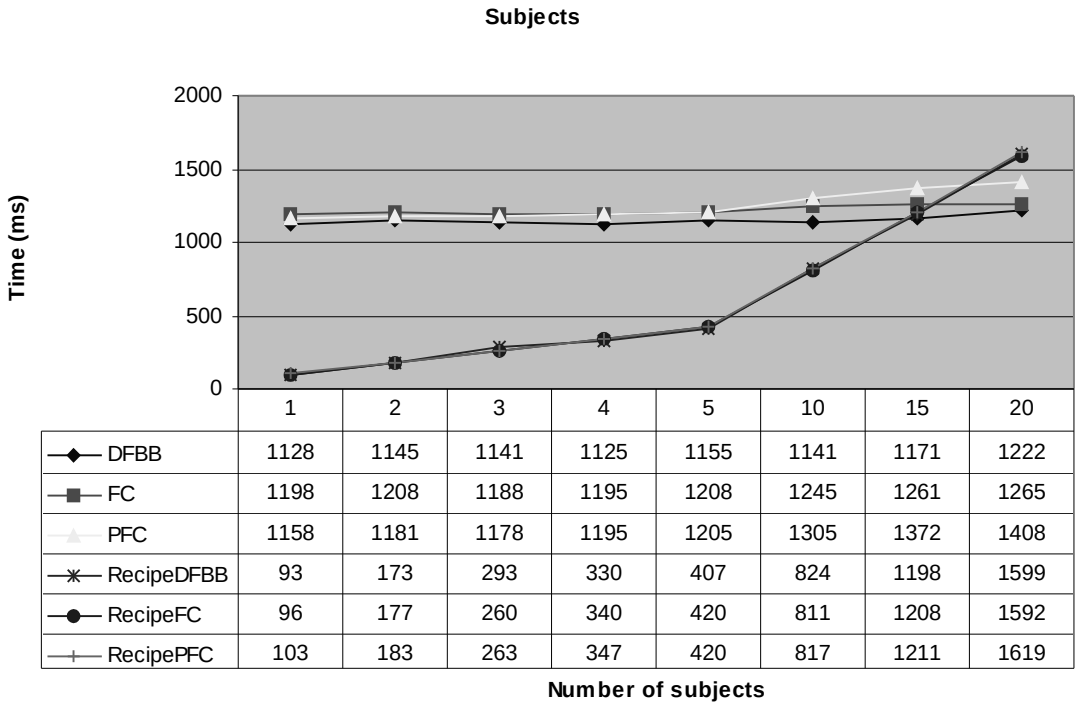


	0	1	2	3	4	5	6	7
DFBB	1118	1288	1939	2567	2884	2240	2176	1869
FC	1185	1378	1575	1969	2032	1832	1942	1842
PFC	1155	1345	1535	1966	2016	1836	1949	1836
RecipeDFBB	97	103	86	177	250	143	153	186
RecipeFC	96	103	87	177	247	143	150	186
RecipePFC	103	113	97	180	260	150	157	197

The parameters used for this test were 100 recipes, 2 meals, 1 subject, 1 suggestion, 0 meal parameters, 0 forbidden ingredients and 0 forbidden categories. The number of nutritional requirements was varied.

DFBB is the standard Depth First Branch and Bound algorithm, FC is the Forward Checking algorithm and PFC is the Partial Forward Checking algorithm. Results with the Recipe prefix use the recipe model[18] while those without the prefix use the parameter model[19].

[18] Section 3.3.2
[19] Section 3.3.1

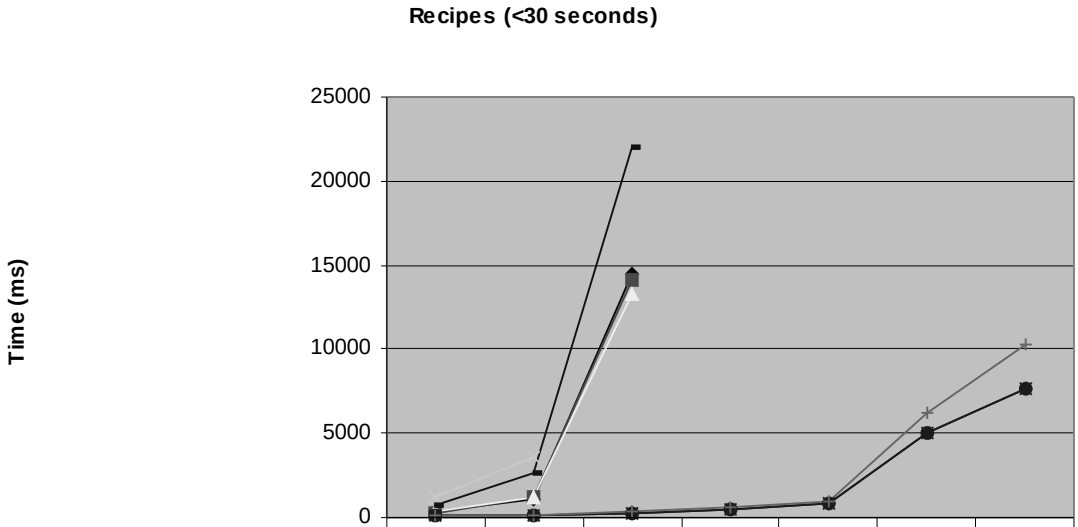


	1	2	3	4	5	10	15	20
DFBB	1128	1145	1141	1125	1155	1141	1171	1222
FC	1198	1208	1188	1195	1208	1245	1261	1265
PFC	1158	1181	1178	1195	1205	1305	1372	1408
RecipeDFBB	93	173	293	330	407	824	1198	1599
RecipeFC	96	177	260	340	420	811	1208	1592
RecipePFC	103	183	263	347	420	817	1211	1619

The parameters used for this test were 100 recipes, 2 meals, 1 suggestion, 0 meal parameters, 0 nutritional requirements 0 forbidden ingredients and 0 forbidden categories. The number of subjects was varied.

DFBB is the standard Depth First Branch and Bound algorithm, FC is the Forward Checking algorithm and PFC is the Partial Forward Checking algorithm. Results with the Recipe prefix use the recipe model[20] while those without the prefix use the parameter model[21].

[20] Section 3.3.2
[21] Section 3.3.1



	50	100	300	500	1000	3000	5000
DFBB	350	1135	14531				
FC	280	1218	14133				
PFC	320	1211	13299				
PFC w / Increased defaults	1205	3545					
RecipeDFBB	63	100	283	504	808	5047	7707
RecipeFC	66	126	287	510	821	5030	7617
RecipePFC	77	107	327	571	968	6265	10308
RecipePFCIncreasedDefault	734	2577	21998				

The parameters used for this test were 2 meals, 1 subject, 1 suggestion, 0 meal parameters, 0 nutritional requirements 0 forbidden ingredients and 0 forbidden categories. The number of recipes was varied.

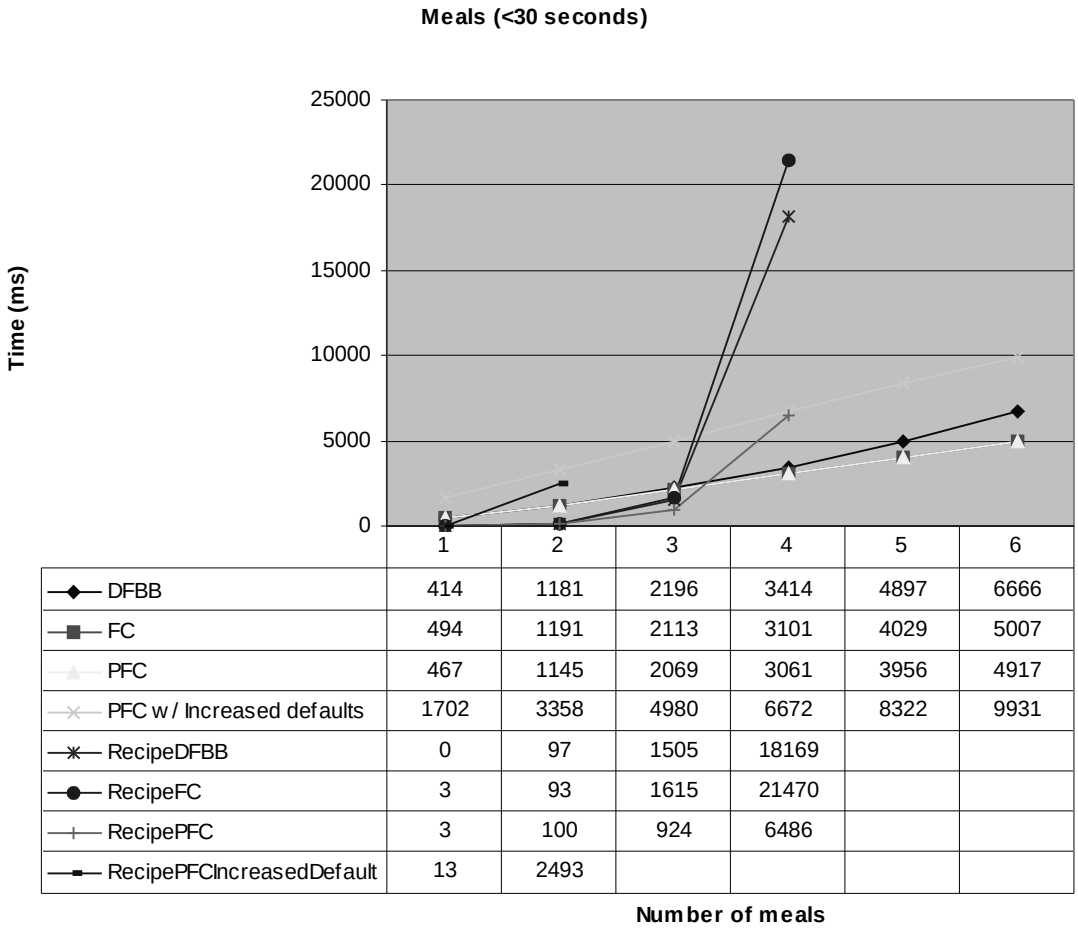
The PFC w/ increased defaults tests used increased parameters. The number of subjects was increased to 20, the number of meal parameters to 3 and the number of nutritional requirements to 7.

The RecipePFCIncreasedDefaults tests also used increased parameters. The number of subjects was increased to 5, the number of meal parameters to 3 and the number of nutritional requirements to 7.

DFBB is the standard Depth First Branch and Bound algorithm, FC is the Forward Checking algorithm and PFC is the Partial Forward Checking algorithm. Results with the Recipe prefix use the recipe model[22] while those without the prefix use the parameter model[23].

[22] Section 3.3.2
[23] Section 3.3.1

Figure A-2



	1	2	3	4	5	6
DFBB	414	1181	2196	3414	4897	6666
FC	494	1191	2113	3101	4029	5007
PFC	467	1145	2069	3061	3956	4917
PFC w / Increased defaults	1702	3358	4980	6672	8322	9931
RecipeDFBB	0	97	1505	18169		
RecipeFC	3	93	1615	21470		
RecipePFC	3	100	924	6486		
RecipePFCIncreasedDefault	13	2493				

The parameters used for this test were 100 recipes, 1 subject, 1 suggestion, 0 meal parameters, 0 nutritional requirements 0 forbidden ingredients and 0 forbidden categories. The number of meals was varied.

The PFC w/ increased defaults tests used increased parameters. The number of subjects was increased to 20, the number of meal parameters to 3 and the number of nutritional requirements to 7.

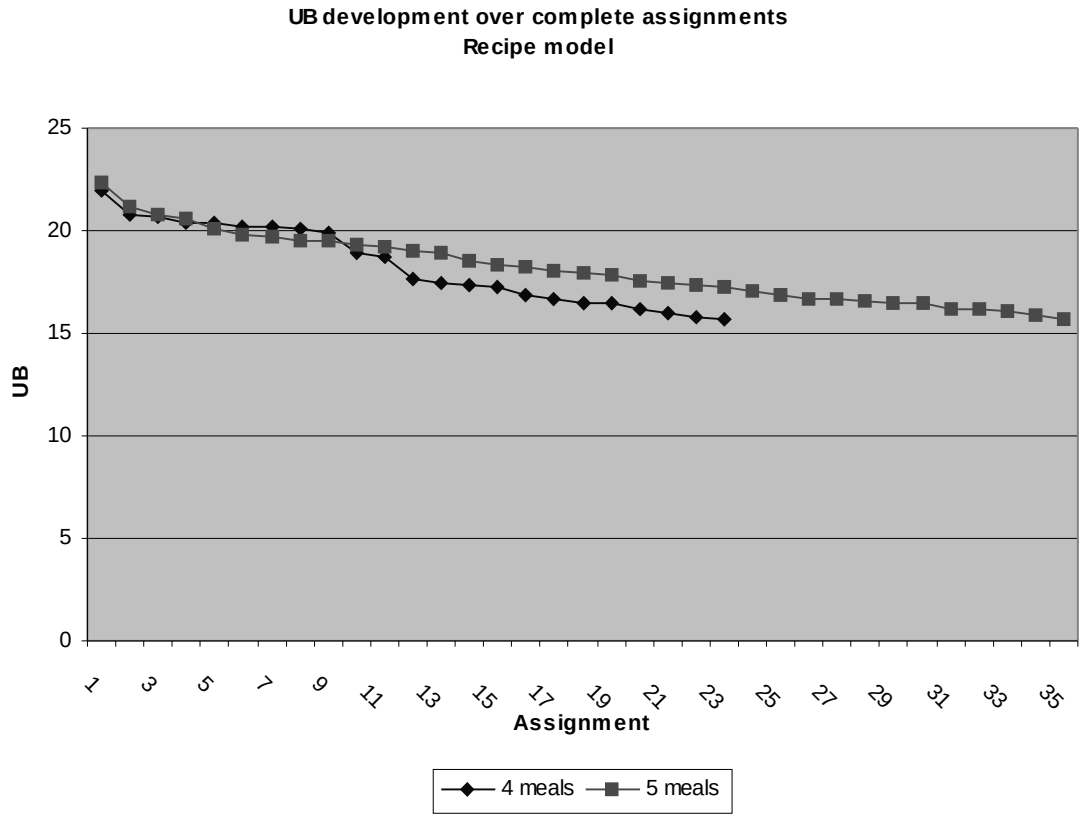
The RecipePFCIncreasedDefaults tests also used increased parameters. The number of subjects was increased to 5, the number of meal parameters to 3 and the number of nutritional requirements to 7.

DFBB is the standard Depth First Branch and Bound algorithm, FC is the Forward Checking algorithm and PFC is the Partial Forward Checking

algorithm. Results with the Recipe prefix use the recipe model[24] while those without the prefix use the parameter model[25].

[24] Section 3.3.2
[25] Section 3.3.1

Figure A-3

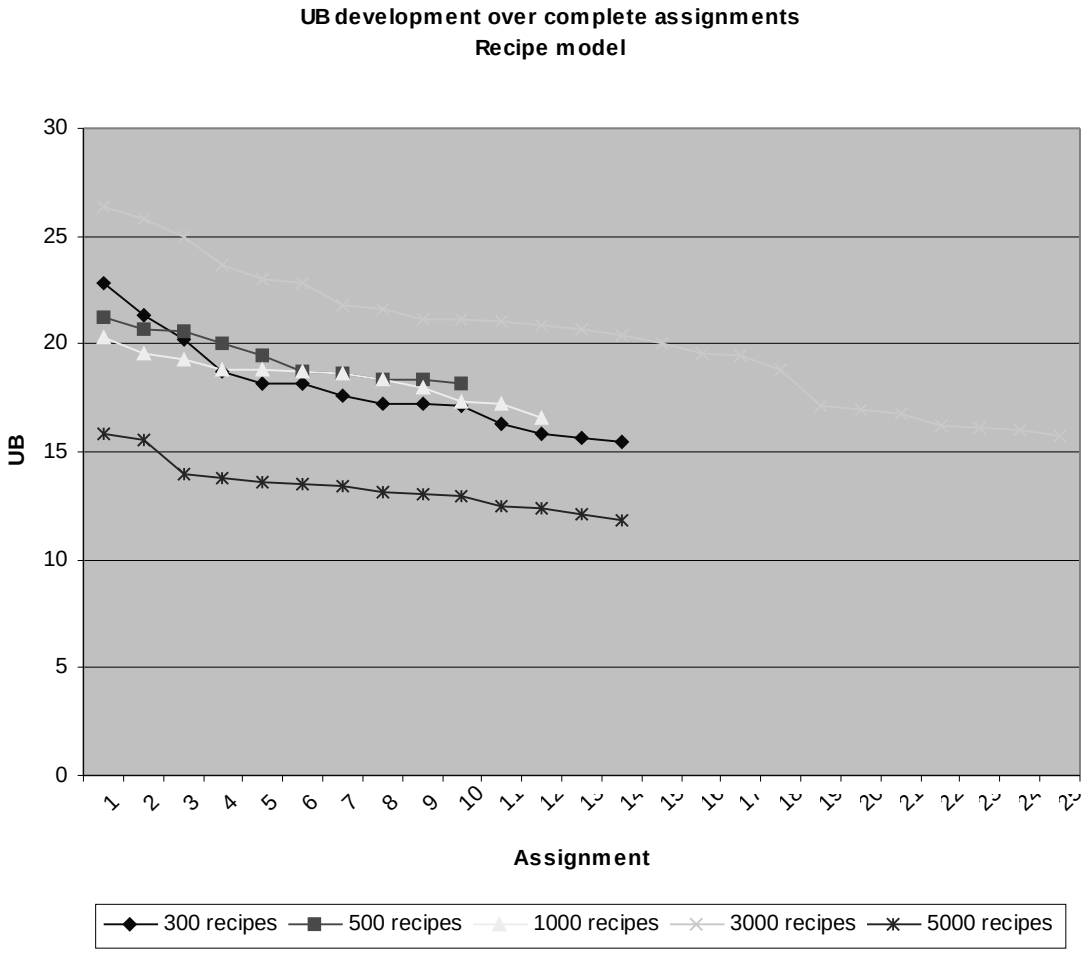


The parameters used for this test were 100 recipes, 5 subjects, 1 suggestion, 3 meal parameters, 7 nutritional requirements 0 forbidden ingredients and 0 forbidden categories. The algorithm used was Partial Forward Checking.

4 meals								
Assignment	1	2	3	4	5	6	7	8
UB	21,99	20,75	20,7	20,42	20,42	20,18	20,17	20,14
Assignment	9	10	11	12	13	14	15	16
UB	19,95	18,89	18,74	17,66	17,49	17,4	17,24	16,83
Assignment	17	18	19	20	21	22	23	
UB	16,71	16,5	16,49	16,2	16	15,76	15,73	

5 meals								
Assignment	1	2	3	4	5	6	7	8
UB	22,35	21,2	20,74	20,58	20,07	19,79	19,68	19,53
Assignment	9	10	11	12	13	14	15	16
UB	19,49	19,28	19,24	19,01	18,89	18,52	18,31	18,26
Assignment	17	18	19	20	21	22	23	24
UB	18,06	17,91	17,88	17,54	17,43	17,36	17,27	17,01
Assignment	25	26	27	28	29	30	31	32
UB	16,83	16,69	16,65	16,6	16,48	16,47	16,22	16,16
Assignment	33	34	35					
UB	16,08	15,93	15,72					

Figure A-4



The parameters used for this test were 2 meals, 5 subjects, 1 suggestion, 3 meal parameters, 7 nutritional requirements 0 forbidden ingredients and 0 forbidden categories. The algorithm used was Partial Forward Checking.

300 recipes								
Assignment	1	2	3	4	5	6	7	8
UB	22,82	21,31	20,19	18,71	18,2	18,16	17,6	17,27
Assignment	9	10	11	12	13	14		
UB	17,2	17,13	16,26	15,86	15,67	15,47		

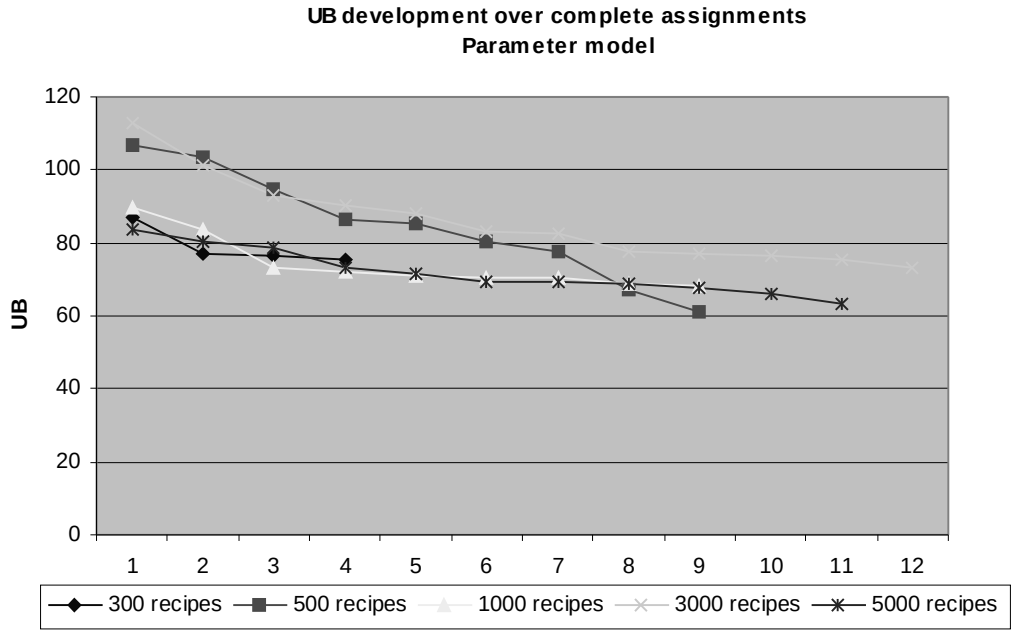
500 recipes								
Assignment	1	2	3	4	5	6	7	8
UB	21,25	20,72	20,59	20,06	19,47	18,7	18,6	18,38
Assignment	9	10						
UB	18,31	18,21						

1000 recipes								
Assignment	1	2	3	4	5	6	7	8
UB	20,32	19,52	19,26	18,86	18,79	18,73	18,65	18,31
Assignment	9	10	11	12				
UB	18	17,34	17,24	16,62				

3000 recipes								
Assignment	1	2	3	4	5	6	7	8
UB	26,38	25,81	24,94	23,63	23,04	22,84	21,78	21,59
Assignment	9	10	11	12	13	14	15	16
UB	21,15	21,12	21,02	20,83	20,68	20,39	20,07	19,57
Assignment	17	18	19	20	21	22	23	24
UB	19,49	18,86	17,11	16,93	16,77	16,21	16,13	16,06
Assignment	25							
UB	15,76							

5000 recipes								
Assignment	1	2	3	4	5	6	7	8
UB	15,83	15,55	14	13,81	13,57	13,5	13,4	13,17
Assignment	9	10	11	12	13	14		
UB	13,09	12,93	12,48	12,39	12,15	11,79		

Figure A-5



The parameters used for this test were 1 meal, 20 subjects, 1 suggestion, 3 meal parameters, 7 nutritional requirements 0 forbidden ingredients and 0 forbidden categories. The algorithm used was Partial Forward Checking.

300 recipes				
Assignment	1	2	3	4
UB	86,74	76,88	76,76	75,2

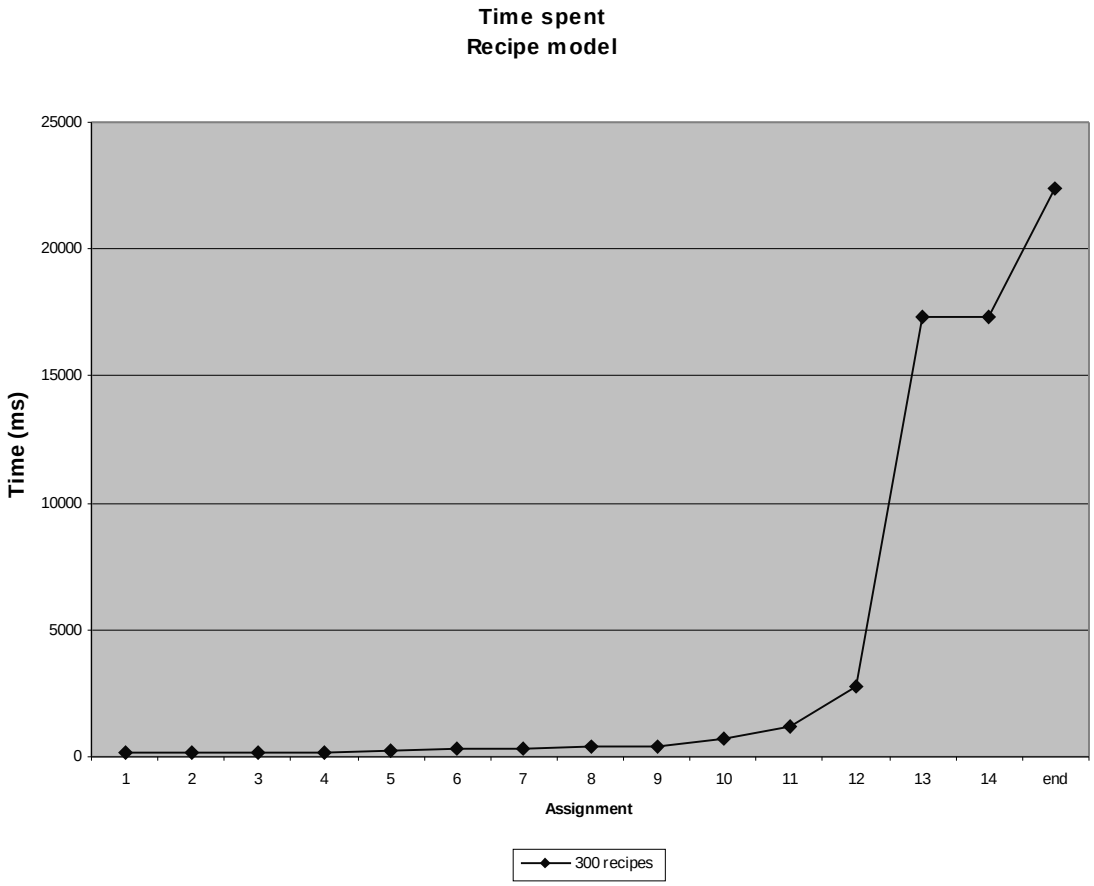
500 recipes								
Assignment	1	2	3	4	5	6	7	8
UB	106,6	103,4	94,84	86,18	85,16	80,42	77,48	67,26
Assignment	9							
UB	61,1							

1000 recipes								
Assignment	1	2	3	4	5	6	7	8
UB	89,86	83,66	72,98	72,22	70,8	70,42	70,34	68,86
Assignment	9							
UB	68,52							

3000 recipes								
Assignment	1	2	3	4	5	6	7	8
UB	113,08	101,48	93,06	90,38	87,9	83,3	82,8	77,56
Assignment	9	10	11	12				
UB	77,1	76,72	75,36	73,2				

5000 recipes								
Assignment	1	2	3	4	5	6	7	8
UB	83,48	80,38	78,76	73,02	71,7	69,48	69,36	69,02
Assignment	9	10	11					
UB	67,96	66,08	63,52					

Figure A-6



Only the graph for 300 recipes is shown here but the others show similar properties, as can be seen in the data tables.

The parameters used for this test were 2 meals, 5 subjects, 1 suggestion, 3 meal parameters, 7 nutritional requirements 0 forbidden ingredients and 0 forbidden categories. The algorithm used was Partial Forward Checking.

300 recipes								
Assignment	1	2	3	4	5	6	7	8
Time (ms)	130	130	130	140	200	321	321	411
Assignment	9	10	11	12	13	14	end	
Time (ms)	411	741	1152	2744	17355	17355	22382	

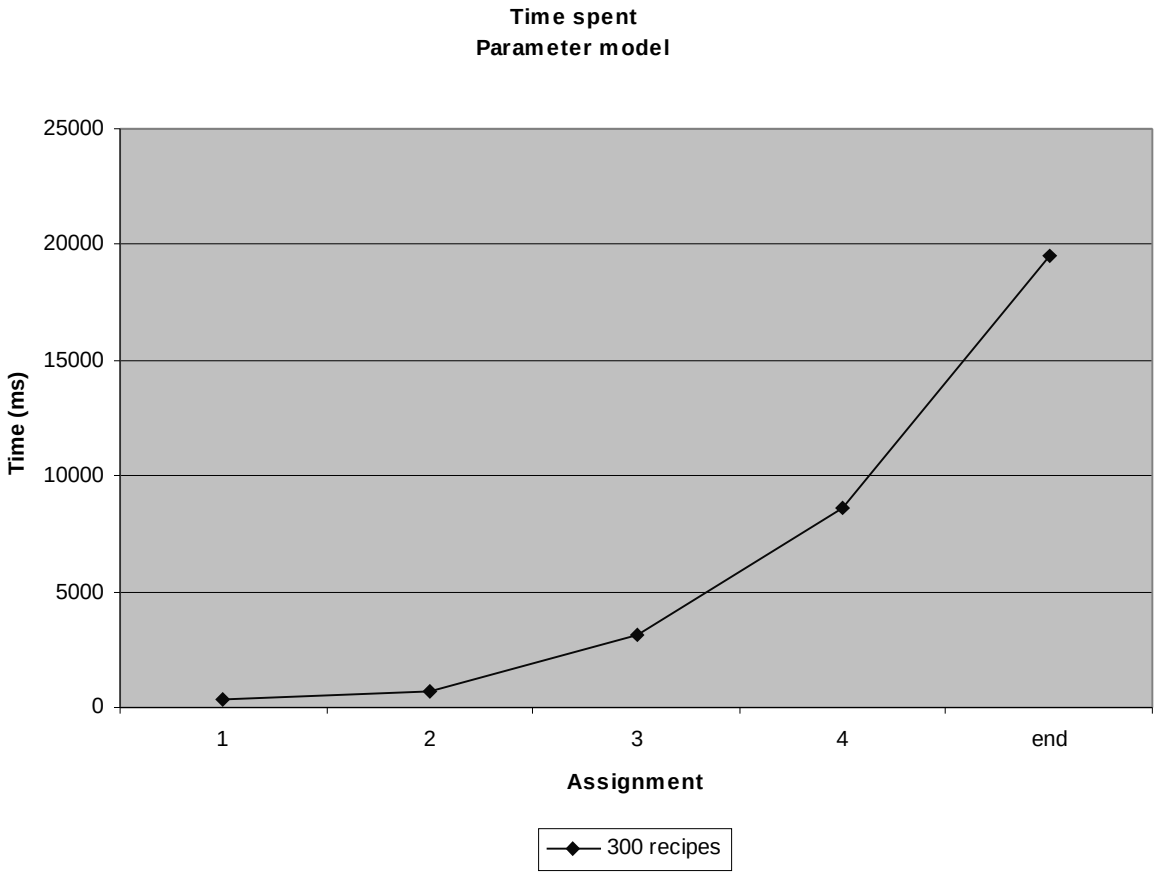
500 recipes								
Assignment	1	2	3	4	5	6	7	8
Time (ms)	201	211	311	551	551	2023	4617	22933
Assignment	9	10	end					
Time (ms)	31626	50223	71073					

1000 recipes								
Assignment	1	2	3	4	5	6	7	8
Time (ms)	391	391	391	401	411	431	431	451
Assignment	9	10	11	12	end			
Time (ms)	471	501	55981	116228	293833			

3000 recipes							
Assignment	1	2	3	4	5	6	7
Time (ms)	1202	1202	1202	1202	1212	1232	1242
Assignment	8	9	10	11	12	13	14
Time (ms)	1322	1543	1713	3185	3185	3185	3185
Assignment	15	16	17	18	19	20	21
Time (ms)	3185	3195	4236	8282	21001	21001	21001
Assignment	22	23	24	25	end		
Time (ms)	281445	786531	1008581	1385973	2254022		

5000 recipes							
Assignment	1	2	3	4	5	6	7
Time (ms)	1943	1953	1983	2243	2644	2804	3104
Assignment	8	9	10	11	12	13	14
Time (ms)	34990	34990	116748	242679	242679	360699	510104
Assignment	end						
Time (ms)	3128759						

Figure A-7



Only the graph for 300 recipes is shown here but the others show similar properties, as can be seen in the data tables.

The parameters used for this test were 1 meal, 20 subjects, 1 suggestion, 3 meal parameters, 7 nutritional requirements 0 forbidden ingredients and 0 forbidden categories. The algorithm used was Partial Forward Checking.

300 recipes					
Assignment	1	2	3	4	end
Time (ms)	330	671	3104	8592	19498

500 recipes								
Assignment	1	2	3	4	5	6	7	8
Time (ms)	721	941	1062	1142	1352	1402	1422	2924
Assignment	9	end						
Time (ms)	11567	54168						

1000 recipes								
Assignment	1	2	3	4	5	6	7	8
Time (ms)	4146	4216	4386	11236	21902	23895	59396	128795
Assignment	9	end						
Time (ms)	202531	511416						

3000 recipes								
Assignment	1	2	3	4	5	6	7	8
Time (ms)	18417	18857	19288	19839	21391	22412	59596	60167
Assignment	9	10	11	12	end			
Time (ms)	62290	73956	111330	1234615	4268338			

5000 recipes							
Assignment	1	2	3	4	5	6	7
Time (ms)	32116	35501	77161	97911	270759	277569	292571
Assignment	8	9	10	11	end		
Time (ms)	765070	1472437	2712400	3612825	10544582		

Decrease in plan quality using five seconds maximum search time		
	Parameter model	*Recipe model*
Number of recipes	% decrease	% decrease
50	0	0
100	0	0
300	2	2,5
500	10	2
1000	6,5	4
3000	100	23,5
5000	100	13,5

This table is based on the UB tests above. For the parameter model only one meal was planned and for the recipe model two meals were planned. The percentages are the result of comparing the UB of the latest complete assignment five seconds into the search to the UB resulting from a completed search. The two results showing a 100% decrease in plan quality are those that were unable to produce a complete assignment in five seconds time.

Appendix B – Key classes and code organisation

The project is divided into four tiers: UI, Application Logic, Data Access and Data Storage. Apart from these tiers there are also the Data Objects and Exceptions packages, which are used by all the tiers to some extent.

The UI tier has been developed in a separate project and won't be detailed here. The Data Storage tier manages all hard drive (currently) or database access. Data Access contains methods and classes for managing data. The Application Logic tier contains all the different solving algorithms, constraints and classes for managing subjects, recipes and meal plans.

The call chain is UI→Application Logic→Data Access→Data Storage. There are no method calls in the opposite direction or over more than one link (for example UI to Data Access) of the chain.

The UI tier

As mentioned above this tier has been developed in a separate project and while I have made some simple changes and fixed some bugs that have arisen during the development of the planner I do not have a clear view of this tier or the classes contained within. The `main()` method can be found in the MealPlanning class.

The Data Access tier

Data Access consists of the following classes:

- **PlansDA** contains wrappers for the methods found in PlansDS. Instantiated by the PlanManager class.
- **RecipeDA** contains wrappers for RecipeDS methods, methods for matching assignments to recipes in the parameter model and a recipe list sorting method. It also contains the methods used for generating testing recipes. Instantiated by the RecipeManager class.
- **SubjectDA** contains wrappers for SubjectDS as well as the methods used for generating testing subjects. Instantiated by the SubjectManager class.

The Data Storage tier

This tier also has a helper package denoted Parsers that contains the parser used for reading recipes in XML-format.

Data Storage consists of the following classes:

- **PlansDS** contains methods for the saving and loading of meal plans. Instantiated by the PlansDA class.

- **RecipeDS** contains methods for the saving and loading of recipes. Saving is only possible in an internal format used for testing. Internal recipe data and external XML recipes can be loaded. Instantiated by the RecipeDA class.
- **SubjectDS** contains methods for the saving and loading of subjects. Instantiated by the SubjectDA class.

The Application Logic tier

This tier has a helper package, Constraints, that contains all the different constraints available in the system.
Application Logic consists of the following classes:

- **ConstraintManager** contains methods for setting up, storing and updating constrains based on the data sent to the Solver class from the UI. Also contains the Collaborative Filtering methods. Instantiated by the Solver class.
- **DFBB** is an interface containing base variants of methods needed by the various solving algorithms. Cannot be instantiated.
- **DFBBBasic** contains the basic DFBB algorithm and its *k*-best variant. Does not use forward checking or dynamic variable ordering.
- **DFBBFC** contains the forward checking algorithm and its *k*-best variant. Does not use dynamic variable ordering.
- **DFBBFCDVO** contains the forward checking algorithm and uses dynamic variable ordering.
- **DFBBPFC** contains the partial forward checking algorithm and its *k*-best variant. Does not use dynamic variable ordering.
- **DFBBPFCRecipeModel** contains the modified k-best PFC algorithm that ensures variation between the *k* solutions.
- **Pareto** contains methods to filter a set of solutions for those that are Pareto optimal.
- **PlanManager** contains wrappers for the methods in PlansDA. Instantiated by the UI.
- **RecipeManager** contains wrappers for RecipeDA methods as well as methods for calculating the cost of a recipe and domain generators for the parameter model. Instantiated by the UI.
- **Solver** contains the various methods that can be called by the UI to generate meal plans. `generateIterativePlan()` uses the parameter model while `generateAltPlan()` uses the recipe model. Instantiated by the UI.
- **SubjectManager** contains methods for managing subjects using only their names, the idea being that the UI should not have to use the Subject class. Instantiated by the UI.

The Data Objects package

The Data Objects package consists of the following classes:

- **Assignment** represents an (partial) assignment in the DFBB algorithms. Consists of one or more SingleAssignment.
- **Ingredient** represents an ingredient from a database perspective. Contains a name and a price.
- **IngredientAvailability** represents an ingredient from an availability perspective. Contains a name, an amount, a unit and an expiration date.
- **IngredientRequirement** represents a required ingredient in a recipe. Contains a name, an amount and a unit.
- **MealDate** contains a date in "YYYY-MM-DD" format as well as a meal number to handle several meals planned for one day. This class also contains methods for comparing and incrementing MealDates.
- **MealPlan** is the represantation of a meal plan, contains one or more MealSuggestion.
- **MealSuggestion** represents a single meal, contains a recipe name and a MealDate.
- **NutritionRequirement** is used by the nutritional constrains. Contains the name of the nutritional value as well as minimum and maximum values.
- **Preference** represents a recipe rating. Contains a recipe name and a rating (1-5).
- **Recipe** represents a recipe. Contains cost, time consumption, difficulty, ingredient requirements, nutritional values and categorisation.
- **SingleAssignment** represents the assignment of a value to a variable. Contains a variable name and a value.
- **Subject** represents a person that meals can be planned for. Contains a name, recipe ratings, ingredients/categories to avoid and desired values for recipe cost, time consumption, difficulty and nutritional values.
- **Variable** represents a variable in the solving algorithms. Contains a variable name and a domain.

Using threading functionality

To use the threading functionality an UI needs to run the Solver as a thread. First call `setParameters()` that takes the same arguments as `generatePlan()`, this saves the parameters locally in the Solver class to make them available during the search. To start the search you need to create and start the Solver thread, see the code example below.

. At any point during or after the search you can call `stopAndReturn()` in the Solver to get the best plan found so far or an empty plan if no complete plans have been found. You may also find `isDone()` useful, it will return true if no search is currently running.

Note that threading currently only works for the recipe model. The parameter model uses several method calls and cannot be approached in the same way.

Example: Setting up a Solver thread

```
Solver theSolver = new Solver();
theSolver.setParameters(<parameters>);
new Thread(theSolver).start();
```

Running tests

The test class contains the various methods that were used for the tests described in this report. `doAltBenchmark()` uses the recipe model while `doIterativeBenchmark()` uses the parameter model. These two methods run through all the parameter combinations using hard coded algorithms and write the results to hard coded text files. If we want to test just a single parameter combination we use `doAltBenchmarkWithParameters()` or `doIterativeBenchmarkWithParameters()`, these methods return a string containing the resulting times.

The parameters supplied to the methods are number of subjects, number of recipes, number of meal parameters, number of nutritional requirements, number of categories, number of ingredients, number of meals and algorithm name, in that order. For example if we want to test the recipe model by planning two meals, using PFC, for four subjects with three meal parameters each, using a recipe database of 300 recipes and no additional restrictions the call would look like this:

```
doAltBenchmarkWithParameters(4, 300, 3, 0, 0, 0, 2, "dfbbpfc");
```

For the UB test a modified version of the PFC algorithm was used. When we want to run a UB measuring test we supply “dfbbpfcbench” as the algorithm name to the parameter methods mentioned above, results are written to a hard coded text file.

Note that all the test methods assume the existence of pre-generated data files for the various parameter combinations. These are generated by the two methods `generateSubjectFiles()` and `generateRecipeFiles()` found in the same class. Data files should only be generated once (not for every test) if you wish to compare results from different runs.

Appendix C – Acronyms

BD	–	Decreasing Backward Degree
CBJ	–	Conflict-directed Backjumping
CF	–	Collaborative Filtering
CSP	–	Constraint Satisfaction Problem
DFBB	–	Depth First Branch and Bound
DG	–	Decreasing Degree
DVO	–	Dynamic Variable Ordering
FC	–	Forward Checking
FD	–	Decreasing Forward Degree
LB	–	Lower Bound
MAX-CSP	–	Maximum Constraint Satisfaction Problem
MRV	–	Minimal Remaining Values
PFC	–	Partial Forward Checking
UB	–	Upper Bound
WCOP	–	Weighted Constraint Optimisation Problem