A Constraint Programming Model for Automated Generation and Optimisation of Study Plans

Tianshu Wang

A thesis submitted for the degree of Bachelor of Advanced Computing (Honours) at The Department of Computer Science The Australian National University

May 2019
Except where otherwise indicated, this thesis is my own original work.

Tianshu Wang
31 May 2019
To my family and friends, who have accompanied me for such a long, bittersweet time.
I take this opportunity to express my appreciation to my supervisor, Sylvie Thiébaux, who kept motivating me and providing me with professional academic advice along the whole process. My thanks also go to Lincoln Smith, an expert in the Data Mining field, who worked in College of Engineering and Computer Science at The Australian National University. With permission from my supervisor, he offered me a well-designed data mining Python program to help me overcome the first obstacle in the project. Last but not least, I am eternally grateful to my parents, who provided me with financial support for my academic pursue and continuous encouragement. Without help and support from those people, I could not have finished this work.
Abstract

Scheduling an effective study plan is one of the time-consuming activities for the students at the university. Considering the varied degree requirements, study plans for each program need to comply with these regulations in order to fulfill the degree completion requirement. However, students are intended to take their individual interests into consideration of their corresponding study plans. An efficient mechanism is necessary in designing an effective and robust study plan scheduler. This research aims to propose a solution for the scheduling problem using Constraint Programming (CP) technique, based on data available from the University Website and from the users of the system. The outcome of this research is an interactive website that would construct an appropriate study plan based on user preferences. A number of evaluations have been done to investigate the further potential of the applied mechanism in the regarding problem.
Contents

Acknowledgments vii

Abstract ix

1 Introduction 1
  1.1 Motivation 1
  1.2 Objectives 1
  1.3 Contribution 2
  1.4 Outline 2
  1.5 Summary 3

2 Background and Related Work 5
  2.1 Background 5
    2.1.1 Constraint Satisfaction Problem 5
    2.1.2 MiniZinc – An Adaptive Constraint Programming Language 5
  2.2 Related Work 7
  2.3 Summary 7

3 Approach 9
  3.1 Data Mining 10
  3.2 Pre-processing 11
  3.3 Model Architecture 12
    3.3.1 Vocabulary 12
      3.3.1.1 Decision Variables 12
      3.3.1.2 Input 13
    3.3.2 Predicates and Functions 15
      3.3.2.1 Constraints 17
    3.3.3 Objective Function and Relaxation 19
  3.4 Website Construction 20
    3.4.1 Graphical Interface 20
    3.4.2 Data Interaction 20
  3.5 Summary 22

4 Evaluation 23
  4.1 Evaluation on Solvers 23
  4.2 Evaluation on Execution Efficiency 23
  4.3 Evaluation on Planning Feasibility and Flexibility 25
List of Figures

2.1 Constraint Satisfaction Problems . . . . . . . . . . . . . . . . . . . . . . . . 6
3.1 Overview of Working Process . . . . . . . . . . . . . . . . . . . . . . . . . 9
3.2 Degree Program Requirement Structure . . . . . . . . . . . . . . . . . . . 10
3.3 An Example of Degree Requirement . . . . . . . . . . . . . . . . . . . . . . 17
3.4 Graphical User Interface - Part 1 . . . . . . . . . . . . . . . . . . . . . . . 21
3.5 Graphical User Interface - Part 2 . . . . . . . . . . . . . . . . . . . . . . . 22
4.1 Selected Branches in MCOMP Requirements . . . . . . . . . . . . . . . . . 24
4.2 (a) Running Time in Different Cases; (b) The Number of Total Con-
    straints in Different Cases . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
List of Tables

4.1 Performance of Solvers on MCOMP Degree ........................................ 23
4.2 Feasibility of each Specialisation in Different Semesters ......................... 25
4.3 Reasoning for AI Specialisation Unsatisfiability .................................. 26
4.4 Reasoning for DS Specialisation Unsatisfiability .................................. 27
Chapter 1

Introduction

Constraint programming (CP) has contributed significantly to Artificial Intelligence (AI) planning problems, despite many other applications in the fields of graph theory, algorithm, and operational research [Rossi et al.]. The extensive application of CP is due to the fact that it helps formulate a problem as a set of constraints elegantly and it is capable of solving multi-dimensional problems efficiently. This report, however, considers how CP could be used to solve the problem of automatically generating and optimising study plans.

1.1 Motivation

The diversity of programs offered by ANU requires students to understand information about program requirements, courses offerings, and relations between courses, such as "A is a prerequisite, co-requisite or incompatible course of B." However, information placed in the Programs and Courses Website might not be the best comprehensible for students, since they need to switch from page to page to determine whether a study plan could be eligible from the perspective of requirements, timing, and individual interest. In this case, arranging their study plan manually becomes a non-trivial combinatorial work. An automated study planner would help students choose and sequence their courses, and avoid mistakenly choosing courses that do not contribute to satisfying their degree’s requirement. Also for academics or professional staff working in ANU, when they want to create a new degree program, major, or specialisation, automated study plan generation would lower the risk of offering unfeasible degree programs, by helping them to test the validity of their newly designed programs and understand the causes of the infeasibility.

1.2 Objectives

The question we focus on is: how can students choose a study plan (courses, schedule), that best fits their interests and complies with the requirements of their degree program? This study plan generation problem can be elegantly and efficiently formalised as a Constraint Optimisation Problem, and solved with CP solvers.
Firstly, we treat both programs and courses as entities and construct classes with features for courses (e.g., credit units, offerings) and their stated relations (e.g., course A is a prerequisite course for course B, course A is in one of required courses for program C) by scraping unstructured data from P&C website.

Secondly, we build a MiniZinc model to solve our constraint satisfaction problem (CSP), and as a result to generate an eligible plan which at the same time best fits users’ interests or preferences (including hard and soft preferences). Users have the possibility to request a modification to the proposed study plan, in particular by requiring that some of the proposed courses be replaced.

Thirdly, we facilitate the interaction of the user with the planner, by capturing the degree structure and the user preferences, as well as displaying the generated plans via a web interface.

Fourthly, we conduct a number of experiments to observe the quality and efficiency of our planner system with the size of the problem increasing. We also demonstrate the usefulness of our study planner on several scenarios.

1.3 Contribution

• **Data Scraping and Structuring**
  Lincoln Smith kindly offered a data scraper specific for the P&C website. The obtained data includes program requirements for Master of Computing and that for Master of Computing (Advanced).

  We created a structure representing relations between courses, courses offerings, and we sorted, structured and preprocessed those data accessed by our data scraper in a way readable for MiniZinc.

• **AI modeling**
  We developed a CP model in the MiniZinc platform for our course planner system. The resulting model could generate and re-generate study plans based on acquired information and consideration of the users’ preference and eligibility.

• **Graphical User Interface Construction**
  We constructed a website tool to make our work easily usable. Also, we evaluated the efficiency and the ability to fit the plan with the preference of users with some experiments.

1.4 Outline

• Chapter 2 presents background and related work on constraint programming.

• Chapter 3 explains approaches I applied or implemented.

• Chapter 4 evaluates the performance.

• Chapter 5 concludes my report and discuss the future work.
1.5 Summary

In this chapter, we explained our motivation of doing this work in some degree of details in Section 1.1, stated our objectives in Section 1.2, listed our contribution in Section 1.3, and in the end, outlined our report structure in Section 1.4.
Chapter 2

Background and Related Work

In this chapter, we briefly recapitulate background knowledge about constraint programming - the field our problem falls into, and MiniZinc - the platform we depended on to build our model. Secondly, we investigate a range of prior related work.

2.1 Background

2.1.1 Constraint Satisfaction Problem

"In which we see how treating states as more than just little black boxes leads to the invention of a range of powerful new search methods and a deeper understanding of problem structure and complexity."


A constraint satisfaction problem (CSP) refers to a problem that can be formulated as a triple - a collection of variables, a collection of domains and a collection of constraints, as Figure 2.1 shows [Russell and Norvig 2016]. A CSP offers a way to solve a problem by describing the problem explicitly first and then using generic ways to search for solutions efficiently.

A range of real-world problems can be formulated as CSPs and solved by computers - many classic logical problems, such as Eight Queens Puzzles, Map Coloring, Sudoku which falls in the realm of Artificial Intelligence, and scheduling problems, for instance, university timetabling scheduling, and the Job Shop Scheduling Problem, which fall under the Operational Research field.

2.1.2 MiniZinc – An Adaptive Constraint Programming Language

CP modelling is a technique for solving CSP. In our work, we implemented our model in MiniZinc, a standard CP language [Nethercote et al. 2007]. The difference between MiniZinc and other CP languages is that it incorporates different general-purpose solvers and standardizes them by reducing models into a commonly com-
Constraint Satisfaction Problem

Figure 2.1: Constraint Satisfaction Problems

patible lower-level representation \[\text{Nethercote et al., 2007}\]. Thus it extends the adaptability and helps constraint writers target a solver that best fit the regarding problem better.

Technically, it takes two steps to solve constraints by MiniZinc:

1. A constraint writer inputs constraints in MiniZinc and clarifies a target solver, for instance, Gecode \[\text{Schulte et al., 2010}\], OSICBC \[\text{Forrest et al., 2018}\], Choco \[\text{Jussien et al., 2008}\] or a finite domain (FD) solver \[\text{Carlsson et al., 1997}\].

2. MiniZinc parses solver-specific constraints to FlatZinc, where FlatZinc was implemented to convert different solver-based models into a low-level expression \[\text{Nethercote et al., 2007}\].

Generally, the task for MiniZinc is assigning values for all defined variables in a constraint network \[\text{Bessiere, 2006}\], and either seeking for a number of satisfiable solutions or optimising an objective function, where the value of a formula comprised of variables is stated to be maximised or minimised.

In this searching process, solvers iteratively communicate domain reduction of decision variables to every constraint that involves the corresponding variable, and this is known as Propagation \[\text{Vilain and Kautz, 1986}\]. There are many heuristics that can be applied to the propagation process, such as Arc Consistency \[\text{Bessiere, 1994}\], Forward Checking \[\text{Bessiere et al., 1999}\], and some hybrid heuristics like Backmarking \[\text{Bacchus and Grove, 1995}\] which combines Forward Checking and Backtracking \[\text{Ginsberg, 1993}\].
2.2 Related Work

Prior to our work, which falls into the scheduling problem class, a significant number of researchers have already stepped into this field and facilitated its development by proposing various mechanisms.

Many methods apply local search over the space of assignments [Lourenço et al., 2003]. One is the hill-climbing approach [Xi et al., 2004], in which we keep looking for a better solution from the neighbours of the current value for each variable until a fixed point. Although this approach is guaranteed to find a better solution compared to previous one iteratively, it is likely to be trapped in local maxima. Another one that has the potential of jumping out from the neighbouring of the initial node is Simulated Annealing [Van Laarhoven and Aarts, 1987], which tolerates some degree of worse or equal solutions compared with the current one, but decreasing the probability of accepting them throughout the search process.

There are also some non-local search approaches which seek a better solution in the solution space. For instance, Tabu search [Burke et al., 2007]. It is similar to local search but does not move to a best neighbouring node if the latter does not exist in the Tabu list, or it exists in it with aspiration conditions satisfied. This approach shares the same search space (solution space) with that of another method - Large Neighborhood Search (LNS) [Ropke and Pisinger, 2006]. In LNS, except available solutions, it involves randomisation into finding the optimal solution, which converts the optimising process into an assignment by destroying partial assignment in a solution and repairing it and resulting in the possibility of finding a prioritised one. There are other approaches frequently involving randomisation into the searching process to avoid local optima, such as Genetic approach [Abramson and Abela, 1991], a bio-inspired approach aroused by The Evolutionary Theory by Darwin, which represents a solution as a chromosome and it select chromosomes based on their fitness value (resembles the property in the objective function), does crossover, mutation iteratively until hitting a critical point.

Even though we have not found the same work done as ours, our work resembles the university timetabling generation problem with a certain degree of juxtaposition. After investigating this similar problem, we found that CP modelling is the mainstream solution and that the above techniques for scheduling, such as Tabu search [Burke et al., 2007], Simulated Annealing [Van Laarhoven and Aarts, 1987], and Clustering [Shatnawi et al., 2010], could also easily be applied to it.

2.3 Summary

In this chapter, we introduced the background for CSP and MiniZinc in Section 2.1 then reviewed the literature pertaining to our problem in Section 2.2.
Chapter 3

Approach

Figure 3.1: Overview of Working Process

The architecture of our system is shown in Figure 3.1. Given a set of degree requirements, the first process is gathering user information. On this basis, our
system does mining for necessary information about courses and degree programs, including locating and extracting the data needed. With obtained data, the next step is to structure and export it to our designed MiniZinc model. Thirdly, the applied MiniZinc solver finds the optimal solution for our model. Besides, if the user is not satisfied with some courses in our plan, they refine the current schedule until being satisfied. Afterwards, we built a web-based user interface to make our system more accessible to users. Difficulties include building a general model structure for different programs and automatically generating these models. In this chapter, we discuss these methods and their implementations in detail.

3.1 Data Mining

Figure 3.2: Degree Program Requirement Structure

As Figure 3.2 indicates, a requirement tree is composed of

- **Root node**
  A logical **AND** for including all following children nodes.

- **Internal nodes**
  1. A logical connective **AND** or **OR**, which represents the relation between its children;
  2. **Course-level requirements** - defines the minimum number of courses from a specific level that should be taken;
3. **Minimum sum of units**, which stands for the total number of units for several combined requirement lists. This item is necessary occasionally. To illustrate, in Figure 3.2, the requirement for courses list F does not set a lower bound for students, and therefore the overall credit goal is set to constrain that there must be enough courses also be taken in F.

- **Leaf nodes**
  Courses lists with credit requirements (the maximum or the minimum of credit units should be taken in this list) for them respectively.

To obtain and make use of data from a degree requirement, our scraper should be able to navigate programs and courses web pages to find related objects (such as courses, specialisations) and inspect relations between them. In the first step, we extracted the target data that matches Regex expressions. Also, we applied a tool package embedded in Python - BeautifulSoup4 \[Richardson, 2015\] to crawl those matched data on websites efficiently.

We designed a scraper that does an iterative search and represents these nested data as a tree. By doing that, we could extensively apply this scraper to more than one programs for those consist of the same general elements, regardless of their length or structures.

We stored data in our objects - Program Order and Courses after received them from HTML. **Program Orders** are laid out in a tree. For the **Courses** object, we store some necessary information about a course and use it when delivering data to MiniZinc.

Information about involved courses and specialisations is only revealed in the respective course pages, and we need to switch from pages to pages to obtain that information. In this case, we solved page switching issues by making use of a typical pattern of the ending cue on those pages for useful information - contents are all arranged between <h2> tags. As a result, our scraper keeps extracting data until it reaches the end of the <div> or the next <h2>.

P&C website contains degree programs designed by different people, some detailed designs, for example, margin length and space length between lines are sometimes different. Therefore, we put efforts to fix this compatibility issue to create a general scraper that could be applied in diverse degree programs (possibly for the future). We compared and used the maximum common divisor of each measurement in the HTML code for iterations. Our scraper then tries searching for data by stepping forward with the minimum value for those measurements first, and increasing a standardized step size each time until it finds its target.

### 3.2 Pre-processing

The task after collecting all data was to convert those data contained in lists, dictionaries, and arrays to a MiniZinc encoding. MiniZinc accepts two kinds of files - model files and data files. Considering that different programs have different re-
quirement structures, creating a generally applicable model file (constraints) complementing specific data files for all degree programs is not feasible. The system should be able to generate specific data file, and model files with specific requirement lists according to the data received about degree requirement, and general constraints for rules like timing, course offerings, and the number of taken courses for each semester.

The scraper sends an object containing all needed information to the function building MiniZinc files, as what our scraper collected is a nested, tree-structure data, including lists and lists of lists, which could not be easily split up.

To sort out data after obtaining it from the scraper, we unzip containers in the lists, label each course set consecutively, and then record data. Parentheses interpret the nested structure. In this case, our system could untangle the order of the program requirement by iteratively matching left and right parentheses through matching the newest right bracket to the last left one, also, some qualifications about completion of compulsory number of high-level courses (such as 8000-level courses for graduates and 4000-level for undergraduates) can be mapped to those corresponding lists.

We managed to include a sufficient and necessary set of information about courses. As we discussed earlier, data files in MiniZinc should consist of every possibly involved course and its related data, though irrelevant courses are burdens to our MiniZinc model. In our problem domain, which focuses on Master of Computing degree, we could distinguish courses into courses for graduates, and related courses designed for undergraduates, which are only added as incompatible courses. The question is, how can we make sure that we include all associated courses? If we only search for related courses once, it is possible that newly added courses have some associated courses that are not incorporated into the system. Therefore, we stop when we reach a fixpoint where there is no new course being added. We confirmed that the resulting array of related courses is the minimum necessary set of courses.

### 3.3 Model Architecture

There are plenty of CP platforms and great tools for designing solvers for specific problems. The reason that we chose MiniZinc is that, firstly, it is relatively small and therefore could be applied efficiently to our small-scale problem; secondly, it offers an integrated development environment with different general solvers and a standard interface for programmers. Constructing a model in MiniZinc enables us in principle to compare and select between a wide range of solvers, based on their searching efficiency on our problem.

#### 3.3.1 Vocabulary

##### 3.3.1.1 Decision Variables

A set of decision variables denote the solution to our problem. They are stated at the beginning of our model, and the model solves for each of them in the end.
Takes. An essential variable array in our problem domain is takes, which determines whether a course is chosen, and if chosen, it denotes the semester the course arranged in; if not, it is set to 0.

Units. Credits contained in a course could be a fixed value or a variable in a range depending on how it has been designed. Therefore, in our model, we have an array units represents the final decision of how many units the course contains for each course in the current study plan, for example, students can decide to take a project course COMP4560 with either 12 units or 24 units. In our model, unit values are determined only by the constraint model instead of users.

3.3.1.2 Input

In this section, we introduce all constants and domains in our data model, whose value is either straightforwardly dependent on user inputs or is indicated by the degree program that the user enrolls in.

Courses. Each course pertaining to the degree program that being processed is represented by its course code as an element belonging to Courses. Many arrays are consist of courses or set of them, such as Course Lists, Prerequisite, Corequisite, Incompatible as well as Graduate_courses and Undergrad_courses which are also introduced in this section.

Semesters. Every year and every semester, students enrol into ANU, and the university offers various courses. In our problem domain, every enrolled semester and course-offering semester falls into one of four categories, regardless its exact year and date - the first semester in odd years (denoted by odd_first), the second semester in odd years (odd_second), the first semester in even years (even_first) and the second semester in even years (even_second). Semester information impacts significantly on whether it is possible to generate a feasible plan. This type of elements constitutes Offered semesters. Based on this, the type of enrolled semester is recorded as a constant - Enrolled semester, and an array offered semesters depicts in which semester(s) each involved course is offered.

Graduate courses and Undergrad courses. We separated courses into courses for graduates and those for undergraduates, in which graduate courses are the focus, and undergraduate courses are brought up as (mostly) incompatible courses of some graduate courses. Thus they are not given more detailed information.

Course lists. Courses are arranged in corresponding lists according to the contents specified in requirement pages.

Available Units. As mentioned in courses might have more than one distribution of contained credit units. This array, available units is composed of sets of available units for all courses that are likely to be taken.

Prerequisite, corequisite, and incompatible courses. Courses might relate with other courses as requisite courses and incompatible courses. Assuming course $C_i$, its requisite courses are specified as prerequisite courses (described in logic in 3.3.2) - courses meant to be taken at least one semester earlier than $C_i$, and corequisite courses (3.3.2) - similarly mean those should at least be taken at the same time with
In our model, three constraints about the requisite and incompatibility rule were constructed with help from our designed predicates (3.3.2) for each involved course respectively.

Unfortunately, those data are not available for our scraper to extract, due to the difficulty of parsing natural language from P&C website. Therefore, we collected those data by hands and stored them as arrays.

Requisite courses are one or several sets of equivalent courses where each set contains at least one course that is necessary to be taken prior to the intended course. Therefore, requisite courses are represented by two-dimensional arrays. Only incompatible courses are one-dimensional array, as each course in there is indifferently not compatible with its respective course.

For each course, the format (size) in those arrays is fixed in order to align in MiniZinc and to be recognized. We designed the format based on the maximum size in both dimensions of those arrays, where array prerequisite needs a size of $3 \times 3$, corequisite is in $1 \times 1$ and incompatible is $1 \times 3$. There exist courses whose size of arrays is smaller than the fixed size; in this case, extra space in those arrays are padded with None.

To illustrate, in our system, the requisite and incompatibility of the course COMP6310 are represented as below:

**Prerequisite** -

```
[ [COMP6300, None, None],
  [COMP6700, COMP6710, None],
  [COMP6331, None, None] ].
```

It means that to take COMP6310, one must have taken COMP6300, COMP6700 or COMP6710, and COMP6331.

**Incompatible** -

```
[COMP2310, None, None]
```

This indicates that if one has taken COMP2310 before, then he/she would not be able to take this.

**Preference.** Preference is a user-defined integer variable for each course. Its value varies from 0 to 5 in the integer range. Notably, a course $C_i$ has preference($C_i$) = 0 means that not taking $C_i$ is a hard constraint; on the other hand, if preference($C_i$) = 5 then taking $C_i$ becomes a hard constraint; every value between 1 to 4 shows a preference in an ascending order for how much the student is willing to take this course. The preference value is obtained through the front end (3.4), and this is the object to be maximized in the objective function (Section 3.3.3).
3.3.2 Predicates and Functions

In this section, we introduce the main framework of our model, which are composed of predicates and functions. They both play an essential role in CP modelling, especially in models containing complex constraints. Defining predicates and functions helps to write simplified and concise constraints. Afterwards, when the MiniZinc solver works towards solving constraint, those predicates and functions would be unravelled and substituted by the original arguments. It gives us programmers the capacity to provide building blocks for a particular use. Equally importantly, the readability of the MiniZinc program is enhanced.

**Predicate** requirement_node

**Param:**
- array[int] of var int: takes
- array[int] of courses: requirement
- int: signal
- int: req_unit

\[
\begin{align*}
& \text{(if signal then} \\
& \quad \text{%% when the required number of credits is a lower bound} \\
& \quad \text{sum (c in requirement where takes[c] > 0) } \ast \text{ (credit[c]) } \geq \text{ req_unit ;} \\
& \text{else if signal == -1 then} \\
& \quad \text{%% when the required number of credits is an upper bound} \\
& \quad \text{sum (c in requirement where takes[c] > 0) } \ast \text{ (credit[c]) } \leq \text{ req_unit ;} \\
& \text{else} \\
& \quad \text{%% when the required number of credits is a fixed value} \\
& \quad \text{sum (c in requirement where takes[c] > 0) } \ast \text{ (credit[c]) } = \text{ req_unit ;} \\
& \text{end} \\
\end{align*}
\]

The predicate **Requirement Node** (Figure 3.3.2) reduces the complexity as well as the length of encoding a nested requirement lists. It is used to represent each requirement node in its regarding degree program. A variable *signal* denotes which type of requirement each one is, either represent an upper bound, a lower bound, or a fixed bound and defines the corresponding constraint for each.

**Predicate** level_criteria

**Param:**
- array[int] of var int: takes
- set of courses: list1
- array[int] of courses: level8
- int: req_unit

\[
\text{(sum (c in (array2set(level8) intersect list1) where takes[c] != 0) } \ast \text{ (time_unit[c]) } \geq \text{ req_unit);} 
\]
In order to bound the course level for specific requirement list, our model needs to know what the bound is, and where it applies. To achieve this, we wrote a predicate `level_criteria` (3.3.2). Internally, the core relationship is that the cardinality of targeted high-level courses and those in the course list that are chosen to take should be no less than the threshold number.

**Predicate** prerequisite_criteria
*Param:*
- courses: c1
- courses: c2
- array[int] of var int: takes

\[(\text{takes}[c1] < \text{takes}[c2] \text{ AND } \text{takes}[c1] > 0);\]

**Predicate** corequisite_criteria
*Param:*
- courses: c1
- courses: c2
- array[int] of var int: takes

\[(\text{takes}[c1] \leq \text{takes}[c2] \text{ AND } \text{takes}[c1] > 0);\]

The below two predicates (3.3.2 and 3.3.2) indicate the rule of prerequisite and corequisite courses, that is, if taking a course, its prerequisite courses need to be arranged before it, and its corequisite courses need to be taken before this course or at the same semester as the course.

**Function** Unit_Sum
*Param:*
- set of courses: grad_courses
- array[courses] of var semester: takes
- array[1..n] of courses: requirement

\[\forall (c \in \text{grad_courses})
\quad (\sum (c \in \text{requirement where takes}[c] \neq 0) \text{ (time_unit}[c]));\]

The function `unit_sum` (3.3.2) computes the total number of credits for several lists of courses. This function is useful for checking that a minimum number of units requirement pertaining to several lists of courses is satisfied. For instance, in Figure 3.2 it would be applied to compute the total number of courses taken from list E, F, so as to be able to check that this total is at least 36.
3.3.2.1 Constraints

The main body of our model consists of constraints, which describe the final solution to our problem. In this section, we divide all constraints into three types:

1. general constraints - which describe ground rules in our problem;
2. a degree constraint - which indicates a flattened requirement tree;
3. user constraints - which specifies the individual preference for study plans.

Firstly, we introduce the general constraints.

The constraint **Credit Unit** (3.3.2.1) is about multiple units choice. Owing to variable options for how many credits a course could contain, such as 6 units, 12 units or 24 units, our planner should be able to schedule units for courses to get a solid plan that best fits.

**Constraint** Credit Unit

**Param::**
array[courses] of var semester: takes
array[grad_courses] of set of int: time_unit_available
array[grad_courses] of var units: time_unit;

forall (c in grad_courses)
(time_unit[c] in time_unit_available[c]);
The timing constraint (3.3.2.1) - based on information about offering semesters for each course, as well as the starting semester of each student, we could decide when to take a course.

**Constraint Timing**

*Param:*:

- set of courses: grad_courses
- array[courses] of var semester: takes
- array[grad_courses] of set of semesters: offered_semester

forall(c in grad_courses)
((takes[c] != 0) =>
(to_enum(semesters, (((start_semester - 1 + takes[c]) - 1) mod no_of_semesters) + 1) in offered_semester[c]));

On top of those listed above, we defined constraints:

- Undergraduate courses should not taken;
- For each semester, the overall credits should be 24 units (equivalent to 4 standard courses);
- For each course taken, requisite and incompatibility policy should be applied (using prerequisite, corequisite, and incompatible predicates);
- For each course, their assigned credit should be in its stated credit range;

Secondly, we designed the degree constraint as a combination of requirement node (Section 3.3.2) unit sum and level criteria connected by logic “AND”, “OR” and separated by brackets. As a whole they give the full information of any degree requirements.

Thirdly, regarding user preference defined in Section 3.3.1 we enforce that taking courses with the least preference 0, and not taking those with the maximum value 5, which is explicitly written in MiniZinc as below.

**Constraint Hard Preference**

*Param:*:

- set of courses: grad_courses
- array[courses] of var semester: takes
- array[courses] of var int: preference

forall(c in grad_courses)
((preference[c] == 5 -> takes[c] != 0) AND (preference[c] == 0 -> takes[c] == 0));
3.3.3 Objective Function and Relaxation

Study plans generated by our planner are supposed to fit the user’s preference as much as possible (based on its eligibility). Therefore, we have an objective function which maximizes the \textbf{Preference} value for all chosen courses.

There are two phases where their objective functions are slightly different:

1. Initial Planning Phase

   In this phase, the user has already assigned courses with \texttt{preference} value, and a first plan must be generated that optimizes these preferences given the constraints.

   \[
   \text{solve maximize } \sum (c \text{ in grad_courses where takes}[c] \neq 0) \times (\text{preference}[c]);
   \]

2. Refinement Phase

   This phase occurs when the user, after seeing the generated plan, decides that he does not like one of the courses proposed, and requests its replacement. There are two possibilities:

   (a) The course is replaceable, and an alternative plan can be generated that replaces only this course and still matches the constraints. Therefore, our planner replaces the course by another one.

   (b) The replacement of the course causes some hard constraints to no longer maintained, which occurs in two scenarios:

      i. The course is a mandatory course;

      ii. The replacement of the course is incompatible with some courses in the current plan, as it could be a required course for a latter course.

   Our system is not able to generate a plan for the first case, so it notifies the user that there are no available plans. However, in the latter case, our system relaxes the constraint that all other courses in the plan are all supposed to still be taken if possible. Meanwhile our system attempts to include another course to replace the undesired ones, and finds a trade-off in between: it minimizes the penalty - the distance between those to-be-replaced courses with 0 which stands for not being taken, and the distance between arrangement of any other courses in the new plan and that in the old plan, which keeps other preferred courses in the same sequence. The below MiniZinc code also explains this.

   \[
   \text{solve minimize }
   \begin{align*}
   &\%\% \text{ old_plan}[c] = -1 \rightarrow c \text{ is selected to be removed from the old plan.} \\
   &\sum (c \text{ in grad_courses where old_plan}[c] > 0) \times (\text{abs(old_plan}[c]-\text{takes}[c])) \\
   &+ \sum (c \text{ in grad_courses where old_plan}[c] == -1) \times \text{abs(takes}[c]);
   \end{align*}
   \]
3.4 Website Construction

Our purpose for presenting our system as a web-based tool is simple - it is cumbersome to operate our system by switching between MiniZinc and Python, for both us and users. By developing a website with a graphical interface, we can expect our system to be self-explanatory for users. Also, we could more efficiently operate and evaluate its functionality.

3.4.1 Graphical Interface

We built the graphical interface based on some mature libraries in JavaScript \[\text{Flanagan, 2006}\]. In our design, the degree requirement tree is represented as a tree shape intuitively. We applied d3 \[\text{Zhu, 2013}\], which helped us quickly realise this visualisation with adequate space to implement our desired functions. In order to express the degree of preference for courses, users can click on each node, and colour of the corresponding node change to show their degree of preference. Also, BootStrap \[\text{Efron and Tibshirani, 1994}\] enabled us to generate a table to represent our resulting study plan and to operate on it to select to-be-replaced courses.

3.4.2 Data Interaction

In our system, interaction plays an essential role, as each step is driven by more data from either the user or our back end. Also, due to that our system is dependent on crawling data from P&C website in real-time, which causes a delay in responding users, the interaction should be relatively efficient to make our system tolerable.

The interaction between our system and users includes these below:

- users click on nodes representing courses and specialisations to express their preference
- our system generates plans for users
- users refine it by clicking and excluding undesired courses in a plan
- users finish refining on the generated plan and deliver the data

All these steps require data flow from the front end to the back end or vice versa. However, rendering the whole page for trivial editing is costly. JavaScript solves this problem by Asynchronous JavaScript and XML (Ajax) \[\text{Woychowsky and Woychowsky, 2007}\] to realize efficient data transmission. Whenever users set preference value for courses, we save those data in our JavaScript variable, and when they finish, Ajax is called and transfer the variable in a JSON object \[\text{Crockford, 2006}\], and our server is able to receive the variable in the back end.

On the other hand, after our system finishes generating an optimal plan, the Ajax also help to return the resulting data from the back end to the front end by putting results in the Success function.
Figure 3.4: User Interface of Our System. Refer to Figure 3.1 for user guidance.
To realize communication between data from the back end and that from the front end, we need to build a web server. As our work was mainly centred with Python - we crawled data from websites by Python and transferred data from Python to MiniZinc, we decided to choose a Python-based framework for our web server, and finally selected Flask over Django. These two are both popular web frameworks, though the reason choosing Flask is that, Flask could offer us a micro-framework which helps us make a connection between our front page and our back end without loading extra unnecessary modules, which are mostly provided by Django though.

3.5 Summary

In this chapter, we discussed on our approach, from data crawling, pre-processing, the core implementation, to the user interface of our system. We introduced our data mining technique in Section 3.1. After that in Section 3.2, we briefly covered our data reformulation from raw data to components in our MiniZinc model. Thirdly, we presented the essentials in our constraint model in Section 3.3. Last but not least, we introduced our graphical user interface about our design of the front end and its interaction with the back end in Section 3.4.
Our work is not only a tool helping students generate study plans but also a mechanism enabling us to evaluate degree quality, in terms of feasibility and flexibility of possible study plans. In this chapter, we aim to assess our work by analyzing possible solvers, running efficiency with varying size of problem domain based on several (faked) degree requirements and planning feasibility for the Master of Computing (MCOMP) degree.

4.1 Evaluation on Solvers

Solvers perform differently depending on the problem domain, in terms of execution speed. In this part, we still use the model built for Master of Computing degree as an example, to observe the capability of solvers. The result is in Table 4.1.

For our problem, the only usable solver on current MiniZinc is OSICBC [Forrest et al., 2018]. However, FD solver [Codognet and Diaz, 1996] was also applicable to our problem before it was removed from those available, so unluckily, we did not have a chance to obtain information about it.

<table>
<thead>
<tr>
<th>Solvers</th>
<th>Status</th>
<th>Running Time(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gecode 6.1.0</td>
<td>Timeout</td>
<td>&gt; 180000</td>
</tr>
<tr>
<td>OSICBC 2.9/1.16</td>
<td>Runnable</td>
<td>4288</td>
</tr>
<tr>
<td>Chuffed 0.10.3</td>
<td>Incompatible</td>
<td>-</td>
</tr>
<tr>
<td>Gecode Gist 6.1.0</td>
<td>Incompatible</td>
<td>-</td>
</tr>
<tr>
<td>findMUS 0.1.1</td>
<td>Incompatible</td>
<td>-</td>
</tr>
<tr>
<td>Globalizer 0.1.6.0</td>
<td>Incompatible</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.1: Performance of Solvers on MCOMP Degree
the same problem as ours; secondly, even if there exist similar works done by other researchers, we cannot compare them in the same level, as assumptions and models vary over different universities and even in different departments in the same university, such as "how many semesters does a student regularly take to graduate from this degree" and "how many courses should be taken in each one semester". This is not a single case occurred in our work but a fact faced by research on constraint problems [Burke et al., 1997] [Doulaty et al., 2013].

In this section, we focused on evaluating how the size of the requirement node would impact the running time.

Regarding program requirements of MCOMP, we pruned some of the branches out, formed some smaller new degrees, and observed with the reduction of domain complexity, and how the time efficiency would change.

Figure 4.1: Selected Branches in MCOMP Requirements

Figure 4.1 shows branches in MCOMP requirements, which are a pair of ‘either’ and ‘or’ in the original requirements. With a fixed total number of decision variables, we compared nine cases below where:

1. With 'Either' and 'Or'
2. Without 'Either'
3. Without 'Or'
4. Without the first specialisation in 'Or'
5. Without the second specialisation in 'Or'
6. Without the third specialisation in 'Or'
7. Without the first and the second specialisation in 'Or'
8. Without the first and the third specialisation in 'Or'
9. Without the second and the third specialisation in 'Or'

As suggested by Figure 4.2, although the amount of time required varies when the number of branches and constraints change, the time complexity is constrained by other factors, such as how much each constraint narrow down the possible domain of our model.
§4.3 Evaluation on Planning Feasibility and Flexibility

Due to a bug in MiniZinc, which consistently leads to an infinite duplicated set of results, we were unable to detect the exact number of plans for different course sets, which could explicitly show the flexibility of different specialisations. Instead, we reported whether it is possible to generate at least one eligible plan under different specialisations, which represents planning feasibility.

From the perspective of program designers, whether future students can graduate from a proposed new degree program, regardless of their starting point in ANU, would be a vital issue to consider. Therefore, it adds value to test how different combinations of courses started in different semesters would impact the existence of eligible plans.

As Table 4.2 shows, learning Artificial Intelligence as a specialisation is not feasible for students starting from the first and the second semester in odd years. The reason, as observed in Table 4.3, is that COMP8620 is not a possible option, and for this specialisation, it must be taken. Then we checked the requisite and incompatibility of COMP8620, and we found out the principal reason as follows.

<table>
<thead>
<tr>
<th>Specilisation</th>
<th>S1 in Odd Year</th>
<th>S2 in Odd Year</th>
<th>S1 in Even Year</th>
<th>S2 in Even Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artificial Intelligence</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Data Science</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Human Centred Design and Software Development</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

Table 4.2: Feasibility of each Specilisation in Different Semesters
There are two sub-requirements for Data Science (DS) specialisation. For students in this specialisation, three courses are in the first one, and all of them are required to be taken, and one of six courses in the second branch is supposed to be learned. We found that the second sub-requirement can always be satisfied regardless of semesters, although the first one can never be fully satisfied. As a result, the DS specialisation is never feasible. Regarding this case, we focused on the feasibility of the first requirement list on each semester in this evaluation, and the result was presented in Table 4.4. Based on the outcome, we analysed the reason why DS specialisation is not applicable in each semester as follows.
§4.4 Summary

In this chapter, we focused on evaluating our work. Firstly, we evaluated on available solver about how each solver in MiniZinc performs on our problem in Section 4.1. Secondly, we analysed factors that potentially impact running efficiency in Section 4.2, from the perspective of developers; Thirdly, we investigated on the reasons why specialisations could be incompatible with degree requirements in Section 4.3.
Conclusion

In this report, we attempt to present our work in a big picture with sufficient details. Firstly, we propose the necessity of adopting a systematic mechanism for study plan generation and optimisation problem for ANU students. Following this, we introduce our system and its principles on how it tackles the issue of data unavailability by data mining, pre-processing and model construction. Instead of exhausting variable domain to find an eligible study plan as traditional way suggests, our heuristic approach takes constraints for our problem into account and excludes impossible solutions as early as possible. Meanwhile, our system enables students to construct preference-centred study plans, as well as refining study plan iteratively. Besides, we build our system as a web-based tool which helps it to be more accessible. In the end, we evaluate our work on different factors, from how it performs internally to how it could be otherwise applied externally.

5.1 Future Work

In the future, we expect to extend our system from one or two Master degrees, which are two-year programs, to more large-scale degrees, for example, Bachelor of Advanced Computing (Honour) - an undergraduate degree (4 years). To achieve this, we might need to overcome many challenges that occur when problem size goes broader. On the other hand, we plan to design a reasoning system which could help course designers to detect the feasibility of under-designed degrees more intelligently.
Appendix A: Project Description
1 Project Title
ANU Study Planner

2 Project Description
The project focuses on solving a common question that ANU students have: how can I choose a study plan (courses, schedule), which fit my interests and the requirement of my degree program? The project will produce a web-based tool that enables them to construct such study plans in a range of ways, from completely automatically to interactively, via a web interface.

The challenges include extracting relevant information from program and courses, coping with the size of the constraint satisfaction problem, and finding effective ways of interacting with the student to solve the problem. We may not have time to address all of these challenges.

3 Learning Outcomes
1. Artificial Intelligence (Constraint Programming, Planning & Scheduling)
2. Human-Computer Interaction and Web development
3. Document Analysis and Data Processing
INDEPENDENT STUDY CONTRACT
PROJECTS

Note: Enrolment is subject to approval by the course convenor

SECTION A (Students and Supervisors)

UniID: ______u6342392_____
SURNAME: _______Tianshu_______ FIRST NAMES: _______Wang______________
PROJECT SUPERVISOR (may be external): _______Sylvie Thiebaux________________________
FORMAL SUPERVISOR (if different, must be an RSSCS academic): _______Sylvie Thiebaux__________________
COURSE CODE, TITLE AND UNITS: _______COMP4560-Advanced Research Project__________

COMMENCING SEMESTER □ S1 □ S2 YEAR: 2018-2019 Two-semester project (12u courses only): ×

PROJECT TITLE:
ANU Study Planner

LEARNING OBJECTIVES:
1. Artificial Intelligence(Constraint Programming, Planning & Scheduling)
2. Human-Computer Interaction and Web development
3. Document Analysis and Data Processing

PROJECT DESCRIPTION:
The project focuses on solving a common question that ANU students have: how can I choose a study plan (courses, schedule), which fit my interests and the requirement of my degree program? The project will produce a web-based tool that enables them to construct such study plans in a range of ways, from completely automatically to interactively, via a web interface.

The challenges include extracting relevant information from program and courses, coping with the size of the constraint satisfaction problem, and finding effective ways of interacting with the student to solve the problem. We may not have time to address all of these challenges.
ASSESSMENT (as per the project course’s rules web page, with any differences noted below).

<table>
<thead>
<tr>
<th>Assessed project components:</th>
<th>% of mark</th>
<th>Due date</th>
<th>Evaluated by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report: style: 45</td>
<td></td>
<td>(min 45, def 60)</td>
<td>(examiner)</td>
</tr>
<tr>
<td>(e.g. research report, software description...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Artefact: kind: 45</td>
<td></td>
<td>(max 45, def 30)</td>
<td>(supervisor)</td>
</tr>
<tr>
<td>(e.g. software, user interface, robot...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Presentation: 10</td>
<td>(10)</td>
<td></td>
<td>(course convenor)</td>
</tr>
</tbody>
</table>

MEETING DATES (IF KNOWN): Weekly

STUDENT DECLARATION: I agree to fulfil the above defined contract:

Signature: Tian Zhu Wang  Date 22 July 2018

SECTION B (Supervisor):

I am willing to supervise and support this project. I have checked the student's academic record and believe this student can complete the project. I nominate the following examiner, and have obtained their consent to review the report (via signature below or attached email)

Signature: Sylvie Thiebaux  Date 22 July 2018

Examiner:

Name: John Slaney  Signature: 
(Nominated examiners may be subject to change on request by the supervisor or course convenor)

REQUIRED DEPARTMENT RESOURCES:

Appropriate time of Sylvie Thiebaux
Lincoln Smith has offered to help regarding the extraction task, as this project is of interest to the new Software Engineering Institute.
SECTION C (Course convenor approval)

…………………………………………………………
Signature

…………………………………………………………
Date
Appendix C: Artefact Description
List of files

- data_process.py
- general.mzn
- index.py
- plan.txt
- re-plan.txt
- README.md
- README.pdf
- test1.dzn
- test1.mzn

- experiments
  - experiment1.dzn
  - experiment1.mzn
  - experiment2.dzn
  - experiment2.mzn
  - experiment3.dzn
  - experiment3.mzn
  - experiment4.dzn
  - experiment4.mzn
  - experiment5.dzn
  - experiment5.mzn
  - experiment6.dzn
  - experiment6.mzn
  - experiment7.dzn
  - experiment7.mzn
  - experiment8.dzn
  - experiment8.mzn
  - experiment9.dzn
  - experiment9.mzn

- static
  - css
    - style.css
  - images
    - 2x_anu_logo_small.png
    - 2x_anu_logo_small1.png
    - 2x_anu_logo_small_over.png

- templates
  - tree.html

- __pycache__
  - data_process.cpython-35.pyc
  - index.cpython-35.pyc
1 Statement

These above programs were implemented by myself, except for

1. data_process.py - I implemented this program on the basis of a scraper library that built by Lincoln Smith;

2. tree.html - I developed JavaScript part in this program based on an online website which shows the basic usage of d3.js library.

2 Explanation

1. data_process.py. This is our pre-processing program which converts data from P&C to MiniZinc model components (for many programs).

2. general.mzn. It declares constants, variables, constraints and the output format that are applicable to every program.

3. index.py. Data is received from the front end, and through this program, data is delivered to the back end.

4. plan.txt & re-plan.txt. When our constraint model works, in the initial phase, the result (course scheduling) from our constraint model is stored in plan.txt; and in the refining phase, the result is stored in re-plan.txt.

5. test1.mzn & test1.dzn. They are comprised of a completed model generated by data_process.py for Master of Computing without setting preference for courses.

6. experiment1.mzn & experiment1.dzn .. experiment9.mzn & experiment9.dzn. They are used to evaluate the relationship between number of constraints, branches and the corresponding running time.

7. style.css & images. These files define the style of our user interface. Based on the rules in Flask, these static files go to the static folder.

8. tree.html. We built our user interface in this HTML file (includes JavaScript script) in templates, as regulated by Flask.
Appendix D: README File
Artefact Description

This is an Automated Study Plan Generator which is applicable for Master of Computing (MCOMP) students in ANU. Due to lack of data, and significantly varied degree requirements formats, this generator can only be applied for MCOMP.

Here, we attempt to explain the functionality of these programs below:

- data_process.py
- test1.mzn, test1.dzn and general.mzn
- experiment1.mzn & experiment1.dzn to experiment9.mzn & experiment9.dzn
- plan.txt and re-plan.txt
- style.css
- tree.html
- index.py

Pre-processing

- data_process.py. The goal in this program is converting data from P&C to MiniZinc model components (for many programs).

This was firstly written by Lincoln Smith, and kindly given to Tianshu Wang for the use of this Honour project.

Tianshu Wang is responsible for the function buildAModel and some other assistant functions. Lincoln wrote the scraper program.

Many thanks to Lincoln for his great help and contribution in this project.

Constraint Model

We built our model in MiniZinc, a standard language for Constraint Programming.

*.mzn. This type of files is used to produce course schedule (selecting OSICBC solver when testing it).

*.dzn. This type of files are data files in MiniZinc.

- test1.mzn and test1.dzn are comprised of a completed model generated by data_process.py for Master of Computing without setting preference for courses.
- general.mzn declares constants, variables, constraints and the output format that are applicable to every program.
- In the initial phase, the result from our constraint model is stored in plan.txt. In the refining phase, the result is stored in re-plan.txt.
- Those experiment files experiment1.mzn & experiment1.dzn to experiment9.mzn & experiment9.dzn are used to evaluate the relationship between number of constraints, branches and the corresponding running time.

Website Construction

This web-based system is under Flask - a microframe server.

- Based on the rules in Flask, we store style.css and several images in static folder, and the main page tree.html (includes javascript code) is in templates.
- index.py receives data from the front end, and deliver it to the back end.

Usage

In Windows environment, open Anaconda Prompt, after getting into the target folder, write these two lines of command:

```bash
set FLASK_APP=index.py
flask run
```

You shall get a result like this below, and the next step is to open the url to access our website:

```
Environment: production
WARNING: Do not use the development server in a production environment.
```
In our system, click Let's Plan when you finish the first input of course preference.

After generating a plan, to refine the plan, you should click on the undesired courses in the plan and after that click on Update.

However, anytime during using the system, resetting preference for courses is allowed.

**Environment Requirement**

We developed and tested this artefact with Python 3.5.5, BeautifulSoup 4.6.0, Flask 1.0.2, MiniZinc 2.2.3 under Windows10.

We recommend testing with the same or higher version of software or libraries that have been applied.
Bibliography


Rossi, F.; BecK, P. V.; and Walsh, T. Constraint Programming. (cited on page 1)


