EPFL

CHILI
computer-human interaction
in learning and instruction

LEARN
Center for Learning Sciences

An Engine producing Orchestration Graphs

Author:
Samuel Bélisle

Professor and supervisor:
Pierre Dillenbourg

June 6, 2025

Acknowledgements

I would like to thank Pierre Dillenbourg who presented the concept of Orchestration Graphs and the first ideas about designing an engine which would help in producing them. Then, he provided supervision and insightful ideas throughout the semester.

Special thanks to Abhinand Shibu for his high interest in my project, for useful technical advice and for boosting my determination when it was low.

I acknowledge the use of ChatGPT (GPT-4o) for providing minimal working code that I could build upon during the long research through frameworks and of GitHub Copilot for continuous minor code contributions.

Abstract

This semester project designed and implemented parts of an application that could help teachers in their lesson planning, in particular the creation of an orchestration graph. The scope of the project was to develop a prototype integrating some parts of the required elements. Hence, we created a Python engine that could manipulate and evaluate activities with respect to a simplified model of lessons, activities and pedagogical concepts. Then, we integrated this engine in an interactive local application built with PyQt and QML.

Contents

1 Introduction

Imagine a teacher with the goal to teach trigonometry to a class of students which are fourteen years old. A classical approach would be to introduce the context where trigonometry applies, tell them about the formulas and show how to use them with a few examples and then give exercises for the students to practice their application. Another approach would be to give them a fun group activity related to the topic which organises their mind for the rest of the lesson. Then, let them struggle on a difficult problem which requires trigonometrical formulas, let one student explain their situation to the rest of the class and finally tell them about the formula. We represent these two approaches in figure 1 and 2 as orchestration graphs where the horizontal axis represents time and the position on the vertical axis indicates in which modality the activity takes place.

Figure 1: The classical approach represented as an orchestration graph.

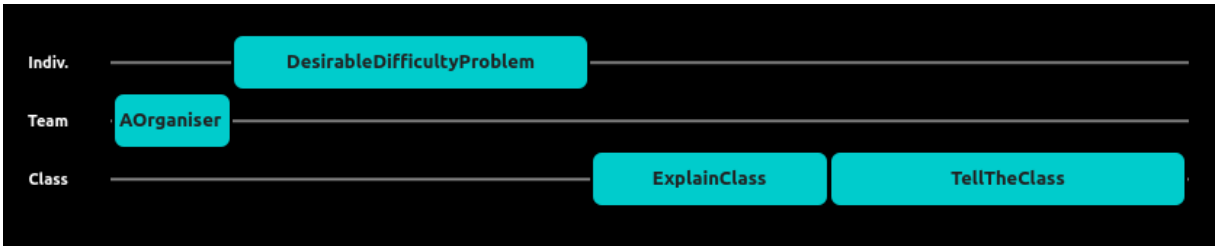

Figure 2: The other approach represented as an orchestration graph.

However, that teacher have a lot more options, and it can be challenging for a teacher to explore those possibilities and find a lesson plan that is adapted to them, to the lesson and the constraints such as the available time or the study plan.

This projects takes the first steps in the broad goal of creating a platform that could support teachers in diversifying lesson plans with evidence-based suggestions and foster collaboration between educators and learning scientists. This platform would have three main parts, it would:

1. let learning scientists and teachers build a common library of pedagogical activities and empirically validated design principles.

2. help teachers design a specific lesson plan by letting them choose activities in this library and produce an orchestration graph as the lesson plan
3. refine and adapt the chosen activities to the specifics of the lesson, teacher and students through a discussion with generative AI.

This application as a whole is too big for the scope of a Semester Project. Hence, we focus on part 2. We use a tiny simplified library based on intuitive guesses and let the interactions with an AI as an idea for later development.

2 Our model of a lesson

We model a lesson as a sequence of activities which take the class from an initial state into a state where they have acquired knowledge and enhanced their capabilities with respect to a specific subject or task. The model reduces the complex state of each student's understanding and capabilities into two numbers, p_{depth} and p_{fluency}. The value p_{depth} represents their depth of understanding and p_{fluency} represents their fluency in achieving the task, both ranging from zero to one.

Note that the model could be enhanced by defining new axis encoding more information. This is discussed in section 7.

On the one hand, each activity has an effect on those values. For instance, the "PracticeApplication" in Figure 1 (which in our example would consists of giving triangles for the students to apply trigonometrical formulas) would increase their fluency but would not deepen their understanding very much. On the other hand, this activity requires a certain depth of understanding before being applicable. To encode such behavior, we assign to each activity a pair of conditions $(c_{\text{depth}},\ c_{\text{fluency}})$ and a pair of effects $(e_{\text{depth}},\ e_{\text{fluency}})$. This means that this specific activity can be applied only for a class in a prior state $\left(p^i_{\text{depth}},\ p^i_{\text{fluency}}\right)$ satisfying $c_a \leq p_a$ for each axis a and would lead them into a new state $\left(p^{(i+1)}_{\text{depth}},\ p^{(i+1)}_{\text{fluency}}\right)$ where $p^{(i+1)}_a := p^i_a + e_a$ for each axis a. For clarity, we often represent the effect as $(+e_{\text{depth}},\ +e_{\text{fluency}})$.

The two approaches represented in Figures 1 and 2 can thus be represented with the Figures 3 and 4 respectively. Each rectangle represents one activity in the lesson which takes the students from their initial state in the bottom-left corner to their next state in the top-right corner. One could have represented each lesson as an arrow but the rectangular representation adds readability and helps to avoid confusion with the arrows for transitions defined in 2.2. For instance, the activity named *TellTheClass* has been given a condition of $(0.1, 0.6)$ and an effect of $(+0.4, +0.2)$. In Figure 3, it takes the class from a state represented by $(0.15, 0.6)$ (which satisfies the condition) to the new state $(0.55, 0.8)$.

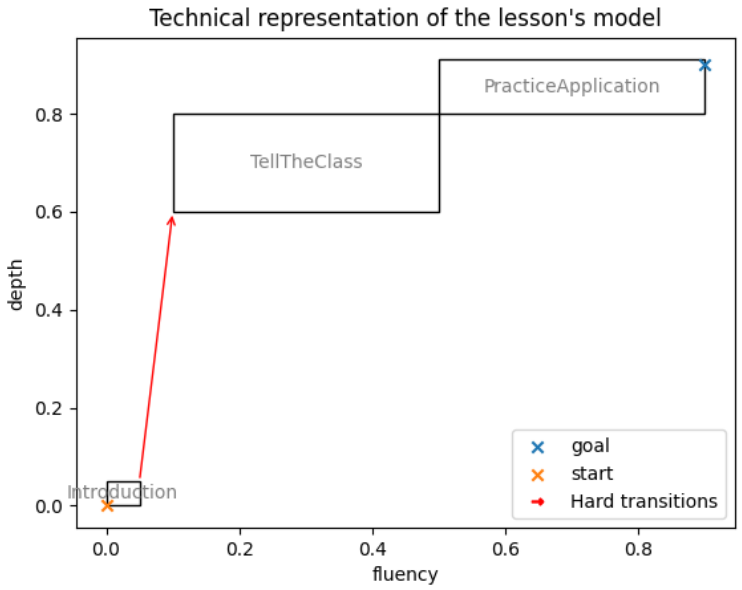

Figure 3: Our model's representation of the classical approach.

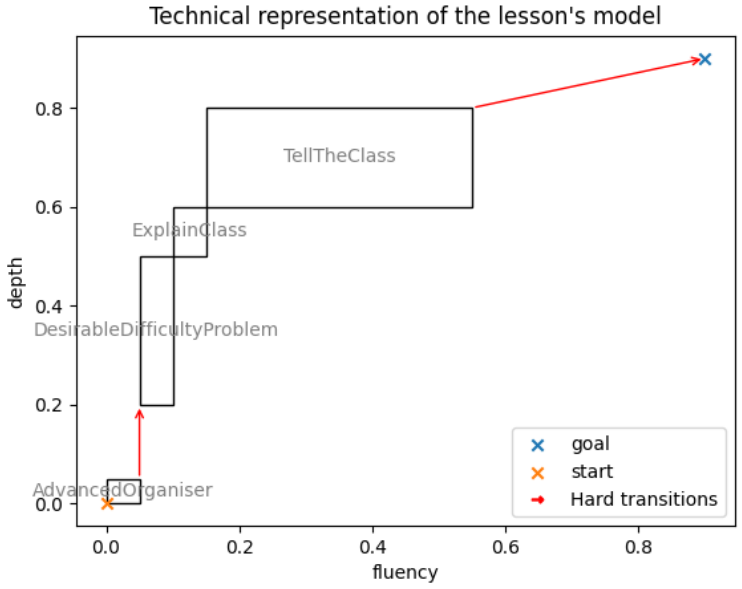

Figure 4: Our model's representation of the other approach.

The goal for the teacher is then to find a sequence of activities satisfying the conditions and reaching a desired state while respecting the class constraints (typically time). The teacher can choose an activity from a library of general activities, adapt it to the specific topic and learning goals, and then add it in the sequence.

This transition from a general activity representing high-level pedagogical concepts to a precise activity adapted to the specific topic and with determined content and duration is a key step in the elaboration of a lesson plan. We call *the precise activity with determined content* an Instantiated activity. As this process has to adapt to the specifics of the lesson (in particular the topic and learning goals but also the age and prior knowledge of the students and the preferences of the teacher) this instantiation process can't be hard-coded in our application. This would require a specifically designed large language model to interact with the teacher and help them adapt the general activity into an instantiated activity adapted to their situation. Our current implementation skips this interaction with an AI while instantiating an activity.

We also encoded in the activities a maximal number of repetitions within a lesson and the modality in which the activity takes place. The three modalities considered are *individual*, *in teams* and *in complete class* but one can extend this.

In order to have less rigid activities, we decided to let those activities define a minimal and maximal time length and adapt the effect linearly. For instance, an activity could have a minimum time of 20 minutes with minimal effect (+0.1, +0.2) and a maximal time of 40 minutes with maximal effect of (+0.15, +0.4) and then, if the teacher selects an effective time of 30 minutes, the effect of the instantiated activity would be (+0.125, +0.3), the linear interpolation of the two defined extremes. In the current version of the application, there is a default time value that is defined for each activity. The teacher can edit the time of an activity in the lesson plan by clicking on it and inserting the new value in the info panel.

2.1 Principles

The process of creating an orchestration graph for a specific lesson is more complex than what our model can describe. Moreover, this process is not linear in the sense that the teachers typically choose the activities in order of their importance or impact and not by choosing the first activities first. These two facts explain two critical decisions about the application:

1. The user should always be free to take their own decision. The engine only provides recommendations that they can ignore.
2. None of those decisions is set in stone and they should be able to reorganize their lesson freely at any time.

2.2 Transitions

These two principles have guided many decisions. For instance, this introduced the concepts of transitions and gaps. A transition originally refers to the transition between two activities in a given sequence but we gave it a more general meaning. It refers to the *distance* between a state reached by the class (either their original state or reached after an activity) and a desired state (either the lesson's goal set by the teacher or the

condition for starting the next activity). If this distance is zero or small, all is well and the transition is considered valid within our model. However, if this distance exceeds a defined threshold, then the transition is considered non-valid. We call them hard transitions. However, this is not *bad*. It follows from the principles above that the user can make insertions which creates hard transitions. The engine will notice it and make recommendations to improve the situation.

3 Installation

The engine and the application have been built in Python 3.10. The instructions to run the source code are available in the `README.md` file from this public Github repository (github.com/Katokoda/OG_QML).

The source code is available and can be run after installing some python packages. Alternatively, one can run a built version which do not require any installation (but unzipping an archive), but the textual printing might not work in that case.

4 The engine

We here present what file handle which part of the engine. Note that we skip over files and details necessary to the GUI application which will be presented in section 6.

The file `params.py` contains parameters modifiable from the user such as the plane names and the treshold mentionned in section 2.2.

The basics for manipulating the class state and the activities conditions and effects are written in the file `pValues.py`. In particular, this code handles addition of an effect on a state, measures distance and finds out where an activity will effectively start when the teacher adds it in a place where the class state does not entirely respect the activity condition. This file also defines `InterPVal` which handles the linear interpolation between the minimal and maximal effects.

Those `InterPVal`s are initialised when reading from inputs file in the `inputData` folder. This is used to define `ActivityData` defined in the file `Activity.py` which hold all necessary information corresponding to a general activity in the available library which is defined in `Library.py`.

The instantiation process is handled in `InstantiatedAct.py`. In particular, the instantiated activity has to be adjusted when it's starting state is *moved* by the insertion (or suppression) of a prior activity earlier in the lesson. This files contains minimal tests which can be run following the instructions in the `README.md` file.

The chore of the engine is written in `OrchestrationGraph.py`. This file contains extensive tests that give an insightful vision of what it handles. We detail it below.

4.1 The OrchestrationGraph class

One can initialise a lesson by linking it to a library (read from file), giving it a maximum duration (in minutes), an initial state and a goal state.

Then, one can insert an activity from the library to a given position in the current activity sequence. There is also the option to remove an instantiated activity or to move it to a new position. These operations trigger a re-structuration of all the activities. For instance, if the current lesson contains simply an instantiated activity taking the students from the initial state $(0, 0)$ to $(0.2, 0.3)$, but the user inserts in the first position an activity with condition $(0.5, 0.5)$ and effect $(0.2, 0.1)$, then the prior activity has to be moved in order to start from the state reached after this new activity, $(0.7, 0.6)$. One can see an which illustrate the need for re-structuration in the tests cases within `OrchestrationGraph.py`.

The Orchestration Graphs can be printed either as text or showing the technical representation as in figures 3 and 4. We also added to possibility to load or save the state of the lesson in a .pickle file. This functionality was actually necessary for the technical printing to work within the GUI. Read the information in `MyOGPrinter.py` for more details.

Finally, we offer the function `setGapFocus(idx)` which computes recommendations for a specific transition. Then, one can use the function `autoAddFromSelectedGap()` to add the recommended activity at this position. This should be used only if the selected transition is hard. One can also use `autoAdd()` to select the *hardest* transition and then add the recommended activity there. These recommendations occur according to section 4.2.

4.2 Recommendations

Note: When speaking about distance we choose to refer to the euclidean norm (ℓ^2) of the vectors. We could choose another metric such as the Taxicab norm (ℓ^1) or define our own metric which gives non-uniform weights to the axis. This definition occurs in `pValues.py`

The approach we started with for recommending an activity is to answer this question: What activity is the best when starting from a state s_{start} to a goal s_{goal}?
This formulation of the question means that repeating this question with all hard transition until there are no more of them is equivalent to applying a greedy algorithm to create the lesson plan.

This question is not trivial and we explored a few heuristics that aim to answer it. Those heuristics compute a score for each activities with respect to a given starting state and goal.

The first naive idea was simply to give a better score to applicable activities that "teach the most", i.e. which have the biggest effect. However, this has bad properties. For

instance, for a transition from $(0, 0)$ to $(0.5, 0)$, an activity with effect $(0, 0.5)$ would seem useful although it does not progress at all for that specific goal. It does not "teach in the good direction".

The second heuristic first evaluates the distance between s_{start} and s_{goal} (this is the hardness of the transition). Then, for a specific activity A, it computes the distance between s_{start} and the starting state of A as well as the distance between A's reaching state and s_{goal}. The score for the activity A is the difference between the hardness of the transition and the sum of the two computed values. This score is bigger if the future transition that the insertion of A would leave are smaller, if the activity removes most of the *incomprehension*. In Figure 5 we can see an example where the insertion of the activity "PracticeMemory" (modeled as having only an effect on fluency) leaves two hard transitions. This second heuristic is named `distRemoved` in the `Efficience.py` file.

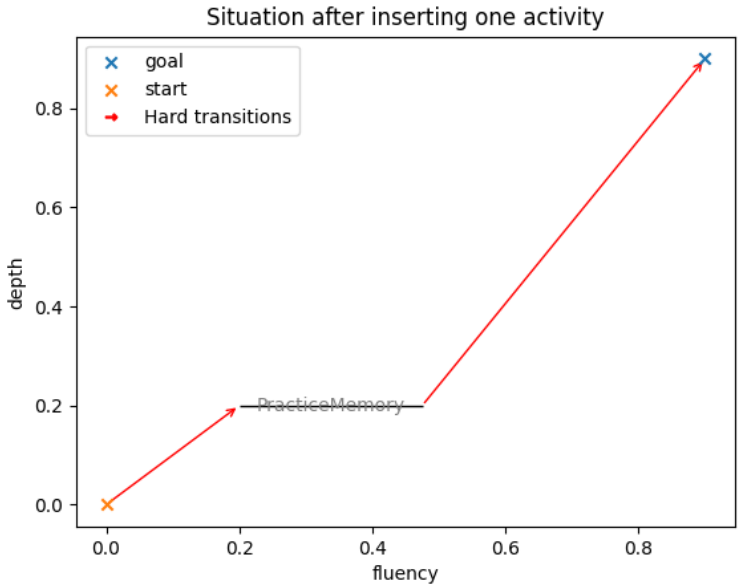

Figure 5: The two transitions left after inserting an activity.

The third heuristic computes `distRemoved` but divides the result by the time that the activity would take. Indeed, reducing the distance by a quarter using a quarter of the allowed time is better than removing a third of the distance using all the time allowed for the lesson. This heuristic is named `distRemoved_over_usedTime` in the `Efficience.py` file and is the default heuristic used by the engine. It has fairly similar results as `distRemoved` but has a better behavior especially in extreme situations such as choosing the shortest activity when they all *bridge the gap* by starting at s_{start} and reaching s_{goal}.

The last considered heuristic is named `leftTime_over_leftDist`. It is rather elegant but exhibits some rather catastrophic properties. The initial idea was to evaluate the resulting lesson more than the potential activity per say. The formula is then to

compute the total time remaining and to divide it by the total distance remaining for the lesson after the insertion of the activity. With this formula, lessons that *leave the most time for the rest of the activity* are selected first. This makes the local choice depend on global properties such as the time budget. We show some catastrophic results in figure 6, where the engine keeps recommending short non-useful activities.

This last heuristic lacks a fundamental property of the score to be positive only if it helps in reducing the total *distance* of the hard transitions. Studying examples made clear that all proposed heuristics actually lacks the converse of this property (there are situation where score is non-positive but the insertion would move activities and reduce gaps further in the lesson plan). Requiring both properties (the score should be positive if and only if it helps) inspires a new heuristic presented in section 7 as we noticed it too late it and implementing this requires a deep reorganization.

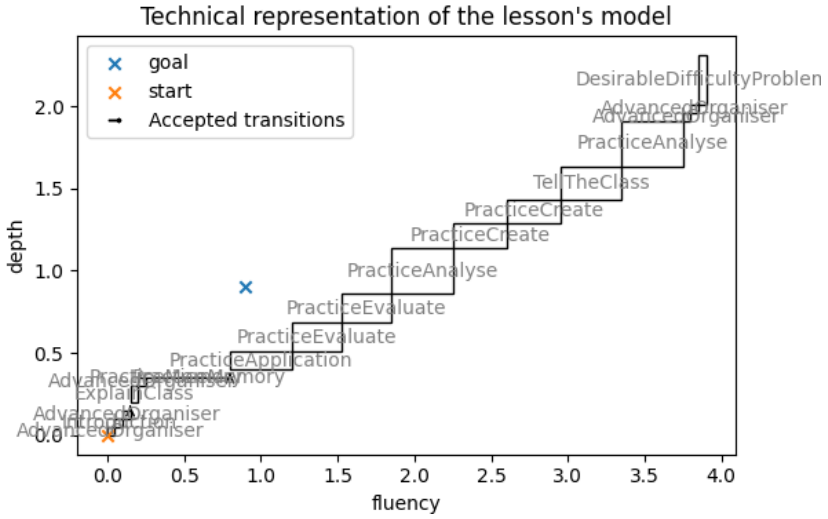

Figure 6: Following the recommendations of `leftTime_over_leftDist` yields strange results.

5 The long exploration

After having a working engine in Python (which, at the time, was really different and lacking most of the functionalities presented above), the goal was clear: Design an interactive application which lets the user use the engine intuitively. However, there where many paths to reach that goal. The main ways were to create an application running in a web browser, either communicating with a python server or with an equivalent to the Python Engine built in the client side.

Here, on a more personal tone, I present the frameworks/ideas I explored and rejected:

1. Streamlit (streamlit.io) because it seems to rigid and lacks drag and drop.

2. HTMX (htmx.org) because I managed to have drag and drop and communication with the server but not both together.
3. React (react.dev/learn) because I could not achieve what I wanted.
4. Tkinter (wiki.python.org/moin/TkInter) was kept as a fallback option in case I could not manage with PyQt and QML, as I already completed small projects with it.

Finally, I chose to combine PyQt (pypi.org/project/PyQt6/) and QML (doc.qt.io/qt-6/qmlapplications.html).

That journey was chaotic and I can't be sure that the final choice was the right one but I am really proud of the final application. Now, let's return to the presentation.

6 The application

The application uses PyQt and QML and is built over the engine presented in section 4. The `main.py` file loads the graphical interface from the QML files in the folder GUI. Then, the state of the python data-structures is read by the application through `@pyqtProperty` and the engine receives the user's input trough `@pyqtSlot`s.

The technical representation (as seen in Figure 1 for instance) used `matplotlib`. However, this packages conflict heavily with PyQt. In order to have both working at the same time, the applications saves the current state to a file and spawns a subprocess which will read it and print it. This made necessary to split the class in two halves, one holding the data and the other interacting with PyQt. This solution is the result of a long and unexpected work.

The application shows the general activities in the right panel and the user can drag them to hold them in a specific position in the orchestration graph. On can also select activities or transitions to see or update details about them. There are live information that evolve as the lesson plan changes. The buttons of the top panel call actions explained in 4.1 and are deactivated when the conditions are not met.

There are two active bugs. The first one is a minor visual glitch of the background lines that occur when hovering an activity over a selected transition. The second one is not consistent. Sometimes, one transition is blocked and no activities can be dropped there. This seems to happen when selecting-deselecting an instantiated activity and a transition alternatively in quick succession. Similarly, there are a few design choices that cause problem, for instance, the information box is not always entirely visible when hovering over one of the exclamation mark which mark hard transitions. Sadly, we had no time to fix these minor problems.

7 Limits and future development

In this section, we briefly list ideas for future development or generally ideas we had not enough time to explore.

Let's first talk about ideas that were out of scope for this project. As already explained in section 2, the communication with an AI agent during the instantiation of an activity was out of scope. Other ideas involved adding more dimension such as a specific dimension for various types of cognitive abilities, a dimension representing raw knowledge, the motivation of the students or their fatigue level. We could define a few dimensions representing different groups of students or maybe make a distinction between what they can do on their own or with external support (that would capture their Zone of Proximal Development). Finally, a required development would be to create specific interactions between the activities and base the model and the data on scientific knowledge about pedagogy.

We had three main feasible ideas of improvement that were not fully explored.
Currently, the engine only recommends to *add* activities. After exploring situations such as in Figure 6, it is clear that the engine should be able to recommend removing some activities or changing their order.
While writing the report we discovered new things about the scoring heuristics described in section 4.2. With current knowledge, we would recommend another approach which is to perform full simulation of the insertion of the activity and evaluate the resulting lesson plan. This approach would remove problems that comes from the locality of the current evaluation scheme. It would satisfy the *if and only if* property mentioned at the end of that section.
Similarly, the current scoring system feels really rigid. The score is deterministic and the *autoAdd* button always adds the best-scoring activity with the least possible flags (and chooses the first in the list in case of score equalities). Ideally, it would use the score to sort the activities and non-deterministically choose some of the best ones as a proposition to the teacher.

Finally, we list concrete implementations details that we had in our list but had no time to finish. Most visibly, we left unclear textual information in the library which are flags and recommendations for the user. As written, the goal was to have little symbols showing this visually and intuitively and a text explaining it when hovering them.
One detail that feels missing is that the mouse currently do not react to what it is hovering. It should change shape to indicate that dragging or clicking is possible, for instance.