Apprentissage de la programmation par les problèmes : génération automatique d'exercices et recommandation

Théo Barollet

▶ **To cite this version:**

Théo Barollet. Apprentissage de la programmation par les problèmes : génération automatique d'exercices et recommandation. Intelligence artificielle [cs.AI]. Université Grenoble Alpes [2020-..], 2023. Français. NNT : 2023GRALM067 . tel-04586998

HAL Id: tel-04586998
https://theses.hal.science/tel-04586998

Submitted on 24 May 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Laboratoire d'Informatique de Grenoble

Apprentissage de la programmation par les problèmes: génération automatique d'exercices et recommandation

- Learning problem-based programming: automatic generation of exercises and recommendations.

Présentée par :

Théo BAROLLET

Direction de thèse :

Fabrice RASTELLO DIRECTEUR DE RECHERCHE, INRIA CENTRE GRENOBLE-RHONE-ALPES	Directeur de thèse
Florent Bouchez-Tichadou UGA	Co-encadrante de thèse

Rapporteurs :

DOMINIQUE COLNET
PROFESSEUR DES UNIVERSITES, UNIVERSITE DE LORRAINE
THIBAULT CARRON
MAITRE DE CONFERENCES HDR, SORBONNE UNIVERSITE

Thèse soutenue publiquement le **24 novembre 2023**, devant le jury composé de :

FABRICE RASTELLO DIRECTEUR DE RECHERCHE, INRIA CENTRE GRENOBLE-RHONE-ALPES	Directeur de thèse
DOMINIQUE COLNET PROFESSEUR DES UNIVERSITES, UNIVERSITE DE LORRAINE	Rapporteur
THIBAULT CARRON MAITRE DE CONFERENCES HDR, SORBONNE UNIVERSITE	Rapporteur
HAMID CHAACHOUA PROFESSEUR DES UNIVERSITES, UNIVERSITE GRENOBLE ALPES	Président
FRANÇOIS BOUCHET MAITRE DE CONFERENCES, SORBONNE UNIVERSITE	Examinateur

Invités :

FLORENT BOUCHEZ-TICHADOU
MAITRE DE CONFERENCES, UNIVERSITE GRENOBLE ALPES

Résumé

L'enseignement de l'informatique est un sujet important puisque de plus en plus de personnes ont besoin de compétences en programmation ou en informatique dans leur travail. Quand on programme, il est inévitable que des bugs apparaissent. Le débogage est une des tâches principales de la programmation et pourtant est peu enseigné dans les cursus d'informatique. On apprend souvent à déboguer "sur le tas" et devient plus efficace au fur et à mesure que l'expérience s'accumule.

Dans cette thèse, nous essayons d'enseigner le débogage et de fournir des exercices d'entrainement pour les programmeurs novices. Il existe déjà plusieurs Systèmes de Tutorat Intelligents (ITS) sur des sujet plus restreints comme l'algèbre au collège. Ces systèmes hébergent une base de donnée d'exercices et fournissent une recommandation sur le prochain exercice à effectuer. Nous verrons qu'il est probablement trop difficile d'adapter ces systèmes de recommandation à une base de donnée d'exercices de débogage étant donnée la complexité de l'activité de débogage. Nous étudierons brièvement la génération automatique d'exercices de débogage. Cela montrera des résultats loin de nos attentes et demandera un changement d'approche vis à vis de notre enseignement du débogage.

Nous présentons Agdbentures, un jeu d'apprentissage du débogage avec une difficulté croissante et des exercices de débogagnes manullement conçus. Chaque exercice prend la forme d'un mini-jeu qu'il est impossible de gagner tant qu'il existe des bugs dans le code source. Beaucoup d'efforts ont été fait, à travers l'ajout d'éléments ludiques, pour garder l'engagement de l'apprenant. Les premiers résultats expérimentaux avec des étudiants de L2 sont encourageants puisqu'ils ont en pratique appréciés chercher et résoudre les bugs dans les exercices d'Agdbentures.

Pendant le développement d'Agdbentures, nous avons réalisé que la supervision d'un programme externe est une tâche commune à beaucoup d'applications de l'enseignement de l'informatique, par exemple les outils de visualisation du déroulement d'un programme. Nous avons développé *Easytracker*, une bibliothèque de supervision du déroulement d'un programme avec une API indépendante du language de programmation utilisé. *Easytracker* est disponible pour le language C (ou n'importe quel language supporté par GDB) et Python. *Easytracker* est utilisé comme une bibliothèque pour abstraire beaucoup de l'ingénierie d'Agdbentures concernant la supervision d'un programme. Nous avons développé plusieurs autres outils basés sur *Easytracker* que nous utilisons en cours pour montrer le déroulement d'un programme. Cela démontre les avantages d'*Easytracker* quant au développement de tels outils en permettant d'allouer tous les efforts dans la visualisation directement d'une représentation de la mémoire d'un ordinateur.

Abstract

Teaching computer science is an important issue as more and more people need to use programming or computer science skills in their jobs. When programming, it is inevitable that some bugs occur. Debugging is a main task in programming and is seldom taught in computer science curricula. We usually learn to debug the hard way and become more efficient after gaining more experience.

In this thesis, we will try to teach debugging and provide training exercises for novice programmers. There are already several Intelligent Tutoring Systems (ITS) on simpler topics like secondary school algebra. They host an exercise database and provide a recommendation on the next exercise to do. We will see that it might be too difficult to adapt these recommendation systems to a database of debugging exercises given the complexity of the debugging activity. We will briefly study the automatic generation of debugging exercises. This will prove to have results far from the expectation and will require a change in the approach to teaching debugging.

We present Agdbentures, a debug practicing game that features increasing difficulty and carefully designed debugging exercises. Each exercise takes the form of a small mini-game that is impossible to win without fixing some bugs in the source code. Many efforts were made to increase the engagement of learners through the addition of ludic aspects to Agdbentures. The first experimental results Agdbentures with CS1 students are encouraging as they indeed enjoyed very much searching and fixing the bugs in Agdbentures exercises.

While developing Agdbentures, we realized that the monitoring of an exterior program is a common task in many computer science teaching applications, for example, all visualization teaching tools. We developed *Easytracker*, a framework to monitor program execution with a language-agnostic API for C (or any GDB-supported language) and Python. *Easytracker* is used as a library to abstract many of Agdbentures engineering regarding monitoring. We developed several other tools based on *Easytracker* that we use in lectures to show program execution. This demonstrates the benefits of *Easytracker* for writing such tools because all the effort can be put into visualization from a memory model.

Remerciements

Je tiens à remercier Fabrice Rastello et Florent Bouchez-Tichadou, mes deux directeurs de thèse qui m'ont accompagné pendant des stages de M1 et M2 et enfin pendant un peu plus de trois ans pour la réalisation de ce doctorat. Grâce à leur aide et leur temps, j'ai pu changer de direction de recherche lorsque c'était nécessaire. Ils m'ont aussi permis de travailler plus longtemps que prévu sur ma thèse, ce temps fut précieux et a permis l'aboutissement des deux principales contributions de ce doctorat.

Je remercie Thibaut Carron et Dominique Colnet, les rapporteurs de ce manuscrit pour avoir pris de leur temps pour critiquer mon travail en détails. Merci aux autres membres du jury, Hamid Chaachoua et François Bouchet pour leurs retours et les discussions sur mon travail.

Je remercie aussi mes collègues de l'équipe CORSE : Chukri, Nicolas T et Nicolas D, Lucie, Valentin, Guillaume, Auguste, Hugo, Manu, Christophe et Imma pour les discussions cette fois sur autre chose que mon travail. Merci une deuxième fois à Manu et Christophe et une première fois à François pour les collaborations sur *Easytracker*. Merci aussi une deuxième fois à Imma pour son soutien autant administratif que ses conseils avisés pendant la fin de la thèse.

Merci à Steve Kommrush avec qui j'ai travaillé au tout début de mon doctorat, qui a permis le développement des deux premiers chapitres de ce manuscrit et qui m'a transmis une partie de son expertise en apprentissage machine.

Merci à mes deux parents pour m'avoir transmis le goût des sciences et un minimum de conscience de travail.

Et enfin un merci juste pour Alexandra pour m'avoir soutenu chaque jour depuis quelques années maintenant et qui je l'espère trouvera l'épanouissement professionnel qu'elle cherche et qu'elle sera fière des projets personnels qu'elle mène.

Contents

Introduction

Computer science teaching gains more and more interest because of the increasing usage of computers in different fields. Different profiles may have to learn computer science or programming. They can be students in computer science majors or people newly needing programming skills in their everyday work for example for data processing and statistics.

Programming naturally introduces bugs. Debugging is a complex process but rarely taught as is in the many technical courses of computer science curricula. Students usually learn to debug in autonomy or in an indirect way during lab sessions where a teacher can often help to find bugs. With experience in programming, it is easier for experts to take a step back and be more methodical when searching for bugs [75], but for novices, having strong programming skills does not necessarily imply good debugging skills [76], making it necessary to teach debugging on its own and adapt the exercises to novice learners.

Many computer science curricula cannot add specific timeslots to learn to debug; hence there is a need for systems that allow students to learn and practice debugging on their own. Being able to learn debugging in autonomy can also be a nice addition for people not involved in a computer science curriculum but need programming skills. However, struggling to find a bug may be discouraging. In particular, if there is no one to help or external incentive. To mitigate this effect, it is crucial that the slope of difficulty in exercises rises slowly.

In order to keep the students engaged, it is also possible to add some ludic aspects to the debugging exercises, similar to what exists for teaching programming [70, 71, 72]. Ludic aspects are already used several times to increase engagement in the learning of various concepts [65, 66, 67]. In the case of debugging learning, this has a bonus side effect of mitigating the frustration that can arise from not being able to solve a bug.

In this thesis, we will explore the applications of statistical methods and machine learning to the development of such a debug practicing system. There are already several intelligent tutoring systems that can recommend or generate some exercises in other fields like secondary school algebra [1, 2]. We will try to evaluate the methods they use and some original methods for debugging exercise recommendations and generation.

It may already be too hard to have such a system that can do both generation and recommendation, so it might not be possible to even add some ludic aspects to it. In this thesis, we will also study the addition of ludic aspects to a pedagogical sequence of debugging exercises independently of these statistical methods. We were able to test this pedagogical sequence with CS1 students and observe differences with regular debugging teaching.

The result is close to what we could call a video game. This raised interesting questions in terms of engineering for pedagogical resources in computer science teaching in general (not only debugging). This will also be discussed in this thesis and a framework is introduced to help the development of pedagogical software related to computer science.

Chapter 1

Exercises recommendation

In our context of learning debugging by solving exercises, we need to recommend an appropriate exercise to each student. In this chapter, we will explore different methods to recommend debugging exercises.

However, to our knowledge, there is no debugging exercise database. Whereas there are already some in many other disciplines: algebra [3], grammar [4] or abstract games like chess, [1] for example. Building such a database for debugging exercises will be discussed in Chapter 2.

We take inspiration from two main fields for recommendations: the already existing recommender systems literature [6] and rating systems used in sports and games based on the ELO rating system [12] (this will be detailed in Section 1.1).

These methods have not been designed for learning environments. We will use exercise databases from various learning contexts and see how the methods behave and how we can improve them. We use different disciplines to evaluate the methods so the results should generalize to any exercise database from various disciplines (including debugging).

The main difference between generic recommendation and a learning environment is that we should find a way to model learners' current knowledge. This task is referred to as Knowledge Tracing [25] (abbreviated KT) and already documented (again detailed in Section 1.1). Depending on the recommending method, this may need expert input if applied to a new field. The expert need to define different knowledge components to acquire a given skill. These components takes the form of a tree, for example fraction computation is a knowledge component of secondary school algebra and putting fractions on the same denominator is a knowledge component of fraction computations. Each exercise would be related to one or more knowledge component and this will be an input of the recommending system.

Recommending problem definition In this section, we define the Recommendation problem we are trying to solve.

We have a set of students S and a set of exercises E (respectively called users and items in generic recommendation literature). We also have a set of past observations: we know for any students the exercises they tried and if they succeeded or failed. This is a set of triples (s_i, e_j, r) in $S \times E \times 0, 1$ where $r = 1$ means that student s_i did the exercise s_j correctly and zero incorrectly. This can be any metric instead of pure correctness: number of tries, time to solve the exercise, or we can use a combination. A fourth value can be added to this history of past

1. https://lichess.org/training

observations which is the time at which the student attempted the exercise.

The problem we are trying to solve is to find a next good exercise (or several exercises) that will be interesting for the student to try, knowing their result to the past exercises. Observe that even if we can predict if the student will succeed or not in a new exercise, knowing if it is interesting is currently a really hard problem that is still open and studied in learning theory in psychology. This will be developed in Section 1.4.

For now, we will see the recommendation problem as a predicting problem. We try to predict if a given student s_i will succeed in an exercise e_j. This still sounds not easy but is in the scope of knowledge tracing.

Many recommending systems are mature and used in online shopping and multimedia recommendation. These systems need to predict users' tastes in given items (movies, music, items in an online shop). These tastes are quite static in time and, it is much easier to derive a recommendation from a taste prediction. We can simply recommend most liked new item or the N-most liked ones based on the taste prediction. In our learning environment context, the recommendation problem is much more difficult: student proficiency is supposed to evolve over time and is not static anymore. Moreover we already explained that deriving a recommendation from knowing if a student will succeed or not in a given exercise is not trivial.

1.1 Background

In this section, we will introduce the ELO rating system and some techniques issued from recommender systems literature mostly from a predictive point of view only. We will nonetheless make learning-relative comments when needed. Modeling of learners' knowledge will be discussed in a subsection dedicated to knowledge tracing.

1.1.1 ELO Rating and Its Derivatives

The ELO rating system was first introduced to measure chess players' performance in the US in the 1960s [12]. It is actually a basic predictive model for a match outcome. It assumes that player performances follow a logistic distribution [12].

We can compute the expected probability of winning for each player given the rating difference. Let R_A and R_B be the respective rating of player A and B. Their respective probability of winning are :

$$E_A = \frac{Q_A}{Q_A + Q_B} \text{ and } E_B = \frac{Q_B}{Q_A + Q_B}$$

where

$$Q_A = 10^{\frac{R_A}{N}} \text{ and } Q_B = 10^{\frac{R_B}{N}}$$

N is a scale parameter such that for each difference of N rating points the expected score is magnified ten times compared to the opponent expected score. For example, this parameter in the original Elo implementation in the chess federation is 400.

After observing the outcome, we compare it to the expected probability of victory and change the players rating proportionally to the surprising of the result.

$$R'_A = R_A + K.(S_A - E_A)$$

where S_A is the observed score of player A and K is another system parameter controlling the amount of rating points exchanged after each game.

Extensions Several extensions now exist for ELO ratings. We will see two of them that may be of interest for educational systems. The first one is the Glicko system [13]. It adds a variance parameter in one's rating that is also updated after a match outcome. It models the regularity of performance that can vary between players. The Glicko rating also adds a confidence parameter. After each match, we gain confidence in the computed rating if the result is expected. Moreover, at regular intervals, the confidence in every player's ratings is decreased. In chess this can be useful for several reasons. As an example a player could train a lot without playing ranked games so after that their rating is not accurate anymore. The reverse can also be true that without practice for a long time, performance may decrease. This can be a basic modeling of forgetting (more elaborated modeling of forgetting is presented in Subsection 1.1.3).

The Whole History Rating (WHR) [14] is a completely different rating system. The idea is quite simple: the rating of players is not updated incrementally after each game they play but the whole system recomputes all ratings for everyone after each game. It is actually tractable in terms of computing cost for big databases (the article reports about 4 minutes for 750K games). In current implementations, the model simply features player ratings and also gives a victory probability in a given match. However, we could imagine more complex modeling to be applied in educational contexts.

Application to educational systems Even though most educational systems feature more complex knowledge models. We can still easily imagine a direct use of ELO ratings or some derivatives in educational contexts. Players would be separated between learners and exercises. Exercise difficulty and student proficiency would be their ratings and the victory probability would be a student successfully solving an exercise. However, such a simple model should have less prediction accuracy than regular knowledge tracing models. Moreover, in their initial form, these rating models only feature one-dimensional ratings. This is not a problem in games where all skills related to this game are usually needed when playing. This may easily not be the case in an educational context where some exercises can focus on specific skills.

Anyway, these rating models are easy to implement and can help to obtain some first modeling before a more complex model has enough data. This use is reported in [15] where the authors try to predict the correctness of answers in geography quizzes. The authors indeed report worse prediction accuracy than knowledge tracing models but the basic ELO rating is still a good starting point. They believe that the increase of ELO rating complexity might benefit the prediction but may not be worthwhile regarding the implementation complexity as the main advantage of ELO rating is its simple implementation.

1.1.2 Collaborative Filtering

In this section, we describe current collaborative filtering methods from a multimedia or online shopping point of view. These are the topics where collaborative filtering is the most used and the literature usually refers to these use cases. We will see in Subsection 1.1.3 specific applications to knowledge tracing.

We consider a set U of N *users*, a set I of M *items*, and a set of *ratings* R. These sets are usually given as *records* $(u, i, r_{u,i})$, representing how much $(r_{u,i})$ a given user u likes item i. The main assumption of collaborative filtering is that if we observe two similar users they should also be similar for unknown ratings, at least better than using a random prediction. We make predictions not only based on users' past observations but on all users past observations so we can find similar users.

This research field drew a lot of attention with the *Netflix prize competition* in 2006 [7]. The company opened some of the statistics they gathered over the years and provided a one million dollar price to teams that will improve their current proprietary collaborative filtering algorithm by 10%.

Datasets preprocessing It is sometimes necessary or advantageous to perform pre-processing of the data before trying to extract information. For example, it was possible to improve the classification error in the MNIST database (a common multi-class classifier dataset for handwritten digits) from 12% to 8%, keeping the same linear classifier only by using deskewing pre-processing [8]. This is also the case in a knowledge tracing context. The ASSISTment system that provides an algebra exercises database and some recommendation based on knowledge tracing released a public dataset with some student statistics that are publicly available. Many corrections to the ASSISTment dataset [16] are proposed by Xiong et al. [26]. They are now included in the public dataset that we use in Section 1.3. We also propose some more pre-processing to common educational datasets in Subsection 1.3.1.

Memory Based Collaborative Filtering

From the assumption of collaborative filtering (similar user history implies similar ratings on unknown items), we can easily derive a basic recommender system. Given a similarity metric, the prediction for a new item would be the average rating for other users weighted by similarity:

$$r_{u,i} = \frac{\sum_{u' \in U_i} S(u, u').r_{u',i}}{\sum_{u' \in U_i} S(u, u')}$$

where $S(u, u')$ is the similarity metric between user u and u' and U_i is the subset of users that already rated item i. Averaging can be reduced to only the top-N similar users.

This method has the same benefits and drawbacks as the ELO rating system. It has less predictive power than more recent methods described in the next paragraph. Another drawback is that it does not scale well with a huge number of users or items [6]. This limit is only seen with datasets in the order of millions of users or items. However, its implementation is far less complex than model based methods.

Its simple implementation made it the first method to be widely used in a commercial context for example with *Amazon* adopting it as early as 2003 [9] [2].

Model based collaborative filtering

Model based collaborative filtering [6] methods rely on more generic data mining and machine-learning techniques. They have better scalability than memory-based methods.

2. The *Amazon* implementation used a similarity between items and not users.

Matrix factorization Matrix factorization (MF) is a widely used technique in recommender systems, as illustrated by its extensive usage in the *Netflix Prize Competition* [17].

From the definition at the beginning of Subsection 1.1.2 with N users and M items, we can build a sparse rating matrix $X \in (\mathbf{R}^+)^{N \times M}$. The goal of matrix factorization is to find two matrices $W \in \mathbf{R}^{N \times k}$ and $H \in (\mathbf{R}^+)^{M \times k}$ (usually with low rank $k \ll N, M$) such that X is close to $WH^\intercal$. This is an optimization problem written as:

$$\underset{\substack{W \in \mathbf{R}^{N \times k} \\ H \in \mathbf{R}^{M \times k}}}{\operatorname{argmin}} \sum_{(i,j) \in \Omega} (X_{ij} - w_i h_j^\intercal)^2 + \lambda(\|W\|_F^2 + \|H\|_F^2) \tag{1.1}$$

Where λ is a regularization meta-parameter, $\|.\|_F^2$ is the Frobenius norm [18] and w_i and h_j are the i^{th} and j^{th} line of W and H respectively. Equation 1.1 may vary in regularization terms (bias, sparsity penalty...) and can incorporate a loss function between X_{ij} and $w_i h_j^\intercal$. We can now estimate unknown ratings within the product $WH^\intercal$. In other words, we look for signatures for users and items in the same latent space of dimension k (i.e., vectors of rank k), such that the outcome of the user rating an item is close to the dot product of these signatures.

This optimization problem is non-convex in general, but different methods exist [17]. The Alternating Least Square (ALS) method is the most popular method as it converges better than the Stochastic Gradient Descent (SGD) method due to non-convexity. When large-scale data is needed, as ALS is not easy to parallelize, Coordinate Descent is preferred [19, 18].

We will see applications of matrix factorization techniques to knowledge tracing in Subsection 1.1.3 and our contributions in Section 1.3.

Deep Neural Networks Recent works try to apply deep learning to recommender systems which look like a natural direction given the recent deep learning success in many fields such as Natural Language Processing. These systems need a lot of data to be trained, but this is made possible with the increasing use of large-scale recommender systems that can gather a huge amount of data.

A natural deep-learning model can be derived from Matrix Factorization. Instead of obtaining the predicted value with the dot product of user and item signatures, we obtain it with a learnable non-linear function. This non-linear function can be a whole deep neural network, for example, a multi-layer perceptron [39, 40].

However, this path has to be explored with care because comparison with baseline results from other methods, like classic Matrix Factorization, is not often reproducible [41, 42]. In order to give the most sensible results, recommender system frameworks like *Lenskit* [43], can be used to fairly compare algorithms and have reproducibility "out-of-the-box" [44].

Cold Start Problem and Online Settings The *cold start* problem is a typical problem in recommender systems that corresponds to the initial phase of a "nude" system (no data collected yet). The lack of data makes the prediction accuracy unreliable at that early stage. Matrix Factorization techniques are not designed to tackle the cold start problem but, some extensions seek to solve it partially [20, 21]. As we are not focused on prediction accuracy, we will not consider these extensions in this chapter but we will, in Subsection 1.3.2, try to evaluate when the cold start problem ends, that is, when there is enough data for Matrix Factorization to start giving results. Trivedi et al. [45] try to solve a cold start problem in an Intelligent

Tutoring System environment with spectral clustering to help refine raw prediction, but they work on the raw features of datasets without student or item signatures.

Even after a cold start, the system usually benefits from new data in general. This is referred to as online recommendation, and Matrix Factorization is widely studied in such a context [22, 23, 24]. These works consider extremely large datasets, about the order of millions of users and items, but it is still feasible to redo a factorization after adding a few elements, as we will see for instance in Subsection 1.3.2 and 1.3.3.

1.1.3 Knowledge Tracing

Knowledge tracing is the name of the task of modeling learners' knowledge. It is a mandatory task in any adaptive learning environment.

Depending on the method used, knowledge tracing must come with a set of skills and a hierarchy given by an expert. The exercises would also need to be labeled. These expert-made labels are named knowledge components (Knowledge Component). An exercise can work on several knowledge components. Some knowledge components are prerequisites for more advanced components.

Historically, the knowledge tracing community was independent of the recommender systems community. Knowledge tracing methods started to take inspiration from existing recommender systems with the growth of big intelligent tutoring systems and the development of collaborative filtering [27, 28]. From recommender systems terminology, in a knowledge tracing setting, users are *students* and items are *problems*.

However, there is a major difference between knowledge tracing and raw recommendation. The learning process is dynamic by nature whereas tastes and buying choices for example are more static in time. This will be discussed in Section 1.3.

Bayesian Knowledge Tracing

Bayesian Knowledge Tracing (BKT) [25] is a widely used sequential knowledge tracing method. It requires a set of Knowledge Components input and exercise labeling from an expert.

BKT models student learning with hidden Markov models. Each Knowledge Component has two states: mastered or not. When answering an exercise, there are four parameters that the model estimates:

— a probability to initially master the Knowledge Component before any practice.

— a probability to master the Knowledge Component after each practice (transit in the literature).

— a probability of guessing the correct answer even if the Knowledge Component is not mastered.

— a probability of failing although the Knowledge Component is mastered (slip in the literature).

This model achieves reasonably good predicting power for its few parameters. It is usually a baseline to compare other knowledge tracing methods [29].

It is rather easy to add some extensions such as taking into account item difficulty [5] or adding a forgetting model [30] where the mastered state of a Knowledge Component can go back to a non-mastered state if not practiced for too long.

Deep Knowledge Tracing

Deep Knowledge Tracing (DKT) [31] as its name suggests, uses deep neural networks to model student knowledge.

Like classical sequential models such as Bayesian Knowledge Tracing, Deep Knowledge Tracing captures sequential aspects of knowledge tracing with recurrent neural networks. The main difference with classical models is that Deep Knowledge Tracing does not need any Knowledge Component input. Moreover, the latent Knowledge Components discovered by Deep Knowledge Tracing as a by-product of training can be used to automatically discover Knowledge Component relationships [31].

Like many neural network architectures, many extensions can be added to increase accuracy [32, 33]. These extensions usually need richer inputs for the neural network [34, 35].

Some Deep Knowledge Tracing models take direct inspiration from recommender systems [36], matrix factorization can be used as a pre-training method for certain parameters such as static exercise signatures. However, time awareness is crucial in a learning environment and some extension to Deep Knowledge Tracing tries to take into account the time between exercises [37].

With the high number of parameters, DKT can also take as input a smaller scale than a whole exercise. Problems are usually split into smaller components that are the *problem steps*. We will see in Subsection 1.3.1 why we recommend a first regrouping pass in order to work with whole problems.

Like in the collaborative filtering literature in Subsection 1.1.2, more care has to be taken in the evaluation and reproducibility of results [26]. Especially when compared with common baselines that do not take the same input, one has to not bias the dataset in any direction.

1.2 ELO Rating for Recommendation

In this section, we will try to apply ELO ratings to different educational contexts and see if we can already use their predictive power while keeping a simple implementation. We could use a basic ELO rating before a more elaborated system gathered enough data to start producing more accurate predictions.

1.2.1 Simple ELO Rating on a Predefined Order of Exercises

We briefly mentioned in this chapter introduction that recommending an exercise is already hard even with a good prediction accuracy. So as a first step, we will choose a really specific sub-problem. We assume we already have a small set of exercises that are intended to be done in a linear order. These exercises and the order were done by a human expert beforehand as a teaching sequence.

We would like to know if the next exercise on the list is worthwhile to do for a student. Otherwise, the student can skip it.

Toxicode Compute-IT presentation

We contacted *Toxicode*: a company that develops web-based games for children to introduce programming concepts.

We focus on their most popular game *Compute-IT* (`https://compute-it.toxicode.fr/`). Students are given a program and they need to move a pebble as if it was controlled by the program (see Figure 1.1). If the student makes a mistake by moving the pebble outside the program trace, the level is restarted from the beginning and the pebble comes back to its starting point. This way they have to simulate the program in their head. New constructs and basic control flow can be added throughout the game. It is intended to work without any teacher intervention so the first levels are really progressive. The introduced concepts are (the concepts are given in chronological order):

— basic statements (left, right, up, down)

— repeat control flow. This is a *for* loop with a fixed number of iterations and without index.

— conditional control flow. They are introduced with only a *then* clause, later an *else* clause is added. The condition is based on the color of the circle under the pebble when the statement is executed.

— while loop. The condition is of the same type as conditionals.

— basic function calls

— recursive calls. No variables are introduced yet. This is done in some other *Toxicode* games. This makes recursive calls much more accessible.

These concepts are mixed together in several levels before adding a new concept. New concepts are introduced alone with only basic statements, only after a few levels they are mixed with other concepts.

Currently, *Toxicode* developed 60 levels unlocked in a linear path by the student (see Figure 1.2 for a screenshot of the path).

This game was featured in an *hour of code* event (`https://hourofcode.com/us/learn`) which gave them a large audience. They kindly agreed to collaborate with us and share the statistics they gathered.

These statistics contain about 2 million records, a record being a single attempt at a level. A record contains:

— a user pseudonymised ID.

— the level name.

— whether the attempt was successful.

— the time the student spent on this attempt.

— the number of steps after which the student failed or succeeded. This means the number of times the student moved the pebble.

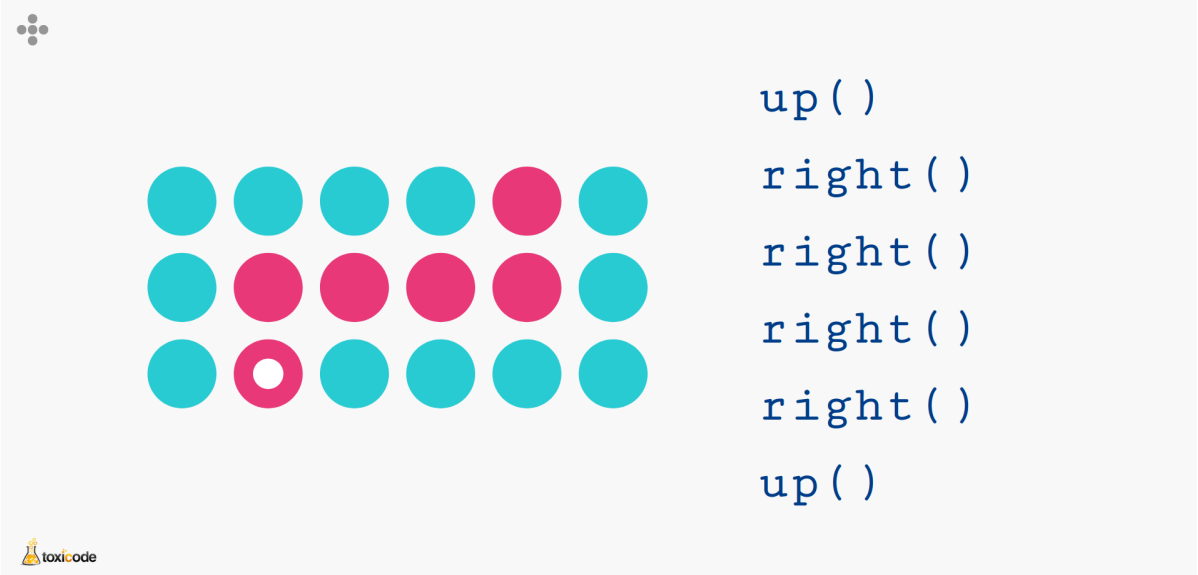


Figure 1.1 – A screenshot of *Compute-IT*. The player has to move the small pebble (the white circle) using the keyboard arrows by following the program instruction. This is the 3rd level, it introduces the fact that the colors on the map are decorative and the player should not blindly follow them. After 4 moves to the right the pebble will be on the last column.

We have no timestamps of records so we cannot take into account multiple play sessions and the time that passed between sessions. However, the vast majority of children use *Compute-IT* throughout a single session.

Problems with data gathered with children left in autonomy After early experimentation and looking at the data, we found out that some part of the data was not exploitable at all. There were many attempts in the first levels (in the range of several dozen) and the number of steps before failure was quite random. We expect this number of steps to grow after each attempt because children would be able to reproduce their past moves and try new ones even if they do not understand the code. We believe these children were left to use *Compute-IT* at home and enjoyed moving the pebble on the screen without thinking about the learning context. The people at *Toxicode* could not confirm where it came from but the vast majority of the statistics come from children playing at home and not in a class context with a teacher.

We removed these records from the statistics before doing further experiments.

Detecting levels that can be skipped

These statistics allow people at *Toxicode* to know whether the difficulty gap between levels is acceptable. When too many children fail at a given level, it means the gap between the previous level is too big. So they add a scaffolding level. The current path they have is now quite stable. This is motivated by the fact that if children play *Compute-IT* alone they have a high chance of dropping the game if they do not succeed after many tries.

However, those new scaffolding exercises may be numerous and some children may not need them and only use the system once. This can be a nice addition to know if a level can

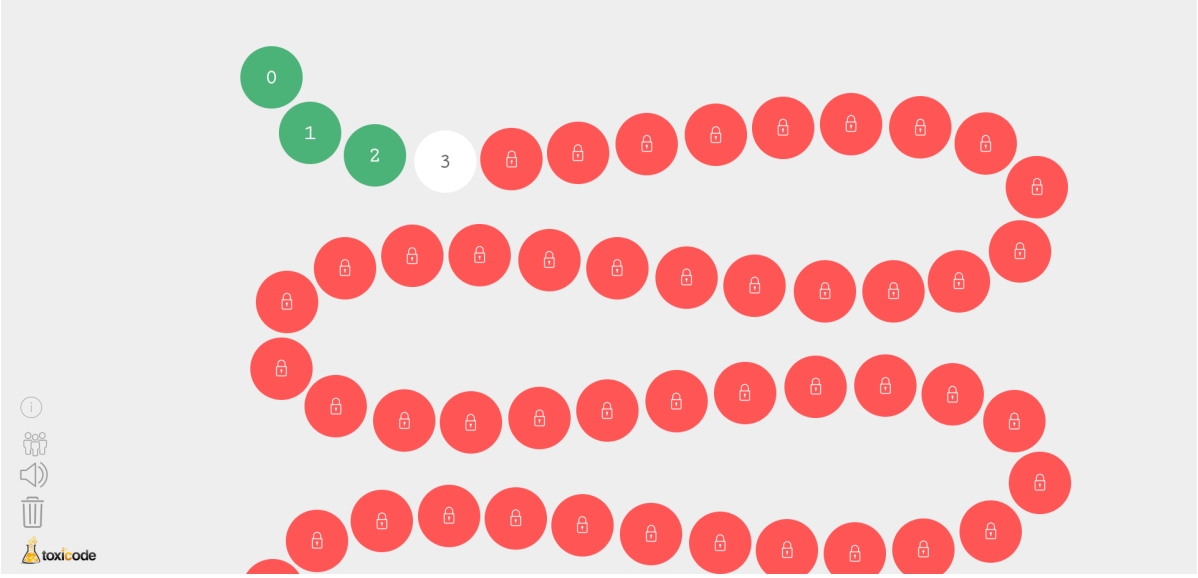


Figure 1.2 – A screenshot of the level list in *Compute-IT*. The three first levels are solved (the numbering starts at 0) and 4th one is unlocked.

be skipped without making the next levels too hard. If the probability that the children solve the level at first try the next two levels is above a certain threshold, we skip the next level. We may apply this procedure recursively to skip a series of levels but, this should need a better prediction power than an ELO rating. We can try to perform this level skipping idea because levels are supposed to have increasing difficulty.

Even if a direct implementation of ELO rating would mean to use only success or failure as input, we also use two other metrics to predict if a child will succeed in an exercise:

1. the number of tries before success
2. the total time to solve the exercise

However, the timescale to solve an exercise is not linear. The difference between solving an exercise in 10 or 20 seconds is not the same as between 600 and 610 seconds. For this reason, we consider a log scale for time. We give a minimum solving time of 10 seconds for all exercises such that our log timescale starts at 0 if a child solves an exercise in less than 10 seconds. We also clamp the solving time at 10 minutes. In this section, all mentions of time to solve a level will be the result of this preprocessing.

Levels covariance

Before doing further experiments, we want to measure how the number of attempts or time to solve previous levels correlates with further levels. If these kinds of metrics are independent between levels, it will be really hard to predict anything meaningful with simple methods.

For each level, we measure how the number of attempts or time to solve correlates with the same metric for each past level. We use Spearman's rank correlation coefficient as a correlation metric. It assesses how well we can express the current level result as a monotonic function

of past level results. It is expressed for two random variables X_i and Y_i and their respective rank variable $rg(X_i)$ and $rg(Y_i)$ as:

$$r_s = \frac{\text{covariance}(rg(X_i), rg(Y_i))}{\sigma_{X_i}\sigma_{Y_i}}$$

where σ is the standard deviation of a random variable. We use simple correlation to have a rough idea if knowledge used in past levels can be reused in the current level. Proper knowledge modeling will be done in Section 1.4.

These measurements are reported on Figure 1.3. We expected to see some correlation with levels in the same category (not introducing new concepts) and to see columns with less correlation when a new concept is introduced. Each column represents a level and each element in a column represents the correlation with previous levels. We looked at the average value of columns to know what levels have the less correlation with previous levels. First of all, these levels with less correlation with previous levels are not the same with both metrics. Regarding the number of tries, they are:

— *condition_simple* (level 11/60) it introduces the concept of conditional with a really short program.

— *conditional_repeat_is_not_while* (level 14/60)

— *conditional_negated_repeat_simple* (level 37/60)

— *conditional_negated_double_repeat_with_else* (level 45/60). Both this level and the *conditional_negated_repeat_simple* are part of a serie of quite difficult levels with many concepts merged or nested in different ways.

— *function_simple* (level 47/60) it introduces the concept of functions with imperative calls to a short function.

Levels designated like *"concept"_simple* introduce the *concept* in the name. So *condition_simple* and *function_simple* are actually levels that introduce respectively conditionals and function calls. However, the other levels are just merging concepts together. The other levels that introduce concepts do not show these darker columns. We don't know why we cannot see all the levels introducing concepts with this method as they should be less correlated with previous levels.

We expected time measurement to give the same "difficult" levels. However, with this timing metric, they are:

— *repeat_multiple* (level 7/60). This level features sequential repeats.

— *while_repeat* (level 34/60). This level nests a repeat loop inside a while loop

— *condition_negated_repeat* (level 41/60)

Anyway, the correlation values with individual levels are quite low. However, the fact that we can clearly see some "difficult" levels with the two different metrics and that these levels are different and does not correspond in general to the introduction of new concepts is puzzling (at least some corresponds to new concepts). We let this work as it is for now and do not go further in this direction. We saw in this section that unexpectedly these two metrics do not show the same correlation between exercises. However, conclusions on levels would be much stronger if they were true for both of these metrics.

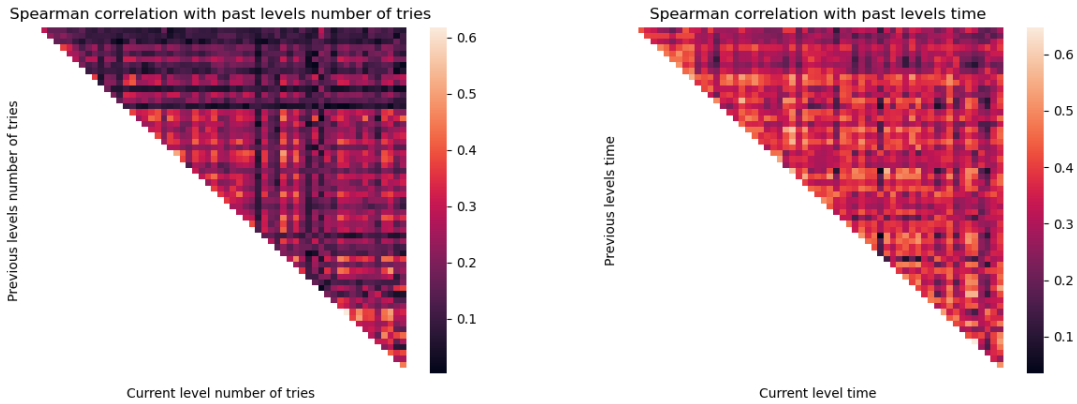


Figure 1.3 – The heatmaps are halved because the correlation is symmetrical. Individual columns tell if a particular level is correlated with previous levels' results. Respectively, individual lines tell if the level will be correlated with future levels.

Prediction with another metric

In this section, we will order students by accuracy or speed to solve levels and see if this order can be a good predictor for children's results. We still use logarithmic timescale but in this case, it is only for curve readability purposes.

Before trying any prediction we simply order students for three different levels with two metrics:

— the number of tries to solve the level.

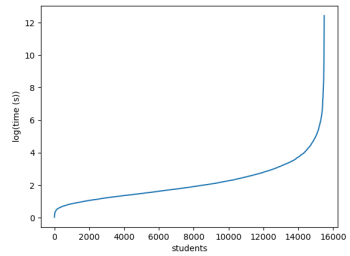
— the total time taken to solve the level (including failed attempts).

We chose *first_steps* (1/60), *condition_simple* (11/60), and *tetris* (43/60) which is one of the last levels. This is reported in Figure 1.4.

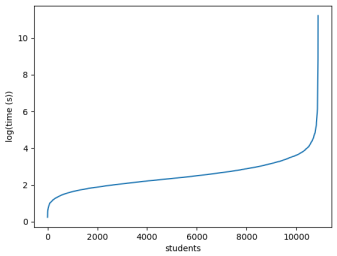
With these figures, we remark that the number of children attempting each level decreases the more advanced the level is (when looking at the size of the ordering). This is due to many children dropping *Compute-IT* after some levels. We see on Figure 1.4d that some children spend many attempts in the very first level which consists of only 3 moves right. We do not have any real-life interpretation for this behavior, so we decided to filter these long tails at the end of each curve before doing some predictions. The majority of children succeed in *first_steps* and *condition_simple* at the first try but this is not the case for *tetris*. We expected that children attempting an advanced level like *tetris* would have understood most of what *Compute-IT* has to teach and maybe take some time to think and solve the level or succeed in a few tries. It could be interesting to see if children that are attempting several dozen times *tetris* are doing it for all previous levels.

Since many children succeed on the first try many of the levels, we will consider only time ordering for prediction. After filtering children that perform several dozen attempts, we sort each children by their time to solve for each level and see how this ordering is preserved between levels.

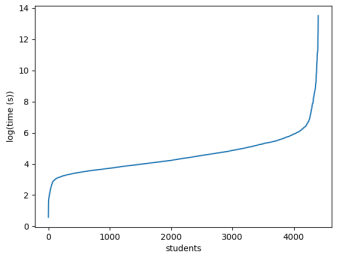
We will evaluate this by trying to predict children's position in this order at a given level. If we can accurately predict this, it means that this ordering carries some information about



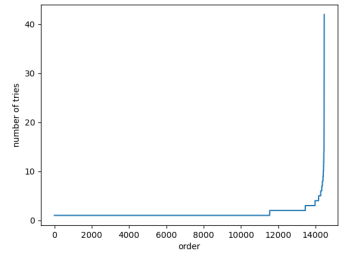
(a) Time empirical distribution to solve *first_steps*



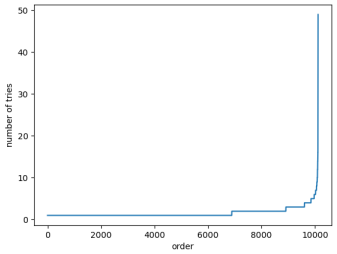
(b) *condition_simple*



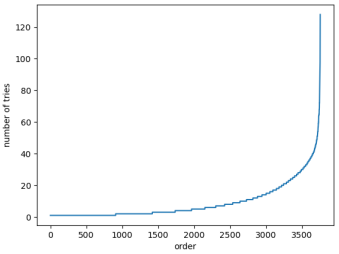
(c) *tetris*



(d) Number of tries distribution to solve *first_steps*



(e) *condition_simple*



(f) *tetris*

Figure 1.4 – Empirical distributions of time and the number of tries to solve *first_steps*, *condition_simple*, and *tetris*. The maximum values for time (printed in a logarithmic scale) are in the order of several hours, we believe this is due to student leaving the computer and coming back to solve the level after some time while the computer was still running. Each student is plotted, this way we can get an idea of how the student's time to solve each level and the number of tries is distributed. The small difference between the number of students with time to solve and the number of tries is due to incomplete data for some records so they cannot be computed.

children's results. After predicting a position we could predict a time to solve the level of children with similar positions and compare it with the actual solving time. However, as we are interested in understanding if this order is meaningful we will compare the position prediction with the actual position.

As the number of children at each level is not the same, we normalize the position and compare normalized values for prediction. 0 is the fastest time to solve a level and 1 is the longest. We clamp the time to solve to 15 minutes. This way an error in prediction such as 0.55 instead of 0.5 is interpreted as: we expected a solving time being in the 55% fastest, however, the real solving time is in the 50% fastest.

We will compare four prediction methods:

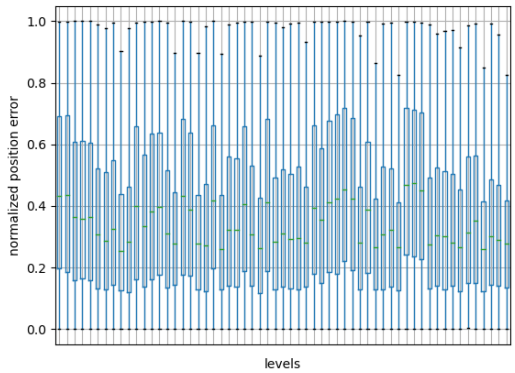
— A random baseline, we uniformly predict a position between 0 and 1.

— A prediction reusing the previous level normalized position $p_n = p_{n-1}$.

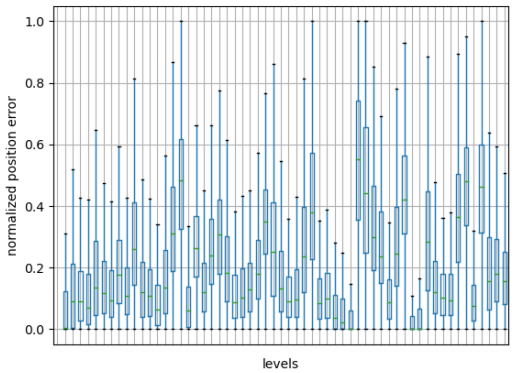
— A prediction averaging all previous level normalized positions $p_n = \sum_{i<n} p_i/(n-1)$.

— A prediction with a weighted average of previous level normalized positions, the weights are taken from the correlation heatmap in Figure 1.3 for each level. $p_n = \sum_{i<n} c_{ni} * p_i / \sum_{i<n} c_{ni}$ where c_{ni} is the correlation between levels n and i in Figure 1.3.

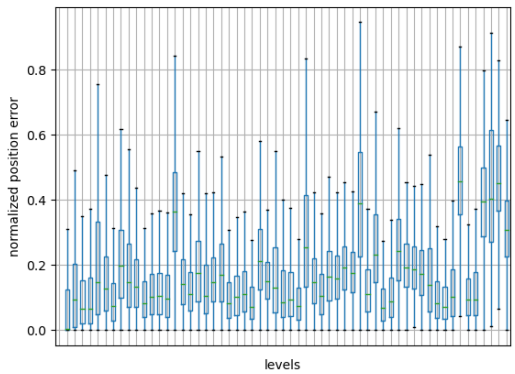
The prediction error is reported on Figure 1.5. We see that some levels have strong outliers with the three prediction methods. Like on Figure 1.3, these levels with strong outliers do not correspond to levels that introduce new concepts. For the three prediction methods and especially when averaging, the prediction error is in a much shorter range than the baseline and some levels are decently predicted.



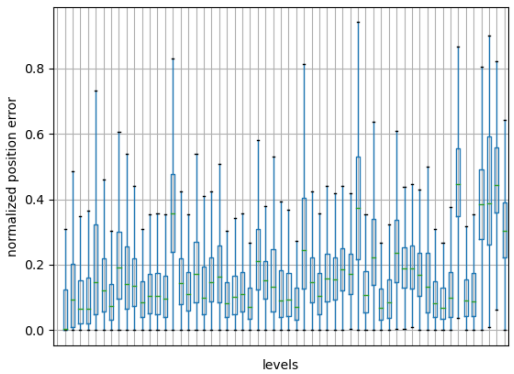
(a) Uniform prediction



(b) Prediction from the last level



(c) Average normalized positions from previous levels



(d) Average normalized positions from previous levels weighted from correlation

Figure 1.5 – Normalized positions prediction error by level for the four described methods.

A comparison of the average error by method is reported on Figure 1.6. We see that the worst predicted levels (the bars that are higher on the figures) are really close to the random baseline. A more interesting result is that prediction method complexity does not add much prediction accuracy. This is even stronger between the two averages. Weighting the average with the correlation coefficients from Figure 1.3 makes a negligible difference at a cost of increased complexity. This may be due to the small amount of information we were able to extract. The averaged errors, however, have a shorter range of values than only looking at the

last level for prediction.

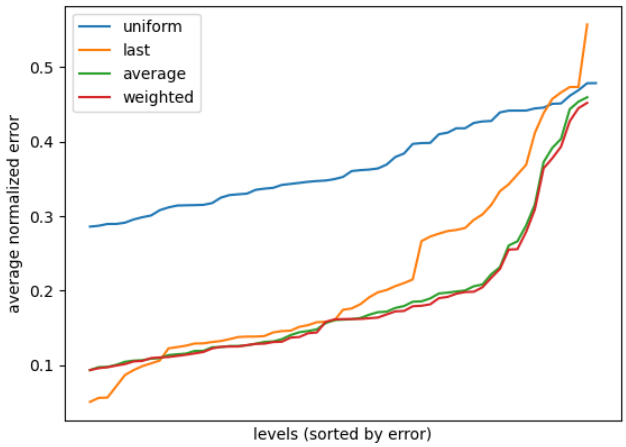


Figure 1.6 – Sorted average normalized error by level for the four methods

Having some levels with prediction accuracy close to the random baseline, we would need to find a way to discriminate these levels from the others. We could use Figure 1.5 to find a threshold when we would use our simple prediction method (averaging previous levels for example). We could choose a threshold of for example 0.2 which represents a bit more than 80% of levels.

1.2.2 Recurrent neural network to simulate ELO Rating

In the previous section, we tried another prediction method than an ELO-based method Even if the predictive power were not fully satisfactory we could still gain understanding about the dataset we used. In this section, we will try some other methods based on neural networks. The modeling of the neural networks can be seen as a minimalist version of knowledge tracing. We will use recurrent networks to encode the evolution of knowledge. In *Compute-IT* levels are in order so we could use only linear architectures but this would not be generic when we will not have a given order in the future. One of the architectures will, however, be done in a feed-forward way to see if we could apply it to skip *Compute-IT* levels.

We will succinctly present three architectures that look more or less promising with different targets. The recurrent layers will be LSTM [46] with sometimes several layers.

— The first architecture is a simple feed-forward success predictor for the next levels. It takes as input a vector with success on past levels, each dimension being a level. So the dimension is tied to our problem, here the input size is 64 (64 levels in *Compute-IT*). It contains some dense linear layers to increase the dimension up to 256, then some denser linear layers to reduce the dimension to the number of levels. All layers are ReLu activated except the last layer that is producing the prediction and has a sigmoid activation. It produces a prediction for all levels. We see that not all elements of the output vectors are compared to a target (for example not done levels) and used for backpropagation.

This network should be able to reproduce input from past levels in the output vector. We will call this network *simple-success*.

— The next model is totally different. The target this time is to predict the time to solve the next level (we use log scale time). The input is a sequence of vectors with two elements, the time to solve and the number of tries for each previous level. Each input vector is encoded with dense layers up to dimension 128. An LSTM layer encodes this sequence into a vector of size 128. Several dense layers decode this state up to a time prediction of dimension 1 for the next level. This architecture will be called *rec-time*.

— We now try the same idea to predict success. The architecture is the same as *rec-time* except that we add a third dimension to the input vectors which is the position from section 1.2.1. We thought that the position from previous levels will carry more useful information to predict raw success instead of time to solve. We call this model *rec-success*.

The *Compute-IT* dataset is divided into 70% samples for the training set, 10% for validation and 20% for evaluation. The inputs are then prepared for each network. Unfortunately, the prediction accuracy of all of these networks is close to random (even 0.47 for *simple-success* which means we would better predict a random value than using the network). We voluntarily kept simple architectures to gain some understanding of the dataset. As the result are really not promising (even though the networks converge), we conclude that neural networks for prediction should be used at least with highly parametrized models. The current models already have a decent number of parameters for the problem size. So we abandon the use of neural networks for exercise skipping with *Compute-IT*.

Conclusion on level skipping with Toxicode

In this section, we wanted to gain some understanding of children's progress in *Compute-IT* so we could recommend skipping some levels. We saw that we need some additional work to prepare a clean dataset from the raw statistics. This should be the case for every real-life scenario and especially when children are left without teacher assistance. Even if we did not succeed in our task, we gained some understanding of the dataset we had. We believe this is an important step before developing more complex models instead of being "in the dark" regarding the dataset. In the case of *Compute-IT*, it is puzzling that a simple neural network is not able to reproduce a simple scheme such as an ELO rating. Several explanations are possible, the network architectures we tested are not adequate, the data we used are not clean enough and we need some more preprocessing or we may not have enough data. However, this additional work to obtain a satisfactory neural network may not be suitable as a simple metric such as the success rate to a level already gives many insights on difficulty. If we really would to offer a personalized experience to children, maybe some simple clustering methods would be more appropriate as we should not overengineer a system just to offer a level-skipping feature.

1.2.3 Application to abstract games teaching without Predefined Order

In this section, we will try to work in a different domain to see how our conclusions with *Toxicode* generalize to other domains. Having a predefined order of levels offers some inter-

esting perspectives regarding prediction, but many other real-life applications do not have this order. We do not know yet if our hypothetical debug exercise database will contain a predefined order of exercises or not. It is more likely not the case. We chose to work with abstract games exercises: there should be many open data online and a lot of statistics are already done in this domain.

In abstract games, the main way to increase one's playing strength apart from playing many games is to solve many problems. They can be found in books or online. When done online, we can imagine a recommending algorithm that gives the next problem to the user. However, it would be really hard to evaluate statistically two recommendation algorithms because it would require large-scale experiments and measuring player's strength in tournaments.

Trying to obtain chess problems database

From a statistical point of view, chess problems are a natural direction. Open-source chess websites such as `https://lichess.org` have an open problem database containing more than three million problems with category labels and more than 20 million users. Moreover, the problems and users already have an ELO rating. Our goal is to use our past experiments to give a better prediction than the regular ELO rating implemented by Lichess. However, the user statistics on problems are not open and not anonymized regarding the usernames. When we contacted Lichess team, they could not share with us the records because of this lack of anonymity. They did not have time to add this anonymity layer to their database so it could be extracted and shared.

At first, we wanted to deploy our own instance of Lichess to gather player statistics but this was not an easy task and we did not spend a lot of time on it.

Application to go problems database

After Lichess attempt, we got in touch with the developer of a game of go (`https://en.wikipedia.org/wiki/Go_(game)`) problem website named *TsumegoHero* `https://tsumegohero.com/` (a go problem is called *tsumego* hence the name of the website). The game of go is much less popular in Europe than chess but this website is still fairly known in the European go community. Around 300 people on average use the website each day. The website problem database contains about 6000 problems. They are organized in collections of about 200–250 problems each. Problems in a collection have about the same difficulty (however they are always a few big outliers in terms of difficulty). The main difference with chess problems is that the problems do not have labels and thus cannot be sorted by category or such.

The website developer wants to add some small mini-games around go problems but this is still a work in progress. Currently, each problem is associated with an "experience" value set by human experts proportionally to the problem's difficulty. By solving problems users gain experience points and increase their level after reaching a certain amount of experience points. Levels are used to unlock some small helps to solve problems like hinting the first move for example. There is also a leaderboard of players who reached the highest levels.

The website already records some statistics and we start with a small ELO implementation to evaluate the statistical problem difficulty and compare it with the experts given experience values. The first "mini-game" is to solve many problems to increase one's own rating and the best scores are displayed after each month.

Implementation of an ELO rating in *TsumegoHero* The implementation part was quite specific. First, *TsumegoHero* was the first project of the main developer. This has a few consequences, like the website codebase is plain PHP and HTML/CSS and the database is not open and does not have a development version. There is, however, a test version of the website hosted online but the code is not versioned nor open. This meant that we needed to write the implementation without testing locally. We sent some files to the developer containing a not tested ELO rating implementation in PHP. We do not know the modification he made before sending it to production.

The problems were initialized with an arbitrary ELO value derived from their *experience* value given by human experts. A new "rating" mode was made in *TsumegoHero* where a new problem would be recommended. The recommendation algorithm is quite simple but should be sufficient. We select the problems that were not done by the user in the last six months. We choose randomly one in a range of ELO ratings around the user's rating (+50 and -50 points). In the rare case where no problems fit this condition, we extend the window incrementally until some problems are found.

After success or failure, we update the user and problem ratings according to the result following the ELO scheme.

Results after a one-year experiment After the ELO rating implementation, no additional statistics were added so we could follow usage statistics but the ELO rating stayed a black box from outside. The results in this section will be only based on the feeling of the users we could gather with casual discussion online or at tournaments with other go players.

The new "rating" mode received a lot of early interest from users. I heard several people talking about it online or in tournaments. The mode was indeed often used instead of the classical "collection" mode.

However, the ELO rating was not well calibrated and the system could not make the difference between hard and really hard problems. They had about the same ratings.

It appeared that our mapping from experience points to the initial ELO was not really satisfactory. This is not a big problem because ELO ratings will converge with time as there are many users on the website compared to the problem collection size. However, it changes the range of our rating, although we tried to align the rating with other go rating online. This is not a major problem but, the rating scale was really different from the usual online go ratings.

After a few months of use, the usage of the rating mode decreased. We believe this comes from these two reasons and that we could not test the ELO rating beforehand to correctly calibrate it. ELO rating has the advantage of having only a few parameters. Actually, they still need to be tested in advance by simulation or using past statistics.

The developer of *TsumegoHero* decided to change the ELO rating to a much simpler metric: the completion rate of the problem. The completion rate for a problem is the number of users who completed the problem divided by the total number of users.

This metric has the same flow of not being able to see the difference between hard and really hard problems but it is much simpler and much more understandable. We could imagine more of a success rate instead of a completion rate to take into account multiple failed attempts before solving the problem. This may be able to distinguish really hard problems.

Conclusion on application to abstract games Anyway, this "completion rate" is the metric still in use today and the "rating" mode is reasonably popular among users. Such a simple metric is still able to perform basic recommendations in the game of go and, we believe abstract games in general. However, it is impossible with this method to obtain a statistical understanding of the problem categories. In the case of chess problems, there are some methods to automatically derive labels from an expertly made label list. If manual labeling is not an option, experts would also need to write discriminant functions based on the solution of the problem given by engine top moves. Given an engine that is strong enough to find problem solutions, it should be possible to do this in many abstract games. However, these categories are made by human experts. There should be a way to find these categories statistically, but the efforts might not be worth it. Manually labeling problems is hard for big problem sets but the vast majority of abstract games today have open-source engines that can solve problems so we can derive a labeling from the engine moves.

Conclusion of this section and generic statistical work related to learning environment

During this section, we gained many insights that model simplicity is a quality that should not be underestimated. Simplicity can be preferred to accuracy for example in the *Tsumego-Hero* example. This indirectly suggests that a predefined order of levels is a quality for an exercise database because teachers can carefully follow simple metrics such as the success rate add adapt the exercises for everyone. This, however, cannot be used to drive or help any automatic generation method. The fact that we may want to skip easy exercises can be mitigated by having an engaging environment like in *Compute-IT*.

We also saw that data quality is an important factor and is even necessary to develop more complex statistical systems. In the next section, we will leave for a while our applied scenarios and we will try to work directly on mature educational datasets used in the knowledge-tracing community to evaluate complex models.

1.3 Collaborative filtering

In this section, we will target the more generic recommendation problem without any learning applications contrary to the in the previous section. We will nonetheless use datasets used in the knowledge-tracing community because working on purely recommendation datasets such as media recommendation may not generalize well to our topic. These datasets are produced over several years of recording some Intelligent Tutoring Systems (ITS) activity.

We will focus on matrix factorization methods. These are traditionally used in contexts where the available data is not very sensitive to time, for instance, movie tastes and shopping habits. In contrast, students learn each time they practice and should normally improve with time, so it would make sense to take history into account when analyzing datasets, making predictions and doing structural studies. However, we do not know how much impact this has on the results. The main topic of this section is whether taking history into account is necessary or if it is still possible to make good predictions when considering datasets as timeless. Recommending a good problem in terms of teaching is not an easy task, but it is even more difficult when we cannot reliably predict whether the student will succeed or fail, and how long it will take them to do so. In the rest of this section, we will study how the

matrix factorization algorithm behaves in three datasets from the educational data mining community compared to one dataset from the traditional collaborative filtering community. We will often deliberately *leave out* chronological information in the educational datasets to see how much information can still be extracted, compared to a traditional dataset.

1.3.1 Datasets and preprocessing

Before doing further experiments, we studied matrix factorization (MF) on four different datasets, and found that not all datasets were directly usable without some pre-processing, compared to classical datasets. We use the basic version (L2 regularization) of Equation 1.1 with a fixed rank of 20 (apart from Subsection 1.3.5 where we measure the impact of rank variation). We use coordinate descent for the optimization [18] because some experiments in sections 1.3.2, 1.3.3, and 1.3.5 require numerous factorization.

Table 1.1 – Raw data sets overview

Data set	Users	Problems	Steps	Steps occurring once	Mean samples per step	Samples
Algebra I 2006-2007	1338	5644	418 060	314 198	5.4	2 270 384
Bridge to Algebra I 2006-2007	1146	14 787	202 672	46 935	18.1	3 679 199
ASSISTment09	4217	17725	26 688	3123	13.0	346 660
ML-1M	6040	3706	N/A	N/A	269.9	1 000 209

Table 1.2 – Preprocessed data sets overview

Data set	Users	Problems	Samples	Density	Mean samples per problem	Success percentage
Algebra I 2006-2007	1147	3111	152 709	0.043	49.1	0.79
Bridge to Algebra I 2006-2007	1068	8736	235 147	0.025	26.9	0.91
ASSISTment09	2025	12587	238 746	0.009	19.0	0.98
ML-1M	6040	3706	1 000 209	0.045	269.9	N/A

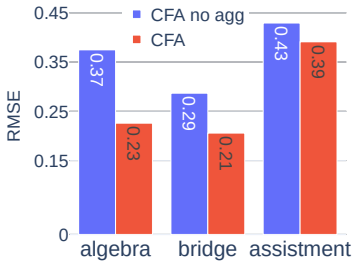


Figure 1.7 – RMSE between no aggregation and aggregation for correct first attempt

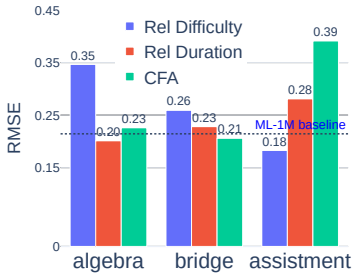


Figure 1.8 – RMSE for duration, correct first attempt and difficulty

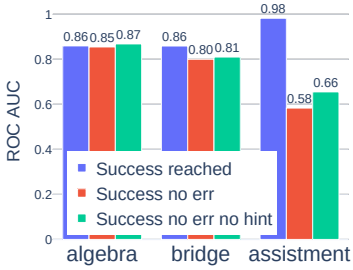


Figure 1.9 – ROC AUC for success and derivatives (higher is better)

Educational Datasets and Pre-processing We will use three common educational datasets for the rest of this section: Algebra I 2006–2007 [10], Bridge to Algebra I 2006–2007 [11]

(both of these come from the Cognitive Tutor problem set) and ASSISTment09 (we use the corrected and collapsed version of the dataset) [16]. All three datasets record scaffolding (intermediate questions) problem statistics (also called *steps* in Cognitive Tutor datasets—we will use both terms here). For each record (also named *sample*), we extract:

1. A student ID and a *main problem* ID;
2. A *scaffolding problem* ID;
3. A *timestamp* when a student starts a step and the *duration* to complete the step;
4. If the student succeeded at his first attempt: *Correct-First-Attempt* (CFA);
5. The number of *hints* and *errors* of the student for this step.

To our surprise, these datasets are not very usable without pre-processing in comparison with well-established recommender system datasets like the MovieLens dataset. We will compare most experiments with the ML-1M version of this dataset, which will act as a "control" dataset: multimedia recommendation datasets being the canonical use of matrix factorization for recommender systems. The two main reasons for this poor usability that we try to mitigate with pre-processing are the following:

— The notion of scaffolding problem is not standardized between the datasets and is hard to use as it is. Some of them are optional, which makes the number of steps for a main problem vary between students. The step order may also change between users, which makes matching between users more difficult at the problem level.

— There is no guarantee of the minimum number of occurrences for a student or a step. Moreover, many steps are attempted by a single student across the whole dataset, as seen in Table 1.1 (especially for the Algebra I dataset, where steps can be generated for a student from a template, and are thus unique. These constitute up to three quarters of the steps).

Our first pre-processing pass, which is motivated by the very low number of samples per step on average, corresponds to aggregating all the steps of a common main *(student/problem)* pair together. Aggregating timestamps and duration is straightforward (the beginning of a problem is the beginning of the first step and the total duration is the sum of the steps' duration). To aggregate "Correct-First-Attempt" we take the mean across a *(student/problem)* pair so, we obtain a floating point value between 0 and 1 instead of a boolean value.

Simply aggregating hint and error count by summing them is not satisfactory because ultimately we want to have an idea of how much a student struggled on a problem. Summing these quantities is not sufficient to access some basic information such as "*Has the given student reached the end of the problem or given up?*" This information is not provided in the datasets, so we had to build a proxy variable. To answer this question, we need to know, for a given problem, the number of basic steps it decomposes into. To find this quantity, which we call the *problem size*, we counted for each problem/student pair the number of samples. For a student who succeeded (possibly with hints and intermediate mistakes), the problem size and this number should match. We assumed that for a given problem, the most represented number (among all students) was the actual problem size. We believe that this high representativeness comes from the fact that the ITS providing the datasets give enough hints for most students

to reach the end of the problem before giving up. This makes the number of hints and errors valuable information to measure the difficulty of a problem for a given student.

Once we have a boolean proxy indicating success by reaching the end of a problem, we can derive two variables: reaching the end without errors and reaching the end without hints. We can also build a difficulty variable to aggregate the hint and error counts: we sum up the two counts with a 0.5 coefficient for hints. We represent a failure by assigning a difficulty value of twice the maximum value.

After aggregation, we have six variables (called *target variable* or simply *targets* from now on) of interest for each student/problem interaction: *duration* (0–1 scale value), *difficulty* (0–1 scale value), *correct-first-attempt* (0–1 scale value) and *success-reached* (boolean value), *success-no-error* (boolean value), *success-no-hint* (boolean value). The first three are normalized per problem so that for each problem, the "worst" student gets a value of 0, and the best one a value of 1 (giving rise to what we called above a 0–1 scale value). ML-1M has a single target which is the movie rating (also normalized for comparison). After aggregation, we filter out users who have done fewer than 20 problems and problems that are done fewer than 5 times (same threshold than for ML-1M). Table 1.2 shows the size of the datasets after pre-processing.

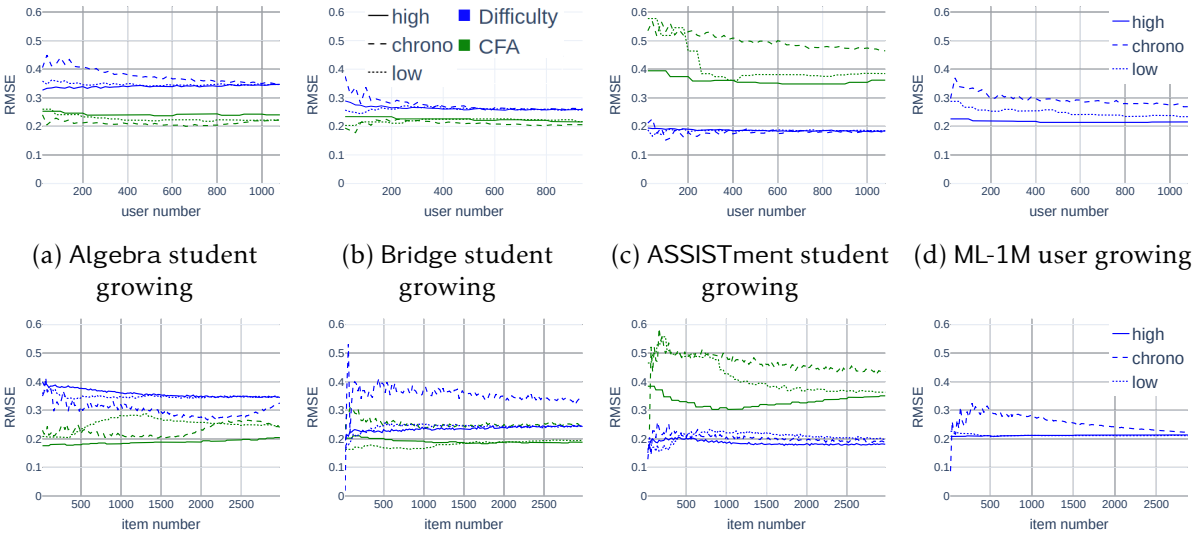
Influence of Aggregation on Datasets Figures 1.7, 1.8, and 1.9 report the ability of a factorization to accurately model the different target variables on the four datasets and compare the effect of aggregation. For 0–1 scale variables we use the root mean square error (RMSE – the smaller the better) as the error metric, and for boolean variables we use Receiving Operating Curve Area Under Curve (ROC AUC—the closer to 1 the better). For ML-1M, the only possible target variable is the movie rating. We report it in Figure 1.8 as a horizontal line.

In Figure 1.7, we see that the aggregation and filter procedure improve the prediction quality of the Cognitive Tutor datasets in a notable way, but only by a small margin for the ASSISTment dataset. We believe that if the pre-processing removed about one third of the problems and half of the students, the density would still be very low compared to the others. In all the remaining experiments, we will use the aggregated versions of the datasets.

It is hard to find any trend regarding the RMSE differences in Figure 1.8. Variations seem to indicate that different datasets favor different target variables. Matrix Factorization can have about the same prediction capability for educational datasets and multimedia datasets if the target variables are chosen carefully, which suggests that situations call for pre-analyses in order to select the target variable which will be the most accurately predicted.

In Figure 1.9 we can see that accuracy on success classification is reasonably good. However, we cannot explain the difference in ASSISTment between success-reached and the two other success target variables. This might stem from the aggregation procedure that relies on approximated methods to obtain the number of steps in a problem. We will not do any further experiments on these target variables (which are boolean), as they are barely comparable with the 0–1 scale variable of the ML-1M dataset.

The density of the data set does not look to be an important factor which suggests that Matrix Factorization succeed to extract the information it can. However, we cannot explain the differences between the datasets for having different behaviors about target variables.



(a) Algebra student growing (b) Bridge student growing (c) ASSISTment student growing (d) ML-1M user growing

(e) Algebra problem growing (f) Bridge problem growing (g) ASSISTment problem growing (h) ML-1M item growing

Figure 1.10 – RMSE evolution for student and problem set growing on the four data sets
If not specified the legend is the same as (b).

1.3.2 Online Prediction

In this section, we will try to evaluate the point at which there is enough information to predict reasonably well with Matrix Factorization techniques. This allows the system to stop using whatever bootstrapping technique it was using to solve the cold start problem.

To evaluate this we start by getting either the full student set (and no problem) or the full problem set (and no student). We then progressively add new problems in the first case and new students in the second case, adding 20 new elements at each iteration. At each iteration, we redo a full factorization and evaluation as if the system was complete.

We independently measure RMSE of correct-first-attempt and difficulty variables. To evaluate whether the order in which new elements are added makes a difference or not, we considered three orders: (i) elements sorted by their number of occurrences either in decreasing (*high* density first); (ii) or in increasing (*low* density first) order; (iii) and following the chronological order (*chrono*). Only the chronological order makes sense in an online context, but we still use the number of occurrence orders to evaluate whether or not we benefit from a higher density.

We report in Figure 1.10 the results of this experiment. We can see that for three out of four datasets (not ASSISTment), adding elements by highest density makes the system converge really fast (about 200 elements for Bridge), which was to be expected as those elements carry the most information. For all datasets, adding elements by the lowest density, as we might expect, makes the system converge really slowly. We believe that the extremely accurate prediction on some of the curves for the first few iterations of the growing process is due to overfitting (recall that the factorization uses a rank of $k = 20$ in those experiments).

Still, there are some artifacts to these results. In Figure 1.10e, the previous claims are reversed for difficulty target. Maybe this is a hint that this aggregated variable may not be

robust enough on all systems. Our advice is to systematically test target variables on a system to make sure that the ones we choose are consistent and can be trusted.

Finally, we do not observe any "dramatic" drop of the RMSE in curves representing the chronological order that we could clearly label as the "cold start" (although it sometimes takes a few "adds" to stabilize). Of course, the highest accuracy is obtained whenever all the data is used, but this suggests that Matrix Factorization accuracy start to get close to the maximum early in the process. However, bear in mind that we only evaluated our ability to model existing data (we evaluate on the matrix we factorize), but did not evaluate our ability to predict (by evaluating on the remaining, not factorized, part of the matrix).

1.3.3 Student or Problem Prediction Kernels

In this section, we search for a subset of students and problems where the prediction is more accurate than on the rest of the dataset. Having such a subset can be of interest in various ways: further analysis like signature clustering might work better on a subset with high accuracy prediction, or this can be a first step towards building a confidence measure for new predictions using a similarity measure with this accurate subset.

Note that we briefly tried some clustering algorithms on the student and problem signatures given by Matrix Factorization, but they were not promising. We will explain in Subsection 1.3.5 how the signatures we can obtain with Matrix Factorization may not be appropriate to such a study.

1.3.4 Iterative Filtering

We describe an iterative procedure to filter students and problems that have the least accurate predictions.

We alternately remove students and problems: at each iteration, we remove the 8% of the considered set that are the least accurately predicted in terms of RMSE (or 15 elements if 8% is lower than 15). We report in Figure 1.11 and Figure 1.12 the evolution of the density of the rating matrix and RMSE for the difficulty target variable.

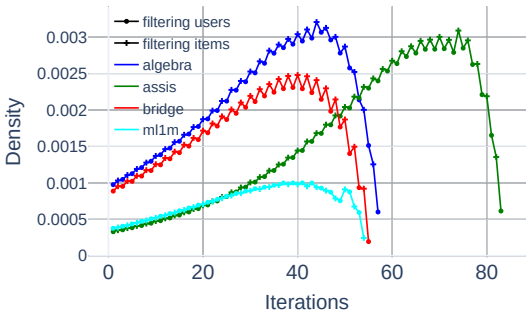


Figure 1.11 – Evolution of density during filtering with target **Difficulty**

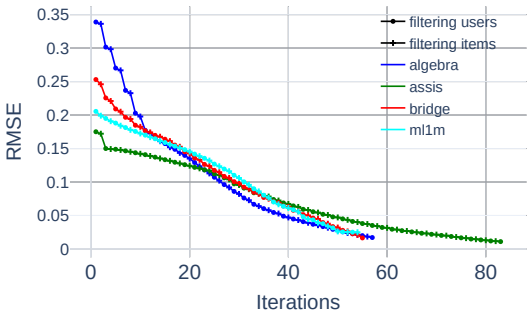


Figure 1.12 – Evolution of RMSE during filtering with target **Difficulty**

Figure 1.11 presents the variations in density as we progressively remove students and problems. Interestingly, removing items usually increases the density while removing students decreases it in the three educational datasets, meaning that the students that solved

many problems are viewed as "problematic" by the system. This behavior is not observed in the reference ML-1M dataset. Figure 1.12 confirms the tendency that removing students, in the beginning, tends to improve prediction accuracy. This result is disturbing as it means that, for the educational datasets, Matrix Factorization prefers less dense matrices with regard to the users, i.e., less information for a given student. This suggests that Matrix Factorization perform best when a problem was done by many students, but when the students have done few problems. What is interesting here is that this scenario is the one that resembles most closely the ML-1M dataset: by having students that did fewer problems, we are indeed eliminating students that likely *progressed* during the experiment; hence whose behavior cannot be represented by a single vector across all their interactions. This is a first solid hint that Matrix Factorization alone does not seem suited to educational datasets, as it shuns chronological subtleties.

1.3.5 Influence of Rank Variation

In this section, we repeat the experiments from previous sections with different rank values. In addition to rank $k = 20$ that we already measured, we use rank 5 and a rank of 1. We deliberately choose a rank of 1 to mimic a Whole History Rating (WHR) [14]. Even though it is not an exact correspondence, we believe that the information extracted by a Matrix Factorization with rank 1 can also be extracted by a WHR.

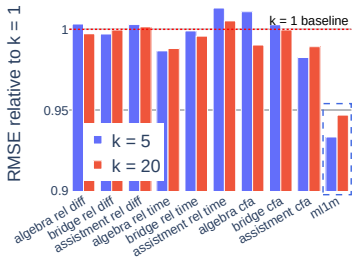


Figure 1.13 – Evolution of flat factorization RMSE depending on rank relatively to k = 1

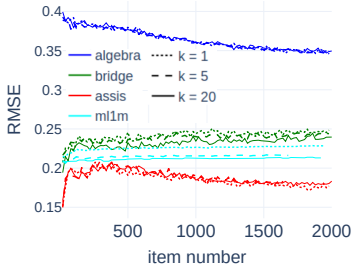


Figure 1.14 – Evolution of RMSE while adding problems with target **Difficulty**

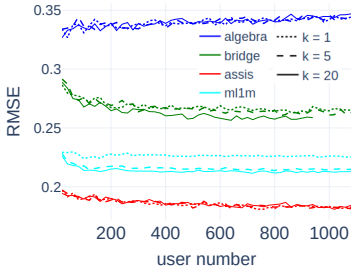
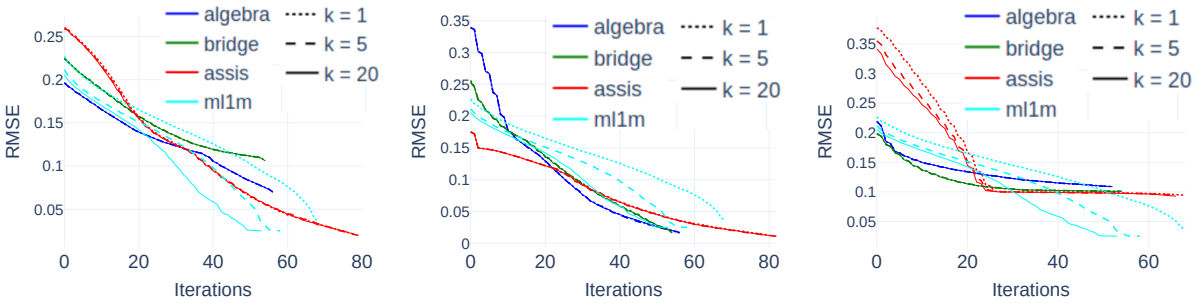


Figure 1.15 – Evolution of RMSE while adding students with target **Difficulty**

We see in Figures 1.13, 1.14, 1.15 and 1.16 a clear difference between the educational datasets and ML-1M regarding the influence of ranks. This benefit from rank increase agrees with the intensive use of Matrix Factorization techniques in multimedia recommender systems. However, the benefit of such an increase for educational datasets is almost negligible. This is particularly apparent in Figure 1.16, where the ML-1M RMSE curves get lower with increasing rank while all other curves are nearly indistinguishable by rank. This shows that, when the chronological information is not used, vectors of size 5 or 20 do not improve accuracy compared to a simple vector of size 1, i.e., a single float. This suggests that we cannot do better than assign a single number to problems and students, which could be interpreted as having a "difficulty" rating for problems and a "skill" rating for students, mimicking a WHR rating system. This is a second strong hint that educational datasets do not have the same structural properties as datasets from multimedia recommenders and that if we want to



(a) Duration (b) Difficulty (c) Correct First Attempt

Figure 1.16 – Evolution of RMSE during filtering with various target

extract more information and discover a better characterization of students and problems, it is necessary to consider chronological information.

Conclusion on Matrix factorization

We applied preprocessing to common educational datasets to try to improve the accuracy of Matrix Factorization techniques. While these did improve the results, we also showed that when Matrix Factorization techniques from the collaborative filtering community are directly applied, they do not benefit from having ranks higher than 1, meaning that the attribution of a single value to students and problems is about as effective as we can get. This seems to indicate that Matrix Factorization techniques might not be the most efficient model to extract static information from these datasets, or, more probably, that static information is scarce. Still, in the eventual absence of more sophisticated analyses in a recommender system, Matrix Factorization can be used to extract a crude measure of what could be labeled as a level of difficulty of a problem and a level of proficiency or skill of a student. We believe that this stems from the fact that, unlike users in multimedia recommender systems, students change over time as they are faced with new problems but also from outside interactions not recorded in ITS, hence chronological information needs to be taken into account in order to improve accuracy and make predictions.

1.4 Recommending in a learning environment

In Section 1.3, we saw some limits of applying raw recommender systems techniques to a learning context. We also showed the necessity of using dynamic methods for generic knowledge tracing.

In this section, we will keep working with common educational datasets and not specific applications like in Section 1.2.

1.4.1 Recommending an exercise after prediction

We told at the beginning of this chapter that deriving a recommendation from a result prediction is not trivial at all. To our knowledge, this problem is seldom addressed in the current knowledge tracing community.

We are afraid that this problem is more in the educational psychology field where we do not have any expertise. Educational psychology can provide some forgetting models [87], learning theories [88], and motivation models [89] that could be used by a human expert. Actually, the first knowledge-tracing methods are directly derived from educational psychology.

However, the modeling needed to accurately predict the results to a given exercise could be used to derive useful knowledge about the system. Such information could be for example extracting a set of categories or finding difficult exercises. In this section, we will try two methods that will not provide a direct recommendation but could help a human expert to give a recommendation or evaluate an existing pedagogical sequence.

1.4.2 Neural Network with a confidence metric

An interesting result for a prediction system (and thus a recommender system) could be to give a confidence rating along with the prediction. This is useful for example if some samples are especially hard to predict so we know that we cannot trust the prediction too much. This is the primary use in the small literature regarding neural networks with confidence metrics [47].

We also hope to be able to recommend exercises based on the confidence metric. If the confidence in the prediction is really low, maybe this is an interesting exercise to recommend. If an exercise is too easy or too difficult the system should see it and have high confidence in the result.

There are many existing heuristics for 2-class classifiers [47] (into which our success-failure prediction task falls). One could derive different confidence measures from the activations of the output neurons. However, these measures are not trained as an additional output of the neural network.

Another interesting architecture that incorporates confidence directly in the loss function is proposed [48]. They use a metaphor of students passing a written test. The student can get full points to question where they are confident. Then they can be given some hints for the remaining questions and get partial points. In neural network terms, the network has an additional output along the prediction which is its confidence in the given prediction. The network can use the actual target value to perform a prediction but this will be penalized in the loss function. This is described in equations below, taken from [48] where p_i' is the output prediction, p_i is the raw original prediction, y_i is the target and c is the confidence metric. For the loss L, L_t is the usual prediction loss, L_c is a log penalty for low confidence and λ is a meta parameter.

$$p_i' = c * p_i + (1 - c) * y_i \tag{1.2}$$

$$L = L_t + \lambda L_c \tag{1.3}$$

This way the confidence parameter is completely part of the training process. Despite the simplicity of adding a confidence metric to an existing scheme, we were not able to have anything mature enough to perform further experiments. We do not know if this is only due to engineering problems and lack of expertise in machine learning experimenting or if there is indeed a dead end.

After this early experimentation, we nonetheless came up with another architecture with a confidence metric incorporated into the training process. We reuse the simple recurrent network from Subsection 1.2.2, however, we predict two bounds for the success probability.

The error propagated during training is computed with the two bounds and there is a big penalty if the lower bound is higher than the higher bound. This is described in the following equation where t is the target (0 or 1 for success or failure), l is the lower bound probability, h is the higher bound probability, β is a meta parameter:

$$L = \text{MSELoss}(l, t) + \text{MSELoss}(h, t) + \beta * (l > h) \tag{1.4}$$

Unfortunately, we also did not pass the early experimentation phase. However, this time this may come from engineering problems because our loss function does not look like a usual loss function in the machine learning field and we reuse a base network from Subsection 1.2.2 that was not mature.

We think this confidence metric track is worthwhile to explore because it can be a powerful addition to mitigate usual recommender system problems applied to education. It could be able to call for a human expert instead of performing a poor recommendation. However, we may not have the expertise to fully develop these systems.

1.4.3 Direct application of Hawkes processes to extract knowledge components

It is beneficial to be able to extract knowledge components without expert intervention. It can be applied to huge problem sets that are not labeled. For example, we could use such a method in the go problems set in Subsection 1.2.3.

A recent article was published introducing the use of Hawkes processes to model student knowledge [38]. A Hawkes process is a self-exciting point process. Many punctual elements in time contribute to the final value.

We will first describe their architecture as a knowledge tracing model and then see how we tried to apply it to extract Knowledge Component. Here is the main equation from their article that describes the modelization where $\lambda(x_i)$ is the probability to succeed in the next exercise and the x_js is the student history :

$$\lambda(x_i) = \lambda_0(x_i) + \sum_{x_j} \alpha_{x_j,x_i} \kappa_{x_j,x_i}(t_i - t_j) \tag{1.5}$$

Let us decompose the previous equation. The left term $\lambda_0(x_i)$ is the inherent difficulty of the problem. It is composed of two parameters of the model, the sum of the inherent Knowledge Component difficulty and a per-problem difficulty.

The α values are coefficients of cross-effect matrices between Knowledge Component. In other words, it measures how success or failure in KC x_j influences the current KC x_i.

The κ function is a kernel function to capture time decay. In their implementation it is expressed as:

$$\kappa_{x_j,x_i}(t_j - t_i) = \exp\Big(-1(1 + \beta_{x_j,x_i})\log(t_i - t_j)\Big) \tag{1.6}$$

Like α values, β values are learnable parameters in a cross-effect matrice between Knowledge Component but this time for time decay. Again, the term cross-effect means how success and failure in past exercises influence the current exercise.

As success and failure for past interactions should be taken into account differently, α and β values are duplicated for success and failure respectively in the same matrices. The matrices that contain these values are of shape $2S \times S$ where S is the number of Knowledge Component. These matrices are learned in a factored way so actually 4 smaller matrices are learned instead

of 2 which greatly reduces the number of parameters from $2 \times S^2$ to $2 \times S \times K + S \times K$ where $K << S$ is the rank of the factorization.

This model shows good results compared to DKT baselines [38]. However, we are not interested in the predictive power of the model (because we think it is still insufficient to perform a good recommendation afterward) but its ability to model cross-effects between Knowledge Component to try to extract them. As their work is accessible online and easily reproducible we could modify their architecture in a way that Knowledge Components are not an input anymore. Thanks to the factorization reparametrization for learning we can also increase the size of cross-effect matrices: however our change still massively increases the number of model parameters. First, we do not use an inherent difficulty $\lambda_0(x_i)$ for Knowledge Component, only for exercise. Also for cross-effect matrices, we do not do it per-KC cross-effect but per-exercise cross-effect.

We gain some first insights by training this modified model on the 10 most done exercises in *ASSISTments_12* datasets. We remove all occurrences of other exercises in the dataset and observations.

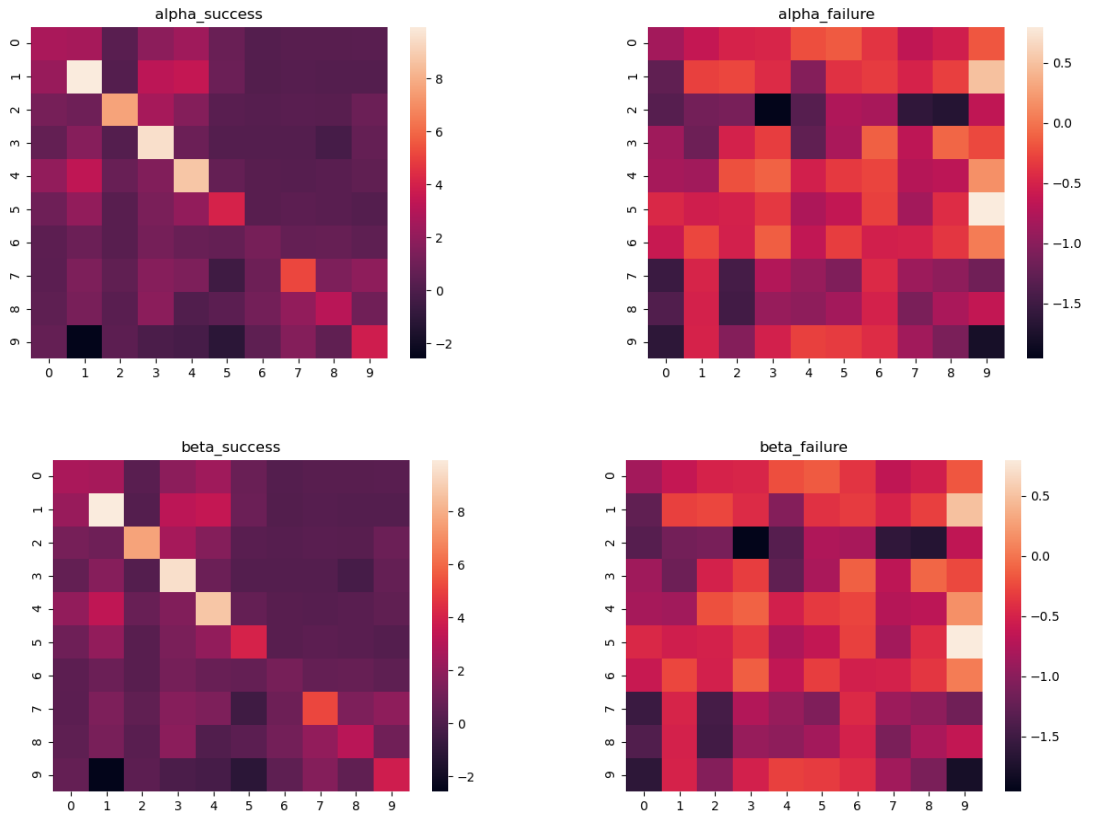


Figure 1.17 – Cross effect matrices on a small subset of 10 most done exercises in different Knowledge Component. Colors represent the α and β values, higher values mean higher cross-effects between exercises.

Results are reported on Figure 1.17. We see that the diagonal with success parameters for α and β are really marked showing that there are few cross-effects between exercises. How-

ever, the failure parameters look more like noise and we will not be able to extract Knowledge Component with these parameters. To our surprise the β parameters could be used to detect Knowledge Component, we thought that time decay was not valuable information in our task.

Before trying on a whole subset of the dataset, we will look at the cross-effect matrices for the exercises inside a single Knowledge Component. We will still look at both cross-effect matrices to see if the results are the same as in the previous experiment regarding the success/failure difference. We repeated this experiment for several Knowledge Component in the *ASSISTments12* dataset but only a Knowledge Component with really few exercises (around 400) will be shown in this manuscript for readability purposes. The figures should be able to be zoomed on with the online version of this manuscript.

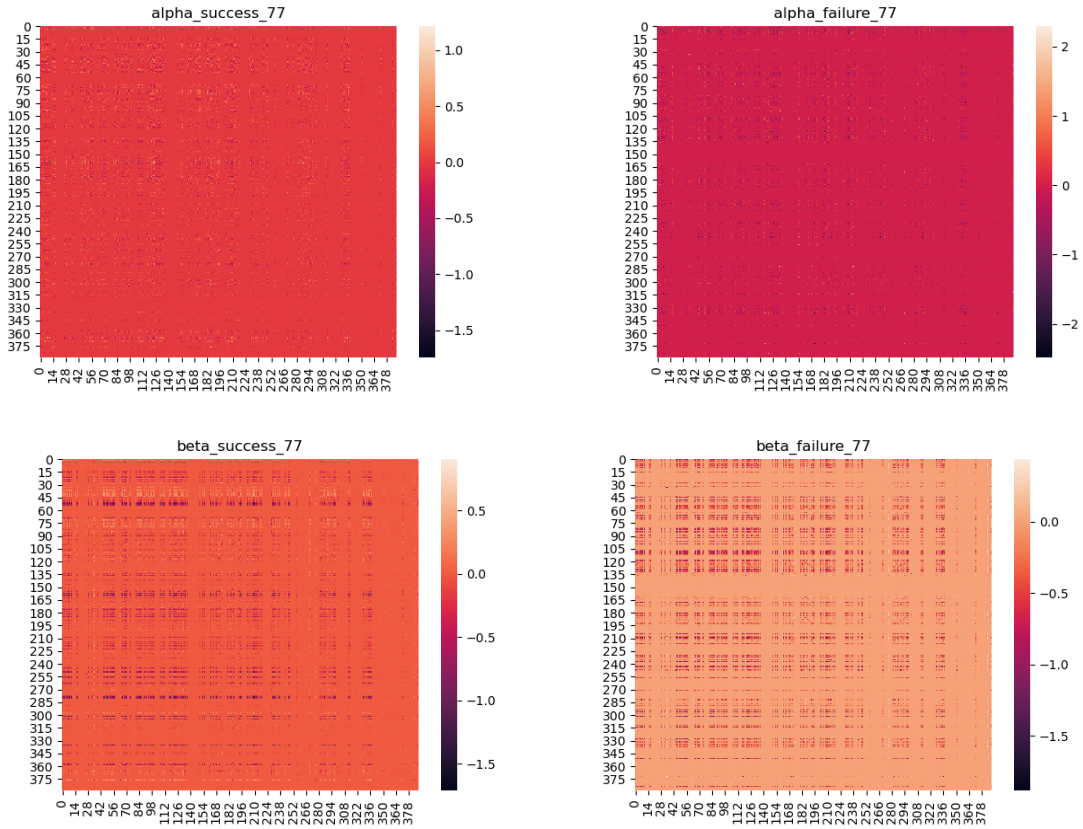


Figure 1.18 – Cross effect matrices on exercises inside the same Knowledge Component (here KC number 77). Each pixel represents the cross-effect between two exercises.

Results are reported on Figure 1.18. We see that cross-effect matrices are quite uniform which is a good sign inside the same Knowledge Component. More artifacts can be seen with the β values cross-effect matrix. Similar conclusions can be taken on the other Knowledge Component that are not shown here.

When trying to detect Knowledge Component in the whole set of exercises, we will look at the success part of the α cross-effect matrix after observing the results of these two experiments. After training the model, a Knowledge Component would be a subset of exercises with a high cross-effect between each other. We look for an order of exercise where the cross-effect

matrices have some high values on blocks on the diagonal.

Finding such an order is actually an interesting problem but we use a naive algorithm to test our method. The algorithm looks like an insertion sort.

1. We start at exercise index 0
2. For the current exercise we sort each exercise with a higher index by decreasing cross-effect with the current exercise.
3. We keep going with the next exercise and go back to step 2.

This algorithm is really simple to implement and should be able to build a Knowledge Component from an exercise in this Knowledge Component. But it may need a threshold value to decide to start a new Knowledge Component and do not sort of exercises that have a low cross-effect. We could also add a preprocessing phase after raw blocks are fine to check if each exercise is in the block where it has the most similarities.

Visual results are too big to be shown in this manuscript. Anyway, we could not extract any satisfactory subset of exercise. This is quite disappointing given how encouraging preliminary results were and how we believed we should still be able to extract at least the most visible Knowledge Components from the dataset. Some more experiments could be done by using a smaller dataset with less Knowledge Component or trying on other datasets like the one in Section 1.2.

Conclusion on the recommendation in learning environment

Some interesting tasks can still benefit from the use of statistical methods in a learning context while still avoiding knowledge from learning theories in psychology. However, in two different tasks, we could not find methods to satisfactory solve them. The methods we tried required a lot of engineering, we do not know yet if the failures are due to actual problems in the methods or wrong engineering.

Conclusion

In this chapter, we explored several statistical methods to perform a recommendation in a learning context.

We saw that data quality is necessary to design a system. We also should not be afraid to adapt a system to a specific field and this lack of genericity can in fact greatly increase the knowledge we have of the data.

Complex models find a lot of limitations in a learning environment. The knowledge gathered of the data is of great help in designing a recommending system with a simple algorithm tailored for a single field.

In the future, and in particular after we have a set of debugging exercises, we will try to avoid the need for using a recommending system and we will use other metrics to know if some more exercises are needed. We could add some recommendation features in the future but, this will stay in a basic form. We will start thinking about adding it after some solid knowledge of students' usage of our system is gained.

Chapter 2

Debugging exercises generation

In the previous chapter, we studied exercise recommendations even though we do not have a database of debugging exercises yet. We call a debug exercise a buggy program and its specification where the student's task is to fix the bug. As to our knowledge, there is no debugging exercise database, the goal of this chapter is to explore different ways of automating the generation of debugging exercises.

The first task to generate a debug exercises is to generate a buggy program. Generic automatic program generation is already a difficult task. Thus, generating good debug exercises directly will be difficult, however some recent works start to obtain satisfactory results on really small subsets like bash script generations from a textual specification [49].

Each section in this chapter will be relatively independent so each of them will present their background work.

2.1 Generating random programs

A natural idea to try to generate programs is to try to firstly generate random programs. Random programs do not look human-written so this should be a problem regarding debugging teaching. They would rarely arise in real-life scenarios. We will nonetheless study this to gain understanding about program generation.

In this section, we will only work on the random generation task. Another important task to allow us to use our programs in a teaching scenario would be to find a way to select programs of interest. This task will not be formally addressed but we will consider the quality of our generation regarding the ability to write such a selection algorithm. If we think it would be hard to discriminate an interesting program from a program to discard, we will consider the method unsatisfactory. So our goal in this section is to generate correct random programs and discuss whether some of them could be used in a teaching scenario.

2.1.1 Background on random program generation

Random program generation is already a research field of interest used to test compilers [50, 54]. They randomly generate AST nodes based on random semantic such as "input," "output" and "work" variables and then random expressions and statements using these variables. The generated programs are usually special enough to enter specification corner cases and test the compiler behavior on unusual programs.

This can be a problem in our case to have purely unusual programs because the associated specification of the program to describe the correct behavior can be long and hard to understand. The random generation can be biased in many ways to achieve certain goals, we will study some of them in this section.

From the testing community, constraints are derived from the program specification [51]. Then solvers are used to generate AST nodes such as a part of the specification is actually tested in the generated program. Instead of using solvers, deep learning can be used to bias generation provided enough training data [54, 55] (see Subsection 2.1.3 and Subsection 2.3.1).

2.1.2 Sampling in language grammar to generate *Compute-IT* programs

It is easy to ensure the correctness of a generated program if we sample tokens in grammar rules. As we use the grammar rules in the generation, we know that the program will at least be syntactically correct. To evaluate this simple generation method, we will choose a really simple grammar. We will use the grammar of *Compute-IT* programs from the previous chapter.

There are only a few rules and constructs in this simple grammar:

— **basic statements** left, right, up and down statements and function calls.

— **code blocks** a list of statements and control flow.

— **repeat loops** they have two parameters, the number of repetitions and a code block as a body.

— **conditionals** the conditions can only be applied to colors under the pebble. They have only four parameters: color, whether the condition is negated or not and two code blocks for the branches (the else clause can be empty).

— **while loops** they have three parameters, the two from a condition and a loop body.

— **function definitions** there are no function arguments so they only have two parameters, a name and a code block as the function body.

We first generate the number of functions for the generated program. We use placeholder names for functions, we call them `f1`, `f2`, etc. Once the program is generated functions could be renamed if needed.

To generate a basic block we randomly choose the number of statements and control flow. This generation has to be constrained otherwise it can result in an infinite loop. This is constrained by pondering between the different rules and generating more often single statements because they are AST leaves.

Budgeting the generation A method we tried to have more control over program length is to use a cost function for each rule and assign a random finite budget from the start. This is an approximation of complexity. We can also use this cost function to forbid some grammar rules by assigning an arbitrarily high cost.

We generate programs recursively with this decreasing budget, each rule can take different options and decrease the budget accordingly. For the conditional rule that can increase the number of branches, we first randomly split the budget between the two branches. Otherwise,

the positive branch may take all the allocated budget and force the else branch to be a single statement. The first statement in each code block costs no budget and only adding more statements to a code block costs some budget.

```
repeat(8) {
    up()
    down()
    down()
}
repeat(5) {
    right()
}
if GREEN {
    left()
    left()
    down()
    down()
}
```

```
if NOT GREEN {
    down()
}
if NOT GREEN {
    left()
}
if BLUE {
    down()
    down()
    up()
}
if NOT PURPLE {
    down()
    right()
} else {
    up()
    down()
}
```

Figure 2.1 – Two generated programs with a budget of 5. A budget of 5 is not enough to generate two or more functions and include function calls.

On Figure 2.1, we can see two examples of generated programs with this method. We print the AST using the *Compute-IT* toy language turned into pseudo-code. However, we do not know how to choose interesting programs yet. We think we can have better hints about this if we try to generate a map with a program. For example, with the two examples on Figure 2.1 it is hard to know which one is the more interesting because there is not map attached to it. It will be easier to judge a pair of a program and a map for *Compute-IT*. If it is hard to generate a *Compute-IT* map for the program, then we should discard it. This will be explored in Section 2.4.

The two generated program examples nearly look usable, but this is much less true with higher budgets and function calls. We can see such a program on Figure 2.2. We see on lines 2 and 3 redundant conditions. We could also include verification of dead code and redundant conditions during generation but this requires a lot more work [52]. If it happens to be needed, we will check and remove them statically afterward. Function calls can easily lead to infinite loops in program execution. As program termination is an undecidable problem, it is again suitable to bound the execution time after generating inputs and running the program. However, finding good program inputs (in our case *Compute-IT* maps) for an exercise may be an interesting problem and good program inputs can simplify the generation tasks. In our case as the program inputs are complex, we will better generate it afterward using the generated program. If this proves to be too hard, this grammar sampling method would be unsuited even for simple programs that include only non-recursive function calls.

```
function f1() {
    if NOT PURPLE {
        if NOT PURPLE {
            right()
            up()
        }
        if NOT PURPLE {
            right()
        } else {
            down()
        }
    }
    if BLUE {
        up()
    }
}

f1()
f1()
if GREEN {
    up()
}
f1()
if NOT RED {
    if PURPLE {
        left()
    } else {
        f1()
        f1()
    }
}
```

Figure 2.2 – A program with a budget of 10 and function calls.

2.1.3 Generation with LSTM models

Long Short-Term Memory neural networks (LSTM) has already proven many times its usefulness in generation tasks [56]. They are essentially a recurrent neural network with a more elaborated recurrent cell with some learnable forget functions so information can be passed between nodes far away in the generated sequence while the network remains trainable. Following some previous work to generate random programs with LSTM models [54], in this section, we will try to generate random programs using LSTM models. The authors generate random OpenCL kernels used to test compilers. The model is trained on a database created from downloading open-source OpenCL code from Github. It is worth noting that their model does not work at the token level but at the character level. The model is thus quite consequent, they use a 3-layer LSTM with 2048 nodes per layer, totalizing 17 million parameters.

There are two main differences in our case. The first one is the availability of training data. In our small subproblems of generating *Compute-IT* exercises, we only have about several

dozens of problems. We cannot think about training any statistical model with this little data. We will leave for a moment this task and try to generate some C loop bodies. Many open-source C programs can be found on Github so we can gather a lot of training data. Moreover, if we can generate C programs, we can later translate them into *Compute-IT* programs statically. However, C programs feature variables which *Compute-IT* programs do not. We will have to remove declarations and transform assignations into *Compute-IT* basic statements. However, we can keep the control flow generated by such a model.

The second main difference is that the authors need to generate dummy variable names for OpenCL. As we will remove the variables from the C code we generate, we do not need to be able to generate variable names. We will name all variables with the same token <*var*>, so we will generate expressions like <var> = <var> + 2*<var>. Our model can thus be at the token level and not at the character level. This allows us to have a much smaller model with fewer parameters, we use a single-layer LSTM with 2048 hidden nodes.

Experimental setup We realized that downloading all open-source C code on Github is intractable with our hardware (including access to computing clusters that we use to train machine learning models in all this thesis). We only downloaded the Linux kernel that we think is reasonably big and contains enough loop bodies. The downloaded code contained more than 10 million lines of code.

To extract loop bodies, we used an external tool in a polyhedral compiler *PoCC* [53]. As polyhedral compilation focuses on loop bodies this compiler can easily extract them. We extracted about 12,000 loops from the Linux kernel. The tool also renames variables so it is easier to match them and replace them with a dedicated token. However, nearly half of the loop bodies contain only a single line. We decided to keep the variable names, if we wanted to translate it to *Compute-IT* programs, we would use this information. To keep our token-level model we assign unique tokens to the few normalized variable names present in the extracted loops.

We can see on Figure 2.3 examples of loop bodies extracted and transformed from our dataset. We observe on the transformed versions of the loop that as variable names are only single characters there may be only small differences between token-level and character-level models.

The experimental results are quite disappointing. Even if the training loss of the model seems to go down and suggests convergence. The token to mark the end of the generation is almost never generated whereas programs generated by the original work are of descent length. This makes the generated programs absurdly long compared to the training data. We do not think this is related to the token-level instead of the character-level model because the end token is correctly placed in our training data. We spent some time investigating this issue but increasing and decreasing the model size did not solve this issue. We finally dropped this method because we thought that even if we keep small programs, they are too rare and we do not see apparent reasons to explain this so we did not want to keep going on a track we do not understand.

Conclusion on random program generation

We do not have yet any satisfactory method to generate random programs that could be easily transformed into exercises. Our goal was only to first generate some programs that we

```
// Example one
for (a = 0; a < d; a++) {
  if (a == 0) {
    A(b, "  ");
  } else if ((a > 0) && ((a % 16) == 0)) {
    A(b, "\n  ");
  }
  A(b, " %02x", c[a]);
}

// Example two
for (d = 0; d < a; d++)
{
  c[d].g = e;
  e += (c[d].b - c[d].f) + 1;
}

// Example three with single statement body
for (a = 0; a < c; a++)
{
  A(d[a].b);
}
```

Figure 2.3 – Three loop body examples. They will be tokenized before training our LSTM model.

could transform to *Compute-IT* programs, then we would generate a map. However, the fact that even random program generation is a difficult task leads us to try other ways of exercise generation than program generation. We will not study program generation that would be derived from an exercise specification like automatic program synthesis would do from a functional specification. This will require an expertly made specification of the learning objectives. We think we must first use these learning objectives in handcrafted exercises before trying to automatically derive exercises. We also saw that input generation is a task that we could try to generate interesting exercises by generating maps for *Compute-IT* programs (see Section 2.4).

2.2 Program mutation to introduce bugs

We saw that the program quality of our generation methods is quite poor. In this section, we will try to improve the program quality by taking existing programs and mutating them to introduce bugs so as to make a debugging exercise.

Program mutation is already a topic in the automatic bug fixing community [57]. Automatic bug fixing needs to mutate the program to solve bugs. We will see whether some of the mutating methods are easily reversible to introduce bugs instead of fixing them.

2.2.1 Background on automatic bug fixing

We do not consider the semantic-base bibliography because they rely on formal specifications and satisfiability solvers. We saw in the previous chapter that even for simple prediction it is hard to translate it to an actual learning benefit. We think expressing learning goals in terms of satisfiable constraints is not reasonable. We will look at recent machine-learning models for automatic bug fixing and the more classical exploration approaches.

Most of the recent works include complex machine-learning models. Some models are imported from the language natural processing community and treat the bug-fixing task as a translation task [58]. The task, however, has to be split into two subproblems, locating the bug and fixing the bug. Both of them are achieved with dedicated machine-learning models [58, 59]. The part that would interest us is the bug-fixing part. In both works [58, 59], they use generative models: LSTM [58] or a Generative Adversarial Network (GAN) [59].

We saw in the previous section that unguided generation with machine learning models is not an easy task and requires at least complex models and huge datasets. We should not be able to deploy and correctly train such massive models as we did not manage to do it for simple LSTM models. However, many older works on automatic bugs repair feature programs mutation and exploration to fix bugs [60, 61] [1]. Nakamura and Ishiura are using equivalent mutation (program mutation preserving program equivalence) and random sampling in a set of mutations. They want to keep equivalence because this precise work focuses on finding compiler bugs so they want to generate test cases for compilers. The program quality for teaching they obtain is close to the one we had using our random sampling technique. This is due to teaching not being their generation goal but compiler testing. However, Weimer et al. use more elaborated genetic algorithms to explore possible mutations. They also want the mutations to not keep equivalence as they want to fix a bug. To keep or reject program candidates they have a set of passing/failing tests before mutation. A program is accepted if at least one new test is passing instead of failing and all previously passing tests are still passing. The test suite is given as an input to their algorithm. Their method could be adapted for our mutation case provided we have these tests. We could imagine automatically deriving these tests from a set of knowledge components we want the exercise to have. But again, even using success prediction for a learning recommendation is hard. Deriving a test suite from knowledge components, then generating a debugging exercise that will be interesting as a learning material should be too complex overhaul.

Conclusion on program mutation

Program mutation looked promising to generate debug exercises. Although we did not perform any experiments, we believe this should increase the program quality from the previous section. Some state-of-the-art methods look nearly applicable to our case so we could experimentally try them and adapt them to our debugging exercise generation task.

We could put an arbitrary amount of engineering and research work into having a satisfactory generation method. However, it will not solve a major problem we had in the previous section being the fact that we cannot adapt our generation to learning needs and have a good pedagogical quality for our exercises. Knowledge tracing models studied in the previous

1. Nakamura and Ishiura are mutating programs for pure program generation and target compiler testing and not bug fixing.

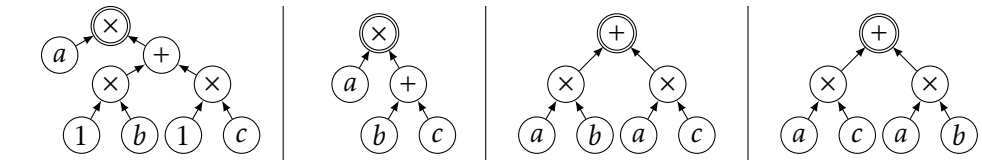


Figure 2.4 – Examples of Computations. The four AST are equivalent.

chapter are designed to predict an exercise outcome. The learning modeling is not designed to be the input of a generation task.

2.3 Checking program equivalence

The mutations we want to apply in the previous section are supposed to introduce bugs. They would not keep program equivalence, ie the mutated program will have different outputs for certain inputs. Program equivalence is useful because we can know which of the mutations we applied introduced bugs, meaning that they broke program equivalence. We can also discard the exercise if the mutated program is equivalent to the original one. In another context than program generation, we can use program equivalence as a learning tool to notify students if their solution is equivalent to the original program.

The program equivalence is studied by the compiler research community. It can be used to validate compiler optimizations or program synthesis, for example. While it is formally undecidable, in practice, semi-algorithms are used [62, 63]. They rely on solvers and static evaluation. Exploring the search space can be intractable with real-life programs but it should not be the case with our debugging exercises. In the next section, we will see that it is hard to perform equivalence checking even in our short program case and why we finally dropped these methods.

Choice of the equivalence checking task We could focus on the *Compute-IT* grammar for equivalence as we already have a basic random generator. However, the choice of the equivalence checking task was made for a more pragmatic reason. The work in this section was made chronologically before all other works including machine learning. It is a collaboration with a team at Colorado State University which is already proficient in machine learning research so we could gain some expertise. Because of the unsatisfactory results for our teaching application we will not apply it to *Compute-IT* grammar.

The team works in the compiler optimizations field and is therefore interested in program equivalence [64]. They target linear algebra programs. To keep the programs simple in a first stage, they focus on a subset of their target language with only scalar values (no vectors, matrices or tensors). Example ASTs can be seen on Figure 2.4.

Their goal is to train a Graph Neural Network model to find a sequence of transformations that preserves equivalence between two programs. If the model finds a correct sequence, it means the two programs are equivalent. It is easy to apply this sequence of transformations to a program and check if the output is indeed the other program. If the system outputs a wrong sequence, we can statically verify it and label the pair as not equivalent. Thus there cannot be false positive pairs, only false negative as false positive are quickly verifiable.

We did not contribute to this Graph Neural Network model but we designed a baseline model for equivalence checking so it can be compared to our simpler model. We first train an autoencoder to reproduce input programs, this gives us a program embedding. A program is given as input to our model and the goal of the model is only to reproduce it as output. However, there is a constraint that the program must be embedded in a 128-bit code. The embedding size was chosen after several experiments and is tied to the input programs length. The program is encoded into this vector by the first half of the model, the "encoder". Then the vector is decoded into a program by the second half of the model, the "decoder". Once this autoencoder is trained, we keep only the encoder part and we train a simple 2-class feed-forward classifier on the program embeddings to check equivalence. The autoencoder is composed of an LSTM layer [46] followed by some linear layers. Our model is not able to provide a transformation sequence, it can only classify equivalence.

Experimental results Programs are randomly generated as a sequence that can be encoded by the LSTM layer. It is a similar method than in Subsection 2.1.2 with random sampling in the language grammar. We also apply transformations to generate output trees and have a sequence of transformations that leads to equivalent trees.

The experimental results of our simple model are reported in [64]. The accuracy of 73% is much less than the graph neural network model whose accuracy is above 90%. This is expected as the Graph Neural Network model is much more complex than our simple feed-forward model. However the Graph Neural Network requires a huge amount of data and computing power for training.

If we would use such a model for our mutation problem, we would need to look more at the precision of the model rather than its accuracy. In our case, it is acceptable to have false positive results. It would mean that the programs are still believed as equivalent so we should generate more mutations to introduce a bug. We would have more mutations that needed. It would, however, be a problem if we wrongly label two trees as not equivalent so we believed a bug is introduced whereas the program is correct.

2.3.1 Generation with autoencoder models

Once we trained our autoencoder to encode and decode program embeddings. We had the idea to go back to our generation task and use this time only the decoder part of the model. With a random embedding as input, it could decode it and generate a program. Such a simple process did not provide any satisfactory results. Actually, raw autoencoders are not suited for a generation task. Variational Autoencoders are more suited to this task and can model the output distribution so we can sample from it. Compared to regular autoencoders, the embedding between the encoder and decoder parts represents a latent space with a distance measure. This way a small variation in embedding should result in a small variation in the decoded output. This method was tried in [55]. The Variational Autoencoders is guided by grammar rules so it generates syntactically correct programs. However, the generated programs are not really human readable and should not be a good base for debugging exercise.

Conclusion on equivalence checking

We did not make significant contributions to the equivalence-checking task. We can argue that our joint work with Kommrusch et al. however is a contribution to the equivalence-

checking task but we are not the main contributors to this research. If we actually need equivalence checking, we will use state-of-the-art equivalence checking.

Pragmatically for this thesis, the main contribution of this work is for us to gain some skills in doing machine learning research and engineering by working with proficient researchers in this field.

2.4 Generating inputs for *Compute-IT* programs

We saw in Section 2.1 that generating inputs for a program might help to understand if the generated program is a valid candidate for the generation task, i.e., if it can be used as a basis for a debugging exercise. In this section, we will investigate this claim while trying to generate color maps for fixed *Compute-IT* programs. We will use the simple random generator that samples into *Compute-IT* grammar. An input for such a program is quite simple, the only input is the color of each circle on the map. We will pick the size of the map at random and only generate colors.

We developed a small *Compute-IT* interpreter. It takes a color map as input and an AST generated randomly as in Subsection 2.1.2. It also has a "step-by-step" mode used for input generation. In this mode, it does not take a color map as input and needs external color input to resolve conditions. This external color input will be provided by the input generation algorithms we will describe in this section.

2.4.1 Color constrain maps

The first component of our input generation algorithm is a map keeping track of color constraints depending on the program branches. When a branch is encountered, the interpreter takes a branch at random and add a constraint on the map. For example, if a condition is `if color != blue` and the positive branch is taken the color constraint is that the color under the pebble must be different from blue to match the current execution. It makes no decision regarding input generation. Its goal is to keep track of color constraints and trigger an exception when constraints are unsatisfiable, for example the same circle must be blue and green at the same time because during execution two branches adding this constraint were taken when the pebble was on the same circle. We can then reject this input candidate because no input would match the execution trace from the interpreter.

Constrains encoding The map contains a list of possible colors for each possible emplacement. When the generation algorithm encounters a condition during program interpretation (a loop or condition test), it will forward this condition and the branch that will be explored to our constraints map. The map keeps track of the pebble position and forwards the condition only to the according constraints. We call positive and negative branches the then/else conditional branches and staying/exiting loops respectively. The constrain encoding is quite straightforward, we hold a set of possible colors. We only have to negate and intersect color sets and return an exception when the set is empty. Since our color set is small (6 colors), we evaluate conditions on each color to know which one will be in the positive/negative branches. We do this instead of solving it from the condition expression that can become complicated if logical operators are used. A python-like pseudo-code implementation of this algorithm is shown on Figure 2.5.

```
1  # color_set is a list of possible colors for the given map position
2  # It is a boolean list, at initialization every color is possible at each position.
3  def apply_branch(color_set: list[bool], condition: ASTCondition, branch: bool):
4      branch_set = compute_possible_color(condition)
5      if branch == False:
6          negate(branch_set)
7
8      color_set = intersect(color_set, branch_set)
9
10     if is_empty(color_set):
11         raise InvalidContraintError()
12
13     return color_set
14
15 def compute_possible_color(condition: ASTCondition):
16     # as our color set is reduced we can evaluate the condition with each color
17     color_set = list(size=N_COLORS)
18     for color in range(N_COLORS):
19         # the evaluation function is provided by our interpreter
20         color_set[color] = condition.evaluate(color)
21     return color_set
```

Figure 2.5 – A pseudo-code implementation of the color constraints encoding. The map maintains a color set for each position and keeps track of the pebble position to update the according color set.

The set-related functions `negate` and `intersect` on line 6 and 8 are simple element-wise not and and operators respectively. The `is-empty` function tests if no inputs are possible at the cursor position to obtain the wanted execution trace.

Once all the constraints are computed and the program is fully interpreted, we can sample in the available color set for each position on the map to provide an input that will match the last execution trace.

2.4.2 Program trace enumeration

As we work on small programs where the trace should be humanly enumerable, we set a trace length limit of 30 instructions and exhaustively explore all possible branches inside that trace length limit. Thanks to the constraint map we can catch execution traces with no possible matching inputs. The exploration is a binary tree where a node corresponds to a conditional (in a loop or simple conditional statement) with positive and negative branches. In the case of a condition with no else clause, we can just consider the else branch as an empty block. Each node keeps a copy of the constraints map so it can be easily backtracked if the color set becomes empty and the execution trace is unsatisfiable.

This algorithm efficiency might be improved but it is really simple to implement. As the generated programs are quite small and we limit ourselves to short traces, the exploration time is less than a second for a single program. It can happen for rare generated programs that no inputs are possible if we want a trace shorter than 30. Our input generation procedure

then acts as a filter to reject these programs and generate new ones.

2.4.3 Deriving a difficulty measure from execution trace

The *Compute-IT* interpreter returns the trace of the successive moves on the map. This execution trace is useful to rate the difficulty of a program/map pair. We use the length of the trace as a refinement to the difficulty derived from the budget we use during AST generation in Subsection 2.1.2. However, it is natural that the length of execution is not all responsible for the exercise difficulty. Repeating patterns, for example, are not difficult but have a lengthy trace. We would need to take more time studying the execution trace to describe what are difficult traces compared to others provided we already have an inherent complexity given from AST generation. However, this simple length criterion can already help us to reject programs that take too long to execute whereas the program code could be short.

Conclusion on input generation

In this section, we presented a simple algorithm to generate *Compute-IT* maps tied to previously generated programs. These program/map pairs cannot form debugging exercises but we still gained some knowledge from this. If we would like to use the generated program/map pairs, we would need to adapt the color sampling at the end of the map generation to have groups of color to look like the existing *Compute-IT* exercises. This way, maybe we could generate at least *Compute-IT* exercises.

Even if we can generate correct maps for a given program, the program quality and map pedagogical quality are inferior to handcrafted program/map pairs by the *Toxicode* team. This is not a surprise as it is not easy to create an exercise difficulty measure and derive learning objectives for an exercise from a set of knowledge components.

However, we saw that generating inputs helped us to reject some programs during the generation and indeed can help in the program generation task. Even if the program generation and the input generation are quite simple procedures, we think that generating both program and input in a single procedure is not realistic as the procedure would get much more complex.

Conclusion

This work in this chapter was quite exploratory. We saw that simple sampling generation algorithms are not sufficient to obtain good-quality programs. However, even with more elaborate methods, we would still have a problem. The generated exercises would not be tied to any pedagogical concepts. We could imagine generating a huge number of exercises and labeling them with knowledge components. But we saw in the previous chapter in Subsection 1.4.1, that the automatic discovery of knowledge components is not an easy task.

In this chapter and the previous one, our goal was to both generate debugging exercises and recommend some debugging exercises based on an evaluation of students learning. We saw that both of these tasks are too challenging mainly due to the fact that it is hard to model student debugging knowledge in a way that is usable for the generation and recommendation algorithms. With the knowledge we gained from this chapter and the previous one, we will radically change our direction to teach debugging. Instead of generating an exercise database

and recommending exercises inside this database, we will manually design a pedagogical sequence. The expertise would be put into carefully designing exercises with specific concepts instead of engineering and training statistical models. This way, we don't have the problem of not being able to properly model student debugging knowledge. But, we have to ensure that the difficulty gap between exercises is not too high, as what was done in *Compute-IT* in the previous chapter.

Chapter 3

Agdbentures

We want to provide some debugging exercises so students can learn debugging as is and not as a "side-product" of lab sessions. Students must be able to use Agdbentures on their own so they can work from home with only minimum teacher intervention. In this chapter, we will add ludic aspects to the debugging exercises to increase student engagment and increase autonomy. Our first goal is to study if we manage to add these ludic elements and that students enjoy using Agdbentures.

From the previous chapters, we concluded that a good way to obtain debugging exercises is to manually design them. This is not a problem to have only several exercises in the range of a few dozen as we also saw in the recommendation chapter that carefully designed exercises in a linear order may be sufficient. So in this chapter, we will apply both of these conclusions and design debugging exercises that are intended to be done in a linear order.

3.1 Motivations

One difficulty when creating debugging exercises is that novices are much more proficient to debug their own code (this is also true for experts, but this effect is less important [75]). In existing debug practicing systems, this led to the creation of only small programs, with the drawback that there are not so many places where the bugs might be, and students are not trained in how to find the location bugs in medium to large projects. We believe it is important to provide an environment that matches more closely what students will be confronted to in their learning and work experience.

In the previous chapters, we targeted learning in autonomy. We will keep this use case in this chapter. We want to keep students engaged so it is easier for them to work on their own, provided the exercise difficulty is well designed. We work on the addition of many ludic aspects to our series of debugging exercises.

For these reasons, we developed *Agdbentures* (pronounced a-GD-bentures), a debug practicing game for the C programming language, with the following features:

— Shows the importance of being able to observe the program state: this is provided by having a visual representation of the program being debugged.

— Is fun to play: the exercises are presented as "levels" of a video game, which can only be "played" correctly when bugs are fixed. The bugs themselves are also chosen so that the buggy behavior is interesting or amusing.

— Each level reuses and extends the source code from previous levels. Over time, the students become familiar with a large codebase of foreign code and can be trained to search for bugs in "real-life-sized" programs.

— Teaches how to use a debugger: Agdbentures is based on GDB, the GNU Debugger. Commands are presented and made available to the students as they progress in the game.

— Presents the same common bugs multiple times in different contexts, so students can recognize them in the future. There are many common bugs in novice code [76, 77], this will be detailed in Section 3.3.

We chose the C programming language because we already had a target audience in our class of CS1 students whose application language in many courses is C. Some of them participated in a first experimentation phase of Agdbentures (see Section 3.8).

3.2 Background on debugging courses

There are already several debug practicing systems. The oldest we found is *DebugIt* [78]. It includes 20 debugging problems on small programs that are selected in a predefined order. They are based on common novice bugs, but there is no progression in the difficulty of the bugs. It is a plain debug practicing system and does not include any type of visualization.

Ladebug [79] separates the task of finding and fixing the error. When an exercise is given, the student must first locate the buggy line. Once identified, the student is allowed to modify the code to fix the bug. This is motivated by explicit teaching of debug strategy and the fact that students may add errors while searching for bug sources. *Ladebug* is based on *PythonTutor* [90] backend, so it supports reverse debugging (stepping "back" on previous lines of code). However, only short programs are presented to the student. Separating finding and fixing a bug are interesting but as we try to show a method that can be applied to future projects we may only do the same as a scaffolding method or even not at all.

Gidget [80] is an online debugging game for computer science learning in general. However, they introduce programming concepts by debugging, learners need to fix programs first. After all these debugging exercises, the exercise now requires programming skills instead of debugging. It is designed as a game and gathers statistics about learners' engagement.

Debugging is also sometimes explicitly taught in introductory programming courses [81, 82, 83]. They consist of plenary sessions to explain debugging strategies and some practice on exercises. The plenary sessions are a valuable addition as we can ensure that we convey the most important elements and they are correctly understood by students. However, we will have to do without it as we focus on autonomy so Agdbentures should be able to be played at home. We may imagine some video recording that can be watched at some point but in the beginning, we will try to convey every external information with written instructions in each debugging exercise.

Using games to teach CS concepts is already known to be more engaging to students [73], especially in introductory programming courses [68]. Interestingly, game programming can also be used as material to teach introductory programming, in high school [71, 72] and in CS1 [70] to increase engagement. Regarding debug teaching specifically, game codes have

also been tried for a high school audience [74, 72]. These works comfort us in developing a ludic environment and using basic video game source code as exercise material.

3.3 Novice bugs and debugging methodology

3.3.1 Debugging methods

Several works already try to define the different skills involved in debugging [84, 81].

From these works, we will target four main categories, their skillsets are wider than the one we present here but we focus on the skills that we believe can be explicitly taught:

— *Understanding the program specification and architecture*. Before trying to find some bugs, it is mandatory to understand what the expected behavior of the program is. Then depending on the difficulty of the bug, the programmer needs to be familiar with the program architecture and the role of the different functions. Some bugs may require only an approximate knowledge of the different modules, but some may require a great expertise about several internal functions. The understanding of the program expected behavior can actually be applied at the function level. It is already a big step to find a bug when a programmer has identified the buggy function. This is also true for a variable role. If a broken invariant is found about a variable, it is then much easier to only track this variable state until the invariant is broken.

  The goal of a programmer at this point is to find an observation during the program execution that mismatches with the expected behavior.

— *Formulate hypotheses on the bug source and verify them*. Once an unexpected behavior is found, the cause of this bug may not be obvious yet. From the program understanding, a programmer needs to formulate hypotheses on the bug source. It is not an easy task to verify these hypotheses to confirm the bug source. It is hard to be sure of a bug source by simply reading the code and not testing a hypothesis. Usually, several hypotheses are made and the most probable are tested first.

  It happens that after all hypotheses have been tested, a programmer has not found the source of the bug yet despite observing some incorrect behaviors. In that case, it often means it is time to take a break or ask for someone's help. There are also some popular methods like the rubber duck debugging method where we explain our reasoning to an imaginary duck to proofread it. Some programming methods like peer programming are also trying to limit the number of times when this kind of debugging dead-end happens.

— *Program execution tracing*. This is the task of observing different execution points of the program in order to validate or not previously made hypotheses. These methods can also be used simply to observe unexpected behaviors. The difficulty of this task is to decide when to observe the program state and what to observe. This is derived from the bug source hypotheses made previously.

  Different methods can be used to trace program execution. The most basic one is just reading the source code and trying to figure out what will happen. This mental simulation for example is typically the kind of task that relies heavily on the programmer's notional machine.

However, as we already said in this section the mental simulation of the source code should not be appropriate for many bugs that go beyond the "typo" level. The program state has to be observed to find incorrect behaviors. Probably the most used tracing method is printing variable values or tokens to track branching. This is an intuitive method that is often done even when a programmer is not familiar with the debugging methodology. Some teachers do not consider printing values as a valid debugging method or a failure to apply "correct" debugging methods. However, several professional programmers use printing value as an initial debugging method and only use a debugger if printing fails to find the bug. We also believe that printing is a valid debugging method and can be appropriate in certain use cases.

Another common method is commenting out some parts of the source code to find for example where certain functions are called or what causes an infinite loop. A risk with this method is finding a false-positive bug and adding a new bug while trying to fix the first one.

Lastly, some tools can be used to aid debugging. They can track memory management like *Valgrind* or be plain debuggers like *GDB* that can control program execution and observe the memory. Even if most of these tools are powerful, they need to be explicitly learned before being used in real-case scenarios.

— *Fixing a bug once it is found*. This is usually closer to actual programming. It is, however, still part of the debugging process because one has to be careful not to introduce new bugs while doing this.

All this becomes much harder when several bugs are present. In this case, a bug can be fixed but the program is still incorrect until the last bug is fixed. There can, however, be independent bugs, in this case, it is the same as fixing single bugs. This can happen for example when a bug is observable only when the first one is fixed.

We will not cover methods to find hypotheses like peer programming or rubber duck debugging in Agdbentures. As we consider printing as a valid debugging strategy, it will also be the main debugging method in the first exercises and we will progressively try to teach the usage of GDB.

3.3.2 Background on novice bugs

There are already several bug classifications [77], these generic classifications do not target teaching specifically. They are used as a basis for bug classification software or making awareness among programmers so they can more often avoid them beforehand. We are more interested in classifications for novice bugs [81], this time the focus is direct teaching and analysis of misconceptions that cause these bugs.

In this section, we will simply make a list of bugs that we extracted from these reports (and some additions that we did not see there). This will be a base for the level development where we will pick one or several bugs in the list and try to design a level around it. So some bug ideas may look very specific because we already thought about some levels around it. We also try to give some abstract level ideas when possible for generic bugs. When we will have finished and tested sections of Agdbentures with students, we can come back to this list and see how many types of bugs we covered.

Language-agnostic bugs

— *off-by-one errors*. These are ubiquitous errors that can happen any time a bound is involved. It can be a loop bound, some data structure traversal, or a bound in a recursive terminal condition, for example.

— *arithmetic*. This can be a wrong score computation in a mini-game. A minimum score could be needed to reach the exit but, this score is impossible to obtain with the buggy computation.

— *linked list*. We lose the rest of the list after inserting at the head position. We can have an infinite loop during searching if the last cell loops back inside the list.

— *random number generation*. A randomly generated sequence is not correctly seeded or is too short and is repeated during the program execution. We can consider that fixing a seed to make a program deterministic and facilitate debugging is a debug skill. We did not plan any level that requires this yet even if it could be interesting.

— *classic algorithms.* We can imagine many bugs in famous generic algorithms. We can do the shortest-path algorithm in a graph without checking for already visited nodes for example (or we could take any common algorithm and add a bug in it). By "algorithmic bug," we mean that the bug is that we implemented an incorrect algorithm. This category is arbitrarily large but we are not sure that levels with this kind of bug are interesting because it requires knowledge of these famous algorithms if we want to train to debug and not algorithmic thinking. We will try to keep algorithms in all our levels reasonably simple or, we could keep these bugs for optional late levels.

Bugs related to the C language As Agdbentures levels will be written in C, we can add some bugs specific to the C language.

— *dynamic allocation*. `malloc` is done with the size of a pointer to the structure instead of the structure size. We pass a structure by copy and not by reference as a function argument. We use a dynamically allocated value after it was freed. Because of pointer aliasing, we free twice the same block.

— *segmentation fault*. There are many ways to obtain a segmentation fault in C. The difficulty of fixing a segmentation fault is more when searching for the location of the bug as it may be caused by many reasons. It can be the time to introduce some memory-leak detection tools like *Valgrind*. We have an engineering question to find a way that the bug has the same consequence on all machines as dynamic memory bugs are often machine dependent.

— *pointer aliasing*. Two arrays can be incorrectly shared or, all rows of a matrix are shared instead of separate copies.

— *dynamic size array*. We forget to copy elements back into the new array when we do a reallocation.

— *array size*. Because of incorrect static array access, variables next to it in the stack can

be changed. This behavior will also require engineering to ensure it is not machine-dependent.

— *assignment instead of comparison*. Using assignment instead of comparison is a valid operation in C but is nearly always a typo. It is a common source of bugs and it is a good habit to always check it.

— *infinite loops*. With decreasing loops, it can be easy to wrongfully use unsigned integer types out of habit. For loops that iterate every two steps, we should not use equality comparison but greater or lower. Infinite loops are a misuse of language, they can occur outside of loops, for example with infinite recursion calls because of a wrong termination condition or a not converging iteration.

— *string manipulation*. We forget the `\0` at the end of a string. We give the arguments to `strcat`, `strcpy` and other variants in the wrong order. We compare strings with equality and not string comparison functions. We compare `string\0` and `string\n\0`.

— *forget `break` in `switch` statement*. This is a common mistake even if experts have not programmed in C for a long time. Requiring adding a `default` block in a `switch` statement to solve a bug can also be interesting as it is considered a good practice to always have a `default` block.

There is a last category that we call "harder-to-fix bug." This is not a bug per se but we think we have to take into account if some levels require more work to fix a bug once it is found. We consider for example that adding a whole block is "hard-to-fix." We take this into account because usually most of the difficulty to fix a bug comes from finding its cause. An example of this is off-by-one errors, they may be hard to detect but once the error is found, fixing it is just increasing or decreasing a value. We thus make this "harder-to-fix" category to differentiate bugs that require significant work to fix once the cause is found.

3.3.3 Type of exercises

From the two previous sections, we can imagine some debugging exercises with bugs taken from our list. We would like to have some interactions between the bug and the level design and visualization. So if we do not find an interesting way to make a bug type happen in terms of level design, this bug time will not be present yet in Agdbentures. For a long time, we will not be exhaustive regarding our list from the previous section.

From the various skills required for debugging and especially program understanding, we believe adding some exercises that only focus on program understanding and does not require bug fixing is a good addition. This will emphasize to students that the time taken to read through the program is valuable when trying to do further bug research. We thought about simple MCQ about variable or function roles or asking what would be some function outputs for given inputs. From an engineering point of view, we will first develop the bug-fixing levels that require much more work and design time to do some first experiments with Agdbentures. If this experiment is a success, we will take some time to think about and add these "understanding-only" levels.

3.4 Presentation of Agdbentures

Figure 3.1 – A screenshot showing the fourth level of Agdbentures. The main character is on the path, its position depends on the values of `player_x` and `player_y` in the program on the right. The two flags mark the start and expected end position of the character. The light pillar shows the exit position. The path marks the expected movement of the character and continues after the exit to what will be the fifth level. The red texts are screenshot annotations and not part of Agdbentures.

Agdbentures is a graphical game with currently a dozen levels and several more in development. Each level consists of a debugging exercise where we give the player (student) a program in the C language containing bugs (the *level code*). The player needs to fix the bugs to advance to the next level. In order to make the game more attractive, but also to showcase the importance of having information about the internal state of the program being debugged. Agdbentures creates a visual representation of the level program by inspecting it at runtime. This is achieved by running the level programs using GDB, the GNU Debugger, with an abstraction framework. This abstraction framework is a consequent amount of work and will be presented in its own chapter.

3.4.1 The visual representation

The main element of the graphical window of Agdbentures is the feedback of the level program state, rendered as an RPG-like 2D world. In the first levels of the game, level code consists only of the manipulation of global variables such as `player_x` and `player_y`. These variables are monitored by Agdbentures and used as coordinates to display a character in the 2D world. In the screenshot example of Figure 3.1, we see the main character near the center, on a path. The *code window*, on the right of the screen, shows that the next line being executed

will call the `forward` function. Based on the current direction of the character (here, facing right) it will update the position variables (incrementing variable `player_x` in this case).

At the end of a level, i.e., when the level program ends, the character's position should be at a particular location called the "exit." It is visually represented as a yellow column of light, which we can see under the rightmost flag. In the example presented here, a bug in the `forward` and in the `turn_right` function will make the character stray from the path and fail to reach the exit.

Conceptually only a few graphical elements are not a direct representation of the program state but are the addition of Agdbentures. However, they take most of the visual representation window space (even more during the tutorial levels):

— The decorations (trees, logs, water... on Figure 3.1) are added to make the levels more visually appealing. They may guide the student when debugging. For example, the path on the map is a decoration to show the expected movements of the character. The flags on Figure 3.1 are used to mark the starting point of the character and the expected end position. The exit position is explicitly displayed. The exit position is read from the program memory so it is counted in the program state. We used free online assets for the sprites. Sometimes, we could not find a corresponding free asset so some sprites are placeholders.

— A scripted character from Agdbentures called the *Wise Old Developer* (the *WOD*), which can be seen on Figure 3.1 next to the leftmost flag. They are not part of the level code, so they do not appear in the level in a standalone way (outside Agdbentures). The *WOD* purpose is to provide information to the student at specific moments (usually at the beginning of the level).

Decoupling the graphical part from the level code allows us to present simpler level code to students, while still having an attractive visual representation of the internal state of the level.

3.4.2 An actual debugging session

In the GUI of Agdbentures, we mimic as much as possible a GDB debugging session. Agdbentures levels are standalone programs that are perfectly capable of being executed outside the framework. It is in theory possible to solve levels with a regular GDB session, or even just using printing strategies. Agdbentures can be seen as an "upgraded debugger," employing visualization to guide students in their debugging strategies while still providing GDB interface.

The terminal that launches Agdbentures becomes a GDB console emulation (shown on the left of Figure 3.1). It shows all GDB output to the student. Results of print statements are also displayed in this console (as in a regular GDB console). Commands that are typed in this console are sent to GDB.

However, initially, the main way to interact with GDB is by using the GUI buttons. Clicking these displays the command in the GDB console as if it was manually typed, for instance "next" and "step." We provide this functionality as the GDB command line can be intimidating to novice programmers. They progressively get accustomed to seeing commands being sent and results being displayed in the console. We plan to progressively remove the GUI buttons so students will have to use the console, as they would have to do when debugging their

own projects using GDB. So with practice, students can reproduce their debugging session in a plain GDB without Agdbentures' help. This claim would need to be proven in future work.

The level source code is displayed in a read-only window (shown on the right of Figure 3.1). The next line to be executed is highlighted (as GDB would print it); at startup, it is the first command of the `main` function. Currently, for technical reasons, the code edition must be done in a separate IDE window (using the *Open in editor* button). In the future, we plan to have a single window for both tasks, by developing an ad hoc *visual studio code* plugin. We made this decision to have separate windows so we can produce Agdbentures with less engineering effort and gather student feedback earlier. Student feedback will also help us to choose the final form Agdbentures should take, regarding a *visual studio code* plugin or maybe even a full-features single window that would be close to a video game.

3.4.3 Choices in the visual updates

Agdbentures can determine whether the GDB process is currently executing the level program or stopped. When the program is stopped, GDB is waiting for a command such as "next," "step". The visual representation is updated every time the level code is stopped. Agdbentures explores the variables in the memory of the level program. Variables whose values have changed since the previous stop trigger updates on the corresponding graphical part: moving the main character, hiding/showing an object, making walls appear, etc.

A consequence of this is for example that the main character moves from the previous position to the current one in a straight line (and ignoring obstacles), even if during the level execution the actual path was different. In a previous version of Agdbentures we used a hidden watchpoints mechanism: the variables needed to update the visualization were watched during execution. When a watchpoint was hit, we could update the visualization and silently continue execution. This was transparent for the player, and made the game look "nicer," as the actual path of the main character was being shown. But this design had a major flaw that motivated its removal and the actual design: Seeing the trace of execution even though the program is not stopped is not the behavior by default in a debugging session. The usual behavior is to just have information on the *current* state. Information on the intermediate states is lost unless one chooses to keep this information via tracing for instance.

One has to manually see the execution trace by using print statements, single-stepping or breakpoints, for example. Compared to the actual design, students were not incentivized to stop the program in zones of interest if they wanted to observe its state.

Admittedly, Agdbentures already offers some scaffolding for the students, by displaying on the graphical interface information that one would otherwise need to fish, but we felt that also showing in-between steps was giving too much. In general, we made many choices in Agdbentures, always trying the keep the balance between: On the one hand, having an engaging environment that can easily be seen as a game, hence showing visual feedback on what is happening, which also helps to debug and show the importance of observing the program state; On the other hand, not veering too much from an actual standalone debugging session, as we want students to develop skills transferable to real usage. The danger of Agdbentures scaffolding is that outside of this environment, students might not be able to recreate what was given to them for free in Agdbentures.

3.4.4 Intrusions in Agdbentures

It is critical that the general behavior of the level being debugged does not differ when run standalone or in Agdbentures; otherwise we say that Agdbentures is *intrusive*. Also, there is a problem if something happens in the debugging session that is not a result of students' actions. Some mechanisms are not intrusive, so we can have the best of both worlds. For example, when developing Agdbentures we found out that because of the GUI students do not pay much attention to the console and it is easy to miss compilation errors. This can be misleading because the bug can be solved but if we introduced a compilation error, we may think that the bug still remains. We now catch these compilation errors and display them in a pop-up window that cannot be missed. We can see on Figure 3.2 this pop-up window. Previously we could only see the compilation error in the terminal on the low left. We control the color in which we print this output in the terminal but the color depends on the color palette that is used by the user so we cannot ensure this will be eye-catching.

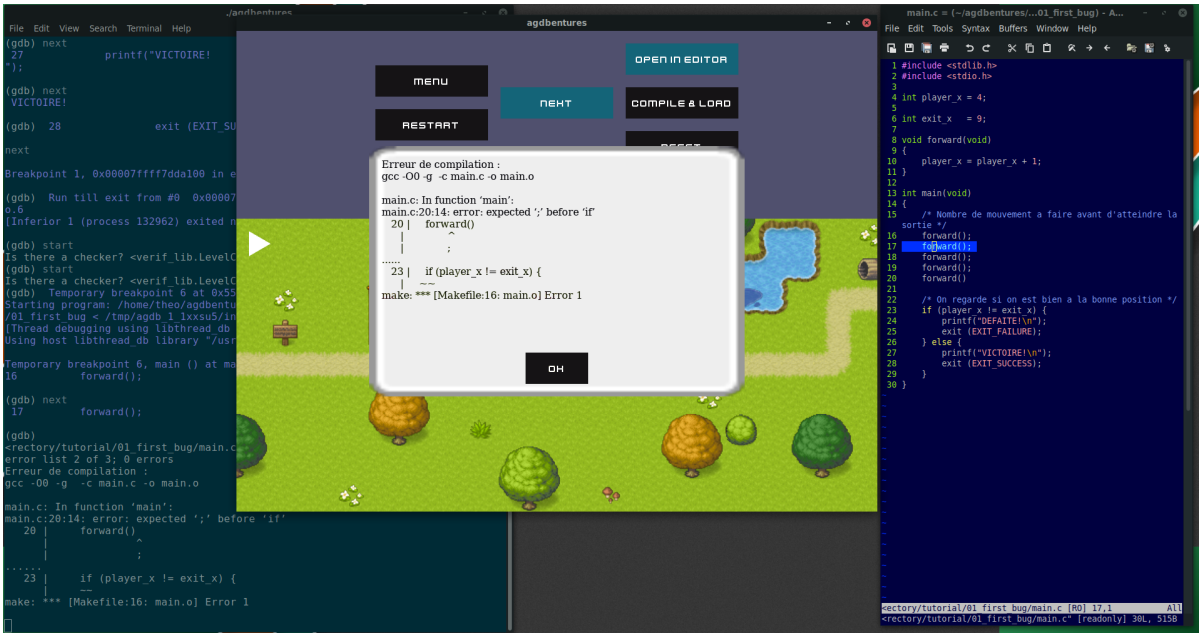
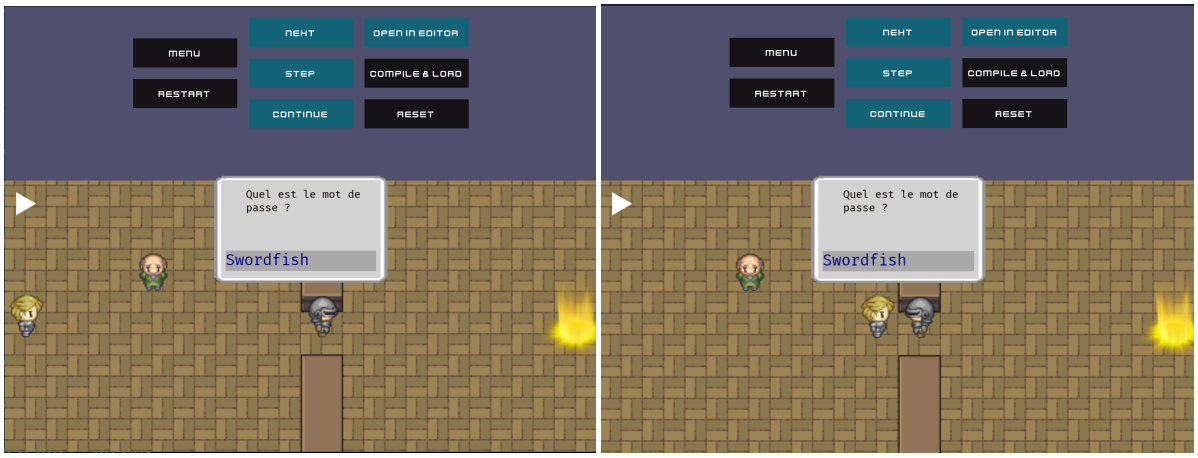


Figure 3.2 – The compilation pop-up we added in the main window to make compilation errors visible. We can also see this error in the low left of the terminal. We see on the code visualization that the source is updated when a compilation is done.

However, there are currently some elements that we want to do from a game perspective, but we did not find a way to have them in a fully non-intrusive way. The main example of this is when expecting user input, as is the case in a level titled "password." The regular behavior is to receive input from the terminal. However, this breaks the "gaming experience" (the game should be "played" in the graphical window). It is very easy to miss that an input is expected, so students would probably get stuck not knowing what to do.

Currently, the prompt is caught by Agdbentures, opening a dialog box in the graphical window. Now there is a choice to make: do we update the visual representation to represent the program state at the moment of the prompt or not? In the "password" level, a non-playing character asks the main character a password when facing each other. The least intrusive

would be not to update; however, this might mislead the player into thinking the prompt occurs sooner in the program: showing the two characters far apart and making the player believe this is a bug. The two different behaviors can be seen on Figure 3.3.



(a) The default behavior would not update the visual representation.

(b) This is the actual behavior with an update when the prompt is displayed.

Figure 3.3 – An example of the "password" level with the two update policies. Once the map is displayed we run a `continue` command. Agdbentures will stop this command when an input is needed on the standard input and display a prompt window. We also see that the guard turns around when the player gets close to it.

In the end, we decided to go for updating the visual representation even if, technically, the level is not stopped from Agdbentures's point of view. We can argue that as we stopped the program to display the prompt the player position is actually close to the guard in the program execution so updating the visualization is the correct state. We can see on Figure 3.3 a white triangle on the top left with both policies. This white triangle means that the program is still "playing" (thus the visualization should not be updated in theory). If the program is stopped, this is replaced by a video pause sign. Technically, a program is still running when waiting for a standard input so, we kept the "playing" mode. This is currently implemented with a breakpoint on a function in the level code that encapsulates the reading on standard input. A lot of engineering is required to resume the program and stop it where the command sent by the student should have stopped, but it is mandatory as the prompt window mechanism should be transparent to the student.

However, if a student looks into the breakpoint lists or carefully follows the breakpoints numbering they should be able to detect that a hidden breakpoint was placed by Agdbentures. We do not know yet if this can mislead students and we would need to find another solution. There is also the risk that students delete this internal breakpoint which will completely break the level in Agdbentures. This kind of question between user experience and non-intrusiveness is still open.

Two other mechanisms in Agdbentures have the same questions as this prompt window:

— Updating the visualization when the program exits. We also chose to force an update because issuing a `continue` command right at the start would not produce anything visually and just be a winning/losing screen.

— Displaying messages in pop-up windows to give instructions to students. These messages are special `printf` functions that open a pop-up window in Agdbentures in addition to the regular `printf` call. In the level code, it is only a `printf` call encapsulated in a special function that Agdbentures placed a hook on.

3.5 The game engine

We purposefully reuse a lot of code between levels, so students become as familiar with the source code as if it was their own. This allows us to propose interesting levels with hundreds of lines of code without being overwhelming for students.

We believe the common codebase between levels we call the game engine is one of the main contributions of Agdbentures. To our knowledge, exercises in debug practicing systems are independent and do not take advantage of students doing several exercises to increase the codebase size and try to progressively remove toy examples.

The background in-game justification of the game engine is that each level represents a "snapshot" of a project under development: initially, the project is very small, but increases in size as "time" advances, sometimes even undergoing refactoring or massive changes in the data structures. The project itself is a rudimentary RPG game, with basic movements and interactions, and no graphical interface. It is a C code library that is used by Agdbentures levels not a feature of Agdbentures environment, like the visual input prompt, for example. The *WOD* presents themselves as the game developer and asks the player for help in finding bugs in the game. The levels can be run independently in a console for example (displaying only the "prints" in the program), but are usually run using Agdbentures, which creates a graphical representation of the game being developed using mechanisms presented previously.

Initial levels consist of a single C file, but at some point, the source code for levels is separated into a *game engine* part, which is reused between levels, and a *level* part, specific to each level.

3.5.1 Game engine incremental versions

Four versions of the game engine of increasing complexity are currently available in Agdbentures. The other features we developed are not yet attached to a game engine version and used in actual Agdbentures levels. The student sometimes has to debug the game engine code, but most of the time they only must be able to understand it to understand the level code. At first, we wanted to keep students' modifications of the game engine throughout levels. We thought the game engine could be versioned and the student modifications would only be patches inside a history. However, we were afraid of history conflicts or student fixing bugs in a not intended way that would be detected only multiple levels in the future. While this could mimic some real case scenarios where a programmer might discover some bugs in previously written code after introducing new use cases, we decided such bugs would be too hard to find and we would have no control over their possible appearance or not. So after each level, the teacher version of the game engine is used for subsequent levels.

We extensively describe the features of each version in this section, the last current version of the game engine is about 700 lines of code that the student will gradually understand. All other features need to be written in the level code.

— *simple map*. This is the first version of the game engine introduced to the student. It features a 2D map represented as a matrix of characters, basic movement functions for the main character, and the addition and removal of elements on the map. There are also basic verification for the movement of the player like *the destination is not a wall* (the '#' character).

— *input-command*. This introduces interactive movements of the main character via the standard input, instead of predefined movements.

— *read-map-str*. The initial 2D map is built from the parsing of a multiline string representing entities on the map.

— *map-stack*. Multi-map handling, stored as a stack. It is a poor data structure for a game, but very interesting in terms of debugging. We use them to mimic the call stack: entering a building in the game is done by calling a function and pushing a map on the map stack; leaving the building is returning from the function, popping it from the call stack and the top map from the map stack. Most bugs related to this engine version are related to variable scope as we enter/exit several functions. Another source of bugs is aliasing as several maps in the map stack can wrongfully be aliases of each other.

Each new version of the game engine has a small explanation of the reason for the changes given in the level that introduces them. Sometimes some refactoring occurs, and past designs of the game engine are not able to support new use cases. Refactoring is very important in actual program development, but rarely shown in class: here is a side benefit of Agdbentures, showing the need for the refactoring and the benefits of the design. Indeed, passively observing good programming practices or showing why bad designs are not efficient is beneficial to students [85].

Levels that introduce a new game engine version guide the student more. They might feature a bug placed in the game engine, or the level code not using the new engine correctly, forcing the students to read through the new code. Understanding the code structure and an explanation about the code is still part of the debugging process [84].

3.6 Level list

The version that is discussed in this manuscript is available at `https://gitlab.inria.fr/CORSE/agdbentures/-/tree/MainTheoPHD`. A lot of refactoring was done until now and some intern students added and modified several levels. This work can be seen on the main branch but is not part of this thesis. We divided the level list into different difficulty sections. We will present each of them and detail some typical levels for each section. The reader is encouraged to launch Agdbentures and try the levels if they wish to. The instructions are available in the README file at the root of the repository. However, Agdbentures is only available for Linux. If the reader is not able to launch Agdbentures, it should still be possible to follow this document and get an idea of its content.

3.6.1 Tutorial levels

The first six levels are considered the "tutorial" and focus on the understanding of the Agdbentures environment. At the end of the tutorial, the student should be able to edit,

compile and run the program, and understand the link between the visual representation and the program execution. Tutorial levels consist only of single-file C source code, reusing and augmenting the code with new functionality between levels. The student can get used to already knowing part of the code from previous levels. So the game engine will come naturally as an abstraction of all this common code.

We detail the concepts introduced in the tutorial in their dedicated levels:

— *introduction*. This is a really important level as it is the first one students encounter when launching Agdbentures. In this level, only the "next" button is available (but any command can be typed in the console). The player just walks a few steps forward by increasing a global variable in a *forward* function. A single button is available. It is the *next* button that will execute the next *forward* call and move the player. There is no bug and just a few presses on the *next* button will finish the level (see Figure 3.4). This level shows how to execute a single statement in the program and the student can see the result in the console and the code window.

Figure 3.4 – A screenshot of the first level. Some more instructions are given when the player reaches the sign. All the flags are the start and exit points of further tutorial levels that all happen along the same path.

— *first_bug*. The second level is also important because it shows how to edit and compile the code. It also introduces the *WOD* and the validation framework. The level starts with the *WOD* entering the level and explaining who they are. There is a simple bug in this level, it lacks a *forward* call to reach the exit. The code cannot be modified yet, the edit code button becomes available after the first complete run. After this, the *WOD* explains how to edit and compile the code. Instead of adding a *forward* call the *WOD* suggests modifying the position of the exit instead. However, it is forbidden to move the exit, so the validation framework does not pass. After this, the *WOD* tells to add a *forward* call instead.

We will see in Section 3.8 that this level did not meet our expectations and need some change. The main problem was the introduction of the validation framework that is not

well understood after this level. The way we introduce it proved to be misleading.

— *refactoring*. This level is quite short and shows that code from previous levels can be modified. From now on, when this happens there will be an explanation of the change by the *WOD*. The branches of the exit condition are wrongly executed due to a lack of parenthesis.

— *bugs_hiding*. The player has to follow a path with two turns, the novelty of this level is that two bugs are present. The first error is due to the *turn_right* function body being *copy pasted* from the *turn_left* body. The other error is due to an inversion on the y-axis. This level is the one on Figure 3.1. The second error is only observable once we have corrected the *turn_right* function.

— *stairs*. This level has a really long repeating path that the player needs to follow (see Figure 3.5). The bug is an off-by-one error in the loop that repeats the player's movements. This level introduces the *continue* command. We do not provide breakpoints yet. We introduce what we call *magic-breakpoints* (we chose the name so, it suggests a scaffolding feature). By right-clicking somewhere on the map, students can place a *magic-breakpoint*. When the character moves on a *magic breakpoint*, the program is stopped. We think about removing this kind of scaffolding in the last levels of Agdbentures. If students need this functionality, they would need to place a conditional breakpoint themselves. This scaffolding could be removed when an advanced level would need a conditional breakpoint for the first time, for example.

Figure 3.5 – A screenshot of the fifth level. This level quickly introduces zoom/dezoom. We use it to take screenshots that span the whole level view. The blue pillar on the path is the position of the *magic-breakpoint*.

— *key*. This last level does not introduce new concepts but the student has to fix three bugs. The player is walking along a path with several turns, there is a key on the path that opens the door to access the exit (see Figure 3.6). The first bug that is encountered is a bug similar to *stairs*, the loop bounds are correct but function *turn_left* is swapped with a *turn_right* in the loop body. Then, the player does not move far enough to reach the door so executing the rest of the code makes the player stuck in front of a wall. When trying to open the door, a global variable tells if the player holds the key. This global variable is initialized to false but never set when the player picks up the key. These three bugs are only observable once the previous one is fixed.

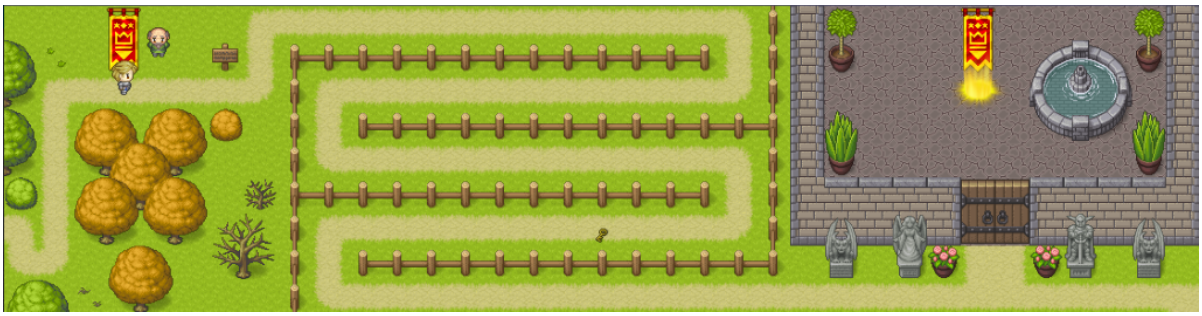


Figure 3.6 – A screenshot of the sixth level. The screenshot is taken after one press of the "next" button so that the key is placed on the path (it is the first instruction in the source code).

Note that the tutorial currently does not introduce students to strategies based on printing/tracing. However, we will see in Section 3.8 that this perturbs students so this will change in the future.

3.6.2 Basic levels

The next group contains 6 levels: it is the *basic* section and is dedicated to introducing the game engine and making it grow. The levels start to have features we might expect in a game: having a map structure, interacting with non-playing characters, and controlling the character's movement with the arrow keys.

Two levels in this section are regular levels that do not grow the *engine* code. They are the "password" level presented several times before and a level named "key." In this level, two successive doors block the exit and two keys are randomly placed on the way and need to be picked up. This level contains a single bug that is a typo in a comparison `key_count = key_used` instead of `key_count == key_used`. If this condition is valid, the door will not open. All the levels in this section should not need breakpoints to be solved because their execution trace is short enough, but some may be easier to solve with it.

Most game engine levels are just textually introducing the new engine version with some source code to read. The level that introduces the `simple_map` version (the first one) is more polished. A wall is surprisingly placed around the player when they walk forward. Students have to search the game engine source code and find the function that removes an entity from the map and removes a wall tile to make a way to the exit. We think this kind of interaction is interesting to make students familiar with the game engine source code. We plan to do this for the other game engine levels.

3.6.3 Medium levels

The last currently available section is called the *medium* section. It contains 4 levels with longer code (about 200 lines of code without the game engine code). The bugs are not necessarily more complex than in the *basic* section, but the difficulty comes from finding them. These levels are available but not yet tested with students because of short in-person interview evaluations (see Section 3.8). It is, however, available in a partially playable way.

One level is the last game engine level that introduces the `map-stack` version. The three other levels are quite different from each other, not all of them use the `map-stack` game engine version even if it is available:

— *pet-follow*. It features a sinuous path and, a small pet follows the player by copying their move with a small delay. This is implemented with a circular buffer. Each player move command is written in the circular buffer and, the first move of the buffer is read. There is a single bug in this level coming from command string aliasing instead of copying. All the commands in the buffer are actually aliases of the same command that is always the last move of the player. Once the commands are correctly copied, the pet correctly follows the player from a constant distance.

— *local-struct* and requires the `map-stack` game engine version. In this level, the player gathers coins outside and can use them in a shop to buy some armor to defeat a monster that guards a bridge leading to the exit. The shop is a small independent map that the player can enter and exit. This is implemented as a function call that pushes the shop map when entering and pops it when exiting. The gold count is a global variable. The bug in this level is that the player equipment structure is passed as a copy and not a reference to the shop function. So when the player exits the shop, they still lost their gold but the equipment they had before entering is the same. The modifications to the equipment are done locally in the shop function. After the equipment is passed by pointer, it is correctly bought and the changes last after the return of the shop function. This level is one of the first we developed and inspired the whole Agdbentures project.

— *crepes*. It uses the same mechanic as the "local-struct" level with different buildings and functions for each one. The player has to go to the mill and gather some flour and go to the henhouse to gather eggs. They must give both of these ingredients to a cook in a third building that will make crepes out of it and the player will be able to finish the level. The player can only hold a single item at most. So they have to go twice to see the cook, deposit the first ingredient, gather the other one, and come back. There is again a single bug in this level, the maps are not saved after exiting and are always recreated from scratch. So even if it makes flour and egg infinite, when an item is given to the cook it disappears the next time the player enters the building. The cook cannot have the two items at the same time because the player can hold a single item and has to go outside to pick the other ingredient. Once the maps are saved between calls, the cook can have both ingredients and the level can be solved.

3.6.4 Work in progress levels

Finally, a *difficult* section is under development with harder-to-find or fix bugs. We have only a few of these levels planned as we will think more about it when Agdbentures is more polished as their development may be tied to internal engineering.

— We plan a level with the `map-stack` game engine version where there is a main map and three maps that should be copies of the same basic maps and play the role of a tower with multiple stairs. Different actions have to be done at each stair but the maps are actually aliases of the same map so each modification is applied virtually to "all the stairs." The bug can be fixed by implementing the copy of the maps and storing them as an array instead of the same pointer.

— We also thought about what could be the last level of Agdbentures and give a lot of challenges. We thought about a level inspired by the *Baba is you* video game[69]. All the "physics" rules would be a part of the 2D map. An additional difficulty for this last level would be that before trying to search for some bugs, the level is already a puzzle by itself that must be solved. As the rules are part of the level itself, nearly every arbitrary bug can be put in the level. This would, however, require a huge amount of level-specific code (see in the next section).

Currently, several levels in development are just the results of quick ideas and consist of just a standalone correct C program developed by interns. We think we might add interesting bugs afterward. This includes famous mini-games like *snake*, *sokoban*, *minesweeper* or the Hanoi towers. The source code alone is already quite long, all these levels would be in this last section of Agdbentures that we would consider more of a challenge and not required to complete Agdbentures "experience."

The development of current and future levels is detailed in the next section about implementation.

3.7 An extensible implementation

Agdbentures has two main building blocks written in Python: the graphical window and the level management. Enough work was done in software engineering so adding a new level requires writing mainly specific C-code for this level (the code given to students), and a bit of level-specific python code. Some levels may require graphical mechanisms not present in Agdbentures yet. For example, when we added the "password" level we also had to implement the user input mechanism in Agdbentures. Some expertise in the system is still required to ensure correct communication between the level-specific python code and the graphical interface. We plan to have a cleaner API in the future to ease the writing of new levels for external users.

3.7.1 The GDB monitoring framework

We use an external generic framework (the red dashed box on Figure 3.9) that provides several abstractions over GDB and helps Agdbentures implementation. GDB is run as a subprocess, and the program level to debug is itself run by GDB. The control of GDB is synchronized with commands, sent as blocking calls which return when the command is finished. We provide direct access to GDB input and output with read and write queues, and an API allows easy access to the program memory and reading values as if they were regular Python variables. We can directly forward student and GUI commands to GDB.

This framework named *Easytracker* is presented in the next chapter of this thesis. While developing Agdbentures we realized that this program monitoring and GDB abstraction may actually be a more generic task that could be used in other teaching applications. So chronologically, we stopped Agdbentures for several months to develop this generic abstraction framework and use it for Agdbentures development.

An example of *Easytracker* use for Agdbentures development is presented on Figure 3.7. This is only a small portion of the generic memory reading code used for the visual update. The `find_var_in_frame` function checks if the given variable name is present in the given

```python
def find_var_in_frame(frame, name):
    if name in frame.variables:
        return frame.variables[name].value
    return None

def find_var_in_frames(frames, name):
    for f in reversed(frames):
        var = find_var_in_frame(f, name)
        if var is not None:
            return var
    return None

def find_map(frames):
    return find_var_in_frames(frames, "map")[0]

def find_char(level_map, char):
# the level_map argument would be the output of a previous find_map call
    if level_map is not None:
        for y, line in enumerate(map.tiles):
        # line is a C char*
            for x, val in enumerate(line):
            # val is a C char, Easytracker represents this in Python a string.
                if char in val:
                    return x, y
    return None, None
```

(a) Example of generic code in Agdbentures to access the 2D map and characters position when the 2D map is a reference to a map object in the stack.

```c
typedef struct map_s {
  int width;
  int height;

  char **tiles;

  int player_x;
  int player_y;
  direction player_direction;
} map;

int main() {
  map* map = init_map(WIDTH, HEIGHT);
  ...
}
```

(b) The map structure defined in first versions of the game engine and a minimalist example.

Figure 3.7 – Correspondance between the C level source code and Agdbentures thanks to *Easytracker*.

```
def run(self):
    tracker = init_tracker(self.level_exec)
    while not tracker.program_exits:
      # waits for a command from the GUI or the terminal
      command = get_command_from_gui()
      tracker.send(command)

      # when easytracker returns, it means the program is stopped
      # so we can inspect the memory
      program_memory = tracker.get_memory()
      self.visual_update(program_memory)

    # the loop exited so, the program terminated
    if tracker.exit_code == 0:
      self.success()
    else:
      self.failure()
```

Figure 3.8 – Pseudo code of the level main loop written in an imperative way.

`Frame` object returned by *Easytracker*. It is possible by iterating on the frames to find a reference to the map structure in the program memory provided we know the variable name. In our levels, we always use the variable name `map` when the map is directly accessible in the `main` function (and not encapsulated into a generic `game` structure like later game engine versions). We see on Line 14 that we access the first element of the map object returned by *Easytracker*. It is because the memory model returned tries to mimic the actual memory structure. As the map in levels is a pointer to the map structure, it is represented as a tuple with a single element in Python to emulate a reference. Thanks to *Easytracker* we can easily write high-level functions like `find_char`. That one looks for the coordinates of a given character on the map. Internally, the map is represented as a 2D array of characters (see Figure 3.10).

Easytracker also helps with program control. Agdbentures main loop is synchronized with GDB commands. After a command is received (from the console or GUI), it is sent and processed by GDB. When the command is finished, the program state is fetched and the visualization is updated. *Easytracker* helps with the command synchronization so this can be written in an imperative way. However, the code is still quite verbose and will not be presented in more detail here.

Instead, a pseudo-code of this main loop is presented on Figure 3.8. Some mechanisms like internal breakpoints are omitted. The prompt, message and update-on-exit mechanisms described in Subsection 3.4.4 use this feature. To implement this mechanism we defined a high-level function `register_breakpoint(funcname, callback)` that internally places a breakpoint on the given function and calls the provided callback when this breakpoint is hit. In practice, we listen for specific functions in the game engine (the one displaying a message or waiting for user input for example). This allows Agdbentures triggering specific events, for example, displaying the prompt waiting for user input. As we already discussed in Subsection 3.4.4, every time we place an internal breakpoint, we add a bit of intrusion to the debugging session. Again, this is a problem if students delete an internal breakpoint because it would break Agdbentures but we believe these additional graphical mechanisms contribute to the general user experience.

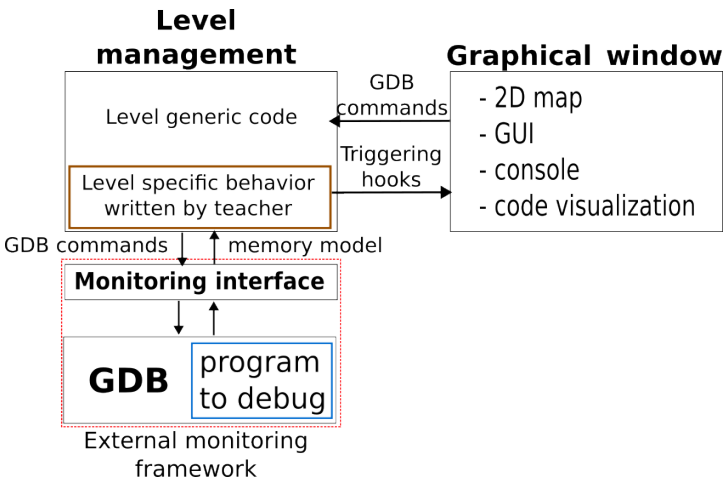


Figure 3.9 – Architecture of Agdbentures

3.7.2 The graphical window

The graphical window contains the level visualization, GUI buttons to send commands to GDB, the console, and a window to show the level source code, highlighting the next line to be executed. All the graphical elements are implemented with an open-source Python library called *arcade* (`https://api.arcade.academy/en/latest/`).

The level visualization code is mostly generic to all levels because it knows from the game engine version how the 2D map is stored in memory. In the tutorial, there is no game engine per se, but the memory structure is the same across the tutorial (using global variables) so most of the visualization code can be kept generic.

The graphical interface also provides general hook functions that can be triggered by message passing from level-specific code. This is mostly used to react to some events in the level state, for example "When the door opens, make the door sprite disappear." Since testing if the door is open depends on how the door state is encoded in the program and can vary between levels, this needs to be scripted.

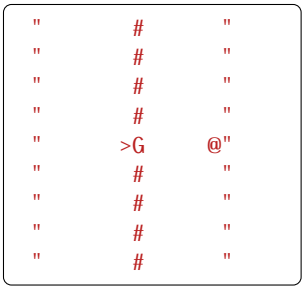
The graphical sprites are mapped from characters in the 2D map. Common characters have predefined sprites in Agdbentures but some levels need to overwrite them or even add their own sprite unique to this level.

For example, Figure 3.10 shows the internal 2D map in the "password" level when screenshots on Figure 3.3 are taken.

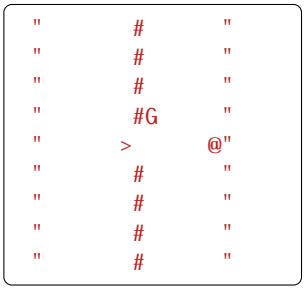
3.7.3 The level manager

All the per-level scripting is done in Python, extending Agdbentures generic classes. The level-specific code rarely exceeds 100 lines even for quite complex memory structures, such as accessing values on specific map positions in the map-stack game engine version. This is thanks to *Easytracker*, which makes accessing the program memory much easier, and Agdbentures abstractions.

The general level management is the part that "knows" what to look for during the execution of level programs (e.g., global variables, or fields in structures), and converts this



(a) The map state when screenshots on Figure 3.3 are taken.



(b) The map state after a correct execution so the guard leaves a passage.

Figure 3.10 – The map is 14x9 and is what would be seen if the program is run without Agdbentures and we print the map. Spaces are empty spaces and can be filled with decorations. Characters >, @ and # are the player, the exit and the walls respectively. The G character (the guard) is a character specific to this level so we have to define a custom sprite for this character. We can see again that the *WOD* is not part of the program code.

information into instructions for the graphical part.

Before the level is given to the student, a preprocessing phase generates the files from the game engine code and the level code. The generated folder contains everything the students need to run the program in a standalone way. This process is described in Subsection 3.7.5.

On Figure 3.11, we can see the actual code for the "password" level. We will take some time to detail this as it will show much internal work of Agdbentures as well as its current development state. First, we see that all the level-specific code is in a dedicated class inheriting a common `Level` class provided by Agdbentures. This is the class that will be dynamically loaded by the level manager. The level manager will call the `arcade_pre_run` function before running the level so any customization can happen.

All of Agdbentures abstractions are encapsulated into the `AbstractLevel` class, so we can virtually override anything that Agdbentures does. However, Agdbentures graphical code might range in a few thousand lines of code so, it is unreasonable to override core functionalities. Fortunately, all our levels need only little customization (like the "password" level example).

We can see on lines 3, 4 and 5 that a `map` object is provided by the abstract level class. This `map` object holds reference to all the elements on the map and their attached behavior. These objects can hold a reference to sprite object so they can be displayed on the screen. We can get references to the player object and named objects. The names are given in some level metadata (this will be detailed in Subsection 3.7.5).

The code related to the *WOD* is quite straightforward in this example (see lines 7 to 11). Some special trigger conditions could be written but here we can simply use the hook feature presented in Subsection 3.7.1. Here, the *WOD* just tell their introduction message when the `place_player` function is called, which sets `player_x` and `player_y`, which also triggers the appearance of the player on the map. The *WOD* messages are written in the level metadata as comments in the `main.c` file.

Then from lines 13 to 27, we can see some code to fetch the guard position. As the guard is a custom sprite for this level, we need to tell how to read its position from the program state. We define a local function `set_guard_position` that takes the program memory as

```
1  class Level(AbstractLevel):
2      def arcade_pre_run(self):
3          player = self.map.player
4          wod = self.map.named_objects["wod"]
5          guard = self.map.named_objects["guard"]
6
7          # WOD code
8          wod.visible = True
9          wod.place_at(3,3)
10
11         self.register_breakpoint("place_player", lambda: wod.talks('intro'))
12
13         # Interaction between the player and the guard
14         def set_guard_position(memory):
15             log.debug(f"Setting position for {guard}")
16             frames = memory["stack"]
17             old_x = guard.coord_x
18             old_y = guard.coord_y
19
20             level_map = find_map(frames)
21             guard.coord_x, guard.coord_y = find_char(level_map, 'G')
22
23             if guard.coord_x != old_x or guard.coord_y != old_y:
24                 log.debug(f"New position for {guard}")
25                 guard.has_changed = True
26
27         guard.set_position = set_guard_position
28
29         def player_custom_update(player, memory):
30             if player.is_on_left(guard):
31                 guard.direction = cst.Direction.LEFT
32                 guard.has_changed = True
33
34         player.post_update = player_custom_update
35     ...
```

Figure 3.11 – Specific code loaded by the level manager to override default behaviors. We may move such simple *WOD* code fully to level metadata.

an argument and updates the guard position accordingly. We see that we use the 2 generic functions `find_map` and `find_char` described in Figure 3.7. This code should depend on the game engine version, that is why we currently do it manually in the level-specific code. For example, in later versions of the game engine, the current map may be located in an array of maps that would need to be accessed differently. On Line 27, we override the guard object default behavior to find its position.

On lines 25 and 32, we see that we notify Agdbentures when a sprite is updated so it can trigger a redraw. Lines 29 to 34 make the guard turn around and look at the player when it gets close to it. The `post_update` function of the player object is called every time the player moves.

We believe that Agdbentures graphical behaviors are reasonably encapsulated so level-specific code is quite an easy task. This abstraction may not be in its final state but without

too much expertise in the system and some examples from other levels, it should be feasible to add new levels for external instructors. It still takes several hours to actually write a level code and implementation in Agdbentures but if some additional behaviors that are not possible in the current Agdbentures version are needed it should be easy to add them with the help of the main developers.

3.7.4 Level validation framework

There is a validation framework that checks assumptions on the level execution trace after it is solved. After a seemingly successful execution (exit code 0 of the level), the level is run a second time—in the background, so as not to be intrusive—with added breakpoints and watchpoints to check assumptions. The program is run in another instance of *Easytracker* so we can stop the program as much as we want it will not interfere with students' debugging session. We make generic verification, e.g., that the main character is really on the exit position when the level ends, or that its position is only modified in the "`forward`" function (and only with ±1 increments), as well as per-level assumptions such as "the `has_object` boolean value is set to `true` only when the character position is the same as the object." This is an example taken from the last level of the tutorial where the player picks up a key on the map. Once the key is picked up, this boolean is set.

Since students are given full permission to edit the level code, it is always possible to circumvent "validation checks" and validate a level without actually having fixed the bugs. So the goal here is not to prevent cheating but more pedagogical: these verification are safeguards so students will not accidentally validate a level, genuinely thinking they have fixed a bug but missing the point (and hence the learning).

On Figure 3.12, we can see the validation code for the last level of the tutorial. We do not use the "password" level example this time because its validation code is really short and only checks that the guard correctly displayed its message. This level is called "key" and the player needs to pick a key on their way to open a door locking the exit. One bug in this level is in the code that picks up the key so, we make some validation for it. The functions `pre_validation` and `post_validation` are called by the level manager when a validation phase is run. The `pre_validation` function is configuring the validation phase and if some checks need to be done when the program exits they can be put in the `post_validation` function. When the function `failed` from the checker is called (on lines 10, 20, 29, and 32), an exception is raised and caught by the checker. This exception ends the validation phase and we know it has failed. If no exception is raised, the validation phase passes and the next level is unlocked.

This validation code performs two checks:

— When the player picks up the key, they must be standing on the key. This is done by placing a watchpoint on the `has_key` variable on Line 5. The `check_on_key` function just reads the program state and compares the two positions.

— The door has to be correctly open at one point during the execution. The player must stand in front of the door and hold the key. This is checked by stopping when the checks to open the door are done in the level code (see Line 6). Again the checking function just reads the program state and performs the according comparisons.

The validation code is generally more verbose that the graphical code. We may be able to find some common abstractions between the validation of different levels but, the verbosity

```python
class Level(AbstractLevel):
    ...
    def pre_validation(self):
        self.checker.has_opened_door = False
        self.checker.register_watch('has_key', self.check_on_key)
        self.checker.register_breakpoint('try_open_door', self.check_open_door)

    def post_validation(self):
        if not self.checker.has_opened_door:
           self.checker.failed("Player has not opened the door")

    def check_on_key(self):
        tracker = self.checker.tracker
        plx = tracker.get_variable_value('player_x', "int")
        ply = tracker.get_variable_value('player_y', "int")
        keyx = tracker.get_variable_value('key_x', "int")
        keyy = tracker.get_variable_value('key_y', "int")

        if plx != keyx or plx != keyy:
            self.checker.failed("Player is not on the key when has_key changes")

    def check_open_door(self):
        tracker = self.checker.tracker
        plx = tracker.get_variable_value('player_x', "int")
        ply = tracker.get_variable_value('player_y', "int")
        doorx = tracker.get_variable_value('door_x', "int")
        doory = tracker.get_variable_value('wall_y', "int")
        if plx != doorx and plx != doorx + 1 or ply != doory:
            self.checker.failed("Player is not on the door when opening it")
        has_key = tracker.get_variable_value('has_key', "int")
        if not has_key:
            self.checker.failed("Player opens the door but does not have the key")
        self.checker.has_opened_door = True
```

Figure 3.12 – Validation code for the last level of the tutorial. In this example, all variables read from the program are global variables.

should come more from the number of different verification than the verbosity of each one. Still, it should be feasible for external instructors to write their own validation code with the provided examples of other levels.

3.7.5 A word on Agdbentures levels development

Generating the source code for levels is much more complex than one could expect at first glance. Indeed, we need to juggle level-specific code, and game engine code that evolves during play. Both of them can contain bugs or not. We also want to have reference versions without bugs for testing and for some levels with multiple bugs intermediate versions where some bugs are fixed but not others. We also have versions with a “false fix,” the bug is fixed but by also changing the program specification to test if the validation phase is able to catch it. Moreover, since Agdbentures itself is a project in development, all of this is also in constant evolution from our perspective, making changes in the levels, bugs, and game engines!

In order to keep consistency in game engines and level code, we decided against creating multiple copies of the various codes. Instead, we use common files with heavy usage of the C preprocessor directive. This allows us to then generate the source files for particular levels by defining/undefining preprocessor macros using the `unifdef` *UNIX* utility tool. We can also generate a development version with symbolic links to the original files. So instructors can compile and test a standalone version of the level while the changes are propagated to the level files.

There is a little metadata written in the C code of the level to allow this preprocessing to happen. This metadata contains mostly the written instruction given to the student by the *WOD* and the game engine version to use. They can also encode some limited generic graphical behavior if the memory structure is common.

```
1  /* @AGDB
2   * level_title: password
3   * exec_name: password
4   * engine_name: simple_map
5   * engine_tags:
6   *
7   * available_commands: next step edit continue
8   *
9   * player_mode: simple_map
10  * arcade_maps: main main.tmx
11  * OBJguard: char_rep G
12  *
13  * HINT1 how is the password verified ?
14  *
15  * WOD: message intro
16  ... <content of the WOD message>
17  * EndOfMessage
18  *
19  */
```

Figure 3.13 – Metadata written in `main.c` for the “password” level.

We can see on Figure 3.13 the metadata for the “password” level. The `level_title` line is a display title in Agdbentures. The `engine_tags` line is often empty, it specifies for certain engine levels some preprocessor directives to apply. This is only used to introduce some bugs in the engine for this level. As we progressively add commands in the GUI, this is also configured in the metadata at line `available_commands`. The first time a command is available, Agdbentures makes it blink to catch the attention.

The line `player_mode` tells Agdbentures how to fetch the player coordinates in the program state. In this example, the mode is the name of the engine, but the game engine may change its version and keep the same player position encoding so the two names would be different. There are also some names for the tutorial where there are no game engine yet but the player coordinates encoding changes.

A `tmx` file is mentioned on Line 10. Each level has at least a `tmx` file attached to it. This file contains the 2D tile map with the sprite sheet and the different sprite layers of the map. The file is editable with graphical software which eases their creation a lot. This is parsed by the `arcade` library as it is a common extension to store such maps. If we also remember

from Figure 3.11, a guard object is mentioned. The name is provided in the metadata along with the character that will represent it on the character map. This declaration in the form of `OBJ`*name* makes the sprite available in the level code as a named object. We can also see on Line 15 the definition of the introduction message displayed by the *WOD* on Figure 3.11. The line specifying a hint is not yet used by Agdbentures. We are thinking about some hinting mechanism to avoid students being stuck but hinting is a strong pedagogical choice so, we still have to think about it.

This metadata is read as a dictionary by Agdbentures, some entries are mandatory (like `exec_name` or `engine_version`) but arbitrary entries can be defined and used by the level-specific code. However, we do not think we will encounter this use case of arbitrary entries because the metadata is known by the developer when writing the `level.py` file.

3.8 Experimental results

In this section, we will describe our first test of Agdbentures with students. The main difference with previous chapters is that we can directly interact with students and observe how our method benefits or not to learning. However, quantifying learning is still an open problem. Due to the difficulty of this task without large-scale experiments[86], computer education researchers produce more and more qualitative analyses of their methods. We will have a qualitative analysis of the effects of Agdbentures. At first, we want to observe if students can use Agdbentures in autonomy and if they enjoy using it.

3.8.1 Experimental setup

Although the project is still in early development, we ran a first phase of the experiment to observe if students adhere to Agdbentures and if they enjoy using it and solving levels. We also wanted to rule out major flaws in Agdbentures, hoping to discover only minor flaws using student feedback. [1] We expected to discover problems in the UI, in the level difficulty, and more generally in students' autonomy.

We decided to organize individual meetings with some volunteering students, where they could try Agdbentures for about 45 minutes followed by 15 minutes of open discussion about what they felt during their debugging session.

Since the ultimate goal is that students can train autonomously, we tried to have as few interventions as possible during their "play": we handed them a laptop with a fresh game of Agdbentures running without more introduction or explanation. They were, however, asked to say out loud their thinking, if possible, so we had an easier time figuring out if they were stuck or what was bothering them.

Although we finally did not use them, we recorded console activity, and GUI activity, and had automated commits in a git repository of the code they wrote at each compilation. The students were aware of this, it is stated in the document they read and sign before participating in the experiment.

1. An anecdote: the very first thing that the very first student did was already something we did not expect: he started by maximizing the graphical window, hence hiding the console as well as the code window, and as such played the game "half blind" for more than 20 minutes...

3.8.2 Meeting results

We recruited volunteers for this early experiment in a programming and data structure class for second-year university students and selected seven students with academic performance ranging from below average to high. We were surprised to have too many answers. We arbitrarily chose students who answered first to the form while keeping academic performance diversity. Students managed to solve between 4 and 7 levels during the meeting. All students were highly engaged when using Agdbentures and enjoyed solving the levels. Some sessions lasted for about 65–70 minutes compared to the initial 45 minutes expected as it was difficult to make them stop, especially when being in the middle of a level.

In general, students were autonomous, requiring very few external indications on all the tutorial levels but one. This was a relief as we proportionally spent much more time designing those, with more explanations and graphical effects, to ensure the first steps in Agdbentures would be smooth. Still, we observed many problems, the main ones being the following:

— No-one understood the "validation" part. The way we present it is in level (1) to trick the player into fixing the bug the "wrong" way, by modifying the `exit` position, hence making validation fail. We have not decided yet how we will solve this. We will completely redesign this level, introducing the validation later in the tutorial and with more explanations.

  We could also tell in advance the assumptions that will be verified in comments in the level code. This could be seen as an addition to the global specification being *The program needs to have an exit code of 0*. Students will know before running the code what will be an accepted solution. Another idea is to run the validation phase before running the program, students would get the error message just after compilation. These ideas are not mutually exclusive but will for sure need more thinking and experiments.

  The actual version that is used after this manuscript was written is that we do not introduce validation in a dedicated level and the error screen when the validation fails is much more explicit.

— Some students exited and re-entered the current level many times to read again the *WOD* information. We will trigger the *WOD* only once and display a history of the *WOD* interactions somewhere in the GUI so the students can look at the instructions and possible hints whenever they want.

— Nearly all students either asked how they could print values, or refrained from doing so. This shows they understand that Agdbentures is a special environment, but do not know to which extent this modifies the behavior of code they normally use, namely calling the `printf` function. Since printing values is an important debug strategy—often the only one that novice students use—we will add early in the tutorial a level with explanations on how to print values in the GDB console and the C code, showcasing that the output is in the console. This is also done in the current version of Agdbentures after the redaction of this manuscript. We may argue that printing values is not a valid debugging method to teach. However, professional developers still use printing to find bugs, for example, to trace the evolution of a variable value.

— Similarly, there is confusion between what comes from the level source code, and what is added by Agdbentures. In particular, we watch calls to a `message` function in the

level code in order to display strings in pop-up boxes in the graphical window. Since the display is similar to the way the *WOD* talks to the player, this made students believe that such messages were the addition of Agdbentures, while they were, in fact, triggered by the level code. We plan to make changes in the UI to clearly state which messages come from the level and which do not, with two separate histories that save the past messages. This will also solve a problem where students wanting to read again the explanations currently need to quit and re-enter the level.

We are also considering showing in the tutorial how to run the program outside Agdbentures. Even though it is possible, we never show it to the student because they see the path to the files when editing the code. We thought they would want to go into this folder and manually compile the program and run it. This explanation can be done early in the tutorial at a dedicated level or the introduction level. We do not know if this will have a positive impact on students' learning.

Another addition to clarify this difference would be in the *WOD* introduction. When the *WOD* introduce themselves, they could explain that they are not part of the level code and comes from Agdbentures. The student would then be invited to run the level directly without using Agdbentures to see that the program runs correctly but the *WOD* does not talk there.

All these ideas should contribute to showing students what is part of the program and what is part of Agdbentures.

— Another UI issue was that nearly all students wanted to directly edit the code in the read-only window. We plan to solve this issue by having a *vscode* plugin capable of working with Agdbentures, so we only have one window for execution and code edition. Special care must be taken so execution cannot continue after changes have been made unless the level is recompiled and restarted. Also, an IDE window such as *vscode* takes most of the screen space that is already mainly occupied by Agdbentures.

Some students also found a way to solve certain levels in an unintended way, we took note of this and added some checks in the validation phase of the levels.

Students made extensive use of the *magic-breakpoint* to repeat execution quickly and observe multiple times what is happening. This may suggest that this scaffolding feature we added is indeed useful in the early levels of Agdbentures.

During the discussion part of the sessions, all students expressed they enjoyed playing with Agdbentures, many stating the quality was much higher than they expected. All wanted more time to advance to subsequent levels and to be able to install Agdbentures on their personal computers to play in their spare time. Students reported that, compared to debug exercises with bare C code they did in some courses, the usage of gaming elements, but more importantly having visual feedback that really helps to debug, make Agdbentures much more appealing.

Discussions allowed us to confirm that having "easy" levels was not a detractor, even for high-performing students, as they just very quickly solved the first levels, still enjoying them, to arrive at levels more interesting for their skills. It also confirmed that the first levels are easy enough so that struggling students can still progress autonomously in the games—apart from UI problems, no-one got stuck on a level for too long.

It is encouraging that at least for the first levels the bugs are not hard enough to completely block students.

Conclusion on experimental results

The students' adhesion to Agdbentures early levels is encouraging. Even if the current state of Agdbentures is not mature enough to have a perfectly smooth user experience, the changes discovered with this experimental phase sound promising and are affordable with reasonable efforts. Gathering student feedback is essential when designing courses or exercises, there are some issues we could not guess in advance without actually observing students using Agdbentures.

With our simple experimental setup, it is not possible to assert a quantitative improvement in students' debugging skills. We would need repeated experiments over long periods of time. After we take into account the modifications described in this section and add a few more advanced levels, we plan to conduct a larger-scale experiment. We do not know yet how we can quantitatively evaluate the evolution of debugging skills so we might ask students for some self-evaluation.

3.9 Future work

There is still future and ongoing work to do on Agdbentures. We present first the technical improvements we plan.

Firstly, the last available levels need to be tested with students and more advanced levels need to be developed.

The framework to inspect and monitor the level code also supports Python programs (this will be presented in the next chapter). Although it does not use GDB, it supports a common set of commands, which should make it possible to have a version of Agdbentures for Python code. Python is a more common language for CS1 students than C. Obtaining this Python version of Agdbentures should be close to translating level source codes to Python. Some levels are based on bugs specific to C, these levels cannot be translated but we could write new levels with Python-specific bugs.

We have a long-term plan of having all Agdbentures windows encapsulated in a single full-screen window that could be considered as a regular video game. We will work on this when we think Agdbentures is mature enough and just need some engineering improvements and no more levels. We believe this video game environment will make Agdbentures even more appealing but will make developing new levels harder. At this point, we might contact research teams that are used to working with video games and teaching and have the expertise to develop this kind of video game.

Then we also have some ideas regarding the evolution of Agdbentures levels. We can think about adding some levels only requiring program understanding. They can be small questions before each level to ensure that students have a basic understanding of the program source code before any execution.

We currently did not evaluate the pedagogical quality of the levels. We mentioned a larger-scale experiment in the conclusion of the previous section spanning a long period of time. We will have to wait for Agdbentures to become more mature so we could easily distribute it to students.

In the next chapter we will see that our monitoring framework tries to support reverse debugging. Reverse debugging could also be a nice addition to Agdbentures as we saw students extensively use *magic-breakpoints* to quickly replay execution.

Conclusion

In this chapter, we presented Agdbentures and a first experimentation phase with CS1 students. The method diverges widely from the two previous chapters as we now work directly with students instead of working on datasets as a "proxy." This allows us to gather direct feedback and, we can improve our system accordingly.

Agdbentures is not mature yet but most of the intended future work is affordable. The current state of Agdbentures is already promising as even if it is close to a proof of concept, students already find it engaging.

We believe that this way of teaching debugging by developing carefully designed exercises is much more concrete and efficient than what we did in the previous chapters with statistical methods and automatic generation. By spending our time reacting to student feedback, we think the teaching benefit of our work is more direct.

We believe this kind of project is a nice addition to computer science curiculum as mainaining student engagment is necessary to allow continus working and training. Also, students' motivation is benefecial for teachers as it makes them more prone to take some time to develop these projects. In computer science, compared to other subjects, teachers already have part of the expertise needed to develop such projects. We could imagine projects close to video games to learn many other topics as long as teachers have time to design exercises and there relationship with video game elements. However, during Agdbentures development we saw that developing a project with video game elements is outside of generic computer science skills and require a dedicated expertise. We managed to bring Agdbentures to its current state with reasonable self-formation time but we are not sure that all the features we planned for Agdbentures will be in our reach.

Chapter 4

Easytracker and visualization tools for program dynamics

After starting the development of Agdbentures, we realized that monitoring the level execution could be extended to many other uses in computer science teaching. We then put on hold the development of Agdbentures to work on this library that we called *Easytracker*. We used *Easytracker* to develop visualization tools for our own teaching contexts. A lot of engineering work in Agdbentures and these tools was much simpler thanks to this library. In this chapter, we present the motivations and the development of *Easytracker* along with the few teaching tools we made and how we used them in lectures.

4.1 Motivations and background

Computer science teachers use visual representations extensively to illustrate their lectures, especially when teaching programming. Such representations may be either hand-drawn or automatically generated. The *Computer Science Education* community sometimes considers these representations as the visible part of what is called a *Notional Machine* (NM) [91, 92] and defined as *"a pedagogic device to assist the understanding of some aspect of programs or programming"* [93]. Such machines are widely used nowadays [93]. One example is Python Tutor [90], which is not a Notional Machine per se but a generic visualization tool for building Notional Machines. These Notional Machines are usually related to the concept of "program dynamics" [94]. This refers to the difference between the source code of the program and its actual execution on a machine as a dynamic physical entity. It is a crucial concept when learning programming and especially in debugging.

While we can all acknowledge the positive impact of hand-drawn representations on the learning process, we believe that NMs could benefit from automatically generated representations. Indeed, such generated representations are helpful for :

— answering "what if questions" asked in class with live demonstrations

— generating images and videos to be used in the material complementing/replacing lectures

— visualizing real-size problems.

— empowering learners with the ability to self-validate their understanding of the concept the representation focuses on. For example, learners can compare the representation generated by a visualization tool with the one they have in mind without the intervention of the teacher.

The success of Python Tutor with its 10M users in the last decade [95] supports the claim that generated representations are useful. Unfortunately, no generic tool can fit all the specific visualization needs of all particular teacher and student audiences. Moreover, implementing tools that automatically generate custom visual representations currently requires quite a high level of programming commitment from teachers. Said differently, if existing generic tools such as Python Tutor do not fulfill a teacher's need, one must build a tool from scratch to generate the expected visual representation.

In this chapter, we present a Python library called *Easytracker* to assist teachers in building tools able to generate visual representations of a running program. As detailed in Section 4.2, *Easytracker* allows its user to:

— control the execution of a program referred to as the `inferior`, written either in Python or any GDB-supported language

— pause the program and inspect its state at any time of its execution

— build a custom visualization from several provided basic building blocks.

Figure 4.1 shows how a visualization tool interacts with the different layers of *Easytracker*.

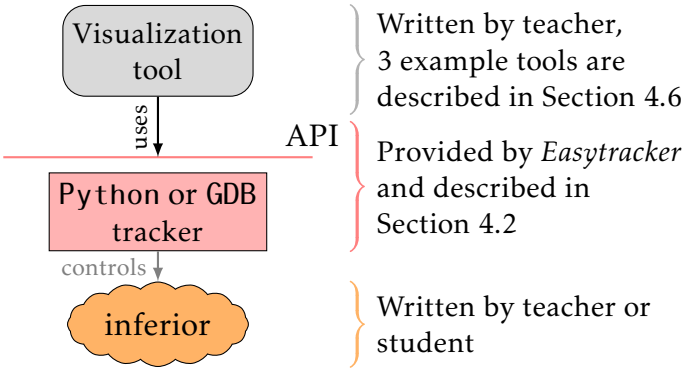


Figure 4.1 – A visualization tool (in gray) is built on top of the control and inspection features (in red) and visualization blocks (in green).

Easytracker comes with a two-fold API to control, inspect and visualize the execution of a program, and two implementations that we call *trackers*: the first one dedicated to Python inferiors, and the second dedicated to inferiors written in GDB-supported languages.

4.1.1 Other visualisation tools

The Computer Science Education community has been working on program visualization for a long time, investing in numerous high-quality visualization tools [96]. While some of them, like Python Tutor [90], are widely used in programming courses, most of them are by design challenging to adapt and extend to fit the teachers' specific needs. That is why the

approach of *Easytracker* is to help teachers design the visualization tools they need, rather than providing a generic all-in-one program visualization software.

Let us refer to Python Tutor [90] to understand the reasons behind *Easytracker*. Python Tutor is a widely used generic online visualization tool. Teachers can use it for building Notional Machines in a particular context with a given focus. Also, despite its name, the tool now supports more programming languages like Java, C, or JavaScript. In Python Tutor, the end user executes its program line by line and sees both the evolution of the stack and the heap.

However, Python Tutor users are stuck with some limited options to represent frames and memory. For example, if a specific teaching use case imposes to represent the memory differently, one would have to accept the daunting task of extending Python Tutor to do so. The philosophy behind *Easytracker* is different, as it provides teachers with building blocks to build the visualization tools that precisely fit their needs. As an example, we used *Easytracker* to build a notional machine picturing the evolution of the call tree for the execution of a recursive function (see Subsection 4.6.3). We also made a similar tool for RISC-V architecture, to observe memory values and registers (see Subsection 4.6.2). Python Tutor does not provide any way to do so. Of course, this specialization comes at the price of having to write some Python code. Nevertheless, as will be shown in Subsection 4.2.3, this Python code is pretty straightforward to write.

On the Notional Machine front, *Easytracker* can be used to automate the generation of the visual part of the notional machine. Using *Easytracker*, a teacher can quickly write a program controlling and inspecting the state of a given Python or GDB-supported program. As a consequence, the only remaining part the teacher needs to write is the visualization of the program state reported by *Easytracker*. For example, a teacher can write a tool to generate diagrams for the stack and the heap after the execution of each line in the source code or at any particular steps in the program [97, 98]. More generally, many of the notional machines categorized by Fincher et al. [93] can have their visual representation automatically generated using *Easytracker* since they focus on showing a particular part of the state of the program.

4.2 Easytracker interface

The interface of *Easytracker* comprises two parts: the *control* interface and the *inspection* interface. The first one, the control interface, provides ways of pausing and resuming the execution of the inferior program. The second one, the inspection interface, focuses on features related to querying a paused program state.

We designed the control and inspection parts of *Easytracker* with a debugging session in mind where the user controls the execution of its program, pauses it under some conditions and inspects its state. This choice allows the programmer to write its visualization tool in an *imperative-like style* where a call to a function from the control interface does not return until the execution of the inferior pauses. In other words, every time the control flow gets out of the control interface, the tool can safely query the state of the inferior using the inspection interface.

Features mentioned in the interface definition are agnostic of the language used to program the inferior. Each tracker has a few specific features, for example, the GDB tracker has a function to access register values. We try to limit as much as possible the number of these features so the interface is as language agnostic as possible.

4.2.1 The Control Interface

The control interface provides the following functions to indicate when *Easytracker* must pause the execution of the inferior.

```
def break_before_line(lineno, maxdepth=infty) -> None:
  """pauses the inferior before executing line lineno"""
def break_before_func(func, maxdepth=infty) -> None:
  """pauses the inferior when entering func"""
def watch_function(func, maxdepth=infty) -> None:
  """pauses the inferior when entering/exiting func"""
def watch_variable(variableId) -> None:
  """pauses the inferior every time the value of
     the variable referenced by variableId changes"""
```

Figure 4.2 – Functions to indicate when to pause the inferior

The `break_before_line` and `break_before_func` functions inform *Easytracker* that the inferior must be paused just before executing a given line or just before entering a given function. Returning from `break_before_func` guarantees the arguments are initialized, hence accessible, when the inferior is paused.

`watch_function` informs *Easytracker* that the inferior must be paused at the beginning (just after entering) and at the end (just before returning) of every execution of `funcname`. `watch_variable` makes *Easytracker* pause the inferior every time the variable identified by `variableId` is modified.

One can use the `maxdepth` optional parameter to tell *Easytracker* to pause the inferior only if the current frame depth is below a given value. Like a debugger, the control interface also provides functions to start/resume the execution of the inferior as described in Figure 4.3.

```
def start() -> PauseReason:
   """starts the inferior and immediately pauses it"""
def next() -> PauseReason:
   """executes one line without jumping into functions"""
def step() -> PauseReason:
   """executes one line with jumping into functions"""
def resume() -> PauseReason:
   """resumes until a pause condition has been reached"""
```

Figure 4.3 – Functions to start and to resume the inferior

Each of these functions returns an instance of `PauseReason`, a class representing why *Easytracker* paused the inferior program. We set a priority for each of the possible pause reasons so the functions above return the condition with the highest priority that triggered the pause. Here is a list of the possible pause reasons, sorted by priority in descending order :

— The inferior exited.

— A watched variable has been modified, or we have reached the boundary of a watched function.

— A function breakpoint has been hit.

— A line breakpoint has been hit.

— The end of a single-stepping control command (`start`, `next` or `step`) has been reached.

This priority list will guide the tracker implementations, higher priority pause reasons will be checked first.

Finally, the functions of the control interface return only when the inferior is paused again or terminated.

4.2.2 The Inspection Interface

This interface defines how a tool can observe the current state of a paused inferior program. Figure 4.4 describes the functions called to know where in the source code the inferior has been paused. Figure 4.5 lists functions users can call to get frames, global variables or to recover the inferior exit code.

```
def get_last_lineno() -> int:
  """returns the number of the last executed line"""
def get_next_lineno() -> int:
  """returns the number of the next line to execute"""
```

Figure 4.4 – Functions related to the inferior source code

```
def get_exit_code() -> int:
  """returns exitcode or None if inferior still running"""
def get_current_frame() -> Frame:
  """returns the innermost Frame (the deepest one)"""
def get_global_variables() -> List:
  """returns a list of Value for global variables"""
```

Figure 4.5 – Functions to inspect frames, variables and to get the inferior exit code

Partial description of the generic memory model through two examples We will use two similar programs in C and Python respectively and observe how they are modeled. We choose really short programs because the model is quite verbose and needs to be shown in single figures in this manuscript but the model is fully recursive and can represent arbitrary data structures.

These programs can be seen on Figure 4.6. The C version uses a pointer to increment a value in `inc` function and the Python version returns and assigns the result.

For development purposes, we implemented a debug tool that steps through the program and dumps a visual representation of the model at each step. We will look at these model representations in several steps to show how it is described. We use small programs because the whole model is dumped, the more complex the program is, the bigger and more verbose the model is.

```python
def inc(a: int):
  return a+1

def main():
  a = 5
  b = 6

  a = inc(a)
  b = inc(b)
```

(a) Python version

```c
void inc(int* a) {
  *a += 1;
}

int main() {
  int a = 5;
  int b = 6;

  inc(&a);
  inc(&b);

  return 0;
}
```

(b) C version

Figure 4.6 – Two programs that increments variable a and b through a function call.

On Figure 4.7 on Page 86, we see both representations at the program startup. We will take some time to describe the different types of nodes in our model.

The rectangle nodes represent stack frames, this is the `Frame` type in our model. `Frame` is the most significant type the *Easytracker* inspection interface defines. It represents a stack frame of the program execution and is composed of a set of local variables and an optional parent frame. This is the root of the generic memory model we define in *Easytracker* interface. `Frame` objects contain the function name, a depth in the stack (the main frame has a depth of 0), and the next line to be executed in the frame.

Before describing the other nodes, we can see that the two representations are different at startup. The C version looks natural, with the main frame and a and b variables not initialized (the memory is sometimes initialized at 0 by GDB or the compiler in debug mode). The next line (and the first) to be executed is line 5 which is the first assignment `a = 5;`.

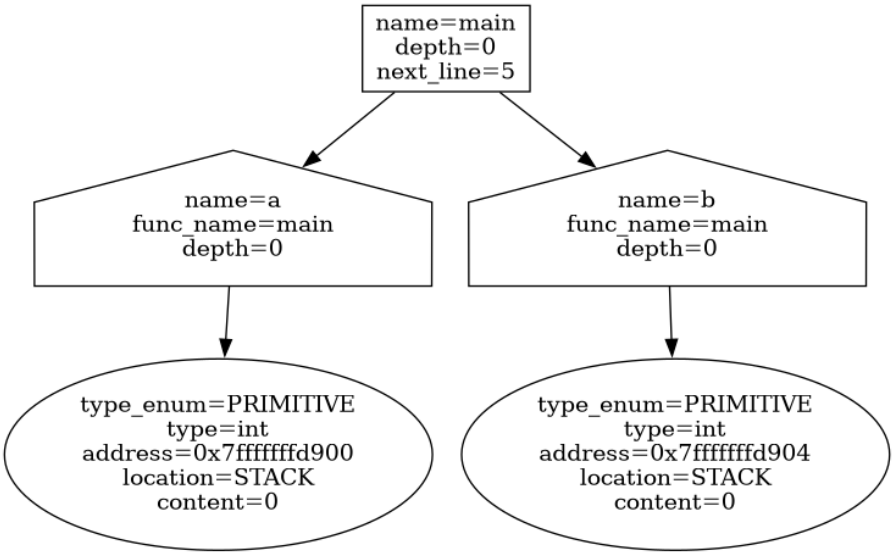
The Python version, however, is less intuitive at startup. Our debug tool even helped us to precisely understand how the Python interpreter sets up variables when calling a function. First, we see that functions are regular global symbols with a `function` type, they are the "house" like nodes with a name `inc` and `main`. In Python all symbols are references to actual Python objects, this is also the case for function names. There will be more detail on how the Python interpreter memory is explored in Section 4.4.

Also at startup, the `main` function is already entered (because we can see the `main` frame object) but the local variables are not already set up. They have to be assigned first before being in the stack. There is a step to assign function arguments before executing a function. This is a technical detail that explains why the next line to be executed is line 4 on the Python example.

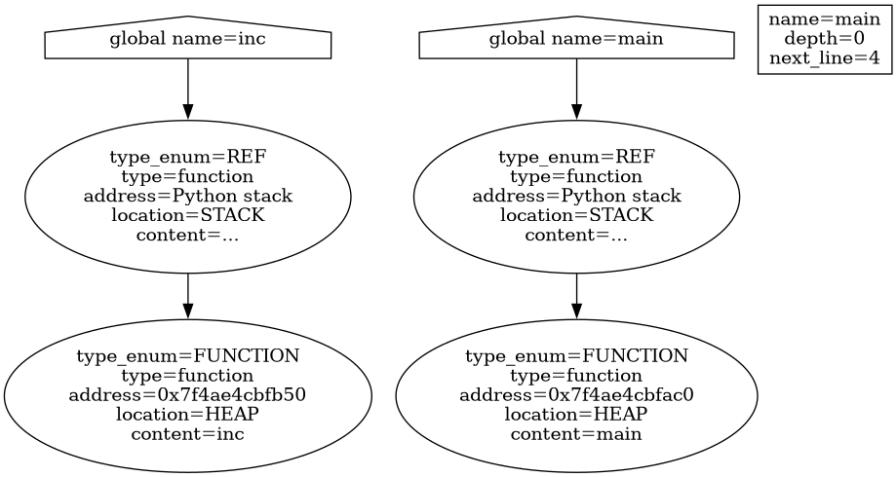
We are not interested in the global function symbols in Python for our explanation, from now on we will remove them from the representation even if they are still present in the debug tool.

To observe a closer representation from C, we can see on Figure 4.8 the memory graph for Python just after a and b are initialized (they are initialized in a single step).

One of the main objectives of the inspection interface is to be able to represent and in-



(a) C version.



(b) Python version, functions are seen as global symbols.

Figure 4.7 – The memory model representation for both programs at startup.

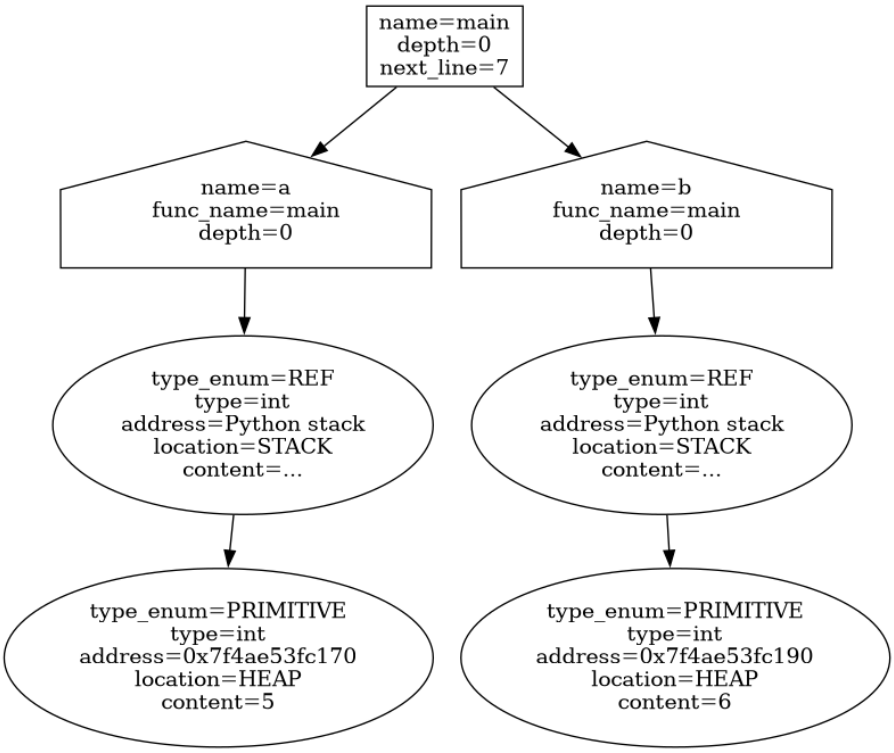
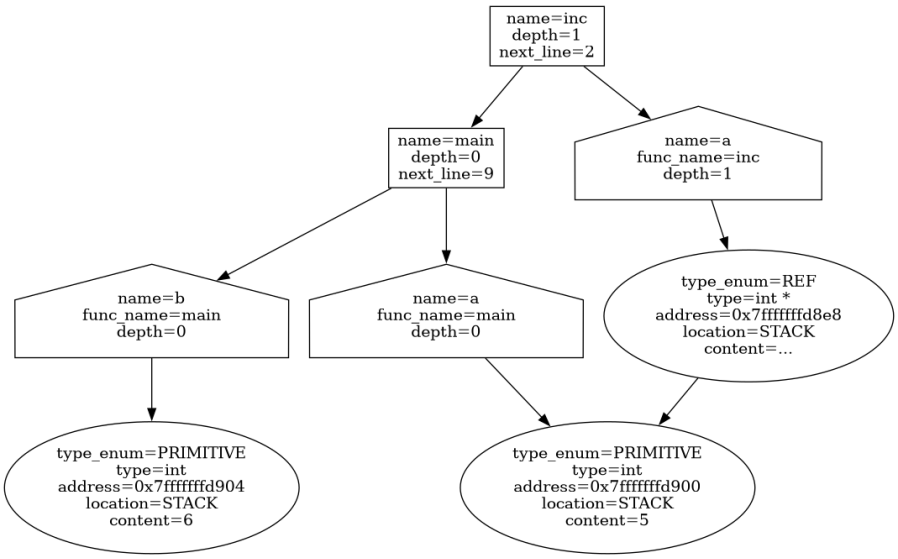


Figure 4.8 – Python version after the first variable initialization (global function names are omitted).

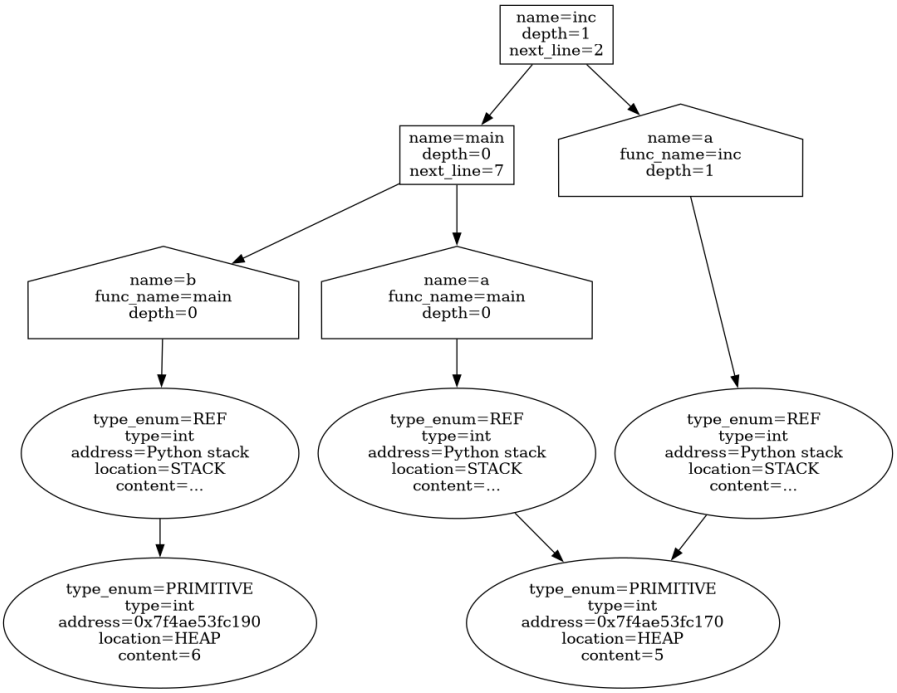
spect the value of a variable regardless of its type and the programming language used by the inferior. To do so, *Easytracker* defines the `Variable` data type to represent any variable of the inferior program *in a language-agnostic way*. It is composed of two main attributes: the variable *name* stored as a Python string and its *value* represented by the `Value` data type, encapsulating a location, an address, an abstract type and the content. These are the "house" shape nodes on the representations we've seen so far. The depth of the stack frame the variable lies into is duplicated with the information given in the `Frame` object. `depth` is an optional parameter because global variables do not live in a stack frame and thus do not have a definition of depth in the stack. We can also note on Figure 4.8 that in Python we cannot know, the address of the name binding in the stack so we give the special value `Python stack` to the name binding address.

In a `Value`, the location attribute indicates whether the value lies on the stack, on the heap or is a global part of the memory. The address attribute represents the memory address where the value is stored. The abstract type attribute represents a language-agnostic type category. For example, this type can be referred to as *primitive* (e.g., a Python `int`, a C `float`), *reference* (e.g., all Python variables, a C-pointer) or *list* (e.g., a Python `list` or `tuple`, a C array). It can also represent a custom data structure (e.g., a user Python class, a user C structure). Finally, the `content` attribute, which data type depends on the value's abstract type, contains the effective value. For example:

— The content of a *primitive* `Value` is a primitive Python object: `int`, `str`, `float` or `bool`.



(a) C version.



(b) Python version (global function names are omitted).

Figure 4.9 – The memory model representation for both programs when the `inc` function is called for the first time.

— The content of a *reference* `Value` is a `Value` object.

— The content of a *list* `Value` is a Python `list` containing `Value` objects.

— The content of a custom data structure is a Python `dict` mapping the name of each field represented by a Python `str`, to its corresponding `Value` object.

We will observe the model representation when they are "the most complex" in these simple examples (i.e., when we are in the `inc` function so there are two stack frames). They can be seen on Figure 4.9.

The representation may look very similar for both programs because both variables in the `inc` function are references but they are conceptually really different. In C, there are two variables named `a` in the stack, one in the `main` function and one in the `inc` function. The one in `inc` is a pointer to the other one. Visually, the two arrows do not mean the same. The pointer one means "It points to the memory location of the other variable." The other one means "The content of this variable is the following memory location." As we told, the `Value` type is essentially a typed memory location.

In Python, the variables do not hold any "value" per se, they are only name bindings and we represent these bindings as a reference to an actual memory location.

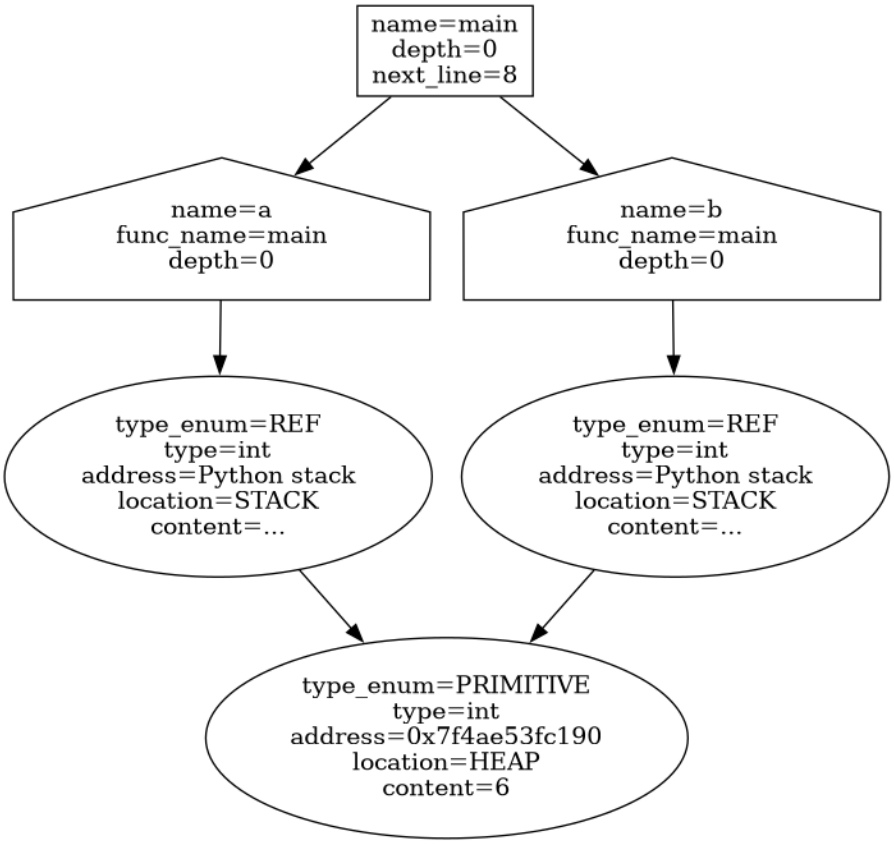


Figure 4.10 – Python version after the incrementation function returned (global function names are omitted).

We see on Figure 4.10, that after the first incrementation, both `a` and `b` values are 6. In

Python the value "6" is a single object in memory and variables are name bindings so they both point to the same object (this will not be true for higher integers and we could see separate allocations above 512). In C, they are two different stack locations so both values will be 6 but they would still be different memory nodes.

Directly accessing the memory in the Python tracker In the Python tracker, *Easytracker* runs in the same interpreter as the inferior (see Section 4.4). Of course, variable values can be accessed through the previously described model. But it can be directly accessed as if the user was manipulating Python memory directly. If teachers wish to write a tool only for Python and use the Python tracker, this is actually a really powerful feature as they can directly read the memory.

Special feature to obtain a Python representation of the memory in the GDB tracker The generic program memory model is designed to be used by visualization programs. However, one may be interested only in the values of variables because they already know what the memory should look like. This is the case in Agdbentures for example. We already know what variables are present in the code and we want to retrieve their value so we can do visualization accordingly.

To this end, we added a special feature only in the GDB tracker. We abstracted the exploration code from Figure 4.19 on Page 98 so we could write another version that builds direct Python values instead of our `Value` objects from the interface. For example, in C, an integer variable named a in the current frame with a value of 5 will be represented as the direct Python value 5.

```
# accessing the value with the model
# we select the value of the variable a in the last frame
a = stack_model[-1].variables["a"].value.content
# accessing the value with the direct access mode
# this code is equivalent to the previous line
a = direct_stack[-1]["a"]
```

Figure 4.11 – Different access to an inner variable value between the model and the direct access.

The different usages between this Python representation and the model can be seen on Figure 4.11. The more complex the data structure is, the shorter and easier the code that uses this direct representation.

We had to define a Python equivalent for every C type. This is straightforward for many C types. We converted C structures to dynamic Python objects. We converted any pointers in C to tuples of size 1 in Python so we do not lose the reference.

4.2.3 A Simple Inspection At Each Step Example

Figure 4.12 shows how we use the interface to implement our language-agnostic stack and heap visualization tool described in Section 4.6. After executing each line, this tool stalls the program and generates one image representing both stack and heap current states. The

function `draw_stack_heap` is not detailed and takes as input the program stack referenced by the topmost `Frame`. It generates a `dot` image of the memory state.

```
1  inferior = sys.argv[1]
2  tracker = python_tracker() if ".py" in inferior else gdb_tracker()
3  tracker.load_program(inferior)
4  tracker.start()
5  img_count = 1
6  while tracker.get_exit_code() != 0:
7      frame = tracker.get_current_frame()
8      draw_stack_heap(frame, f"img{img_count}.svg")
9      tracker.step()
10     img_count += 1
```

Figure 4.12 – Code of our language-agnostic stack and heap visualization tool that steps through the program and generates one image after execution of each line.

Line 1 gets the inferior the tool will execute. Lines from 2 to 9 set the tracker to use. The tool then loads the inferior on line 3 and starts executing it on line 4. The control loop on line 6 is typical of many *Easytracker* tools. In this example, the tool steps through every line of the inferior on line 9, but it could define higher-level pause conditions such as watching a variable and calling the resume function. The program's state gathered thanks to the `get_current_frame` function on line 7 is drawn in an image using the visualization interface on line 8.

All the work has to be put in the drawing, that is all we have regarding the monitoring side. The `draw_stack_heap` function is actually the entry point of a whole module. This will be detailed in the visualization section (see Section 4.6).

Easytracker comes with two implementations of the interface described in the previous section. One is for tracking Python code, and the other can track a program written in any compiled language supported by GDB like C. We wrote both these implementations in Python and they are described in the two following sections.

4.3 GDB Tracker implementation

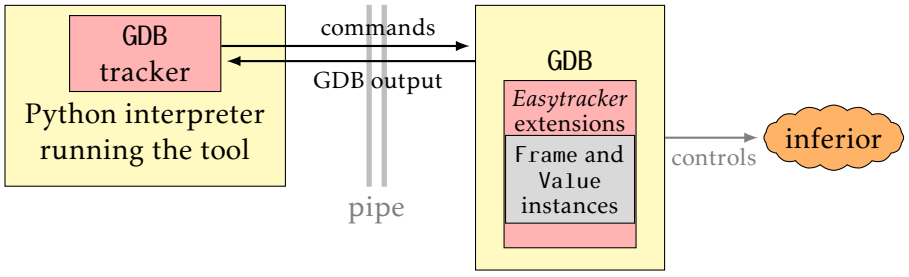


Figure 4.13 – The GDB tracker: yellow boxes are processes and red ones are the tracker implementation

As the name suggests, the GDB tracker is based on GDB and relies on custom extensions

we provide, as shown in the right part of Figure 4.13. These extensions to GDB are made using its Python interface (`https://sourceware.org/gdb/onlinedocs/gdb/Python-API.html#Python-API`). Several GDB internal objects like frames, values or breakpoints for example are accessible through Python scripting. We can alter these objects' behavior or access their internal values. These scripts are run by a dedicated Python interpreter inside GDB.

The GDB tracker runs GDB as a subprocess in Machine Interface mode (MI). The former interacts with the latter through a pipe. To send a command to GDB, the tracker writes the command to the pipe.

4.3.1 Abstracting the MI interface

We use an external Python library called *pygdbmi* that abstracts some of the previously mentioned engineering. The library is in charge of:

— Launching and killing the GDB subprocess.

— Send commands to GDB through a pipe.

— Parse MI outputs coming from the other pipe to clean and human-readable Python dictionaries.

Commands are sent to GDB with a timeout, when the timeout finishes, the library reads the GDB output pipe and returns the GDB response to the command. It is also possible to manually read the GDB output pipe with *pygdbmi* if the previous call could not read all the GDB output. To keep communication time with GDB reasonable, we implemented a waiting loop with an increasing timeout (see Figure 4.14 for a pythonous pseudo-code of the loop).

```
def synchronous_send(command: str) -> list[dict]:
    timeout = 0.01 # seconds
    response = pygdmi.send(command, timeout)
    # check in the response if we found an end of command token
    while not command_ended(response):
        # we progressively increase the timeout to avoid burning CPU for long commands
        if timeout < 2:
            timeout = timeout * 2
        response.append(pygdmi.manual_receive(timeout))
    return response
```

Figure 4.14 – GDB communication loop using *pygdbmi* interface.

This implementation is reasonable and covers many use cases. Under the hood, `pygdbmi` uses the `select` Linux mechanism on the communication pipes. It allows the process to be stopped and woken up after a given timeout or as soon as some data is available on the pipe. This avoids burning a lot of CPU if we would need to manually poll the pipes.

However, if a program has an intensive use of *Easytracker* (like Agdbentures for example) the current implementation would hurt performances. In Agdbentures, we could see some delays between the `step` command being triggered and the player moving on the screen if we do not wait between `step` commands. This delay comes from the timeout needed to stop waiting for new data. The whole timeout is consumed by the `select` loop inside `pygdbmi` before we can check if the command ended. We could use small timeouts like 0.05s but this

will burn more CPU and it would still be a minimum response time that we cannot decrease. It would be nice to get rid of this minimum response time/CPU burning tradeoff.

4.3.2 Modification of *pygdbmi* to remove the minimum response time/CPU burning tradeoff

Unfortunately, *pygdbmi* is tied to this timeout feature because of the `select` mechanism. It does not support asynchronous response natively, by asynchronous we mean returning a response as soon as it is available without using timeouts. We would need to listen for new data on the pipe in the background and not in the main *Easytracker* thread (hence the asynchronous terminology). This change was requested on *pygdbmi Github* repository (`https://github.com/cs01/pygdbmi/issues/41`). After the changes were made locally, a pull request was sent to the library repository and is still waiting for approval [1].

Having a completely asynchronous implementation needed some changes in *pygdmi* interface. Instead of calling functions to read and write to GDB pipes, we write to native Python communication queues (`Queue` objects in the `queue` module). The library users (in our case *Easytracker*) can choose if they wish to perform blocking or non-blocking reads of the GDB output queue, this is a native feature from the `Queue` Python objects. For example, the loop from the previous section becomes:

```
def synchronous_send(command: str) -> list[dict]:
    pygdmi.command_input_queue.write(command)
    responses = []
    while True:
        # blocking read on GDB output
        response = pygdmi.output_queue.read()
        responses.append(response)
        # we check elements one by one looking for the command end token
        if command_ended(response):
            break
    return responses
```

Figure 4.15 – GDB communication loop with modified *pygdbmi* asynchronous interface.

Having a synchronization with command ends and packing responses to the same command together is still a useful feature even in asynchronous mode. We see on Figure 4.15 that no timeouts are used and GDB outputs are consumed whenever they are ready. This greatly increased *Easytracker* performance and allowed to have the implementation of Agdbentures without noticeable delays when communicating with GDB.

Internally, we spawn three threads to communicate with GDB standard IO streams. The IO streams have to be configured in a blocking way and are line-buffered.

— The first thread is for GDB standard input. It makes some blocking reads on the command input queue. After a command is read, it is written on GDB standard input. A pseudo-code implementation can be seen on Figure 4.16.

1. even if there is still no answer from the library maintainers, the publication of a pull request could help another user of the library so they could access our modified version.

— The second thread is for standard output. It makes blocking read on GDB standard output stream. This is not a problem as it is encapsulated in a thread, it will not prevent other parts of the program from running. Once some output is read from GDB, it is written in the output communication queue from *pygdbmi*. A pseudo-code implementation can be seen on Figure 4.17.

— The third thread is for the standard error. It works the same as standard output but on the standard error stream. There is a flag in the output queue from *pygdmi* to tell if the output comes from the standard output or standard error.

```
def write_thread_fn(gdb_input_stream: Pipe, input_queue: Queue):
    while True:
        command = input_queue.get() # blocking read
        # to stop this thread, None has to be sent into the input queue.
        if command is None:
            break
        # we add a new line after the command so the input is flushed
        gdb_input_stream.write(command + "\n")
```

Figure 4.16 – Write thread implementation added to *pygdbmi*.

```
def read_thread_fn(gdb_output_stream: Pipe, output_queue: Queue):
    while True:
        output = gdb.output_stream.read() # blocking read
        # if the stream is closed None is returned
        if output is None:
            break
        output_queue.put(("stdout", output))
```

Figure 4.17 – Read thread implementation added to *pygdbmi*. The standard error thread is the same implementation but with a different stream argument and stream name inside the output queue.

The threads have to be manually stopped in *pygdmi* when *Easytracker* exits. The reading threads are closed when the GDB process exits and the streams are closed, but the writing thread has to be closed by sending None in the input queue.

4.3.3 Sending Python objects through the output pipe

Many of the mechanisms described in this section are implemented directly inside GDB through plugins (the *Easytracker* extensions box on Figure 4.13). Sometimes we have to get output from this plugin. If this output is textual, we can read it from the standard output and parse it. However, sometimes the outputs are a Python representation of the full program memory (as in Subsection 4.3.5) and are quite cumbersome to parse. These outputs are full-fledged Python objects and we have to transfer them between two different Python interpreters (the main interpreter which runs the tool and the tracker and the GDB Python interpreter that runs the plugins). We use Python native module *pickle* to send them through

GDB standard output. Using a Python native module also brings some compatibility when the Python versions between the two interpreters are slightly different as this module is reasonably backward-compatible. The process is quite simple and as follows:

1. Pickle the Python objects in a binary format to Python `bytes` objects.

2. This sequence is encoded using base 64 (again with a native Python module). The sequence will be printed on the pipe so its string representation will be used. This encoding step is used to have a character representation of the `bytes` sequence. This is still not human readable but we can better see whether two different representations are equal during development. Basically, a character in base 64 is 8 bytes of binary data. A character in the base 64 encoding is represented as 8 ASCII characters. To give an idea of the use of this step, if we pickle the Python object 5 the raw binary representation as it would be printed on the pipe is `b'\x80\x04K\x05.'`. The `b"` means that the string is a binary string. After the base 64 encoding this sequence is represented as `b'gARLBS4='` (still binary). This encoding may not be mandatory, but we do it to avoid directly sending binary data into the pipe.

3. The base 64 string is sent on GDB standard output so it can be received by *Easytracker* on the other side of the pipe.

4. The base 64 string is decoded and the byte sequence is unpickled to obtain the Python objects in memory.

The process is easy to encapsulate. However, it adds the constraint that every object that we want to send through the pipe needs to be pickle-able.

4.3.4 Program control

GDB already provides almost all the control commands the *Easytracker* control interface requires. Implementing these commands in the tracker then boils down to calling the proper function from the GBD MI interface. Two features nevertheless are missing in GDB for implementing the *Easytracker* control interface:

— The first one is to implement the `maxdepth` semantic described in the interface section. We implemented it as a Python-based GDB extension, adding custom breakpoints commands that take an additional `maxdepth` parameter. We can override the *stop* function of breakpoints that are called when the breakpoint is hit. If the current frame depth is deeper in the stack than the `maxdepth` argument, then we resume the execution instead of stopping and triggering a breakpoint hit.

— The second one is the *watch_function* functionality. It is easy in GDB to place a breakpoint at the beginning of a function; however, GDB provides no functionalities to place it when a function returns. There is the *finish* command that stops when the current function returns but this does not place a breakpoint, so if the program is interrupted on the way it will not stop later when the program reaches the end of the function. We also cannot place a breakpoint just after the call because we still want to access frame information and variable values so a breakpoint has to be placed inside the function. Luckily, it is possible in GDB to disassemble some code blocks.

When we enter the watched function for the first time, this event is caught in our GDB extension. We disassemble the function code and look for *retq* x86 assembly instruction that returns from the function. We can place a breakpoint at the address of this instruction. Luckily, it is a common practice in compiler designs to write a single function epilogue and thus a single *retq* instruction. However, our design becomes restricted to x86 architecture so *Easytracker* will not have this feature on other architectures. We could still implement the same mechanism for different architectures and choose the condition to find the return instruction depending on the actual architecture.

Implementing the imperative-style control interface requires nevertheless some work. As a call to a function from the control interface should not return until the execution of the inferior pauses, the tracker needs a way to detect the completion of the control command. This is the main use of the `synchronous_send` function described in Figure 4.15.

4.3.5 Memory inspection

Regarding the implementation of the inspection interface, GDB only provides some printing ability. Consequently, we extended it with a custom inspection command that recursively explores variables and builds a graph of `Value` instances and the frames in the stack as `Frame` instances. To that end, we again use the GDB Python interface but this time to access each memory location's type and value.

Recursive exploration We will describe in pseudo-code the algorithm to explore memory values in Figure 4.18 and Figure 4.19. GDB has an internal *gdb.Value* type that represents a typed memory location. We distinguish it from our own `Value` type also representing a typed memory location but with a different interface (see Subsection 4.2.2)that is not part of *gdb* module.

Figure 4.18 is a simplified version of the real implementation but the general algorithm is quite straightforward. For example, static and global variables are not in the same code blocks so both of them have to be explored (code blocks are a hierarchy in GDB to store symbols, it roughly corresponds to scopes in the source code). We omitted the static variables exploration in the example, the code is the same as global variables but we access the `static_block` attribute of a frame object block (i.e., scope) in GDB. We consider these two types of variables to be global variables in the sense of *Easytracker* interfaces. Otherwise, this algorithm is about using GDB Python API to list global variables and frames and explore each symbol value with the algorithm in Figure 4.19.

We see in the `inspect_value` function, that we cache memory values that we have already explored to avoid infinite loops. This cache is also indexed with types because two pointers can have different types and point to the same memory location.

An example of this could be an integer array and another pointer referencing an element in this array. If the reference pointer is explored first and is the first element of the array, the second exploration for the whole array will be a cache hit and will not be explored (see Figure 4.20 for a visual representation of this simple example).

In the `build_value` function we convert GDB values to our `Value` type. All this conversion code should not be tied to C code. We use GDB generic API so any language supported by GDB should be reasonably supported by this memory exploration feature. We still tested many of our implementations with C programs and use C when reasoning and explaining

```python
def build_memory_graph() -> (list[Frame], list[Variable]):
    # we get the deepest frame in the stack from GDB
    current_frame = gdb.get_newest_frame()

    # we first explore the global variables
    global_symbols = current_frame.block.global_block
    global_variables = [build_variable(symbol) for symbol in global_block]

    # then we explore the stack
    stack = []
    while current_frame is not None:
        variables = [build_variable(symbol) for symbol in current_frame.symbols]

        frame = Frame(name=current_frame.name, variables=variables)
        stack.append(frame)

        # the main frame has no parent frame and this attribute will be None
        current_frame = current_frame.parent

    return (stack, global_variables)

def build_variable(symbol: gdb.Symbol) -> Variable:
    gdb_value = gdb.evaluate(symbol)
    variable = Variable(name=symbol.name, value=inspect_value(gdb_value))
    return variable
```

Figure 4.18 – Building the memory graph.

corner cases. We detailed in pseudo-code the implementation for primitive types (integer, characters, floating-point, boolean types) and generic pointers.

Every type is handled manually, we will see some of them:

— **arrays.** After the size of the array is computed from the memory footprint of the array and the footprint of an element. Each element is individually explored.

— **structures.** We can obtain individual field names and values from GDB. We build a dynamic Python object with `Struct` class name. The attributes of this class are dynamically added from the exploration of each field.

— **strings.** We had to write a special case for null-terminated strings. In general string type is `char*` and the programmer knows that there are some other characters after it even if the string is statically allocated. However, in our case, we cannot differentiate a null-terminated string from a simple pointer to a single char without user intervention. We decided that the default behavior would be a null-terminated string for `char*` type. So when we encounter a `char*` type in the exploration, we explore the memory iteratively until we find a null value.

After all this memory graph is built, the `Value` and `Frame` instances live inside the memory of a Python interpreter embedded into GDB. They then need to be transferred through the pipe to the memory of the Python interpreter running the tool. We use the mechanism described in Subsection 4.3.3.

```python
# We have a global cache to avoid reference cycles
# The cache is indexed by (address, type)
CACHE: dict[(int, str), Value] = {}

def inspect_value(value: gdb.Value) -> Value:
    # check in the cache if the value is already explored
    key = (value.address, value.type)
    if key in CACHE:
        return CACHE[key]

    output = build_value(value)

    CACHE[key] = output
    return output

def build_value(value: gdb.Value) -> Value:
    if value.type in PRIMITIVE_TYPES:
        return convert_primitive_value(value)

    elif value.type == POINTER:
        try:
            deref_value = value.dereference()
        # GDB has an exception we can catch if we dereference invalid pointers.
        # Thus, we can avoid segmentation faults when exploring.
        except gdb.MemoryError:
            # We have a special Value object to represent invalid memory
            return INVALID_MEMORY(value.address)
        # We build a special pointer value
        return PointerValue(inspect_value(deref_value))

    elif value.type == ARRAY:
        ...

    elif value.type == STRUCT:
        ...
    ...
    else:
        raise UnkownTypeError()
```

Figure 4.19 – Recursive `inspect_value` function to explore memory values.

Handling heap allocation During memory exploration, we omitted an explanation about dynamic allocation that can often happen in C and GDB-supported language. Again if we take an integer array example. If this array is heap-allocated, we will only have access to `int*` type and we will not know the array size. This is a problem because we do not know how to explore the array elements if we do not know the array size and element size.

We wrote a small library to override the dynamic allocation functions `malloc`, `free`, `calloc`, and `realloc`. Our functions simply call the corresponding functions in the standard library, but before returning, they assign their arguments and return values to local variables. This allows us to fetch these values in GDB. We silently update a global list of heap-allocated blocks and their size (with internal breakpoints that do not stop the program but trigger some user-defined hooks). This list is maintained and updated without user intervention so

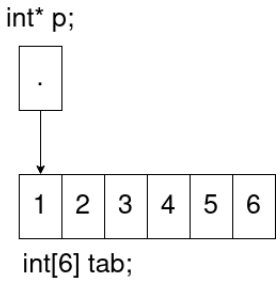


Figure 4.20 – Small example where two values with different types can be explored on the same address. If p is explored first, the type in the cache will be `int`, if `tab` is explored first it will be `int[6]`. We add the type to the cache key so we do not have this problem and the two values are distincts.

our code in Figure 4.19 can know if pointer values refer to heap-allocated blocks and know the corresponding size.

To use our small library instead of standard library dynamic allocation functions, *Easytracker* sets the `LD_PRELOAD` environment variable. Our library will be loaded before the C standard library so if a client code calls `malloc`, our malloc will be transparently called instead of the standard library version.

4.3.6 Handling inferior program standard IO

Some work has to be done to have clean programmable access to inferior standard IO.

standard output and error The standard output and error of the inferior program are regular MI outputs marked as output from the inferior. So handling standard error and output is quite straightforward in theory. However, the output buffering of the inferior program is not the same as if they were launched in a regular tty. If we do not do anything, the output in the command line MI GDB will be printed after each line. However, when GDB is run in a subprocess, the output is printed only when the program exits unless the standard output stream is manually flushed in the code. This is a mechanism from the operating system that sets the stream buffering depending on whether the process is run in a `tty` (like in GDB in a terminal) or not (like in GDB as a subprocess). This is not a desirable behavior because inferior code may not specifically flush output streams and we still want to observe program output as if it was run in a terminal.

We manually set the standard output buffering to line-buffered before the inferior is run. This is done quite transparently using C `__attribute__ ((constructor))` attribute. Any function with this attribute will be run before the `main` function. We again compiled a small library that is loaded with `LD_preload` like the heap allocation tracking library.

standard input Actually, the standard input also has to be set to line buffering before executing the `main` function. However, when running GDB in the command line, GDB knows when the inferior is waiting for standard input or not and redirects input from the command line to the inferior. This behavior is also not the same when running in a subprocess so we have to emulate this.

We have a small function that does not require *Easytracker* to be stopped that directly writes to the GDB input stream. Using our synchronous loop from Figure 4.15, we know if the GDB inferior is running. If the inferior is running we assume it is waiting for input so we directly write to GDB standard input and this will be forwarded to the inferior. If the inferior is not running, giving this input to GDB will likely result in an unknown command for GDB (as the inferior is not running GDB will not forward it and will consume the input as if it were a command). We thus buffer this input, when the inferior will be resumed we will write the buffer content on standard input.

This implementation is not perfect and a carefully chosen input sequence and the inferior program can produce invalid behavior. Unfortunately, the user has to use this feature carefully. If an input is sent while the inferior is running there will not be any problem with standard input.

Conclusion

We are quite proud of some features that are available in the GDB tracker, namely the ability to have a direct Python representation and asynchronous communication with GDB. Even if from a chronological point of view, we developed most of the GDB tracker for the visualization tools described in Section 4.6, Agdbentures, which was developed later, motivated a lot of changes in the GDB tracker and allowed us to fix many bugs.

The GDB tracker is mostly tested with C code. However, as we claim that most tracker features are portable to many GDB-supported languages, we tested some parts of the tracker with Rust code. Even if the tests were not exhaustive and only with a single other language, it was pleasing to see the tools developed for C work right away for some simple Rust code.

4.4 Python Tracker Implementation

Unlike the GDB tracker, the Python tracker runs in the same process as the inferior (see Figure 4.21), significantly simplifying the tracker's inspection part. The Python standard library offers quite an elaborated and easy-to-use memory inspection interface through the `inspect` module (including frames and global variables inspections). Here, besides implementing the class hierarchy described in Subsection 4.2.2, the `Variable` instances can directly encapsulate the Python object they represent.

4.4.1 Monitoring Python programs with `sys.settrace`

The hard job of the Python tracker is then to control the execution of the inferior. Python comes with a basic extendable debugger called `bdb`. However, `bdb` does not support watchpoints, and the function tracking features are not straightforward to implement in this debugger. Consequently, we decided to implement the Python tracker directly on top of the `sys.settrace` functionality that `bdb` internally uses. `sys.settrace` is a feature provided directly by the Python interpreter. This function allows the registration of a trace function called by the interpreter, among other things, just before executing a line of Python source code. It is possible to trace each byte code operation made by the interpreter but the *Easytracker* interface needs no detail below line granularity. The trace function has three parameters set by the interpreter that we can access:

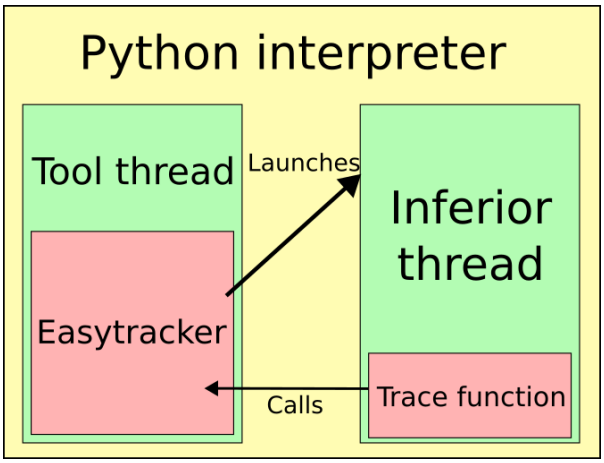


Figure 4.21 – The Python tracker and the inferior both run in the same interpreter in two different threads (see Subsection 4.4.2).

— The current frame described in the `inspect` module.

— The type of the line to be executed (statement, function enter, function return).

— Possibly, the return value if the line type is a function return.

The Python interpreter takes care that the trace function does not apply to itself and only traces each line outside the trace function. Most of the Python tracker work will happen inside this trace function.

To launch the inferior program *Easytracker* must be given a module name. That module must contain a function called `main` (apart from this it can be any Python module). *Easytracker* dynamically loads the main function of the given module and runs it. The trace function is set up after the dynamic load and before the main function is run. So the tracing starts by entering the main function.

4.4.2 Synchronising the Python tracker and user tool

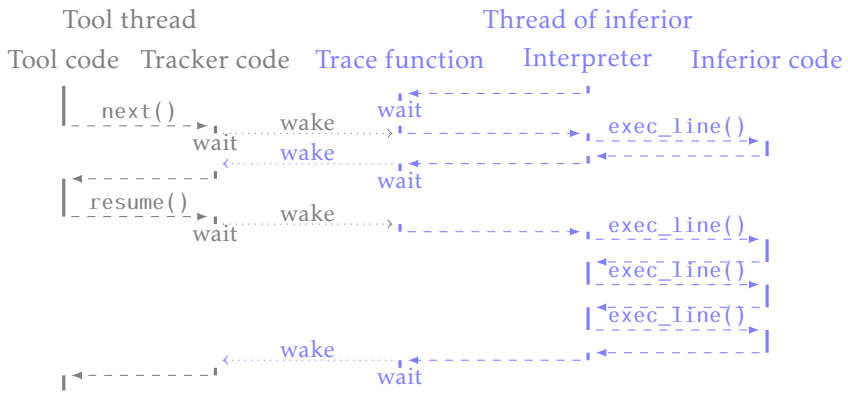


Figure 4.22 – The Python tracker runs the inferior in a thread

Any Python debugger based on `sys.settrace` runs in the same process as the inferior it controls. In our case, this process is the Python interpreter running the tool. As *Easytracker* imposes a call to a function from the control interface should not return until the execution of the inferior pauses, the execution of the inferior must be decoupled from the execution of the tool's code.

We decided to go for a thread-based implementation as shown on Figure 4.22 to implement this decoupling. We call the *tool thread* the main thread of the Python interpreter executing the tool's code. *Easytracker* executes the inferior in a dedicated thread so it can be paused. To have a better understanding of the implementation, Figure 4.22 shows which thread executes which piece of code. The tool thread executes either the code of the tool or the code of the tracker. The inferior thread executes the inferior code as the name suggests. This thread also executes the trace function code registered with `sys.settrace` between the execution of every single line of code of the inferior. This thread-based implementation also allows us to register a trace function only for the monitored program thread and not the main thread running the tool, so only the monitored program is traced. This allows synchronization to be done without being intrusive with the inferior. Thus, *Easytracker* performs the control of the inferior from inside the code of the trace function. Also, it is clear from the diagram that the tool thread waits for the inferior to pause again after calling a control function.

This mechanism is really important to ensure *Easytracker* specification and yet is quite hard to understand. In addition to the threads diagram, we will see a toy program pseudo-code to follow synchronization between the tracker and the inferior on Figure 4.23.

In our case, the thread synchronization using native `threading.Event` is really close to stop/resume being `wait/set`. The only difference is that the flag has to be manually cleared. With the current toy implementation on Figure 4.23, the trace function is called after each line and a call to `step()` has to be done to progress one line by one.

This thread implementation is sufficient in terms of synchronization and, compared to the GDB implementation, the inferior runs in a thread and not a process. Both the tool and the inferior run inside the same Python interpreter. As already stated, this makes introspecting the value of the inferior variables straightforward as both the tracker and the tool can directly access these values. We still have to build the generic model from the memory.

4.4.3 Implementing the control interface

Figure 4.23 shows a toy implementation of a `step` function from the interface. However, it is sufficient to explain how all functions from the control interface of *Easytracker* are implemented.

We define a command enum for each possible function from the control interface. They can be matched in the trace function and the according conditions can be checked.

On Figure 4.24 we can see a pseudo-code implementation for the `next()` function. The `next()` function is exactly like a `step` but if a function is entered it must not stop. So we compare the current frame depth with the target frame depth to stop in line 22.

breakpoints `next` and `step` functions are quite straightforward to implement (we can just add a counter inside the condition if we want to accept the argument as specified in *Easytracker* interface). However, the `resume` function has several stopping conditions. We maintain a list of breakpoints on lines and functions. At each new line or entering a function, we

```python
import threading
import sys
# We use native Event objects for synchronization
synchronization_event = threading.Event()

# the inferior code
def main():
    print(1)
    print(2)
    print(3)

# simplified function to set up the inferior thread
def load_main():
    # sets up the trace function
    sys.set_trace(trace_function)
    # run the main thread
    Thread(target=main).start()

# a simple step function that just resumes execution
def step():
    synchronization_event.set()

# toy trace function
def trace_function(frame, type, return_value):
    # the trace function waits for a command to continue execution
    # as the trace has not returned yet, the inferior thread does not continue
    synchronization_event.wait()

    # Do tracker stuff
    pass

    # in this toy example we only support stepping into the program
    # so we stop execution after each trace function call
    synchronization_event.clear()


if __name__ == "__main__":
    # this would be the tool code that uses the Python tracker instead of toy functions
    # the real tracker functions are encapsulated into a tracker class.
    load_main()
    step()
    step()
    step()
```

Figure 4.23 – A toy example to understand the synchronization between *Easytracker* and the inferior.

check in the breakpoint list if the inferior must stop. Consequently, even when the inferior is resumed through a call to `resume`, single-stepping line by line is done to determine whether *Easytracker* should pause the inferior. Note that this slows the execution down a lot. However, it is not critical for the pedagogical context we target.

```
class PythonTracker:
    def __init__(self):
        ...
        self.command = None
        self.synchronization_event = threading.Event()

    def next(self):
        self.command = COMMAND.NEXT
        # the current frame depth is a state maintained by the tracker
        self.depth_to_stop = self.current_frame_depth

        # we resume the thread execution
        self.synchronization_event.set()

    # the tracker trace function is set up just before launching the inferior
    def trace_function(self):
        # Each call to the trace function maintain some states
        # The current frame depth is one of them
        self.maintain_state()

        if self.command == COMMAND.NEXT:
            if self.current_frame_depth == self.depth_to_stop:
                # if the condition to stop after a next command is met, we stop the thread
                synchronization_event.clear()
                synchronization_event.wait()

        ...

    ...
```

Figure 4.24 – `next()` pseudo-code for the Python tracker

watchpoints Implementing watchpoints is a little more work than breakpoints because values of watched variables need to be maintained and compared. Watched variables in scope are evaluated using the `inspect` module. Two decisions need to be made:

— To know if a variable value has changed, we used the built-in == Python operator. Users can override this but there is no default comparison if this is not implemented for user types apart from identity comparison.

— The same names in different scopes can be watched. We kept a simple stack-based implementation when pushing/popping contexts at each function enter/exit. As Python does not support block scopes and only function scopes, it should follow Python specifications.

function tracking Implementing function tracking in the Python tracker is more accessible than in the one we developed for GDB-supported languages. Indeed, the interpreter calls our trace function after entering a new function and before exiting a function. This makes implementing function tracking as easy as implementing regular breakpoints. We just have to maintain a list of tracked functions.

Conclusion

Python being the language we use to write *Easytracker*, the Python tracker has native access to the inferior memory. However, being executed in the same interpreter as the inferior requires some synchronization. But this was quite elegantly addressed as we saw in Subsection 4.4.2.

The Python tracker is currently used as a base for some of the tools presented in Section 4.6. Future work regarding the Python tracker would be to add unit tests so we will not discover bugs later with new use cases.

4.5 Future backend projects

The GDB and Python backends are reasonably usable and already span many programming languages used in computer science teaching. However, some major languages like Java are not yet supported by *Easytracker*. In this short section, we will discuss some potential implementations of new *Easytracker* backends.

4.5.1 GDB Reverse Debugging

In this thesis, we never focused on reverse debugging. However, this is a debugging practice that progressively gains some more powerful tools and is more and more used industrially. RR[100] (standing for Record and Replay) is a reverse debugging tool based on GDB.

It adds a "reverse" version of many GDB regular commands like `rstep` or `rcontinue`. A powerful debugging feature of reverse debuggers is the combination of data breakpoints and reverse continue. We do not have to stop every time a variable we are interested in is modified. We can stop the program whenever we want and use a reverse continue to go where the variable was modified for the last time. Having an *Easytracker* extension to support these features could allow *Easytracker* to be used in a reverse debugging context.

Some experimental work has been done to support an RR backend but *Easytracker* software engineering is not yet clean enough so that this implementation is done without too much effort. This option will be considered seriously when the GDB tracker is more mature.

4.5.2 Ocaml

Ocaml is often used as an introduction to functional programming. Due to the difference between functional and imperative programming, it could be a nice argument for the robustness of *Easytracker* interface to be usable with functional programming.

To our knowledge, the Ocaml debugger *ocamldebug* does not support extension and scripting as GDB does. This would require developing all features such as sending commands to *ocamldebug* and parsing the debugger output. We would probably do this by first developing an interface library playing the role of *pygdbmi* and abstracting the debugger output even though there is no machine interface.

We also have an issue regarding the functional programming execution model. The Ocaml debugger does not really have a notion of line numbers. Breakpoints are placed on *events* that describe variable change instead of raw line numbers. Single-stepping is also done from event to event. However, events are relatively small-grained execution units, like entering functions or evaluating loop conditions, for example. This is close to bytecode operations in the Python

interpreter so we should not miss any information. The main problem would be to *refer* to specific source lines as the *Easytracker* interface does.

As a remark regarding the last subsection, *ocamldebug* natively supports reverse debugging through the `reverse` command that goes back to the last breakpoint.

4.5.3 Java

Java is a widely used programming language in the industry, and it is also often used in computer science education to teach object-oriented programming. Even if it is object-oriented, we should be able to adapt our memory model to Java as we could do it with Python.

The main issue, like the two current trackers, is how to technically monitor the program execution. We could use the Java debugger as we would do for Ocaml but Java is a more widespread language and there should be some tools to instrument bytecode and monitor execution. Some tools are emerging from the monitoring and validation community like BISM[101] that allows writing high-level expressions on triggers like function entering/exiting or value modification.

4.6 Visualization tools based on *Easytracker*

Early works on *Easytracker* were motivated by the need for Agdbentures to monitor a program. However, *Easytracker* gained its generic interface when we realized that program monitoring could be used extensively in computer science teaching. To demonstrate what can be achieved with *Easytracker*, we describe in this section our own tools we built on top of it. These tools are actually used in lab sessions or lectures by the teachers who contributed to this work.

4.6.1 Python/C Stack and StackHeap Diagrams

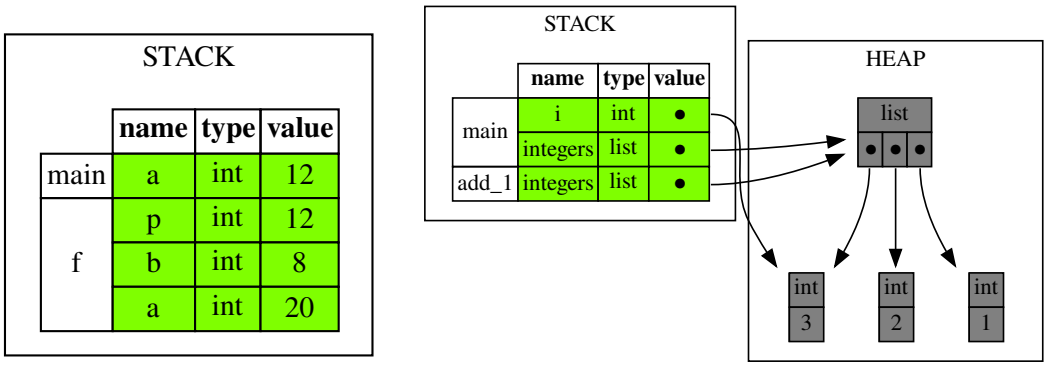


Figure 4.25 – Stack diagram (left) and StackHeap diagram (right) for two different Python programs

We used *Easytracker* to automatically generate stack [99] and stack-and-heap [98] diagrams that we use in our Python programming and C programming course materials. Teachers from

these two classes decided together on a common representation, and thanks to *Easytracker*, they quickly implemented a tool to have a common automated representation "for all the seasons" [98].

This tool takes as input a Python or a C program along with display options and generates either a stack diagram or a stack-and-heap diagram after the execution of every line. We can see the result of this tool on a Python program passing a list as a parameter to a function called `add_1` in Figure 4.25. The left part of the figure is the stack diagram we show at the beginning of the course. We have not yet introduced references when we use such diagrams, as we focus on making students understand stack frames and variable scopes.

Later in the Python course, we introduce the concept of references and emphasize that every variable is a reference in Python. The stack diagram is then augmented with the heap to picture stack-and-heap diagrams, as shown in the right part of Figure 4.25. The tool lets the user decide whether or not to draw instances of primitive types. This example shows instances of primitive types such as `int` as separated objects. This representation is critical in the context of our class to help students understand the notion of reference. Nevertheless, as soon as students assimilate this, inlining primitive types can drastically simplify diagrams. From this example, it is clear that a generic tool such as Python Tutor cannot meet the specific needs of our course.

Figure 4.25 shows the result of the stack-and-heap tool on a C program. We represent invalid pointers with a cross. We used this example to show students that compared to Python, the value of a variable can be in the stack in C, and we can have pointers targeting the stack. This example of a stack-and-heap diagram does not show the values of the pointers (i.e. memory addresses), but a display option can change this.

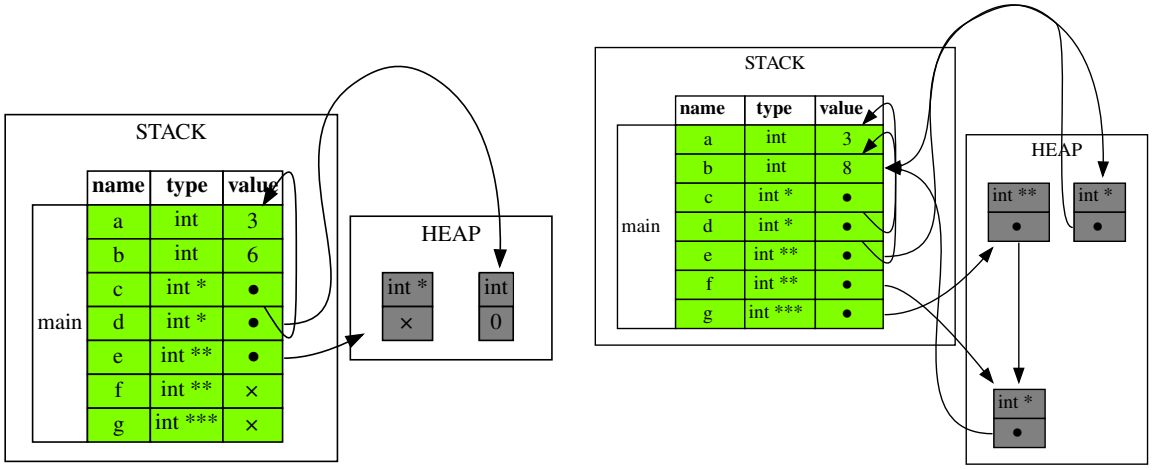


Figure 4.26 – Stack/heap diagram for a C program, this is the same program but the left image is an intermediate representation of the memory while the different pointers are still being assigned.

As the last comment on this tool, as soon as we have a new implementation of *Easytracker* for another programming language, we will have for free a stack and a stack and heap visualization tool for that language. Indeed, as shown in Subsection 4.2.3, the tool's code is language-agnostic except for the line initializing the tracker.

4.6.2 RISC-V Registers and Memory Viewer

We also used *Easytracker* to develop a visualization tool in the context of an assembly programming class. This tool aims to visualize the CPU register and the memory represented as it is, hence a one-dimensional array of values.

Figure 4.27 shows how the tool looks. Again, thanks to the *Easytracker*'s expressiveness and capability to handle programs written in any GDB-supported language, it was easy to execute the program line by line and get register and memory values at each step. This needed some specific GDB commands sent by the tool but only to get register values and *Easytracker* already abstracts all the GDB communication to send commands. In this case, the visualization is implemented directly by the tool using the `dot` framework and a splittable terminal.

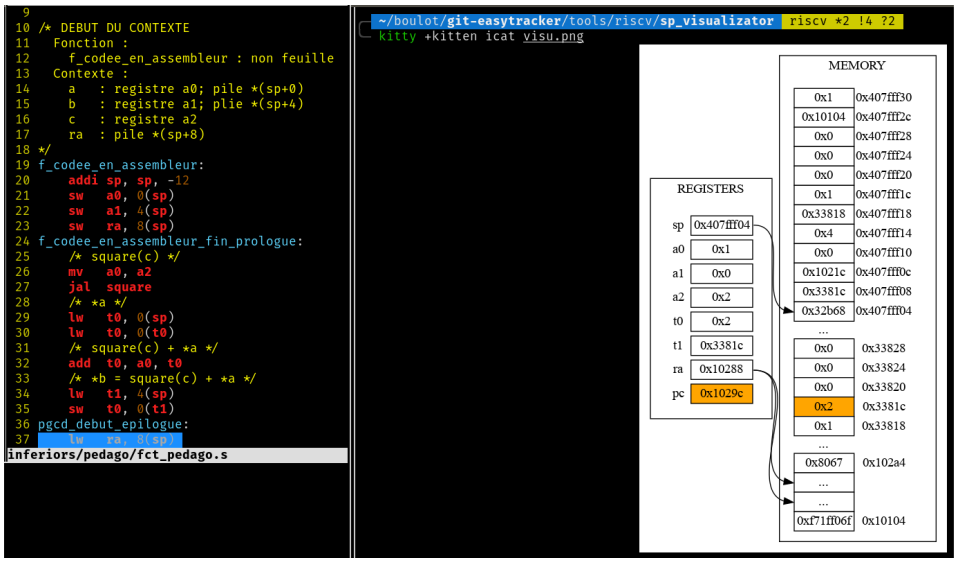


Figure 4.27 – RISC-V registers and memory viewer

4.6.3 Recursive Calls Visualization

To help our students grasp the control flow of the execution of a recursive function, we quickly implemented a dedicated visualization tool using *Easytracker*. This tool takes as inputs a Python program and the name of the recursive function we want to visualize.

Figure 4.28 shows an example of the output of this tool. We can see a new node appearing in the tree at each recursive call. Red nodes are alive calls. When a function exits, the tool changes the corresponding node color to gray. In the meantime, the return value is added to a back edge of the tree. This example focuses on understanding recursive calls; hence, each node displays the content of the array at the time of the call even if it is a shared reference which content changes during the execution.

Thanks to *Easytracker* control interface, it was straightforward to track the entrance/exit of the recursive function and then get the value of parameters/return value with the inspection interface. As for the RISC-V visualization tool, the visualization is implemented directly by the tool and again using `graphviz` and a splittable terminal.

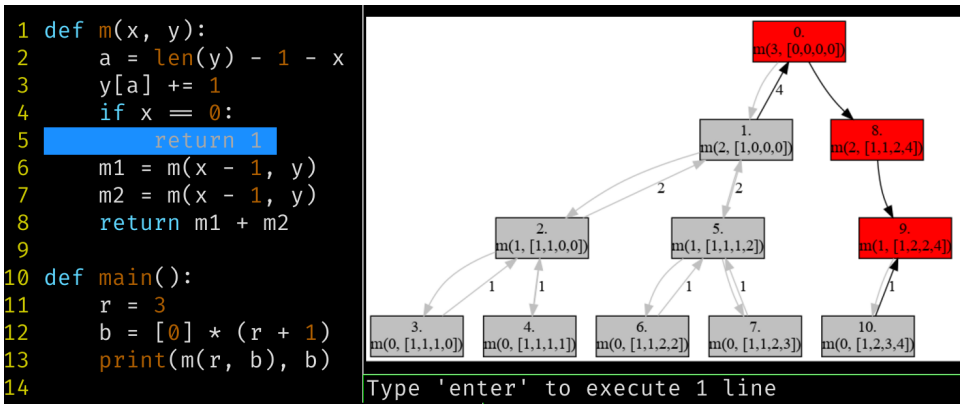


Figure 4.28 – Recursive call tree

4.7 Discussion

In this section, we deliberately focus on the possibilities that *Easytracker* offers to teachers, whose schedules are often very busy. From our point of view, providing a library for writing visualization tools more quickly should allow teachers to efficiently design custom tools that are easy to evaluate in their courses.

We were able to see how much time *Easytracker* saved us when it came to implementing the visualization tool for the RISC-V processor architecture course, as *Easytracker* offers teachers to really focus on what they want to show, taking care of all the technical details of how to control the program and how to inspect its state.

The tools presented in Section 4.6 were all written using the *Easytracker* interfaces. This is a way to show that our library works and to illustrate how these interfaces allow writing various tools.

While we use some of them in support of our courses, the objective here is not to highlight their pedagogical contribution, which could be the subject of a thorough evaluation on cohorts of students, but to how easy writing a tool can be. We have paid particular attention to documenting the code of these tools so that they can serve as a starting point for teachers who wish to write their own.

Easytracker makes it easy to write customized visualization tools, which could then be distributed to students. Before developing *Easytracker* and the stack and heap visualization tool, we provided a visualization module to the student of our introductory Python programming course. Students had to instrument their code with calls to functions in this visualization module to generate visual representations and debug their programs. We found that most students using the tool were those who were already comfortable with programming. The students who had difficulty did not use it very much. We believe this indicates the need for a turnkey tool such as the stack and heap viewer presented in this work.

We used the stack and heap visualization tool to introduce the notion of reference in our Python programming lecture. This introduction is done with the following three lines of Python: In these lectures, students asked many questions like “what happens if we add an element to another sub list?” or “what happens if we delete a sub list?" Thanks to the stack and heap visualization tool, the teacher can answer, “let’s see!" Having a direct visualization of students’ questions is a nice addition to lecture interactivity between students and the

```
l = [[]] * 4
l[0].append(7)
print(l)
```

Figure 4.29 – References and a dangerous list comprehension

teacher because it is easier to formulate questions and students can quickly see if they observe a behavior they were not expecting.

Future work on visualization building blocks *Easytracker* offers several abstractions to write all these tools, the monitoring code is straightforward and we only have to write visualization code. However, writing this visualization requires some time and expertise. We do not have found interesting visualization building blocks yet but this may be the topic of future research.

Conclusion on visualization tools

We showed that developing new tools is accessible. However, most of the tools were developed by teachers already familiar with *Easytracker* (one of them was not a contributor so could be considered as an external teacher). A true test would be for *Easytracker* to be used by teachers that cannot communicate with us directly and would need to use traditional communication methods like git issues or email. We believe we should keep working on having good software engineering practices so external teachers could get started with *Easytracker* easily.

Pedagogically validating the tools would be the next objective, but many computer educators are already familiar with this kind of research. Once we are satisfied with *Easytracker* state we will focus more on developing and polishing the tools so we can evaluate them.

Conclusion

In this chapter, we described two implementations of the *Easytracker* interface, in Python and using GDB. Both implementations are already used in other projects, especially the GDB tracker with Agdbentures. However, we want *Easytracker* to be a public project and that external teachers to use it. Even if this is our main goal, there are still a few problems to solve before making the code public. The main one is to completely stick to good software engineering practices and extensively test the implementations. We still regularly do some bug-fixing after discovering bugs with Agdbentures or developing a new tool. *Easytracker* has a research article dedicated to it which is a condensed version of this chapter, we hope it will help promote *Easytracker* so many teachers can use it after it is publicly available. We already presented *Easytracker* in small seminars at our laboratory scale. Some teachers gained interest in *Easytracker* but are not yet using it in their course.

Conclusion

This is the end of this thesis.

We saw that a completely automatic system with minimal intervention from a teacher is too hard to develop if it must perform both generation and recommendation of debugging exercises. The required work in this case is vastly consisting of engineering work and requires great expertise in statistical methods and machine learning. Moreover, even this statistical expertise may not be sufficient because usual statistical methods are not directly applicable to the learning environment due to the dynamic nature of the learning process. And, even if we manage to obtain an accurate statistical system, it is not easy to obtain an efficient system that improves student's learning. This makes this statistical system not the default way to go if we want to improve students' learning. However, once we have a working system we can add simpler statistical methods to improve certain features of the system like an optional recommendation or a statistical difficulty rating.

By using a teacher's expertise to carefully design debugging exercises, we can tailor them to be compatible with the addition of ludic aspects to the pedagogical sequence. The pedagogical sequence can be used without the need for complex recommendation and generation systems. This became Agdbentures. All these additions also require some notable engineering work. However, developing such systems requires generic programming expertise and no specific expertise like machine learning anymore. If we add heavy video game elements, we have to be aware that this is a specific expertise and not generic computer science expertise. We do not know yet if a pluridisciplinary team is needed for Agdbentures development.

From an engineering point of view, we proposed *Easytracker* to abstract and facilitate part of this engineering work that should actually be needed for most computer science teaching applications even outside debugging. *Easytracker* suggests that some engineering work can be shared between computer science instructors. As with any software library, this needs that a user community is actively using it and developers are maintaining it. The tools made with *Easytracker* can also be shared among the teacher's community. We have to actively think about this and communicate about *Easytracker* and its uses if we want such a community to exist and for *Easytracker* to reach its full potential.

An interesting question that is not addressed in this thesis is the fact that we unconsciously targeted computer science majors' students for our debugging exercises (like taking the C programming language as an application language). This is natural because it is easier to design some exercises for our students. Debugging is equally needed for people not majoring in computer science and using computer science as a tool. We do not know what differences we need to apply to Agdbentures so we could have such a system for everyone doing computer science. We at least feel the benefits of using *Easytracker* for the development of Agdbentures because we would surely need to change the language to the Python programming language

(or maybe a toy language dedicated to Agdbentures but we should be able to write simple Python programs if needed). This change may still require a bit of engineering work apart from translating some existing levels to Python but most of the abstractions are already done in *Easytracker*. It will be interesting to study if the debugging skills and common difficulties are the same between people majoring and not majoring in computer science as we usually refer to everyone as a single "novice" category.

Bibliography

Intelligent Tutoring Systems

[1] Jean-François Nicaud, Denis Bouhineau, and Thomas Huguet. « The Aplusix-Editor: A new kind of software for the learning of algebra ». In: *Intelligent Tutoring Systems: 6th International Conference, ITS 2002 Biarritz, France and San Sebastian, Spain, June 2–7, 2002 Proceedings 6*. Springer. 2002, pp. 178–187.

[2] Kenneth R Koedinger, John R Anderson, William H Hadley, Mary A Mark, et al. « Intelligent tutoring goes to school in the big city ». In: *International Journal of Artificial Intelligence in Education* 8.1 (1997), pp. 30–43.

[3] Albert T Corbett, Kenneth R Koedinger, and John R Anderson. « Intelligent tutoring systems ». In: *Handbook of human-computer interaction*. Elsevier, 1997, pp. 849–874.

[4] Mahmoud J Abu Ghali, Abdullah Abu Ayyad, Samy S Abu-Naser, and Mousa Abu Laban. « An intelligent tutoring system for teaching English grammar ». In: (2018).

[5] Antonija Mitrovic, Brent Martin, and Pramuditha Suraweera. « Intelligent tutors for all: Constraint-based modeling methodology, systems and authoring ». In: (2007).

Collaborative Filtering

[6] Jesús Bobadilla, Fernando Ortega, Antonio Hernando, and Abraham Gutiérrez. « Recommender systems survey ». In: *Knowledge-based systems* 46 (2013), pp. 109–132.

[7] Robert M Bell and Yehuda Koren. « Lessons from the netflix prize challenge ». In: *Acm Sigkdd Explorations Newsletter* 9.2 (2007), pp. 75–79.

[8] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. « Gradient-based learning applied to document recognition ». In: *Proceedings of the IEEE* 86.11 (1998), p. 2290.

[9] Greg Linden, Brent Smith, and Jeremy York. « Amazon. com recommendations: Item-to-item collaborative filtering ». In: *IEEE Internet computing* 7.1 (2003), pp. 76–80.

[10] J. Stamper, A. Niculescu-Mizil, S. Ritter, G.J. Gordon, and K.R Koedinger. *Algebra I 2006-2007. Development data set from KDD Cup 2010 Educational Data Mining Challenge. Find it at* `http://pslcdatashop.web.cmu.edu/KDDCup/downloads.jsp`. 2010.

[11] J. Stamper, A. Niculescu-Mizil, S. Ritter, G.J. Gordon, and K.R Koedinger. *Bridge to Algebra 2006-2007. Development data set from KDD Cup 2010 Educational Data Mining Challenge. Find it at* `http://pslcdatashop.web.cmu.edu/KDDCup/downloads.jsp`. 2010.

Elo rating and derivatives

[12] Arpad E Elo. *The rating of chessplayers, past and present*. Arco Pub., 1978.

[13] Mark E Glickman. « The glicko system ». In: *Boston University* 16 (1995), pp. 16–17.

[14] Rémi Coulom. « Whole-history rating: A Bayesian rating system for players of time-varying strength ». In: *International Conference on Computers and Games*. Springer. 2008, pp. 113–124.

[15] Radek Pelánek. « Applications of the Elo rating system in adaptive educational systems ». In: *Computers & Education* 98 (2016), pp. 169–179.

Matrix Factorization

[16] Mingyu Feng, Neil Heffernan, and Kenneth Koedinger. « Addressing the Assessment Challenge with an Online System That Tutors as it Assesses ». In: *User Model. User-Adapt. Interact.* 19 (Aug. 2009), pp. 243–266. DOI: 10.1007/s11257-009-9063-7.

[17] Yehuda Koren, Robert Bell, and Chris Volinsky. « Matrix factorization techniques for recommender systems ». In: *Computer* 42.8 (2009), pp. 30–37.

[18] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit S. Dhillon. « Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems ». In: *IEEE International Conference of Data Mining*. 2012.

[19] Cho-Jui Hsieh and Inderjit S Dhillon. « Fast coordinate descent methods with variable selection for non-negative matrix factorization ». In: *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2011, pp. 1064–1072.

[20] Ke Zhou, Shuang-Hong Yang, and Hongyuan Zha. « Functional matrix factorizations for cold-start recommendation ». In: *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. 2011, pp. 315–324.

[21] Uroš Ocepek, Jože Rugelj, and Zoran Bosnić. « Improving matrix factorization recommendations for examples in cold start ». In: *Expert Systems with Applications* 42.19 (2015), pp. 6784–6794. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2015.04.071. URL: https://www.sciencedirect.com/science/article/pii/S0957417415003139.

[22] X. He, Hanwang Zhang, Min-Yen Kan, and Tat-Seng Chua. « Fast Matrix Factorization for Online Recommendation with Implicit Feedback ». In: *ArXiv* abs/1708.05024 (2016).

[23] Xin Luo, Yunni Xia, and Qingsheng Zhu. « Incremental Collaborative Filtering recommender based on Regularized Matrix Factorization ». In: *Knowledge-Based Systems* 27 (2012), pp. 271–280.

[24] Steffen Rendle and Lars Schmidt-Thieme. « Online updating regularized kernel matrix factorization models for large-scale recommender systems ». In: Jan. 2008. DOI: 10.1145/1454008.1454047.

Knowledge Tracing

[25] Albert T Corbett and John R Anderson. « Knowledge tracing: Modeling the acquisition of procedural knowledge ». In: *User modeling and user-adapted interaction* 4.4 (1994), pp. 253–278.

[26] Xiaolu Xiong, Siyuan Zhao, Eric G Van Inwegen, and Joseph E Beck. « Going deeper with deep knowledge tracing. » In: *International Educational Data Mining Society* (2016).

[27] Nguyen Thai-Nghe, Lucas Drumond, Tomáš Horváth, Lars Schmidt-Thieme, et al. « Multi-relational factorization models for predicting student performance ». In: *KDD Workshop on Knowledge Discovery in Educational Data (KDDinED)*. Citeseer. 2011, pp. 27–40.

[28] Jill-Jênn Vie and Hisashi Kashima. « Knowledge tracing machines: Factorization machines for knowledge tracing ». In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 750–757.

[29] Radek Pelánek. « Bayesian knowledge tracing, logistic models, and beyond: an overview of learner modeling techniques ». In: *User Modeling and User-Adapted Interaction* 27 (2017), pp. 313–350.

[30] Mohammad Khajah, Robert V Lindsey, and Michael C Mozer. « How deep is knowledge tracing? » In: *arXiv preprint arXiv:1604.02416* (2016).

[31] Chris Piech, Jonathan Spencer, Jonathan Huang, Surya Ganguli, Mehran Sahami, Leonidas Guibas, and Jascha Sohl-Dickstein. « Deep knowledge tracing ». In: *arXiv preprint arXiv:1506.05908* (2015).

[32] Sein Minn, Yi Yu, Michel Desmarais, Feida Zhu, and Jill Vie. *Deep Knowledge Tracing and Dynamic Student Classification for Knowledge Tracing*. Sept. 2018.

[33] J. Zhang, Xingjian Shi, Irwin King, and D. Yeung. « Dynamic Key-Value Memory Networks for Knowledge Tracing ». In: *Proceedings of the 26th International Conference on World Wide Web* (2017).

[34] Sein Minn, Yi Yu, Michel C Desmarais, Feida Zhu, and Jill-Jênn Vie. « Deep knowledge tracing and dynamic student classification for knowledge tracing ». In: *2018 IEEE International conference on data mining (ICDM)*. IEEE. 2018, pp. 1182–1187.

[35] Liang Zhang, Xiaolu Xiong, Siyuan Zhao, Anthony Botelho, and Neil T Heffernan. « Incorporating rich features into deep knowledge tracing ». In: *Proceedings of the fourth (2017) ACM conference on learning@ scale*. 2017, pp. 169–172.

[36] Liangbei Xu and Mark A. Davenport. « Dynamic Knowledge Embedding and Tracing ». In: *Proceedings of the 13th International Conference on Educational Data Mining, EDM 2020, Fully virtual conference, July 10-13, 2020*. Ed. by Anna N. Rafferty, Jacob Whitehill, Cristóbal Romero, and Violetta Cavalli-Sforza. International Educational Data Mining Society, 2020.

[37] Thomas Sergent, François Bouchet, and Thibault Carron. « Towards Temporality-Sensitive Recurrent Neural Networks through Enriched Traces ». In: *Proceedings of the 13th International Conference on Educational Data Mining, EDM 2020, Fully virtual conference, July 10-13, 2020*. Ed. by Anna N. Rafferty, Jacob Whitehill, Cristóbal Romero, and Violetta Cavalli-Sforza. International Educational Data Mining Society, 2020.

[38] Chenyang Wang, Weizhi Ma, Min Zhang, Chuancheng Lv, Fengyuan Wan, Huijie Lin, Taoran Tang, Yiqun Liu, and Shaoping Ma. « Temporal cross-effects in knowledge tracing ». In: *Proceedings of the 14th ACM International Conference on Web Search and Data Mining*. 2021, pp. 517–525.

Neural Network Recommender Systems

[39] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. « Neural collaborative filtering ». In: *Proceedings of the 26th international conference on world wide web*. 2017, pp. 173–182.

[40] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. « DeepFM: a factorization-machine based neural network for CTR prediction ». In: *arXiv preprint arXiv:1703.04247* (2017).

[41] Maurizio Ferrari Dacrema, Paolo Cremonesi, and Dietmar Jannach. « Are we really making much progress? A worrying analysis of recent neural recommendation approaches ». In: *Proceedings of the 13th ACM conference on recommender systems*. 2019, pp. 101–109.

[42] Joeran Beel, Corinna Breitinger, Stefan Langer, Andreas Lommatzsch, and Bela Gipp. « Towards reproducibility in recommender-systems research ». In: *User modeling and user-adapted interaction* 26.1 (2016), pp. 69–101.

[43] Michael D Ekstrand, Michael Ludwig, Joseph A Konstan, and John T Riedl. « Rethinking the recommender research ecosystem: reproducibility, openness, and lenskit ». In: *Proceedings of the fifth ACM conference on Recommender systems*. 2011, pp. 133–140.

[44] Vito Walter Anelli, Alejandro Bellogin, Antonio Ferrara, Daniele Malitesta, Felice Antonio Merra, Claudio Pomo, Francesco Maria Donini, and Tommaso Di Noia. « Elliot: a comprehensive and rigorous framework for reproducible recommender systems evaluation ». In: *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2021, pp. 2405–2414.

[45] Shubhendu Trivedi, Zachary A Pardos, Gabor N Sarkozy, and Neil T Heffernan. « Co-Clustering by Bipartite Spectral Graph Partitioning for Out-of-Tutor Prediction. » In: *International Educational Data Mining Society* (2012).

[46] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. « Learning to forget: Continual prediction with LSTM ». In: *Neural computation* 12.10 (2000), pp. 2451–2471.

[47] Hugo Zaragoza. « Confidence Measures for Neural Network Classifiers ». en. In: ().

[48] Terrance DeVries and Graham W. Taylor. « Learning Confidence for Out-of-Distribution Detection in Neural Networks ». en. In: arXiv:1802.04865 (Feb. 2018). arXiv:1802.04865 [cs, stat]. URL: http://arxiv.org/abs/1802.04865.

Random program generation

[49] Shikhar Bharadwaj and Shirish Shevade. « Explainable natural language to bash translation using abstract syntax tree ». In: *Proceedings of the 25th Conference on Computational Natural Language Learning*. 2021, pp. 258–267.

[50] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. « Random testing for C and C++ compilers with YARPGen ». In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–25.

[51] Eyal Bin, Roy Emek, Gil Shurek, and Avi Ziv. « Using a constraint satisfaction formulation and solution techniques for random test program generation ». In: *IBM Systems Journal* 41.3 (2002), pp. 386–402.

[52] Gergö Barany. « Liveness-driven random program generation ». In: *arXiv preprint arXiv:1709.04421* (2017).

[53] Louis-Noel Pouchet, Cedric Bastoul, Uday Bondhugula, and Sven Verdoolaege. *The polyhedral compiler collection*. https://web.cs.ucla.edu/~pouchet/software/pocc/. 2009–2013.

Neural Networks for program generation

[54] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. « Synthesizing benchmarks for predictive modeling ». In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2017, pp. 86–99.

[55] David Lynch, James McDermott, and Michael O'Neill. « Program synthesis in a continuous space using grammars and variational autoencoders ». In: *Parallel Problem Solving from Nature–PPSN XVI: 16th International Conference, PPSN 2020, Leiden, The Netherlands, September 5-9, 2020, Proceedings, Part II 16*. Springer. 2020, pp. 33–47.

[56] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. « A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures ». In: *Neural Computation* 31.7 (July 2019), pp. 1235–1270. ISSN: 0899-7667. DOI: 10.1162/neco_a_01199. eprint: https://direct.mit.edu/neco/article-pdf/31/7/1235/1053200/neco_a_01199.pdf. URL: https://doi.org/10.1162/neco%5C_a%5C_01199.

Program mutations, testing and bug-fixing

[57] Heling Cao, YangXia Meng, Jianshu Shi, Lei Li, Tiaoli Liao, and Chenyang Zhao. « A survey on automatic bug fixing ». In: *2020 6th International Symposium on System and Software Reliability (ISSSR)*. IEEE. 2020, pp. 122–131.

[58] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. « Sequencer: Sequence-to-sequence learning for end-to-end program repair ». In: *IEEE Transactions on Software Engineering* 47.9 (2019), pp. 1943–1959.

[59] Geunseok Yang, Kyeongsic Min, and Byungjeong Lee. « Applying deep learning algorithm to automatic bug localization and repair ». In: *Proceedings of the 35th Annual ACM symposium on applied computing*. 2020, pp. 1634–1641.

[60] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. « Automatically finding patches using genetic programming ». In: *2009 IEEE 31st International Conference on Software Engineering*. IEEE. 2009, pp. 364–374.

[61] Kazuhiro Nakamura and Nagisa Ishiura. « Random testing of C compilers based on test program generation by equivalence transformation ». In: *2016 IEEE Asia pacific conference on circuits and systems (APCCAS)*. IEEE. 2016, pp. 676–679.

Program Equivalence

[62] Shubhani Gupta, Aseem Saxena, Anmol Mahajan, and Sorav Bansal. « Effective use of SMT solvers for program equivalence checking through invariant-sketching and query-decomposition ». In: *Theory and Applications of Satisfiability Testing–SAT 2018: 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9–12, 2018, Proceedings 21*. Springer. 2018, pp. 365–382.

[63] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. « Semantic program alignment for equivalence checking ». In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 1027–1040.

[64] Steve Kommrusch, Théo Barollet, and Louis-Noël Pouchet. « Equivalence of dataflow graphs via rewrite rules using a graph-to-sequence neural model ». In: *arXiv preprint arXiv:2002.06799* (2020).

Ludic aspects and games for teaching

[65] Marcus Leaning. « A study of the use of games and gamification to enhance student engagement, experience and achievement on a theory-based course of an undergraduate media degree ». In: *Journal of Media Practice* 16.2 (2015), pp. 155–170.

[66] Lisbett Shirley Paguay Yupanqui. « Ludic activities for strengthening the vocabulary of the english language ». B.S. thesis. Universidad de Guayaquil Facultad de Filosofia, Letras y Ciencias de la ..., 2019.

[67] Sigrid Jordal Havre, Lauri Väkevä, Catharina R Christophersen, and Egil Haugland. « Playing to learn or learning to play? Playing Rocksmith to learn electric guitar and bass in Nordic music teacher education ». In: *British Journal of Music Education* 36.1 (2019), pp. 21–32.

[68] Adilson Vahldick, Antonio José Mendes, and Maria José Marcelino. « A review of games designed to improve introductory computer programming competencies ». In: *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. 2014, pp. 1–7. DOI: 10.1109/FIE.2014.7044114.

[69] Baba is you. https://www.hempuli.com/baba/.

Using game development for Computer Science education

[70] Jessica D Bayliss and Sean Strout. « Games as a" flavor" of CS1 ». In: *Proceedings of the 37th SIGCSE technical symposium on Computer science education*. 2006, pp. 500–504.

[71] Katie Seaborn, Magy Seif El-Nasr, David Milam, and Darren Yung. « Programming, PWNed: Using digital game development to enhance learners' competency and self-efficacy in a high school computing science course ». In: *Proceedings of the 43rd ACM technical symposium on computer science education*. 2012, pp. 93–98.

[72] Mohammed Al-Bow, Debra Austin, Jeffrey Edgington, Rafael Fajardo, Joshua Fishburn, Carlos Lara, Scott Leutenegger, and Susan Meyer. « Using game creation for teaching computer programming to high school students and teachers ». In: *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*. 2009, pp. 104–108.

[73] Chris Johnson et al. « Game Development for Computer Science Education ». In: *Proceedings of the 2016 ITiCSE Working Group Reports*. ITiCSE '16. Arequipa, Peru: Association for Computing Machinery, 2016, pp. 23–44. ISBN: 9781450348829. DOI: 10.1145/3024906.3024908. URL: https://doi.org/10.1145/3024906.3024908.

[74] Chiung-Fang Chiu and Hsing-Yi Huang. « Guided debugging practices of game based programming for novice programmers ». In: *International Journal of Information and Education Technology* 5.5 (2015), p. 343.

Debugging and debug teaching

[75] Leo Gugerty and Gary M Olson. « Comprehension differences in debugging by skilled and novice programmers ». In: *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*. 1986, pp. 13–27.

[76] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. « An analysis of patterns of debugging among novice computer science students ». In: *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*. 2005, pp. 84–88.

[77] V Vipindeep and Pankaj Jalote. « List of common bugs and programming practices to avoid them ». In: *Electronic*, *March* (2005).

[78] Greg C Lee and Jackie C Wu. « Debug It: A debugging practicing system ». In: *Computers & Education* 32.2 (1999), pp. 165–179.

[79] Andrew Luxton-Reilly, Emma McMillan, Elizabeth Stevenson, Ewan Tempero, and Paul Denny. « Ladebug: An online tool to help novice programmers improve their debugging skills ». In: *Proceedings of the 23rd annual acm conference on innovation and technology in computer science education*. 2018, pp. 159–164.

[80] Michael J. Lee. « Gidget: An online debugging game for learning and engagement in computing education ». In: *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2014, pp. 193–194. DOI: 10.1109/VLHCC.2014.6883051.

[81] Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. « Debugging: a review of the literature from an educational perspective ». In: *Computer Science Education* 18.2 (2008), pp. 67–92.

[82] Tilman Michaeli and Ralf Romeike. « Improving debugging skills in the classroom: The effects of teaching a systematic debugging process ». In: *Proceedings of the 14th workshop in primary and secondary computing education*. 2019, pp. 1–7.

[83] Irvin R Katz and John R Anderson. « Debugging: An analysis of bug-location strategies ». In: *Human-Computer Interaction* 3.4 (1987), pp. 351–399.

[84] Chen Li, Emily Chan, Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. « Towards a framework for teaching debugging ». In: *Proceedings of the Twenty-First Australasian Computing Education Conference*. 2019, pp. 79–86.

[85] Ana Selvaraj, Eda Zhang, Leo Porter, and Adalbert Gerald Soosai Raj. « Live coding: A review of the literature ». In: *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. 2021, pp. 164–170.

Methodology

[86] Emma Marsden and Carole J Torgerson. « Single group, pre-and post-test research designs: Some methodological concerns ». In: *Oxford Review of Education* 38.5 (2012), pp. 583–616.

Educational Psychology

[87] David A Nembhard and Napassavong Osothsilp. « An empirical comparison of forgetting models ». In: *IEEE Transactions on Engineering Management* 48.3 (2001), pp. 283–291.

[88] Hans G Furth and Harry Wachs. *Thinking goes to school: Piaget's theory in practice*. Oxford University Press, USA, 1975.

[89] Edward C Tolman. « A cognition motivation model. » In: *Psychological review* 59.5 (1952), p. 389.

Program Visualization and Notional Machines

[90] Philip J. Guo. « Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education ». In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE '13. Denver, Colorado, USA: Association for Computing Machinery, 2013, pp. 579–584. ISBN: 9781450318686. DOI: 10.1145/2445196.2445368. URL: https://doi.org/10.1145/2445196.2445368.

[91] Benedict du Boulay, Tim O'Shea, and John Monk. « The black box inside the glass box: presenting computing concepts to novices ». In: *International Journal of Man-Machine Studies* 14.3 (1981), pp. 237–249. ISSN: 0020-7373. DOI: https://doi.org/10.1016/S0020-7373(81)80056-9. URL: https://www.sciencedirect.com/science/article/pii/S0020737381800569.

[92] Juha Sorva. « Notional Machines and Introductory Programming Education ». In: *ACM Trans. Comput. Educ.* 13.2 (July 2013). DOI: 10.1145/2483710.2483713. URL: https://doi.org/10.1145/2483710.2483713.

[93] Sally Fincher et al. « Notional Machines in Computing Education: The Education of Attention ». In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '20. Trondheim, Norway: Association for Computing Machinery, 2020, pp. 21–50. ISBN: 9781450382939. DOI: 10.1145/3437800.3439202. URL: https://doi.org/10.1145/3437800.3439202.

[94] Juha Sorva. « Reflections on threshold concepts in computer programming and beyond ». In: *Proceedings of the 10th Koli calling international conference on computing education research*. 2010, pp. 21–30.

[95] Philip Guo. « Ten Million Users and Ten Years Later: Python Tutor's Design Guidelines for Building Scalable and Sustainable Research Software in Academia ». In: *The 34th Annual ACM Symposium on User Interface Software and Technology*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1235–1251. ISBN: 9781450386357. URL: https://doi.org/10.1145/3472749.3474819.

[96] Juha Sorva, Ville Karavirta, and Lauri Malmi. « A Review of Generic Program Visualization Systems for Introductory Programming Education ». In: *ACM Trans. Comput. Educ.* 13.4 (Nov. 2013). DOI: 10.1145/2490822. URL: https://doi.org/10.1145/2490822.

[97] Toby Dragon and Paul E. Dickson. « Memory Diagrams: A Consistant Approach Across Concepts and Languages ». In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. SIGCSE '16. Memphis, Tennessee, USA: Association for Computing Machinery, 2016, pp. 546–551. ISBN: 9781450336857. DOI: 10.1145/2839509.2844607. URL: https://doi.org/10.1145/2839509.2844607.

[98] Paul E. Dickson and Toby Dragon. « A Memory Diagram for All Seasons ». In: *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. ITiCSE '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 150–156. ISBN: 9781450382144. DOI: 10.1145/3430665.3456317. URL: https://doi.org/10.1145/3430665.3456317.

[99] *Call stack as diagram*. https://notionalmachines.github.io/nms/CallStackAsDiagram.html. 2010 (accessed January 22, 2022).

Instrumentation

[100] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. « Engineering Record And Replay For Deployability: Extended Technical Report ». en. In: arXiv:1705.05937 (May 2017). arXiv:1705.05937 [cs]. URL: http://arxiv.org/abs/1705.05937.

[101] Chukri Soueidi, Ali Kassem, and Yliès Falcone. « BISM: bytecode-level instrumentation for software monitoring ». In: *Runtime Verification: 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6–9, 2020, Proceedings 20*. Springer. 2020, pp. 323–335.