FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Data-Driven Hint Generation for Alloy using Historial Student Submissions

Ana Inês Oliveira de Barros

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Nuno Macedo

Second Supervisor: Prof. Ana Cristina Ramada Paiva

July 25, 2023

Data-Driven Hint Generation for Alloy using Historial Student Submissions

Ana Inês Oliveira de Barros

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Alexandra Mendes
Referee: Prof. J. Creissac Campos
Referee: Prof. Nuno Macedo

July 25, 2023

Resumo

À medida que a tecnologia se envolve com sistemas críticos, os profissionais começam a adotar especificações formais para desenhar e implementar "software" de confiança. No entanto, escrever e entender especificações formais pode ser um desafio, mesmo com ferramentas do estado-da-arte. No momento de escrita, o Alloy4Fun é a única plataforma de avaliação automática para ensinar especificações formais. Mesmo assim, esta plataforma poderia beneficiar de "feedback" personalizado para utilizadores, um componente essencial do processo de aprendizagem. Os sistemas de geração de dicas automáticas para programação têm uma longa história de investigação com foco em técnicas orientadas a dados. As ideias do estado da arte usam submissões anteriores de alunos para inferir a correção mais comum para chegar a uma submissão correta.

Este trabalho insere-se no âmbito de um projeto financiado pela FCT, o SpecRep, que inclui tarefas para melhorar o "feedback" fornecido pelo Alloy4Fun. Experiências recentes com o Alloy4Fun indicam que a maior parte dos estudantes precisam de ajuda a perceber porque é que as suas especificações falham. Esta situação torna o estudo independente mais difícil, levando a estudantes desmotivados. Felizmente, o conjunto de submissões dos estudantes está aberto ao público, abrindo a possibilidade de desenvolver abordagens baseadas no histórico para Alloy.

Neste trabalho apresentamos o HiGenA, o gerador de dicas para Alloy, uma ferramenta que permite a geração de dicas baseadas no histórico para Alloy integrado no Alloy4Fun. Ela aproveita-se das tentativas de resolução de exercícios de estudantes passados para fornecer dicas a novos estudantes. As dicas geradas são baseadas em operações de edição no nível da árvore sintática abstrata e consistem em pequenas mensagens de texto que guiam os estudantes para uma solução.

O objetivo do HiGenA é melhorar o processo de aprendizagem dos estudantes. Para perceber se atingimos este objetivo, conduzimos experiências e comparamos os resultados deste trabalho com as técnicas de geração de dicas do estado-da-arte para Alloy e outras linguagens de programação. Também conduzimos um estudo com antigos estudantes de Alloy e um professor para avaliar a qualidade e eficácia das dicas geradas. Os resultados mostram que o HiGenA melhorou significativamente a técnica de reparação anterior para Alloy e que as dicas geradas contribuem para o processo de aprendizagem.

Palavras-chave: Alloy; Geração de dicas baseada em dados; Espaço de soluções

Abstract

As technology takes on critical tasks, professionals are starting to embrace formal specifications to design and implement reliable software. However, writing and understanding formal specifications can be challenging even with state-of-the-art tools. At the time of writing, Alloy4Fun is the only automated assessment platform for teaching formal specifications. Nonetheless, this platform could benefit from personalized user feedback, an essential component of the learning process. Automatic hint-generation feedback systems for programming have a long history of research focusing on data-driven techniques. The state-of-the-art ideas take previous students' solutions to infer the most common fix steps to a successful submission.

This work resides in the context of an FCT-funded project, SpecRep, which includes tasks to improve the feedback provided by Alloy4Fun. Recent experiments with Alloy4Fun indicate that most students need help understanding why their specifications fail. This situation leads to unmotivated students by making independent study more difficult. Fortunately, the dataset of students' submissions is publicly available, opening the possibility of developing history-based approaches for Alloy.

This work presents HiGenA, the Hint Generator for Alloy, a tool that allows the automatic generation of history-based hints in Alloy integrated with Alloy4Fun. It takes advantage of students' past attempts to solve an exercise to provide hints to new students. The generated hints are based on edit operations at the abstract syntax tree level and consist of small textual messages guiding students toward a solution.

The goal of HiGenA is to improve the student learning process. To understand if we achieved this goal, we conducted experiments and compared the results of this work with state-of-the-art hint generation techniques for Alloy and other programming languages. We also conducted a user study with former Alloy students and a teacher to evaluate the quality and effectiveness of the generated hints. The results show that HiGenA significantly improved over the previous Alloy repair technique and that the generated hints contribute to the learning process.

Keywords: Alloy; Data-driven hint generation; Solution space

Acknowledgements

Firstly, I would like to thank my supervisors, Prof. Nuno Macedo and Prof. Ana Paiva, for their guidance and support throughout this work. I would also like to thank the participants of the user studies who took the time to help me evaluate the work presented in this thesis. Finally, I want to thank my family and friends for their support and encouragement throughout this journey.

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project EXPL/CCI-COM/1637/2021.

Ana Barros

Contents

List of Figures

List of Tables

Listings

Symbols and Abbreviations

APTED	All Path Tree Edit Distance
AST	Abstract Syntax Tree
CFG	Control Flow Graph
CRISP-DM	Cross Industry Standard Process for Data Mining
DFG	Data Flow Graph
FOL	First-Order Logic
GDS	Graph Data Science
ITAP	Intelligent Teaching Assistant for Programming
ITS	Intelligent Tutoring System
MDP	Markov Decision Processs
MOOC	Massive Open Online Courses
PMAXSAT	Partial Maximum Satisfiability
PSP	Problem Solving Policy
RL	Relational Logic
RTED	Robust Algorithm for the Tree Edit Distance
TAR	Temporal Alloy Repair
TED	Tree Edit Distance
WCC	Weakly Connected Component

Chapter 1

Introduction

1.1 Context and Motivation

Software has become essential in our lives in the last few years. It underpins many of the daily services we use, from banking to transportation and entertainment. However, software is not perfect. Software contains bugs, which are errors in the code that cause the software to misbehave [38].

The problem is even more severe in safety-critical systems, software systems used in situations where the consequences of a failure can be catastrophic. Safety-critical systems include air traffic control systems, nuclear power plants, and medical devices [49].

To solve this problem, professionals have developed techniques to validate and verify software systems. These techniques are called formal methods: mathematically founded techniques to analyze software [27].

The development cycle of a software system using formal method is presented in Figure 1.1. The first phase is defining the requirements of the system. Then, formal specifications formalize the requirements. Next, there is a design and implementation phase. Formal specifications are a key component of this process as they formally define software requirements and are used to validate and verify the software system [27].

Formal specifications are usually written in formal specification languages. These are languages that provide a way of formalizing software requirements. One example of these languages is Alloy [27], an open-source formal modeling language that can declare structures and events and specify constraints.

The Alloy language is similar to first-order logic with relational operators, and the syntax and semantics are straightforward and uniform. Alloy also comes with Alloy Analyzer for automatically analyzing the user-defined models. The analyzer provides many features, the most relevant being a graphical interface for visualizing scenarios and their counter-examples. These characteristics made Alloy popular as a teaching tool in formal methods courses at various universities worldwide [26].

Teachers who taught some of these courses noticed they were missing features that could ease Alloy's adoption by students and professionals [33]. To address these difficulties, they developed

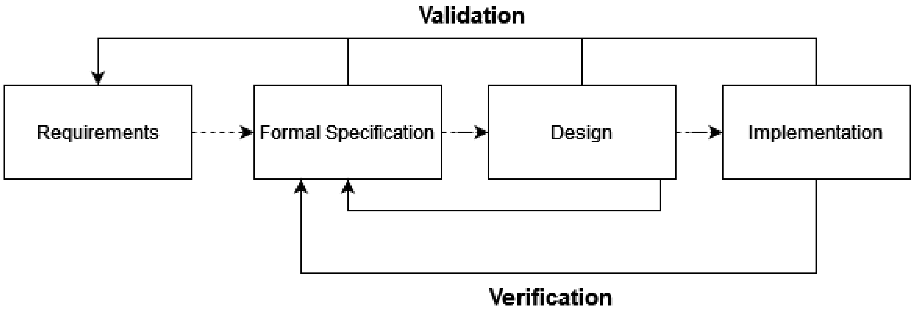

Figure 1.1: Formal methods development cycle.

Alloy4Fun, a web application to introduce Alloy to new students.

At its core, Alloy4Fun gathers the Alloy Analyzer's main features, bringing them to the web. However, the idea behind this platform is to provide the functionality of a tutoring system regarding exercises and automated assessment. In Alloy4Fun, the teachers can create simple specification challenges and easily share them with students who will attempt to solve them. These challenges work for both independent study and as exercises to be done during class.

With the deployment of Alloy4Fun, an opportunity appeared to collect information on how users were using the platform. Collected data could support research on Alloy usage patterns or students' learning processes. Thus, the creators of Alloy4Fun added a feature for recording anonymous user interactions, which produced a dataset of historical student submissions. This dataset is now publicly available, opening the possibility of developing history-based approaches for Alloy.

1.2 The Problem

Over the last few years, Alloy4Fun has been present in various universities' formal methods course editions. The tutors of some of these courses published a paper [33] describing their findings regarding the most common bottlenecks when learning Alloy and identified some learning breakdowns. One of them was the difficulty in debugging an erroneous specification.

Upon attempting to solve an exercise, the only feedback available for the user is the result (correct or not) and a set of counter-examples, if any. Although counter-examples have a dedicated graphical representation, interpreting these diagrams is often challenging.

Several factors might contribute to this problem. Initially, students using Alloy4Fun are still getting used to the concept of the counter-examples and their diagram representation. Furthermore, students are still new to the language and might be uncomfortable with first-order logic.

These difficulties have consequences. Students are left relying on the teacher for assistance. However, it can be troublesome for teachers to attend to every student in bigger classes. Additionally, independent study sessions can become frustrating as students need help interpreting their mistakes. Taking everything into account, students feel unmotivated because they are missing better feedback, an essential component of the learning process.

1.3 Objectives

This work aims to study existing techniques for hint generation in online programming tutors and adapt them to the formal specification context. The extensive Alloy4Fun dataset is available to the public, which opens the possibility of developing a history-based hint-generation technique for Alloy. The idea is to use the existing historical student submissions to find typical repair patterns or evolution paths that previous students have followed to fix their erroneous specifications. Additionally, the calculated steps must be translated into hint messages that will guide the student toward a correct submission (without disclosing the complete solution).

The developed technique should achieve better results than the state-of-the-art tools. Furthermore, it is especially crucial that the hint generation tool is able to generate timely feedback. Otherwise, in the educational context, students might feel frustrated when trying to solve the challenges. In the end, the objective is to deploy the tool in Alloy4Fun.

1.4 Hint Generation Technique for Alloy

This work resulted in a tool named HiGenA, the Hint Generator for Alloy. This tool allows the automatic generation of history-based hints in Alloy and integrates with Alloy4Fun. The technique exercised by HiGenA depends on the concept of a solution space, which consists of a directed graph where states are student submissions, and edges represent edits that transform one state into another.

HiGenA locates the student submission in the graph upon receiving a hint request. Then, it finds a path between the student submission and a solution. Finally, it generates a hint message based on the first transition of the path found. The transitions contain the edit operations at Alloy's abstract syntax tree level that transform one state into another. The generated hint is a small textual message that encourages the student to apply the same edit operations to their submission to get closer to a solution.

As a result of this work, we have successfully developed a hint generation technique that outperforms the state-of-the-art tool, TAR [9]. Additionally, we conducted a user study to evaluate the quality of the generated hints and the results show that students found the generated hints useful and easy to understand.

1.5 Document Structure

The remainder of the documents is structured as follows. Chapter 2 presents the background knowledge and context required to understand the rest of the document. This chapter includes a detailed explanation of the Alloy language and a presentation of the Alloy4Fun web platform. Chapter 3 describes the current state-of-the-art which includes the different types of feedback available to students and the diverse techniques for hint generation used by Intelligent Tutoring Systems. Chapter 4 presents the solution and the implementation of HiGenA. Chapter 5 presents the evaluation of HiGenA along with the results obtained. Finally, Chapter 6 presents the conclusions and future work.

Chapter 2

Background

This chapter presents the background of the work presented in this document. Its objective is to provide the reader with the necessary context to understand the rest of the document. It comprises two main parts: Section 2.1 briefly introduces the Alloy language and its different concepts while Section 2.2 presents the Alloy4Fun platform, along with its main features.

2.1 Alloy Overview

In Chapter 1, we introduced Formal methods. These are mathematically founded techniques to validate and verify software systems. Formal specifications are a key component of these techniques as they formally define software requirements. Alloy is an example of a formal specification language.

The Alloy language is simple and straightforward but powerful enough to express complex software systems. Its simplicity comes from the fact that it is based on first-order logic with relational operators, which is familiar to mathematicians and computer scientists. Furthermore, it allows the user to declare structures and events and specify constraints [27].

2.1.1 Static Structure

In Alloy, entities are represented by signatures. To declare a signature, the user uses the `sig` keyword. Signatures represent sets inhabited by atoms from a finite universe. Signatures can also extend other signatures, inheriting the atoms from the extended signature. For each signature, the user can declare their multiplicity. Table 2.1 shows the different multiplicity constraints in Alloy.

Table 2.1: Alloy multiplicity constraints.

Alloy	**UML**
set	0..*
lone	0..1
some	1..*
one	1

In Alloy, fields represent relations that contain sets of tuples from the universe. The user can declare fields in signatures and specify their multiplicity and types. These tuples are always subsets of the Cartesian product of the source and target type signatures.

Listing 2.1 shows an example of a simple Alloy model of a filesystem using signatures and fields. This model declares several signatures: `Dir` represents a directory, `File` represents a file, `Root` represents the root directory, `Object` represents an object contained in a directory, `Name` represents the name of an object, and `Entry` represents an entry in a directory. Some signatures extend others, e.g., the `Dir` signature extends the `Object` signature. The field `entries` is a set of tuples with pairs of `Dir` and `Entry` atoms. The multiplicity constraints specify that for each `Entry` in the `object` field, there is exactly one `Object`. Similarly, for each `Entry` in the `name` field, there is exactly one `Name`.

```
sig Object {}
sig Name {}
sig File extends Object {}
sig Root extends Dir {}

sig Dir extends Object {
    entries: set Entry
}

sig Entry {
    object: one Object,
    name: one Name
}
```

Listing 2.1: Alloy simple example of signatures and fields.

2.1.2 Relational Formulas and Expressions

Alloy provides typical logic operators, quantifiers, and atomic formulas for expressing constraints in first-order logic (FOL). However, specifying complex constraints in FOL can be tedious and error-prone. For this reason, Alloy also provides relational logic (RL), which is a logic that extends FOL with relational operators, derived atomic formulas, and transitive and reflexive closures.

Table 2.2 shows the RL operators and formulas by category. Each entry's significance is written in mathematical notation and in FOL. This table does not include closures since they are not possible to express in FOL.

There are two types of closures: transitive and reflexive transitive. The transitive closure of a relation `R` is denoted by `^R` and is defined as `R + R.R + R.R.R + R.R.R.R. + ...`. On the other hand, the reflexive transitive closure of a relation `R`, denoted by `*R`, is defined as `^R + iden`.

Table 2.2: Relational logic in Alloy.

Category	Atomic Formula	Math Notation	FOL
Subset	`R in S`	$R \subseteq S$	$\forall x_1,\ldots,x_n \cdot (x_1,\ldots,x_n) \in R \rightarrow (x_1,\ldots,x_n) \in S$
	`R not in S`	$R \not\subseteq S$	
Set Equality	`R = S`	$R = S$	$R \subseteq S \wedge S \subseteq R$
	`R != S`	$R \neq S$	
Cardinality	`some R`	$\|R\| > 0$	$\exists x_1,\ldots,x_n \cdot (x_1,\ldots,x_n) \in R$
	`no R`	$\|R\| = 0$	$\forall x_1,\ldots,x_n \cdot (x_1,\ldots,x_n) \notin R$
	`lone R`	$\|R\| < 2$	
	`one R`	$\|R\| = 1$	
Set Operators	`R + S`	$R \cup S$	$(x_1,\ldots,x_n) \in (R+S)$
	`R & S`	$R \cap S$	$(x_1,\ldots,x_n) \in (R\&S)$
	`R - S`	$R \setminus S$	$(x_1,\ldots,x_n) \in (R-S)$
RL constants	`univ`	$\top$	$\forall x \cdot (x) \in univ$
	`none`	$\emptyset$	$\forall x \cdot (x) \notin none$
	`iden`	id	$\forall x_1,x_2 \cdot (x_1,x_2) \in iden \leftrightarrow x_1 = x_2$
RL operators	`R -> S`	$R \times S$	$(x_1,\ldots,x_n,y_1,\ldots,y_m) \in (R \rightarrow S)$
	`~R`	R°	$(x_1,x_2) \in (\sim R) \leftrightarrow (x_2,x_1) \in (R)$
	`R :> S`		$(x_1,\ldots,x_n) \in R \wedge (x_n) \in S$
	`S <: R`		$(x_1,\ldots,x_n) \in R \wedge (x_1) \in S$
Composition	`R . S`		$\exists z \cdot (x_1,\ldots,x_{n-1},z) \in R \wedge (z,y_2,\ldots,y_m) \in S$

An example of a constraint using closures is the following: `Object in Root.*(entries.object)`. This example indicates that all objects are reachable from the root.

Listing 2.2 shows an example of a FOL formula and its relational logic equivalent. The first expression is a pure FOL formula followed by relational logic simplifications. The last relational logic formula is more concise and easier to read and write than the FOL formula.

This formula starts by declaring an atom `o` contained in the set of the signature `Object` excluding the signature `Root`. Then, it indicates that some atom exists from the composition `object.o`. The field `object` is a set of tuples with pairs (`Entry`,`Object`). Thus, the result of this composition is a set of `Entry` atoms. So, putting it simply, the formula states an entry exists for every object except the root. This example shows how relational logic is a powerful tool for expressing complex constraints in Alloy.

2.1.3 Facts, Predicates, and Functions

Facts usually contain constraints, one per line. They are used to define the model's static structure. A fact is a Boolean formula that must be satisfied by all instances of the model. On the other hand, assertions are defined using the **`assert`** keyword and specify named constraints to be checked individually.

```
    all o: univ | o in Object and o != Root implies
        some e: univ | e→o in object
    all o: univ | o in Object and o != Root implies
        some e: univ | e in object.o
    all o: univ | o in Object and o != Root implies some object.o
    all o: univ | o in Object and o not in Root implies some object.o
    all o: univ | o in Object-Root implies some object.o
    all o: Object-Root | implies some object.o
```

Listing 2.2: FOL to relational logic example in Alloy.

Alloy also allows the user to declare functions and predicates defined using the **`fun`** and **`pred`** keywords, respectively. Predicates are a special case of Alloy functions. They take a number of parameters and return a Boolean value. Predicates work as reusable constraints and only hold when invoked in facts, commands, or other predicates. So, once a predicate is defined, it can be used in Boolean expressions. Functions have the same structure as predicates, but they can return any type of value and show up in the visualizer. They might define values to use inside facts, assertions, and other functions.

Listing 2.3 shows an example of a fact in Alloy that contains the constraint defined in Listing 2.2. Additionally, it contains an example of a function and a predicate for specifying that a directory can not contain itself.

```
pred acyclic [r: univ → univ] {
    no ^r & iden
}

fun descendants [d: Dir] : set Object {
    d.^(entries.object)
}

fact {
    // An entry some for every object except the root
    all o: Object-Root | implies some object.o

    // A directory cannot be contained in itself
    acyclic[entries.object]
    // or
    all d: Dir | d not in d.^descendants

}
```

Listing 2.3: Example of a fact in Alloy.

2.1.4 Verifying Structural Properties

Alloy is a modeling language and comes with the Alloy Analyzer for automatically analyzing user-defined models. It applies model checking, a technique used to validate software systems. More specifically, model checking consists of verifying whether a given model satisfies a given property, where a model is a system representation and a property is a condition that the system must satisfy.

The Alloy Analyzer comes with a bundle of SAT (boolean satisfiability) solvers for model checking. SAT solvers are programs that determine whether a given Boolean formula is satisfiable. If the formula is satisfiable, the solver returns the set of values for the formula's variables that make the formula true. Otherwise, the solver returns the proof of unsatisfiability.

In the case of the Alloy Analyzer, the solver takes the model's constraints and finds structures that satisfy them. For unsatisfiable models, the solver finds structures that violate the constraints. These structures are called counter-examples and are used to prove the unsatisfiability of the model.

Moreover, the Alloy Analyzer provides a graphical interface called Alloy Visualizer for visualizing scenarios and their counter-examples. It allows the user to understand the model's structures and relations and navigate the various instances and counter-examples.

Alloy provides a set of commands for analyzing models. There are two types of analysis commands: (1) the **`check`** command that asks for a counter-example to a given property and (2) the **`run`** command that asks for a scenario of a given property. If the property is not satisfied, the **`check`** command finds a counter-example. Otherwise, if the property is satisfied, the **`run`** command finds an example of a valid scenario where the property holds.

Commands always have a scope that imposes a limit on the universe that the Analyzer will consider. The default scope is limited to 3 atoms per top-level signature. To specify a different scope, the user can use the **`for`** keyword for top-level signatures and the **`but`** keyword for specific signatures.

Listing 2.4 shows an example of both **`check`** and **`run`** commands in Alloy. In this case, the **`run`** command is verifying that the specific scenario of an empty file system is valid. The **`check`** command is verifying that the property of `NoPartitions` is valid for a finite scope.

2.1.5 Dynamic Structure

In Alloy 6, the language introduced the concept of mutability. This concept allows the user to declare mutable signatures and fields using the keyword **`var`**. Introducing mutability in Alloy turns instances into infinite sequences (traces) of snapshots (states).

Listing 2.5 shows an example of a mutable signature and field in Alloy denoted by the **`var`** keyword. This example represents a dynamic structure of nodes in a leader election protocol. The `Elected` signature is mutable and represents the elected node. Additionally, both the `inbox` and `outbox` fields are mutable and represent the messages received and sent by the nodes.

```
assert NoPartitions {
    // All objects are reachable from the root
    Object in Root.*(entries.object)
}

check NoPartitions for 6

run {
    // An empty file system
    Object = Root
}
```

Listing 2.4: Analysis example in Alloy.

2.1.6 Temporal Logic

Alloy 6 introduced temporal logic to specify the properties of traces. This temporal logic includes a set of both past and future operators. Future operators include **`always`**, **`eventually`**, and **`after`** operators. Past operators include: **`historically`**, **`once`**, and **`before`** operators.

We can define temporal behavior in Alloy through arbitrary restrictions. However, the standard is to work with Alloy 6's transition system, which facilitates the specification of temporal logic formulas. Formulas without temporal logic operators always define the system's initial state. On the other hand, temporal logic operators allow writing temporal logic formulas to specify the valid traces of the transition system. Formulas using these operators create a relationship between consecutive states of the system and specify the events that can occur.

An event is composed of three parts: (1) the Guards, which specify when an event can occur; (2) the Effects that indicate the changes caused by the event; and (3) the Frame conditions that specify what remains unchanged. Furthermore, Guards usually contain no temporal logic operators but might contain past operators to recall previous states. In contrast, effects and frame

```
open util/ordering[Id]
sig Id {}

sig Node {
    succ : one Node,
    id : one Id,
    var inbox : set Id,
    var outbox : set Id
}

var sig Elected in Node {}
```

Listing 2.5: Dynamic structure example in Alloy.

```
pred initiate [n: Node] {
    // guard
    historically n.id not int n.outbox

    // effect
    outbox' = outbox + n->n.id

    // frame conditions
    inbox' = inbox
    Elected' = Elected
}

fact init {
    no inbox
}

fact events{
    always (
        some n: Node | initiate[n]
    )
}
```

Listing 2.6: Alloy 6 dynamic example with temporal operators.

conditions use only **after** and ′ (next state) operators.

Listing 2.6 portrays an example of an Alloy 6 event for initiating a node in a leader election protocol. A dynamic fact contains this event, while a static fact indicates the system's initial state (does not contain any temporal logic operators).

2.1.7 Dynamic Model Checking

Although mutability introduces infinite traces, analysis commands only return traces that can be represented finitely (traces that loop back at some point). The default verification method in Alloy 6 is bounded model checking via SAT with a default number of steps of 10. These are the number of steps the Analyzer will explore before looping back. However, it is possible to change the number of steps or to support unbounded model checking by installing other model checkers.

Listing 2.7 shows an example of a **check** command in Alloy 6 that verifies the `AtLeastOneLeader` property. In this case, the command is checking if the property holds for a finite scope of 3 and more than 1 step.

```
assert AtLeastOneLeader {
    fairness implies eventually (some Elected)
}

check AtLeastOneLeader for 3 but 1.. steps
```

Listing 2.7: Dynamic analysis example in Alloy 6.

2.2 Alloy4Fun Overview

In Chapter 1 we briefly introduced Alloy4Fun, a web-based platform for teaching and learning Alloy. This section provides a more detailed overview of the Alloy4Fun platform and its main features, which are described in [33].

2.2.1 Introduction

Alloy4Fun brings to the web the main features of the Alloy Analyzer, such as the ability to create and analyze models. The interface of Alloy4Fun presents an online editor with syntax highlighting where users can create and edit their own models. Teachers can use this editor to create specification challenges for their students. Students use the editor to solve these challenges and learn at their own pace.

2.2.2 Challenges

Figure 2.1 shows an example of a challenge[1] in Alloy4Fun with a simplified model of a file system. Each exercise is a different predicate in the code editor that the student must complete.

A challenge in Alloy4Fun consists of the user trying to reach a specification written by the teacher. To create a challenge, the teacher must write a model with public predicates and hidden parts. The students must correctly fill in these empty predicates to reach the correct specification and complete the challenge. Correct answers must be semantically equivalent to the teachers' specifications, but not necessarily syntactically equivalent.

For each public predicate, the teacher must write a secret **`check`** command. To create a hidden predicate in Alloy4Fun, the teacher must write a special comment *`//SECRET`* immediately before. These commands are hidden from the students and are used to check if the students' answers are equivalent to the desired specification written by the teacher. The correct specification for each challenge is also hidden from the students in a separate predicate.

2.2.3 Sharing Feature

Alloy4Fun allows its users to share their models and instances using links. In the interface, there is one button to share a model and another to share an instance. Clicking one of these buttons generates a permanent link to the model or instance.

When the teachers finish creating a challenge, they can share it with their students. Sharing a model generates a public and a private link. The private link contains the full model written, while

[1] Available at `http://alloy4fun.inesctec.pt/N5DGYQxrHv4LQZQ3H`

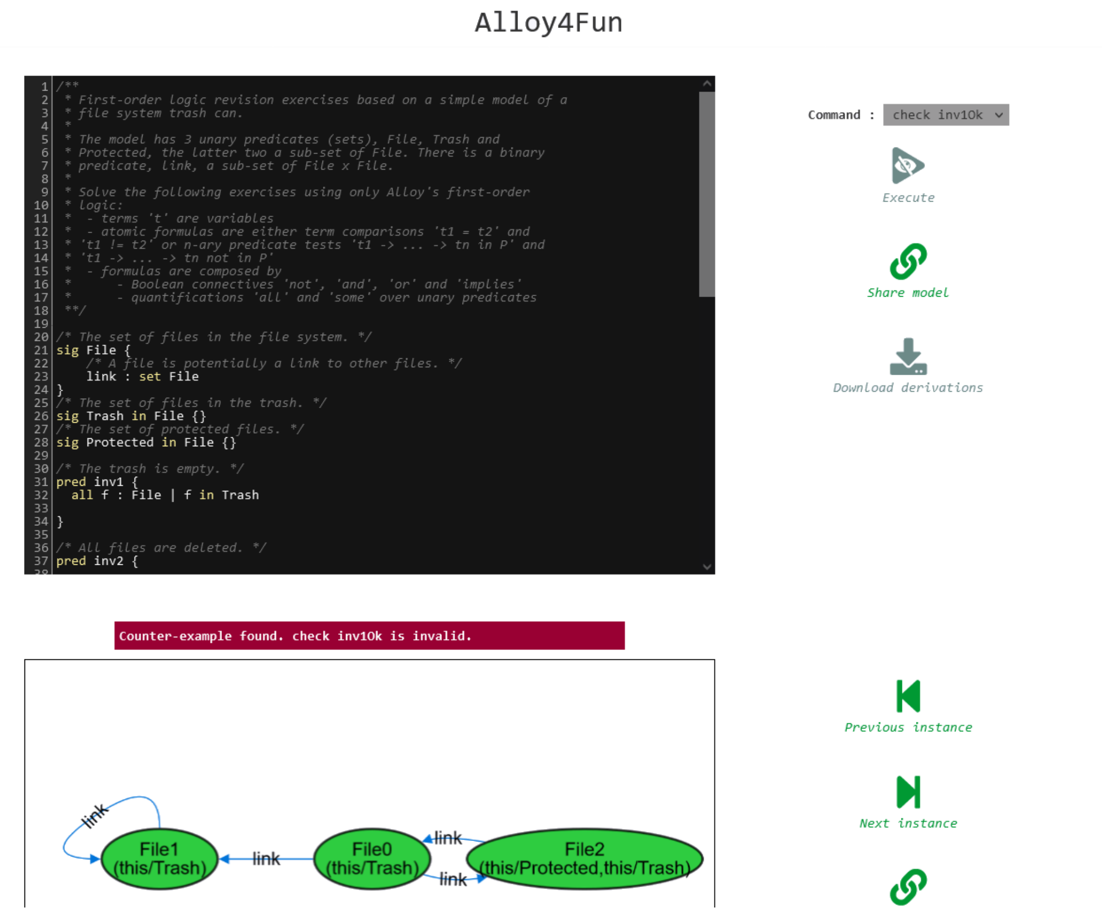

Figure 2.1: Alloy4Fun challenge example.

the public link hides the secret paragraphs. This public link is the one that the students must use to access the challenge.

Similarly, students can share their solutions with their teacher when they solve a challenge or when they need help and want to receive feedback.

2.2.4 Automated Assessment

Behind the scenes, Alloy4Fun uses the Alloy Analyzer to check the correctness of the user's answer through **check** commands that belong to the model but are hidden from the user. The teacher writes these commands using the secret comment *//SECRET*. They automatically check if the students' answers are semantically equivalent to the teachers' answers. As the students' answers do not need to be syntactically equal to the teachers' answers, there is a range of possible solutions.

So, when the students want to check if their specification is correct, they have to execute these analysis commands. Alloy4Fun's interface provides a dropdown box to select the command to execute. For instance, if the user wants to check the first predicate's correctness, they must select

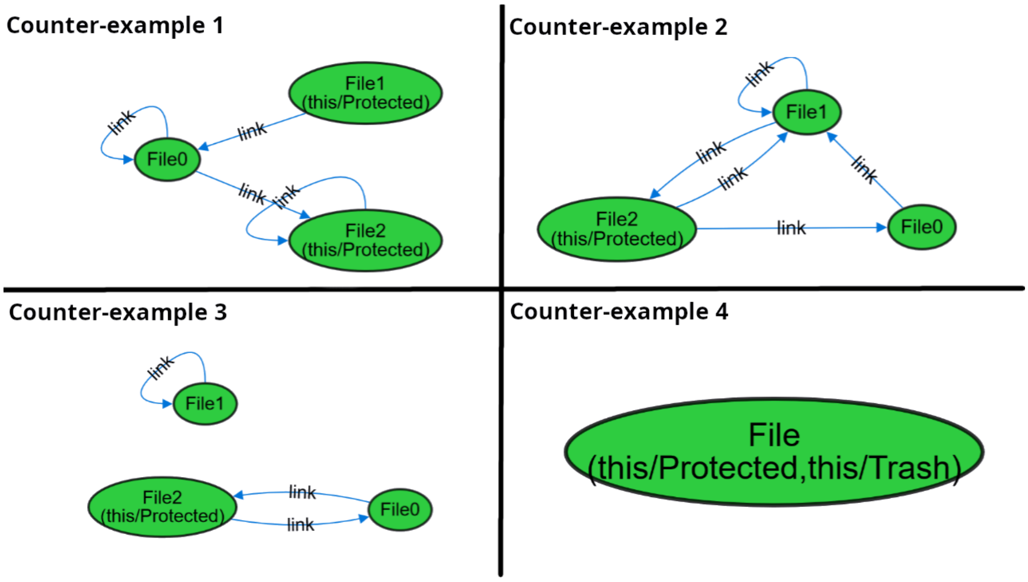

Figure 2.2: Some Alloy4Fun counter-examples for challenge depicted in Figure 2.1.

the respective command for that predicate. Then, the user has to click the *Execute* button to run the command.

After executing the command, Alloy4Fun presents the results of the analysis. For a `check` command, if the specification is correct, Alloy4Fun presents a green message indicating that the command failed to find any counter-example. Otherwise, a red message shows up, and the first counter-example is presented below the editor. The latter situation is shown in Figure 2.1.

At its core, Alloy4Fun works as a tutoring system. Users can write their answers and check their correctness through the click of a button. However, from the user's perspective, they do not feel like they are working only with a plain Alloy model. The advantage of Alloy4Fun is that the user does not need to understand the concept or how to verify the properties of the model in Alloy, making it ideal for beginners. Instead, they can focus on correctly defining properties to solve the challenge.

2.2.5 Debugging Specifications

Figure 2.2 shows some counter-examples presented for the exercise from Figure 2.1. By clicking on the *Next* and *Previous* buttons, the user can navigate through the counter-examples. These counter-examples are presented in graph form, where each node represents an atom, and each edge represents a relation. Moreover, the user can customize the theme and layout of the counter-examples for better visualization.

At the moment of writing, this is the only way available to help the user understand why their specification is incorrect. For simple models, this approach is enough to understand the problem. However, students struggle to understand the counter-examples for more complex models.

2.2.6 Data Collection

Alloy4Fun collects anonymous data about how users interact with the platform. It stores all shared models, shared instances, and every model when executing a command. So, whenever a student runs a command, the platform stores several things in its database. It stores the full model, the result of the analysis (satisfiable or not), the timestamp, and the id of the model it derived from.

Storing every student submission allows Alloy4Fun to build a derivation tree of all models developed after a shared model. Although all data collected remains anonymous, a branch in this tree typically represents a session where a student is trying to solve a challenge. If the student successfully solves the challenge, the branch contains the steps taken by the student to reach the correct solution. Additionally, generating a new link in these branches creates a fork that represents a new session.

The owner of a shared model with secrets can download the derivation tree of the challenge. Thus, this data then allows tutors to understand how students solve the challenges, if they are learning and understanding new concepts or facing difficulties.

Some tutors have been maintaining a dataset[2] of models submitted in the Alloy4Fun during various editions of formal methods courses in the University of Minho (UM) and the University of Porto (UP). In total, it contains about 100,000 models submitted between 2019 and the spring of 2023. Since the challenges are publicly available, the dataset also contains submissions from participants outside of the classroom context.

This data is used for research purposes, such as developing data-driven hint-generation techniques like the one presented in this document.

2.3 Conclusion

In this chapter, we presented the Alloy formal specification language and Alloy4Fun. These two concepts represent the required background knowledge to understand the rest of this document.

To summarize, Alloy is a formal specification language that allows users to analyze and create models of systems. It comes with tools like the Alloy Analyzer that uses model checking to verify the correctness of models.

The Alloy4Fun platform takes advantage of the Alloy Analyzer to provide a web-based interface for creating and analyzing models. This platform is used in the educational context as a tutoring system for teaching and learning Alloy. This platform also stores user data, including all students' submissions for research purposes.

In the next chapter, we present the state-of-the-art in the field of hint-generation techniques for programming tutoring systems.

[2] The dataset is publicly available at `https://zenodo.org/record/8123547`

Chapter 3

Literature Review

3.1 Introduction

This chapter presents a systematic review of the literature on automatic hint-generation techniques. The review is organized into six main sections. Section 3.2 details the systematic review process used to collect, analyze and summarize the existing literature on the topic. Then, Section 3.4 introduces the field of Intelligent Tutoring Systems and the different types of feedback that can be provided to students.

After that, we start presenting the different techniques that have been proposed for automatic hint generation. These techniques are divided into two main categories: the techniques that do not use the student's historical data to generate hints, described in Section 3.5, and those that do, presented in Section 3.6. Moreover, for each technique, we describe its hint-generation process, its evaluation, obtained results and limitations.

Additionally, Section 3.7 presents some techniques for comparing student data to ease the process of hint generation using historical data. Lastly, Section 3.8 presents this review's conclusions, including a small discussion of the findings from the literature.

3.2 The Systematic Review Process

The systematic review process is a method to collect, analyze and summarize the existing literature on a given topic. To maximize the reliability of the final results, this process was divided into five steps, as suggested by [29]: (1) framing questions for a review, (2) identifying relevant work, (3) assessing the quality of studies, (4) summarizing the evidence, and (5) interpreting the findings.

Two questions guided this review. The first question was: What hint-generation techniques for programming exist? The second question was: How do these techniques perform? These two questions directly impacted the search strategy: The first question aided in identifying relevant work. In contrast, the second one served as a way to assess the quality of the studies.

3.2.1 The Search Strategy

The search strategy involved searching for relevant work in a wide range of scientific databases to capture as many relevant studies as possible. The most visited databases were: Scopus, Google Scholar, Springer Link, ACM Digital Library, ScienceDirect, IEEE Xplore, and Semantic Scholar. Furthermore, this strategy also included using various internet engines to search web pages that could provide references and relevant information.

Using the keywords “hint generation” in Scopus resulted in 20,040 hits. However, the majority of these hits were not relevant to the topic. So, to remove the irrelevant hits, we adjusted the query to include works with other keywords. The final query was “automatic hint generation AND tutoring AND historical student data”, and reduced the number of relevant hits to 155.

Finally, we used this query to search the databases mentioned above. The final search results were sorted by relevance and manually reviewed to identify relevant work. Additionally, we followed a snowballing approach which involved searching for references in the papers and following them to discover other pertinent papers [57].

3.2.2 The Review Process

The review process involved reading the papers, assessing their quality, and extracting the relevant information. This revision process implied reading the titles followed by the abstracts of the papers and, if necessary, reading the full text. Furthermore, to evaluate the papers’ quality and identify the most relevant papers, we checked whether the paper was published in a peer-reviewed journal and if it answered the two questions. Additionally, we also considered the following criteria: (1) the type of the paper (conference paper, journal paper, book chapter, etc.), (2) the research question, (3) the results, and (4) the discussion.

The type of the paper helped identify the type of work in the document, and research questions identified the paper’s primary goal. The results described the techniques present in the study. Finally, the discussion provided the conclusions of the paper.

3.2.3 Summarizing and Interpreting the Findings

The final steps of the systematic review process implied summarizing and interpreting the findings. While the summary of these findings involved reading the papers and extracting the most relevant information, their interpretation involved comparing the results of the papers and identifying the commonalities and differences between them. The summary of the findings is presented in the following sections.

3.3 Introduction to Hint generation

Hints have always been common in areas that involve logic challenges. For example, games in the entertainment industry often rely on hints to assist the player in completing puzzles [52]. In the education area, students usually use hints to correct mistakes or to compare their approach to the solution proposed by the teacher [13]. Teachers also tend to give hints to students to help them remember concepts or to teach something new [23].

Until recently, there was little research on automatic hint generation and data-driven tutoring. However, the impact of the rise of Massive Open Online Courses (MOOCs) in the last few years led to urgent research of hint mechanisms at scale [39]. This chapter describes the different aspects of computer hint generation and proposed techniques.

3.4 Intelligent Tutoring Systems

Providing automatic feedback to students comprises the field of Intelligent Tutoring Systems, most commonly referred to as ITSs. An ITS aims to give the same functionality as a human tutor by providing exercises and automatic feedback. The type of feedback an ITS gives comes in three forms: correctness feedback, error-specific feedback, and next-step hints [46]. The author of [31] presents some of the feedback techniques implemented by education systems categorized by each type of feedback.

3.4.1 Correctness Feedback

The first type of feedback is correctness feedback, indicating whether a student's solution is correct. Typical educational systems that employ this type of feedback includes QuizJet for Java programming [22], QuizPack for C programming [7], and M-PLAT [37].

The most common technique for correctness feedback is comparing the student's solutions with pre-defined correct values. Then, depending on whether the student's solution matches these values, the system presents the feedback message indicating the correctness of the solution.

Another popular method implemented by ITSs is to use a suite of test cases as an oracle, as shown in the LISP tutor [12], one of the first ITSs. In Alloy4Fun, there is a similar approach. The tutor can define analysis commands that are used to check the correctness of the solution by comparing the semantic equivalence of the student submission with the one provided by the tutor.

3.4.2 Error-specific Feedback

Error-specific feedback indicates what is wrong with an incorrect solution. A solution can be syntactically or semantically incorrect. Most systems return the compiler's error messages without additional information for syntax errors, e.g., Ludwig [48], Ask-Elle [18], and Alloy4Fun, which returns the Alloy Analyzer errors. On the other hand, the most common approach for providing semantic feedback is to design test cases that match common errors and return the corresponding feedback message.

3.4.3 Next-step Hints

Finally, next-step hints show the student the correct next step to solving a problem. This feedback case is much harder to generate and will be the focus of this work.

Next-step hints show the student the correct next step to solving a problem. There are two ways to provide these hints: given on-demand by the student or provided based on specific events (e.g., when the student makes a mistake). Research states that on-demand hints tend to achieve higher learning rates [45].

3.4.4 Classification of Hint Generation Techniques

There are several techniques employed by ITSs for hint generation. For this work, we classify these techniques into two groups: the ones that use historical student submission and those that do not.

The former category includes techniques that make extensive use of the students' previous submissions to generate hints. To summarize, it implies storing historical student submissions throughout the lifetime of the system and using a technique that takes advantage of this data to generate hints for future students.

The latter category includes techniques that do not store or that do not make extensive use of the student's previous submissions for hint generation. Instead, these techniques usually involve other sources of information, such as the problem's domain, programming strategies, automated repair, or common errors made by students.

The following sections summarize the different existing techniques for each category, how they perform, and their limitations.

3.5 Hint Generation Approaches Without Historical Data

This section describes part of the extensive body of work on hint-generation techniques that do not generate hints from historical student submissions. This set of techniques includes simpler approaches like pre-defining hints and more complex approaches like automated repair, each with its own advantages and disadvantages. Moreover, it includes techniques that use strategy-based models that tend to apply to specific programming languages and domains.

3.5.1 Pre-defined Hints

The most common way to provide next-step hints is to pre-define hints when building a tutor. For example, creating hint messages for each state a student might reach while doing an exercise. This approach does not scale since it takes much work, even for simple problems [15]. It is only viable for domains with a reduced number of possible states.

3.5.2 Strategy-based Models

Hong's PROLOG [21] presents a technique for identifying the programming strategy used by the student. This approach implies modeling the strategies typically used in the programming language and giving feedback based on the status of the student's solution.

This tutor uses grammar rules to model Prolog programming techniques. If these rules can successfully parse the student's program, the system can identify the strategy the student is trying to follow. Then, for each possible strategy, there is an example reference program that the system parses using the same rules. The comparison between the resulting Abstract Syntax Tree (AST) and the student's solution AST allows the system to diagnose errors in the student's solution. This diagnosis can be used for hint generation to provide next-step hints.

Similar to the first approach, the main problem with this technique is that it does not scale. Providing solutions for each problem using different strategies and a set of grammar rules for identifying each programming strategy can take much effort. Although it might not require much work in the case of Prolog, it is unfeasible for other programming languages or challenges that allow a diversity of solutions. Additionally, creating different parsing rules for each strategy is not a simple task to accomplish.

There are many modeling techniques similar to this one [31]. However, each technique only deploys in a single specific system instead of being practiced across several tools. There are two reasons for this phenomenon. Firstly, each modeling technique only applies to specific programming languages. Each programming language has its concepts and programming paradigms, so it is unfeasible to cover a high number of concepts and paradigms. Secondly, it requires much time and effort to devise a modeling technique for a standard programming language with specific concepts. So, researchers usually lack resources for applying the technique to a new programming language or concept.

3.5.3 Automatic Repair

ITSs have been using automatic hint generation to guide students in fixing their erroneous code. This technique is called automated repair and consists of applying automated repair techniques to fix a student's solution and deriving hints from the sequence of repairs [35]. At the moment of writing, this technique is the only one that have been proposed for the specification domain. For that reason, this section will focus on automatic repair for specifications instead of other domains.

There are two possible classifications for automated repair approaches [17]. The first type is search-based, which consists of applying change operators for searching for solutions and validating them with the oracle. An example of this technique is present in [11] and consists of an approach to fix OCL constraints against existing model instances. SpeciFix [43] is another search-based technique that repairs pre- and post-conditions of Eiffel routines using test cases.

The second type of automated repair is semantics-driven which encodes the repair as a constraint to generate the correct fix. A technique presented in [8] fits this category: It repairs B-models using machine learning and repairing the state machine rather than the broken specifications.

Although automated repair for specifications is explicitly still unexplored, some approaches exist for Alloy: ARepair [53, 54], BeAFix [5, 6], and TAR [9]. All these techniques are mutation-based, meaning they generate mutants - a copy of the model with small syntactic transformations - to repair the specification.

Firstly, ARepair uses AlloyFL [55], a mutation-based fault localization framework that takes test cases and the model and returns a list of suspicious AST nodes. Then, it checks if the mutation provided by AlloyFL causes previously failing tests to pass. Otherwise, the tool enters an iterative process of synthesizing code until meeting a criterion that depends on the search strategy chosen.

One problem with ARepair is that the fix provided often passes all the required tests but does not hold all the expected properties. So, to avoid overfitting, some effort is required to create a set of tests that allows the tool to achieve an acceptable level of accuracy.

On the other hand, BeAFix works better for the Alloy4Fun context since it uses the check commands of an Alloy specification as oracles. Furthermore, it uses a different fault localization tool for Alloy, FLACK [58], which, unlike AlloyFL, only runs once for the initial model.

BeAFix defines a set of mutation operators, which are then combined to generate fix candidates. Then, it tests the mutated expressions against the oracles. For optimization purposes, BeAFix applies two pruning strategies by exploiting suspicious locations and many failing oracles. However, since the typical use case of Alloy4Fun would be to fix a small predicate that failed against a single oracle, these techniques are ineffective in this context.

Unfortunately, both ARepair and BeAFix only support the non-temporal version of Alloy and their performance do not match the requirements for hint generation. It is especially crucial in the educational context that the hint-generation tool can provide timely feedback. Otherwise, students feel frustrated when trying to fix their solutions.

This issue of timely feedback was the primary motivation for TAR, the Temporal Alloy Repair tool for timely automated specification repair, which ended up being the first mutation-based technique for repairing Alloy 6 first-order temporal logic specifications. Its purpose is to repair submissions for specification challenges like those in Alloy4Fun. The repairing technique employed by this tool involves searching for syntactic mutations to repair the incorrect specification according to a single oracle. For timely hint generation, the technique implements pruning techniques based on previous counterexamples.

The evaluation presented for TAR indicates that it considerably performs better than ARepair and BeAFix (considering only the subset of Alloy language supported by them). However, TAR could only fix 56% of the erroneous specifications in under 1 minute, while it failed to fix 46% on time due either to time-out or to an exhaustion of the search space.

Although TAR achieved good results, it still needs performance improvements to generate hints faster. Furthermore, there is no information on the tool's performance for depth levels above three, which might indicate that results for these levels are not satisfactory.

Moreover, Zheng et al. [58] presented a new automated program repair approach for software specifications written in Alloy 4.1 named ATR. ATR is a templated-based repair tool, and it is similar to BeAFix as it relies on assertions. Unlike test cases used in ARepair, these assertions are compatible with specification practices in Alloy.

Figure 3.1 depicts an overview of ATR. Similarly to previously described tools, ATR starts with obtaining a list of suspicious expressions from FLACK, the same fault localization tool used

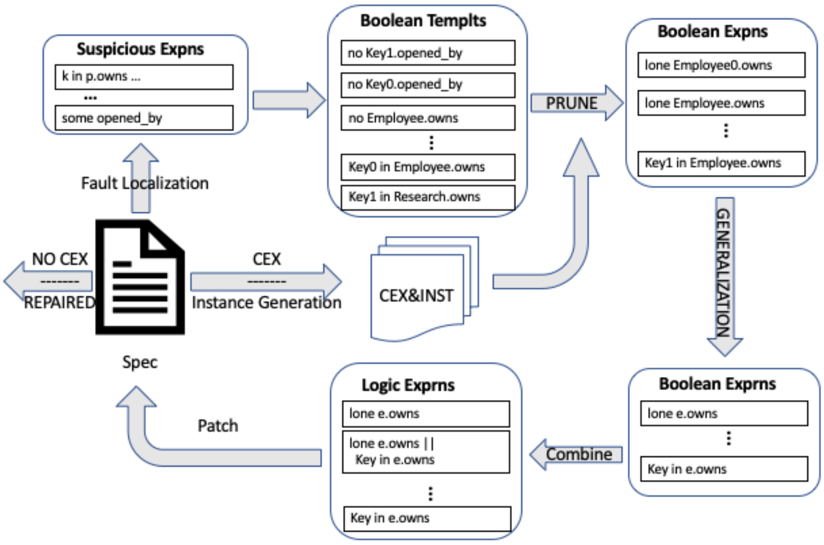

Figure 3.1: Overview of ATR [58].

by BeAFix. Then, ATR queries the Alloy Analyzer for counterexamples (instances of the specification that do not satisfy the property).

For the next step, the technique requires satisfying instances with as few differences as possible. ATR uses a partial max satisfiability solver (PMAXSAT) to achieve this. A PMAXSAT solver tries to find an instance that satisfies all hard constraints and as many soft constraints as possible. So, ATR sets the counterexample as a soft constraint and the property and specification as hard constraints. This problem definition causes the solver to generate satisfying instances as similar as possible to the counterexample.

Next, ATR generates boolean templates using atoms, signatures, and fields and prunes them based on their results evaluated from the pairs of satisfying instances and counterexamples. The tool then generalizes the remainder templates to Alloy expression and combines these using logic operators.

In the final step, ATR patches the suspicious statements using the generated expressions and uses the AlloyAnalyzer to check for counterexamples using the new specification. If Alloy finds any, ATR repeats the process using these counterexamples. Otherwise, it means that ATR successfully repaired the specification.

ATR was able to repair 66.3% faulty specifications, outperforming ARepair's repair rate of 9.8% and BeAFix's repair rate: of 50.9%. It also achieved an average run time of 364.4 seconds (6 minutes) for each fix. This value outperforms the 1569.8 seconds of BeAFix but exceeds the 91.9 seconds of ARepair. Additionally, experiments show that ATR can repair nontrivial bugs and synthesize new expressions to complete empty predicates. Generated repairs usually have a short syntax edit distance, similar to manual repairs.

3.6 Hint Generation using Historical Student Submissions

This section presents several techniques that use historical student submissions to generate hints. The main idea behind these approaches is to take advantage of the ways students solved problems in the past to suggest hints to students who are currently solving the same problem using the same strategy.

3.6.1 Markov Decision Process Based Techniques

The Hint Factory [4] was the first case of a data-driven hint generation approach. The objective of this tool was to support the development of a logic-proof tutoring system, which is a system that provides hints to students to help them solve logic proofs.

This technique is present in various tools, such as Deep Thought, a propositional logic tutor for introductory discrete math courses [25], and iList, a tutor for learning linked lists [16]. It is also present in games like BOTS, a serious game to teach basic programming that generalizes this technique to the educational game context [20].

The Hint Factory technique uses many historical student submissions submitted without previous hint feedback to build student models. Then, for each problem, it creates a student model by recording the student's sequence of actions as a state. If the student reaches a state similar to one present in the stored models and the matching state has a successor closer to the goal, the system generates a hint. Thus, the system generates hints based on previous students' past attempts to reach the solution.

The Hint Factory uses Markov Decision Processes (MDPs), a reinforcement learning technique to generate hints automatically. MDPs require a state set S, an action set A, transition probabilities P, and a reward function R. Upon executing an action *a*, the system transitions from the state *s* to state *s'* with probability *P(s' | s, a)* and an expected reward *R(s' | s, a)*.

This method uses the current premises and conclusion as the state and students' inputs as actions. Therefore, a student's set of attempts to solve an exercise constitutes a graph with a sequence of states (each state represents the solution up to that point) connected by actions. Each action represents the student's input that transforms one state into another.

The Hint Factory system combines all graphs of students' solutions into a single graph representing the set of states, which represents all of the paths students have taken to solve the exercise. Then, it applies the reinforcement learning technique to find the optimal solution to the MDP.

This step involves setting specific reward values for each state and action. The Hint Factory applies a large reward for the goal state, penalties for incorrect states, and a cost for each action (to penalize longer solutions). Moreover, it applies a value iteration reinforcement technique using Bellman backup (shown in Equation 3.1) for assigning rewards for each state [51].

$$V(s) = R(s) + \gamma \max_a \sum_{s'} P_a(s,s')V(s') \tag{3.1}$$

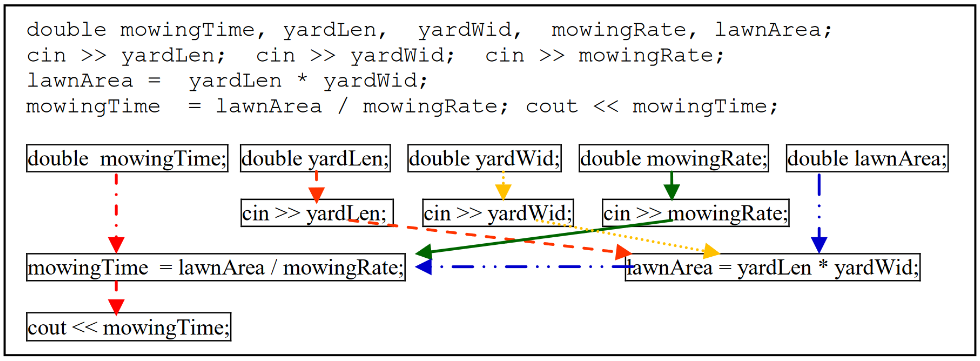

Figure 3.2: Example of a linkage graph for a simple program [28].

Equation 3.1 calculates the values *V(s)* for each state *s*. The value of a state is the sum of the reward for the current state *R(s)* and the discounted value of the next state *V(s')*. The discount factor γ is usually a value between 0 and 1 that determines the importance of future reward. The higher the value, the more important future rewards are. In the case of the Hint Factory, this discount factor is set to 1. Finally, $P_a(s,s')$ is the probability that action *a* will transition from state *s* to state *s'*. The value iteration process stops when there is little change in the function over the entire state space.

Although the Hint Factory generated hints for 91% of students' requests, this method still has some limitations. There are no hints available for students that try solving issues with a unique approach (if there are no similar attempts, the technique cannot infer an optimal path). Similarly, the system cannot generate hints for the first students to solve an exercise since no data is available. Additionally, for most programming languages, the chance that students will have written similar programs can be low in some exercises.

3.6.1.1 Linkage Graphs Representation

The authors of [28] propose a representation for student programs based on linkage graphs to complement the automatic hint generation approach using MDPs. A linkage graph is a directed acyclic graph where nodes are program statements, and directed edges represent the order dependencies.

Figure 3.2 shows an example of a simple program and its respective linkage graph representation. A linkage graph is a directed acyclic graph where nodes are program statements, and directed edges represent the order dependencies. For example, the statement *double yardLen;* appears before the statement *cin » yardLen;* in the program, and both access the same variable *yardLen*. Therefore, the linkage graph adds an edge representing this order dependency from the first to the second statement with the label *yardLen*.

A single trace through the graph that connects statements that modify the same variables is called a linkage. The linkage graph is the set of all linkages combined. This technique chooses to represent the linkage graphs as two-dimensional matrixes, so that equivalent programs have equal matrix representations.

Table 3.1: Two-dimensional matrix representation for the linkage graph in Figure 3.2 [28].

	v0	v1	v2	v3	v4
0. double v0;	•				
1. double v1;		•			
2. double v2;			•		
3. double v3;				•	
4. double v4;					•
5. cin>>v1;		•			
6. cin >> v2;			•		
7. cin >> v3;				•	
8. v4 = v1 * v2;		•	•		•
9. v0 = v4 / v3;	•			•	•
10. cout << v0;	•				

Table 3.1 shows the matrix representation for the linkage graph in Figure 3.2. The matrix has a row for each statement and a column for each anonymized variable. Thus, the columns correspond to the order of the variables, and the rows to the order of the statements. Each filled cell represents a dependency between a statement and a variable. There are no variable dependencies among statements 0 to 4 and 5 to 7 since there are no rows with the filled cells in the same column.

For the hint generation process, this technique starts with creating linkage graphs from the set of previous students' solutions (linking correct and intermediate states).

Creating these linkage graphs requires an additional file provided by the instructor. This file lists the variables for the given programming problem, their types, their name terms (phrases that might compose the variable's name, e.g., *mowing rate*), and how it is assigned. Alternatively, a bag-of-words approach could generate this file from the problem description. Upon analyzing students' programs, the system identifies the variables and assigns them to the ones in the specification file. The system also creates representative variables for the remaining ones, normalizes all names, and sorts statements.

After this step, each complete program is a sequence of states composed into a large graph with respectively mapped equivalent states. The sequence of states represent the steps the student took when writing the program, and the equivalent states represent the respective linkage graphs.

The following step is to assign the required values for the MDP. The reward function is similar to the one used in the Hint Factory [4].

Upon receiving a hint request, the system creates a linkage graph from the student's current program. Then, it searches for the most similar linkage graph, and the MDP selects the next best state. Comparing the two linkage graphs matrixes results in finding the differences between the two, like missing items or statements in a different order. The system then uses these differences to generate the hint.

If the system fails to find a matching state, it tries to modify existing solutions to match the student's program. This process includes adding new rows and columns and splitting ones if needed. The rest of the steps are the same as the ones described above but use the modified linkage graph instead.

The authors did two experiments to test the effectiveness of the system. In the first experiment, the system generated hints for 14 of 16 submissions (87.5%) using 37 correct solutions. In the second experiment, the authors manually selected similar solutions instead of using the best-matching ones. The result, in this case, was 66.6%, but the authors suggest improvements (like detecting variable name reuse) to raise this percentage to 86%.

Unlike an AST, the representation of student programs as linkage graphs captures the data flow of the student's solution. The data flow of student submissions can present the student's problem-solving process, which provides relevant information for hint generation using the MDP approach. Furthermore, clustering these linkage graphs helps aggregate solutions with similar problem-solving approaches. So, given a student's program, it is likely that the system might find a student's solution with the same approach to the problem, which can make the hint generation process more effective.

3.6.2 Path Construction

One of the problems with the Hint Factory approach (presented in Section 3.6.1) is that it can not generate hints for unique student attempts. This problem negatively impacts hint generation for the Python programming language, where there are many ways to write a simple program with the same functionality. This problem was the primary motivation for ITAP, the Intelligent Teaching Assistant for Programming, developed to teach Python programming. The creator of ITAP introduces an algorithm that improves on the idea of the Hint Factory [46].

Like the Hint Factory, ITAP works with the solution space concept, defined as the graph where states are students' submissions, and edges represent edits that transform one state into another. However, instead of using MDPs for hint generation, ITAP uses path construction, an algorithm presented for the first time in [47], which consists of determining what steps to take to go from an incorrect state to a correct solution.

When a student submits an incorrect solution, the first step of the path construction algorithm is to find the optimal goal state by determining the closest and most popular correct solution to the given state (the student's submission). Then, the second stage is to optimize the goal state.

This step involves finding the minimum number of edits to help the student improve their code instead of showcasing every difference between the two states. This process requires optimizations since this search grows exponentially. In the end, either ITAP successfully finds the optimal goal state in a reasonable time or fails to find one. If the latter happens, the algorithm uses the closest goal state (the previously found one) as the optimal one.

The third step of the path construction algorithm is to identify the optimal path between the given state and the optimal goal solution. So, the algorithm generates a power set of all possible valid edits between the given state and the goal, which generates paths between the two states. These paths are then added to the solution space so that they might help future students.

Then, this technique ranks each intermediate state in a path based on a set of properties that computes a state's desirability. This metric identifies the states the student should follow to reach the solution and organizes them in an optimal order (optimal path).

Finally, with the most desirable next-step state identified, the system is ready to generate a hint. The edges between each state in the path indicate the macro-edits to transform one state into another. Thus, ITAP turns these edits into hint messages.

As stated by the author, ITAP became arguably the first approach that could generate hints 100% of the time. So, to evaluate ITAP's performance, the author measured the time it took the tool to generate hints. The used dataset consisted of 41 problems with 11.051 incorrect states in total (with both syntactic and semantic errors).

For individual hint requests, this hint generation process for the first hint request took less than 5 seconds for 94.2% of the states, and only 1.1% took longer than 10 seconds. The author also found that more complex problems tend to take longer to generate hints.

3.6.3 Program Synthesis

The authors of [30] argue that previous data-driven approaches limit the granularity with which individual changes can be tracked. So, instead of using the entire program as a state, they propose a data-driven approach based on program synthesis that models programming in textual edits. Thus, given an incorrect program, it finds the sequence of edits that transform it into a correct solution.

This program synthesizing approach consists of the idea that students' programs are a sequence of lines and that line edits are the basic operations that transform programs. The first step of the algorithm is to identify the most commonly occurring line edits by storing the interactions between students and the system. The interaction history is called trace and consists of a sequence of line edits (a sequence of characters that the student inserted and removed). Then, after applying a canonicalization process, the algorithm starts by counting the number of times each line and line edit appears in the set of all traces. Finally, it calculates the conditional probability of applying a line edit given a line.

$$P(u \to v|u) = \frac{count(u \to v)}{count(u)} \tag{3.2}$$

Equation 3.2 shows how the algorithm calculates the conditional probability of applying a line edit to a line u that transform it into a line v, notated as $u \to v$. The probability is calculated by dividing the number of times the line edit appears in the set of all traces, *count*($u \to v$), by the number of times the line appears in the set of all traces, *count*(u).

With the catalog of line edits and their probabilities, the system uses a best-first search algorithm to find the sequence of line edits that transforms the incorrect program p_0 into a correct solution. The algorithm maintains a priority queue of potential solutions. For each program in the queue, it also stores a score and the sequence of edits to reach it from p_0. The first program in the queue is always p_0 with a score of 1.

Each iteration of the algorithm extracts the program with the highest score from the queue. If the program is correct, the algorithm ends. Otherwise, it searches for edits to apply to the program. The new program that results from these edits is added to the queue. Its score is the product of the score of the parent program and the conditional probability of applying the edit. This score calculation ensures that shorter sequences of common edits are preferred. When the algorithm ends, the tool is able to provide hints based on the sequence of line edits from the correct program.

This approach successfully fixed up to 70% of incorrect programs in under 10 seconds. One advantage of this approach is that it does not require an understanding of the program to fix it. However, it requires the system to store the entire history of interactions between students and the system, which is not currently available in Alloy4Fun.

3.6.4 Mixed-Initiative Program Synthesis

Andrew et. al [19] argue that, although program synthesis techniques can fix student bugs and generate hints automatically, they lack the deep domain knowledge of a teacher. Thus, they often do not address underlying students' misconceptions despite generating functionally correct fixes. Furthermore, these fixes tend to be stylistically poor and potentially misleading. So, the author proposes a mixed-initiative approach that combines the strengths of data-driven program synthesis and the deep domain knowledge of human teachers. This approach allows both the teacher and the program synthesis backend to take turns in the hint-generation process in pursuit of the common goal: providing reusable teacher-written feedback at scale.

The data-driven program synthesis technique learns code transformations from examples of bug fixes. These transformations consist of sequences of rewrite rules that are applied to the AST of the student's program. These transformations allow: (1) clustering the set of incorrect submissions by the type of error they contain; (2) generating bug fixes; (3) propagating teacher-written feedback to all incorrect submissions that the same transformation can fix.

The teacher's role is to review the clusters of erroneous student submissions. After reviewing the clusters, the teacher can write helpful feedback for the cluster. This feedback can include explanations of the error, manually written hints, or references to relevant course material. The whole process is depicted in Figure 3.3.

Sometimes historical submissions might not exist yet (e.g., for new challenges). To address this issue, the program synthesis technique can learn transformations from teachers' fixes instead. This process is shown in Figure 3.4. The first step requires teachers to write feedback and a stylistically correct fix for a single student's submission. Then, the program synthesis technique can learn from each teacher's bug fix and propagate them to other incorrect submissions. Finally, the teacher reviews the propagated fixes by accepting or modifying them for the synthesis tool to relearn from new updates.

The synthesis tool generated fixes for 87% of the submissions in the dataset. Teachers spent 20min on average in each cluster and reported at least one poor fix, further supporting the need for a mixed-initiative approach. Additionally, teachers reported that the tool's clusters and fixes were helpful for an easy and quick review. Propagating a teacher-written fix took 2 seconds and 20

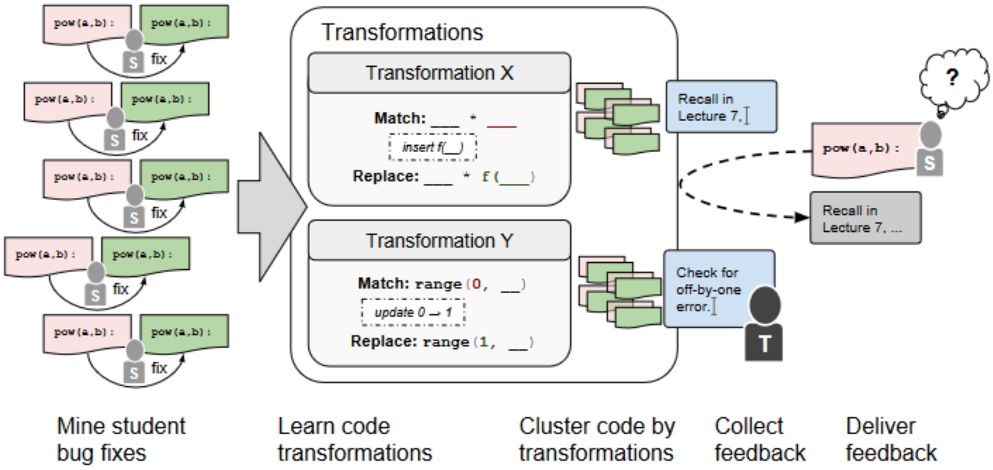

Figure 3.3: Process of learning from students' bug fixes [19].

seconds on average. Teachers reused more propagated fixes rather than feedback, which suggests that feedback does not always generalize to new submissions.

This approach fills its main goal of helping teachers understand the nature and distribution of student errors in introductory programming exercises. However, the need for teachers to write feedback for each cluster of submissions can be a major drawback in some cases. For example, for more complex programming challenges, the diversity of bugs might result in many clusters, which would require a lot of time to review. Finally, the tool's fixes might not always be correct, which can lead to misleading feedback.

3.6.5 Inferring Problem Solving Policies

Typically, a student takes several steps to solve a programming challenge. Thus, it is possible to represent the current student's work as a partial solution $\psi \in S$, where S is the set of all solutions. From the first solution to the last, the student's work steps can be represented as a sequence of partial solutions $T = \{\psi_0, \psi_1, \ldots, \psi_n\}$.

Taking this representation into account, the authors of [44] believe that the first step of the hint generation process is to decide what partial solution an expert would suggest, for any student, they transition to next. To solve this problem, the authors propose learning a Problem Solving Policy (PSP). A PSP is defined as the policy $\pi : S \to S$ (where S is the set of all partial solutions) that specifies a partial-solution $s' = \pi(s)$ for all $s \in S$.

The authors argue that uncommon partial solutions tend to reflect mistakes made by students. These students are less likely than an average student to follow the same transitions as an expert. Simply put, the popularity of transitions captures the next step that a biased population of students would take. Thus, if the authors created a policy based on the student's transitions, they would be trusting a group of students that make mistakes. Hence, to avoid this issue, they base their policies on the number of times a partial solution is submitted because a popular partial solution suggests that the average student found the answer useful at a certain point.

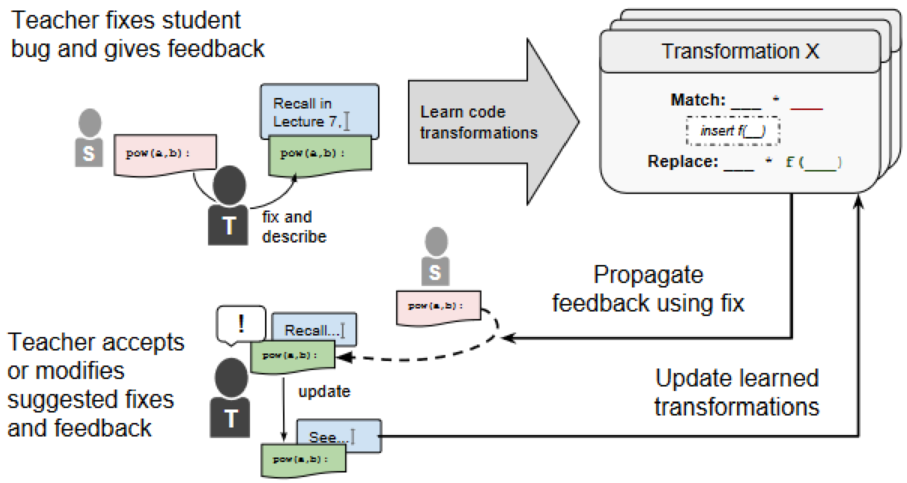

Figure 3.4: Process of learning transformations from teachers' fixes [19].

A simple PSP based on this property would be to, for any partial solution, choose the next partial solution with the highest popularity. However, since the most popular solution is the empty program, it would often encourage the student to delete their progress. To avoid this issue, the authors try to find a balance between visiting popular solutions and making progress by searching for the optimal desirable path to the solution.

The authors present two algorithms based on these assumptions. The first one is called the Poisson Path, and it aims to find the path that takes the shortest time to be generated by the average student. This algorithm assumes that the time it takes for an average student to generate a partial solution is a Poisson process. Moreover, the popularity of the partial solution can be an estimation for the Poisson rate parameter.

$$\gamma(s) = \underset{p \in Z(s)}{arg\,min} \sum_{x \in p} \frac{1}{\lambda_x} \tag{3.3}$$

The Poisson Path $\gamma(s)$ is shown in Equation 3.3, where the $Z(s)$ is the set of all paths from the current partial solution s to the final solution, and λ_x is the popularity of the partial solution x. To compute this path, the authors suggest using Dijkstra's shortest path algorithm, where each edge weights $1/\lambda_x$. The generated PSP will be the first partial solution on the path $\gamma(s)$.

The second algorithm is called Independent Probable Path and returns the most probable path to the solution that an average student would take. Although there are no transition probabilities between partial solutions for an average student, the authors have the partial solution count, which is proportional to the probability of an average student submitting a partial solution. So, with this information, this algorithm calculates the Independent Probable Path using Equation 3.4. $Z(s)$ is the set of all paths from the current partial solution s to the final solution, λ_x is the number of submissions of the partial solution x, and k is the total number of submissions. This calculation can also be done using Dijkstra's shortest path.

$$\gamma(s) = \underset{p \in Z(s)}{arg\,max} \prod_{x \in p} p(\psi_t = x) = \underset{p \in Z(s)}{arg\,min} \sum_{x \in p} -\log(\frac{\lambda_x}{k}) \tag{3.4}$$

The authors tested the generated PSPs using two different complexity problems with an expert map indicating what experts believe to be the next best solution for all partial solutions. Additionally, they compared the two previous algorithms with previously proposed algorithms (e.g., the MDP from the Hint Factory [4] and ITAP [46]) and vanilla baseline algorithms (e.g., the Most Common Path).

The results demonstrated that, for both complexity problems, the Poisson Path algorithm exhibited superior performance. Its accuracy was determined by evaluating how frequently it accurately predicted the next best solution. Specifically, the Poisson Path algorithm achieved an impressive accuracy rate of 95.9% and 84.6% for the respective problems. The Independent Probable Path algorithm came in second with accuracies of 95.5% and 83.3%. The MDPs achieved less effective results with 80.5% and 49.0% accuracy. Additionally, although the ITAP approach got good results in the second problem (78.2%), it often suggested that the student undo progress.

3.7 Comparing Student Data

Most of the previously described approaches involving student historical data imply the comparison of students' programs with each other. This section provides two techniques that might auxiliary this task: canonicalization and AST comparison algorithms.

3.7.1 Canonicalization

The system should not work with the raw student's code for the hint-generation process. Instead, it should determine the necessary content and abstract the data as required [46]. Program representations come in different forms, such as text, AST, Control Flow Graph (CFG), Data Flow Graph (DFG), Linkage Graph, and so on.

An AST is a common representation of a program. It contains purely syntactic information, such as its structure, the types of variables, and the types of expressions. It also can suffer transformations without harming the semantics of the program. These transformations are called canonicalizations [46].

Canonicalization is the process of transforming a program into a canonical form that is independent of the original program's syntax and semantics. Kelly [46] presents several canonicalization techniques for minimizing irrelevant differences between ASTs. The system runs these until the ASTs stop changing. However, these canonicalizations might cause the loss of information required for creating the hint message. To prevent this issue, the author pairs each canonicalization with a reification technique to undo any changes.

The first canonicalization presented by the author is the process of variable anonymization. This process consists of replacing all names with unique default names. The system maintains a dictionary of variable names and their corresponding anonymization to assist the process. This dictionary maps names that should remain unchanged (e.g., built-in function names, imported module names) to themselves and maps all other names to a unique anonymization. The additional needed information is stored as metadata.

The second canonicalization is the simplification of the AST. This process implies simplifying operations that are in one expression into multiple statements. For example, the expression $a = b = c$ is separated into $b = c$ and $a = b$. Other simplifications include traditional compiler optimizations like function inlining [24], constant folding [56], copy propagation [1], and dead code removal.

Function inlining consists of replacing a function call with the function's body. This recursive process ends when the function body does not contain more function calls and is useful for reducing the number of unnecessary helper functions introduced by students.

On the other hand, constant folding replaces constant expressions with their value instead of waiting for runtime. Next, copy propagation propagates the values of the variables into the code by keeping track of variables that have not changed since their assignment and replacing them with their value. Finally, dead code removal removes code that is not used by the program, like comments.

Another canonicalization is ordering the AST to standardize the order of commutative operations. In the case of ITAP, it imposes an order on Boolean and binary operations, comparisons, and conditionals that makes it easier to compare ASTs.

Lastly, the author presents a set of canonicalizations that target novice code oddities. Although the presented canonicalizations are specific to the Python language, it is important to note that the author created these by analyzing inefficient student code. Thus, it is possible to create a set of canonicalizations for other languages that might be useful to target specific domains or common novice student oddities specific to that language.

The author gathered datasets of several problems to evaluate the canonicalizations. In these datasets, the average of parseable submissions was 263, and, from these submissions, 157.63 (35%) had a unique AST. The reductions of unique ASTs were rather low, reaching 8% for anonymization and 9% for other canonicalizations. However, the author argues that canonicalizations should achieve higher reduction percentages when applied to more complex problems.

3.7.2 The Tree Edit Distance Measure

Several hint-generation approaches described in this chapter involve finding the most similar submissions to the current student's submissions or the differences between different submissions. For systems representing submissions by their ASTs representation, it could benefit from AST comparison algorithms.

One way to measure the similarity of tree-structured data (like ASTs) is to use the Tree Edit Distance (TED) measure [3]. This measure represents the minimum-cost sequence of node edit operations that transform one tree into another. Three types of operations are considered for labeled ordered trees: (1) deleting a node and connecting its children to its parent, (2) inserting a node between a node and its children, and (3) renaming the label of a node.

Algorithms for calculating the TED assign a cost to each operation. Thus, the cost of an edit sequence is the sum of the costs of the individual operations. In the end, the TED returned by the algorithm is the minimum cost of all possible edit sequences.

The recursive solution for calculating the TED decomposes trees into subtrees and subforests. The algorithm decomposes the forests into smaller problems at each step by deleting the forests' leftmost or rightmost root node. Then, the distance between the two forests is the minimum distance for the smaller problems. The number of subproblems depends on the choice between left and right at each step. The TED algorithms try to minimize the number of subproblems.

Path decompositions can determine the choice between the left and right solutions. This process decomposes root-leaf paths into subtrees and subforests that compose subproblems. Again, for each subproblem containing two subtrees, there must be a path strategy that determines what path to follow for path decomposition.

The Robust Algorithm for the Tree Edit Distance (RTED) [40] computes the optimal path strategy and runs it with the general TED algorithm. RTED was the standard for computing the TED since it performed better than its competitors and was independent of the tree shape. Unfortunately, RTED requires quadratic memory, up to twice the memory of its competitors.

For this reason, RTED was discontinued and replaced by All Path Tree Edit Distance (APTED) [42, 41]. APTED runs as fast as RTED and has better memory efficiency as it releases memory earlier in the first step of the algorithm. It is the state-of-the-art TED algorithm at the time of writing.

3.7.3 Edit Scripts Between ASTs

One must find an edit script to obtain the node edit operations that transform one AST into another. An edit script is a sequence of edit operations at a node level that, when applied sequentially to the first AST, results in the second AST. [34] In the context of hint generation finding the edit script between the student's submission and the correct solution allows the generation of hint messages based on what the student needs to change in their submission.

The algorithms for finding edit scripts always involve two steps. The first step is to find the pairs of mappings between the nodes of the two ASTs. Mappings are pairs of nodes (one node from each AST) considered equivalent. These mappings are valid if each node belongs only to one mapping and if each pair involves nodes with identical labels. Then, the second step is to derive an edit script from these mappings. The edit script heavily depends on the quality of the mappings from the previous step and can be generated from valid mappings in polynomial time. The main challenge is to find a suitable maximal valid mapping between two ASTs. [34]

There can be many edit scripts for the same transformation. However, the quality of the edit script depends on its length and the number of operations it contains. Thus, the state-of-the-art algorithms for finding edit scripts aim to minimize the number of operations or total cost of the edit script. Moreover, some also consider different sets of operations for the edit script. Some consider only the update, add, and delete operations, which allows them to compute the optimal edit script in polynomial time. Meanwhile, others also consider the move operation. The move operation consists of moving a node from one position to another but can be decomposed into a delete and an add operation. Unfortunately, finding the shortest edit script is NP-hard when considering move operations. So, the algorithms that consider the moves use heuristics to approximate the optimal solution. [34]

APTED and GumTree [14] are two algorithms that can compute the mappings between the nodes of two ASTs. APTED only considers the update, add, and delete operations while preserving node ancestor relationships. It also allows for assigning different costs to each operation and has a worst-case time complexity of $O(n^3)$. GumTree, on the other hand, considers the move operation and takes advantage of the work of Chawathe et al. [10] to generate the edit script.

The Chawathe et al. algorithm is an optimal quadratic algorithm that computes edit scripts containing move actions between two sequential versions of hierarchical data. Meanwhile, the GumTree algorithm finds differences between two versions of the same code at the AST level and builds on the set of edits and edit script generation from the Chawathe et al. algorithm. The GumTree algorithm computes mappings by considering subtree operations and simplifies the edit script generated by the Chawathe et al. algorithm.

3.8 Conclusion

In this chapter, we looked into the state-of-the-art of hint generation techniques. From these approaches, we can conclude that there is no universal approach for generating next-step hints. Each approach has its strengths and weaknesses and might be more suitable for some domains than others. For example, in the case of techniques using students' historical data, it is essential to analyze the available data to understand students' submissions, like the number of unique submissions, the number of correct submissions, the number of unique paths, the types of transitions, etc. This analysis will help capture the most important characteristics of the data for determining the most suitable hint-generation technique.

Additionally, we concluded that the higher-quality hints should be similar to those given by a teacher or an expert. The idea behind this conclusion is that teachers have a more profound knowledge of the domain necessary to address students' misconceptions. However, using student historical data, the generated hints reflect the students' knowledge instead of the knowledge from a tutor. Therefore, the generated hints end up being stylistically poor or potentially misleading [19].

The authors of [19] mentioned this problem and suggested the Mixed-Initiative Program Synthesis approach as a solution. The generated hints with this solution contain the knowledge provided by teachers. Unfortunately, this approach requires teacher effort, which might not always be available. Also, we did not find more proposals to solve this problem. So, hint-generation techniques should focus on using data from students that tend to make fewer mistakes to avoid misleading students. An example of techniques that try to achieve this is presented in Section 3.6.5.

In the next chapter, we present our solution to the problem of hint generation for Alloy4Fun. This solution is based on some ideas and techniques presented in this chapter.

Chapter 4

Data-Driven Hint Generation for Alloy

4.1 Introduction

This chapter presents HiGenA, the Hint Generator for Alloy. It comprises the following main sections: (1) the solution and (2) the implementation. The solution section describes the proposed solution, HiGenA, and how the hint generation algorithm works. The implementation section contains the details of the implementation and integration of this tool in Alloy4Fun, including the technologies used and the different stages of the implementation. In the end, we have a conclusion section that summarizes this chapter's key aspects.

4.2 The Solution Overview

The proposed solution is called HiGenA, the Hint Generator for Alloy. HiGenA allows the automatic generation of history-based hints for Alloy4Fun. It is a data-driven solution that analyzes how students have approached challenges in the past and leverages this knowledge to understand how students from the present should advance toward a solution. This knowledge is presented to the students as hints, small pieces of information that guide them through the resolution process.

To achieve this, HiGenA works with the historical data of student submissions from Alloy4Fun. It analyzes the students' submissions and creates a graph that captures the students' progress when solving a challenge. The graph is then used to generate hints for future students. This idea of working with a graph of student submissions is inspired by the works presented in Section 3.6.

The following items describe the several aspects of HiGenA's hint generation algorithm:

- Graph Structure: The structure of the graph expected by the hint generation algorithm. The creation of this graph is done offline, and can be rebuilt from time to time as new data is collected. The rest of the stages make part of the hint generation algorithm, which rans on demand when a student requests a hint.

- Path Finding: This stage is crucial for the hint generation algorithm as it is responsible for finding the optimal path that leads the student toward a solution.
- Path Construction: When the algorithm cannot find a path to a solution, HiGenA adds new edges to the graph to connect the student's submission to a solution. This step is also necessary when the path found by the algorithm is too long.
- Hint Message Generation: This is the final stage of the hint generation algorithm. It is responsible for generating the hint message that is presented to the student. The message is based on the first edge of the path found by the algorithm.

The following sections describe each one of these stages in detail.

4.2.1 Graph Structure

As mentioned before, HiGenA requires a graph of student submissions with a specific structure to generate hints successfully. In these graphs, each node represents a student's submission labeled as correct or incorrect, and each edge represents a transition between two submissions.

The direction of the edge indicates the order of the submissions. As explained in Section 2.2.6, Alloy4Fun builds a derivation tree of all models developed after the sharing of a model. For instance, if the derivation tree contains *submission 1* followed by *submission 2*, the graph will contain a directed edge from *submission 1* to *submission 2*.

Moreover, each submission should be unique in the graph, meaning that no two nodes represent the same submission. Finally, both transitions and nodes must contain their respective popularity values. The popularity of a transition represents how often students went through this transition when solving a challenge. The popularity of a submission represents the frequency with which a student went through a specific submission when solving a challenge.

Finally, all submissions in the graph should be related to the same exercise. Otherwise, it would become difficult to analyze the graph and extract useful information from it if it was a mix of submissions from different exercises. As stated in Section 2.2.2, Alloy4Fun challenges contain multiple exercises. In this context, an exercise is an empty public predicate that students must complete. Thus, it is important that we filter the data to only consider submissions related to a single predicate before creating the graph. In the end, there should be one graph per public predicate in the challenge.

An example of a graph created by HiGenA is shown in Figure 4.1. Green nodes represent correct submissions, while red nodes represent incorrect submissions.

One of the key aspects of the hint generation algorithm that HiGenA employs is comparing student submissions with each other. However, comparing submissions in their textual form is difficult and ineffective. Thus, HiGenA compares submissions by their ASTs. Therefore, each node in an HiGenA graph contains the AST of the student's submission it represents for identification and comparison purposes.

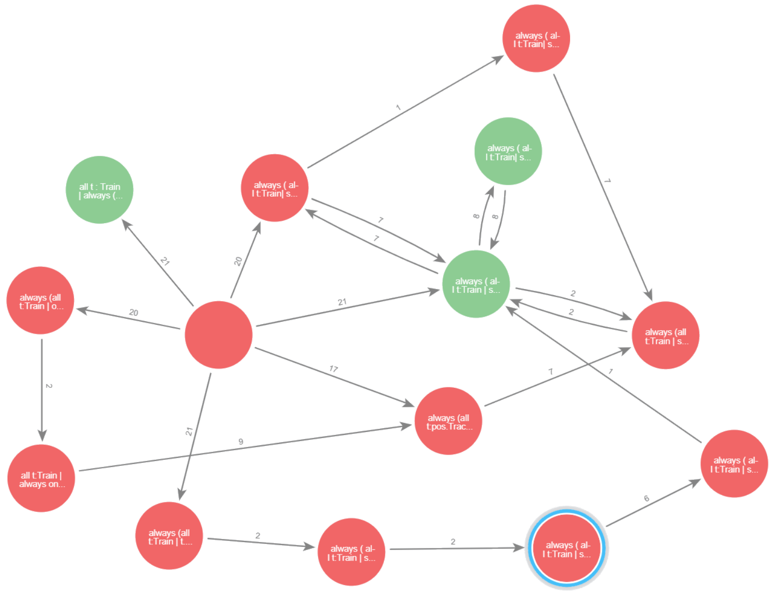

Figure 4.1: Example of a graph created by HiGenA.

The Alloy4Fun platform does not store the ASTs of the student's submissions. Instead, it stores the student's model in its textual form. Thus, HiGenA needs to parse the student's model and obtain the AST of the student's submission.

Each one of the ASTs in the graph is created by parsing the student's submission and obtaining the canonicalized version of the AST by applying canonicalization techniques. By doing this, we ensure that insignificant differences between submissions, like variable names, do not affect the comparison between ASTs.

Section 3.7.2 shows some insights into how ASTs can be compared by presenting the TED metric, which measures the minimum edit distance between two ASTs. For instance, a TED of 0 indicates that the two ASTs are identical, while a TED of n indicates that the two ASTs are n edit operations away from each other.

An edit operation is defined as an operation that transforms one AST. Examples of edit operations include inserting a node, deleting a node, etc. A sequence of edit operations that when applied to one AST transforms it into another is called an edit script. As described in Section 3.7.3, there are some algorithms that can extract the edit script from two ASTs. These algorithms can consider different types of edit operations and generate edit scripts of various sizes.

Figure 4.2 shows an example of two Alloy expressions, and their corresponding ASTs. These two expressions were submitted by students trying to solve the same challenge in Alloy4Fun. The challenge is to model a simple file system trash can.

Note that to transform the first AST into the second, we need two edit operations: (1) delete the node **`always`** and (2) update node `Trash` to `File`. Thus, the TED between these two ASTs is two. Also, note that the edit script in the figure encompasses the two edit operations in order to successfully transform the first AST into the second.

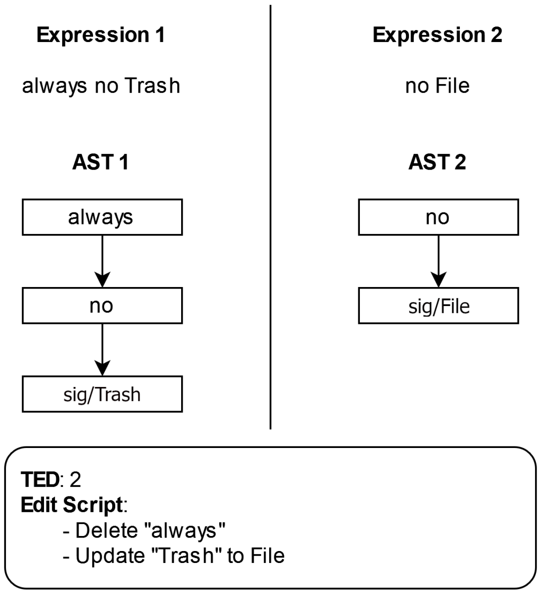

Figure 4.2: Example of TED and edit scripts.

In the final graph, each directed edge contains the TED between the two submissions it connects, plus the edit script that transforms the first submission into the second. The hint generation algorithm then uses this information to determine the next step a student should take to progress toward a solution.

The hint generation algorithm is described in detail in the next subsections.

4.2.2 Path Finding

The first step of HiGenA's hint generation algorithm is pathfinding. In this step, when a student requests a hint, HiGenA starts by trying to locate the student's current submission in the graph. Each submission is unique in the graph, so if the student's current submission is present, HiGenA will successfully find the node that represents it. If it is not present, HiGenA creates a new node for the student's current submission and adds it to the graph and connects it to another node using the path construction technique described in the next section.

After locating the student's current submission in the graph, HiGenA aims to find the optimal path from the student's current submission to a solution. The first edge of this path indicates the transition the student should make to progress toward the solution, serving as content for generating the hint.

As discussed in Chapter 3, an alternative approach to searching for the optimal path is to search for the next best state instead. However, as discussed in Section 3.6.5, searching for an optimal path for hint generation leads to better results than searching for the next best state. This approach guarantees that the algorithm will be guiding the student toward a solution.

HiGenA considers several options for finding the optimal path. In the first option, HiGenA considers the optimal path to be the path with the lowest total TED. This solution translates to a shortest path problem, where the edge weights are the TED of each edge. In other words, HiGenA aims to find the path that requires the fewest number of edit operations to reach a solution. consequently, the hint generation algorithm will suggest the student to reach the most similar solution to their current submission.

This solution is similar to the one proposed by ITAP (presented in Section 3.6.2), which also generates hints by finding the path to a solution that requires the minimum number of edits. However, unlike HiGenA, ITAP starts by finding the optimal goal state, which is the closest solution to the student's current submission, even if no path exists between them. This could be done, for instance, by calculating the TED between the student's current submission and each solution and then selecting the one with the lowest TED. Then, ITAP constructs the optimal path, if it does not exist already.

HiGenA's algorithm differs from ITAP's as it only relies on existing paths in the graph, unless no path to a solution exists (as discussed in the following subsection). The objective of this approach is to take advantage of the students' attempts stored in the graph as much as possible. This way, students will go through the same steps as their peers, which are the steps that past students took during their learning process. The idea is to allow the students to go through the same learning process as their peers, but with the help of hints to keep them motivated and engaged to avoid frustration when they have trouble solving a challenge.

Another alternative for defining the optimal path is to consider the popularity of each submission or transition, as discussed in Section 3.6.5. HiGenA considers two different paths based on popularity: the path that traverses the most popular transitions and the path that traverses the most popular submissions. These two path definitions are inspired by the Poisson Path presented in Section 3.6.5. The intuition behind these two approaches is that uncommon partial submissions reflect mistakes and should be avoided, while popular ones reflect the most valuable submissions for students.

After finding the optimal path using one of the options mentioned above, HiGenA evaluates the path to determine if it is a good candidate for hint generation. This is required to handle trial-and-error attempts, where consecutive submissions do not progress toward a solution. In these cases, the formed path is not ideal for hint generation as it does not contain the sequence of steps to reach a correct solution.

These types of paths are problematic since the hints generated would encourage the student to make mistakes instead of getting close to a solution. Figure 4.3 shows an example of a problematic path for hint generation. In this example, the current submission is **`no`** `Trash`, and the solution is **`no`** `Trash` **`and no`** `Protected`. However, students in the past have submitted **`always no`** `Trash` before reaching the solution, creating a problematic path. This path is problematic because, upon a hint request for the current submission, the hint generation algorithm would suggest the student to add **`always`** to the current submission, which is incorrect.

We need to identify and avoid these types of paths to deal with these cases. However, to do so, we need to define some criteria to determine if a path is problematic. In a good path, each state gets increasingly more similar to the final submission. In other words, the TED between the

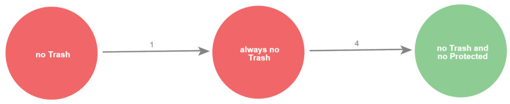

Figure 4.3: Example of a problematic path for hint generation.

first submission and the solution should be equal to the sum of the TEDs of all edges in the path starting from that state.

Therefore, a good path for hint generation is equivalent to a direct path from the current state to the closest correct state since both paths should have the same total TED. Therefore, we can evaluate the path by checking if the total TED of the path equals the TED between the current and correct states. If the total TED is equal, then this path is not problematic and we proceed to the message generation step. Otherwise, we resort to the path construction technique again to create a path connecting the current submission to the closest solution directly, like in the case of a new submission in the graph.

Going back to the example in Figure 4.3, the total TED of the problematic path is higher than the TED between the current submission and the solution (5 vs. 3). Therefore, the hint generation algorithm would avoid this path and create another one connecting the current submission to the closest solution with a total TED of 3.

4.2.3 Path Construction

What if there is no path to a solution? HiGenA is a data-driven approach where the resulting final graph structure is completely dependent on the data used to construct it. The final result is a graph that contains the various ways in which students approached past challenges. It encompasses single-attempt solutions, multiple attempts, and unsuccessful attempts. These collective attempts serve as a foundation for the hint generation algorithm, even the unsuccessful ones. These attempts result in two types of branches where the leaf node is either a solution (the student solved the challenge) or an incorrect submission (the student failed to solve the challenge).

As seen in the previous subsection, if the student's current submission is located in a branch leading to a solution, the hint generation algorithm successfully finds a path and generates a hint. However, if no path exists, the hint generation algorithm employs a strategy to create a path when one is absent. This strategy is inspired by ITAP, outlined in Section 3.6.2.

In ITAP, the path construction algorithm creates both transitions and nodes, as each transition only encapsulates a single edit operation. However, in HiGenA, each transition encapsulates multiple edit operations, eliminating the need for intermediate states. Thus, when no path to a solution exists, HiGenA starts by finding the closest solution by calculating the TED between the student's current submission and each solution. If there are multiple solutions with the minimum TED, HiGenA selects the one with the highest popularity. Then, HiGenA connects the two submissions with a transition that encapsulates the TED and edit the script between them.

This strategy allows HiGenA to take advantage of all existing data in the graph, even the unsuccessful attempts. In a way, creating new paths is equivalent to creating a new hypothetical scenario where the student from the past that tried to solve the challenge in this way did not give up and successfully reached a solution.

As mentioned previously, this strategy is also employed when the student's current submission is new to the graph. After creating a new node, HiGenA also connects it to the closest solution using the same strategy. Additionally, it is also employed when the path finding algorithm has returned a path that is problematic for hint generation.

Finally, it is important to note that HiGenA always requires at least one solution to be present in the graph. Otherwise, it is not possible to create or generate paths. This limitation is inherent to HiGenA but can easily be addressed by populating the graph with at least one solution. This step can easily be done by the instructor when creating the challenge, as they can attempt to solve it themselves and add their solution to the graph. Thus, as long as there is at least one solution in the graph, HiGenA can generate hints 100% of the time, due to the path construction strategy described above.

4.2.4 Hint Message Generation

Upon obtaining the optimal path, the final step is to generate a hint message that will help the student make progress toward a solution, by following the first transition of the path and reaching the next state. This implies that the resulting message should hint the student to make one of the edit operations present in the first transition of the path.

The hint message generated by HiGenA is a small piece of text composed of two parts. The first part gives the student an idea of how far they are from the solution, and it is based on the TED between the student's current submission and the solution. An example of a hint message for a submission with a TED of 1 from a solution is *"You are just one step away from the solution!"*. This message will inform the student about the number of changes they need to make to reach a solution.

The second part of a hint is based on the edit operations present in the first transitions of the path. In each HiGenA graph, each transition contains an edit script with several edit operations. One could try to concise all edit operations into a single hint message. However, these lists of operations vary in size and complexity - some are simple and short, while others are long and complex. Generating hints involving several operations can confuse the student or result in a long and complex hint message that has too much information. Thus, we chose to generate hints based on a single operation since it will encourage the student to take a single step towards the solution (decrease the TED by 1). This approach is also employed by ITAP, as described in Section 3.6.2.

When the edit script contains a single operation, the hint message is based on the unique operation. However, in the case when the edit script contains many operations, HiGenA has to decide which one to use to apply to the hint message. The operation could be chosen randomly, or we could prioritize some operations over others. However, HiGenA chooses always the first operation of the edit script, as the script is a sequence of operations that must be applied in the specified order to reach the desired AST.

As stated in Section 3.7.3, there can be several types of edit operations. In HiGenA, we consider four: addition, which adds a new node to the AST; deletion, which removes a node from the AST; update, which updates the label of a node to some value; and move, which moves a node from one place to another in the AST.

The final step is to translate an edit operation to a hint. In HiGenA, translating an edit operation to a hint is done by using a message template for each operation type, similar to what is done in ITAP (presented in Section 3.6.2). The message template contains placeholders for operation-specific information and can be customized to fit the Alloy language domain.

Figure 4.2 contains two different submissions. If the left one is the student's current submission, and the right one is the solution, HiGenA will generate the following hint message: *"You are almost at the solution. It seems like you have unnecessary information in your expression. Try simplifying your expression by deleting the* **`always`** *temporal operator."*.

The first part of the message indicates that the student is close to the solution since the TED between the two submissions is small (2). The rest of the message is based on the edit script of the first transition of the path. The edit script contains two edit operations: one deletion and one update. Since the deletion operation appears first in the edit script, HiGenA uses the deletion operation to generate the hint message. The message template for a deletion operation is completed with the **`always`** keyword. HiGenA also identifies this keyword as a temporal operator, which is also included in the final message to help students understand their mistakes.

4.3 HiGenA Implementation

The implementation of HiGenA follows the steps of the CRoss Industry Standard Process for Data Mining (CRISP-DM), a well-known data mining process model. Figure 4.4 depicts its main steps: (1) data understanding, (2) data preparation, (3) modeling, (4) evaluation, and (5) deployment. These steps are not necessarily sequential and can be repeated multiple times [36].

The following subsections describe each step in the same order as shown in the figure. Exceptionally, the evaluation step is presented separately in Chapter 5.

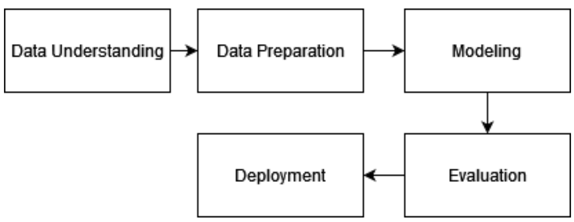

Figure 4.4: CRISP-DM data mining process [36].

Table 4.1: Dataset overview [32].

ID	Challenge	Predicates	Entries
sDLK7uBCbgZon3znd	Trash FOL	10	4092
YH3ANm7Y5Qe5dSYem	Classroom FOL	15	5893
PQAJE67kz8w5NWJuM	Trash RL	10	4361
zRAn69AocpkmxXZnW	Classroom RL	15	6341
gAeD3MTGCCv8YNTaK	Graphs	8	3211
zoEADeCW2b2suJB2k	LTS	7	3382
jyS8Bmceejj9pLbTW	Production line	4	898
bNCCf9FMRZoxqobfX	Production line (v2)	10	4903
aTwuoJgesSd8hXXEP	Production line (v3)	10	3175
JC8Tij8o8GZb99gEJ	CV	4	1199
WGdhwKZnCu7aKhXq9	CV (v2)	4	393
9jPK8KBWzjFmBx4Hb	Trash LTL	20	5279
FwCGymHmbqcziisH5	Train Station	18	1264
QxGnrFQnXPGh2Lh8C	Train Station (v2)	15	8158
PSqwzYAfW9dFAa9im	Courses	15	14884
JDKw8yJZF5fiP3jv3	Courses (v2)	15	7632
dkZH6HJNQNLLDX6Aj	Social Network	8	22690

4.3.1 Data Understanding

Before starting the data mining process, it is crucial to understand the data we are working with. This step involves studying the dataset's structure, characteristics, and quality. This step is known as data understanding.

The dataset for this project is a collection of student submissions from the Alloy4Fun platform between 2019 and 2023, described in Section 2.2.6. In reality, the dataset is a collection of inner datasets, one for each challenge available in the platform.

Table 4.1 shows an overview of each dataset. There are 11 challenges in total, each with a different number of predicates and submissions. The "Social Network" challenge has the most submissions, with 22690 submissions. On the other hand, the "CV (v2)" challenge has the least submissions, with 393 submissions. In total, the dataset has 97755 submissions. The number of predicates indicates the number of predicate bodies the students must fill in to solve a challenge. The maximum number of predicates found is 20, while the minimum is 4. The total number of predicates is 188.

Each dataset has 11 features in total. Table 4.2 shows the name of each feature, its description, and its type. Notice that some features might include missing values when applicable. For example, the `theme` feature is only available for sharing entries.

The *_id* feature serves as a mandatory unique identifier for each submission, represented as a string of constant length. Both the *derivationOf* and *original* features act as foreign keys, referencing other submissions. In this context, if the value of the *derivationOf* feature is equal to x, it indicates that the submission is a child of the submission with an *_id* equal to x. Similarly, if the value of the *original* feature is equal to a, it signifies that the submission is a descendant of the submission with an *_id* equal to *a*. It is important to note that while the *original* value can refer to the submission itself, the *derivationOf* value cannot.

The *time* feature is the timestamp of the submission's creation. The format of this feature is a string in the format "YYYY-MM-DD HH:MM:SS" or "MM/DD/YYYY, HH:MM:SS AM/PM". On the other hand, the *code* feature is a string that contains the model of the submission with comments. In the case of the original submission, the *code* feature contains the model with the secrets.

The *sat* feature contains a value representing the result of the executed command. When the value is -1, it means that the command failed to execute (e.g., caused by the presence of a syntax error). Otherwise, the value can be either 0 or 1, which represents a successful execution. The value 0 represents that no counter-example was found, the value 1 represents that a counter-example was found. If the command executed is one of the check commands of the challenge and not created by the user, the value of the *sat* feature indicates whether the submission is correct (0) or incorrect (1).

Figure 4.5 shows a histogram based on the distribution of the `sat` feature, which represents the correctness of a submission. The histogram shows that the majority of submissions are incorrect (40036 submissions), followed by correct submissions (28294 submissions), and finally, failed executions (28066 submissions). The rest of the submissions are non-executions (1359 submissions).

The *cmd_i* feature is also only available for executions and contains the index of the executed command. Meanwhile, the features of the *cmd_n* and *cmd_c* only apply to successful executions and contain the name of the executed command and a boolean value indicating whether this command was a check, respectively.

Lastly, the *msg* feature contains the error or warning message, if any, while the *theme* feature contains the visualization theme. Note that the *theme* feature is only available for shared entries (see Section 2.2.3 for more details on Alloy4Fun's model sharing feature).

Table 4.2: Dataset features [32].

Feature	**Description**	**Type**
_id	the id of the interaction	string
time	the timestamp of its creation	string
derivationOf	the parent entry	string
original	the first ancestor with secrets	string
code	the complete code of the model	string
sat	command's result. or -1 for errors [only for executions]	number
cmd_i	the index of the executed command [only for executions]	number
cmd_n	the name of the executed command [only for successful executions]	string
cmd_c	whether the command was a check [only for successful executions]	boolean
msg	the error or warning message [if any]	string
theme	the visualisation theme [only for sharing entries]	JSON object

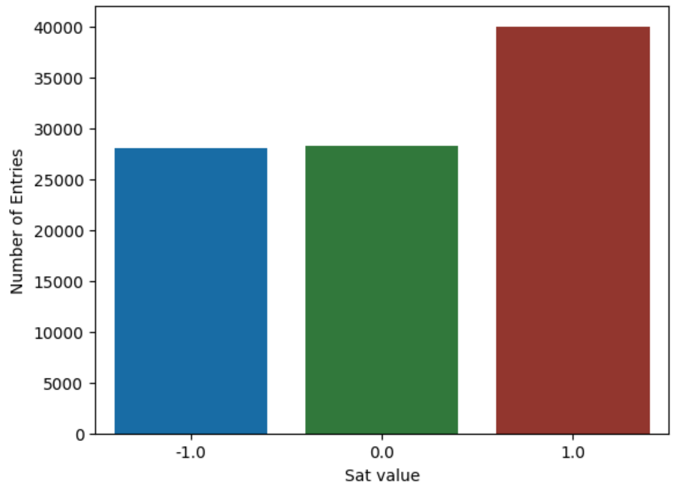

Figure 4.5: Distribution of the correctness of student submissions.

4.3.2 Data Preparation

The data preparation step aims to transform the data into a suitable format for the modeling step. This step also includes data cleaning to remove any noise and unnecessary information from the dataset. The following subsections describe the data preparation process for this project.

4.3.2.1 Original Submissions

The submission in the dataset with the *_id* equal to the challenge's ID contains the original challenge model with hidden secrets. An example of an original challenge model is depicted in Listing 4.1. We refer to these submissions as the original entries since they contain empty predicates for the students to fill in and secrets containing the solution written by the teacher. These submissions are replaced in the final dataset by submissions generated from them.

We generate two types of submissions for each one of the original entries: empty and solution submissions. As stated previously, these submissions contain empty predicate bodies for the students to fill in. In Listing 4.1, there is one empty predicate body, predicate `prop1`. We generate submissions for each empty predicate in original entries and refer to them as empty submissions. These submissions contain the same code as the original entries, the same *_id*, and have a *sat* value of 1, representing an incorrect submission. These submissions are used to help preserve the derivation chain of every student. A derivation chain is a sequence of submissions that are derived from each other. For example, if a student submits a model derived from another student's submission, the two submissions are part of the same derivation chain. The original entries are the root of every derivation chain, as every student attempt starts with an empty predicate to be filled in.

Besides containing empty predicates, the original submissions have secrets with the teacher's solution for each predicate. In Listing 4.1, there is one solution to predicate `prop1`, which is predicate `prop1o`. Therefore, we also generate what we refer to as solution submissions, which

```
var sig File {
  var link : lone File
}
var sig Trash in File {}
var sig Protected in File {}

// initially the trash is empty and there are no protected file
pred prop1 {

}

//SECRET
pred prop1o {
  no Trash+Protected
}
```

Listing 4.1: Original entry example from the Alloy4Fun dataset.

contain the teacher's solution for each empty predicate. Having the teacher's solution is additional data that can be used to generate hints by populating the graph with new solutions. All these solutions have a different *_id* from the original entry and a *sat* value of 0, representing a correct submission. Additionally, we set the *derivationOf* and *original* values of these submissions to the *_id* of the original entry to specify the derivation chain of the submission.

To sum up, we generate an empty submission and a solution submission for every predicate to be filled in by the students for each challenge. So, if a challenge has n predicates, we generate n empty submissions plus n solution submissions containing the teacher's solution for a specific predicate. Then, we remove the original submissions from the dataset and replace them with empty and solution submissions for each predicate. This process generates $2n$ submissions for each challenge, where n is the number of predicates to be completed in the challenge. In total, we generate 376 submissions and remove the original 17 submissions from the dataset, bringing the total number of submissions of the dataset to 98114.

4.3.2.2 Data Cleaning

This section describes the data-cleaning process. The data-cleaning step aims to remove any noise and unnecessary information from the dataset.

Some features like the *cmd_i* feature in the dataset have missing values. As stated in Table 4.2, this feature is only available for executions, so rows with missing values for this feature are non-executions. Non-executions can represent, for example, the sharing of a model. We removed these 1342 submissions because they are irrelevant to the problem of hint generation as they do not represent a student's attempt to solve the challenge.

Other rows have missing values for the *cmd_n* feature. Coincidentally, whenever a row has a missing value for this feature, it also has a missing value for the *cmd_c* feature. Additionally, all these rows have a non-empty *msg* feature. Once again, as stated in Table 4.2, *cmd_n* and *cmd_c* are only available for successful executions. Therefore, rows with missing values for these features are cases where a syntactic error occurred, generating an error message. In conclusion, the code present in these submissions is not syntactically correct. However, HiGenA's hint generation algorithm (described in Section 4.2) requires the AST of each submission, which the Alloy parser fails to obtain in these cases. Therefore, we removed these 28066 rows from the dataset.

There are other rows with a non-empty *msg* feature. However, these messages are warning messages, which do not represent syntactic errors, and therefore the submission can be parsed. We keep these submissions in the dataset.

Another feature that requires analysis is the *cmd_c* boolean feature. However, the number of *false* values is relatively small (46 for the full dataset). These values represent the cases where the command was not a check (see Table 4.2). When the command is not a check, it translates to the student not verifying the correctness of their solution, which does not count as an attempt to solve the challenge. Therefore, we remove these cases as they are irrelevant to the task at hand. We also drop rows where the command was a check created by a student instead of by the author of the challenge.

Regarding the code of the submissions, we remove any comments and empty lines from the code. This helps to reduce the dataset's size and remove noise and unnecessary information from the dataset.

4.3.2.3 Derivation Chains

The *derivationOf* feature is crucial to understand the relationship between submissions and creating the final graph. If we are not careful, we can create holes in the derivation chain when removing submissions from the dataset, leading to a situation where a submission has a parent that is not present in the dataset. This situation makes it impossible to reconstruct the steps taken by the student to solve the challenge. Therefore, the resulting graph might contain isolated subgraphs, making finding paths for hint generation more difficult.

Dealing with this problem requires special care to preserve the derivation chain. So, upon removing a submission from the dataset during the previous stage, we search for its child submission. If it exists, we set the child's parent to the parent of the removed submission. This process is called short-circuiting, illustrated in Figure 4.6. In this example, we remove submission B from the dataset. Submission C is the child of submission B, so we set the parent of submission C to the parent of submission B, which is submission A.

Another situation that causes the loss of information about the derivation chain is splitting the dataset by predicate. As stated in Section 4.2, HiGenA works with graphs with submissions for one predicate only. So, we split each challenge into multiple datasets, one for each predicate. Then, each predicate's dataset is processed independently to create the corresponding graph. However, since the derivation chain is a chain of submissions across all predicates, splitting the dataset creates holes in the chain.

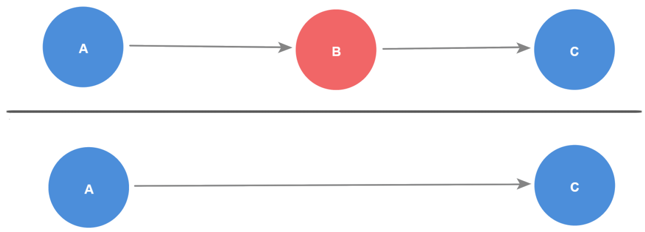

Figure 4.6: Illustration of short-circuiting the derivation chain.

Thus, to preserve the derivation chain after splitting the datasets by predicate, we update the *derivationOf* feature of each submission to point to the nearest previous submission from the same predicate. We iterate the derivation chain backward for each submission until finding one submission that belongs to the same predicate. This submission is set as the new parent of the current submission. If we do not find a submission that belongs to the same predicate, we will eventually reach the original submission of the challenge. In this case, we set the current submission's parent as the empty submission for the predicate.

Figure 4.7 illustrates this process for the dataset of one challenge. In this example, submission O represents the original submission of the challenge and submissions $O1$ and $O2$ represent the empty submissions that were generated from submission O for predicates *Blue* and *Red*, respectively. All blue submissions belong to the *Blue* predicate and all red submissions belong to the *Red* predicate. Simply splitting the dataset by predicate creates holes in the derivation chain.

To solve this problem, we start iterating the derivation chain backward for each submission. The original submission O is replaced by the corresponding empty submission for each predicate ($O1$ for the *Blue* predicate and $O2$ for the *Red* predicate). Submission A belongs to the *Blue* predicate and its parent is the original submission O. So, we set its parent to the empty submission $O1$ (the empty submission for the *Blue* predicate). On the other hand, submission B belongs to the *Red* predicate and its parent belongs to a different predicate (A belongs to the *Blue* predicate). So, we iterate the derivation chain backward until finding the original submission. Then, we set the parent of submission B to the empty submission $O2$ (the empty submission for the *Red* predicate). Submission C belongs to the *Red* and its parent does too, so no update is necessary. Finally, submission D belongs to the *Blue* predicate, but its parent does not. So, we iterate the chain backward until finding either the original submission or a submission that belongs to the same predicate. In this case, we find submission A and set it as the parent of submission D. The final result consists of two derivation chains, one for each predicate, where the starting point is the empty submission for the predicate. Note that the original submission O does not belong to any chain (it is discarded from the dataset).

These two processes make it possible to acquire the derivation chain of students' submissions for each predicate, even if the student jumps arbitrarily from one predicate to another when solving the challenge. Each chain starts with the empty submission for that predicate (an empty predicate) and ends at the last student's attempt to fill the predicate.

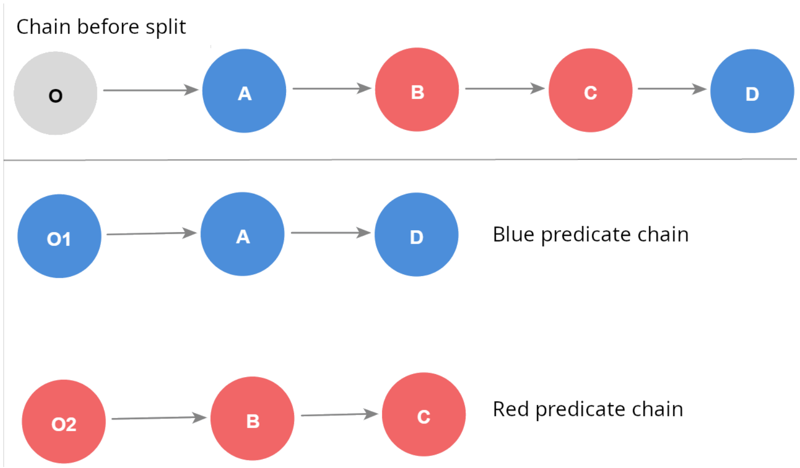

Figure 4.7: Example of handling predicate split.

4.3.2.4 Feature Selection and Engeneering

Not all features available in the dataset are relevant to the problem of hint generation. At this point, we can filter the dataset to include only the relevant features that are required by the hint generation algorithm.

The *time* feature containing the timestamp of a submission and the *theme* feature do not provide useful information for the task at hand and are removed from the dataset. However, the *derivationOf* feature is crucial to understand the relationship between submissions. It gives us the parent of each submission, allowing us to identify student sessions and, consequently, the different steps the student took to solve the challenge. Additionally, this feature allows us to connect submissions in the graph.

We also remove the *msg* feature because it is irrelevant to the hint generation process. Similarly, we discard the *original* feature as its value is equal for every submission in a challenge's dataset. For example, in the dataset for the *c1* challenge, all submissions have a *original* value of $c1$.

Moreover, for each dataset, there is a one-to-one relationship between the *cmd_i* and *cmd_n* as the *cmd_i* feature is the index of the command in the *cmd_n* feature. Therefore, we remove the *cmd_i* feature as it is redundant, and we keep the *cmd_n* feature as it is more descriptive since it contains the command's name.

Duplicate code submissions could also be removed from the dataset. However, removing duplicate code submissions causes the loss of information about student sessions. Without duplicate code submissions we can't calculate the popularity of a solution since we don't know the number of times they appear in the dataset. Identifying the most popular solutions can be useful for discovering what were the most valuable states for students when solving a challenge which is useful for the hint generation algorithm.

On the other hand, the dataset contains empty student submissions that we discard since they do not provide any useful information. These submissions are the result of a student running

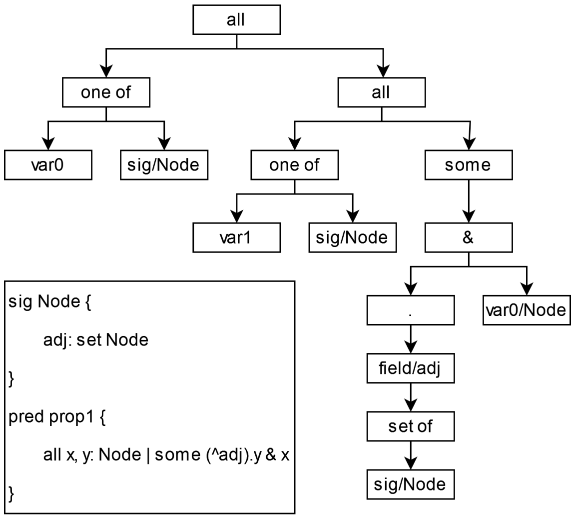

Figure 4.8: Example of an Alloy expression and its corresponding AST.

the challenge without writing anything. However, we do not remove empty submissions which represent each derivation's starting point (described in Section 4.3.2.3), as they are necessary to keep the derivation chains.

Each dataset entry contains the complete code of the Alloy model, but we are only interested in the Alloy expression that the student wrote for a specific predicate and its AST. Thus, we extract the expression from the respective predicate and save it as a new feature in the dataset called *expr*. Then, we parse this expression to obtain its AST using the Alloy parser and save it as a new feature in the dataset called *ast*.

4.3.2.5 Alloy AST Representation

After obtaining the AST of the student's expression, we need to represent it in textual form. We translate each node of the AST into a string representation that can be easily parsed by the hint generation algorithm. At the same time, we transform the AST into its canonical form to reduce the final number of different submissions in the graph.

Figure 4.8 shows an example of an AST for the Alloy expression `all x, y: Node | some (^adj).y & x`. The root node of the AST is the `all` operator. This operator is a quantifier that binds the variables `x` and `y` to the `Node` signature. In the AST representation, the variables declarations are represented separately in quantifier subtrees.

Upon using a variable in an expression, the node's label is the name of the variable followed by the type (*var0/Node*). In the case of fields, the node's label has a prefix *field/* followed by the field name and a child indicating its multiplicity and type (*set of Node*). Signatures have the prefix *sig/* followed by the signature name.

In Section 3.7.1, we described canonicalization techniques that transform ASTs into a canonical form. These techniques are useful for hint generation as they abstract the ASTs and reduce the number of states in the final graph. We applied two canonicalization techniques to the ASTs of the submissions in the dataset: *sorting commutative operations* and *variable anonymization*.

Sorting commutative operations requires identifying the commutative operations in Alloy. We identified the following operations as commutative: `AND`, `OR`, `&&`, `||`, `&`, `=`, `!=`, `<=>`, **`iff`**, `+`. We then sort the operands of these operations in alphabetical order. For example, the expressions `a & b` and `b & a` are equivalent with the same AST representation.

Variable anonymization replaces the variable names in the AST with a generic label. For example, in Figure 4.8, the variable `x` is replaced with `var0` and the variable `y` is replaced with `var1`. The implementation of variable anonymization involves maintaining a mapping between the original variable names and the generic labels. Upon declaring a new variable, we add a new entry to the mapping. When we use a variable in an expression, we replace the variable name with the generic label in the mapping. Again, this technique minimizes the differences between ASTs since the same variables with different names are represented in the same way. For instance, `all a: Node` is equivalent to `all b: Node`.

4.3.2.6 Data Export

By the end of this process, we have a processed dataset of student submissions for each predicate in a challenge. Each dataset is stored in a CSV file with the name of the predicate inside the folder named after the challenge. For example, for a challenge named *Challenge1*, the dataset for the predicate *p1* is stored in the file *Challenge1/p1.csv*.

Each CSV file contains the following features: *_id*, *derivationOf*, *sat*, *expr*, and *ast*. The *_id* feature contains the identifier of the submission. The *derivationOf* feature contains the identifier of the parent submission. The *sat* feature contains a boolean indicating whether the submission is correct or not. The *expr* feature contains the student's expression. Finally, the *ast* feature contains the AST of the student's expression.

In total, we have 66426 submissions, where 28451 are correct, and 37975 are incorrect. This distribution can be seen in Figure 4.9. Notice that there are no submissions with a *sat* value equal to -1, as we discarded all submissions that could not be parsed.

4.3.3 Creating the Graph

The next steps require creating, storing, modifying, and querying graphs. The tools of choice for this task are graph databases. Graph databases are a type of NoSQL database that uses graph structures to store data where nodes represent entities and edges represent relationships between entities. In this work, we use Neo4j[1] as our graph database and the Neo4j Java Driver[2] for connecting to the database and executing queries. In Neo4j, queries are written in the Cypher query language[3].

[1] https://neo4j.com/
[2] https://neo4j.com/docs/getting-started/languages-guides/java/java-intro/
[3] https://neo4j.com/docs/getting-started/cypher-intro/

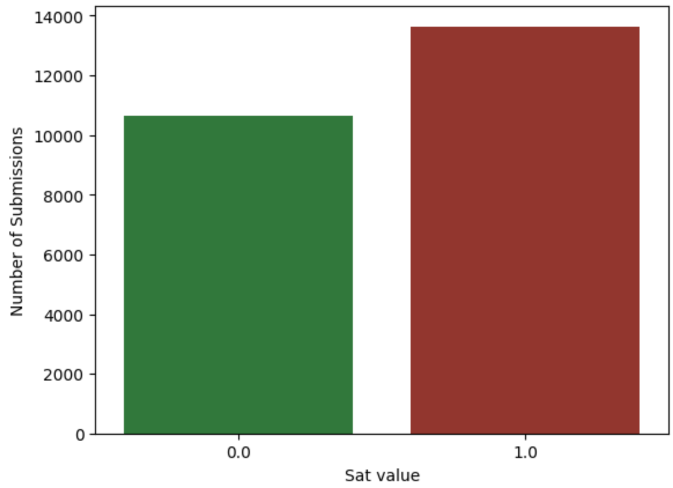

Figure 4.9: Distribution of correct and incorrect submissions.

We chose Neo4j because it is a popular graph database with a large community and a well-documented API. Neo4j also provides a data science library called Graph Data Science (GDS) that contains algorithms for graph analysis, like path finding, which is useful for the hint generation algorithm. Finally, Neo4j has a web interface that allows us to visualize the graphs and the results of the queries, which is useful for debugging and visualizing the state of the graph.

With the processed datasets, we create a graph for each predicate in each challenge. This step requires creating a separate database in the Neo4j server for each graph, and importing the data from the CSV files into the database. The import process starts by creating a node for each submission. Then, we connect nodes when one submission is the derivation of another.

The two steps require two queries which are shown in Listing 4.2. In this example, we are importing the data for the predicate *prop1* from the challenge *9jPK8KBWzjFmBx4Hb*. The first query iterates the rows of the CSV file and creates a node for each submission. A submission belongs to the *Submission* label. The second query identifies the parent submission of each submission and creates an edge between them with a random identifier. Additionally, in another query, we assign submissions with a *sat* value of 1 to the *Correct* label and submissions with a *sat* value of 0 to the *Incorrect* label.

After importing the data, the graph for the predicate *prop1* looks like Figure 4.10. It contains 153 nodes and 152 edges. Notice how the graph contains a root node from which all other branches originate. This root node represents the empty submission of the challenge from where all students start solving the exercise. If there was a submission disconnected from the root node (no path from the root to the submission), it would mean that some derivation chains were broken during the data preparation process.

Also, notice how it contains a lot of correct submissions (green nodes) in comparison to incorrect submissions (red nodes). This indicates that the challenge is easy since most students were able to solve it correctly.

At this point, the set of nodes in the graph contains repeated nodes with the same AST since

```
LOAD CSV WITH HEADERS FROM 'file://9jPK8KBWzjFmBx4Hb/prop1.csv' AS row
    MERGE (s:Submission {
        id: row._id,
        code: row.code,
        derivationOf: CASE WHEN row.derivationOf IS NULL THEN '' ELSE
row.derivationOf END,
        sat: toInteger(row.sat),
        expr: CASE WHEN row.expr IS NULL THEN '' ELSE row.expr END,
        ast: CASE WHEN row.ast IS NULL THEN '' ELSE row.ast END
    })
RETURN count(s) AS count

MATCH (s:Submission), (d:Submission)
WHERE s.id = d.derivationOf AND s.id <> d.id
MERGE (s)-[r:Derives {id: randomUUID()}]->(d)
RETURN count(r) AS count
```

Listing 4.2: Queries for importing data into Neo4j.

we haven't addressed them in previous steps yet. Consequently, the graph contains repeated edges between nodes with the same AST. We take this opportunity to count the number of repeated nodes and edges in the graph, which translates to the popularity of a submission and the popularity of a transition, respectively. So, for each node, we count the number of nodes with the same AST and store the result as the *popularity* property of the node. Similarly, for each edge, we count the number of edges with sources with the same AST and with targets with the same AST and store the result in the new *popularity* property of the edge.

After setting the popularity of each node and edge, having a graph with repeated nodes is not necessary anymore and becomes a problem, as it makes it more difficult to analyze the graph. For example, if we want to know all the derivations of a node, we would have to look at the outgoing edges of all the nodes with that specific AST. However, if we are able to concise all the information in a single node, we can just look at the outgoing edges of that node.

To remove repeated nodes, we create edges of type *equal* between nodes with the same AST to capture the equality relationship between nodes. Figure 4.11 shows the graph for the predicate *prop1* from the challenge *9jPK8KBWzjFmBx4Hb* after connecting all equivalent nodes. These relationships are temporary and disappear upon removing the repeated nodes from the graph.

Then, we run the Weakly Connected Components (WCC) algorithm on the graph considering only the edges of type *equal*. This algorithm is available in Neo4j's Graph Data Science (GDS) library and its implementation of the WCC algorithm based on [50] and [2]. The WCC algorithm assigns each node to a component, a set of connected nodes. Two nodes are connected if there is a path between them. The component identifier is stored in the *componentId* property of a node.

Notice that the number of components is equal to the number of unique ASTs in the graph. Thus, the objective is to keep only one node from each component and remove the rest since they all represent the same AST. In the case of components with only one node, no action is required.

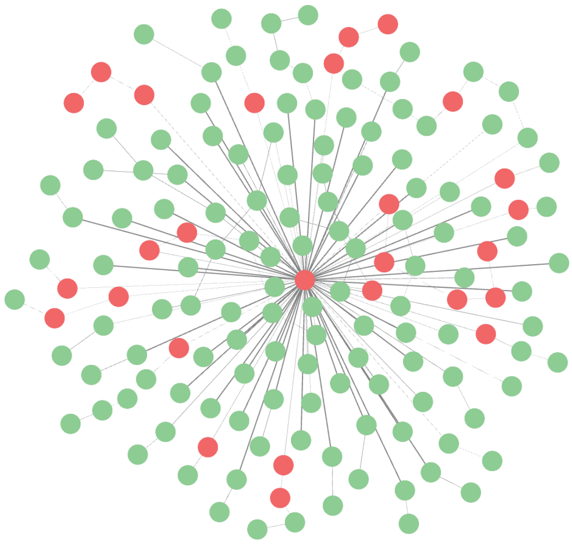

Figure 4.10: Graph with repeated nodes.

Otherwise, we choose one node randomly from each component to keep and remove the rest. However, by simply deleting each node we will be losing information about transitions on the graph. For example, if we delete a node that has outgoing edges, we will be losing the information about the transitions that start from that node.

For this reason, for each node we want to delete, we connect their parents to the chosen node and make the chosen node the parent of the children of the node we are deleting. This way, we concise all transitions of the same AST in a single node.

This process is depicted in Figure 4.12. This is an artificial example with a small number of nodes and edges to illustrate the process. In this example, nodes with the same color and label represent the same AST and each edge represents a derivation from the source node to the target node. Additionally, the number on the edge represents the popularity of an edge and the size of the node represents the popularity of the submission.

The left side of the figure shows the graph before removing equivalent nodes. In this side, each edge and node has a popularity of one. Moreover, nodes with the same AST are not aggregated yet, since they represent different submissions despite being equivalent.

On the other hand, the right side of the figure is the resulting graph after removing nodes with the same AST. There are only three nodes left in the graph since there were only three unique ASTs, and every AST kept their derivations. For instance, although the rightmost *AST3* node was removed, no derivation chain was broken and node *AST1* keeps deriving *AST3*. Also notice how the popularity of the nodes and edges is preserved, since *AST1* and *AST2* have a popularity of two. Similarly, the popularity of the edge between *AST2* and *AST3* is two since there are two derivations between these two nodes in the left side.

This technique does not destroy any derivation chain and makes all nodes in the graph have unique ASTs. However, it might create edges with the same source and destination nodes. These transitions indicate that no change was made to the submission, so they are not useful for hint

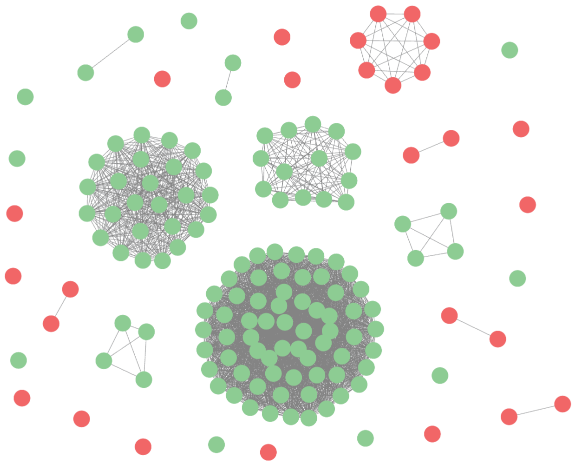

Figure 4.11: Graph components of equal nodes.

generation. For this reason, we remove these edges from the graph.

The final graph for the predicate *prop1* is shown in Figure 4.13. In this figure, the bigger nodes represent the most popular submissions, and the thicker edges represent the most popular transitions.

Without the repeated nodes, we have 26017 nodes and 39349 edges across all graphs. The total number of correct submissions is 4295, and the total number of incorrect submissions is 21722.

Finally, the last step for setting up the graph involves adding two new properties to all edges. The first property is the *TED* property, which represents the TED between the source and target nodes of the edge. To obtain the TED between two ASTs, we use the state-of-the-art algorithm described in Section 3.7.2. This algorithm is called APTED, and its implementation is available in GitHub[4].

The second property is the *operations* property, which contains the edit script that transforms the source AST into the target AST. As described in Section 3.7.3, calculating the edit script between two ASTs requires two steps: finding mappings between nodes of the ASTs and deriving the edit script from these mappings. For the first step, we used two different algorithms: the APTED algorithm and the GumTree algorithm, available at Github[5]. For the second step, we use the Chawathe et al. [10] algorithm implementation from the GumTree's repository.

The result is a sequence of edit operations of different types. There are four possible types of edit operations: *add*, *delete*, *update*, and *move*. Examples of these operations are shown in Figure 4.14, where *AST1* is the AST before applying the operation and *AST2* is the resulting AST after applying the operation.

The *add* operation represents the insertion of either a node (*Insert*) or a subtree (*TreeInsert*) in the AST. It contains a parent node and a position to indicate where to insert the new node or subtree. In the example of the figure, the *TreeInsert* operation inserts the subtree into the implicit node *root*. The position indicates that the subtree should be inserted as the parent's first child.

[4] https://github.com/DatabaseGroup/apted
[5] https://github.com/GumTreeDiff/gumtree

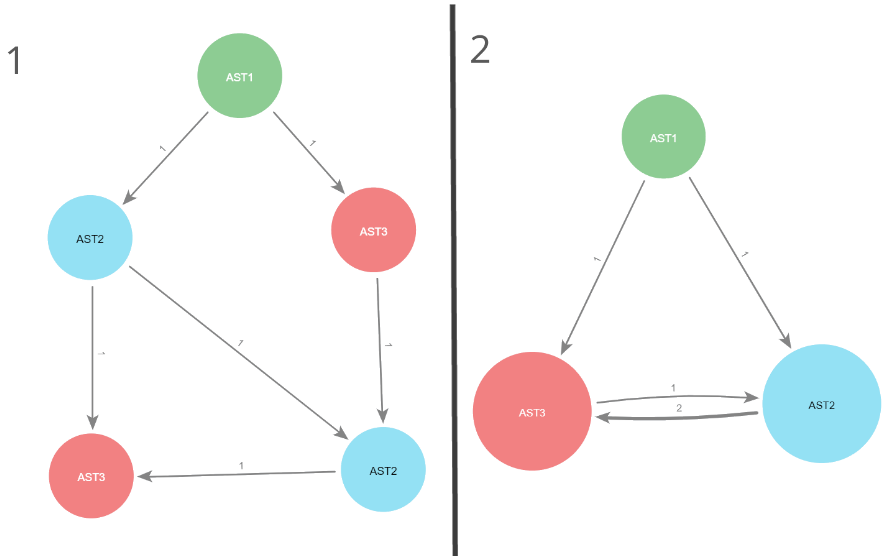

Figure 4.12: Example of removing equivalent nodes from a graph.

The *delete* operation represents the deletion of a subtree from the AST. The *update* operation represents the update of a node's label. It contains a parameter *node* that contains the label of the node to update and a parameter *value* that contains the new label of the node. Lastly, the *move* operation involves moving a subtree from one position to another. It contains the same parameters as the *add* operation.

Although HiGenA accepts move operations and subtree operations in the edit script, these types of operations are not considered by APTED in the calculation of the TED. This causes the TED to be different from the length of the edit script. For example, we might have an edit script with one insertion operation to insert a subtree with three nodes. In this case, the TED will be three despite the the edit script containing only one operation.

Before calculating the edit script, we do an additional step to handle special cases with commutative operators. For instance, consider the following two Alloy expressions: `no Trash and no Protected`, and `no File and no Protected`. After canonicalization, the ASTs of these expressions will look like the ones on the top part of Figure 4.15.

Notice that both ASTs are sorted alphabetically due to the canonicalization process for commutative operators. For the first AST, the branch with the `sig/Protected` leaf appears before the branch with the `sig/Trash` leaf. In the second AST, the branch with the `sig/Protected` leaf appears after the branch with the `sig/File` leaf.

If we calculate the edit script to transform the first AST into the second one, we will obtain a *move* operation to move the `sig/Protected` leaf from the first branch to the second branch, followed by an *update* operation to change the label of the `sig/Trash` leaf to `sig/File`. Although the update operation is correct, the move operation does not make sense in this context. As we are dealing with commutative operators, the order of the branches does not matter. Therefore, if we give a hint to the student to swap the branches, it will not help the student to solve the challenge.

To solve this problem, we implemented a technique that runs before calculating the edit script.

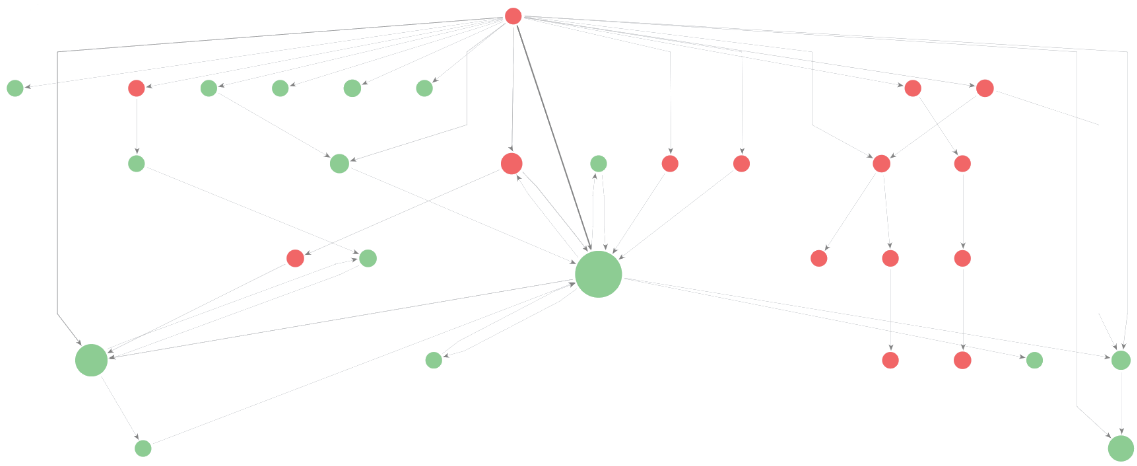

Figure 4.13: Final graph after aggregating equivalent submissions.

This technique checks whether both ASTs contain the same commutative operator. If they do, it checks whether they have a branch in common in different positions and it swaps the branches in the first AST to match the second AST and minimize the number of operations in the edit script.

The bottom part of Figure 4.15 shows the ASTs after applying this technique. Notice that the branch with the **sig**/Protected leaf is now in the same position in both ASTs. Therefore, the edit script will contain only one *update* operation to change the label of the **sig**/Trash leaf to **sig**/File.

This technique is limited to commutative operators since the order of the branches does not matter. For non-commutative operators, the order of the branches matters. Therefore, we cannot swap the branches in this case.

4.3.4 Hint Generation Algorithm

With the graph set up, we can now describe the implementation of the hint generation algorithm. This algorithm starts with a hint request. A hint request contains the student's current expression, the model code (for parsing purposes), the predicate name, and the challenge ID. It is assumed that the student's current expression is incorrect.

Whenever our tool receives a request, the first step is to parse and apply the canonicalization techniques to the AST of the student submission. This process allows us to obtain the canonicalized AST of the student's expression. We then search for a node in the graph with the same AST or add it to the graph if it does not exist. This node is called the *current submission*.

When the submission is not found in the graph, it means that it is a new submission. After adding this new node to the graph, we have to connect it to the rest of the graph. To do so, we apply a technique described in Section 3.6.2. This technique consists of creating paths between the current node and the closest correct state in the graph. In this case, the closest correct state is the solution submission with the lowest TED to the current node.

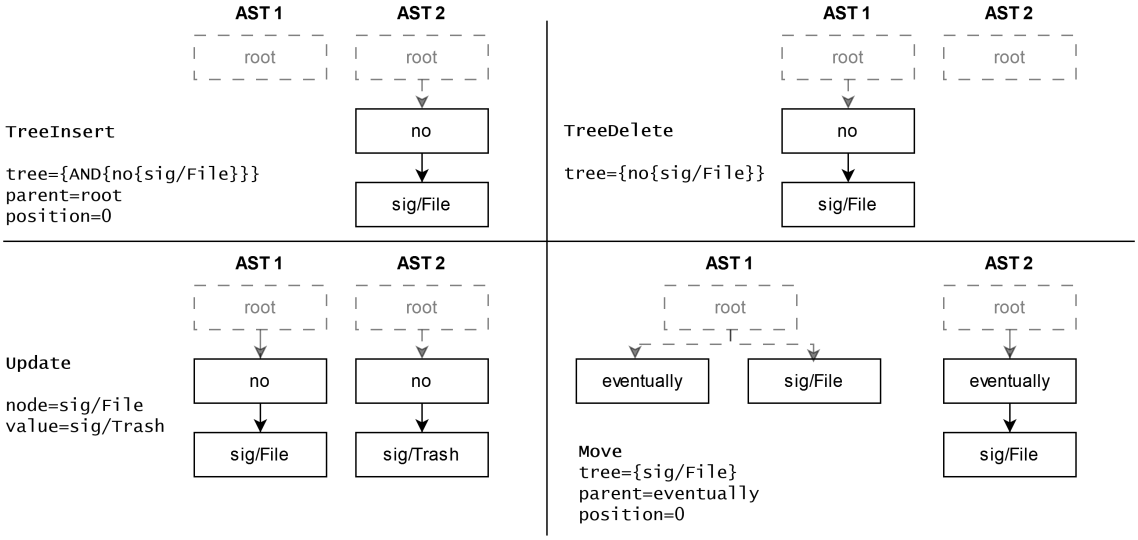

Figure 4.14: Example of types of edit operations.

Finding the closest correct solutions is done by iterating over all solutions in the graph and calculating the TED between the current node and each solution. The one with the lowest TED is the closest correct solution. In the case where there are multiple solutions with the same minimum TED, we choose the most popular one. To make this process more efficient, we iterate through the solutions in the graph starting from the most popular ones and we stop the search when we find a solution with a TED equal to 1.

After finding the closest solution, we create an edge from the current node to this state. This edge represents the path that the student should take to reach a correct solution, and it contains the information necessary to generate the hint message.

On the other hand, when the student's submission already exists in the graph, the algorithm aims to find the optimal path to a correct solution. The optimal path contains the optimal sequence of steps that leads the student to a solution. This path's first step (derivation edge) is the edge used for hint generation. However, defining the optimal path is not trivial, and many metrics exist that we can work with.

We implemented three different definitions for the optimal path: (1) the one with the lowest TED, (2) the path with the most popular transitions, and (3) the path with the most popular submissions. The user can specify the desired option in the hint request.

The path with the lowest TED is the one that minimizes the sum of the TEDs of all edges in the path. By minimizing the TED, we obtain the path with the least number of edit operations necessary to reach a solution. This metric is the most intuitive, but it does not take into account the popularity of the edges and nodes in the graph. Instead, it guides the student to the closest reachable solution. Its implementation is straightforward, using Dijkstra's path-finding algorithm with the TED as the edge weight.

The second possible path definition is the one with the most popular transitions. As mentioned before, each edge in the graph contains a popularity value that represents the number of times the

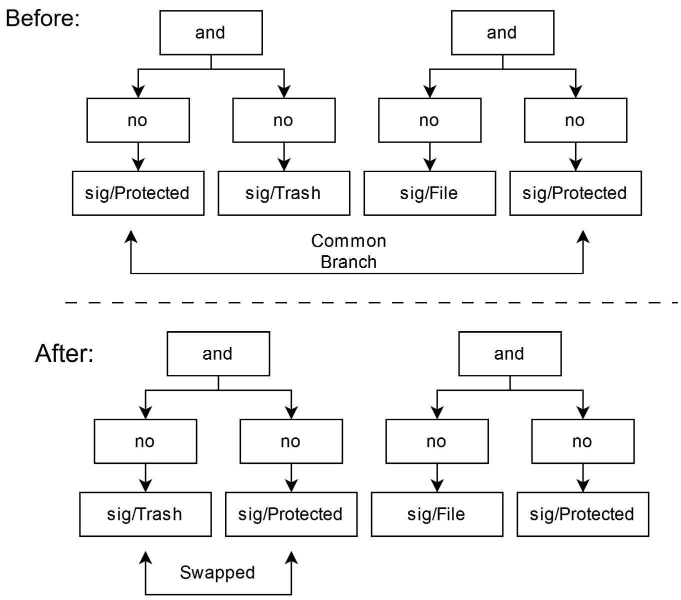

Figure 4.15: Example of swapping branches in commutative operations.

edge was traversed. However, we cannot use this value directly as the edge weight for Dijkstra's algorithm since the algorithm would always evade the most popular edges. To solve this problem, we create a new property for each edge which is the inverse of the popularity of the edge, and use it as the edge weight for Dijkstra's algorithm.

The final path definition is the one that traverses the most popular submissions. Each node in the graph contains a popularity value that represents the number of times the node was visited. To apply Dijkstra's algorithm, we create a new property for each edge that contains the inverse of the popularity of the destination node of the edge. This new property is used as the Dijkstra algorithm's edge weight.

The latter two path definitions are inspired by the Poisson Path presented in Section 3.6.5. The intuition behind these two options is that the most popular edges and nodes are the most valuable ones for the students. The most popular edges are the ones that students are more likely to traverse, and the most popular nodes are the ones that students are more likely to reach.

Listing 4.3 shows the Cypher query used to find the optimal path. The query receives the ID of the current node and the name of the graph projection to use. The projection is used to define the graph's structure and the properties to use in the path-finding algorithm. Neo4j's GDS library provides a function that implements Dijkstra's algorithm: `gds.shortestPath.dijkstra.stream`. This function receives the projection and the source and target nodes and the relationship weight property. In this example, we specified the TED as the relationship weight property and defined the target node as any correct submission in the graph. Thus, the result of this query is the path with the lowest TED between the current node and any correct submission in the graph, if it exists.

```
MATCH (source:Submission {id: "$NODE ID$"}), (target:Correct)
```

```
WHERE source.id <> target.id
CALL gds.shortestPath.dijkstra.stream('$PROJECTION$', {
    sourceNode: source,
    targetNode: target,
    relationshipWeightProperty: 'TED'
})
YIELD totalCost, path
RETURN
    totalCost,
    nodes(path) AS path,
    relationships(path) AS rels
ORDER BY totalCost
LIMIT 1
```

Listing 4.3: Dijkstra's shortest path algorithm in Neo4j.

When the optimal path is found, before proceeding to the hint generation, we check the quality of the path. This check is necessary to avoid problematic paths that can lead to misleading hints. Assessing the quality of a path is done by comparing the TED of the found path with the TED between the current node and the solution. When the TED of the path is greater than the TED between the current node and the solution, we create a new path from the current node to the solution. This new path is used for hint generation instead of the previous optimal path.

Dijkstra's algorithm always returns a path to a solution when it is possible. However, there might not be a path to a solution from every submission in the graph. Figure 4.13 is an example of this situation. This is caused by students giving up on solving an exercise which results in a branch of a graph that ends in an incorrect submission. To handle these cases, we employ the path construction technique again, equally to the case where the student's submission is not in the graph.

This section presented the process of finding the optimal path to a solution and the process of constructing a path when necessary. Listing 4.4 presents a simplified Java pseudocode of the process to get a better understanding of the algorithm. The full implementation is available in the project's repository[6].

```
public Hint generateHint(String ast) {
  // Get source node
  sourceNode = db.getNodeByAST(ast);
  // New node
  if (sourceNode == null) {
    // Create the node with the AST
    source = db.addIncorrectNode(expression, ast, code);
    // Create a path to the most similar solution node
    return createPath(sourceNode);
```

[6] https://github.com/anaines14/HiGenA

```
  }
  // Node is not new
  path = getShortestPath(sourceNode);
  if  (path == null) {
    // Create a path to the most similar solution node
    return createPath(sourceNode);
  }

  //  Compute TED between the source and the target nodes
  String srcAST = sourceNode.get("ast").toString(), dstAST = targetNode.
 get("ast").toString();
  int srcDstTED = TED.computeEditDistance(srcAST, dstAST);

  // Evaluate path:
  if (totalTED <= srcDstTED) {
    // Path found is good
    hint = new Hint(srcDstTED, firstEdge);
    return;
  }

  // Path found is bad (TED higher than TED(src, dst)).
  // Create a better path to a solution node
  return createPath(sourceNode);

  // Generate hint message
  hint = new Hint(sourceNode, targetNode, firstEdge);
}

private Hint createPath(Node source) {
  // Get the most similar correct node
  Node target = db.getMostSimilarNode(source.get("ast").toString(), "
 Correct");

  // Create edge between the two nodes
  edge = db.addEdge(source, target);

  // Generate hint message
  return new Hint(source, target, edge);
}
```

Listing 4.4: Hint Generation simplified pseudocode.

4.3.4.1 Hint Message Generation

The final step of the hint generation process is to generate the hint message for the student. The hint message should be a textual message that guides the student through the optimal path toward the solution. For this reason, the hint message should be based on the first edge of the path since it will indicate the next step that the student should perform to get closer to the solution.

We translate one edit operation from the first edge's edit script into a textual message to generate a hint message. However, each operation in the edit script can have different types and information. So, to generalize the transformation of operations into textual messages, we created a set of hint templates, one for each type of operation. These templates are textual messages containing placeholders for the operations' information.

Each template was created with the help of ChatGPT's language model[7] that generates text based on a prompt. We created prompts that specified that the generated text should be similar to what a teacher would say to a student to help them solve the challenge. Then, we presented the full challenge model along with the current student's submission, the solution where the student should arrive, and the edit operation that should be performed.

An example of a prompt is presented in Listing 4.5. This prompt was used to generate a hint for one of the incorrect submissions present in the graph for predicate *prop1* from challenge *9jPK8KBWzjFmBx4Hb*, which is the graph depicted in Figure 4.13. The prompt contains the full challenge model, the student's submission (`no File`), and the solution (`no (Trash+Protected)`). If we calculate the edit script between the student's submission and the solution, one of the edit operations is an *insert* operation that inserts the `+` operator. We indicate this operation in the prompt and the result is a textual hint with detailed information about the specific example that we can generalize for other cases.

The generated text was manually reviewed and corrected to remove any errors. We also removed any information specific to the example used in the prompt and replaced it with placeholders. Then, we adapted each template to accommodate the different types of operations and the information they contain.

The prompts generated by the language model always contained some context regarding the function of Alloy keywords and operators. This information is necessary for the student to understand the Alloy language and help debug their erroneous submissions. Thus, to present this information in the hint message, we created a description for the most popular Alloy keywords and different templates for specific cases where the student is missing variables.

Additionally to the templates, we generate a small sentence indicating how close the student is to a solution. This sentence aims to give students a sense of progress and encourage them to continue solving the challenge. The final hint message is the result of combining this sentence with the hint template.

An example of a hint for an update operation where the TED between the student submission and the solution equals one looks like this: "One step away from the solution! Instead of using

[7] https://openai.com/blog/chatgpt

```
--------
Prompt:
--------
Consider the following Alloy model:
var sig File {
  var link : lone File
}
var sig Trash in File {}
var sig Protected in File {}

A teacher created an exercise where the students have to write predicates
    to complete the model specification. The requirement of the first
    predicate says: "initially the trash is empty and there are no
    protected file".

The student wrote: "no File" which is incorrect.
The teacher wants to help the student reach the solution: "no (Trash+
    Protected)".

Generate the hint the teacher should give the student so that he adds the
    "+" operator to their expression.

----------------
Generated text:
----------------
The hint the teacher should give the student is:

"Think about how the 'no' keyword works in Alloy. It negates the existence
     of a specific signature. In this case, you want to specify that there
    are no files that belong to either the Trash or the Protected signature
    . So, you need to combine the two signatures using the '+' operator.
    Try incorporating '+ (Trash + Protected)' into your expression."
```

Listing 4.5: Example of a prompt used to generate a hint template.

the transpose operator ('∼') to transpose a relation, try using the transitive closure operator ('^') to get the transitive closure of a relation.” In this case, the hint indicates that the student requires only one step. It also describes each operator’s function.

For a delete operation where the TED equals 2, hints are similar to the following: “Near a solution! It seems like you have unnecessary elements in your expression. You can simplify your expression by deleting the identity relation constructor ('**iden**'). If you want to keep it, try to fix your expression another way and reach a different solution!”. This hint indicates that the student is close to a solution but that they have extra information that needs to be removed. Additionally, the hint motivates the student to try an alternative solution.

Finally, a possible hint for an insert operation where the TED equals 3 could be: “Keep going! You can use variables to help specify the condition. Consider introducing a new variable of type State to your expression using the universal quantifier ('**all**').” This hint indicates that the student

Figure 4.16: Hint example in Alloy4Fun.

is further away from the solution in comparison with the other hints, but it still motivates the student to continue solving the challenge.

The delete and update hints were generated using the information directly present in the operation used. The update operation implied that a node with label $\sim$ should be replaced with a node with label ^. The delete operation suggested that the node with label **`iden`** should be removed. On the other hand, the example of the insert hint is slightly more complex.

In the insert example, the operation requires the addition of a node with label **`all`** to the expression. However, every time a node with label **`all`** is missing, the declaration of a variable is also missing. HiGenA then applies this logic to generate the hint. The hint generation tool looks for the operation containing the **`all`** subtree and extracts the variable type from one of the subtrees.

Finally, an example of a move hint looks like this: "Keep going! It seems like the temporal operator ('historically') is not in the right place. Try moving it inside the disjunction operator's ('||') expression."

4.3.5 Deployment in Alloy4Fun

The deployment of HiGenA involved the integration of the hint generation tool with Alloy4Fun. The integration was done by adding a new endpoint to the Alloy4Fun API (*POST /higena*) that receives the student's model, the ID of the challenge, and the name of the predicate that the student is trying to complete. Then, the API calls the hint generation tool with the received information and waits for the tool to return the generated hint. During this time, HiGenA performs the necessary operations and requests to the Neo4j database to be able to able to generate the hint.

Additionally, the front end of Alloy4Fun was updated to display the hint to the user. We added a new button with the label *Show HiGenA hint* to the interface that the user can click to request a hint. When the user clicks the button, the front end sends a request to the API with the information necessary to generate the hint. When the API returns the hint, the web page displays it in a blue box below the code editor. Figure 4.16 illustrates the display of a hint in Alloy4Fun.

Figure 4.17 shows this process in the architecture diagram. Both the API and HiGenA are written in Java. The front end of Alloy4Fun uses the Meteor [8] framework, a full-stack JavaScript

[8] https://www.meteor.com/

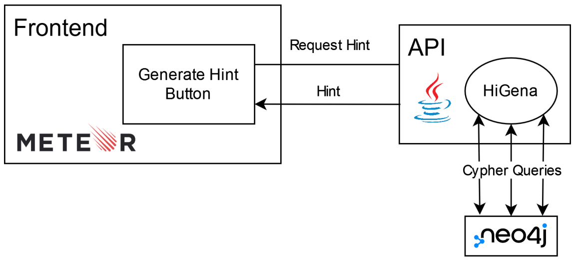

Figure 4.17: Hint request process in Alloy4Fun.

platform for building web and mobile applications. We also use Docker[9] to ensure a simple and ubiquitous development environment.

4.4 Conclusion

In this chapter, we presented our solution to the problem of hint generation for Alloy4Fun using historical student data. The problem at hand was caused by the lack of feedback provided to the students when they failed to solve a challenge, making it difficult for them to understand their mistakes and correct them without the help of a teacher.

To solve this problem, we worked with the dataset of historical student submissions collected from Alloy4Fun during 2019 and 2023. We preprocessed the dataset and created graphs of submissions where directed edges represent transitions between submissions that students have done in the past. Finally, we searched for optimal paths in these graphs to generate hints for the students using the differences between the ASTs of students' submissions.

In the next chapter, we present the results of our evaluation of HiGenA and discuss its effectiveness, efficiency and limitations in addressing the problem of hint generation for Alloy4Fun.

[9] https://www.docker.com/

Chapter 5

Evaluation

5.1 Introduction

In the chapter, we present the evaluation methods, results, and the analysis of results for the evaluation of HiGenA. This evaluation aims to assess to which extent HiGenA contributes to the state-of-the-art in the field of automated data-driven hint generation for Alloy.

With this evaluation, we aim to answer the following research questions:

- How effective is HiGenA in generating hints? e.g., how does HiGenA leverage student data to generate hints?
- What is the performance of HiGenA? e.g., how long does it take to generate a hint?
- What is the quality of the hints generated by HiGenA? e.g., do the hints generated by HiGenA help students solve the challenges?

This chapter is organized into three main sections, each answering one of the research questions. First, we want to evaluate the effectiveness of the technique, by analyzing the hint-generation algorithm. Then, we analyze the performance of HiGenA to understand whether the technique is fast enough to generate hints in a timely manner. Finally, we look into the quality of the hints generated by HiGenA.

As a quick note, in this chapter, we do not evaluate the capability of HiGenA to generate hints. As stated previously, as long as there is at least a solution in the graph of student submissions, HiGenA will generate a hint. This is due to the fact that HiGenA uses a path construction technique that always creates a path in the graph to a solution when no other path exists.

This evaluation combines both quantitative and qualitative methods to assess HiGenA thoroughly. Additionally, we compare the results of HiGenA with the state-of-the-art tool, TAR [9].

5.2 Effectiveness Evaluation

In this section, we evaluate the effectiveness of HiGenA. We start by evaluating the canonicalization techniques' impact on the graph of student submissions. Then, we analyze to what extent HiGenA uses the information provided by historical student submissions to generate hints.

5.2.1 Canonicalization Evaluation

As mentioned in Section 4.3.2.5, we apply two canonicalization techniques to the ASTs of student submissions: *sorting commutative operations* and *variable anonymization*. In this section, we evaluate the effects of both techniques on the graph of student submissions.

The canonicalization techniques aim to obtain a canonical form for ASTs to eliminate redundant differences. Consequently, the number of unique states in the graph of student submissions should decrease after applying the canonicalization techniques. Additionally, the canonicalization techniques impact the number of hits in the graph.

We want to understand whether HiGenA is, in fact, using the information provided by historical student submissions to generate hints. If students tend to submit different submissions from students in the past, then the hint generation algorithm employed by HiGenA is not benefiting from the historical data available.

Therefore, we will look into the number of hits in the graphs. A hit is when a student submission already exists in the graph when requesting a hint. As stated in Section 4.2.2, the first step of the path-finding algorithm is to find the submission of the student requesting the hint in the graph. If the submission does not exist in the graph, the algorithm creates a new node for the submission and adds it to the graph. Then, the hint-generation algorithm resorts to the path construction technique to create a path to a solution.

A reduction in the number of unique submissions in the graph and a higher number of hits means that the canonicalization techniques are effective.

5.2.1.1 Evaluation Methodology

We use the dataset of student submissions from the Alloy4Fun platform to evaluate the canonicalization techniques described in Section 2.2.6. In total, the dataset contains 97755 submissions. However, before going through the canonicalization process, we filter the dataset to prepare it for the graph creation process. The data preparation process is described in Section 4.3.2, but we modify it slightly to fit the evaluation process, mainly by modifying the canonicalization step.

We evaluate each canonicalization technique separately to assess its effects on the graph of student submissions and to compare the results of each technique. Then, we evaluate the combination of both techniques. We use the same dataset for all evaluations but apply a different data preparation process for each.

The difference between each data preparation process is the canonicalization techniques applied to the ASTs of student submissions. For example, if we are evaluating the effects of sorting

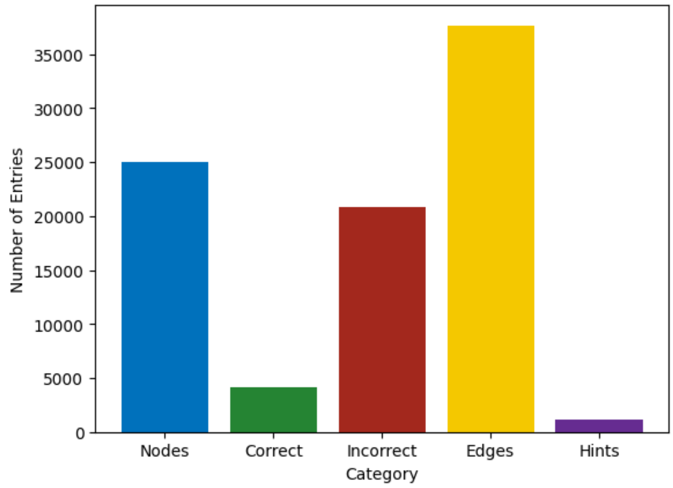

Figure 5.1: Statistics of the dataset used for the hint-generation evaluation.

commutative operations, we only apply this technique to the dataset during the data preparation step.

On the other hand, to evaluate the number of hits in the graph, we split the dataset into two parts: a training set and a test set. The training set contains submissions submitted before June 2023 and the test set contains only incorrect submissions submitted between June and July 2023 without duplicates.

The training set is used to create the graph of student submissions, while the test set is used to evaluate the hint-generation process. For each submission in the test set, we generate a hint. However, before generating each hint, we rebuild the graph of student submissions so that submissions from the test set do not influence the hint-generation process of other submissions.

After the data preparation step, the training set contains 63561 submissions, with 27191 correct submissions and 36370 incorrect submissions. After creating the graphs of student submissions, there are 25018 nodes across all graphs, including 4158 correct nodes and 20860 incorrect nodes. The total number of edges is 37665. The test set contains 1180 incorrect submissions, which we will use to generate the same number of hints. Figure 5.1 plots the distribution of the number of nodes and edges across the graphs of student submissions in the training set compared to the number of hints to be generated from the test set.

For each hint generated, we note if there was a hit in the graph of submissions.

5.2.1.2 Evaluation Results and Analysis

Table 5.1 shows the results of the evaluation of the canonicalization techniques. The middle column shows the number of unique states in the graph for each canonicalization technique, and the last column shows the reduction in the number of unique states compared to not applying any canonicalization technique.

The first row contains the number of unique states in the graph without applying any canonicalization technique: 29402 states. Out of the two canonicalization techniques, sorting commutative

Table 5.1: Canonicalization evaluation results.

Canonicalization	Number of unique ASTs	Reduction (%)
None	29402	0 (0%)
Sorting Commutative	28897	505 (1.72%)
Variable Anonymization	26583	2819 (9.59%)
Both	26017	2434 (11.51%)

operations has the least effect on the number of unique states (1.72% fewer states than without canonicalization). On the other hand, variable anonymization has a more significant impact on the number of unique states (9.59% fewer states). Thus, variable anonymization is the most efficient technique out of the two.

The last row shows the results of applying both canonicalization techniques to the dataset. The number of unique states in the graph is 26017, which is 11.51% fewer states than without canonicalization. As expected, the combination of both techniques significantly impacts the number of unique states in the graph.

The results show a difference between the impact of the two techniques, as variable anonymization reduces the number of unique states by 7.87% more than sorting commutative operations. This difference might be because variable anonymization covers a broader range of differences than sorting commutative operations. For instance, there are only two possible ways to sort two operands for one binary commutative operation: $a = b$ and $b = a$. On the other hand, one can use an infinite number of variable names for a variable in Alloy. Without variable anonymization, if ten students write the same expression with different variable names, the graph will end up with ten more submissions despite all representing equivalent ASTs.

Nonetheless, the results of the evaluation show that the impact of both canonicalization techniques together is relatively small (less than 12%). This result indicates that there might still be redundant differences in the ASTs of student submissions that the canonicalization techniques do not cover. Therefore, we need to investigate Alloy ASTs further to find other canonicalization techniques useful for the task at hand.

Figure 5.2 shows the percentage of times the hint-generation algorithm created a new node in the graph of student submissions. The results show that 73.1% of the time, the submission requesting the hint did not exist in the graph yet. Consequently, only 26.9% of the time, the graph already contained the submission.

Unfortunately, the number of hits in the graph is low. This might be an indicator that the canonicalization process is not as effective as we thought and could be improved to increase the number of similar submissions and hits in the graph. However, we believe that the number of hits is low due to the nature of Alloy language, which allows students to write the same requirement in many different ways. Nonetheless, the number of hits is expected to increase as time passes and more students submit their solutions which will be added to the graph.

Despite the small impact of the canonicalization techniques, they still contribute to the overall quality of the hints generated by HiGenA. The canonicalization techniques directly impact the edit

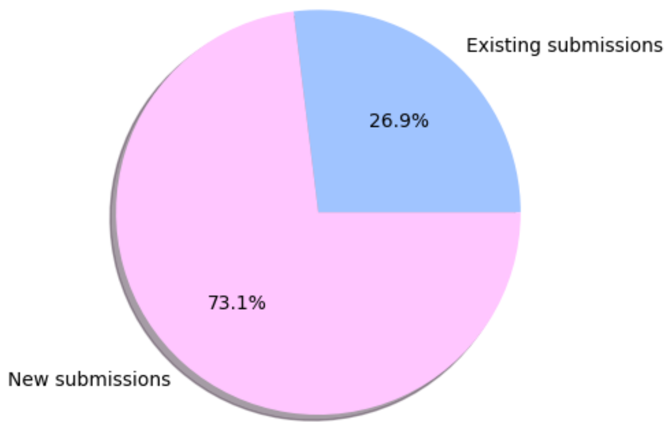

Figure 5.2: Percentage of times the hint-generation algorithm created a new node.

script generated between two ASTs. Sorting commutative operations can avoid generating hints for changing the order of operands and eliminating variable name differences can avoid generating hints for renaming variables, both of which would be unhelpful to students. Thus, we believe it is worth keeping these techniques in HiGenA, and we aim to explore other techniques in the future.

5.2.2 Hint-Generation Evaluation

In this section, we evaluate the hint-generation process of HiGenA, where we analyze the effectiveness of the algorithm. Once again, the objective is to find out to what extent HiGenA uses the information provided by the historical student submissions to generate hints.

We will measure how HiGenA leverages previous student submissions to generate hints. Previously, we looked at the number of hits in the graph of student submissions. Now, we will look at the number of times the hint-generation algorithm resorts to the path construction technique to create a path to a solution. These two cases are the only ones where we employ the path construction technique to generate hints. Thus, our objective with HiGenA is to make sure that the path construction technique is only employed when necessary so that the algorithm can benefit from the historical data available as much as possible.

The algorithm creates a new path in the graph to replace a problematic path that already exists and that was taken by students in the past. A problematic path is a path that leads to a solution with a total TED greater than directly connecting the source to the solution. How HiGenA handles problematic paths is explained in Section 4.2.2.

By avoiding problematic paths, HiGenA is not using the problem-solving history of students effectively to generate hints. Thus, a low number of new paths created by the algorithm indicates that HiGenA is benefiting from student submissions from the past to generate hints.

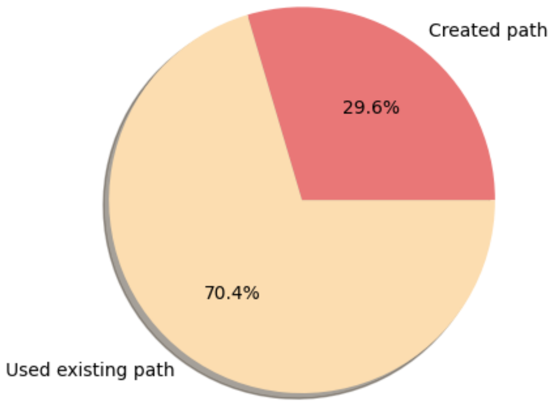

Figure 5.3: Percentage of times the hint-generation algorithm created an alternative path.

5.2.2.1 Evaluation Methodology

We use the same dataset as in the previous evaluation to evaluate the hint-generation process. We use the training set to build the graph of student submissions and the test set to evaluate the hint-generation process.

For each hint generated, we take note of whether a new path was created in the graph to replace the found problematic path.

5.2.2.2 Evaluation Results and Analysis

Figure 5.3 shows the results of the evaluation of the number of times an alternative path was created in the graph. These statistics do not include times when a path was created to connect a new submission to the rest of the graph.

These results show that 70.4% of the time, the hint-generation algorithm did not create an alternative path. This means that only 29.6% of the time did the algorithm avoid using the solving process of previous students.

These results indicate that if there is a hit in the graph upon a hint request, the hint-generation algorithm will most likely use paths students took previously to generate hints. This is a good indicator that the hint-generation algorithm considers the problem-solving process of previous students useful and will take advantage of it when possible. Moreover, as new data becomes available, the number of hits in the graph will increase, which should increase the number of times HiGenA uses the solving process of previous students.

5.3 Performance Evaluation

In this section, we evaluate the performance of the hint-generation algorithm of HiGenA. The objective is to assess the algorithm's ability to generate hints in a reasonable time. The tool mustn't take too long to generate hints. Otherwise, the student might become frustrated and give up on the problem. Thus, with this evaluation, we will generate hints for a set of student submissions and measure the time it takes to generate each hint.

5.3.1 Evaluation Methodology

In this evaluation, we use the same dataset as in the previous section. We use the training set to create the graph of student submissions and the test set to generate hints.

For each generated hint, we take note of the time it took to generate the hint in milliseconds. The time is measured from the moment the hint-generation algorithm starts to the moment it finishes, which excludes the time it takes to parse the student expression.

The machine used for this evaluation has an AMD Ryzen 5 3600X 6-Core Processor 3.80 GHz, 16 GB of RAM, and runs Windows 10. The evaluation was performed using the HiGenA tool version 1.0.0. and Neo4j version 1.5.8.

5.3.2 Evaluation Results

The results of the evaluation show that, on average, it takes 64.03ms for HiGenA to generate a hint. The longest time it took to generate a hint was 325.59ms, while the shortest time was 27.16ms. Moreover, it took HiGen 75.55 seconds to generate all the hints for the test set (1180 hints).

These results indicate that HiGenA outperforms TAR, which repaired 35% incorrect specifications in 2 seconds for the dataset used in [9].

5.4 Quality Evaluation

The final evaluation we perform is a quality evaluation of the hints generated by HiGenA. This evaluation involves both quantitative and qualitative analysis.

The quantitative analysis involves evaluating the mapping algorithms employed by HiGenA to find mappings between ASTs. The objective is to find which algorithm is more efficient at finding mappings between ASTs, which impacts the overall quality of the hints generated by HiGenA.

On the other hand, the qualitative analysis involves two user studies. The objective is to understand how helpful the hints generated by HiGenA are to students.

5.4.1 Mapping Evaluation

In Section 4.3.3, we described the process of creating a graph of student submissions. One of the steps in this process is to obtain the edit script between ASTs connected by an edge in the graph. Calculating the edit script requires two steps: finding mappings between nodes in the ASTs and deriving the edit script from these mappings.

There are two state-of-the-art algorithms for finding mappings between nodes in ASTs: GumTree [14] and APTED [42, 41]. In HiGenA, we implemented both algorithms to compare the resulting edit scripts generated by the two.

5.4.1.1 Evaluation Methodology

We use the dataset of student submissions from the Alloy4Fun, described in Section 4.3.1, to evaluate the mapping algorithms. This dataset was used to create several graphs of student submissions, which we use for evaluation. However, we modify the graph construction process to fit the evaluation process. Depending on whether we are evaluating GumTree or APTED, we apply the corresponding mapping algorithm when calculating the edit script between two ASTs. The rest of the graph construction process is described in Section 4.3.3.

After creating all graphs, we query each one to obtain all edit scripts in a graph. The query is shown in Listing 5.1. It returns the expression of the source and target nodes of the edge and the edit script between them. We store all edit scripts of all graphs in a single dataset and add two columns to identify the challenge and predicate the submissions belong to.

```
MATCH (a:Submission)-[r:Derives]->(b:Submission)
RETURN a.expr as source, r.operations as editScript, b.expr as target
```

Listing 5.1: Cypher query to obtain all edit scripts in a graph.

After obtaining a dataset of edit scripts using both mapping algorithms, we merge the two datasets into a single dataset on the challenge, predicate, source expression, and target expression columns. This merge allows us to compare the edit scripts generated by the two algorithms for the same submissions. Finally, we count the length of each edit script and generate a column containing the difference between the length of the edit script generated by GumTree and the version generated by APTED. We then calculate the mean and standard deviation of the values of this column and the mean of the length of the edit scripts generated by each algorithm.

5.4.1.2 Evaluation Results and Analysis

Table 5.2 shows the results of the evaluation. The shorter column describes the category of each group of edit scripts. The *Equal* category contains edit scripts of the same length for both algorithms. The *APTED* category contains edit scripts that are shorter for APTED when compared with GumTree, and the *GumTree* category contains edit scripts that are shorter for GumTree when compared with APTED. The quantity column indicates the number of edit scripts falling into each category. The mean length columns show the mean length of the edit scripts in each category for each algorithm (APTED/GumTree). Meanwhile, the mean diff and standard deviation diff columns show the mean and the standard deviation of the difference between the edit script lengths of the two algorithms in each category.

The mean and standard deviation columns show each category's mean and standard deviation of the difference between the edit script lengths of the two algorithms.

The results show that most of the time (72.08%), the edit script length is equal for both algorithms. However, there are cases where one algorithm generates a shorter edit script than the other. In 17.21% of the cases, APTED generates a shorter edit script than GumTree. In 10.71% of the

Table 5.2: Edit script length evaluation results using GumTree and APTED.

Shorter	Quantity (%)	Mean APTED/GumTree Length	Mean Diff	Standard Dev.
Equal	72.08%	2.62/2.62	0	0
APTED	17.21%	7.08/8.39	-3.55	2.90
GumTree	10.71%	14.21/10.63	5.82	7.37

cases, GumTree generates a shorter edit script than APTED. Thus, APTED generates shorter edit scripts than GumTree 6.5% more often than GumTree generates shorter edit scripts than APTED.

As expected, when the edit script length is equal, the mean length of the edit scripts generated by both algorithms is also equal. Consequently, the mean and standard deviation of the difference between the edit script lengths are zero. However, when the edit script lengths differ, the mean and standard deviation have different values.

The means of the difference between edit script lengths were calculated by subtracting the length of the edit script generated by GumTree from the length generated by APTED. Thus, when APTED generates a shorter edit script, the mean is negative (-3.55), and when GumTree generates a shorter edit script, the mean is positive (5.82).

These results indicate that, in general, when APTED generates a shorter edit script, the difference between the lengths is usually smaller than when GumTree generates a shorter edit script. This conclusion is supported by the mean length of the edit scripts. When APTED generates a shorter edit script, the length of the edit scripts is 7.08 on average: even when GumTree generates a shorter edit script, the length of the edit scripts is 10.63 on average.

Moreover, when APTED generates a shorter edit script, the standard deviation is 2.90; when GumTree generates a shorter edit script, the standard deviation is 7.37. Thus, the difference between the lengths of the edit scripts generated by APTED and GumTree are more consistent when APTED generates a shorter edit script.

For the most part, picking either algorithm to generate the edit script will result in the same length. However, equal length does not mean the edit scripts are identical. So, we calculated the percentage of identical edit scripts for this category. The pie chart in Figure 5.4 shows the results: 95.1% of the edit scripts with equal length are identical, while 4.9% are not.

To conclude, the results show that, for the most part, picking either algorithm will generate the same edit script. However, there is a small percentage of cases where each algorithm generates different edit scripts. From these cases, most of the time, APTED generates a shorter edit script than GumTree. On the other hand, when GumTree generates a shorter edit script, the difference between the lengths is much more significant. Thus, it might be beneficial for hint generation to identify the cases where GumTree greatly outperforms APTED and use GumTree in those cases.

Additionally, the edit script length is sometimes equal, but the edit scripts are not identical. Once again, one should identify these cases and understand which edit script is more appropriate for the hint-generation process. We will leave both these tasks for future work as they are not the focus of this thesis and might require manual inspection of the edit scripts. Either way, we leave

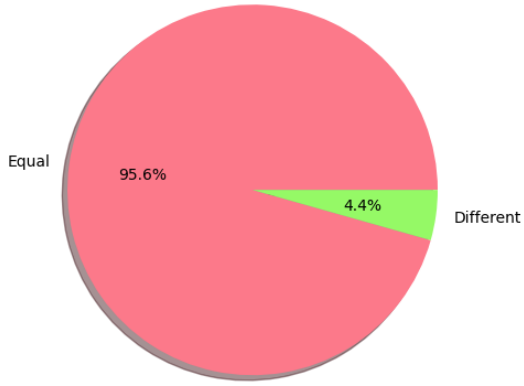

Figure 5.4: Percentage of identical edit scripts of equal length.

both implementations available in the HiGenA tool and allow the user to choose which algorithm to use. We set the default algorithm to APTED as it generates shorter edit scripts more often.

5.4.2 Student Study

In this section, we present the results of the student study. We start by presenting the study's methodology and the data collected. Then, we present the results of the study and discuss them.

5.4.2.1 Student Study Methodology

The target participants of this study are students that have previously enrolled in a course that uses Alloy4Fun as a learning tool for learning Alloy. It involves solving two Alloy4Fun challenges that contain many submissions in the respective graph. The first challenge is the *graph* challenge[1], which requires students to complete the specification of the model of a graph. This challenge contains eight predicates to be filled in by students, and the main emphasis of this challenge is to practice the use of the ~ and ^ operators.

We use the full dataset of submissions available to create the graphs for this challenge. Table 5.3 shows the statistics for these graphs. The `weaklyConnected` predicate is the predicate with the highest number of incorrect submissions and the highest number of edges in the graph. Moreover, if we compare this predicate with other predicates from the full dataset, it is in the top three predicates with the highest number of incorrect submissions. Thus, we believe this predicate should be difficult for students to solve.

The second challenge is the `LTS` challenge[2], which involves the specification of a model of a labeled transition system comprised of states and transitions. This challenge contains seven predicates for students to complete, and the main emphasis of this challenge is to practice working with ternary relations. Dealing with ternary relations is a more complex task that students tend to struggle with. Thus, we expect that students will find it difficult to solve this challenge.

[1]Available at http://alloy4fun.inesctec.pt/28fwdmjL79X4SQ9EP
[2]Available at http://alloy4fun.inesctec.pt/gqS3qTTn4B62NYmJX

Predicate	Submissions	Correct	Incorrect	Edges
undirected	150	29	121	246
oriented	94	24	70	150
acyclic	78	15	63	126
complete	104	18	86	191
noLoops	59	15	44	106
weaklyConnected	251	16	235	412
stronglyConnected	77	16	61	128
transitive	58	16	42	82

Table 5.3: Graph challenge statistics.

The statistics for the graphs of this challenge are shown in Table 5.4. In general, the statistics between predicates are unbalanced. While some predicates reached over 200 incorrect submissions (`inv4`), others only reached 41 submissions in total (`inv6`). This is due to the fact that students might skip some predicates and move on to the next ones. Thus, we are expecting to evaluate whether HiGenA motivates students to solve all predicates.

The study protocol employs the A/B testing methodology. The students are randomly assigned to one of two groups. The first group solves the first challenge (the graph challenge) with free access to HiGenA hints, while the second group solves the first challenge without the help of any hints. After solving the first challenge, the groups are switched: the first group solves the second challenge (the LTS challenge) without hints, while the second group solves the second challenge with the help of HiGenA.

Both groups are given a time limit of 30 minutes to solve at least six predicates of each challenge. However, if a student desires to continue solving the challenge after the time limit expires, they are allowed to do so. Students are free to choose which predicates they want to solve and can give up on solving a challenge before the time limit expires. During the full duration of the study, we record each student's computer screen and the room's audio and ask students to think aloud while solving the challenges.

The students are also allowed to consult the set of slides used in the classes, which contains Alloy syntax and examples of Alloy models. The students are not allowed to use any other resources. An exception is made in the graph challenge, where students consult the links present in the comments of predicates to help understand the requirements.

Before starting the study, the students are given a form to fill in to understand their background in Alloy. Then, after the first challenge, students fill in the second part of the form, which contains questions about the challenge difficulty and their current motivation to continue solving the challenge. Finally, after the second challenge, students fill in the third part of the form, which contains additional questions regarding their experience with HiGenA. The form is available in Appendix A.1.

Predicate	Submissions	Correct	Incorrect	Edges
inv1	49	11	38	86
inv2	35	7	28	51
inv3	133	9	123	229
inv4	309	24	285	430
inv5	149	15	134	201
inv6	41	18	23	63
inv7	173	9	164	222

Table 5.4: LTS challenge statistics.

5.4.2.2 Preliminary Student Study Results

Unfortunately, due to a lack of candidates and timing constraints, we were only able to recruit two students from the Faculty of Engineering of the University of Porto to participate in the study. This number is not enough to draw any conclusions about the effectiveness of HiGenA. However, we believe that their feedback is still valuable to understand the potential of HiGenA.

Both students are master's students in Informatics Engineering at the Faculty of Engineering of the University of Porto and between 20 and 25 years old. Both students also attended the same Alloy course in the past, but in different editions. The last time the student assigned to the first group had classes related to Alloy was one year ago, while the student assigned to the second group had classes in the last 6 months. However, the first student admitted having studied a bit of Alloy before the day of the study to refresh his knowledge. The second student admitted to not feeling confident as they had not practiced Alloy in a while. Lastly, both students had experience with Alloy4Fun.

Table 5.5 shows the results of the first challenge for the first student. Column "Solved" indicates whether the student solved the predicate (Yes) or not (No), or "First Try" if the student completed the predicate without any incorrect submissions. Column "Used hints" indicates whether the student generated any hints (Yes) or not (No). Column "Time" indicates the time in minutes spent by the student to complete the challenge. Column "Generated hints" indicates the number of hints generated by the student for this challenge.

This student was able to complete all predicates under the time limit, despite not being required to do so. This is a good indicator that the student was motivated to solve the challenge, which was then confirmed by the student's answer to the form question "How motivated are you to continue practicing Alloy in Alloy4Fun?", which was 5 out of 5. This student also reported that the challenge was easy, scoring 2 out of 5 in the form question "How difficult was the challenge?".

In total, the student generated 13 hints (9 unique hints), which is a good indicator that the student was able to use HiGenA effectively. In the end, the student used HiGenA to complete 3 out of 8 predicates, which is less than half of the predicates. The reason for this is that the student was able to complete 5 predicates on the first try. However, whenever the student failed to complete a predicate on the first try, they always requested a hint from HiGenA, which shows that the student was interested in receiving help from hints.

Predicate	Solved	Used hints	Time (min)	Generated hints
undirected	Yes	Yes	17:30	13
oriented	First Try	No		
acyclic	First Try	No		
complete	First Try	No		
noLoops	Yes	Yes		
weaklyConnected	Yes	Yes		
stonglyConnected	First Try	No		
transitive	First Try	No		

Table 5.5: Results from the first challenge.

When asked about the quality of the hints, the student answered 4 out of 5 in the form question "How satisfied were you with the quality of the generated hints?". The student also answered that the provided hints were somewhat relevant to the predicate they were trying to solve in the form question "Were the provided hints relevant?".

The second student solved this challenge without the hint feature enabled. They completed 4 out of the 8 predicates. They tried to solve the other 4 predicates, but failed to debug their erroneous specifications and gave up before the time limit expired. The student reported that the challenge was a 3 out of 5 in difficulty and that it could have been easier if they had studied Alloy before the day of the study.

The student also reported that they were not feeling very motivated to continue practicing Alloy in Alloy4Fun, scoring 2 out of 5 in the form question "How motivated are you to continue practicing Alloy in Alloy4Fun?". When we asked why they were not feeling motivated, the student answered that they feel counter-examples provided by Alloy4Fun are not very helpful for debugging specifications, making it frustrating to understand what is wrong with their model. Dealing with this problem is the main motivation behind the development of HiGenA.

For the second challenge, the first student did not get to use hints to solve the challenge. Out of 7 predicates, the student solved 5 of them and exceeded the time limit by 5 minutes. They rated the challenge a 5 out of 5 in terms of difficulty. Despite not being able to complete the challenge, the student reported that they were still very motivated to continue practicing Alloy in Alloy4Fun, scoring 5 out of 5 on the motivation scale.

In the rest of the form questions, the student reported that using Alloy4Fun with the help of hints increased their confidence, speed, and performance when solving the challenge. The student also felt more engaged and reported that the hints generated enhanced their understanding of Alloy concepts. Although the student had forgotten about some Alloy concepts, some hints helped them remember those concepts thanks to the descriptions provided by HiGenA.

Lastly, the student recommends the use of HiGenA to other students and prefers to use Alloy4Fun with the help of hints. In the end, the student commented that the hints rarely helped, but when they did, which is usually when the student was near a solution, they were very helpful. They also said that the hints were intuitive and easy to understand.

Meanwhile, the second student solved the challenge using hints. The results are shown in

Predicate	Solved	Used Hints	Time (min)	Generated hints
inv1	First Try	No	30	12
inv2	First Try	No		
inv3	No	Yes		
inv4	No	No		
inv5	No	Yes		
inv6	No	No		
inv7	-	No		

Table 5.6: Results from the second challenge.

Table 5.6. Out of the seven predicates, the student was able to solve 2 of them on the first try and failed to solve the rest. The student reached the time limit, so they did not get to try to solve the last predicate, *inv7*.

Unfortunately, they solved the first two predicates on the first try and failed to write an expression without syntax errors for predicate *inv4* and *inv6*, which left them with only two predicates to generate hints. However, even with the help of hints, the student was not able to solve these two predicates. In total, they generated 12 hints, but 6 of these were repeated messages.

The student reported that the challenge was a 4 out of 5 in difficulty because they often struggle with ternary relations. However, their motivation increased compared to the first challenge, scoring 3 out of 5 in the form question. Regarding hints, the student claimed that generated hints were not relevant at all and that they were not satisfied with the quality of the hints, scoring 3 out of 5 in the quality question. The hints did not increase their confidence, speed, or performance. Moreover, the hints did not enhance their understanding of Alloy concepts.

Nonetheless, when questioned whether they would recommend the use of HiGenA to other students, the student answered "neutral" instead of "no" because they believe that HiGenA has the potential to become a useful tool for students. Additionally, when questioned whether they would prefer to use Alloy4Fun with or without the help of hints, the student answered that they prefer to use Alloy4Fun with the help of hints because they believe that hints can be helpful to open the student's mind to new solutions.

As a final comment, the student said that they felt like the second challenge was much more complex than the first one and should have been similar in difficulty. They also noted that although the hint messages were easy to understand, they would prefer for the hint messages to be more specific on what is wrong with the student's expression. Most of the time, the student felt that their expression was correct, but neither the counter-examples nor the hint message helped them understand the problem with their specification.

5.4.2.3 Analysis of the Student Feedback

It is important to note that the results of this study are not statistically significant due to the small number of participants. However, it is a preliminary study that will be used to plan future studies with more participants.

Moreover, some circumstances might have influenced the results of the study. The students were using a computer that was not their own, which might have made them uncomfortable. Also, the researchers observed and recorded the students during the study, which might have made them nervous. Lastly, both challenges were timed, which might have pressured the students to solve the challenges as fast as possible. However, we can still get some insights from the results.

When we compare the number of predicates left unsolved by the students, we can see that the first student solved more predicates when using hints than when they were not using HiGenA (8 vs. 5). However, the opposite happened with the second student (4 vs. 2). This is probably due to the second challenge being much more complex than the first one and not necessarily because of the use of hints.

On the other hand, both students agreed that they would prefer to use Alloy4Fun with the help of hints. This might be an indication that students prefer to have the option to get help when necessary, even though the second student never found any of the hints helpful.

Another interesting result is that students rarely find the hints helpful when they are far from a solution. The first student only found hints helpful whenever they were close to a solution. The second student only got to experience hints generated when they were far away from a solution. Therefore, they also never found hints to be helpful to them.

This makes sense because when a student is far from a solution, a simple the operation to the AST of their expression still results in an incorrect expression, which might still be very different from the solution. Additionally, HiGenA might generate one hint based on an operation that, when applied to the expression's AST results in an invalid expression if the student does not make additional changes.

To illustrate this, consider the following example that happened during the study. The second student was trying to solve predicate *inv6* and wrote the following expression: `all s1:State, s2:State, e:Event, s3:State, s4:State | s1→e→s3 in trans and s2→e→s4 in trans implies s1=s2`. The generated hint pointed out that the student should delete the "in" keyword from their expression. However, if the student were to delete only the "in" keyword, the resulting expression would be invalid. Besides, the student was confused as to which one of the two "in" keywords the hint was referring to. Thus, the student ended up ignoring the hint.

Another important aspect mentioned by the second student was that the HiGenA focuses on helping the student solve the challenge. However, they feel it would be more helpful if hints were more informative on what is wrong with their expression. As the student mentioned, their main difficulty is debugging their expression. Therefore, if hints help the student identify their mistake, then, they can use their own knowledge to repair the expression by themselves. This is an essential point that we should consider in future work.

Moreover, both students agreed that the language used in the hint messages was easy to understand. Mainly the use of keywords such as "delete", "add", "replace" and "move" made it clear to the students what action the hint suggested. The use of description templates also helped the students understand the hints. Finally, both the length of the hint messages and the time it took to generate the hints were also considered appropriate by the students.

5.4.3 Teacher Study

The teacher study aims to compare the hints generated by HiGenA with the hints given by a teacher and to get feedback from the teacher regarding the quality of the hints generated by HiGenA.

5.4.3.1 Teacher Study Methodology

We collected the top 10 most popular incorrect submissions for both challenges used in the student study. Then, we ask the teacher to write a hint message for each of the incorrect submissions. We indicate that the hint should be written in the form of a single mutation. After that, we show the hint messages generated by HiGenA for the same incorrect submissions and ask the teacher to give feedback on the quality of the hint. The form used to collect the teacher's feedback is available in Appendix A.2.

5.4.3.2 Preliminary Teacher Study Results and Analysis

The teacher study was conducted with one participant. The participant in this study was a teacher from the University of Minho with experience in teaching Alloy. Although the number of participants is small, it is a preliminary study that will be used to plan future studies with more participants.

Of the ten hints written by the teacher, four were equal to the hints generated by HiGenA, five were different, and one was similar. Note that we consider two hints to be equal if they suggest exactly the same operation to the AST of the student's expression and similar if they address the same problem with the student's expression but suggest different operations. For example, the similar hint written by the teacher was "Change ^ to *", and the hint generated by HiGenA was to add the * operator under the dot join operator expression. Both hints suggest that the student should use the * operator but represent different types of operations to the AST.

The feedback given by the teacher regarding the quality of the hints generated by HiGenA was positive. The teacher said that five out of the ten hints were good. All of the hints that the teacher considered good were also the ones that were equal or similar to the hints written by the teacher.

The teacher also mentioned that one of the hints did not target the main problem with the student's expression. This makes sense since HiGenA does not prioritize any operation in the edit script over the others, which might generate irrelevant hints. Despite this, the teacher still considered the hint somewhat helpful, as it targets a more insignificant problem with the student's expression.

Another hint was considered to be confusing by the teacher. This hint represents a move operation: "One step away from the solution! It seems like the field adj is not in the right place. Try moving it to the inside of the dot join operator ('.') expression. Try moving it so that you correctly ensure the required property.". This can indicate that the hint messages suggesting move operations should be improved.

Moreover, the teacher also mentioned that one of the generated hints was good in terms of quality but was trying to guide the student to a solution that was not the one intended by the teacher. The teacher believes the tool should guide the student toward a better solution.

Upon generating the hints used in this study, HiGenA considers the best solution to be the nearest one in the graph of student submissions. However, there might be a discrepancy between what HiGenA believes is the best solution and what the teacher considers the solution the student should aim for. Although HiGenA can use other metrics to determine the best solution, it is not clear which metric would be the most similar to the teacher's opinion. Therefore, this is an aspect that we should look into in future work.

Finally, the last two hints were considered to be unhelpful by the teacher without any further explanation. Both these hints were very different from the hints written by the teacher. While the teacher's hints indicated that the student should change the composition of the expression, the hints generated by HiGenA suggested that the student should replace the dot join operator with the equality operator. Nonetheless, it is unclear what aspect of these hints could be improved without additional information.

5.5 Conclusion

In this chapter, we looked into the several parts of HiGenA that influence the overall quality of the tool.

We started by evaluating the tool's effectiveness by trying to understand the impact of canonicalization techniques on the graph of student submission. We concluded that the effect is relatively small, reducing the number of unique states in the graph by less than 12%. However, the canonicalization techniques still positively impact the quality of the hints generated by HiGenA, leading us to believe they are worth keeping. However, in the future, we might want to look into other canonicalization techniques that we can apply to Alloy ASTs to reduce the number of unique states in the graph.

Next, we looked into how HiGenA uses historical student submissions from Alloy4Fun to generate hints. The results show that most of the time, the student submission does not exist in the graph yet. However, we believe this result will change as more data is collected. Moreover, we found that HiGenA only creates new paths present on the graph 29.6% of the time, which means that HiGenA tends to take advantage of student solving process to generate hints.

Additionally, we evaluated the performance and efficiency of HiGenA to compare it with TAR, the state-of-the-art tool for repairing Alloy specifications. TAR could only fix 56% of the erroneous specifications in under 1 minute, while it failed to fix 46% on time either to time-out or to an exhaustion of the search space. Meanwhile, HiGenA generates hints 100% of the time, as long as a solution exists on the graph, and, on average, it takes 64.03ms to generate a hint, outperforming TAR by a large margin.

In the next experiments, we focused on evaluating the quality of the hints generated by HiGenA. First, we evaluated the two state-of-the-art algorithms for finding mappings between nodes in ASTs based on the resulting length of the edit script. The results were conflicting. Most of the time, both algorithms perform similarly, but there are cases where one algorithm outperforms the other. Thus, we decided to keep both algorithms available in HiGenA, and we will leave the task of finding which algorithm is better for each case to future work.

Finally, we conducted two pilot studies to evaluate the quality of the hints generated by HiGenA, one with students and one with a teacher. Both studies had a small number of participants, and the results were not statistically significant. However, we believe that the results are still a good indication of the quality of the hints generated by HiGenA.

The results of the student study show that students only found hints useful when they were close to a solution. Additionally, the second student mentioned that they would prefer to have more informative hints that would help them identify their mistakes instead of hints that focus on helping them solve the challenge.

The results of the teacher study show that the teacher found most of the hints helpful and sometimes, there was no difference between the hints generated by HiGenA and the hints written by the teacher. However, some of the generated hints differ from the hints given by the teacher in some cases. Thus, we believe that HiGenA can be used as a complementary tool to help teachers give feedback to their students.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The goal of this document was to present the field of automated data-driven hint generation for programming education and our solution to the problem of generating hints for the Alloy language.

In the first chapter of this document, Chapter 1, we present the context and motivation, the problem to solve, and the document's structure. Following this, Chapter 2 presents the necessary knowledge to understand the rest of the document, including an overview of the Alloy language and the Alloy4Fun platform.

Chapter 3 includes the systematic literature review of the state-of-the-art tools for hint generation. This Chapter presents the systematic literature review process taken and the results of the search. These results include a description of the hint-generation techniques found and the results of the evaluation of these tools.

Chapter 4 details the proposed solution, HiGenA. It includes the solution conception and the implementation details.

Finally, Chapter 5 presents the evaluation of the developed solution. This evaluation includes evaluating several aspects of HiGenA, including the evaluation of the generated hints and the evaluation of the tool's performance.

In conclusion, we can say that we successfully achieved the goals of this work. We designed and implemented an effective data-driven hint-generation tool using historical student submissions from the Alloy4Fun platform.

It was a challenging task due to the nature of the data available. The focus of this project was to generate hints that would help students reach a solution. However, the submissions from Alloy4Fun were created by students that were still learning the Alloy language. Therefore, the dataset contained many incorrect submissions, which impacted the resulting graphs of data. Nevertheless, we successfully achieved the goal by developing HiGenA with very positive results, outperforming the state-of-the-art tool for repairing Alloy specifications.

Lastly, some aspects of the solution could still be improved in future work. From the evaluation section, we were able to identify some of these aspects. In terms of effectiveness, we should improve the canonicalization process to increase the number of hits in the graph. Additionally, we still need to conduct further user studies to get more insights into the quality of the generated hints. Either way, generating the best hints possible is a challenging task that is outside the scope of this work, and it is still an open problem in the field of automated data-driven hint generation.

6.2 Future Work

From the evaluation section, we were able to pin down some aspects of the solution that can be improved. These aspects are related to the effectiveness of the hint-generation algorithm and the quality of the generated hints.

To improve the effectiveness of HiGenA, we should start by improving the canonicalization process. In the evaluation section, we discussed that the results achieved by the canonicalization process were not as good as expected. The canonicalization techniques are essential to improve the quality of the generated hints and to increase the number of hits in the graph. Therefore, we should study other canonicalization techniques from the literature and choose the best ones for the Alloy ASTs.

Another interesting addition to improving the effectiveness of HiGenA would be to generate hints for syntactically incorrect submissions. At the time of writing, HiGenA only generates hints for semantically incorrect submissions using the Alloy parser to obtain the AST of a submission. However, we should study research techniques for generating hints for syntactically incorrect submissions in the future.

Next, we should study other criteria for the hint-generation algorithm to improve the quality of the generated hints. The default algorithm is based on the TED criterion, a good criterion for finding the optimal path between two ASTs. However, we have implemented other measures for pathfinding in HiGenA HiGenA that have not been evaluated. These measures are based on the submissions' popularity and transitions' popularity. We should evaluate these criteria to see if they improve the quality of the generated hints.

Similarly, we need to look further into the results by comparing the two mapping algorithms employed by HiGenA. Depending on the case, each algorithm achieves better results than the other. However, we have yet to find a way to detect which algorithm is better for each case to improve the quality of the generated hints.

Another improvement related to edit scripts would be to merge some of the operations in the edit script. Sometimes, the edit script contains an *insert* and *delete* operation for the same node in different positions, which could be simplified with a move operation. This problem is not a big issue, but it could positively influence the quality of the generated hints.

Additionally, in the user study with students, the main complaint was that hints were only helpful when the student was close to the solution. When the student is far from the solution, the generated edit script contains many operations. Therefore, a solution to this problem would be to aggregate multiple operations into a single hint message. This could be done by grouping operations that are related to each other or by detecting patterns of operations that are often applied together. However, this is not a trivial task and would require further research.

The last critique of the generated hints, which requires our attention, was that they often failed to help the student understand the problem of their solution. Even though the generated hint allows the student to reach a solution, the student still does not understand why their solution was incorrect. Therefore, we should explore other ways of communicating the problem of the student's solution through a hint based on the edit script.

Finally, before working on these improvements, we must focus on acquiring more participants for both user studies we conducted to evaluate the solution further. With more participants, we could perform a more in-depth analysis of the results and draw more accurate conclusions to prioritize what to improve on our solution first.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[2] Richard J. Anderson and Heather Woll. Wait-free parallel algorithms for the union-find problem. In Cris Koutsougeras and Jeffrey Scott Vitter, editors, *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 370–380. ACM, 1991.

[3] Nikolaus Augsten and Mateusz Pawlik. Compare your trees with ease. tree-edit-distance.dbresearch.uni-salzburg.at/ (accessed 2023-01-22).

[4] Tiffany Barnes and John C. Stamper. Toward automatic hint generation for logic proof tutoring using historical student data. In Beverly Park Woolf, Esma Aïmeur, Roger Nkambou, and Susanne P. Lajoie, editors, *Intelligent Tutoring Systems, 9th International Conference, ITS 2008 Proceedings*, volume 5091 of *Lecture Notes in Computer Science*, pages 373–382. Springer, 2008.

[5] Simón Gutiérrez Brida, Germán Regis, Guolong Zheng, Hamid Bagheri, ThanhVu Nguyen, Nazareno Aguirre, and Marcelo F. Frias. Beafix: An automated repair tool for faulty Alloy models. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021*, pages 1213–1217. IEEE, 2021.

[6] Simón Gutiérrez Brida, Germán Regis, Guolong Zheng, Hamid Bagheri, ThanhVu Nguyen, Nazareno Aguirre, and Marcelo F. Frias. Bounded exhaustive search of Alloy specification repairs. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021*, pages 1135–1147. IEEE, 2021.

[7] Peter Brusilovsky and Sergey A. Sosnovsky. Individualized exercises for self-assessment of programming knowledge: An evaluation of quizpack. *ACM J. Educ. Resour. Comput.*, 5(3):6, 2005.

[8] Chenghao Cai, Jing Sun, and Gillian Dobbie. Automatic B-model repair using model checking and machine learning. *Autom. Softw. Eng.*, 26(3):653–704, 2019.

[9] Jorge Cerqueira, Alcino Cunha, and Nuno Macedo. Timely specification repair for Alloy 6. In Bernd-Holger Schlingloff and Ming Chai, editors, *Software Engineering and Formal Methods - 20th International Conference, SEFM 2022, Berlin, Germany, September 26-30, 2022, Proceedings*, volume 13550 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2022.

[10] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, 1996*, pages 493–504. ACM Press, 1996.

[11] Robert Clarisó and Jordi Cabot. Fixing defects in integrity constraints via constraint mutation. In Antonia Bertolino, Vasco Amaral, Paulo Rupino, and Marco Vieira, editors, *11th International Conference on the Quality of Information and Communications Technology, QUATIC 2018*, pages 74–82. IEEE Computer Society, 2018.

[12] Albert T. Corbett and John R. Anderson. Locus of feedback control in computer-based tutoring: impact on learning rate, achievement and attitudes. In Julie A. Jacko and Andrew Sears, editors, *Proceedings of the CHI 2001 Conference on Human Factors in Computing Systems, 2001*, pages 245–252. ACM, 2001.

[13] Stephen Cummins, Alistair Stead, Lisa Jardine-Wright, Ian Davies, Alastair R. Beresford, and Andrew C. Rice. Investigating the use of hints in online problem solving. In Jeff Haywood, Vincent Aleven, Judy Kay, and Ido Roll, editors, *Proceedings of the Third ACM Conference on Learning @ Scale, L@S 2016*, pages 105–108. ACM, 2016.

[14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher, editors, *ACM/IEEE International Conference on Automated Software Engineering, ASE '14*, pages 313–324. ACM, 2014.

[15] Jeremiah T. Folsom-Kovarik, Sae Lynne Schatz, and Denise M. Nicholson. Plan ahead : Pricing ITS learner models. 2010.

[16] Davide Fossati, Barbara Di Eugenio, Stellan Ohlsson, Christopher W. Brown, Lin Chen, and David G. Cosejo. I learn from you, you learn from me: How to make iList learn from students. In Vania Dimitrova, Riichiro Mizoguchi, Benedict du Boulay, and Arthur C. Graesser, editors, *Artificial Intelligence in Education: Building Learning Systems that Care: From Knowledge Representation to Affective Modelling, Proceedings of the 14th International Conference on Artificial Intelligence in Education, AIED 2009*, volume 200 of *Frontiers in Artificial Intelligence and Applications*, pages 491–498. IOS Press, 2009.

[17] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Trans. Software Eng.*, 45(1):34–67, 2019.

[18] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L. Thomas van Binsbergen. Ask-Elle: an adaptable programming tutor for haskell giving automated feedback. *Int. J. Artif. Intell. Educ.*, 27(1):65–100, 2017.

[19] Andrew Head, Elena L. Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. Writing reusable code feedback at scale with mixed-initiative program synthesis. In Claudia Urrea, Justin Reich, and Candace Thille, editors, *Proceedings of the Fourth ACM Conference on Learning @ Scale, L@S 2017*, pages 89–98. ACM, 2017.

[20] Andrew Hicks, Barry W. Peddycord III, and Tiffany Barnes. Building games to learn from their players: Generating hints in a serious game. In Stefan Trausan-Matu, Kristy Elizabeth Boyer, Martha E. Crosby, and Kitty Panourgia, editors, *Intelligent Tutoring Systems*

- *12th International Conference, ITS 2014, Proceedings*, volume 8474 of *Lecture Notes in Computer Science*, pages 312–317. Springer, 2014.

[21] Jun Hong. Guided programming and automated error analysis in an intelligent Prolog tutor. *Int. J. Hum. Comput. Stud.*, 61(4):505–534, 2004.

[22] I-Han Hsiao, Sergey A. Sosnovsky, and Peter Brusilovsky. Adaptive navigation support for parameterized questions in object-oriented programming. In Ulrike Cress, Vania Dimitrova, and Marcus Specht, editors, *Learning in the Synergy of Multiple Disciplines, 4th European Conference on Technology Enhanced Learning, EC-TEL 2009, Proceedings*, volume 5794 of *Lecture Notes in Computer Science*, pages 88–98. Springer, 2009.

[23] Gregory Hume, Joel Michael, Allen Rovick, and Martha Evens. Hinting as a Tactic in One-on-One Tutoring. *Journal of The Learning Sciences - J LEARN SCI*, 5:23–47, 1996.

[24] Wen-mei W. Hwu and Pohua P. Chang. Inline function expansion for compiling C programs. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI), 1989*, pages 246–257. ACM, 1989.

[25] Marvin J. Croy. Graphic Interface Design and Deductive Proof Construction. https://pages.charlotte.edu/marvin-croy/publications/jcmst/ (accessed 2023-01-03), 2013.

[26] Daniel Jackson. Alloy tools FAQ: How has Alloy been used in teaching? https://alloytools.org/faq/how_has_alloy_been_used_in_teaching.html (accessed: 2023-01-01).

[27] Daniel Jackson. *Software abstractions: logic, language, and analysis*. The MIT Press, revised edition edition, 2016.

[28] Wei Jin, Tiffany Barnes, John C. Stamper, Michael John Eagle, Matthew W. Johnson, and Lorrie Lehmann. Program representation for automatic hint generation for a data-driven novice programming tutor. In Stefano A. Cerri, William J. Clancey, Giorgos Papadourakis, and Kitty Panourgia, editors, *Intelligent Tutoring Systems - 11th International Conference, ITS 2012, Proceedings*, volume 7315 of *Lecture Notes in Computer Science*, pages 304–309. Springer, 2012.

[29] Khalid S Khan, Regina Kunz, Jos Kleijnen, and Gerd Antes. Five steps to conducting a systematic review. *Journal of the Royal Society of Medicine*, 96(3):118–121, 2003.

[30] Timotej Lazar and Ivan Bratko. Data-driven program synthesis for hint generation in programming tutors. In Stefan Trausan-Matu, Kristy Elizabeth Boyer, Martha E. Crosby, and Kitty Panourgia, editors, *Intelligent Tutoring Systems - 12th International Conference, ITS 2014, Proceedings*, volume 8474 of *Lecture Notes in Computer Science*, pages 306–311. Springer, 2014.

[31] Nguyen-Thinh Le. A classification of adaptive feedback in educational systems for programming. *Syst.*, 4(2):22, 2016.

[32] Nuno Macedo, Alcino Cunha, and Ana C. R. Paiva. Alloy4fun dataset for 2022/23, July 2023.

[33] Nuno Macedo, Alcino Cunha, José Pereira, Renato Carvalho, Ricardo Silva, Ana C. R. Paiva, Miguel Sozinho Ramalho, and Daniel Castro Silva. Experiences on teaching Alloy with an automated assessment platform. In Alexander Raschke, Dominique Méry, and Frank Houdek, editors, *Rigorous State-Based Methods - 7th International Conference, ABZ 2020, Proceedings*, volume 12071 of *Lecture Notes in Computer Science*, pages 61–77. Springer, 2020.

[34] Yana Malysheva and Caitlin Kelleher. An algorithm for generating explainable corrections to student code. In Ilkka Jormanainen and Andrew Petersen, editors, *Koli Calling '22: 22nd Koli Calling International Conference on Computing Education Research*, pages 13:1–13:11. ACM, 2022.

[35] Jessica McBroom, Irena Koprinska, and Kalina Yacef. A survey of automated programming hint generation: The HINTS framework. *ACM Comput. Surv.*, 54(8):172:1–172:27, 2022.

[36] Chanin Nantasenamat. The Data Science Process. `https://towardsdatascience.com/the-data-science-process-a19eb7ebc41b` (accessed 2023-01-27).

[37] Alberto Nuñez, Javier Fernández, José Daniel García, Laura Prada, and Jesús Carretero. M-PLAT: multi-programming language adaptive tutor. In *The 8th IEEE International Conference on Advanced Learning Technologies, ICALT 2008*, pages 649–651. IEEE Computer Society, 2008.

[38] Jonathan S. Ostroff. Formal methods for the specification and design of real-time safety critical systems. *J. Syst. Softw.*, 18(1):33–60, 1992.

[39] Laura Pappano. The Year of the MOOC. *The New York Times*, 2012.

[40] Mateusz Pawlik and Nikolaus Augsten. RTED: A robust algorithm for the tree edit distance. *Proc. VLDB Endow.*, 5(4):334–345, 2011.

[41] Mateusz Pawlik and Nikolaus Augsten. Efficient computation of the tree edit distance. *ACM Trans. Database Syst.*, 40(1):3:1–3:40, 2015.

[42] Mateusz Pawlik and Nikolaus Augsten. Tree edit distance: Robust and memory-efficient. *Inf. Syst.*, 56:157–173, 2016.

[43] Yu Pei, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Automatic program repair by fixing contracts. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Proceedings*, volume 8411 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 2014.

[44] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas J. Guibas. Autonomously generating hints by inferring problem solving policies. In Gregor Kiczales, Daniel M. Russell, and Beverly P. Woolf, editors, *Proceedings of the Second ACM Conference on Learning @ Scale, L@S 2015*, pages 195–204. ACM, 2015.

[45] Leena M. Razzaq and Neil T. Heffernan. Hints: Is it better to give or wait to be asked? In Vincent Aleven, Judy Kay, and Jack Mostow, editors, *Intelligent Tutoring Systems, 10th International Conference, ITS 2010, Proceedings, Part I*, volume 6094 of *Lecture Notes in Computer Science*, pages 349–358. Springer, 2010.

[46] Kelly Rivers. *Automated Data-Driven Hint Generation for Learning Programming*. PhD thesis, Carnegie Mellon University, USA, 2017.

[47] Kelly Rivers and Kenneth R. Koedinger. Automating hint generation with solution space path construction. In Stefan Trausan-Matu, Kristy Elizabeth Boyer, Martha E. Crosby, and Kitty Panourgia, editors, *Intelligent Tutoring Systems - 12th International Conference, ITS 2014, Proceedings*, volume 8474 of *Lecture Notes in Computer Science*, pages 329–339. Springer, 2014.

[48] Steven C. Shaffer. Ludwig: an online programming tutoring and assessment system. *ACM SIGCSE Bull.*, 37(2):56–60, 2005.

[49] Jennifer Spencer. Safety-Critical System. `https://www.technipages.com/definition/safety-critical-system` (accessed 2023-01-25).

[50] Michael Sutton, Tal Ben-Nun, and Amnon Barak. Optimizing parallel graph connectivity computation via subgraph sampling. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018*, pages 12–21. IEEE Computer Society, 2018.

[51] Richard S. Sutton and Andrew G. Barto. Reinforcement learning: An introduction. *IEEE Trans. Neural Networks*, 9(5):1054–1054, 1998.

[52] Elizabeth S. Veinott and Elizabeth T. Whitaker. Leaving hints: Using player in-game hints to measure and improve learning. In Constantine Stephanidis and Margherita Antona, editors, *HCI International 2019 - Late Breaking Posters - 21st HCI International Conference, HCII 2019, Proceedings*, volume 1088 of *Communications in Computer and Information Science*, pages 222–230. Springer, 2019.

[53] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. Automated model repair for Alloy. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018*, pages 577–588. ACM, 2018.

[54] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. ARepair: a repair framework for Alloy. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019*, pages 103–106. IEEE / ACM, 2019.

[55] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. Fault localization for declarative models in Alloy. In Marco Vieira, Henrique Madeira, Nuno Antunes, and Zheng Zheng, editors, *31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020*, pages 391–402. IEEE, 2020.

[56] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.

[57] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. pages 38:1–38:10, 2014.

[58] Guolong Zheng, ThanhVu Nguyen, Simón Gutiérrez Brida, Germán Regis, Marcelo F. Frias, Nazareno Aguirre, and Hamid Bagheri. FLACK: counterexample-guided fault localization for Alloy models. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021*, pages 637–648. IEEE, 2021.

Appendix A

Appendix

A.1 Study with Students

Here we present the form filled out by students who participated in the study after using the hint-generation tool to solve an Alloy4Fun challenge. It contains three sections: the first one is about the student's background, the other two are about the student's experience with solving the challenge without and with the hint-generation tool, respectively.

HiGenA Study B

* Indica uma pergunta obrigatória

1. What is the last time you used Alloy? *

Marcar apenas uma oval.

- In the past 6 months
- In the past 12 months
- Last year
- More than 1 year ago
- Never

2. Are you familiar with Alloy? *

Marcar apenas uma oval.

- I'm really confortable with Alloy
- I know the basics
- I'm a little rusty
- I don't know Alloy

3. How did you learn Alloy?

Marcar apenas uma oval.

- I had classes with a teacher
- I learned by myself

4. Are you familiar with Alloy4fun? *

Marcar apenas uma oval.

- Yes
- No

Phase 1 - Before hints

Answer this section after solving the exercise without hints.

5. Rate the difficulty of this challenge. *

Marcar apenas uma oval.

	1	2	3	4	5	
Very						Very Hard

6. Imagine you have an exam. How motivated are you to continue practicing Alloy in Alloy4Fun? *

Marcar apenas uma oval.

	1	2	3	4	5	
Very						Very motivated

Phase 2 - After hints

Answer this section after solving the exercise with hints.

7. Rate the difficulty of this challenge. *

Marcar apenas uma oval.

	1	2	3	4	5	
Very	◯	◯	◯	◯	◯	Very Hard

8. Imagine you have an exam. How motivated are you to continue practicing Alloy in Alloy4Fun? *

Marcar apenas uma oval.

	1	2	3	4	5	
Very	◯	◯	◯	◯	◯	Very motivated

9. Were the provided hints relevant? *

Marcar apenas uma oval.

- ◯ Yes, the hints were highly relevant
- ◯ Yes, the hints were somewhat relevant
- ◯ Neutral
- ◯ No, the hints were not very relevant
- ◯ No, the hints were not relevant at all

10. How satisfied were you with the quality of the generated hints? *

Marcar apenas uma oval.

	1	2	3	4	5	
Very	◯	◯	◯	◯	◯	Very satisfied

11. Did the hints increase your confidence in solving the exercises? *

Marcar apenas uma oval.

- Yes, significantly
- Yes, to some extent
- Neutral
- No, not significantly
- No, not at all
- Outra: ____________________

12. Did the provided hints help you solve the exercises faster? *

Marcar apenas uma oval.

- Yes, significantly
- Yes, to some extent
- No, not significantly
- No, not at all
- Outra: ____________________

13. In your opinion, did using the hints improve your overall performance in solving the exercises? *

Marcar apenas uma oval.

- Yes, improved significantly
- Yes, improved to some extent
- Neutral
- No, did not improve significantly
- No, did not improve at all

14. Did you find it easier to stay engaged and not give up solving the exercise when using the hints? *

Marcar apenas uma oval.

- Yes, significantly easier
- Yes, somewhat easier
- Neutral
- No, somewhat difficult
- No, significantly difficult

15. Did the generated hints enhance your understanding of Alloy concepts? *

Marcar apenas uma oval.

- Yes, significantly
- Yes, to some extent
- Neutral
- No, not significantly
- No, not at all

16. Would you recommend using this hint generation tool to your classmates studying Alloy? *

Marcar apenas uma oval.

- Yes, definitely
- Neutral
- No, definitely not

17. Which approach do you prefer for solving the exercises? *

Marcar apenas uma oval.

- Solving without any hints
- Solving with hints
- No preference

18. Why?

19. Please provide any additional comments or feedback regarding the hints or your experience using them

A.2 Study with a Teacher

Here we present the form filled out by the teacher who participated in the study for the evaluation of HiGenA. The study aims to compare hints given by the teacher with hints generated by HiGenA and get feedback from the teacher about the generated hints. We collected the top 10 most popular incorrect submissions from two Alloy4Fun challenges. Then, for each of the submissions, we ask the teacher to provide a hint to help the student solve the challenge. After that, we ask the teacher to evaluate the hints generated by HiGenA for the same submission.

Hint Generation Evaluation

In this form, we will explore two challenges from Alloy4Fun. Specifically, we will focus on the top 10 most popular incorrect submissions from these challenges. The purpose of this study is to compare the hints generated by our tool with the hints provided by Alloy tutors. We also seek feedback from tutors regarding the hints generated by our tool.

This form consists of 10 questions, each comprising two parts:

1st Part:

You will be given an Alloy model, a requirement, and an incorrect Alloy expression written by a student. Your task is to provide a brief hint (maximum 4 lines) that can assist the student in correcting their expression and achieving a solution. The hints should be in the form of single mutations (e.g.: Try to change operator X to operator Y).

2nd Part:

You will receive the same Alloy model, requirement, and incorrect expression as before. However, this time, you will also be provided with a hint generated by our tool, which you need to evaluate and provide feedback on. Furthermore, we will present you with the solution that our hint generation tool considers to be the closest solution to the student's expression and is used to generate the hint. If you disagree with this solution, please include your ideal solution in your feedback.

* Indica uma pergunta obrigatória

1.1 Weakly connected graph

Consider the following Alloy model of a graph,
available in Alloy4Fun in this link.

```
/*
Each node as a set of outgoing edges, representing a directed graph without multiple
edged.
*/
sig Node {
    adj : set Node
}
```

Requirement:

- The graph is weakly connected, ie, it is possible to reach every node from every node ignoring edge direction.

Student expression:

- ```
  all n: Node | Node in n.^adj
  ```

1. Write the hint you would give to the student that wrote this expression. *
  ```

1.2 Weakly connected graph

Consider the same Alloy model of a graph,
available in Alloy4Fun in this link.

```
/*
Each node as a set of outgoing edges, representing a directed graph without multiple edged.
*/
sig Node {
	adj : set Node
}
```

Requirement:

- The graph is weakly connected, ie, it is possible to reach every node from every node ignoring edge direction.

Student expression:

- all n : Node | Node in n.^adj

Nearest solution found by our hint generation tool:

- all v : Node | Node in v.*(adj + ~adj)

Hint generated by our tool:

- Keep going! Consider adding a reflexive-transitive closure operator ('*') to get the reflexive-transitive closure of a relation. Think about how you can incorporate this within the dot join operator ('.') expression.

2. Please give some feedback on the hint generated by our tool. *

2.1 Weakly connected graph

Consider the same Alloy model of a graph,
available in Alloy4Fun in this link.

```
/*
Each node as a set of outgoing edges, representing a directed graph without multiple edged.
*/
sig Node {
	adj : set Node
}
```

Requirement:

- The graph is weakly connected, ie, it is possible to reach every node from every node ignoring edge direction.

Student expression:

- all n: Node | Node in n.^(adj + ~adj)

3. Write the hint you would give to the student that wrote this expression. *

2.2 Weakly connected graph

Consider the same Alloy model of a graph,
available in Alloy4Fun in this link.

```
/*
Each node as a set of outgoing edges, representing a directed graph without multiple edged.
*/
sig Node {
	adj : set Node
}
```

Requirement:

- The graph is weakly connected, ie, it is possible to reach every node from every node ignoring edge direction.

Student expression:

- all n: Node | Node in n.^(adj + ~adj)

Nearest solution found by our hint generation tool:

- all v : Node | Node in v.*(adj + ~adj)

Hint generated by our tool:

- One step away from the solution! Instead of using transitive closure operator ('^') to get the transitive closure of a relation, try using reflexive-transitive closure operator ('*') to get the reflexive-transitive closure of a relation.

4. Please give some feedback on the hint generated by our tool. *

3.1 Undirected graph

Consider the same Alloy model of a graph,
available in Alloy4Fun in this link.

```
/*
Each node as a set of outgoing edges, representing a directed graph without multiple edged.
*/
sig Node {
	adj : set Node
}
```

Requirement:

- The graph is undirected, ie, edges are symmetric.

Student expression:

- all n : Node | n in n.adj.~adj

5. Write the hint you would give to the student that wrote this expression. *

3.2 Undirected graph

Consider the same Alloy model of a graph,
available in Alloy4Fun in this link.

```
/*
Each node as a set of outgoing edges, representing a directed graph without multiple
edged.
*/
sig Node {
	adj : set Node
}
```

Requirement:

- The graph is undirected, ie, edges are symmetric.

Student expression:

- all n : Node | n in n.adj.~adj

Nearest solution found by our hint generation tool:

- adj = ~adj

Hint generated by our tool:

- Keep going! Instead of using dot join operator ('.') to perform a relational join between sets or relations, try using equal operator ('=') to specify that the left side is equal to the right side.

6. Please give some feedback on the hint generated by our tool. *

4.1 Undirected graph

Consider the same Alloy model of a graph,
available in Alloy4Fun in this link.

```
/*
Each node as a set of outgoing edges, representing a directed graph without multiple edged.
*/
sig Node {
	adj : set Node
}
```

Requirement:

- The graph is undirected, ie, edges are symmetric.

Student expression:

- adj.~adj in iden

7. Write the hint you would give to the student that wrote this expression. *

4.2 Undirected graph

Hint generated by our tool:
Consider the same Alloy model of a graph,
available in Alloy4Fun in this link.

```
/*
Each node as a set of outgoing edges, representing a directed graph without multiple
edged.
*/
sig Node {
	adj : set Node
}
```

Requirement:

- The graph is undirected, ie, edges are symmetric.

Student expression:

- adj.~adj in iden

Nearest solution found by our hint generation tool:

- adj = ~adj

Hint generated by our tool:

- Near a solution! Instead of using dot join operator ('.') to perform a relational join between sets or relations, try using equal operator ('=') to specify that the left side is equal to the right side.

8. Please give some feedback on the hint generated by our tool. *

5.1 Strongly connected graph

Hint generated by our tool:
Consider the same Alloy model of a graph,
available in Alloy4Fun in this link.

```
/*
Each node as a set of outgoing edges, representing a directed graph without multiple edged.
*/
sig Node {
	adj : set Node
}
```

Requirement:

- The graph is strongly connected, ie, it is possible to reach every node from every node considering edge direction.

Student expression:

-

  all n : Node | Node in n.^adj

9. Write the hint you would give to the student that wrote this expression. *

5.2 Strongly connected graph

Hint generated by our tool:
Consider the same Alloy model of a graph,
available in Alloy4Fun in this link.

```
/*
Each node as a set of outgoing edges, representing a directed graph without multiple edged.
*/
sig Node {
	adj : set Node
}
```

Requirement:

- The graph is strongly connected, ie, it is possible to reach every node from every node considering edge direction.

Student expression:

- all n : Node | Node in n.^adj

Nearest solution found by our hint generation tool:

- all n:Node | Node in n.*adj

Hint generated by our tool:

- One step away from the solution! Instead of using transitive closure operator ('^') to get the transitive closure of a relation, try using reflexive-transitive closure operator ('*') to get the reflexive-transitive closure of a relation.

10. Please give some feedback on the hint generated by our tool. *

6.1 Strongly connected graph

Hint generated by our tool:
Consider the same Alloy model of a graph,
available in Alloy4Fun in this link.

```
/*
Each node as a set of outgoing edges, representing a directed graph without multiple edged.
*/
sig Node {
	adj : set Node
}
```

Requirement:

- The graph is strongly connected, ie, it is possible to reach every node from every node considering edge direction.

Student expression:

- Node in Node.^adj

11. Write the hint you would give to the student that wrote this expression. *

6.2 Strongly connected graph

Hint generated by our tool:
Consider the same Alloy model of a graph,
available in Alloy4Fun in this link.

```
/*
Each node as a set of outgoing edges, representing a directed graph without multiple edged.
*/
sig Node {
	adj : set Node
}
```

Requirement:

- The graph is strongly connected, ie, it is possible to reach every node from every node considering edge direction.

Student expression:

- Node in Node.^adj

Nearest solution found by our hint generation tool:

- all n:Node | Node in n.*adj

Hint generated by our tool:

- Keep going! Instead of using transitive closure operator ('^') to get the transitive closure of a relation, try using reflexive-transitive closure operator ('*') to get the reflexive-transitive closure of a relation.

12. Please give some feedback on the hint generated by our tool. *

7.1 No loops graph

Hint generated by our tool:
Consider the same Alloy model of a graph,
available in Alloy4Fun in this link.

```
/*
Each node as a set of outgoing edges, representing a directed graph without multiple edged.
*/
sig Node {
	adj : set Node
}
```

Requirement:

- The graph contains no loops, ie, nodes have no transitions to themselves.

Student expression:

-

  all n : Node | n not in n.^adj

13. Write the hint you would give to the student that wrote this expression. *

7.2 No loops graph

Hint generated by our tool:
Consider the same Alloy model of a graph,
available in Alloy4Fun in this link.

```
/*
Each node as a set of outgoing edges, representing a directed graph without multiple edged.
*/
sig Node {
	adj : set Node
}
```

Requirement:

- The graph contains no loops, ie, nodes have no transitions to themselves.

Student expression:

- all n : Node | n not in n.^adj

Nearest solution found by our hint generation tool:

- all n : Node | n not in n.adj

Hint generated by our tool:

- One step away from the solution! It seems like the field/adj is not in the right place. Try moving it to the inside of the dot join operator ('.') expression. Try moving it so that you correctly ensure the required property.

14. Please give some feedback on the hint generated by our tool. *

8.1 LTS inv3

Hint generated by our tool:
Consider the following Alloy model of a labeled transition system, available in Alloy4Fun in this link.

```
/*
A labeled transition system (LTS) is comprised by States, a sub-set
of which are Initial, connected by transitions, here represented by
Events.
*/
sig State {
	trans : Event -> State
}
sig Init in State {}
sig Event {}
```

Requirement:

- The LTS is deterministic, ie, each state has at most a transition for each event.

Student expression:

-

  all y : State | lone y.trans

15. Write the hint you would give to the student that wrote this expression. *

8.2 LTS inv3

Hint generated by our tool:
Consider the following Alloy model of a labeled transition system, available in Alloy4Fun in this link.

```
/*
A labeled transition system (LTS) is comprised by States, a sub-set
of which are Initial, connected by transitions, here represented by
Events.
*/
sig State {
	trans : Event -> State
}
sig Init in State {}
sig Event {}
```

Requirement:

- The LTS is deterministic, ie, each state has at most a transition for each event.

Student expression:

- all y : State | lone y.trans

Nearest solution found by our hint generation tool:

- all s : State, e : Event | lone e.(s.trans)

Hint generated by our tool:

- Keep going! You can use variables to help specify the condition. Consider introducing a new variable "Event" to your expression using the universal quantifier ('all').

16. Please give some feedback on the hint generated by our tool. *

9.1 LTS inv3

Hint generated by our tool:
Consider the following Alloy model of a labeled transition system, available in Alloy4Fun in this link.

```
/*
A labeled transition system (LTS) is comprised by States, a sub-set
of which are Initial, connected by transitions, here represented by
Events.
*/
sig State {
	trans : Event -> State
}
sig Init in State {}
sig Event {}
```

Requirement:

- The LTS is deterministic, ie, each state has at most a transition for each event.

Student expression:

- all x : State, y : Event | lone x.trans.y

17. Write the hint you would give to the student that wrote this expression. *

9.2 LTS inv3

Hint generated by our tool:
Consider the following Alloy model of a labeled transition system, available in Alloy4Fun in this link.

```
/*
A labeled transition system (LTS) is comprised by States, a sub-set
of which are Initial, connected by transitions, here represented by
Events.
*/
sig State {
	trans : Event -> State
}
sig Init in State {}
sig Event {}
```

Requirement:

- The LTS is deterministic, ie, each state has at most a transition for each event.

Student expression:

- all x : State, y : Event | lone x.trans.y

Nearest solution found by our hint generation tool:

- all s: State, e: Event | lone ~(s.trans).e

Hint generated by our tool:

-

One step away from the solution! Consider adding a transpose operator ('~') to transpose a relation. Think about how you can incorporate this within the dot join operator ('.') expression.

18. Please give some feedback on the hint generated by our tool. *

10.1 LTS inv1

Hint generated by our tool:
Consider the following Alloy model of a labeled transition system, available in Alloy4Fun in this link.

```
/*
A labeled transition system (LTS) is comprised by States, a sub-set
of which are Initial, connected by transitions, here represented by
Events.
*/
sig State {
	trans : Event -> State
}
sig Init in State {}
sig Event {}
```

Requirement:

- The LTS does not contain deadlocks, ie, each state has at least a transition.

Student expression:

- some State.trans

19. Write the hint you would give to the student that wrote this expression. *

10.2 LTS inv 5

Hint generated by our tool:
Consider the following Alloy model of a labeled transition system, available in Alloy4Fun in this [link]().

```
/*
A labeled transition system (LTS) is comprised by States, a sub-set
of which are Initial, connected by transitions, here represented by
Events.
*/
sig State {
	trans : Event -> State
}
sig Init in State {}
sig Event {}
```

Requirement:

- The LTS does not contain deadlocks, ie, each state has at least a transition.

Student expression:

- some State.trans

Nearest solution found by our hint generation tool:

- all s : State | some s.trans

Hint generated by our tool:

- Keep going! You can use variables to help specify the condition. Consider introducing a new variable "State" to your expression using the universal quantifier ('all').

20. Please give some feedback on the hint generated by our tool. *

End of the form

Thank you for your time!