LUNCH: an Answer Set Programming System for Course Scheduling

Ana Y. F. de Lima[1*] , Briza M. D. de Sousa[1*] , Daniel P. Cardeal[1*] , Jessica Y. N. Sato[1*] , Lorenzo B. Salvador[1*] , Renato L. Geh[1,2] , Bruna Bazaluk[1]

[1]Department of Computer Science
Institute of Mathematics and Statistics
University of São Paulo

[2]Computer Science Department
University of California, Los Angeles

{ayf.lima,brizamel.dias,cardeal.daniel,jyns1703,lorenzobs}@usp.br

renatolg@cs.ucla.edu, bazaluk@ime.usp.br

Abstract. *Timetable scheduling is a known NP-hard problem; despite this, there have been many efforts to enable fast and efficient algorithms and heuristics for such a challenging task. Within the realm of timetable scheduling lies the particularly complex problem of Course Scheduling (CS). The goal in CS is to find an optimal timetable configuration of courses within the constraints set by faculty, course requirements and departmental functions. Answer Set Programming (ASP) is a declarative logic programming paradigm for solving combinatorial search tasks; instead of explicitly writing the solution to the problem, ASP programs define the problem's constraints and knowledge in a high-level language, leaving the model search to a highly optimized solver. In this work, we construct and showcase* LUNCH*, an easily extensible, free and open-source system for course scheduling. Notably, we study the use of ASP for course scheduling within the specific context of the Computer Science Department at the University of São Paulo, showing how* LUNCH *fares against the manual scheduling done in previous years.*

1. Introduction

Timetable scheduling is a daunting task; often, it takes a tremendous amount of effort to manually coordinate time restrictions and find a reasonable time grid that best suits the constraints of all parties. Although manual scheduling is feasible for small problems, the search space of possible solutions quickly scales up to an impractical size as the number of variables and restrictions increases. In fact, it is widely known that timetable construction problems are NP-complete [Cooper and Kingston 2005]. Despite this, it is not unusual for timetable scheduling to be done manually in practice, even in the presence of large quantities of variables.

One practical and particularly interesting example of timetable scheduling is university Course Scheduling (CS). The usual goal in CS is, given a grid of time slots available for classes, to optimize for a valid timetable configuration that adheres to the constraints imposed by course requirements, faculty availability, student preferences, and

[*]Equal contribution.

(possibly multiple) degree(s) prerequisites. All these components make CS a challenging task to solve, both manually and automatically. Surprisingly, from our experience, manually finding valid solutions to CS seems to be the norm in universities.

There have been many approaches to tackling the problem of course scheduling [Schaerf 1999, Carter and Laporte 1998, Daskalaki et al. 2004, Holm et al. 2022, Alghamdi et al. 2020], usually using integer programming [Phillips et al. 2017, Cataldo et al. 2017] or swarm algorithms [Larabi Marie-Sainte 2015, Abayomi-Alli et al. 2019]. However, the first requires careful modeling of real-world constraints as an integer programming problem, while the latter involves meticulous selection of parameters and fitness functions that often depend on the constraints themselves. Indeed, most of the existing algorithms for timetable scheduling either require expert knowledge in order to construct and update the database of CS constraints, or their performance hinges on a careful choice of parameters, which may often depend on the constraints themselves.

Answer Set Programming (ASP) is a relatively recent programming language paradigm for describing and solving combinatorial problems [Baral 2003, Gelfond and Lifschitz 1991]. It is strongly linked with logic programming languages and constraint solving, offering a powerful declarative language for intuitively describing domain knowledge. Importantly, ASP provides a programming language aimed at specifying the set of constraints that determine the feasibility of solutions rather than focusing on the process itself.

For this reason, ASP is especially suited for solving CS, as the many constraints in course scheduling are usually simple and explicit, and thus easily translatable to the ASP language; while implementing the same constraints in more traditional programming languages could result in a much more complex codebase. Furthermore, new constraints in ASP are easy to introduce within existing code, allowing the scheduler to adapt as courses, faculties or other restrictions change.

In this work, we explore the use of ASP for course scheduling. To do so, we construct the Logic-based UNiversity Course scHeduler system[1] (or LUNCH, for short), a free and open-source course scheduler that allows users to effortlessly introduce, remove or modify existing constraints in order to adapt the program to the needs of each school. To do this, we employ the expressive CLINGO ASP language and system [Gebser et al. 2012, Gebser et al. 2017] as an efficient solver for describing complex real-world constraints in CS as rules in ASP.

To demonstrate the capabilities of LUNCH, we present a case study of our system's application in course scheduling for the Computer Science Department of the Institute of Mathematics and Statistics at the University of São Paulo (IME-USP). The department has a total of 58 available subjects that can be offered per semester. Faculty consists of 40 lecturers, with the department usually offering 120 hours of classes per week every semester. We show that the system is capable of efficiently coping with such data, achieving objectively better time grids (with respect to some metrics) compared to traditional manual scheduling. More importantly, these solutions are computed at a fraction of the time of the usual hours-long manual labor done by faculty.

[1] Available at `https://github.com/apoiobcc/lunch`.

This work is structured as follows: Section 2 briefly introduces ASP and formalizes course scheduling. In Section 3, we detail the system and showcase its features, taking the task of scheduling for the IME-USP Computer Science Department as a working example. Next, in Section 4, we evaluate LUNCH's performance and compare it against the manual scheduling procedure currently in place. Finally, Section 5 provides a conclusion and possible future work.

2. Background

We start with a brief review on answer set programming and timetable scheduling.

2.1. Answer Set Programming

Answer Set Programming (ASP) is a powerful declarative logic programming language for solving combinatorial problems. A logic program is a finite set of disjunctive rules of the form

$$h_1, \ldots, h_n \texttt{ :- } b_1, \ldots, b_m, \texttt{ not } b_{m+1}, \ldots, \texttt{ not } b_{m+k}.$$

where, given rule r, $H(r) = \{h_i\}_{i=1}^{n}$ is the *head* and the atoms right of `:-` are the *body* $B(r) = \{b_i\}_{i=1}^{m+k}$. Further, each b_i (preceded or not by a `not`) is called a *subgoal* of r and the operator `not` denotes default negation. Default negation is closely related to negation as failure, although slightly distinct in its interpretation due to the stable model semantics. We denote by $B^-(r)$ the subset of atoms in $B(r)$ which are preceded by `not`. An *atom* is either a constant or a predicate $p(t_1, \ldots, t_n)$, with each t_j either a constant or logical variable. A rule r is *disjunctive* if $|H(r)| \geq 2$, an *integrity constraint* if $|H(r)| = 0$ and *normal* otherwise. A *fact* is a normal rule with no atoms in its body. Intuitively, a rule indicates that the head must be true if the body has been satisfied.

The *Herbrand base* $\mathcal{H}$ of a program is the set of all possible ground atoms built from predicate names and constants. *Grounding* a rule amounts to replacing variables with constants from the Herbrand base in every consistent way; grounding a program consists of obtaining the grounding of each rule in the program. The semantics of a program is the semantics of the program after grounding.

An *interpretation* $\mathcal{I}$ of a program is a valid subset of $\mathcal{H}$. A *model* $\mathcal{I}$ of a program is an interpretation that satisfies all rules in the program. Further, model $\mathcal{I}$ is *minimal* if and only if no other model $\mathcal{J}$ exists such that $\mathcal{J} \subset \mathcal{I}$. The usual semantics of an ASP program, which is precisely the one we adopt in this work, is based around the concept of stability. The reduct $P|\mathcal{I}$ of a (ground) program relative to a set of ground atoms $\mathcal{I}$ is obtained by (1) deleting every rule r such that $B^-(r) \cap \mathcal{I} \neq \emptyset$, and (2) deleting all $B^-(r)$ from every other remaining rule. If $\mathcal{I}$ is minimal, then it is a *stable model* (or answer set) of P. A stable model, in course scheduling, is equivalent to a timetable output that respects all the constraints set by the problem.

CLINGO is a system that implements an answer set solver and grounder for the ASP language [Gebser et al. 2012, Gebser et al. 2017] under the stable model semantics. Crucially, the system implements a solver for optimizing so-called weak rules.

A *weak rule* r in ASP is a tuple (c, w, p), syntactically denoted as

$$\texttt{:}\sim b_1, \ldots, b_m, \texttt{ not } b_{m+1}, \ldots, \texttt{ not } b_{m+k}. \texttt{ [}w\texttt{@}p\texttt{]}$$

where c is an integrity constraint with body $B(c) = \{b_i\}_{i=1}^{m+k}$, w is the weight of the rule, and p is its priority. In contrast with regular integrity constraints, a weak rule does not rule out atoms from the set of stable models; instead, it defines (possibly multiple) partial order(s) over the answer sets of the program. In fact, every priority p defines a partial order $\prec$ for every other weak rule with same priority, with such order given by $w_1 \prec w_2$ for every two weak rules $r_1 = (c_1, w_1, p_1)$ and $r_2 = (c_2, w_2, p_2)$ subject to $p_1 = p_2$. Each stable model of the program is then associated with a weight W which amounts to the sum $W = \sum_i w_i$ for each weak rule $r_i = (c_i, w_i, p_i)$ such that $B(c_i)$ is satisfied. Usually, the goal is to minimize the cost, with the output of the program consisting of the set of all stable models together with their weights.

Weak rules enable ASP solvers to work in layers of optimization, in what effectively amounts to aggregating weak rules by their priority and cascading the weight minimization process from higher to lower priority. This layer-by-layer process helps scale the optimization of solving for the (weighted) stable models. This procedure plays an important role in optimizing for scheduling preferences (also known as *soft constraints* within the timetable scheduling community), as we shall see in the next sections.

Typically, ASP programs consist of a description of possible candidate solutions through choice rules followed by a set of constraints that determine feasibility and optimality (of stable models). A choice rule is an aggregate rule r of the form

$$\{\texttt{p(X}_1\texttt{, ..., X}_n\texttt{) : q(X}_i\texttt{, ..., X}_k\texttt{)}\} \preceq n \texttt{ :- } B(r)\texttt{.}$$

where the content within curly braces has an equivalent meaning to set theory: a set P containing all combinations of predicates of $\texttt{p(X}_1\texttt{, ..., X}_n\texttt{)}$ subject to the grounding of $\texttt{q(X}_i\texttt{, ..., X}_k\texttt{)}$. The operator $\preceq$ defines some comparison (e.g. ==, <, >, etc.) to a positive integer n in order to restrict the size of P. In other words, if the body of the choice rule is true, then it generates all possible combinations of atoms in the head subject to the cardinality constraint given by $\preceq$ and according to some grounding given by the conditional predicate $\texttt{q(X}_i\texttt{, ..., X}_k\texttt{)}$.

ASP further allows other aggregates as extensions to the language to enable rules with cardinality constraints in the body, summation over weights and counting over atom sets [Calimeri et al. 2020].

2.2. Course Scheduling

Let us first describe a common definition of the course scheduling problem, usually referred to as the class-teacher model [de Werra 1985]. Consider a set of m classes $\{c_1, c_2, \ldots, c_m\}$, a set of n teachers $\{t_1, t_2, \ldots, t_n\}$ and p time slots. Furthermore, let $R_{m \times n}$ be a positive matrix such that $r_{i,j}$ indicates the number of lectures taught by lecturer j for the course i. The timetabling problem can then be described as finding a matrix $X_{m \times n \times p}$ such that the following constraints hold

$$\sum_{k=1}^{p} x_{i,j,k} = r_{i,j}, \quad (1) \qquad \sum_{j=1}^{n} x_{i,j,k} \leq 1, \quad (2) \qquad \sum_{i=1}^{m} x_{i,j,k} \leq 1; \quad (3)$$

where $x_{i,j,k} = 1$ if teacher t_j lectures class c_i during slot k, and $x_{i,j,k} = 0$ otherwise. The equation given by (1) ensures that the number of required lectures of a given class are respected, while (2) constraints classes to be assigned to only one teacher and (3) models the fact that a teacher can only give at most one class at a given time.

Table 1. LUNCH output for the first semester of 2023. Bold indicates core courses and colors show the different fields of study these courses belong to.

	MON	TUE	WED	THU	FRI
08h-10h	**CS0110** CS0460|CS5832 CS6937	**CS0329** **CS0210** CS0691|CS6918	**CS0422** **CS4722** CS6937	**CS0323** **CS0350** BI5037	**CS0105** CS0345 CS0420|CS5744
10h-12h	**CS0422** **CS6711**	**CS0323** **CS0350** **CS6711** BI5037	**CS0110** CS0345 CS0460|CS5832 CS0420|CS5744	**CS0329** **CS0210** CS0691|CS6918	**CS4722**
14h-16h	**CS0209** MA0223 CS0336|CS5723	**CS0101** CS0320|**CS5770**	**CS0105** CS6931	CS0427 **CS0102**	CS6931
16h-18h	CS0427 **CS0209**	CS0219|**CS5742** CS0417|CS5768 CS6956	MA0223 CS0336|CS5723	CS0219|**CS5742** CS0320|**CS5770** CS0417|CS5768 CS6956	CS6989

This definition may be further extended to also consider teacher or classroom unavailability at determined time slots, effectively turning the task into a simple search problem [Schaerf 1999]. Subsequent works posed course scheduling as an optimization problem in which candidate solutions are compared based on some criteria of quality, such as the distance between lectures of a teacher or course.

Since ASP is capable of generating a feasible output based only on logic predicates as input and a set of simple constraints describing domain knowledge, we leverage the expressiveness and efficiency of ASP to provide an intuitive and flexible language to describe course scheduling. Particularly, logic programming allows us to shift scheduling from the low-level of matrix optimization to a more symbolic high-level abstraction, which we now describe.

A timetable is a grid of week days, time slots and lectures across multiple courses. We define the course timetabling problem as the task of distributing lectures of each given course in a timetable, subject to some predefined rules. As an example, consider Table 1, where the available time slots go from 8AM to 6PM and week days from Monday to Friday. Here, each time slot, say 10AM on a Tuesday, contains a list of assigned courses; these course assignments then become predicates in an ASP knowledge base, together with data concerning lecturer availability, degree requirements and course information.

Rules, also referred to as constraints, are divided into two main groups: hard and soft constraints. The former is composed out of mandatory rules, which must be satisfied in order for an answer to be considered a solution; for instance, the fact that a lecturer cannot offer two courses at the same time. The latter consists of optimization criteria used to guide the scheduler towards timetables that are perceived as good solutions by some metric; for example, major core courses should not overlap with one another, although such conflicts are allowed to happen if truly necessary.

In ASP, these two types of constraints are usually modeled through a combination of normal rules, disjunctive rules and integrity constraints for hard constraints, and weak rules for soft constraints. A feasible solution to course scheduling is then a stable model of the program, while the optimal solution (with respect to soft constraints) is the *least weighted* stable model. In the next section, we describe LUNCH and give some examples

on how hard and soft constraints can be modeled in ASP and embedded onto our system.

3. LUNCH: a FLOSS System for Course Scheduling

We introduce LUNCH, a FLOSS (Free/Libre and Open Source Software) system for assisting in the process of university course scheduling. LUNCH's goal is to provide a framework for specifying the requirements of each degree program by means of an expressive and declarative language based on ASP. Particularly, we aim to provide a simple yet flexible interface where the user is required only to specify the course requirements in ASP and to provide as input the set of classes, lecturers, and possible time slots for scheduling to take place.

LUNCH acts in a two-part procedure. First, the user passes data containing information about lecturers and courses to be scheduled; the system then populates a knowledge base containing logic predicates summarizing the information from this input. After that, the system reads the (soft and hard) constraints defined by the user, which are usually determined by the nature of the user's domain, together with the generated predicates. These are then passed on to the CLINGO ASP system [Gebser et al. 2017], a highly optimized ASP solver. Finally, CLINGO proposes a set of stable models representing feasible schedules; these are parsed back into LUNCH and displayed to the user in a timetable format.

In order to provide a clear description of LUNCH, we showcase real-world examples of course requirements from the IME-USP Computer Science Department currently in use and show how to translate them into ASP in a format compatible with LUNCH. We then demonstrate the performance of our system across actual data spanning five years and compare them against the timetables (manually) scheduled by the Department.

3.1. Input

To better understand the complexity of course scheduling and how LUNCH is able to cope with this, we must first address what the system takes as input. Essentially, LUNCH accepts two files (usually in CSV) as inputs from the user: the `lecturers` dataset, which contains information concerning the available time slots of lecturers, and the `courses` dataset, which deals with essential information about courses, such as the number of classes per week or which lecturer has been assigned to that course.

The `lecturers` input dataset contains the availabilities and preferences for each week day, along with a unique identifier for each lecturer. Table 2 provides a simplified example of the contents of this file (detailed documentation is available at the repository link); here, lecturer `dknuth` has a preference for their lectures to take place at 8 AM and 2 PM on Mondays and is unavailable from 10 AM to 12 PM on the same week day. Entries (except for the unique identifier) can optionally be empty, in which case the system assumes the lecturer has no preference or restriction that day; for instance, `vneumann` has set no time preference nor restrictions for Mondays in Table 2.

The `courses` input dataset contains data about the courses to be scheduled. Table 3 shows an example of such a dataset. The number of units assigned for each course corresponds to the number of time slots assigned weekly for that class; e.g. CS0101 is a single two-hour class per week, while CS2400 is offered three times a week. It is worth

ID	Mon (P)	···	Fri (P)	Mon (R)	···	Fri (R)
dknuth	8:00-10:00;14:00-16:00		—	10:00-12:00		16:00-18:00
rbluth	10:00-12:00		14:00-16:00	—		—
vneumann	—		16:00-18:00	—		10:00-12:00
mccarthy	—		8:00-10:00	14:00-16:00;16:00-18:00		—
noether	14:00-16:00		10:00-12:00;16:00-18:00	10:00-12:00		10:00-12:00;14:00-16:00

Table 2. The lecturers dataset input contains identifiers, and preferred (marked with a P) and unavailable (marked with an R) time slots for each lecturer. Empty entries denote no preference (for P columns) or no restriction (for R columns).

ID	Name	Major	Units	Time	Lecturer
CS0101	Intro to CS	Computer Science	1	—	dknuth
CS0211	Computer Architecture	Computer Science	2	—	vneumann
MA0311	Galois Theory for CS	Computer Science	2	Mon 14:00-16h00, Thu 16:00-18:00	noether
CS1532	Monte Carlo for Physicists	Physics	3	—	rbluth
CS2400	Foundations of AI	Computer Science	3	—	mccarthy

Table 3. Each entry in the courses dataset input lists the ID, name, major to which that course belongs to, the weekly workload in time slots, an optional predetermined time that is always allocated to that course, and the lecturer's ID.

mentioning that LUNCH also accepts pre-allocating time slots for specific courses; for instance, MA0311 in the aforementioned example has been assigned to Mondays at 2 PM and Thursdays at 4 PM by the Department, and as such the scheduler may not reschedule it.

Once the input has been fed to LUNCH, a knowledge base is populated with the predicates describing the two datasets. These predicates are made available for the user to define constraints through ASP scripts, which are then passed to the CLINGO solver for optimization and inference.

3.2. Declaring Constraints

LUNCH follows a structure based on files. An ASP file encodes one or more constraints to be taken into account during scheduling; these constraints usually represent at least one course or degree restriction. These files may possibly define predicates to be used in other constraint files, although good practice advises against this, as the intent is to make sure constraints are self-contained so that updating the knowledge base is straightforward. Each constraint may use the set of already defined predicates automatically generated by the input procedure.

To provide some intuition and exemplify how soft and hard constraints are implemented within LUNCH, we now look at a particular set of scheduling rules as defined in our real-world case study, showing how they can be written in ASP. Before this, however, we must first go through notation and syntax.

We write HC_i (resp. SC_i) to denote the i-th hard (resp. soft) constraint in our running example. We adopt the same ASP syntax as CLINGO; namely, an upper-case string that starts with a letter within an ASP rule denotes a variable, while lower-case strings are either predicates or constants. A predicate prepended with an `@` indicates an external function written in Python. The evaluation of the function must then return either an atom, integer, or string. This feature enables more expressive constraints to be modeled within ASP.

The constraints below are selected department-required course restrictions for our

running example. An extensive list of all such constraints is available in Appendix A.

HC_2: All courses defined in the input must be offered by the department.
HC_3: Weekly offers of each course must match their assigned units.
HC_4: Classes must be scheduled according to the availability of each lecturer.
HC_6: Core courses offered in the same recommended period must not conflict.
SC_7: Graduate-level courses that belong to the same area of study should not conflict with one another.

HC_2, HC_3 and HC_4 can be encoded through a single choice rule

```
{ class(C, G, T, P) : available(T, P, R) } == N :- course(C, G, T, N).
```

that generates all possible time slot assignments for every course as long as the lecturer is available at that time (as per condition HC_2). These assignments are represented by the predicate `class(C, G, T, P)`, which essentially denotes that the course identified by its ID `C`, taught by lecturer `T` for students of major `G` has been assigned to time slot `P`. As the rule suggests, these assignments are conditioned by the number `N` of units defined in `course(C, G, T, N)` (fulfilling condition HC_3) and the availability of lecturer `T` (fulfilling HC_4), encoded by predicate `available(T, P, R)`, where `R` is one if `P` is a preferred time slot and zero otherwise.

Note that predicates `available(T, P, R)` and `course(C, G, T, N)`, alongside a few other predicates, are automatically generated from the input as previously mentioned. One other such automatically generated predicate is `conflict(C1, G1, C2, G2, P)`, which defines a (potential) conflict between two (distinct) courses taking place at the same time.

```
conflict(C1, G1, C2, G2, P) :- class(C1, G1, _, P),
                               class(C2, G2, _, P),
                               C1 != C2.
```

An important remark is that the presence of a `conflict` is not necessarily undesirable; in fact, any overlap between two courses — even if their assignments are otherwise perfectly valid — configures a `conflict`.

Apart from choice rules, which generate candidate solutions to our problem, we may further add other constraints through normal rules or integrity constraints in order to model the remaining conditions for scheduling. As an example, consider hard constraint HC_6, which calls for core courses to not conflict with other core courses expected to be taken concurrently.

```
:- obligatory(C1, STAGE), obligatory(C2, STAGE),
   conflict(C1, G, C2, G, P).
```

Here, `obligatory(C1, STAGE)` and `obligatory(C2, STAGE)` — another predicate generated by LUNCH from input — indicate that the courses represented by `C1` and `C2` should ideally be taken at the same `STAGE` of the degree, meaning that this integrity constraint removes any interpretation from the set of stable models where both `C1` and `C2` are assigned to the same time slot.

As previously briefly mentioned, soft constraints are implementable through weak rules. The weight of a weak rule acts by penalizing the score of a stable model which satisfies the body of the weak rule, while the rule's priority allows for different soft constraints to be optimized separately. Let us ground this into a more concrete example: we now model SC_7 as a weak rule.

```
:~ conflict(C1, G1, C2, G2, P),
    C1 > C2,
    postgrad(C1), postgrad(C2),
    curriculum(C1, CUR1, _), curriculum(C2, CUR2, _),
    W = @calculate_weight_sc07(CUR1, CUR2),
    Pr = @get_priority("sc07").
    [W@Pr,"sc07",C1,G1,C2,G2]
```

The rule states that, if two graduate level courses `postgrad(C1)` and `postgrad(C2)` are offered on the same time slot `P` (and thus `conflict(C1, G1, C2, G2, P)` holds), then we penalize this by a weight `W` set by a Python function `@calculate_weight_sc07`, which takes as arguments the areas of study `CUR1` and `CUR2` of `C1` and `C2` respectively. Similarly, we establish a priority `Pr` through another external function `@get_priority`. These external functions are implementable by surrounding Python code with `#script(python) #end.` guards as specified in [Gebser et al. 2019]. The condition `C1 > C2` is to ensure that the weak rule applies only once to every pair {`C1`,`C2`}.

As mentioned before, weak rule priorities allow us to separate the optimization of weak rules into layers. Layering acts both as a means to scale optimization — as the number of weak rules after grounding may cause the optimization of the program to become intractable — as well as to prioritize certain soft constraints, making sure these are met before other requirements.

4. Experiments

We now evaluate the performance of LUNCH in a real-world setting. We implement the soft and hard constraints set by the course scheduling department staff and evaluate LUNCH's performance against the historical (manual) scheduling done in the department throughout a period of five years, totaling ten semesters worth of data about faculty time availabilities, preferences, and course offerings.

With respect to weak rule priorities in our experiments, most of the soft constraints were included in the first layer, as we aim to find a grid that optimizes them concurrently. The second layer contains only the single weak rule SC_{10}, described in Table 5. This weak rule grounds a possibly enormous amount of soft constraints, as it not only requires permuting over each of the preferred time slots for every lecturer but also over every one of the courses they teach. Thus, by separating weak rules that have many groundings, we avoid cases where one weak rule eclipses others. We isolate SC_{13} to the third layer for the same reason.

Given that the problem at hand is in the general case NP-complete, it is reasonable to assume that the optimal solution is rarely achieved. Instead, we show how CLINGO's highly optimized solver can yield sufficiently good answers. Not only that but since the optimization is anytime [Zilberstein and Russell 1996], we can obtain better solutions if we have a higher time budget, as discussed in Section 4.2.

In order to objectively measure the quality of the scheduler, we evaluate LUNCH with respect to three distinct metrics. The first two quantify the tightness of soft constraints, while the third shows how the optimization quickly reaches a reasonable solution over time. Note that we do not evaluate against hard constraints as they are always satisfied (otherwise the program would have no stable model and thus would be unsatisfiable).

4.1. Measuring Soft Constraints

We define our two soft constraint evaluation metrics as M_1 the absolute number of occurrences of each soft constraint, and M_2 the sum of penalties, taking into consideration all soft constraints. More formally,

$$M_1 = \sum_{i=1}^{N} C_i, \qquad M_2 = \sum_{i=1}^{N} w_i \cdot C_i; \tag{4}$$

where N is the number of soft constraints in the model (in our case $N = 13$), C_i is the number of times the groundings of i-th soft constraint are satisfied in the program, and w_i represents the weight for weak rule i.

Intuitively, M_1 measures how many soft constraints could not be avoided. Figure 1 compares the output of LUNCH against the manual scheduling done by the department at that time, with each soft constraint colored differently. To effectively extract the M_1 value from the manual timetable, we fix each course to their assigned time slot, forcing the result to be equal to the ground truth and allowing it to be compared with LUNCH. Evidently, SC_{13} appears most frequently in both manual and LUNCH, as Figure 1 shows. This frequency is expected, as SC_{13} penalizes every `conflict` predicate from appearing. Interestingly, this (soft) constraint appears more frequently in LUNCH compared to the manual scheduling, heavily skewing the overall sum against LUNCH. This occurs since this constraint is assigned the lowest weight of all soft constraints, and so LUNCH prioritizes SC_{13} the least.

Instead of measuring the absolute number of occurrences of each penalization, M_2 measures the weighted occurrences, painting a more accurate picture of the true performance of schedulers. Figure 2 shows how much the performance changes under this weighting, clearly indicating how LUNCH prioritizes according to the weak rule weights, with LUNCH reaching up to a 74% reduction in the first semester of 2019 compared to manual scheduling. We use the same approach as the method described in metric M_1 for extracting the M_2 values from the manual timetables.

4.2. Measuring Execution Time

Because of the intractability of course scheduling, LUNCH employs a time limit when computing solutions. When this time limit is reached, LUNCH's solution search halts, and the scheduler returns the best timetable obtained up to that point. Thus, in order to evaluate the time it takes for the system to find a reasonable solution, we run LUNCH under increasingly longer time limits in different machines and compute the M_2 metric for each run.

Figure 3 shows M_2 average and standard deviation values for each of the time limits starting from the first semester of 2019 (2019.1) up to the second semester of 2023 (2023.2). Each bar corresponds to the average performance of one of the hardware described in Table 4, with the gray horizontal line showing the average M_2 value for the manual scheduling and the gray area showing the standard deviation. Runs were computed in parallel to best make use of available hardware.

Ultimately, we found that even under a low time limit, we are able to achieve reasonably good performance comparatively. In fact, our solutions were degrees of mag-

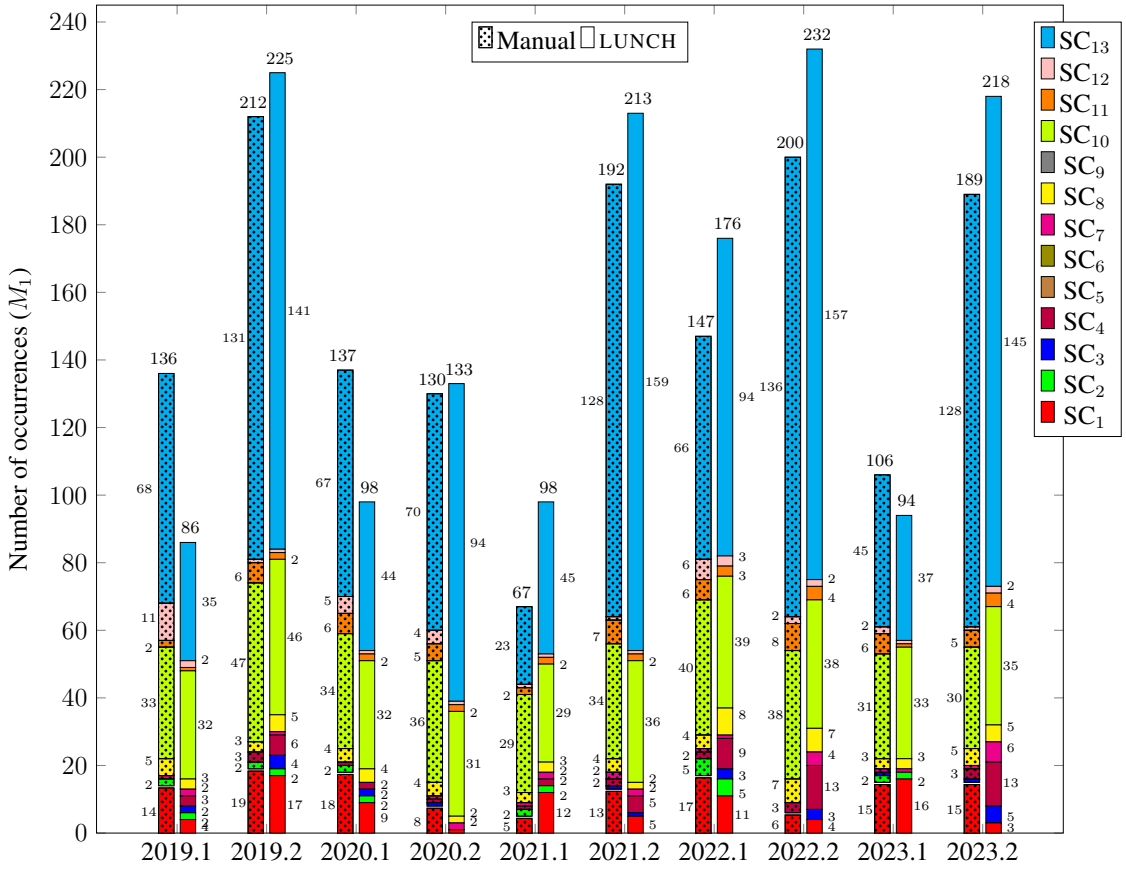

Figure 1. Performance of manual scheduling and LUNCH under the M_1 metric. Different colors show the number of occurrences of different soft constraints.

nitude faster compared to the several hours spent by department faculty when manually scheduling.

5. Conclusion and Future Work

We presented LUNCH, a flexible free and open-source scheduler for course scheduling built around the CLINGO ASP solver. We demonstrated how to model hard constraints in course scheduling as ASP rules and soft constraints as ASP weak rules. By optimizing the weights of weak rules through priorities, we are capable of modeling preferences in an expressive and scalable manner.

Notably, we evaluate LUNCH against the manual scheduling done at the University of São Paulo and found that our system is capable of achieving better timetables (with respect to a reasonably chosen metric) in a fraction of the time. Although such a result is unsurprising, we argue that an essential feature of a real-world implementation of a scheduler is its low entry barrier when it comes to defining constraints, and as such LUNCH provides an intuitive interface layer that allows for an expressive yet straightforward way of updating and adding new restrictions to the scheduler. It is also worth reminding that solutions yielded by LUNCH may serve as an initial timetable for further improvement according to the department's discretion.

Future work can target several directions: One of them would be to apply opti-

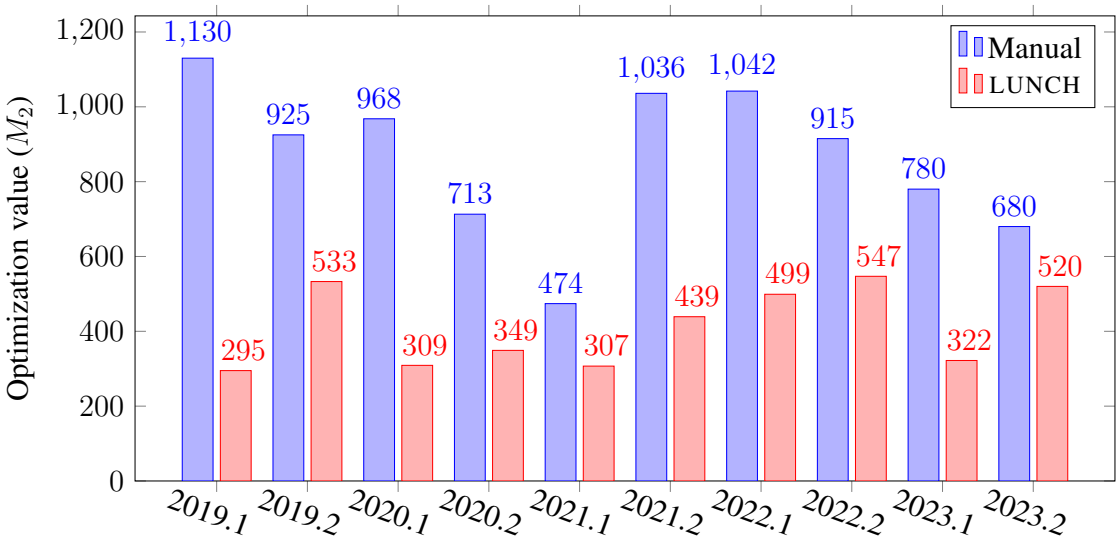

Figure 2. Performance of manual scheduling and LUNCH under the M_2 metric.

Table 4. Hardware configuration of the machines used in experiments.

	CPU Model Name	CPU MHz	Cores	Total Memory (GB)
Hardware 1	11th Gen Intel(R) Core(TM) i5-1135G7	1087	4	8
Hardware 2	11th Gen Intel(R) Core(TM) i5-1135G7	2400	8	8
Hardware 3	Intel(R) Core(TM) i5-3337U	916	8	6
Hardware 4	Intel(R) Core(TM) i5-8300H	1396	8	16

mization tricks in the generation of the input predicates; for instance, by reducing as much as possible the arity of predicates (and thus the number of groundings). Another relevant future work would be to incorporate as input student preferences on which courses to take simultaneously in that semester. Lastly, the development of a user-friendly graphical user interface for running the scheduler and updating the knowledge base is a planned feature.

References

Abayomi-Alli, O., Abayomi-Alli, A., Misra, S., Damasevicius, R., and Maskeliunas, R. (2019). *Automatic Examination Timetable Scheduling Using Particle Swarm Optimization and Local Search Algorithm*, pages 119–130. Springer Singapore, Singapore.

Alghamdi, H., Alsubait, T., Alhakami, H., and Baz, A. (2020). A review of optimization algorithms for university timetable scheduling. *Engineering, Technology and Applied Science Research*, 10(6):6410–6417.

Baral, C. (2003). *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F., and Schaub, T. (2020). Asp-core-2 input language format. *Theory and Practice of Logic Programming*, 20(2):294–309.

Carter, M. W. and Laporte, G. (1998). Recent developments in practical course timetabling. In *Practice and Theory of Automated Timetabling II: Second Interna-*

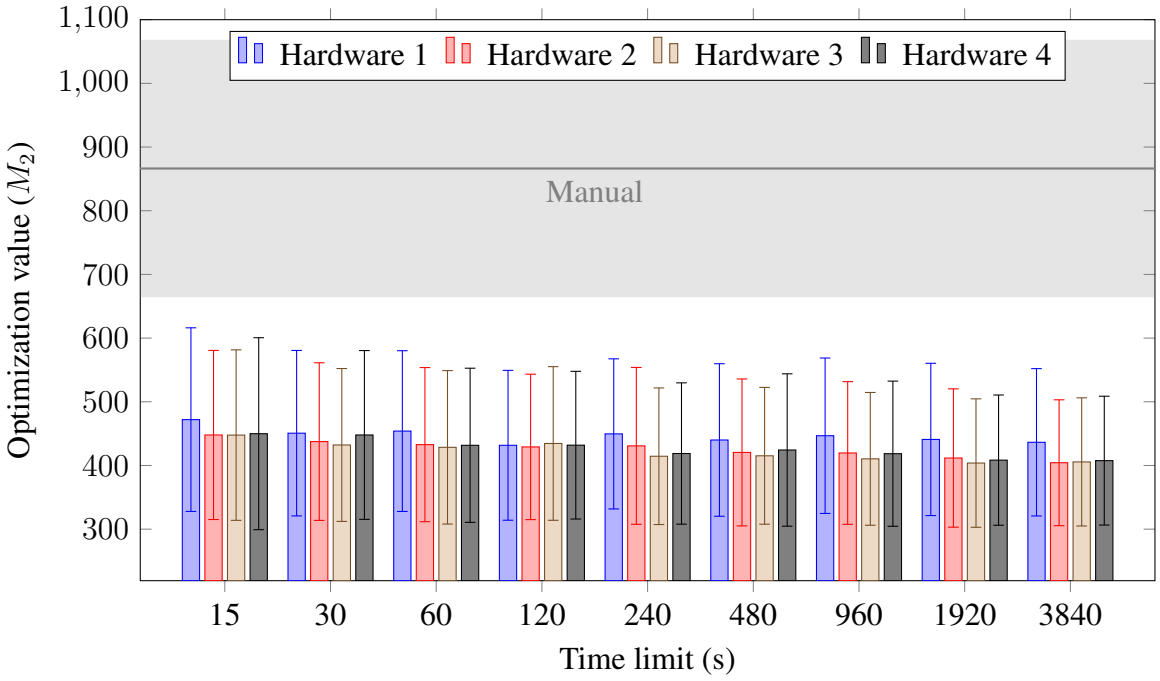

Figure 3. Performance of LUNCH under the M_2 metric conditioned on a time limit for execution halting. The abscissa describes the time limit in seconds and the ordinate the M_2 metric score. The gray horizontal line and area show the average and standard deviation M_2 values for the manual scheduling respectively.

tional Conference, PATAT'97 Toronto, Canada, August 20–22, 1997 Selected Papers 2, pages 3–19. Springer.

Cataldo, A., Ferrer, J.-C., Miranda, J., Rey, P. A., and Sauré, A. (2017). An integer programming approach to curriculum-based examination timetabling. *Annals of Operations Research*, 258(2):369–393.

Cooper, T. B. and Kingston, J. H. (2005). The complexity of timetable construction problems. *International Conference on the Practice and Theory of Automated Timetabling*.

Daskalaki, S., Birbas, T., and Housos, E. (2004). An integer programming formulation for a case study in university timetabling. *European journal of operational research*, 153(1):117–135.

de Werra, D. (1985). An introduction to timetabling. *European journal of operational research*, 19(2):151–162.

Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T., Thiele, S., and Wanko, P. (2019). *Potassco User Guide*, 2 edition.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2017). Multi-shot ASP solving with clingo. *CoRR*, abs/1705.09811.

Gebser, M., Kaufmann, B., and Schaub, T. (2012). Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89.

Gelfond, M. and Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385.

Holm, D. S., Mikkelsen, R. Ø., Sørensen, M., and Stidsen, T. J. (2022). A graph-based mip formulation of the international timetabling competition 2019. *Journal of Scheduling*, 25(4):405–428.

Larabi Marie-Sainte, S. (2015). A survey of particle swarm optimization techniques for solving university examination timetabling problem. *Artificial Intelligence Review*, 44(4):537–546.

Phillips, A. E., Walker, C. G., Ehrgott, M., and Ryan, D. M. (2017). Integer programming for minimal perturbation problems in university course timetabling. *Annals of Operations Research*, 252(2):283–304.

Schaerf, A. (1999). A survey of automated timetabling. *Artificial intelligence review*, 13:87–127.

Zilberstein, S. and Russell, S. (1996). Using anytime algorithms in intelligent systems. *The Springer International Series in Engineering and Computer Science*.

A. List of Hard and Soft Constraints

Table 5 shows all the hard and soft constraints implemented for producing the results in Section 4 and Table 1. To better understand Table 5, we now define the nomenclature used and some background information regarding our case study. A double course is a course whose scheduled time slots must come consecutively and on the same day. A joint course is a course taken by both graduate and undergraduate students. A curriculum is a set of courses (usually pertaining to the same area of study); each curriculum has a (possibly empty) set of mandatory and elective curriculum courses in order to fulfill the requirements of a degree specialization. A science course is one of a set of classes focused on natural or biological sciences; a Computer Science major undergraduate student is required to take at least one of such courses. Similarly, Computer Science majors must also take at least one statistics course from a list of courses given by the Statistics Department in order to fulfill their major degree requirement.

B. Output Example

Table 1 shows the output of LUNCH for the first semester of 2023 (2023.1). Course IDs beginning with the `CS` prefix denote classes offered by the Computer Science Department, while `BI` and `MA` come from the Bioinformatics and Mathematics Departments respectively. Classes whose IDs are represented by numbers in the thousands are graduate courses; otherwise, they are undergraduate classes. For instance, `CS6711` and `BI5037` are graduate courses, while `CS0345` and `MA0223` are undergraduate.

It is worth noting that some graduate-level classes are offered as upper-level undergraduate courses as well, meaning that they are effectively the same classes. Thus, both undergraduate and graduate constraints set by Table 5 should be enforced for these joint courses. These graduate and undergraduate joint courses are shown in Table 1 in the same line, with their course IDs separated by a |. An example of this is the pair `CS0417` and `CS5768`, which are shown as `CS0417|CS5768` in Table 1.

Courses in **bold** are core courses, i.e. mandatory for degree completion. Colors indicate the main area of study of each course (here often denoted as the *curriculum*); computer theory classes are shown in blue, systems in orange, artificial intelligence in

Table 5. List of hard and soft constraints implemented for Section 4.

ID	Constraint	Weight
HC_1	Two classes lectured by the same lecturer cannot conflict unless they are joint.	
HC_2	All courses in the input must be offered.	
HC_3	Courses must be given according to their units.	
HC_4	Courses are scheduled according to the availability of lecturers.	
HC_5	No undergraduate class is scheduled on a Friday afternoon.	
HC_6	No two core courses of same level may conflict.	
HC_7	1st and 2nd year core courses are fixed.	
HC_8	Double courses must be offered consecutively.	
HC_9	A course must not be offered on the same day unless it is a double.	
HC_{10}	Joint courses must be scheduled to the same time slot.	
SC_1	Core courses should not conflict with electives of same level.	0 – 20
SC_2	Core courses should not conflict with other core courses of different levels.	20
SC_3	Mandatory curriculum courses should not conflict with other courses of the same curriculum.	10
SC_4	Curriculum electives should not conflict with other electives of the same curriculum.	10
SC_5	Science courses should not conflict with other core courses.	5
SC_6	Statistics courses should not conflict with core courses from 2nd year forward.	10
SC_7	Graduate courses should not conflict with other graduate courses.	0 – 20
SC_8	Popular graduate courses should not conflict with each other.	25
SC_9	Graduate courses of same are of study should not conflict with each other.	10
SC_{10}	Courses should be offered according to lecturer's preferences.	5
SC_{11}	Classes should not be given on consecutive days.	10
SC_{12}	Classes should not be given at different periods of the day (morning/afternoon).	50
SC_{13}	Avoid all kinds of conflicts.	1

pink and data science in green. A course may be a part of two areas of study, in which case we color course IDs with both colors, e.g. CS0460 or CS0219.

Some of the scheduling constraints shown in Table 5 are evident from timetable Table 1 alone; for example, HC_5 which states no undergraduate course should be offered on a Friday afternoon, SC_{11} which calls for a class not to be offered on consecutive days (e.g. `CS0422` being scheduled on a Monday and Tuesday), and both SC_9 and SC_4 which prevent courses from the same area of study from being offered at the same time slot (i.e. they minimize the number of courses of the same color in a same cell).