A review of the Constraint Programming MOOC on EdX

Augustin Delecluse

TRAIL, ICTEAM, UCLouvain, Belgium

Guillaume Derval

ULiège, Belgium

Laurent Michel

UCONN, USA

Pierre Schaus

ICTEAM, UCLouvain, Belgium

Pascal Van Hentenryck

Georgia Tech, USA

Abstract

This paper delivers a review of our "Constraint Programming" Massive Open Online Course (MOOC) introduced on edX in January 2023. The course leverages the pedagogical solver MiniCP to provide an engaging educational approach by necessitating the students to implement key functionalities such as search components, global constraints and models. This course is the result of several earlier university courses, all of which utilized MiniCP, providing a rich heritage of practical learning and automated grading system. This review is structured to first explore relevant predecessor courses and works, followed by a detailed exploration of the MOOC's learning outcomes and structure. Further, it presents a brief overview of the framework enabling student coding and evaluation. Concluding sections offer a comprehensive statistical analysis of the MOOC's performance, considerations for future advancements, and insightful reflections from this educational endeavor.

2012 ACM Subject Classification Social and professional topics → Computing education; Applied computing → E-learning; Theory of computation → Constraint and logic programming

Keywords and phrases MOOC, edX, Teaching, Constraint Programming, Automation, Testing, Java

Digital Object Identifier 10.4230/LIPIcs.WTCP.2023.3

Funding *Augustin Delecluse*: This work was supported by Service Public de Wallonie Recherche under grant n°2010235 – ARIAC by DIGITALWALLONIA4.A

1 Introduction

We present a comprehensive review of the "Constraint Programming" Massive Open Online Course (MOOC) on edX started in January 2023. This MOOC is structurally centered around MiniCP [16], a pedagogically-oriented, minimalist solver, facilitating an immersive, end-to-end educational approach.

The course content is not merely a compendium of varied topics, but it also underscores the necessity for students to apply, and even program, the concepts they acquire. In this aspect, MiniCP serves as a vital educational tool. It strikes a fine balance between functionality and an intentionally incomplete design, necessitating students to complete its implementation, thereby making it more robust and efficient.

At the onset of the academic year, MiniCP only includes a few constraints, a functional recursive search algorithm, and the requisite infrastructure to operate as a trail-based or copy-based solver. Throughout the course's ten-week duration, students progressively enhance MiniCP's capabilities:

- they engage in programming propagators of most well known constraints;
- they have the opportunity to apply these concepts through various modeling exercises, including notable case studies;
- they implement custom search and branching methods.

This hands-on approach, supported by MiniCP, not only builds their understanding of the theoretical aspects but also equips them with practical problem-solving skills, effectively bridging the gap between theory and application.

The inception of the MOOC can be traced back to courses offered at different universities, all of which utilized MiniCP as their foundational element:

- *Discrete Optimization* by Laurent Michel at UCONN, USA.
- *Constraint Programming* by Pascal Van Hentenryck at Georgia Tech, US.
- *Combinatorial Optimisation and Constraint Programming* (COCP) by Pierre Flenner at Uppsala University, Sweden.
- *Constraint Programming* by Pierre Schaus at UCLouvain, Belgium.

The last course, in particular, was a trailblazer in the adoption of an automated grading system (INGInious [6]). This innovation significantly contributed to the scalability of the grading process for the MOOC, thereby playing a pivotal role in shaping the MOOC's structure. Currently, the course at UCLouvain is entirely composed of the MOOC.

The ensuing sections of this paper are organized as follows. Following a brief exploration of relevant courses and associated works, we delve into the learning outcomes and the structural layout of the course. Subsequently, we offer a concise description of the framework utilized to facilitate student coding and assess their results.

In the latter segments, we present a statistical analysis reflecting the MOOC's performance and engage in a discourse about future prospects. The manuscript concludes with reflections on the invaluable insights gained throughout this enlightening journey.

1.1 Related Works

MOOC's focused on Constraint Programming (CP) include [29, 30, 22]. These primarily delve into the modelling aspect of constraint programming, often utilizing Minizinc [18] as a tool. More specifically, they give problem statements and either present to the students how to model it in CP, or challenge the students to derive a model for it.

Although modelling is crucial to fully comprehend the wide-ranging potential of combinatorial optimization, these courses tend not to delve deep into the specifics of implementation. Additionally, they don't consistently provide an expansive overview of all components present in a CP solver.

It's worth highlighting another notable online course, [26]. This course provides a more in-depth understanding of CP through its application with ECLiPSe [34], but it solely comprises videos and slides, lacking theoretical queries or programming tasks. In contrast, our MOOC explores even more CP specifics, engaging students with multiple-choice questions and programming exercises. It is also adapted on edX, a platform tailored for MOOC's.

2 Learning Outcomes

A course's learning outcomes serve as this critical roadmap. These outcomes outline what knowledge and skills students should possess by the end of the course. They guide the design of the course content, the selection of suitable teaching strategies, and the development of assessments to measure student learning. Our MOOC on constraint programming is no exception. We have developed a set of learning outcomes that not only define what the students will learn, but also the skills and competencies they will acquire, providing a clear understanding of what successful completion of the course looks like. The following section outlines these learning outcomes. Those are split into the solver and modeling skills.

Solvers

- Gain familiarity with the architecture of a constraint programming solver.
- Understand advanced mechanisms within constraint programming, such as state restoration, domain implementation, and fix-point processes.
- Develop the ability to implement global constraints and propagators.
- Understand most popular black-box search techniques, specifically in the context of variable and value selection in constraint programming.
- Learn to implement a depth-first backtracking search within a solver and generic search combinators such as discrepancy search.

Modeling and Theory

- Engage with a wide range of combinatorial optimization problems, focusing specifically on vehicle routing and scheduling problems.
- Develop skills to test, extend, and improve existing code within constraint programming models.
- Understand the balance between pruning strength and time complexity, and the trade-offs that this entails. This also includes becoming familiar with the notion of consistency (domain, bound, etc).
- Gain the ability to manipulate and employ the most frequently used constraints within the field, including but not limited to sum, element, alldifferent, disjunctive, and cumulative constraints.
- Understand the mechanics and application of reified constraints within constraint programming.
- Learn to implement a problem specific search, variable and value heuristics.

Prerequisites to tackle the course include one datastructures and algorithms courses, as well as basic knowledge about Object-Oriented Programming. The target audience is mostly composed of master's students.

In addition to acquiring specific knowledge about constraint programming solvers and modeling skills, this course also aims to instill certain foundational competencies essential to the broader field of computer science. By following the MOOC, the student also develop skills to test, extend, and improve existing code. Understanding the performance of an algorithm is critical in computer science. Students will learn how to benchmark algorithms, which involves assessing and comparing the performance of different algorithms.

By the end of the course, students can tackle a combinatorial optimization problem using Constraint Programming, most notably by relying on MiniCP. Some knowledge gained during the course, such as modeling tips and tricks, can also be useful when using other tools, such as MiniZinc [18].

3 Table of Content

The course content is outlined in Table 1. The lectures are delivered through a series of approximately 4 videos of 15-minutes for each module, featuring a variety of speakers. The content of each module is the same as the one used at the Constraint Programming course at UCLouvain, focusing first on the key components from CP before diving into the most popular constraint from the paradigm. To gauge the students' understanding of the material, multiple-choice questions (MCQs) related to the lecture content are included. Additionally, programming assignments serve both as a practical application and an integral component of the students' final grade in the MOOC. They challenge students to implement filtering algorithms or models for optimization problems.

Course Module	Lecture	Exercises
Introduction	Applications of constraint programming in routing and scheduling. Presentation of CP as a declarative paradigm and implementation details for a N-Queens model.	Model for a graph coloring problem.
MiniCP Solver [16]	Key components of a CP solver: domain implementation for Integer Variables, interfaces for variables and constraints, fixpoint algorithm, DFS and state management through trailing.	Additional constructor for Integer Variables, a domain iterator and the Maximum constraint.
Sum and Element Constraint	Domain and bound consistency [2], Sum and Element [32] constraints, reified constraints, Quadratic Assignment Problem and Stable Matching Problem.	Several propagators for the Element Constraint and a Stable Matching implementation.
Table Constraint	Usage of the Table constraint, usage of bitsets, naive Table constraint implementation (STR) [14] and the Compact-Table constraint [4].	Compact-Table algorithm and use it to model the Eternity problem [23].
All Different constraint	Forward checking for All Different constraint, Regin's algorithm [21] for domain consistent constraint.	All Different with forward checking and Regin's algorithm, and compare the two on the N-Queens model.
Successor Models for Traveling Salesman and Vehicle Routing Problems, Large Neighborhood Search	Circuit constraint [19], its usage for the Traveling Salesman Problem (TSP) and Vehicle Routing Problem (VRP), Large Neighborhood Search (LNS) [25].	Circuit constraint [19], a custom search for an existing TSP model, tune parameters for LNS, transform a TSP model into a VRP model.
Cumulative scheduling	Time-Tabling filtering [10], LNS in scheduling, modeling producer-consumer [27] and packing problems with cumulative [28].	Cumulative decomposition, Time-Tabling filtering, modeling the Resource-Constrained Project Scheduling Problem (RCPSP).
Disjunctive scheduling	JobShop problem, Disjunctive constraint [1], theta-tree datastructure.	Modeling the JobShop problem, branching over the precedences for the JobShop [12], Detectable Precedence and Not-First/Not-Last filtering [33].
Black-Box Search [15, 17, 20, 7, 9, 13, 3]	First fail principle, impact, activity, conflict based and discrepancy search.	Last Conflict, Conflict Ordering and Limited Discrepancy Search.
Modeling	Bin-Packing, Symmetry breaking, Steel Mill Slab Problem [8, 31, 24].	Or with Watched Literals [11] and IsOr constraint, modeling the Steel Mill Slab Problem and apply symmetry breaking on it.

Table 1 Course Modules, Lectures, and Exercises

4 Programming Exercises and grading

The material covered in the course, presented in section 3 is fairly dense, especially for the programming exercises part. Automation has been a critical component in streamlining the exercises for both students and instructors. The establishment of a student project, essentially a MiniCP [16] solver template with some elements left to be implemented, is performed automatically through a grading platform - Inginious [6]. This step creates a git fork of the template, belonging to the teaching team, to which the student is added as collaborator. The template offers students a semi-completed CP solver, with sections primarily related to constraints and models left to be filled during the programming exercises. To ensure students can focus more on the task at hand rather than minor technicalities, they are given a part of a functional implementation along with operational examples. For instance, the first module introduces an N-Queen model as an example, serving as a guide for students in writing their Graph Coloring model. Moreover, each programming exercise proposes to fill gaps within missing implementation rather than create a new file from scratch. Students still have the possibility to create new separated files if they wish, but this format enables them to concentrate on essential aspects and draw from high-quality examples for inspiration.

As an example, to implement the `AllDifferent` constraint [21], the students are given explanations about the filtering, consisting of 4 steps, during the lecture. When presented, each step includes an example the state of the datastructures used within the constraint and the domain of the variables during the step. The same running examples are also given as unit tests, letting the students easily compare their implementation with the behavior presented in the videos. 2 out of the 4 steps are already implemented, the students thus need to implement the missing half and connect the 4 steps within their filtering algorithm. Particular algorithms needed for the filtering presented in the course, such as the computation of a bipartite matching for this constraint, are mostly assumed to be known and are given to the students. They can treat those components as black boxes and focus on the particular features composing the propagators.

Once a student completes a programming exercise, the corresponding unit tests can be initiated. These tests highlight potential mistakes in the students' implementation, and generate the assignment grades, which students can access locally. To share their grades with the instructional team, students can commit and push their work to their individual repositories. The grading platform then executes the repository tests in a secure manner to determine the assignment grade. The entire grading process is fully automated, allowing students to evaluate the performance of their implementations both on their own machines and the grading platform through unit testing. From an instructor's standpoint, they can easily and continuously collect the grades of all students in a CSV format with a simple button press.

Regarding the unit tests crafted by the teaching team, they are split over four main categories for each implementation to fill:

Small tests are designed to cover a minimal number of variables for a given constraint, thus enabling students to readily comprehend the test and potentially use it for quick debugging.

Common mistakes tests are largely based on common errors observed in previous years. While other tests may identify these errors, these tests present the mistakes in a more comprehensible manner for the students.

Runtime tests are crucial for ensuring that students utilize the proposed specific data structures and methods for implementing incremental filtering. These tests involve larger variable domains, to ascertain that students don't iterate over entire domains but rather use smarter strategies like residues. If a student fails such a test, an informative message is generated, indicating to the student that its code takes more time than expected, and guiding to the suggested optimization in the assignment statements.

Search tests apply the implementations to more complex scenarios. While the preceding three categories offer substantial confidence in students' implementations, they don't cover all possible cases. This category attempts to bridge that gap by creating a model incorporating several variables, with only the constraint under test being added to these variables. A depth-first search (DFS) is then executed, examining all solutions to this artificial model. Each solution is checked for compliance with the constraint. In certain instances, the number of nodes explored, inconsistencies detected, or number of solutions found over the search space is also assessed.

It's worth noting that the provided unit tests might not be enough to catch every potential typo written by the students - "*Program testing can be used to show the presence of bugs, but never to show their absence*" [5]. The objective of these tests extends beyond merely achieving robust code coverage, which can be readily obtained via the last test category. The focus also lies on creating understandable examples. The importance of the final two test categories can't be overstated. When students employ the constraints to solve a problem in a practical sense, their model implementation might falter. In such situations, to be as certain as possible that the error resides in the model's composition and not in the implementation of the constraints, students must have strong confidence in their propagation algorithms. These types of tests reduce the chances of students tracing back to a potential error written in a previous module - as such errors are likely to be picked up by the unit tests. Although runtime tests might not be present for all programming exercises, search tests consistently serve as guides for students.

With regard to the common mistakes tests, a notable example is associated with the Circuit Constraint implementation [19]. The algorithm for this, partially presented in Algorithm 1 and given to the students, has expected and incorrect Java translations displayed in Listing 1 and Listing 2, respectively. The error in the incorrect solution lies in the modification of the references to the reversible integers—Java objects—rather than updating the stored values through a `setValue()` method call. This mistake causes failures when the search backtracks and explores the remaining search space because the reversible integers are not properly set up.

When students' codes failed the unit test designed because they translated the code as in Listing 2, they were alerted to the incorrectness of their implementation, but the reason for the failure was neither clear nor easily explained by the test. In a traditional classroom setting, puzzled students would seek guidance from a teaching assistant who would identify the mistake and guide them toward the correct implementation. However, in the context of a MOOC, this issue is addressed through an integrated unit test—shown in Listing 3—which ensures the object references are unique.

Such tests are not typically found in other CP solvers but are invaluable in these instructional situations. Before incorporating this test into the course, students who meticulously translated the proposed pseudocode into their Java implementation would fail the unit tests for this constraint annually, without comprehending why. With the integration of this specific test, questions concerning this issue have ceased, enabling both teaching assistants and students to concentrate on the course content rather than on language-specific programming nuances.

Algorithm 1 Circuit propagation - beginning of the algorithm

Data: *dest*, *orig*: arrays of reversible integers storing the destination and origin of partial path through each Integer Variable x_i, respectively

Input : Integer Variable x_i that has become fixed

$j \leftarrow min(D(x_i))$;
$dest[orig[i]] \leftarrow dest[j]$;
...

Listing 1 Expected solution for Algorithm 1

```java
private void fix(int i) {
  int j = x[i].min();
  int origi = orig[i].value();
  int destj = dest[j].value();
  dest[origi].setValue(destj);
  ...
}
```

Listing 2 Wrong implementation of Algorithm 1

```java
private void fix(int i) {
  int j = x[i].min();
  int origi = orig[i].value();
  dest[origi] = dest[j];
  ...
}
```

Listing 3 An explainable test covering the error from Listing 2

```java
for (int i = 0; i < x.length; i++) {
  for (int j = i+1; j < x.length; j++) {
    assertNotSame(circuit.dest[i], circuit.dest[j], "Use dest[i].
        setValue(...) to update reversible objects, not dest[i] = ...");
  }
}
```

Finally, when the exercises statements and the error messages from the unit tests are not enough to debug the students' code, the students can resort to a discussion forum. Each question on it can be seen and answered to by both the teaching team and other students.

The scores obtained from the unit tests contribute to the final grade for the course. Each of the 10 modules contributes 2 points to the final grade, which is scored out of 20. Within each module, MCQs award 0.5 points, while the remaining 1.5 points come from the programming exercises. Students are permitted an unlimited number of submissions for the programming tasks. However, to discourage brute-forcing the answers, submissions for the MCQs are limited to two per hour.

5 Analytics

Out of the 515 students who enrolled in the course, only 110 attempted the exercises, with 70 of them successfully passing the course by achieving the minimum passing grade of 12/20 on their assignments. One way to gauge student engagement with the exercises is by analyzing the number of commits they made during the course. On average, the 110 students each made about 14.2 commits (median 12, standard deviation 11.5), as depicted in Figure 1. However, since the amount of work each commit represents is indefinite, we can also consider the total number of lines modified in MiniCP as another metric. This is defined as the sum of all lines altered in each commit, which is not synonymous with the net change on MiniCP. When we exclude four outliers who modified more than 3000 lines, we find that an average student changed about 799 lines (median 670; standard deviation 620). This distribution is illustrated in Figure 2 which, similar to the previous figure, displays a typical bell curve with an additional peak around zero.

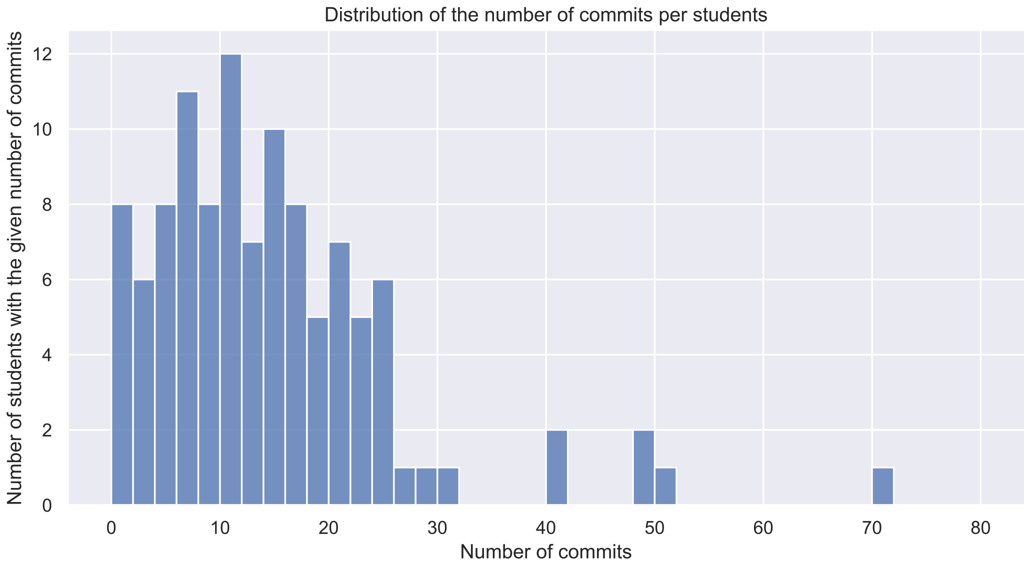

Figure 1 Distribution of the number of commits made by each student during the whole course. Each bin has a width of size 2 (0-1, 1-2, 3-4, ... are grouped together for readability).

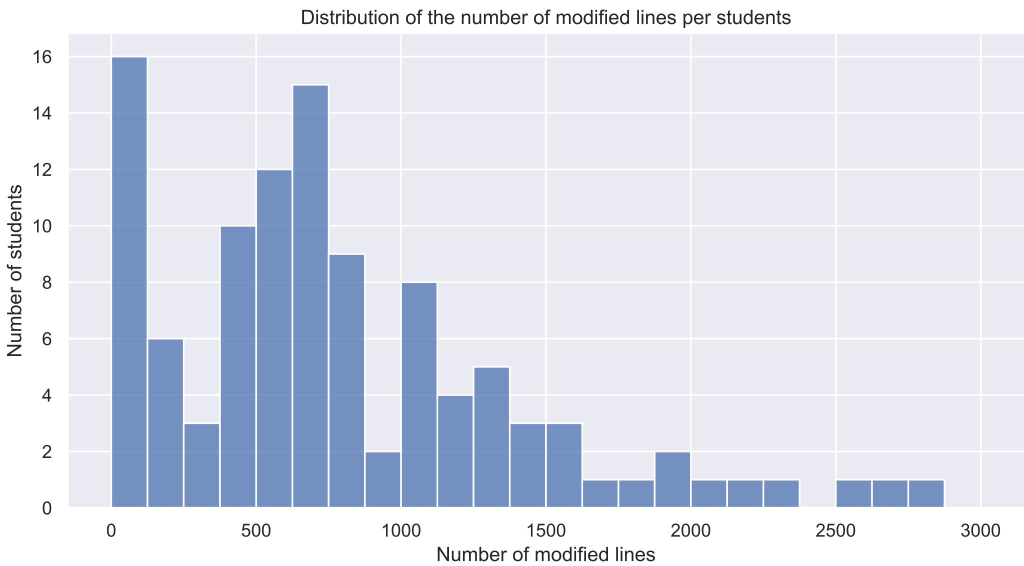

Figure 2 Distribution of the number of lines modified by each student during the whole course. Each bin has a width of size 250.

We can also analyze the activity of the students per week. Figure 3 shows the number of active student (students that made at least a commit that week) each week. The number of students attempting exercices is around 30 each week (a bit more than a fourth of the total number of students who attempted the exercices), with a notable peak in week 8 and in the last weeks of the course. This can also be seen if we look at the number of commits per day and per students, in Figure 4. Week 8 corresponds to the beginning of the Easter holidays in Belgium (where a large number of students were following the course).

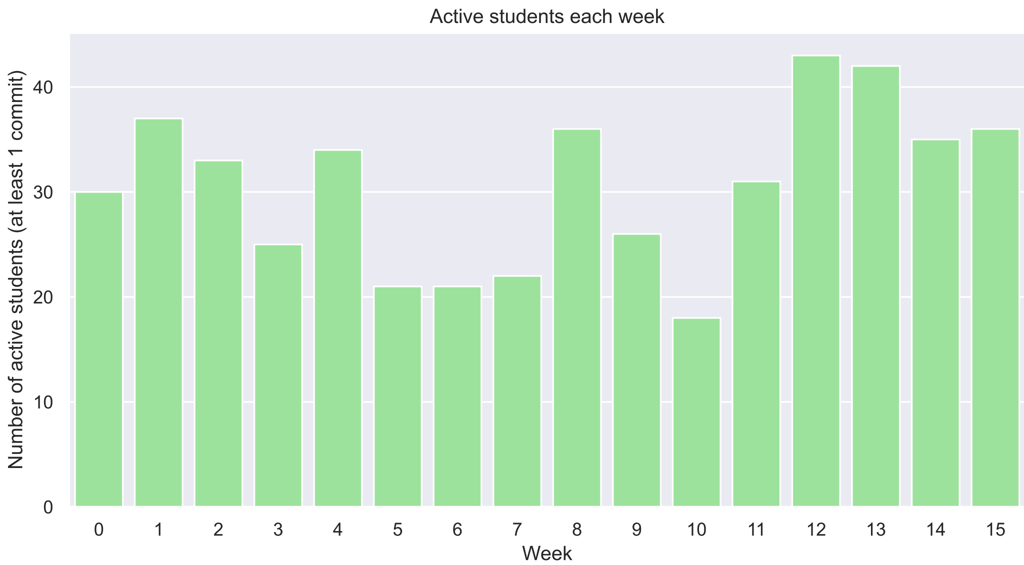

Figure 3 Number of active students per week. Active students are those who made at least one commit during a given week.

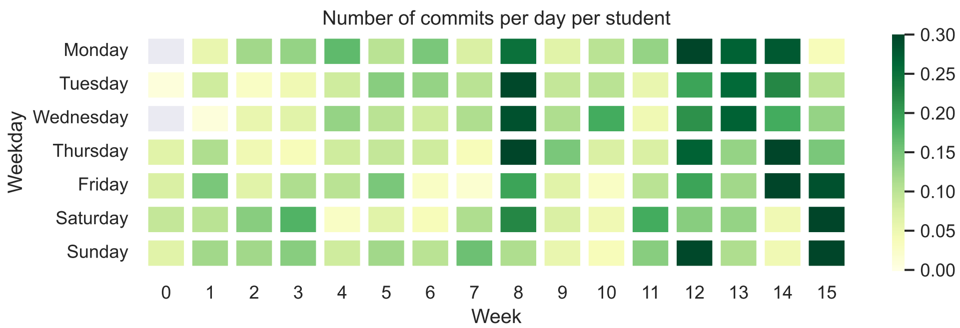

Figure 4 Average number of commits per day per student.

Apart from this, there are no visible patterns. Figure 5 shows the number of "active weeks" (weeks where the students made at least one commit) per students. We see that most students engage multiple times with the exercises, but sometimes in pretty wide interval between two commits (the median "active weeks" being 4).

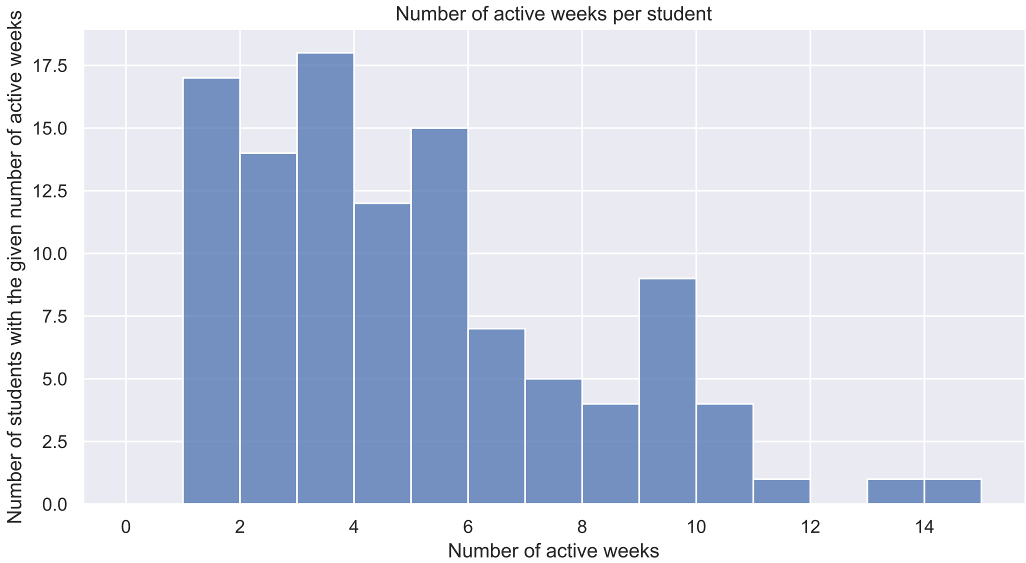

Figure 5 Number of "active weeks" per students. An active week is a week where the student made at least a commit.

It's crucial to mention that while the graphs in this section provide valuable insights into students' commit activities on their repositories, they don't capture all interactions between students and their code. As detailed in section 4, students have the option to evaluate their work locally. Consequently, some students might finish each programming assignment on their personal computers and only make a single commit at the end of the course. This could account for the presence of certain outliers: students with notably few commits yet a large number of modified lines of code.

Finally, here are some handpicked individual feedback gathered from a survey given at the end of the course:

- "Although the programming assignments are extremely difficult, at least for me, a non-CS major guy, they are absolutely rewarding."
- "I didn't finish the assignments yet but I will finish them soon. However, I really liked what I did for the moment. I can see that I learned a lot. "
- "Many times lots of edge cases were not well explain e.g. Conflict Ordering Search did not explain the fundamental difference between Last Conflict Search. Many small details like this made the exercises unnecessarily hard. Exercises where there were many tests to do exercises step by step greatly helped understanding and made it more worthwhile. "
- "I really like the format of the course, watching videos and then doing exercises but I have the impression that sometimes information given for exercises are not enough "
- "It is very clear and easy to understand and it really trains perfectly our skills in programming in CP."
- "Some test (especially in module 6) are not enough sometimes I still struggled completing some parts of exercises because the previous part was not correct although I passed all tests."

From the feedback, it's clear that students find the exercises engaging and feel gratified upon passing all the tests. However, there seem to be instances where the instructions do not fully encompass what is required to complete the assignments. While unit tests partially address this issue, providing comprehensive information and tips for the programming assignments is something we intend to improve in the course's next iteration. Interestingly, one student's feedback revealed that they had managed to pass an earlier assignment (the Maximum constraint, in this case) with an erroneous implementation, which was only detected in subsequent modules via more unit tests. This suggests the need for more robust search tests on exercises students have passed to identify such errors more promptly. Additionally, thanks to the git system implemented in the course, we can access the student's flawed implementation and use it to derive new tests for common mistakes, a feature that future students stand to benefit from.

6 The future of the MOOC

As mentioned in the previous section, based on the feedback from the student, we can put even more effort on the programming assignments. The students find them rewarding but more exhaustive instructions as well as more robust unit tests will help them tackle the programming parts more easily.

It's also important to acknowledge that our MOOC has a specific limitation. It does not focus on developing the students' ability to translate a problem's literary description into a viable model. This means that we do not extensively cultivate problem-solving from scratch in this course, and instead, we provide significant guidance to our students. This type of independence is a skill that's thoroughly developed in [30] where the primary focus is on testing the output, i.e., the solution, rather than guiding through every step of the problem-solving process. We aim to improve this aspect in future iterations of our MOOC.

It's worth noting that our evaluation framework can be adapted to such cases. For example, given one instance to a problem, the students need to add the constraints composing the problem and find one solution. This solution can be represented as a Java class, for which a checker can be added, ensuring the correctness of the solution. This behavior is actually exploited in several assignments, such as for the Eternity problem, for which the solutions are tested. Compared to the Eternity problem, the model itself would not be imposed, only the format of the solution.

Additionally, very few visualization tool are currently given to the students. For assessing the quality of their solution on the presented problems, they are given code printing their solutions in a human readable format. A deeper understanding of the course could be obtained by improving the solution printed, instead using a visualization tool showing the domains of the variables, as well as particular representations of the problems (for instance a map showing the path taken for a TSP).

In future iterations of our course, we intend to invite more experts from our community to contribute their insights on advanced topics. We recognize that some of our students have an appetite for deeper exploration and are eager for resources that can guide them further. By engaging subject matter experts, we can provide those students with the opportunity to delve into more complex aspects of constraint programming.

7 Conclusion

This study presents our approach to teaching constraint programming via a Massive Open Online Course. The course is ambitious, guiding students through the core components of a constraint programming solver, constraint implementations, and modeling aspects, all while utilizing the MiniCP solver. A crucial element that enables tackling this comprehensive curriculum is automation; all participation in and grading of programming exercises are completely automated. By leveraging previous experiences from traditional university courses, the MOOC provides an enriched learning environment, complete with challenging programming assignments. Future versions of this course will aim to increase student engagement with the material and introduce more practical examples of constraint programming. Given the vast data generated by MOOCs, our teaching team has the unique advantage of being able to easily identify and address the areas students find most challenging.

References

1 Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*, volume 39. Springer Science & Business Media, 2001.

2 Christian Bessiere and Pascal Van Hentenryck. To be or not to be... a global constraint. In *Principles and Practice of Constraint Programming–CP 2003: 9th International Conference, CP 2003, Kinsale, Ireland, September 29–October 3, 2003. Proceedings 9*, pages 789–794. Springer, 2003.

3 Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, page 146, 2004.

4 Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In *Principles and Practice of Constraint Programming: 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings 22*, pages 207–223. Springer, 2016.

5 Technische Hogeschool Eindhoven. Onderafdeling der Wiskunde and EW Dijkstra. *Notes on structured programming.* 1969.

6 Guillaume Derval, Anthony Gego, Pierre Reinbold, Benjamin Frantzen, and Peter Van Roy. Automatic grading of programming exercises in a mooc using the inginious platform. *European Stakeholder Summit on experiences and best practices in and around MOOCs (EMOOCS'15)*, pages 86–91, 2015.

7 Jean-Guillaume Fages and Charles Prud'Homme. Making the first solution good! In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1073–1077. IEEE, 2017.

8 Antoine Gargani and Philippe Refalo. An efficient model and strategy for the steel mill slab design problem. In *Principles and Practice of Constraint Programming–CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings 13*, pages 77–89. Springer, 2007.

9 Steven Gay, Renaud Hartert, Christophe Lecoutre, and Pierre Schaus. Conflict ordering search for scheduling problems. In *Principles and Practice of Constraint Programming: 21st International Conference, CP 2015, Cork, Ireland, August 31–September 4, 2015, Proceedings 21*, pages 140–148. Springer, 2015.

10 Steven Gay, Renaud Hartert, and Pierre Schaus. Simple and scalable time-table filtering for the cumulative constraint. In *Principles and Practice of Constraint Programming: 21st International Conference, CP 2015, Cork, Ireland, August 31–September 4, 2015, Proceedings 21*, pages 149–157. Springer, 2015.

11 Ian P Gent, Chris Jefferson, and Ian Miguel. Watched literals for constraint propagation in minion. In *Principles and Practice of Constraint Programming-CP 2006: 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006. Proceedings 12*, pages 182–197. Springer, 2006.

12 Diarmuid Grimes, Emmanuel Hebrard, and Arnaud Malapert. Closing the open shop: Contradicting conventional wisdom. In *Principles and Practice of Constraint Programming-CP 2009:*

15th International Conference, CP 2009 Lisbon, Portugal, September 20-24, 2009 Proceedings 15, pages 400–408. Springer, 2009.

13 Robert M Haralick and Gordon L Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.

14 Christophe Lecoutre. Str2: optimized simple tabular reduction for table constraints. *Constraints*, 16:341–371, 2011.

15 Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. Reasoning from last conflict (s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.

16 L. Michel, P. Schaus, and P. Van Hentenryck. Minicp: a lightweight solver for constraint programming. *Mathematical Programming Computation*, 13(1):133–184, 2021. doi:10.1007/s12532-020-00190-7.

17 Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems: 9th International Conference, CPAIOR 2012, Nantes, France, May 28–June1, 2012. Proceedings 9*, pages 228–243. Springer, 2012.

18 Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *Principles and Practice of Constraint Programming–CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings 13*, pages 529–543. Springer, 2007.

19 Gilles Pesant, Michel Gendreau, Jean-Yves Potvin, and Jean-Marc Rousseau. An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transportation Science*, 32(1):12–29, 1998.

20 Philippe Refalo. Impact-based search strategies for constraint programming. In *Principles and Practice of Constraint Programming–CP 2004: 10th International Conference, CP 2004, Toronto, Canada, September 27-October 1, 2004. Proceedings 10*, pages 557–571. Springer, 2004.

21 Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI*, volume 94, pages 362–367, 1994.

22 Tejas Santanam and Pascal Van Hentenryck. Modern constraint programming education: Lessons for the future, 2023. arXiv:2306.13676.

23 Pierre Schaus and Yves Deville. Hybridization of cp and vlns for eternity ii. *Journées Francophones de Programmation par Contraintes (JFPC'08)*, 2008.

24 Pierre Schaus, Pascal Van Hentenryck, Jean-Noël Monette, Carleton Coffrin, Laurent Michel, and Yves Deville. Solving steel mill slab problems with constraint-based techniques: Cp, lns, and cbls. *Constraints*, 16:125–147, 2011.

25 Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Principles and Practice of Constraint Programming—CP98: 4th International Conference, CP98 Pisa, Italy, October 26–30, 1998 Proceedings 4*, pages 417–431. Springer, 1998.

26 Helmut Simonis. ECLiPSE ELearning Course, March 2019. [Online; accessed 29. Jun. 2023]. URL: https://www.eclipseclp.org/ELearning.

27 Helmut Simonis and Trijntje Cornelissens. Modelling producer/consumer constraints. In *Principles and Practice of Constraint Programming—CP'95: First International Conference, CP'95 Cassis, France, September 19–22, 1995 Proceedings 1*, pages 449–462. Springer, 1995.

28 Helmut Simonis and Barry O'Sullivan. Search strategies for rectangle packing. In *International Conference on Principles and Practice of Constraint Programming*, pages 52–66. Springer, 2008.

29 Peter J Stuckey. Basic modeling for discrete optimization, 2023. Coursera. URL: https://www.coursera.org/learn/basic-modeling.

30 Pascal Van Hentenryck and Carleton Coffrin. Teaching creative problem solving in a mooc. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 677–682, 2014.

31 Pascal Van Hentenryck and Laurent Michel. The steel mill slab design problem revisited. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 5th International Conference, CPAIOR 2008 Paris, France, May 20-23, 2008 Proceedings 5*, pages 377–381. Springer, 2008.

32 Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint processing in cc (fd).

33 Petr Vilím. Global constraints in scheduling. 2007.

34 Mark Wallace, Stefano Novello, and Joachim Schimpf. Eclipse: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, 1997.