UNIKASSEL VERSITÄT | ELEKTROTECHNIK INFORMATIK

Toolgestütztes Lernen im Kontext der Berechenbarkeitstheorie: ein Backend-Prototyp zur automatisierten Überprüfung von Reduktionen

Bachelorarbeit

Eingereicht von: Kathrin Lehmann
Matrikelnummer: 35272036

Vorgelegt im: Fachgebiet Theoretische Informatik/Formale Methoden

Gutachter: Prof. Dr. Martin Lange
Prof. Dr. Albert Zündorf

Betreuer: Dr. Norbert Hundeshagen
M.Sc. Marco Sälzer

Eingereicht am: 21. Dezember 2022

Eigenständigkeitserklärung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

Kassel, den 21. Dezember 2022

Inhaltsverzeichnis

1 Einleitung **3**

2 Grundlagen **4**

3 Anforderungen und Aufbau der Anwendung **6**

3.1 Aufgaben der Backend-Anwendung 7

3.2 Kommunikation mit Datenbank und Frontend 9

3.3 Aufbau der Backend-Anwendung 12

4 Überprüfung der Reduktionseigenschaften **15**

4.1 Totalität 15

4.2 Eigenschaft $x \in A \Leftrightarrow f(x) \in B$ 23

5 Beispielübungen **26**

5.1 Independent Set auf Vertex Cover 26

5.2 Leerheit für NFA auf $\overline{\text{PATH}}$ 34

6 Fazit und Ausblick **38**

1 Einleitung

Reduktion ist eine wichtige Technik in der Informatik. Sie ist nützlich, in vielen Zusammenhängen anwendbar und, wenn man ihr das erste Mal begegnet, gar nicht so leicht zu verstehen. Viele Studierende der Informatik haben Schwierigkeiten damit, gegebene Reduktionen nachzuvollziehen und erst recht damit, eigene anzugeben. Das zeigt sich bei der Korrektur von Hausaufgaben und auch in der Punkteverteilung bei Klausuren. König, Pfeiffer-Bohnen und Schmeck bezeichnen den Reduktionsbegriff in „Theoretische Informatik - ganz praktisch" als „eine der schwersten Hürden" beim Einstieg in die Themen Berechenbarkeit und Komplexität [7].

Häufig auftretende Fehler deuten darauf hin, dass es neben den Schwierigkeiten einzelner Aufgaben grundsätzliche Probleme beim Anwenden der Technik gibt. Das wird deutlich, wenn beispielweise die Richtung der Reduktion nicht stimmt, wenn der Ein- oder Ausgabetyp nicht korrekt ist oder wenn nicht-totale Funktionen angegeben werden. Auch Lösungsversuche, die so grundsätzliche Fehler vermeiden, sind oft nicht korrekt. Sie funktionieren vielleicht für einige Standardfälle oder erfüllen nur eine Richtung der Biimplikation, die eine korrekte Reduktion auszeichnet.

Um eine Aufgabe zum Thema Reduktion zu lösen, muss ein Studierender mehr tun als einen Algorithmus anzuwenden. Es gibt kein Muster, keine feste Abfolge von Anweisungen, die ans Ziel führt. Jemand, der die Technik schon oft angewandt hat, wird Ähnlichkeiten zwischen den verschiedenen Fällen entdecken und für die Suche Strategien entwickeln. Doch für einen Anfänger ist es eine Herausforderung, das gelernte Konzept auf neue Aufgaben zu übertragen.

Aus didaktischer Sicht ist es hilfreich, theoretische Inhalte selbst zu üben. Die Arbeit mit konkreten Beispielen führt die Lernenden an das abstrakte Konzept heran. Im Arbeiten mit verschiedenen Fällen werden Muster erkennbar und die aktive Tätigkeit trägt dazu bei, dass sich der Lernstoff einprägt. Aus diesem Grund wünschen sich Studierende Beispiele und Aufgaben zum Üben. Eine Möglichkeit, Übungen zur Bearbeitung bereitzustellen und schnell und einfach Rückmeldungen zu Lösungsversuchen zu geben, sind Lerntools.

Es gibt also sowohl thematische als auch didaktische Gründe, das Erlernen von Reduktionen gezielt zu fördern. Daraus ergibt sich die forschungsleitende Frage dieser Arbeit: Wie kann man das Erlernen des Themas Reduktion toolbasiert unterstützen?

Unterstützen bedeutet in diesem Zusammenhang, den Lernenden Übungsaufgaben zur Verfügung zu stellen und konkretes und nützliches Feedback zu einem Lösungsversuch

zu liefern. Wenn dieser Vorschlag Fehler wie die oben beschriebenen enthält, soll die Anwendung diese möglichst klar benennen. Das Erkennen und Verstehen dieser Fehler ist die Grundlage für den Lernfortschritt.

2014 wurde von Creus, Fernández und Godoy [2] ein Lerntool vorgestellt, mit dem Studierende unter anderem Reduktionen zwischen NP-vollständigen Problemen trainieren können. In diesem Tool werden Reduktionen in einer neu eingeführten C-ähnlichen Programmiersprache formuliert. Die Studierenden müssen diese Sprache lernen, um Lösungsversuche angeben zu können.

Die vorliegende Arbeit stellt das Backend eines Tools vor, das Studierende beim Erlernen von Reduktionen unterstützen soll. Es handelt sich dabei um einen Protoyp. Prüfungen aller Reduktionseigenschaften sind implementiert, können aber noch verbessert, erweitert oder ersetzt werden. Die Reduktionen werden als Python-Programme formuliert. Dadurch müssen Studierende, die bereits über Kenntnisse in Python verfügen, sich nicht in eine neue Sprache einarbeiten, um die Aufgaben lösen zu können.

Die Arbeit ist wie folgt aufgebaut: In Kapitel 2 werden zunächst die theoretischen Grundlagen der Arbeit erläutert. Kapitel 3 befasst sich mit der Implementierung in ihrer Gesamtheit. Hier wird beschrieben, wie das Backend selbst aufgebaut ist und wie die Kommunikation mit Datenbank und Frontend geregelt ist. Kapitel 4 befasst sich mit der Überprüfung der Reduktionseigenschaften. Anschließend werden in Kapitel 5 zwei Beispiele für mögliche Übungen vorgestellt. Zum Abschluss fasst Kapitel 6 die Ergebnisse der Arbeit zusammen und zeigt Ansätze für die Erweiterung des Tools auf.

2 Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen des Tools erläutert. Im Mittelpunkt steht dabei der Begriff der Reduktion. Die folgenden Definitionen orientieren sich an „Theoretische Informatik - kurz gefasst“ von Schöning [10] sowie „Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie“ von Hopcroft, Motwani und Ullman [6].

Ein *Alphabet* ist eine endliche, nicht leere Menge von Symbolen. Es wird durch das Symbol Σ dargestellt. Ein $Wort$ ist eine endliche Folge von Symbolen eines bestimmten Alphabets. Die Menge aller Wörter über einem Alphabet Σ wird durch Σ^* bezeichnet. Eine (formale) *Sprache* (über Σ) ist jede beliebige Teilmenge von Σ^*.

Wenn Σ ein Alphabet ist und L eine Sprache über Σ, dann lautet das *Entscheidungsproblem* L:

Gegeben: ein Wort $w \in \Sigma^*$.
Frage: Ist $w \in L$?

Anstelle von Wörtern spricht man in diesem Zusammenhang von Instanzen eines Problems. Ist $w \in L$, wird es auch als positive Instanz bezeichnet. Negative Instanzen sind $w \notin L$. Für Aussagen über die Lösbarkeit eines Problems definiert man den Begriff der Entscheidbarkeit.

Definition 1. *Ein Problem $A \subseteq \Sigma^*$ heißt entscheidbar, falls die charakteristische Funktion von A, nämlich $\chi_A : \Sigma^* \to \{0,1\}$, berechenbar ist. Hierbei ist für alle w aus Σ^*:*

$$\chi_A(w) = \begin{cases} 1, & w \in A \\ 0, & w \notin A \end{cases}$$

Es gibt in diesem Fall also einen Algorithmus, der immer stoppt und das Entscheidungsproblem löst. Einen solchen Algorithmus bezeichnet man als Entscheidungsverfahren von A.

Die Technik der Reduktion ermöglicht es, unter anderem die Entscheidbarkeit oder Unentscheidbarkeit von Problemen zu beweisen. Besonders für Unentscheidbarkeitsbeweise ist sie ein wichtiges Werkzeug. Reduzierbarkeit ist wie folgt definiert:

Definition 2. *Seien $A \subseteq \Sigma^*$ und $B \subseteq \Gamma^*$ Sprachen. Dann heißt A auf B reduzierbar – symbolisch mit $A \leq B$ bezeichnet – falls es eine berechenbare und totale Funktion $f : \Sigma^* \to \Gamma^*$ gibt, so dass für alle $x \in \Sigma^*$ gilt: $x \in A \Leftrightarrow f(x) \in B$.*

Die Funktion f wird Reduktionsfunktion oder Reduktion genannt. Sie muss drei Eigenschaften aufweisen:

1. Berechenbarkeit: man kann einen Algorithmus angeben, der f berechnet.
2. Totalität: f(x) ist definiert für alle x.
3. Für alle $x \in \Sigma^*$ gilt: $x \in A \Leftrightarrow f(x) \in B$.

Anhand der Definition erkennt man im Sinne des Themas der Bachelorarbeit, dass eine gegebene Funktion auf Berechenbarkeit, Totalität, und die Eigenschaft $x \in A \Leftrightarrow f(x) \in$

B getestet werden muss. Dabei ist zu beachten, dass es sich bei der Überprüfung der Totalität um ein unentscheidbares Problem handelt [1].

Mit Hilfe von Reduktionen kann man vorhandenes Wissen nutzen, um Entscheidbarkeit oder Unentscheidbarkeit für weitere Probleme nachzuweisen. Dabei nutzt man das folgende Lemma und seine Kontraposition:

Lemma. *Falls $A \leq B$ und B entscheidbar ist, so ist auch A entscheidbar.*

Neben Entscheidbarkeit und Unentscheidbarkeit können auch noch andere Eigenschaften von Sprachen mit Hilfe von Reduktion bewiesen werden. In bestimmten Kontexten wird dafür ein spezieller Typ von Reduktion verlangt. Polynomiale Reduktionen müssen neben oben genannten drei Bedingungen noch eine weitere Anforderung erfüllen:

Definition 3. *Seien $A \subseteq \Sigma^*$ und $B \subseteq \Gamma^*$ Sprachen. Dann heißt A auf B polynomial reduzierbar – symbolisch mit $A \leq_p B$ bezeichnet – falls es eine totale und mit polynomialer Komplexität berechenbare Funktion $f : \Sigma^* \to \Gamma^*$ gibt, so dass für alle $x \in \Sigma^*$ gilt: $x \in A \Leftrightarrow f(x) \in B$.*

Um festzustellen, ob es sich bei einer gegebenen Funktion um eine polynomiale Reduktion handelt, muss daher neben den Eigenschaften 1. - 3. außerdem noch 4. die Zeitschranke überprüft werden.

Hauptaufgabe des Backends ist es, gegebene Funktionen zu untersuchen und zu entscheiden, ob sie die drei oben genannten Eigenschaften aufweisen. Das Tool arbeitet dabei mit einem Datenmodell, dessen Klassen unter anderem Entscheidungsprobleme und ihre Instanzen repräsentieren.

3 Anforderungen und Aufbau der Anwendung

Das Tool besteht aus drei Teilen: Frontend, Datenbank und Backend. Dieses Kapitel beschreibt wie das Tool in seiner Gesamtheit aufgebaut ist und welche Aufgaben das Backend übernimmt. Es schildert das Zusammenspiel mit den anderen Komponenten und beschreibt anschließend den inneren Aufbau des Backends.

3.1 Aufgaben der Backend-Anwendung

Das Lerntool wurde im Rahmen von zwei Bachelorarbeiten entwickelt. Eine davon ist die Arbeit „Frontend-Entwicklung eines Reduktionstrainers für Studierende“ von Clemens Weiße [12], die andere ist die vorliegende Arbeit zur Backend-Entwicklung.

Dabei umfasst die Arbeit von Clemens Weiße:

- die Implementierung der Graphical User Interfaces für Studierende (Reduktionstrainer und Debugger) und Dozierende (Anlegen, Bearbeiten und Löschen von Problemen, Aufgaben und Testfällen)
- die Implementierung der Datenbank, einschließlich der Schnittstelle für die Kommunikation mit der Datenbank, die von Frontend und Backend genutzt wird.

Die Implementierung des Backends, die in dieser Arbeit beschrieben ist, beinhaltet:

- die Überprüfung, ob es sich bei einem Programm um eine Lösung für eine gegebene Aufgabe handelt (Überprüfung der Reduktionseigenschaften)
- die Implementierung der Debugger-Funktionalität
- die Schnittstelle für die Kommunikation mit dem Frontend.

Das Datenmodell der Anwendung wurde zunächst gemeinsam erarbeitet und dann von Clemens Weiße für die Implementierung in einer dokumentenorientierten Datenbank angepasst. Das Format der Anfragen für die Kommunikation zwischen Datenbank, Frontend und Backend wurde gemeinsam vereinbart. Abbildung 1 zeigt schematisch den Aufbau des gesamten Tools.

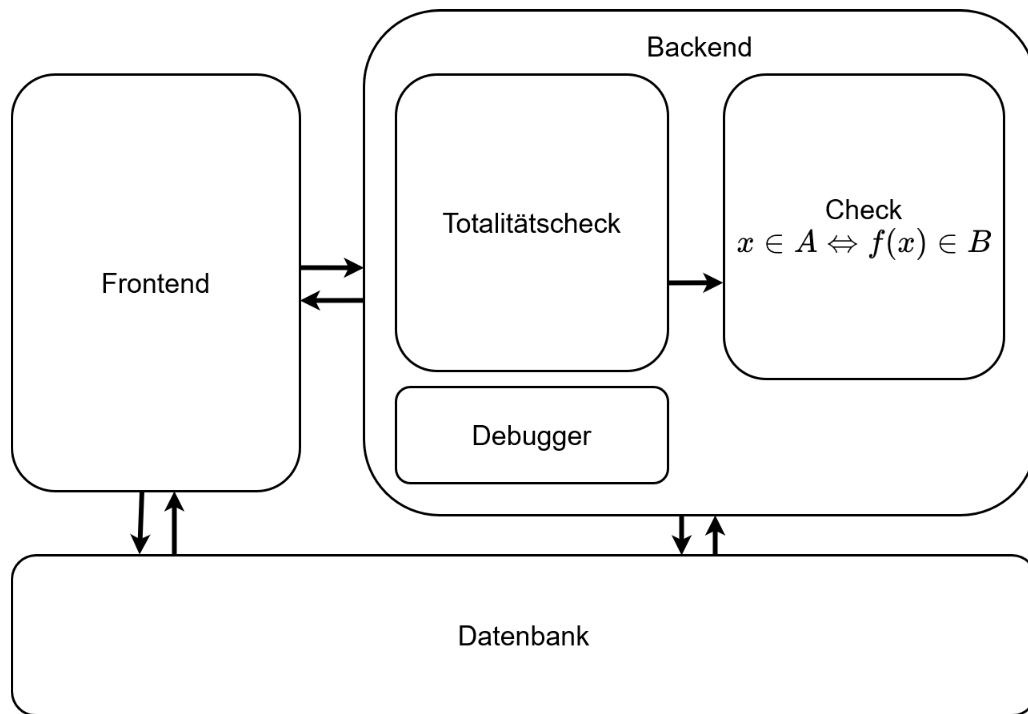

Abbildung 1: Aufbau des Tools

Die Hauptaufgabe des Backends ist die Verarbeitung von Anfragen zur Überprüfung von Reduktionen. Ziel der Verarbeitung ist es, zu entscheiden, ob der gegebene Code die Übung löst, indem er eine Reduktion berechnet. Wenn das nicht der Fall ist, soll konkretes Feedback erzeugt werden, das den Studierenden bei der Suche nach einer richtigen Lösung unterstützt.

Aus der Definition von Reduzierbarkeit in Kapitel 2 ergibt sich, dass dafür die folgenden drei Eigenschaften überprüft werden müssen:

1. Berechenbarkeit
2. Totalität
3. Eigenschaft $x \in A \Leftrightarrow f(x) \in B$

Die Berechenbarkeit ist schon dadurch bewiesen, dass die Funktion in Form von Python-Code vorliegt. Die Prüfungen der Totalität und der Eigenschaft $x \in A \Leftrightarrow f(x) \in B$ bilden die zwei Hauptbestandteile der Backend-Anwendung.

Die zweite Aufgabe des Backends ist die Verarbeitung von Anfragen an den Debugger. Diese Verarbeitung zielt darauf ab, für die gegebene Instanz die Ausgabe der Reduktionsfunktion zu berechnen und festzustellen, ob es sich um eine positive Instanz von Problem 2 handelt.

Falls während der Verarbeitung einer Anfrage Fehler auftreten (z.B. Syntaxfehler in der Reduktionfunktion), sollen diese möglichst genau erfasst und an den Studierenden zurückgegeben werden, damit er die Eingabeinstanz oder die Reduktionsfunktion korrigieren kann.

3.2 Kommunikation mit Datenbank und Frontend

Die Anwendung arbeitet mit einer dokumentenorientierten Datenbank, die Objekte verschiedener Klassen speichert. Drei Klassen sind dabei besonders wichtig für das Backend: Übungen, Probleme und Testfälle. Abbildung 2 zeigt das Klassendiagramm für das Datenmodell.

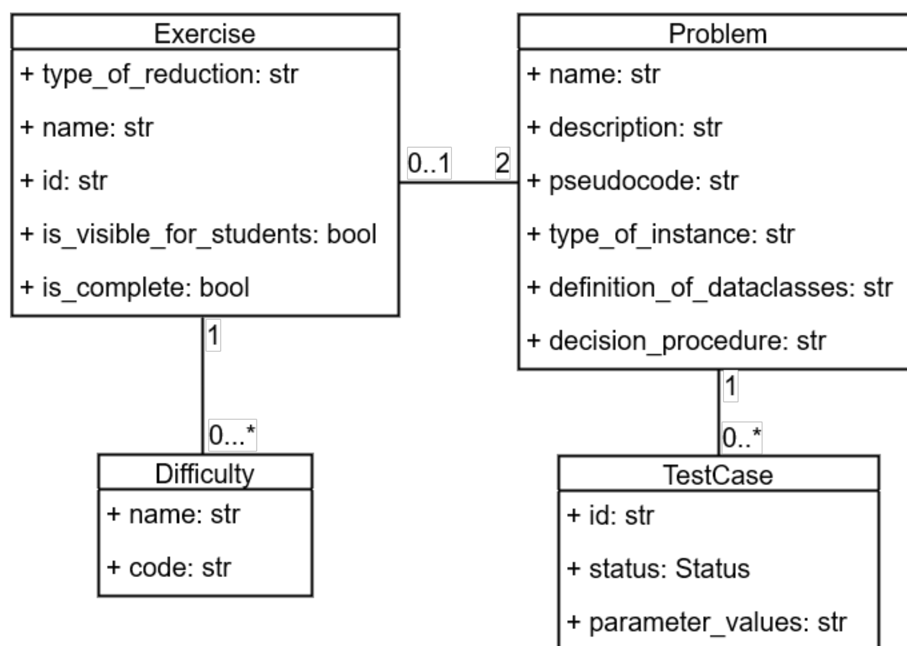

Abbildung 2: Klassendiagramm

Die Klasse Problem bildet den Kern der Anwendung. Sie modelliert Entscheidungsprobleme entsprechend der Definition in 2. Ein Teil der Attribute (Name, Beschreibung,

Pseudocode) wird in erster Linie für die Anzeige im Frontend benötigt. Andere Attribute werden von Frontend und Backend genutzt. Dazu zählen zum Beispiel der Typ der Instanzen und die Definitionen eigener Datenklassen für komplexere Typen wie Graphen oder Automaten. Die Studierenden benötigen sie, um eine Reduktion zu formulieren. Sie werden aber auch im Backend verwendet, unter anderem, um den Rückgabetyp der Reduktionsfunktion zu prüfen. Schließlich werden vom Dozierenden ein Entscheidungsverfahren und eine Menge von Testfällen angegeben. Beides wird im Backend genutzt, um die Korrektheit der Reduktion zu überprüfen.

Jeder Testfall speichert eine Instanz des Problems. Es handelt sich also um einen konkreten Wert vom Typ, der im Problemobjekt festgelegt ist. Jedem Testfallobjekt ist eine ID zugeordnet, denn für das Feedback ist es wichtig, zu wissen für welche Testfälle die Reduktionsfunktion fehlschlägt. Das Attribut Status zeigt an, ob das Entscheidungsverfahren bereits auf den Testfall angewandt wurde. Wenn ja, ist darin gespeichert, ob es sich um eine positive oder um eine negative Instanz handelt. Andernfalls zeigt der Wert an, dass der Testfall noch nicht überprüft worden ist.

Die Klasse Übung nimmt zwei Probleme und verknüpft sie. Eine Übung entspricht einer Reduktion. Die Richtung der Reduktion ergibt sich aus der Zuordnung zu den in der Abbildung als Assoziation dargestellten Attributen problem_1 und problem_2. Neben den Problemen speichert eine Übung noch weitere Attribute für die Anzeige im Frontend (is_visible_for_students...) und eine ID, mit der sie aus der Datenbank abgerufen werden kann.

Die Kommunikation mit der Datenbank erfolgt über eine FastAPI-Schnittstelle. FastAPI ist ein Webframework für Python-APIs. Das Backend sendet zwei Arten von Anfragen an die Datenbank.

Anfragen vom ersten Typ werden nur einmal während der ganzen Überprüfung gestellt. Das Backend kennt die ID der Übung und benutzt sie zu Beginn des Prozesses, um das Übungsobjekt durch eine GET-Anfrage aus der Datenbank anzufordern. Route und Methodenkopf der Schnittstelle sind in Codebeispiel 1 zu sehen. Die Route ist abhängig von der ID der Übung, die außerdem der einzige Parameter der Anfrage ist. Die Response ist die Übung in Form eines dict.

```
@app.get("/exercise/{exercise_id}", response_model=dict)
def get_exercise(exercise_id: str):
```

Codebeispiel 1: GET Exercise Schnittstelle

Ein Objekt kann andere Objekte als Attribute speichern. Während der Verarbeitung im Backend reicht es daher, das Übungsobjekt aus der Datenbank anzufordern. Alle weiteren Daten zu den Problemen und Testfällen sind in diesem Objekt enthalten.

Der zweite Anfragetyp greift schreibend auf die Datenbank zu. Wenn im Zuge der Überprüfung der Status eines Testfalls neu berechnet wird, wird der neue Wert in der Datenbank gespeichert. In diesem Fall wird eine POST-Anfrage mit dem neuen Wert an die Datenbank geschickt. Codebeispiel 2 zeigt die Route und den Methodenkopf der

```
@app.post("/change_test_case_status")
def change_test_case_status(name_of_problem: str, id_of_test_case: str,
↪  new_status: Status):
```

Codebeispiel 2: POST Status Schnittstelle

Schnittstelle. Die Methode nimmt den Namen des Problems, die ID des Testfalls und den neuen Wert für den Status entgegen, sodass der neue Wert an der richtigen Stelle gespeichert werden kann.

Im Frontend kann ein Studierender eine Übung zum Bearbeiten auswählen und sie dann innerhalb eines Trainers bearbeiten. Er kann in einem Editor Python-Code schreiben und per Button abschicken, um prüfen zu lassen, ob es sich dabei um eine richtige Lösung für diese Übung, das heißt also um eine korrekte Reduktion, handelt. Das Frontend erzeugt aus diesem Code und der ID der ausgewählten Übung eine Anfrage, die ans Backend geschickt wird.

```
class Reduction(BaseModel):
    exercise_id: str
    reduction_code: str


class DebugRequest(BaseModel):
    instance: str
    exercise_id: str
    reduction_code: str
```

Codebeispiel 3: Format der Anfrageobjekte

Neben dem Editor stellt das Frontend einen Debugger zur Verfügung. Auf der zugehörigen Seite können Studierende eine Instanz des ersten Problems eingeben. Auf Knopfdruck wird diese Instanz zusammen mit der Übungs-ID und dem Code der Reduktionsfunktion aus dem Editor in einer Anfrage ans Backend gesendet. Dort wird die Ausgabe der Reduktionsfunktion berechnet und geprüft, ob es sich um eine positive oder negative Instanz von Problem 2 handelt.

Der Debugger kann die Fehlersuche erleichtern. Anders als bei der Überprüfung der Reduktion im Editor kann der Studierende selbst entscheiden, welche Instanzen in die Reduktion eingegeben werden. Er kann auf diese Weise schnell feststellen wie die Funktion auf einer bestimmten Kategorie von Eingaben arbeitet. Wenn er zum Beispiel vermutet, dass die Funktion für positive Instanzen korrekte Ausgaben liefert und für negative fehlschlägt, lässt sich das mit Hilfe des Debuggers leicht überprüfen. Codebeispiel 3 zeigt das Format der beiden Anfrageobjekte.

3.3 Aufbau der Backend-Anwendung

Das Backend muss in der Lage sein, mehrere Anfragen gleichzeitig zu verarbeiten. Aus diesem Grund erfolgt die Kommunikation mit dem Frontend in zwei Schritten. Erst wird die zu prüfende Reduktion abgeschickt. Das Backend gibt eine ID zurück, mit der im zweiten Schritt in regelmäßigen Abständen angefragt wird, ob die Überprüfung abgeschlossen ist. Das passiert so lange, bis das Ergebnis zurückgeschickt wird. Die im Backend verwendete Sprache ist Python. Für beide Schritte ist im Backend eine eigene FastAPI-Schnittstelle definiert.

Route und Methodenkopf der ersten Schnittstelle sind in Codebeispiel 4 abgebildet. Die Methode nimmt ein Reduktionsobjekt, bestehend aus Übungs-ID und Code, entgegen und gibt eine UUID zurück. Eine UUID (Universally Unique Identifier) ist eine weltweit einmalige, 16 Bytes lange Identifikation [3], die es ermöglicht, jede Anfrage eindeutig zu identifizieren. In der Methode wird ein Backgroundtask gestartet, in dem die Reduktion überprüft wird.

```
@app.post("/checkReduction")
async def check_reduction(reduction: Reduction, background_tasks:
↪   BackgroundTasks) -> str:
```

Codebeispiel 4: Schnittstelle check_reduction

Codebeispiel 5 zeigt Route und Methodenkopf der zweiten Schnittstelle. Die Methode nimmt eine UUID entgegen, fragt ab, ob die Überprüfung dieser Reduktion abgeschlossen ist und gibt ein Feedback-Objekt zurück, wenn das der Fall ist. Beide Schnittstellen haben den Typ POST.

```
@app.post("/reductionResult")
async def reduction_result(request_id: str) -> None | Feedback |
↪  DebugResult:
```

Codebeispiel 5: Schnittstelle reduction_result

Die Aufteilung sorgt dafür, dass das Tool von mehreren Personen gleichzeitig benutzt werden kann. Jede Anfrage wird in einem eigenen Task bearbeitet. Keine Anfrage kann die Anwendung für andere blockieren.

Das Feedback-Objekt, in dem das Ergebnis gespeichert wird, wird in der Backendanwendung definiert. Es verfügt über vier Attribute. Das erste ist ein Flag, das angibt, ob das Objekt komplett leer ist. Wenn es auf true gesetzt ist, ist die Reduktion vermutlich korrekt. Die Attribute totality_feedback und correctness_feedback speichern jeweils die Ergebnisse der Überprüfung von Totalität und Eigenschaft $x \in A \Leftrightarrow f(x) \in B$. Das Attribut application_error speichert Informationen über Fehler auf Serverseite. Auf Basis des Feedback-Objekts können im Frontend Nachrichten für die Studierenden erzeugt werden.

Sobald eine Anfrage an der Schnittstelle check_reduction eingeht, startet eine Verarbeitung in zwei Schritten. Im ersten Schritt wird die Totalität der Reduktion überprüft.

An dieser Stelle findet zunächst eine Validierung der Eingabe statt, die verhindert, dass Programme mit syntaktischen Fehlern oder solche, die keine Funktion definieren, weiter verarbeitet werden.

Anschließend folgt für alle Programme, die es bis zu diesem Punkt schaffen, der eigentliche Totalitätscheck. Hierbei werden durch eine statische Analyse Elemente im Programmcode gesucht, die das Terminieren des Programms verhindern könnten, und entsprechende Warnungen erzeugt. Validierungsfehler oder Warnungen werden in einem TotalityFeedback-Objekt gespeichert, dass später einen Teil der Response bildet.

Der zweite Verarbeitungsschritt ist die Überprüfung der Eigenschaft $x \in A \Leftrightarrow f(x) \in B$. Das Tool beschränkt sich auf Reduktionen zwischen entscheidbaren Problemen und prüft die Eigenschaft mit Hilfe von Testfällen. Die Idee ist es, zu testen, ob eine Testfallinstanz genau dann im ersten Problem liegt, wenn die Rückgabe der Reduktionsfunktion,

angewandt auf diese Instanz, im zweiten Problem liegt. Dies lässt sich mit Hilfe der Entscheidungsverfahren der beiden Probleme für eine Menge von Testfällen überprüfen. Der genaue Ablauf wird in Kapitel 4 beschrieben.

Das Ergebnis der Überprüfung wird in einem Objekt vom Typ CorrectnessFeedback gespeichert. Dieses Objekt wird nach Abschluss der Überprüfung zusammen mit den zuvor erzeugten Teilen des Feedback in einem dictionary gespeichert. Der Schlüssel ist die ID der Anfrage. Von dort wird es entnommen und ans Frontend zurückgeschickt, wenn die nächste Ergebnisabfrage mit der passenden ID im Backend eingeht.

Die zweite Aufgabe des Backends ist die Verarbeitung von Anfragen an den Debugger. Diese Verarbeitung zielt darauf ab, für die gegebene Instanz die Ausgabe der Reduktionsfunktion zu berechnen und festzustellen, ob es sich um eine positive Instanz von Problem 2 handelt. Falls während dieses Prozesses Fehler auftreten, sollen diese möglichst genau erfasst und an den Studierenden zurückgegeben werden, damit er die Eingabeinstanz oder die Reduktionsfunktion korrigieren kann.

So wie die Überprüfung der Reduktionen im Editor erfolgt die Kommunikation auch in diesem Fall in zwei Schritten. Auch im Debugger soll so die gleichzeitige Verarbeitung mehrerer Anfragen ermöglicht werden und auch hier werden zwei Schnittstellen genutzt. Route und Methodenkopf der ersten Schnittstelle sind in Codebeispiel 6 zu sehen. Die Anfrage an diese Schnittstelle startet die Verarbeitung und gibt eine UUID zurück. Die Ergebnisabfrage nutzt die gleiche Schnittstelle wie die Anfragen aus dem Editor.

```
@app.post("/checkDebugInstance")
async def check_debug_instance(debug_request: DebugRequest,
↪  background_tasks: BackgroundTasks) -> str:
```

Codebeispiel 6: Schnittstelle check_debug_instance

Die Ergebnisse der Verarbeitung werden in einem Objekt vom Typ DebugResult gespeichert. Das Objekt kann fünf Attribute speichern: Zwei davon sind vorgesehen für die Instanz von Problem 2 und einen bool, der anzeigt, ob es sich um eine positive Instanz handelt. Sie enthalten Werte, wenn die Berechnung problemlos abgeschlossen wird. Falls Fehler auftreten, werden diese in den übrigen drei Attributen gespeichert: validation_error, falls die Validierung der Reduktionsfunktion fehlschlägt; reduction_runtime_error, falls während der Ausführung der Reduktionsfunktion ein Fehler auftritt und zuletzt application_error für Fehler auf Seite der Anwendung.

Die Verarbeitung läuft wie folgt: Die gegebene Reduktionsfunktion wird zunächst validiert und für Code, der syntaktisch falsch ist oder offensichtlich keine Reduktionsfunktion enthält, abgebrochen. Anschließend wird die gegebene Funktion auf der gegebenen Probleminstanz ausgeführt. Ausgabe beziehungsweise Fehler werden gespeichert und später durch eine Anfrage an die Schnittstelle aus Codebeispiel 5 abgerufen.

4 Überprüfung der Reduktionseigenschaften

In diesem Kapitel wird beschrieben wie die Überprüfung der Totalität und der Eigenschaft $x \in A \Leftrightarrow f(x) \in B$ im Tool implementiert ist. Beide Abschnitte beschreiben den Ablauf der Überprüfung und das Feedback, das dabei erzeugt wird.

4.1 Totalität

Dieser Abschnitt stellt Programmierkonstrukte vor, die dazu führen können, dass ein Programm nicht total ist und erklärt, wie solche Konstrukte, zumindest teilweise, im Zuge einer statischen Analyse gefunden werden können. Im Anschluss daran werden die Verwendung von Timeouts und das Format des User-Feedbacks erläutert.

Eine vollständige und korrekte Entscheidung darüber, ob eine gegebene Funktion total ist, ist nicht möglich, da es hierbei um ein unentscheidbares Problem handelt. Es ist allerdings möglich, eine Heuristik zu entwerfen, die einige Programmierkonstrukte identifiziert, die das Terminieren verhindern könnten. Ein Programm, das kein solches Konstrukt enthält, wird mit hoher Wahrscheinlichkeit auf allen Eingaben terminieren.

Die Grundlage für die Prüfung der Totalität ist der Zugriff auf den abstrakten Syntaxbaum der Reduktionsfunktion. Jedes syntaktisch korrekte Programm stellt ein Wort dar, das aus der abstrakten Grammatik von Python abgeleitet ist. Das gilt auch für den Code der Reduktionsfunktion. Mit Hilfe der Python-Library ast [4] ist es möglich, den Code einer zu prüfenden Funktion zu parsen und ihren Syntaxbaum zu erzeugen. Codebeispiel 7 zeigt anhand einer kleinen Additionsfunktion wie ein abstrakter Syntaxbaum aussehen kann.

Ein solcher Baum erleichtert die statische Analyse des Codes sehr. An dieser Stelle wäre es auch möglich gewesen, den String, der die Funktion enthält, direkt zu analysieren, indem man nach bestimmten Schlüsselwörtern sucht. Dieser Ansatz wäre aber aufwändiger und fehleranfälliger. Er müsste die Berechnung von Strukturinformationen,

wie zum Beispiel Klammerung, implementieren. Stattdessen liefert ast ein Baumobjekt, dessen Knoten bereits Programmierkonstrukte sind.

```
print(ast.dump(ast.parse('def add(x, y):\r\n    return x + y'), indent=4))

Module(
    body=[
        FunctionDef(
            name='add',
            args=arguments(
                posonlyargs=[],
                args=[
                    arg(arg='x'),
                    arg(arg='y')],
                kwonlyargs=[],
                kw_defaults=[],
                defaults=[]),
            body=[
                Return(
                    value=BinOp(
                        left=Name(id='x', ctx=Load()),
                        op=Add(),
                        right=Name(id='y', ctx=Load())))],
            decorator_list=[])],
    type_ignores=[])
```

Codebeispiel 7: Abstrakter Syntaxbaum Beispiel Additionsfunktion

Die Library stellt außerdem eine NodeVisitor-Klasse zur Verfügung. Darauf aufbauend definiert das Tool einen eigenen Visitor für die Prüfung der Totalität. Dieser Visitor traversiert den Syntaxbaum und sucht dabei nach Konstrukten, die die Totalität der Funktion beeinträchtigen. So erkennt der Visitor beispielsweise eine Schleife und kann Bedingung und Körper daraufhin untersuchen, ob die Schleife womöglich nie terminiert.

Bei diesem Ansatz geht es weniger darum, die Funktion einer von zwei Gruppen (total und nicht-total) zuzuordnen, sondern darum, Programmierkonstrukte aufzuspüren, die das Terminieren verhindern könnten. So wird zum Beispiel eine Warnung ausgegeben, wann immer das Programm eine while-Schleife enthält, obwohl es sich nicht bei jeder while-Schleife um eine Endlosschleife handelt. Der Grund dafür ist, dass die Lernenden ein Gefühl dafür entwickeln sollen, welche Sprachkonstrukte die Totalität beeinträchtigen

können. Sie sollen dadurch einen Zusammenhang zwischen dem theoretischen Begriff der Totalität und dem Formulieren des Codes herstellen.

Im Folgenden wird beschrieben, welche Programmierkonstrukte dazu führen können, dass ein Programm nicht terminiert. Jedes dieser Konstrukte wird durch ein Beispiel veranschaulicht. Anschließend wird beschrieben wie das Tool versucht, das Konstrukt zu erkennen. Dabei wird auch auf die Grenzen der Heuristik eingegangen.

Verschiedene Konstrukte können der Grund dafür sein, dass ein Python-Programm nicht terminiert: Imports, rekursive Aufrufe, while-Schleifen, sowie einige for-Schleifen zählen dazu.

Mit Hilfe von Imports kann man die Funktionalitäten der Python-Standardbibliothek erweitern und viele Aufgaben leichter erledigen. Auch der Kontrollfluss lässt sich mit importierten Funktionen beeinflussen, so stellt das Modul itertools [4] beispielsweise Iteratoren für Schleifen zur Verfügung, darunter auch unendliche. Imports sind daher immer eine Möglichkeit, Konstrukte in das Programm einzulassen, die von einer statischen Analyse nicht erkannt werden. Codebeispiel 8 zeigt einen solchen Fall.

Aus diesem Grund wird bei der Prüfung der Totalität vor imports gewarnt. Man kann davon ausgehen, dass die Übungsaufgaben, für die das Tool entworfen wurde, ohne den Einsatz von Bibliotheken zu lösen sind.

```
import itertools

for i in itertools.cycle([1, 2]):
    pass
```

Codebeispiel 8: Import

Auch Rekursion kann dazu führen, dass ein Programm nicht terminiert. Es kann leicht passieren, dass die Abbruchbedingung nicht richtig formuliert ist oder fehlt. Der Check warnt ganz allgemein bei rekursiven Aufrufen, da es im Kontext der Übungsaufgaben keine große Einschränkung ist, auf diese zu verzichten.

```
def func():
    func()
```

Codebeispiel 9: Rekursion

Das Tool erkennt und warnt vor direkten rekursiven Aufrufen wie dem in Codebeispiel 9. Indirekte Rekursion, bei der zum Beispiel Funktion A die Funktion B und B wiederum A aufruft, wird nicht erkannt. Sie ist aber nicht zu erwarten, da zu Beginn der Überprüfung sichergestellt wird, dass der Code nur eine Funktionsdefinition enthält.

Ein weiteres kritisches Konstrukt sind while-Schleifen. Für dieses Konstrukt sind im Totalitätscheck zwei Arten von Warnungen definiert. Erstens wird immer gewarnt, wenn eine while-Schleife gefunden wird. Zweitens wird die Schleife genauer untersucht. Wenn es Hinweise darauf gibt, dass sie bei der Ausführung endlos laufen könnte, wird eine weitere Warnung erzeugt. Natürlich kann ein Programm eine while-Schleife enthalten und trotzdem auf allen Eingaben terminieren. Eine nicht-terminierende while-Schleife ist aber schneller geschrieben als eine nicht-terminierende for-Schleife. Es reicht schon, eine kleine Anpassung der Schleifenbedingung zu vergessen.

Der Totalitätscheck geht an dieser Stelle noch einen Schritt weiter und untersucht die Schleife genauer, um festzustellen, ob sie terminiert. Wenn die Schleifenbedigung nur aus dem Wert True besteht, wird im Schleifenkörper nach break- oder return-Anweisungen gesucht. Wenn beides nicht vorhanden ist, wird gewarnt. Codebeispiel 10 zeigt einen solchen Fall.

```
while True:
    pass
```

Codebeispiel 10: while-Schleife 1

Für den Fall, dass die Bedingung ein komplexerer Ausdruck ist, werden zunächst alle Operanden aus der Bedingung herausgesucht. Dann wird der Schleifenkörper untersucht. Hier wird angenommen, dass die Bedingung zu Beginn der ersten Ausführung vermutlich zu True ausgewertet wird, da die Schleife in einem sinnvollen Programm mit der Absicht geschrieben worden sein sollte, dass sie zumindest für einige Eingaben ausgeführt wird. Wenn keiner der Operanden innerhalb des Körpers verändert wird, wird die Schleifenbedingung auch weiterhin immer gleich, also zu True ausgewertet. Dann läuft die Schleife endlos. Deshalb wird eine Warnung gesendet, wenn die Operanden in keinen Zuweisungen vorkommen oder auf andere Weise manipuliert werden. Codebeispiel 11 zeigt ein Beispiel für eine nicht-terminierende Schleife, die auf diese Weise erkannt wird.

```
i = 1
while 0 < i:
    pass
```

Codebeispiel 11: while-Schleife 2

Viele Arten von nicht-terminierenden while-Schleifen werden auf diese Weise nicht erkannt. Wenn man beispielweise in Codebeispiel 11 die pass-Anweisung durch i+=1 ersetzt, wird keine Warnung mehr erzeugt. Es ist an dieser Stelle allerdings nicht ohne erheblichen Aufwand möglich, die Manipulation aller Variablen nachzuvollziehen und beliebig komplexe Schleifenbedingungen auszuwerten. Kombiniert man allerdings die vorhandene Prüfung mit Warnungen vor Timeouts zur Laufzeit und einer sinnvoll aufgestellten Menge von Testfällen, ist davon auszugehen, dass sich viele nicht-terminierende while-Schleifen gut identifizieren lassen.

Ohne externe Funktionalitäten wie itertools eine nicht-terminierende for-Schleife zu konstruieren, ist nicht einfach. Das Tool prüft auf zwei Arten dieser Schleifen. Falls es noch andere Möglichkeiten gibt, in Python eine nicht-terminierende while-Schleife zu erzeugen, werden sie durch diese Checks nicht erkannt.

Die eine Art von Endlosschleife itertiert über eine Liste, welche im Körper der Schleife unendlich oft verlängert wird. Codebeispiel 12 zeigt ein Beispiel. Während die Größe von anderen Datentypen wie sets oder dicts zur Laufzeit nicht verändert werden kann ohne einen RuntimeError auszulösen, gibt es bei Listen keine solche Einschränkung.

```
l = [1, 2]
for i in l:
    l.append(i)
```

Codebeispiel 12: for-Schleife 1

Damit solche Schleifen erkannt werden, wird beim Traversieren des Syntaxbaumes eine Liste verwaltet, in der alle Variablen gespeichert werden, die zum aktuellen Zeitpunkt eine Liste speichern. Diese Liste ist aber nicht in allen Fällen vollständig. Zum Beispiel werden Listen, die Attribute eines Eingabeparameters sind, nicht erkannt.

Wenn im Baum eine for-Schleife gefunden wird, die über eine Liste iteriert, wird der Körper der Schleife genauer betrachtet. Wenn die Liste mit Hilfe der Methoden append

oder extend verlängert wird, wird eine Warnung erzeugt. Verlängerungen mit += werden ignoriert, da sie nur eine Kopie der Liste erzeugen. Verlängerungen mit Hilfe von Slicing werden nicht erkannt.

Eine unendliche for-Schleife lässt sich auch mit Hilfe der iter-Funktion konstruieren. Diese Funktion kann man mit zwei Eingabeargumenten aufrufen. Das erste Argument ist eine Funktion, die so lange aufgerufen wird und Werte zurückgibt, bis der Wert, der vom zweiten Argument festgelegt wird, erreicht wird. Wenn der zweite Wert nie zurückgegeben wird, läuft die Schleife endlos. Codebeispiel 13 zeigt wie so eine Schleife aussehen kann.

```
def func():
    return 1

for i in iter(func, 2):
    pass
```

Codebeispiel 13: for-Schleife mit iter

Es wird eine Warnung erzeugt, sobald eine for-Schleife gefunden wird, in deren Kopf die iter-Funktion mit zwei Argumenten aufgerufen wird. Ein solcher Aufruf kann auch nur endlich viele Rückgaben liefern. In diesem Fall kann die Warnung aber trotzdem ein Hinweis darauf sein, dass dieses Konstrukt das Terminieren verhindern kann, wenn es falsch verwendet wird. Auch ein iter-Aufruf mit nur einem Argument kann verhindern, dass die Schleife terminiert, wenn zum Beispiel eine eigene Klasse eine iter- und next-Methode implementiert, die unendlich viele Werte zurückgibt. Da es sich dabei aber um einen sehr speziellen Fall handelt, der in diesem didaktischen Kontext vermutlich nicht auftreten wird, werden Aufrufe mit nur einem Argument vom Totalitätscheck ignoriert.

Zusätzlich zur statischen Analyse des Programmtextes gibt es noch eine zweite Möglichkeit, nicht-totale Funktionen zu erkennen. Sie ergibt sich aus der Implementierung der Überprüfung der Eigenschaft $x \in A \Leftrightarrow f(x) \in B$. Wie in Kapitel 4 beschrieben werden wird, wird im Zuge der dieser Überprüfung die Reduktionsfunktion auf einer Reihe von Testfällen ausgeführt. Wenn das Programm dabei nicht von selbst terminiert, wird die Ausführung nach einer festgelegten Zeit abgebrochen. Ein solcher Timeout kann ein Indiz dafür sein, dass die gegebene Funktion nicht total ist. Daher wird auch in diesem Fall eine Warnung erzeugt und zum User-Feedback hinzugefügt.

Ein Timeout allein beweist noch nicht, dass die Funktion nicht total ist. Er kann auch dann auftreten, wenn die Ausführung terminiert, dabei aber länger dauert als vorgesehen. Der Grund dafür kann die Größe der Eingabe oder die konkrete Implementierung der Funktion sein. Ein exponentielles Verfahren kann einen Timeout auslösen. Die Reduktion kann dennoch total und eine korrekte Lösung der Übung sein.

Umgekehrt kann es sein, dass ein Programm auf allen Testfällen terminiert und dennoch nicht total ist. Das passiert dann, wenn eine kritische Eingabe nicht in den Testfällen vorkommt.

Wenn die Testfälle allerdings sinnvoll ausgewählt sind und sowohl Standard- als auch Randfälle abdecken, können auf diese Weise nicht-totale Funktionen gefunden werden, die bei der statischen Analyse nicht auffallen. Eine Warnung aus der statischen Analyse ist natürlich besonders ernst zu nehmen, wenn die Ausführung der Reduktionsfunktion später tatsächlich durch einen Timeout abgebrochen wird. Auf diese Art ergänzen sich die statischen und dynamischen Warnungen.

Die Ergebnisse des Totalitätschecks werden in einem Objekt vom Typ TotalityFeedback gespeichert. Dieses Objekt ist ein Teil der Response, in der die Ergebnisse der gesamten Überprüfung der Reduktion an das Frontend zurückgegeben werden. Sie bilden damit die Grundlage für das User-Feedback zur Totalität.

Die Klasse TotalityFeedback hat drei Attribute: error, syntactic_warnings und timed_out_test_case. Jedes dieser drei Felder kann leer sein. Wenn die Reduktion total ist, sollte TotalityFeedback leer sein.

Im Feld error werden Fehler gespeichert, die beim Parsen und Analysieren des Syntaxbaumes auftreten. Es handelt sich hierbei um Fehler, die von den Studierenden behoben werden müssen. Dazu zählen zum Beispiel Syntaxfehler im Reduktionscode. Codebeispiel 14 zeigt ein TotalityFeedback Objekt im JSON-Format, bei dem der Totalitätscheck wegen eines fehlenden Doppelpunkts in der Funktionsdefinition abgebrochen wurde.

In der Anwendung sind außerdem Exceptions definiert, die auftreten, wenn bei der Analyse des Syntaxbaumes auffällt, dass der Code offensichtlich nicht den formalen Vorgaben für die Reduktionsfunktion entspricht. Das kann Verschiedenes bedeuten: Zum Beispiel enthält der Code mehr als eine oder keine Funktionsdefinition. Er kann auf oberster Ebene auch mehrere Anweisungen enthalten, statt nur einer Funktionsdefinition - oder nur eine Anweisung von einem anderen Typ. An dieser Stelle werden auch Funktionen abgefangen, die mehr als einen Eingabeparameter erwarten. Auch diese Exceptions werden als error gespeichert.

```json
{
    "error": {
      "error_type": "SyntaxError",
      "message": "expected ':' (<unknown>, line 1)"
    },
    "syntactic_warnings": [],
    "timed_out_test_case": null
}
```

Codebeispiel 14: TotalityFeedback 1

Im Feld syntactic_warnings werden alle Warnungen aus der statischen Analyse gespeichert. Jede Warnung besteht aus einer Kategorie und einer Zeilennummer. Die Kategorien entsprechen den oben vorgestellten Konstrukten und lauten: IMPORT, RECURSION, WHILE_EXISTS, WHILE_NON_STOPPING, LIST_ ITERATOR_ EXTENDED und FOR_ITER.

```json
{
    "error": null,
    "syntactic_warnings": [
      {
        "category": "WHILE_EXISTS",
        "lineno": 3
      },
      {
        "category": "WHILE_NON_STOPPING",
        "lineno": 3
      }
    ],
    "timed_out_test_case": "case0"
  }
```

Codebeispiel 15: TotalityFeedback 2

Das letzte Feld ist timed_out_test_case. Hier wird die ID eines Testfalls gespeichert, bei dem die Ausführung der Reduktionsfunktion abgebrochen wurde, vorausgesetzt, es gibt einen solchen Fall. Codebeispiel 15 zeigt ein Beispiel. Hier warnt die statische Analyse vor einer endlosen while-Schleife, die bei der Ausführung tatsächlich dafür sorgt, dass die Ausführung auf Testfall 0 nicht terminiert.

4.2 Eigenschaft $x \in A \Leftrightarrow f(x) \in B$

Eine korrekte Reduktionsfunktion bildet positive Instanzen des ersten Problems auf positive Instanzen des zweiten Problems ab. Negative Instanzen werden auf negative Instanzen abgebildet. Die Anwendung nutzt die Entscheidungsverfahren der Probleme sowie eine Menge von Testfällen, um zu testen, ob die gegebene Funktion diese Eigenschaft erfüllt oder verletzt.

Zunächst wird jede Testfallinstanz mit Hilfe des Entscheidungsverfahrens des ersten Problems entschieden. Das Ergebnis, True oder False, wird gespeichert. Im nächsten Schritt wird die Reduktionsfunktion auf die gleiche Testfallinstanz angewandt. Die Rückgabe der Funktion ist eine Instanz des zweiten Problems. Auf diese Instanz wird anschließend das Entscheidungsverfahren des zweiten Problems angewandt.

Das Ergebnis wird mit der gespeicherten Ausgabe des ersten Entscheidungsverfahrens verglichen. Stimmen die Ausgaben für alle Testfälle überein, besteht eine gute Chance, dass die Reduktionsfunktion die Eigenschaft $x \in A \Leftrightarrow f(x) \in B$ erfüllt. Der Ablauf für einen Testfall ist in Abbildung 3 dargestellt.

Auf technischer Ebene passiert dabei Folgendes: Python stellt zwei Funktionen mit den Namen eval und exec zur Verfügung. Die Funktion eval kann einen gegebenen Ausdruck parsen und auswerten. Die Funktion exec kann einen String oder ein code object in eine Folge von Python Anweisungen parsen und zur Laufzeit ausführen [4].

Diese Funktionen werden nun verwendet, um die Ausgaben der Reduktionsfunktion und der Entscheidungsverfahren zu berechnen. Die Funktionsdefinition liegt jeweils als String aus der Datenbank oder aus der Anfrage des Studierenden vor. Ein Aufruf der exec-Funktion auf der Definition sorgt dafür, dass sie zur Laufzeit aus der Anwendung heraus aufgerufen werden kann. Anschließend wird ein Funktionsaufruf auf dem Wert der Eingabe (Testfall- oder B-Instanz) zusammengesetzt. Die Auswertung dieses Ausdrucks durch die eval-Funktion liefert dann den Rückgabewert der Reduktion oder des Entscheidungsverfahrens.

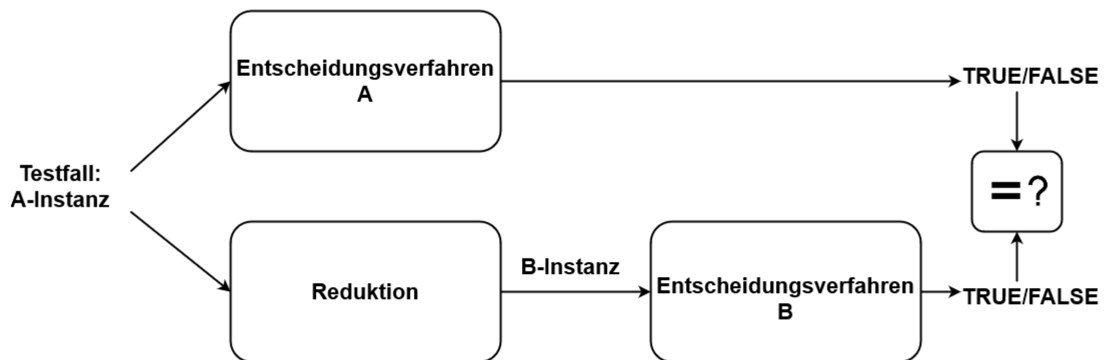

Abbildung 3: Ablauf der Überprüfung

Es ist offensichtlich, dass die Urteilsqualität der Überprüfung stark von der Auswahl der Testfälle abhängt. Beim Erstellen der Übung muss darauf geachtet werden, dass ausreichend viele Testfälle angelegt werden, sodass sowohl Standard- als auch Randfälle abgedeckt sind. Doch selbst dann ist es möglich, dass eine Funktion alle Tests besteht, obwohl sie die Eigenschaft $x \in A \Leftrightarrow f(x) \in B$ nicht erfüllt. Die Ausnahme bilden dabei nur Reduktionen zwischen endlichen Sprachen, bei denen alle Instanzen geprüft werden können. Es ist aber davon auszugehen, dass die Sprachen in den Übungsaufgaben in den meisten Fällen nicht endlich sein werden.

Diese Art der Überprüfung funktioniert nur für Reduktionen zwischen entscheidbaren Problemen. Sie ist außerdem darauf angewiesen, dass für beide Probleme Entscheidungsverfahren vorliegen, die zumindest für die Testfälle eine effiziente Laufzeit haben. Creus, Fernández und Godoy [2] implementieren ebenfalls eine testfallbasierte Prüfung, die aber die B-Instanz auf eine SAT-Instanz reduziert und diese dann mit einem SAT-Solver löst. Bei diesem Ansatz wird beim Anlegen einer Übung statt eines Entscheidungsverfahrens eine Reduktion auf SAT angegeben. Dieses Vorgehen kann von Vorteil sein, wenn eine Reduktion auf SAT leichter zu finden ist als ein Entscheidungsverfahren.

Es ist außerdem wichtig anzumerken, dass das Ausführen der Reduktionsfunktion ein Sicherheitsrisiko darstellt. Da nicht ausgeschlossen werden kann, dass auch schädlicher Code ausgeführt wird, muss sichergestellt werden, dass die Anwendung von einer virutellen Maschine ausgeführt wird.

Im Zuge des Checks wird ein Objekt vom Typ CorrectnessFeedback erzeugt, in dem die Ergebnisse der Überprüfung gespeichert werden. Dieses Objekt kann im Frontend genutzt werden, um dem Studierenden eine Rückmeldung zu seinem Programm zu geben. Es ist Teil der Response, die als Ergebnis der Überprüfung ans Frontend gesendet wird.

Im Programmcode der Anwendung und in den Feedback-Objekten wird die Eigenschaft $x \in A \Leftrightarrow f(x) \in B$ als ‚Correctness' (Korrektheit) bezeichnet, um den Code lesbarer zu gestalten.

Zwei Dinge, die beim Überprüfen der Eigenschaft $x \in A \Leftrightarrow f(x) \in B$ auftreten können, sind für den Studierenden von Interesse: Das erste sind Fehler in der Reduktionsfunktion, die erst zur Laufzeit auftreten. Diese Fehler zeigen sich, wenn die Reduktionsfunktion im zweiten Schritt des Checks auf einen Testfall angewandt wird. Sie verletzen zwar nicht direkt die Eigenschaft $x \in A \Leftrightarrow f(x) \in B$, müssen aber in jedem Fall korrigiert werden. Beim Auftreten eines solchen Fehlers wird ein Objekt vom Typ CorrectnessEr-

```
{
    "error": {
      "exc_type": "IndexError",
      "lineno": 2,
      "message": "tuple index out of range",
      "test_case_id": "case0"
    },
    "failed_test_case": null
}
```

Codebeispiel 16: CorrectnessFeedback für Laufzeitfehler

ror erstellt und im CorrectnessFeedback gespeichert. Codebeispiel 16 zeigt ein solches Objekt, das durch einen IndexError entstanden ist. Dieses Fehlerobjekt enthält den Typ des Fehlers, die Zeilennummer im Code der Funktion, die Fehlermeldung und die ID des Testfalls. Damit werden dem User – vermittelt durch das Frontend – alle Informationen zur Verfügung gestellt, die er braucht, um den Fehler zu beheben. Die Überprüfung wird beendet, wenn ein Laufzeitfehler auftritt, da die Funktion eindeutig noch keine Lösung der Übung darstellt.

Das zweite interessante Vorkommnis ist ein fehlgeschlagener Testfall. Damit ist eine positive Instanz von Problem 1 gemeint, die auf eine negative Instanz von Problem 2 abgebildet wird – oder umgekehrt. Ein solcher Fall beweist, dass die Funktion die Eigenschaft $x \in A \Leftrightarrow f(x) \in B$ nicht erfüllt. Herauszufinden, ob es solche Instanzen innerhalb der Testfälle gibt, ist die Hauptaufgabe des Checks, da sie einen wichtigen Hinweis für den Studierenden darstellen.

```
{
    "error": null,
    "failed_test_case": "case1"
}
```

Codebeispiel 17: CorrectnessFeedback für fehlgeschlagenen Testfall

Wenn ein Testfall fehlschlägt, wird seine ID im CorrectnessFeedback-Objekt gespeichert. In Codebeispiel 17 sieht man einen solchen Fall. Ob es sich um eine positive Instanz handelt, die auf eine negative abgebildet wird, oder umgekehrt, kann im Frontend aus dem Status des Testfalls abgeleitet werden. Die ID allein reicht also, um im Frontend ein Mitteilung an den User zu erzeugen. Die Überprüfung wird an dieser Stelle beendet, da die Funktion die Eigenschaft $x \in A \Leftrightarrow f(x) \in B$ nicht erfüllt und damit auch keine Lösung der Übung ist. Wenn die Reduktion die Eigenschaft aufweist, sollte das CorrectnessFeedback-Objekt leer sein.

5 Beispielübungen

An dieser Stelle wird anhand von zwei Beispielen gezeigt wie eine Reduktion als Übung modelliert werden kann und wie Dozierende und Studierende mit dem Tool arbeiten, um die Aufgabe zu erstellen und zu lösen.

5.1 Independent Set auf Vertex Cover

Eine gute Übung im Rahmen einer Einführungsveranstaltung ist die Reduktion zwischen den Problemen Independent Set und Vertex Cover. Sie lässt sich leicht bildlich darstellen und kann mit sehr wenig Code gelöst werden. Studierende, die die Reduktion schon kennen, können die Übung nutzen, um sich mit der Arbeitsweise des Tools vertraut zu machen. Die Instanzen beider Probleme sind Graphen, die Teilmengen mit bestimmten Eigenschaften enthalten können. Diese Teilmengen sind wie folgt definiert:

Definition 4. *Sei $G = (V, E)$ ein ungerichteter Graph. Eine Teilmenge der Knotenmenge $I \subseteq V$ heißt Independent Set, falls für alle $x, y \in I$ gilt, dass $(x, y) \notin E$.*

Definition 5. *Sei $G = (V, E)$ ein ungerichteter Graph. Eine Teilmenge der Knotenmenge $U \subseteq V$ heißt Vertex Cover, falls für jede Kante $(x, y) \in E$ mindestens einer der beiden Knoten $x, y \in U$.*

Mit Hilfe dieser zwei Definitionen werden jetzt die Definitionen der Probleme Independent-Set und Vertex-Cover eingeführt.

Independent-Set-Problem (INDSET)
Gegeben: ungerichteter Graph $G = (V, E)$ und $k \in \mathbb{N}$
Frage: Enthält G ein Independent Set der Größe k?

Vertex-Cover-Problem (VC)
Gegeben: ungerichteter Graph $G = (V, E)$ und $k \in \mathbb{N}$
Frage: Enthält G ein Vertex Cover der Größe k?

Für diese Reduktion soll jetzt eine Übung im Tool erstellt werden. Ein Dozierender muss dafür zunächst über das Frontend die Übung erstellen, die dann in der Datenbank gespeichert wird. Beim Erstellen der Übung muss er festlegen, wie die Graphen in Python dargestellt werden sollen. Konkret bedeutet das, dass im Frontend eine Klassendefinition in Form von Python-Code angegeben werden muss. Es ist empfehlenswert, diese Klasse als Datenklasse anzugeben.

Datenklassen sind vom User definierte Klassen, die mit dem Decorator @dataclass versehen werden. Für diese Klassen werden bestimmte Methoden, darunter auch __init__() und __repr__(), automatisch generiert [11][4]. Die __init__()-Methode fungiert als Default-Konstruktor und initialisiert den Zustand der Objekte bei ihrer Erzeugung. Die Methode __repr__() gibt eine String-Repräsentation des Objekts zurück [4]. Beide Methoden werden innerhalb der Anwendung benötigt. Erstere, damit innerhalb der Reduktionsfunktion neue Objekte angelegt werden können. Letztere, damit die Objekte im Debugger angezeigt werden können. Die Datenklassen nehmen dem Dozierenden die Aufgabe ab, diese Standard-Methoden selbst zu definieren. Es ist dennoch möglich, statt Datenklassen normale Klassen zu verwenden, solange diese Methoden angegeben werden. Codebeispiel 18 zeigt eine Möglichkeit für die Definition der Datenklasse.

In Datenklassen werden die Member-Variablen mit Type Hints versehen. Python ist eine dynamisch getypte Sprache. Dennoch ist es möglich, Funktionen und Variablen mit Type Hints zu versehen, um anzuzeigen, welche Typen Variablen, Eingaben oder Rückgaben annehmen sollen [9]. Diese Annotationen können dann von externen Bibliotheken und Tools wie mypy [8] oder Typeguard [5] verwendet werden, um statische und zum Teil auch dynamische Typchecks durchzuführen.

Im Reduktionstool werden die Annotationen beim Erstellen der Übung angegeben, um die Typen der Instanzen eindeutig festzulegen. Das ist notwendig, damit die Studie-

renden Reduktionen schreiben, die mit dem gleichen Typ von Objekten arbeiten wie die Entscheidungsverfahren.

Beim Anlegen der Datenklassen ist einiges zu beachten: Nach der Ausführung der Reduktionsfunktion wird überprüft, ob die Ausgabe den richtigen Typ hat. Bei Objekten der selbst definierten Datenklassen wird allerdings nur der Typ des Objekts geprüft, nicht aber die Felder des Objektes. Wenn sichergestellt werden soll, dass die Typen der Attribute den Vorgaben entsprechen, muss die Datenklassendefinition Zuweisungen vom falschen Typ verhindern. Codebeispiel 18 zeigt wie eine Typvalidierung durch Überschreiben der Methode __setattr__ umgesetzt werden kann. Unerlaubte Zuweisungen erzeugen hier einen TypeError, der im CorrectnessFeedback auftaucht.

In jedem Fall sollten sowohl das Anlegen neuer Objekte als auch Zuweisungen der Attribute überprüft werden. Üblicherweise würde man hier vielleicht die Built-in Python-Funktionen isintance() oder type() verwenden, die einen Typvergleich vornehmen beziehungsweise den Typ eines Objekts zurückgeben. An dieser Stelle ist aber zu beachten, dass Vergleiche mit diesen zwei Funktionen für komplexe Typen nicht ausreichend sind. Zum Beispiel lässt sich so nicht prüfen, ob es sich bei einem Wert um eine list oder um eine list[int] handelt. In solchen Fällen kann die Funktion check_type aus dem Package typeguard [5] verwendet werden, die auch in der Anwendung selbst genutzt wird.

Mit Hilfe der selbst definierten Graphklasse kann man den Typen der Instanzen festlegen. In diesem Fall haben die Instanzen beider Probleme den gleichen Typ. Es sind jeweils Tupel, die zwei Elemente enthalten: ein Objekt der Klasse Graph und einen int-Wert für die Zahl k. Codebeispiel 19 zeigt wie der Typ anzugeben ist.

Die Typen der Probleminstanzen müssen immer als Tupel angegeben werden. Der Tupel gibt für die einzelnen Teile der Instanz, das heißt für den Graph und die Zahl k, eine feste Reihenfolge vor. Das stellt sicher, dass die Reihenfolge innerhalb der Testfälle und im Methodenkopf einer korrekten Reduktionsfunktion übereinstimmen. Es wäre aber möglich, die Tupel später durch eine andere Datenstruktur zu ersetzen.

Damit die Überprüfung im Backend funktioniert, müssen außerdem noch Entscheidungsverfahren für beide Probleme sowie Testfälle hinterlegt werden. Beide Probleme lassen sich am einfachsten durch ein Verfahren entscheiden, das für den Graphen alle möglichen Teilmengen aufzählt und dann für jede prüft, ob es sich um ein Independent Set beziehungsweise Vertex Cover handelt. Codebeispiel 20 zeigt das Verfahren für Independent Set. Das Verfahren für Vertex Cover kann analog formuliert werden.

```python
from dataclasses import dataclass
from typeguard import check_type

@dataclass
class Graph:
    vertices: set[int]
    edges: set[tuple[int,int]]

    def __setattr__(self, prop, value):
        if prop == "vertices":
            self._check_vertices(value)
        if prop == "edges":
            self._check_edges(value)
        super().__setattr__(prop,value)

    @staticmethod
    def _check_vertices(vertices):
        check_type("vertices", vertices, set[int])

    @staticmethod
    def _check_edges(edges):
        check_type("edges", edges, set[tuple[int,int]])
```

Codebeispiel 18: Definition der Datenklasse Graph

```python
tuple[Graph,int]
```

Codebeispiel 19: Typ der Instanzen

Independent Set und Vertex Cover sind NP-vollständige Probleme. Die Entscheidungsverfahren, die hier angegeben werden, sind daher keine Polynomialzeit-Algorithmen. Das bedeutet aber nicht, dass sie in diesem Zusammenhang nicht eingesetzt werden können. Hier ist es sehr wichtig, zu beachten, dass die Testfälle in diesem didaktischen Kontext klein sein werden. Die Entscheidungsverfahren sollten daher in den meisten Fällen schnell terminieren. Die Qualität der Überprüfung wird nicht wesentlich dadurch beeinträchtigt, dass kleinere Testfälle gewählt werden. Diese sind auch für die Nutzenden leichter zu verstehen.

Es fehlen noch die Testfälle für das Independent-Set-Problem. Oben wurde der Typ der Instanzen festgelegt. Dieser gibt jetzt auch das Format der Testfälle vor. Stark verein-

```python
from itertools import permutations

def IndSet(instance: tuple[Graph,int]) -> bool:
    '''
    Decides an Independent Set (IndSet) instance:
    given: tuple of Graph G and integer k
    question: Has G an IndSet of size k?
    IndSet: A set of vertices that are not connected by any edge.
    '''

    def verifier(list_of_vertices: tuple[int]) -> bool:
        for i in list_of_vertices:
            for j in list_of_vertices:
                if (i,j) in G.edges:
                    return False
        return True


    G = instance[0]
    k = instance[1]

    if k > len(G.vertices):
        return False

    # a generator for all possible Independent Sets of Size k
    possible_solutions = permutations(G.vertices, k)

    for solution in possible_solutions:
        if verifier(solution):
            return True

    return False
```

Codebeispiel 20: Entscheidungsverfahren für Independent Set

```
(Graph({1,2,3,4,5}, {(1,2),(2,3),(3,4),(4,5),(5,1),(2,5)}), 2)
(Graph({1,2}, {(1,2)}), 2)
(Graph({1,2,3}, set()), 1)
```

Codebeispiel 21: Testfälle für Independent Set

facht könnte man zunächst die Werte aus Codebeispiel 21 angeben, um mindestens eine positive und negative Instanz, sowie einen Randfall, einen Graph ohne Kanten, abzudecken. Allerdings sollte man mehr Testfälle angeben, bevor man die Übung sichtbar macht. Wenn später in einer anderen Übung Vertex Cover auf ein anderes Problem reduziert werden soll, müssen auch für Vertex Cover noch Testfälle hinzugefügt werden.

Die Testfallinstanzen können graphisch dargestellt werden. Abbildung 4 zeigt die drei Graphen. Kandidaten für Independent Sets sind grün markiert. Wie man sieht, ist Testfall 1 eine positive Instanz, zum Beispiel gibt es das Independent Set $\{1, 4\}$. Testfall 2 ist eine negative Instanz. Testfall 3 ist eine positive Instanz, jeder Knoten im Graph bildet ein Independent Set der Größe 1.

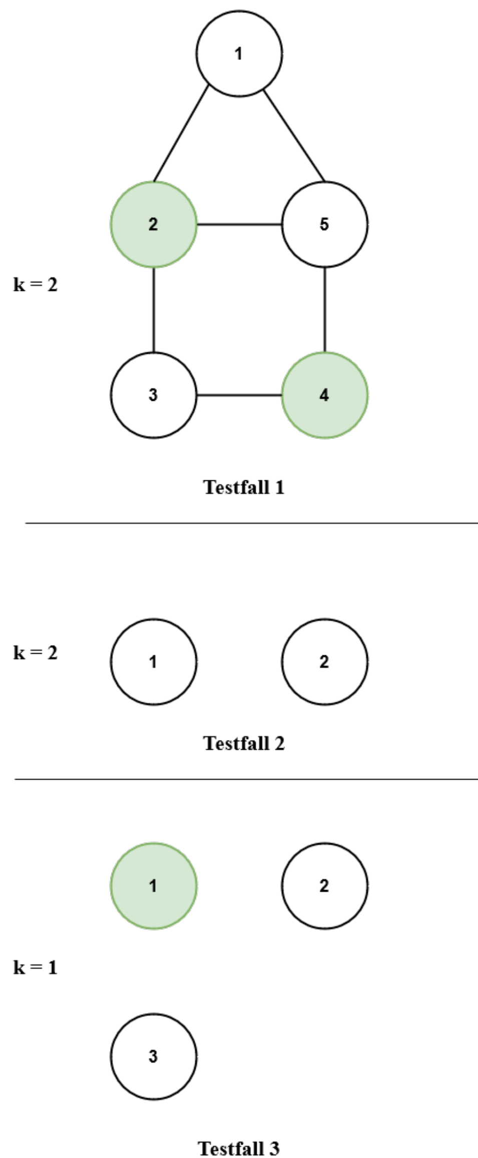

Abbildung 4: Testfälle für Independent Set

Auf diese Weise kann man also zu zwei gegebenen Entscheidungsproblemen eine Übung erstellen. Sobald diese für die Studierenden im Frontend sichtbar gemacht wird, können sie die Übung bearbeiten und ihre Lösungsvorschläge überprüfen lassen.

Angenommen, ein Studierender probiert eine erste Reduktion aus. Vielleicht gibt er als ersten Versuch eine Funktion an, die den Graphen und ein festes k zurückgibt, so wie in Codebeispiel 22.

```
def IndSet_to_VC(indset_instance):
    return (indset_instance[0],1)
```

Codebeispiel 22: Falsche Reduktionsfunktion

```
def IndSet_to_VC(indset_instance: tuple[Graph,int]) -> tuple[Graph,int]:
    '''
    Reduction from Independent Set to Vertex Cover via
    (G,k) is in IndSet <=> (G,|V|-k) is in VC

    '''
    G = indset_instance[0]
    k = indset_instance[1]

    # if obviously negative instance
    if k > len(G.vertices):
        return (Graph(set(),set()),1)

    return (G,len(G.vertices)-k)
```

Codebeispiel 23: Reduktionsfunktion IndSet_to_VC

Schon Testfall 1 wird dann fehlschlagen. Während man mit einem einzelnen Knoten problemlos ein Independent Set angeben kann, enthält der Graph eindeutig kein Vertex Cover der Größe 1. Das Feedback, das ans Frontend geschickt wird, enthält die ID von Testfall 1. Im Frontend sollte dann der Wert dieses Testfalls angezeigt werden. Der Studierende kann sich dieses Beispiel also anschauen und seine Reduktionsfunktion noch einmal überdenken. Vielleicht testet er mit Hilfe des Debuggers eigene Beispiele und erkennt schließlich den Zusammenhang: $(G, k) \in INDSET \Leftrightarrow (G, |V| - k) \in VC$. Davon ausgehend kann er jetzt wie in Codebeispiel 23 eine richtige Reduktionsfunktion formulieren.

Abbildung 5 zeigt die Ausgaben der Reduktionfunktion für die gegebenen Eingaben. Wie man sieht, ist die Eigenschaft $x \in A \Leftrightarrow f(x) \in B$ für alle drei Testfälle erfüllt. Das Feedback enthält keine Warnungen für die Totalität. Die gegebene Funktion löst die Übung.

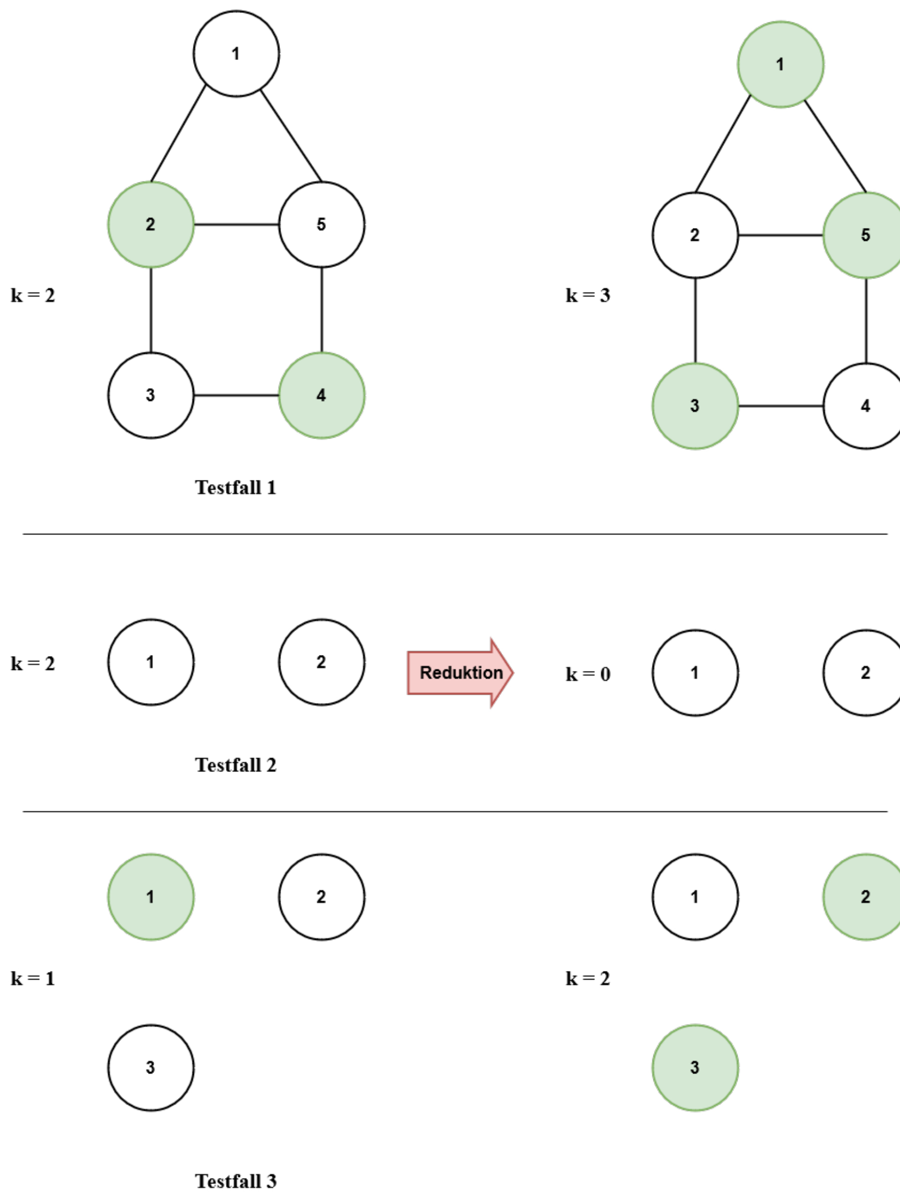

Abbildung 5: Beispiele Reduktion von Independent Set auf Vertex Cover

5.2 Leerheit für NFA auf $\overline{\text{PATH}}$

Eine weitere Übung, die sich gut für den Einstieg eignet, ist die Reduktion von Leerheit für NFA auf das Komplement des Erreichbarkeitsproblems in Graphen. Anders als bei der Übung im letzten Abschnitt unterscheiden sich die beiden Probleme im Typ ihrer Instanzen.

Für das erste Problem sind diese Instanzen nichtdeterministische endliche Automaten. Deren Definition ist angelehnt an die von Schöning [10] sowie Hopcroft, Motwani und Ullman [6], wobei es hier immer nur je einen Anfangs- und Endzustand gibt. Außerdem sind die Transitionen als Relation statt als Funktion definiert. Beides vereinfacht später die Reduktion im Beispiel.

Definition 6. *Ein nichtdeterministischer, endlicher Automat (kurz: NFA) wird spezifiziert durch ein 5-Tupel* $M = (Q, \Sigma, \delta, q_a, q_e)$*. Hierbei sind:*

- Q *eine endliche Menge, die Zustände*
- Σ *eine endliche Menge, das Eingabealphabet,* $Q \cap \Sigma = \emptyset$
- $\delta \subseteq W \times \Sigma \times Q$ *eine Transitionstabelle*
- $q_a \in Q$ *der Anfangszustand*
- $q_e \in Q$ *der Endzustand*

Definition 7. *Sei* $A = (Q, \Sigma, \delta, q_a, q_e)$ *ein NFA und* $w = a_1...a_n \in \Sigma^*$*. Ein Lauf von A auf w ist eine Folge von* $p = q_0, a_1, q_1...q_{n-1}, a_n, q_n \in Q(\Sigma Q)^*$*, so dass für alle i = 0, 1,..., n-1:* $(q_i, a_{i+1}, q_{i+1}) \in \delta$.

Definition 8. *Ein Lauf* $p = q_0, ..., q_n$ *von A auf w heißt akzeptierend, falls* $q_0 = q_a$ *und* $q_n = q_e$.

Definition 9. *Die Sprache eines NFA A ist L(A)* $= \{w \in \Sigma^* \mid$ *Es gibt einen akzeptierenden Lauf von A auf w* $\}$

Die Probleme sind wie folgt definiert:

Leerheit für NFA (EMPT)
Gegeben: NFA $A = (Q, \Sigma, \delta, q_a, q_e)$
Frage: $L(A) = \emptyset$?

Grapherreichbarkeit Komplement ($\overline{\textbf{PATH}}$)
Gegeben: gerichteter Graph $G = (V, E)$ und zwei Knoten $s, t \in V$
Frage: Gibt es in G keinen Pfad von s nach t?

Beim Erstellen der Aufgabe sollte man am besten eine neue Datenklasse für NFAs anlegen. Diese könnte so aussehen wie in Codebeispiel 24, müsste aber noch um die Typvalidierung der NFA-Attribute ergänzt werden. Die Graphen können im Wesentlichen

so definiert sein, wie im letzten Abschnitt. Man kann sich aber an der Kodierung des NFA orientieren, wenn man den Typ der Knoten festlegt. Hier würde man die Knoten wohl als Strings speichern. Da die Instanzen immer Tupel sind, wählt man für EMPT den Typen tuple[Nfa] und für $\overline{\text{PATH}}$ den Typen tuple[Graph, str, str]. Beide Entscheidungsverfahren bestehen aus einer einfachen Erreichbarkeitssuche.

```
@dataclass
class Nfa:
    states: set[str]
    alphabet: set[str]
    transitions: set[tuple[str,str,str]]
    start_state: str
    end_state: str
```

Codebeispiel 24: Klassendefinition NFA

Die Reduktionsfunktion ist in Codebeispiel 25 angegeben. Die Transitionen werden zu Kanten, wenn man die Alphabetsymbole an den Übergängen entfernt. Anfangs- und Endzustand entsprechen den Knoten s und t. Dann gilt, dass es genau dann keinen akzeptierenden Lauf auf dem NFA gibt, wenn im Graph kein Pfad von s zu t existiert.

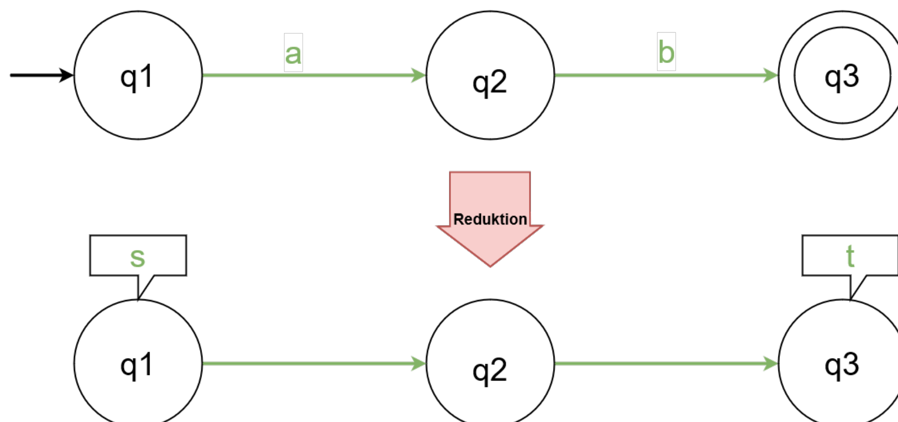

Abbildung 6: Beispiel Reduktion von EMPT auf $\overline{\text{PATH}}$

Abbildung 6 veranschaulicht den Zusammenhang anhand einer negativen Instanz von EMPT. $L(A) = \{ab\}$ und damit nicht leer. Zugleich gibt es in G einen Pfad von Knoten

```python
def reduction_func(instance: tuple[Nfa]) -> tuple[Graph, str, str]:

    states = instance[0].states
    transitions = instance[0].transitions
    start_state = instance[0].start_state
    end_state = instance[0].end_state

    edges = set()
    for transition in transitions:
        edges.add((transition[0], transition[2]))

    return (Graph(nodes = states, edges = edges), start_state,
     ↪  end_state)
```

Codebeispiel 25: Reduktion von EMPT auf $\overline{\text{PATH}}$


```json
{
  "b_instance": "(Graph(nodes={'q2', 'q1', 'q3'}, edges={('q1', 'q2'),
  ↪  ('q2', 'q3')}), 'q1', 'q2')",
  "validation_error": null,
  "reduction_runtime_error": null,
  "application_error": null,
  "b_instance_is_true": false
}
```


Codebeispiel 26: Debugger Ergebnis

q1 zu Knoten q3, sodass die Instanz nicht im Komplement des Grapherreichbarkeitsproblems liegt. Codebeispiel 26 zeigt die Rückgabe, die das Backend zurückschickt, wenn man den NFA aus Abbildung 6 an den Debugger schickt. Die zurückgegebene Instanz entpricht dem unteren Teil der Abbildung. Das letzte Feld zeigt an, dass es sich korrekterweise um eine negative Instanz handelt.

Die Übungsaufgaben in diesem Kapitel sind für einen ersten Einstieg in das Thema gedacht. Bei schwierigeren Aufgaben reicht eine kleine Manipulation der Eingabe nicht aus, um die Lösung zu finden. Aus diesem Grund haben Dozierende die Möglichkeit, im Frontend verschiedene Schwierigkeitsgrade für die Übungen anzugeben. Das bedeutet, dass Teile der Reduktion vorgegeben sind und von den Studierenden wie bei einem Lückentext ausgefüllt werden. Fortgeschrittene Aufgaben, bei denen die Rückgabe von

Grund auf konstruiert wird (siehe zum Beispiel die Reduktion von 3-KNF-SAT auf 3-Färbbarkeit in [10]), können auf diese Weise in kleinere Schritte aufgeteilt werden.

6 Fazit und Ausblick

Die Anwendung, die in dieser Arbeit beschrieben ist, kann in Zukunft im Rahmen von Lehrveranstaltungen eingesetzt werden und Studierende beim Lernen unterstützen. Sie ist aber auch ein Prototyp, der weiterentwickelt und um zusätzliche Funktionalitäten ergänzt werden kann. Vom allem muss sich in der Praxis zeigen, wie das Tool anzupassen ist, sodass es den größten Nutzen bietet.

Es ist anzunehmen, dass Lernende, die sich von reinen Theorieaufgaben eingeschüchtert fühlen, die Aufgaben im Tool besser aufnehmen als solche auf Papier. Die Theorie ist versteckt hinter einer vertrauten Praxis – dem Programmieren in Python. Im Editor kann man ohne große Mühe einen ersten, zweiten und dritten Versuch formulieren und nebenher mit Hilfe des Debuggers erkunden, wo die einzelnen Ansätze fehlschlagen. Das Vorgehen unterscheidet sich damit nicht grundlegend von anderen Aufgaben im Informatikstudium, auch wenn es immer noch darum geht, eine Technik aus der Theorie zu trainieren. Vor allem sollten Anfängerfehler wie Funktionen mit falschem Eingabe- oder Ausgabetyp oder Reduktionen in die falsche Richtung seltener auftreten, wenn jemand eine Weile mit dem Tool arbeitet. Damit sollte sich der Fokus beim Lernen auf die Erfüllung der Eigenschaft $x \in A \Leftrightarrow f(x) \in B$ verschieben, die den eigentlichen Kern der Aufgabe darstellt.

Die Hauptaufgabe des Tools ist die Unterstützung der Studierenden beim Lernen. Zusätzlich dazu wird durch die Befüllung des Tools aber auch eine Datenbank von Problemen erzeugt, die eine nützliche Ressource für die Lehrenden sein kann. Die Dokumente in der Datenbank enthalten unter anderem Beschreibungen, Entscheidungsverfahren und Testfälle und können auch in anderen Zusammenhängen genutzt werden.

Damit das Tool korrekt funktioniert und eine echte Hilfe ist, müssen die Aufgaben richtig erstellt werden. Die Überprüfung der Eigenschaft $x \in A \Leftrightarrow f(x) \in B$ liefert nur dann richtige Ergebnisse, wenn die Entscheidungsverfahren korrekt sind und eine sinnvolle Auswahl von Testfällen zur Verfügung steht. Die einzige Möglichkeit nicht sinnvolle Instanzen zu erkennen - zum Beispiel einen Automaten, dessen Endzustand nicht in der Zustandsmenge vorkommt - besteht darin, sie durch die Methoden der Datenklassen oder die Entscheidungsverfahren abzufangen. Das Tool bietet viele Freiheiten beim Erstellen

von Aufgaben, sodass Entscheidungsprobleme aller Art modelliert werden können. Das bedeutet aber auch, dass die Klassendefinitionen, Entscheidungsverfahren und Testfälle kaum überprüft werden können. Die Aufgaben sollten daher immer auf Fehler getestet werden, bevor sie für die Studierenden sichtbar gemacht werden.

Die Aufgabenerstellung in der Dozierendenansicht ist nicht die einzige Stelle, an der sich aus den Usereingaben Schwierigkeiten ergeben können. Auch wenn das Tool zur Überprüfung von Funktionen gedacht ist, können die Studierenden Python-Code verschiedenster Art an das Backend schicken. Das stellt ein Sicherheitsrisiko dar, da jedes Programm, dass die Validierung zu Beginn besteht, später auch ausgeführt wird. Es ist deshalb wichtig, das Tool immer nur von einer virutellen Maschine ausführen zu lassen, sodass sich mögliche Schäden auf den Rahmen des Tools und der Datenbank beschränken.

Mit Ausnahme der Timeouts sind die Überprüfung der Totalität und der Eigenschaft $x \in A \Leftrightarrow f(x) \in B$ separat implementiert. Eine Möglichkeit, das Tool weiterzuentwickeln, besteht darin, den vorhandenen Totalitätscheck zu erweitern oder komplett zu ersetzen.

Anstatt bestimmte kritische Konstrukte im Code zu identifizieren und Warnungen für diese zu erzeugen, wäre es denkbar, die Sprache für die Reduktionsfunktionen von vornherein einzuschränken. Konstrukte wie while-Schleifen oder Imports könnten im Frontend bereits markiert werden. Eine Reduktion könnte in diesem Fall erst dann abgeschickt werden, wenn sie keine verbotenen Konstrukte mehr enthält. In diesem Fall würde die Überprüfung der Totalität entweder stark eingeschränkt oder ganz an das Frontend abgegeben werden. Falls letzteres passiert, würden einige Konstrukte wie unendliche for-Schleifen vermutlich nur durch Timeouts auffällig werden. Auf diese Weise wird auch die Verbindung zwischen solchen Elementen und dem Begriff der Totalität verschleiert. Man kann schließlich das Verbot akzeptieren, ohne den Zusammenhang zum Terminieren des Programms herzustellen.

Ein weiterer grundsätzlich anderer Ansatz bestünde darin, einen Timer laufen zu lassen, alle möglichen Eingaben aufzuzählen und die Funktion auf diesen Eingaben auszuführen. Wenn eine Eingabe gefunden wird, für die die Funktion nicht terminiert, handelt es sich um eine nicht-totale Funktion. Bei diesem Ansatz besteht die größte Schwierigkeit in der Frage, wie die Aufzählung selbst implementiert wird. Packages wie itertools können helfen, verhindern aber vermutlich auch nicht, dass viele nicht sinnvolle Eingaben aufgezählt werden. Da es in den meisten Fällen unendlich viele mögliche Eingaben gibt, lassen sich auch auf diese Weise nicht alle nicht-totalen Funktionen finden. Nicht

zuletzt bedeutet der Timer auch, dass die Studierenden das Feedback erst mit einer kurzen Verzögerung erhalten, was bei der statischen Analyse nicht der Fall ist.

Ebenso ist es denkbar, den vorhandenen Check beizubehalten und nur einzelne Teile zu verändern oder zu verbessern. So könnte man zum Beispiel auch bei einer for-Schleife mit einem iter-Aufruf mit zwei Argumenten das callable des ersten Arguments für eine Aufzählung von Eingaben aufrufen, um zu sehen, ob einer der Rückgabewerte das zweite Argument ist. Wenn ja, braucht keine Warnung erzeugt zu werden. Auch diese Aufzählung könnte eine Verzögerung der Antwort bewirken.

Man kann auch versuchen, auf dynamische Weise festzustellen, ob eine for-Schleife über eine Liste iteriert. Dazu kann man den Code vor der Schleife ausführen und anschließend zur Laufzeit den Typ der Variable aus dem Schleifenkopf abfragen. Auch hier muss man einen Timeout einplanen, falls es schon in dem Code vor der Schleife ein Konstrukt gibt, dass eine Endlosschleife erzeugt. Selbst wenn das nicht der Fall ist, ist es möglich, dass eine Variable für verschiedene Eingaben Werte verschiedener Typen zugewiesen bekommt. Dieser Ansatz ist daher nicht einmal zuverlässig. Es bleibt aber die Frage, ob es noch andere Möglichkeiten gibt, die Erkennung von Listen zu verbessern, sodass die Warnungen für for-Schleifen verbessert werden können.

Neben der Überprüfung der Totalität bieten sich noch an anderen Stellen Erweiterungen an. Zum Beispiel wäre es denkbar, alle Instanzen, die von den Studierenden im Debugger eingegeben werden, als Testfälle in der Datenbank zu speichern. Dabei sollte sichergestellt werden, dass Instanzen nicht mehrfach gespeichert werden. Auch Eingaben, die keine sinnvollen Instanzen sind, müssten gefiltert werden. Doch wenn das gelingt, würde die Überprüfung der Eigenschaft $x \in A \Leftrightarrow f(x) \in B$ gründlicher werden, ohne dass ein Dozierender von Hand zusätzliche Testfälle eingeben muss.

Im Datenmodell haben Übungen bereits ein Attribut für den Typ der Reduktion. Dieses Attribut wird von der vorhandenen Implementierung nicht genutzt. In einer Weiterentwicklung könnte man beim Erstellen der Aufgabe in diesem Feld ein Polynom angeben. Das Tool sollte dann überprüfen, ob die Rechenzeit der Reduktionsfunktion durch dieses Polynom beschränkt wird. Wie genau diese Überprüfung implementiert werden könnte, ist noch unklar.

Bisher können mit dem Tool nur Reduktionen zwischen entscheidbaren Problemen überprüft werden. Die Technik ist aber auch gerade deshalb so nützlich, weil man mit ihrer Hilfe Unentscheidbarkeit nachweisen kann. Deshalb wäre es sinnvoll, das Tool so zu

ergänzen, dass es auch Reduktionen zwischen unentscheidbaren Problemen überprüfen kann.

Die testfallbasierte Überprüfung ließe sich zumindest teilweise auf unentscheidbare Probleme erweitern. Es wäre zum Beispiel möglich, sich bei der Auswahl der Testfälle auf entscheidbare Teilmengen zu beschränken, wobei man sich überlegen muss, wie Timeouts des zweiten Entscheidungsverfahrens zu interpretieren sind.

Im Fall von semi-entscheidbaren Problemen könnte man die Entscheidungsverfahren durch einen Timeout stoppen, wenn die Ausführung nicht von selbst terminiert. Während so ein Timeout bisher als Fehler der Anwendung eingestuft wird, könnte er in diesem Fall als Hinweis auf eine negative Instanz gewertet werden. So eine Überprüfung wäre aber weniger zuverlässig als die bisherige. Die Timeouts könnten auch dazu führen, dass sich die Antwort verzögert.

Abbildungsverzeichnis

1 Aufbau des Tools 8
2 Klassendiagramm 9
3 Ablauf der Überprüfung 24
4 Testfälle für Independent Set 32
5 Beispiele Reduktion von Independent Set auf Vertex Cover 34
6 Beispiel Reduktion von EMPT auf $\overline{\text{PATH}}$ 36

Literatur

[1] Egon Börger. *Berechenbarkeit, Komplexität, Logik: eine Einführung in Algorithmen, Sprachen und Kalküle unter besonderer Berücksichtigung ihrer Komplexität.* 3., verbesserte und erweiterte Aufl. Braunschweig; Wiesbaden: Vieweg, 1992, S. 54.

[2] Carles Creus, Pau Fernández und Guillem Godoy. „Automatic Evaluation of Reductions between NP-Complete Problems". In: *Theory and Applications of Satisfiability Testing – SAT 2014.* Hrsg. von Carsten Sinz und Uwe Egly. Cham: Springer International Publishing, 2014, S. 415–421.

[3] Peter Fischer und Peter Hofer. *Lexikon der Informatik.* 15., überarbeitete Aufl. Berlin; Heidelberg: Springer, 2011, S. 951.

[4] Python Software Foundation. *Python 3.10.9 Documentation.* 2001-2022. URL: `https://docs.python.org/3.10/` (besucht am 12. 12. 2022).

[5] Alex Grönholm. *Typeguard 2.13.3 Documentation.* 2015. URL: `https://typeguard.readthedocs.io/en/latest/` (besucht am 16. 11. 2022).

[6] John E. Hopcroft, Rajeev Motwani und Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* 2. Aufl. München: Pearson Studium, 2002.

[7] Lukas König, Friederike Pfeiffer-Bohnen und Hartmut Schmeck. *Theoretische Informatik - ganz praktisch.* Berlin; Boston: De Gruyter Oldenbourg, 2016.

[8] Jukka Lehtosalo und mypy contributors. *mypy 0.991 Documentation.* 2012-2022. URL: `https://mypy.readthedocs.io/en/stable/index.html` (besucht am 12. 12. 2022).

[9] Guido van Rossum, Jukka Lehtosalo und Łukasz Langa. *Type Hints.* PEP 484. 2014. URL: `https://peps.python.org/pep-0484/` (besucht am 12. 12. 2022).

[10] Uwe Schöning. *Theoretische Informatik - kurz gefasst.* 5. Aufl. Heidelberg: Spektrum Akademischer Verlag, 2008.

[11] Eric V. Smith. *Data Classes.* PEP 557. 2017. URL: `https://peps.python.org/pep-0557/` (besucht am 12. 12. 2022).

[12] Clemens Weiße. „Frontend-Entwicklung eines Reduktions-Trainers für Studierende". nicht veröffentlicht. n.d.