

Inhaltsverzeichnis

Abbildungsverzeichnis	2
Tabellenverzeichnis	4
1 Einleitung	5
2 Warum Python?	6
3 Skripte ohne spezielle Module	7
4 Python Modul lecture	8
4.1 Sets	9
4.2 Matches	12
4.3 Übersetzung komplexerer Programme	13
4.3.1 Schiebepuzzle	13
4.3.2 Watson	17
4.3.3 Wolf Ziege Kohl	20
4.3.4 8 Damen Problem	23
5 Fazit	29

Abbildungsverzeichnis

3.1	Einfache Summenberechnung in SETLX	7
3.2	Einfache Summenberechnung in Python	7
4.1	Fehler bei Mengen in Mengen	8
4.2	Nutzung von Mengen in <i>simple.stlx</i>	11
4.3	Nutzung von Mengen in <i>simple.py</i>	11
4.4	Ausschnitt aus <code>diff()</code> (SETLX)	12
4.5	Ausschnitt aus <code>diff()</code> (Python)	13
4.6	<code>findPath</code> im Schiebepuzzle (SETLX)	14
4.7	<code>find_path</code> im Schiebepuzzle (Python)	14
4.8	<code>nextStates</code> im Schiebepuzzle (SETLX)	15
4.9	<code>next_states</code> im Schiebepuzzle (Python)	15
4.10	<code>moveDir</code> im Schiebepuzzle (SETLX)	15
4.11	<code>move_dir</code> im Schiebepuzzle (Python)	15
4.12	<code>findBlank</code> im Schiebepuzzle (SETLX)	16
4.13	<code>find_blank</code> im Schiebepuzzle (Python)	16
4.14	<code>evaluate</code> in Watson (SETLX)	18
4.15	<code>evaluate</code> in Watson (Python)	18
4.16	<code>createValuation</code> in Watson (SETLX)	18
4.17	<code>create_valuation</code> in Watson (Python)	19
4.18	Lösung von Watson (SETLX)	19
4.19	Lösung von Watson (Python)	19
4.20	<code>findPath</code> in wgc (SETLX)	20
4.21	<code>find_path</code> in wgc (Python)	20
4.22	<code>pathProduct</code> in wgc (SETLX)	21
4.23	<code>path_product</code> in wgc (Python)	21
4.24	<code>noCycle</code> in wgc (SETLX)	21
4.25	<code>no_cycle</code> in wgc (Python)	21
4.26	<code>add</code> in wgc (SETLX)	22
4.27	<code>add</code> in wgc (Python)	22
4.28	<code>problem</code> in wgc (SETLX)	22
4.29	<code>problem</code> in wgc (Python)	22
4.30	Lösung des wgc-Problems (SETLX)	23
4.31	Lösung des wgc-Problems (Python)	23
4.32	<code>davisPutnam</code> in Davis Putnam (SETLX)	24
4.33	<code>davis_putnam</code> in Davis Putnam (Python)	24
4.34	<code>saturate</code> in Davis Putnam (SETLX)	25
4.35	<code>saturate</code> in Davis Putnam (Python)	25
4.36	<code>atMostOne</code> in Davis Putnam (SETLX)	25
4.37	<code>at_most_one</code> in Davis Putnam (Python)	25
4.38	<code>atMostOneInRow</code> in Davis Putnam (SETLX)	26
4.39	<code>at_most_one_in_row</code> in Davis Putnam (Python)	26
4.40	<code>atMostOneInLowerDiagonal</code> in Davis Putnam (SETLX)	26

4.41	at_most_one_in_lower_diagonal in Davis Putnam (Python)	26
4.42	allClauses in Davis Putnam (SETLX)	27
4.43	all_clauses in Davis Putnam (Python)	27
4.44	solve in Davis Putnam (SETLX)	27
4.45	solve in Davis Putnam (Python)	28

Tabellenverzeichnis

4.1	Mathematische Operatoren für Sets	10
4.2	Funktionen für Sets	10

Kapitel 1

Einleitung

In der Vorlesung „Grundlagen und Logik“ des Moduls Theoretische Informatik I führt der Dozent Prof. Dr. Karl Stroetmann die Programmiersprache SETLX ein. SETLX ist eine auf Java basierende Sprache, die sehr gut geeignet ist, um den Pseudocode aus Vorlesungen ausführbar zu machen. Diese Programmiersprache wirbt damit, dass die Verwendung von Mengen und Listen sehr gut unterstützt wird. Außerdem können Ausdrücke aus der Mengenlehre, so wie andere mathematischen Ausdrücke in einer Syntax, die sehr ähnlich zur mathematischen Notation ist, implementiert werden. [SH15] Da diese Vorlesung bereits im ersten Semester stattfindet und die Studenten parallel dazu eine Vorlesung aus dem Modul Mathematik I besuchen, können die Studenten Themen wie beispielsweise die Mengenlehre schneller kennen lernen. Die komplementäre Auseinandersetzung mit ähnlichen bis gleichen Themen in beiden Vorlesungen ermöglicht das gleichzeitige Lernen für zwei Vorlesungen.

Ein weiterer Vorteil für die Studenten ist, dass die Syntax von SETLX, zusätzlich zum sehr mathematischen Stil, auch starke Ähnlichkeiten zur Programmiersprache C aufweist. Selbst für die Studenten, die zuvor keinen Kontakt mit C hatten ist das ein großer Vorteil, da im ersten Semester parallel zur theoretischen Informatik Vorlesung auch eine Vorlesung mit dem Titel „Programmieren in C“ besucht werden muss. So muss kein starkes Umdenken stattfinden, wenn von SETLX zu C und auch umgekehrt gewechselt wird.

Die Hauptintention dieser Arbeit ist es, zu prüfen, ob es möglich wäre die SETLX-Programme, die in der Vorlesung gezeigt werden, in Python-Skripte zu übersetzen. Ziel dabei ist es, die Eleganz der Programme beizubehalten, damit die Studenten die verschiedenen Algorithmen besser verstehen und erlernen können. Auch wenn es in Python teilweise möglich wäre das Problem über einen anderen Weg, eventuell mit weniger Codezeilen, zu lösen, soll dennoch der Hauptfokus auf den Lernprozess in der theoretischen Informatik liegen.

Kapitel 2

Warum Python?

Viele Informatik-Kurse oder Vorlesungen für Anfänger benutzen die Programmiersprache Python als erste Programmiersprache. Von den 39 besten Einführungskursen für Informatik in den USA verwendeten im Jahr 2014 27 Kurse Python als erste Programmiersprache. [Guo14] Mit 69% ist Python somit mit einer eindeutigen Mehrheit deutlich die meist verwendete Programmiersprache unter diesen Kursen. Einige Internet-Artikel, die die Beliebtheit von heutigen Programmiersprachen beleuchten, referenzieren öfter den Blogbeitrag für die Association for Computing Machinery (ACM). In dem Beitrag wird beschrieben, dass Python Java als häufigste Programmiersprache für Anfänger abgelöst hat. Auch wenn der Artikel bereits 2014 veröffentlicht wurde, lässt sich vermuten, dass die Verbreitung von Python nicht zurückgegangen ist. Grund hierfür ist die steigende Beliebtheit der Sprache nach dem TIOBE Index¹, wie auch ein fünfter Platz in der Statistik von Coding Dojo².

Die Online-Lernplattform Udacity verwendet für den Kurs „Intro to Computer Science“ Python als Sprache, um die Themen der theoretischen Informatik zu erläutern. Diesen Online-Kurs haben bereits über 500.000 Personen besucht.³ Als Proargumente werden die Mächtigkeit, die leichte Erlernbarkeit und die weite Verbreitung aufgeführt.

¹ http://www.tiobe.com/tiobe_index (Stand 09.05.2016)

² <http://www.codingdojo.com/blog/9-most-in-demand-programming-languages-of-2016/> (Stand 09.05.2016)

³Stand 09.05.2016

Kapitel 3

Skripte ohne spezielle Module

Trotz dessen, dass die Erstellung des Python Moduls als Hauptbestandteil dieser Arbeit gesehen wird, ist es durchaus möglich einige SETLX Programme ohne zusätzliche, nicht enthaltene Module anzufertigen. Diese Skripte wurden als erstes angefertigt, um feststellen zu können, ob es möglich ist Python Syntax zu verwenden, ohne die Eleganz des Codes zu verlieren. Einer der Ziele der Übersetzung ist die Eleganz der Programme beizubehalten.

Das erste Codebeispiel aus dem Logik-Skript befasst sich mit der Berechnung einer Summe der Zahlen von 1 bis zur eingegebenen Zahl. Dieses Programm lässt sich auch nahezu eins-zu-eins so in Python abbilden. Das originale SETLX Programm verwendet hierfür eine Menge, die die Zahlen von 1 bis zur eingegebenen Zahl n enthält. Daraufhin wird die Summe aller in der Menge enthaltenen Zahlen mit dem „+/-“ Operator ermittelt und ausgegeben.

```
1  n := read("Type a natural number and press return: ");
2  s := +/ { 1 .. n };
3  print("The sum 1 + 2 + ... + ", n, " is equal to ", s, ".");
```

Abbildung 3.1: Einfache Summenberechnung in SETLX

In Python wurde fast dasselbe Verhalten nachgebildet. Jedoch wurde anstatt eine Menge anzufertigen eine Range der Zahlen von 0 bis n angelegt. Die Summe wird über die in Python bereits integrierte Funktion `sum()` berechnet und daraufhin ausgegeben.

```
1  n = int(input('Type a natural number and press return: '))
2  s = sum(range(n + 1))
3  print('The sum 1 + 2 + ... + ', n, ' is equal to ', s, '.')
```

Abbildung 3.2: Einfache Summenberechnung in Python

Allgemein kann gesagt werden, dass ein SETLX-Programm, ohne spezielle Funktionen oder Strukturen die nicht in Python wiedergefunden werden, meist eine große Ähnlichkeit mit der Python- Implementierung hat.

Kapitel 4

Python Modul lecture

Das Verhalten der Mengen in SETLX ist in einigen Bereichen anders als bei den Mengen in Python. Die Mengen in Python dürfen nur gewisse Werte enthalten, diese müssen unveränderbar sein. Zusätzlich werden auch gewisse Funktionen von den Python-Mengen nicht unterstützt, die in SETLX häufig verwendet werden. Aus diesen Gründen wurde das Python Modul lecture im Rahmen dieser Studienarbeit angefertigt.

Ein großes Problem, das sehr früh erkannt wurde, ist, dass die Mengen in Python auf den Hashwerten der enthaltenen Elemente operieren. Somit ist es beispielsweise verboten Mengen in Mengen zu hinterlegen, da diese verändert werden könnten und somit der alte Hashwert ungültig werden würde. Diese Funktionalität wird jedoch in den SETLX Programmen, der Vorlesungen mehrfach verwendet. Während analoge Python-Skripte den in Abbildung 4.1 gezeigten Fehler zurückgeben. Zwar könnten bereits vorhandene sets in frozensets umgewandelt werden um Mengen in Mengen zu ermöglichen. Allerdings würden auch hier die umgewandelten Mengen nicht mehr geändert werden können, da die Mengen nur so ihren Hashwert beibehalten. Die Unveränderbarkeit der Mengenelemente ist von Python bewusst gewählt. Eine Änderung wird durch die Art der Abspeicherung bereits verhindert. Den Elementen einer Menge werden Hashwerte zugewiesen und sobald diese feststehen dürfen sich die Elemente nicht mehr ändern, da das unmittelbar eine neue Hashberechnung verlangen würde. Eine weitere Möglichkeit wäre die Verwendung von Listen, anstelle von Mengen. Prinzipiell ist das in einigen Python-Übersetzungen der SETLX Programme möglich und wurde so auch teilweise umgesetzt. Es werden andere Datentypen verwendet um die Informationen zu hinterlegen, meist Listen statt Mengen, da für die Ausführung einiger der SETLX-Programme keine besonderen Eigenschaften der Mengen verwendet werden. Ein großes Problem an dieser Lösung ist allerdings, dass Listen nun mal keine Mengen sind und sobald Mengeneigenschaften oder Mengenoperatoren, die nicht für Listen gelten, verwendet werden, Listen eher ungeeignet sind. Der Workaround, besondere Funktionen für die Listen zu schreiben, um das Verhalten von Mengen zu imitieren, wurde auch als Ansatz bedacht, allerdings nach einigen kleinen Beispielübersetzungen wieder verworfen. Da eine wichtige Anforderung, die Erhaltung der Eleganz nicht erfüllt werden konnte. Somit war dies keine Lösung, die so für alles verwendet werden kann. Beispiele für die benötigten Funktionen, die implementiert werden mussten waren das Entfernen von Duplikaten aus einer Liste, so wie die Ermittlung der Differenz zweier Listen und die Ermittlung der Potenzmenge.

```
1 >Traceback (most recent call last):
2 > File "<stdin>", line 1, in <module>
3 >TypeError: unhashable type: 'set'
4
```

Abbildung 4.1: Fehler bei Mengen in Mengen

Allerdings sind die Mengen nicht der einzige Grund, warum das Modul benötigt wird. Ein in SETLX sehr hilfreiches Konstrukt, namens `match`, wird in Python nicht wiedergefunden. Dessen Syntax ist an die sehr bekannte Switch-Case-Syntax angelehnt, welche in Python nicht enthalten ist. Bei Matches können vier verschiedene Datentypen verwendet werden: Strings, Listen, Mengen und Terme. Das Matchen von Strings, Listen und Mengen kann für das Erkennen des ersten Zeichens und dem Rest oder auch das Herauspicken von Paaren verwendet werden. In dem SetlX-Tutorial beispielsweise wird das Matching zur Generierung des Inversen oder das Erstellen einer sortierten Liste aus einer Menge verwendet. [SH15]

Die interessanteste Anwendung von Matches findet sich jedoch, bei Termen. Die

[...]Art von Matchen [in SETLX] ist ähnlich zum Matching das in den Programmiersprachen Prolog und ML gegeben ist.

[SH15] Dieses Matching wird auch in einigen Programmen der Logik-Vorlesung, die als Grundlage dient, verwendet. Deshalb ist es wichtig, dass diese Funktion auch in einer Python Version der Programme möglich ist.

4.1 Sets

In den Vorlesungs-Programmen, die im Fokus dieser Arbeit stehen, werden häufig Mengen, sowie Ausdrücke aus der Mengenlehre sehr ähnlich zur mathematischen Darstellung verwendet. Neben den Mengenoperationen werden zusätzlich diverse Eigenschaften von Mengen implementiert. Beispielsweise wird genutzt, dass Mengen keine Duplikate enthalten. In SETLX wird eine Sortierung der Elemente durchgeführt, wodurch Vorteile in der Programmierung entstehen.

Um die Mengen, wie sie in den SETLX-Programmen verwendet werden, auch in Python verwenden zu können wurden eigene Mengen implementiert, die alle notwendigen Aufgaben erfüllen können.

Als Grundlage für die Mengen wird der Datentyp `SortedListWithKey` aus dem Modul `sortedcontainers` verwendet. Das Besondere an `SortedListWithKey` ist, dass es sich nicht nur um sortierte Listen handelt, sondern die Möglichkeit besteht festzulegen nach welcher Eigenschaft die Listen-Objekte sortiert werden sollen. Unter dem Schlüssel einer `SortedListWithKey` wird diese Eigenschaft hinterlegt und kann auch dort eingesehen werden. In diesem Fall sind die sortierten Listen, die die Elemente der Sets enthalten, als Key eingetragen.

In der Implementierung der Mengen werden für viele Operationen Operatoren verwendet, wobei versucht wurde möglichst nahe der SETLX-Implementierung zu bleiben. Die Ähnlichkeit zu SETLX soll vorhanden sein, da diese bereits sehr mathematisch ist und somit sinnvoller zu lernen ist für die Studenten. Die unterstützten mathematischen Operatoren sind in Tabelle 4.1 zu sehen.

Zusätzlich gibt es noch weitere Methoden, die nicht in der mathematischen Darstellung verwendbar sind, dennoch für typische mathematische Operationen verwendet werden. Diese werden anstatt in der mathematischen Notation, wie Funktionen auf die Mengen ausgeführt. Außer diesen mathematischen Funktionen können noch weitere, nicht-mathematische Operationen, wie beispielsweise die visuelle Darstellung von Mengen, verwendet werden. Eine sehr wichtige Funktion ist die `put`-, beziehungsweise die `_put`-Funktion. Diese sorgt dafür, dass keine Duplikate in ein Set eingetragen werden können. Die `put`-Funktion ruft nur die `_put`-Funktion auf und ist somit keine neue Methode. In Tabelle 4.2 sind die restlichen Mengen-Funktionen aufgelistet.

An dieser Stelle muss noch bemerkt werden, dass die Selektion des i -ten Elements einer Menge auch das Array Slicing (z.B. `[1:2]`), wie es von Python bekannt ist, unterstützt. Mit diesen Operatoren und Funktionen können alle Aufgabestellungen, die in der Vorlesung zur theoretischen Informatik vorkommen, bewältigt werden. Im Nachfolgenden wird eine Anwendung der Mengenoperatoren gezeigt. Das SETLX-Programm, das die Grundlage bietet, wird den Studenten gezeigt, damit sie ein Gefühl dafür bekommen, welche Operation welche Resultate liefert.

In den Beispielen, die in Abbildung 4.2 und in Abbildung 4.3 zu sehen sind, werden einige Operatoren sowohl in SETLX, als auch in Python dargestellt. Zuerst werden die Vereinigung, der Schnitt und die Differenz zweier Mengen gebildet. Daraufhin die Bildung einer Potenzmenge, die

Operator	Bedeutung
+	Bildet Vereinigung zweier Mengen
+=	Ergebnis der Vereinigung wird in die erste Menge geschrieben
-	Bildet Differenz zweier Mengen
*	Bildet Schnitt zweier Mengen
2**	Bildet Potenz einer Menge
%	Bildet symmetrische Differenz zweier Mengen
<	Gibt True zurück wenn die linke Menge kleiner ist
>	Gibt True zurück wenn die linke Menge größer ist
>=	Gibt True zurück wenn die rechte Menge eine Teilmenge der linken ist
<=	Gibt True zurück wenn die linke Menge eine Teilmenge der rechten ist
==	Gibt True zurück wenn beide Mengen dieselben Elemente enthalten
!=	Gibt True zurück wenn die Mengen unterschiedlich sind (Gegenteil zu ==)
in	Prüft ob das gegebene Element in der Menge ist

Tabelle 4.1: Mathematische Operatoren für Sets

Funktion	Bedeutung
str	Stellt die Menge als String dar
contains	Dasselbe wie in (siehe Tabelle 4.1)
Set[i]	Gibt das Element an i-ter Stelle zurück
cartesian_product	Bildet das kartesische Produkt zweier Mengen
arb	Gibt ein beliebiges (in diesem Fall das erste) Element der Menge zurück
random	Gibt ein zufälliges Element der Menge zurück
put/_put	Fügt ein Element in die Menge ein
peek	Gibt das letzte Element der Menge zurück
pop	Gibt das letzte Element der Menge zurück und entfernt es aus der Menge
sum	Gibt die Summe aller Elemente in der Menge zurück

Tabelle 4.2: Funktionen für Sets

```

1  a := { 1, 2, 3 };
2  b := { 2, 3, 4 };
3
4  c := a + b;
5  print(a, " + ", b, " = ", c);
6
7  c := a      * b;
8  print(a, " * ", b, " = ", c);
9
10 c := a - b;
11 print(a, " - ", b, " = ", c);
12
13 c := 2 ** a;
14 print("2 ** ", a, " = ", c);
15
16 print("( ", a, " <= ", b, ") = ", (a <= b));
17
18 print("1 in ", a, " = ", 1 in a);

```

Abbildung 4.2: Nutzung von Mengen in *simple.stl*

```

1  a = Set(1,2,3)
2  b = Set(2,3,4)
3
4  c = a + b
5  print('%s + %s = %s' % (a, b, c))
6
7  c = a * b
8  print('%s * %s = %s' % (a, b, c))
9
10 c = a - b
11 print('%s - %s = %s' % (a, b, c))
12
13 c = 2 ** a
14 print('2 ** %s = %s' % (a, c))
15
16 print('%s <= %s = %s' % (a, b, a <= b))
17
18 print('1 in %s = %s' % (a, 1 in a))

```

Abbildung 4.3: Nutzung von Mengen in *simple.py*

Prüfung einer Teilmengen-Relation und die Prüfung, ob ein Element sich in einer Menge befindet, gezeigt.

Die Syntax zur Erzeugung einer Menge unterscheidet sich bereits, allerdings sind alle Operatoren komplett identisch. Ein weiterer Unterschied der beiden Implementierungen ist die unterschiedliche Erstellung der Ausgabestrings. Dieser Unterschied ist jedoch irrelevant, da im Vordergrund steht, wie die Operatoren eingesetzt werden können.

4.2 Matches

Die Implementierung der Match-Strukturen ist in dem lecture-Module unter dem Verzeichnis `util` in der Datei `parser.py` als Klasse mit dem Titel `MatchParser` zu finden. Für diese Klasse ist es wichtig, dass, die auch im Modul `util` befindlichen Klassen, `TokenType` und `Scanner`, so wie die Hilfsfunktion `is_number` benötigt werden. `TokenType` enthält die IDs für die verschiedenen Token-Arten die auftreten können, `Scanner` erstellt aus einem String eine Liste von Tokens und `is_number` überprüft ob eine Zahl an die Funktion übergeben wurde.

Der Parser erkennt gewisse Operatoren, Funktionen und Klammerungen. Die unterstützten Operatoren sind „+“, „-“, „*“, „/“, „%“, „**“, „&&“, „||“, „<“, „>“, „<=“, „>=“, „==“, „!=“ und „!“. Die unterstützten Funktionen sind „sin“, „log“, „exp“, „cos“, „tan“, „asin“, „acos“, „atan“, „sqrt“ und „ln“. Die erkannte Klammerung besteht nur aus der runden öffnenden Klammer „(“ und der runden schließenden Klammer „)“.

Die wichtigste Funktion für den Benutzer ist `match(self, scheme, value)`. Da die Funktion auf einem erzeugten `MatchParser` ausgeführt wird, sind nur die Variablen `scheme` und `value` für den Anwender interessant. Unter `scheme` wird der zu parsende Ausdruck gegeben und `value` enthält den Wert nach dem gematched werden soll. Wichtig hierbei ist, dass das Matching nur auf Strings basierend ausgeführt werden kann, während Matches in SETLX auch die Verwendung von Literals und die direkte Operation auf den Ausgaben ermöglicht. Dieser Unterschied ist bei einem direkten Vergleich im Code sofort erkennbar. Im Nachfolgenden wird ein Match-Konstrukt, das mathematische Funktionen ableiten soll, in SETLX, mit der neuen Struktur, wie sie in Python entwickelt wurde, verglichen.

```
1  diff := procedure(t, x) {
2      match (t) {
3          case a + b :
4              return diff(a, x) + diff(b, x);
5          case a - b :
6              return diff(a, x) - diff(b, x);
7          case a * b :
8              return diff(a, x) * b + a * diff(b, x);
9      ...
}
```

Abbildung 4.4: Ausschnitt aus `diff()` (SETLX)

Was direkt auf den ersten Blick auffällt ist, dass der Code, der in SETLX sehr kompakt dargestellt wird, deutlich umfangreicher ist. Dementsprechend leidet auch die Leserlichkeit unter der Python-Version. Es ist nicht direkt klar, wie der Code zu lesen ist, da die Ausdrücke als Strings abgebildet sein müssen. Während in Abbildung 4.4 im `return` die Ableitregeln, durch rekursive Aufrufe von `diff()`, zu den mathematischen Funktionen im jeweiligen `case` stehen, sind in Abbildung 4.5 dieselben mathematischen Funktionen als Strings im `match.match`-Teil zu erkennen, allerdings ist nicht sofort ersichtlich was im `return`-Statement steht. Der String, der zurückgegeben wird enthält dieselben Ableitregeln wie sie im SetlX-Code zu sehen sind, allerdings werden die Variablen nicht direkt genannt, sondern durch Platzhalter dargestellt. In den Parametern der `format`-Funktion werden die Platzhalter gefüllt. Die Platzhalter mit dem Präfix „diff_“ werden rekursiv Abgeleitet, wobei dem erneuten `diff`-Aufruf die Werte, die im Match für die jeweilige Variable hinterlegt sind und das „x“ weil nach x abgeleitet wird, übergeben werden. Wenn ein Platzhalter kein Präfix besitzt, so werden nur die Werte aus dem Match herausgelesen und eingesetzt.

```

1  def diff(t,x):
2      match = Match()
3      if match.match('a+b', t):
4          return '{diff_a} + {diff_b}'.format(
5              diff_a=diff(match.values['a'], x),
6              diff_b=diff(match.values['b'], x))
7      elif match.match('a-b', t):
8          return '{diff_a} - {diff_b}'.format(
9              diff_a=diff(match.values['a'], x),
10             diff_b=diff(match.values['b'], x))
11     elif match.match('a*b', t):
12         return '{diff_a} * {b} + {a} * {diff_b}'.format(
13             diff_a=diff(match.values['a'], x),
14             b=match.values['b'], a=match.values['a'],
15             diff_b=diff(match.values['b'], x))
16     ...

```

Abbildung 4.5: Ausschnitt aus `diff()` (Python)

4.3 Übersetzung komplexerer Programme

Es wurden zwar einige SETLX-Programme in Python-Skripte übersetzt, allerdings werden in dieser Arbeit hauptsächlich Programme, die die Eleganz der Programmiersprache SETLX verdeutlichen, genauer betrachtet.

Wie zuvor beschrieben, ermöglicht SETLX dem Programmierer in einem sehr mathematischen Stil zu programmieren. Somit können Personen, die ersten Berührungen mit der Mengenlehre oder von der Mathematik kommen, sowie Studenten, die mathematische Konstrukte verstehen und anwenden müssen, beim Programmieren diese Erfahrungen sammeln.

4.3.1 Schiebepuzzle

Das Schiebepuzzle ist eine Aufgabe die den Studenten mit Lücken als Aufgabe gegeben wird, um Vorlesungsinhalte direkt anwenden zu können. Mit diesem Programm sollen die Studenten eine für Menschen nicht triviale Lösung zu einem Schiebepuzzle berechnen lassen. Aufgrund der Berechnung aller möglichen Pfade, das Puzzle zu lösen, lässt sich das Programm nicht so schnell wie die meisten anderen SETLX-Programme durchführen.

Sowohl das SETLX-Programm, wie auch die Übersetzung in Python definieren zu Beginn die Funktion, mit der aus einem State (einem derzeitigen Zustand des Puzzles, abgelegt in einer Liste) ein String erzeugt werden kann, um eine bessere Visualisierung zu ermöglichen. Bei der Übersetzung ist in dieser Methode nichts großartig Interessantes zu sehen, da in den meisten Zeilen fast eins-zu-eins dasselbe steht. Allerdings ist zu beachten, dass die `for`-Schleifen in SETLX über Listen von 1-3 iterieren, während in Python dafür eine `Range` mit den Werten 0-2 verwendet wird. Es wird allerdings die selbe Ausführung erreicht, da Listen-Indizes in SETLX bei 1 anfangen, während Python die 0 als Index verwendet, um das erste Element aufzurufen.

Die Funktion, mit der letztendlich auch der Pfad vom Start-Zustand zum Ziel-Zustand ermittelt wird, ist in Python die `find_path` Methode. Die Parameter, die übergeben werden, sind identisch zu der SETLX-Implementierung. Die am Anfang der Funktion definierten Variablen weichen vom SETLX-Code um eine Variable ab. Die Variablen `paths` und `states` sind in beiden Versionen zu finden und der Integer `count_iteration` ist in SetlX als `count` zu finden. Allerdings werden zur Prüfung ob neue Zustände hinzukommen unterschiedliche Ansätze verwendet. In der Python-Implementierung keine Menge mit allen entdeckten Zuständen, sondern die Prüfung der Anzahl der verschiedenen Zustände verwendet. Somit vergleicht die äußere `while`-Schleife zwei Integer,

```

1  findPath := procedure(start, goal, nextStates) {
2      count      := 1;
3      paths      := { [start] };
4      states     := { start };
5      explored   := {};
6      while (states != explored) {
7          print("iteration number $count$");
8          count += 1;
9          explored := states;
10         paths    := { l + [s]
11                     : l in paths, s in nextStates(l[-1])
12                     | !(s in states)
13                     };
14         states   += { p[-1] : p in paths };
15         print("number of states: $#states$");
16         if (goal in states) {
17             return arb({ l : l in paths | l[-1] == goal });
18         }
19     }
20 };

```

Abbildung 4.6: findPath im Schiebepuzzle (SETLX)

```

1  def find_path(start, goal, next_states):
2      count_iteration = 1
3      count_states    = 0
4      paths           = Set([start])
5      states          = Set(start)
6      while len(states) != count_states:
7          count_states = len(states)
8          print('Iteration number %s' % count_iteration)
9          count_iteration += 1
10         paths = Set(x + [s]
11                     for x in paths for s in next_states(x[-1])
12                     if not s in states)
13         states += Set(p[-1] for p in paths)
14         print('Number of states: %s' % len(states))
15         if goal in states:
16             return Set(l for l in paths if l[-1] == goal).arb()

```

Abbildung 4.7: find_path im Schiebepuzzle (Python)

während die Vorlage zwei Mengen vergleicht. Abgesehen davon ist der Ablauf sehr ähnlich. Es wird die derzeitige Iteration angegeben, daraufhin die neuen Pfade anhand von `next_states` ermittelt und dann alle derzeitigen Zustände der Pfade ermittelt und die Anzahl ausgegeben. Zuletzt wird noch im Falle, dass das Ziel bereits erreicht wurde, ein beliebiger Lösungspfad zurückgegeben.

In `find_path` muss, für die Ermittlung der möglichen Pfade, die Funktion `next_states` aufgerufen werden. Diese gibt eine Liste zurück, die alle erreichbaren Zustände vom Zustand `state`, der als Parameter übergeben wird, enthält. In SETLX wird diese Liste als Menge zurückgegeben, allerdings werden Listen in diesem Fall bevorzugt, da sie in Python geläufiger verwendet werden und

```

1  nextStates := procedure(state) {
2      directions := { [1, 0], [-1, 0], [0, 1], [0, -1] };
3      [row, col] := findBlank(state);
4      return { moveDir(state, row, col, [dx, dy])
5              : move in directions
6                | row + dx in {1, 2, 3} && col + dy in {1, 2, 3}
7              };
8  };

```

Abbildung 4.8: nextStates im Schiebepuzzle (SETLX)

```

1  nextStates := procedure(state) {
2      directions := { [1, 0], [-1, 0], [0, 1], [0, -1] };
3      [row, col] := findBlank(state);
4      return { moveDir(state, row, col, [dx, dy])
5              : move in directions
6                | row + dx in {1, 2, 3} && col + dy in {1, 2, 3}
7              };
8  };

```

Abbildung 4.9: next_states im Schiebepuzzle (Python)

keine Mengeneigenschaften in dieser Situation benötigt werden. Die Richtungen werden allerdings nicht wie im SETLX-Programm als Menge von Listen, sondern als Liste von Tupeln definiert, da Tupel in Python, so wie Listen in SETLX, unzipped werden können. Eine Änderung, die an der Vorlage unternommen wurde (Abbildung 4.8 Zeile 4) ist, dass die Bewegung von einer Variable `move` auf zwei Variablen `dx` und `dy` aufgeteilt wurden.

```

1  moveDir := procedure(state, row, col, dir) {
2      [dx, dy] := dir;
3      nextState := state;
4      nextState[row][col] := state[row + dx][col + dy];
5      nextState[row + dx][col + dy] := 0;
6      return nextState;
7  };

```

Abbildung 4.10: moveDir im Schiebepuzzle (SETLX)

```

1  def move_dir(state, row, col, direction):
2      (dx, dy) = direction
3      next_state = [list(x) for x in state]
4      next_state[row][col] = next_state[row + dx][col + dy]
5      next_state[row + dx][col + dy] = 0
6      return next_state

```

Abbildung 4.11: move_dir im Schiebepuzzle (Python)

Die Methode `move_dir` erhält dieselben Parameter wie in der Vorlage und gibt den nächsten Zustand zurück, nachdem vom derzeitigen Zustand aus eine Bewegung in entweder die x- oder die y-Achse erfolgt. Einer der Unterschiede zur ursprünglichen Umsetzung ist hier, wie zuvor bereits erwähnt wurde, dass die Richtung als Tupel und nicht als Liste gewertet wird. (Siehe Abbildung 4.11 Zeile 2)

Ein erheblicher Unterschied zwischen den beiden Implementierungen ist das initiale Setzen der Variable `next_state`. Während in SETLX der Parameter `state` dafür verwendet wird und eine einfache Zuweisung erfolgt, muss in Python die `list()` Funktion auf alle Listen in `state` verwendet werden (Abbildung 4.11 Zeile 3). Grund dafür ist, dass ansonsten die Referenzen übergeben werden und somit dann die Werte von `state` ebenfalls geändert werden, wenn sie in `next_state` bearbeitet werden. Durch die `list()`-Funktion wird eine Kopie erzeugt und somit sind `next_state` und `state` zwei unabhängige Listen. SETLX erkennt an dieser Stelle intern, ob ein Objekt geändert wurde und erstellt gegebenenfalls eine Kopie. Deshalb wird dieser Aufruf nur in der Python-Version benötigt.

```

1  findBlank := procedure(state) {
2      for (row in [1 .. 3]){
3          for (col in [1 .. 3]){
4              if (state[row][col] == 0){
5                  return [row, col];
6              }
7          }
8      }
9  };

```

Abbildung 4.12: `findBlank` im Schiebepuzzle (SETLX)

```

1  def find_blank(state):
2      for row in range(3):
3          for col in range(3):
4              if state[row][col] == 0:
5                  return (row, col)

```

Abbildung 4.13: `find_blank` im Schiebepuzzle (Python)

Damit in `next_states` eine Zahl „bewegt“ werden kann, muss über `find_blank` das freie Feld gefunden werden. In der Vorlage wird in `findBlank` eine Menge erzeugt, die alle Reihen-Zeilen-Kombinationen enthält und aus dieser einer der Werte, an denen der übergebene Zustand die 0 enthält, zurückgegeben. In dieser Arbeit wird nur ein Tupel mit einem Reihen- und einem Zeilenwert, an denen der Zustand die Null enthält zurückgegeben. Zu Vergleichszwecken ist die Funktion, so wie sie in Python geschrieben wurde, auch in SETLX implementiert. In SETLX merkt man keine Unterschiede in der Performance. Diese Methode zeigt wie ähnlich der Code in SETLX und Python sein können.

Nachdem die Funktionen alle definiert sind ist der Ablauf komplett identisch zur Vorlage. Es wird die Zeitmessung begonnen, der Start- und End-Zustand definiert, daraufhin der Pfad ermittelt und die Zeitmessung beendet. Abschließend werden der Lösungspfad und die Zeitmessungsergebnisse ausgegeben.

Im Allgemeinen ist der Code in beiden Programmiersprachen sehr ähnlich und der Ablauf, so wie die Syntax, teilweise sogar identisch. Einige Ausdrücke sehen auf Grund der Programmiersprache unterschiedlich aus, erfüllen aber denselben Zweck. Bei der Zeitmessung beispielsweise wird in SETLX nur die Methode `now()` aufgerufen, während in Python `timeit.default_timer()` aufge-

rufen wird und sogar ein Import dafür notwendig ist. Andere Abweichungen sind unterschiedliche Datentypen, die in Python gewählt wurden. Diese werden verwendet um, die Laufzeit etwas zu verbessern, weil die Implementierung, die für Mengen verwendet werden muss, nicht so effizient wie die SETLX-Implementierung ist. An einigen Stellen sind aber auch keine Mengen notwendig und werden deshalb durch Listen ersetzt.

Die genaue Zeit, wie lange das Programm für die Berechnung gebraucht hat, wird in der Kommandozeile ausgegeben. Somit wird den Studenten klar, dass selbst der Rechner diese Berechnungen nicht sofort liefern kann. Um einen Vergleich der Performance von SETLX zu Python zu haben, wird auch die Berechnungszeit der Python-Implementierung aufgeführt. Die Eigenschaften des Rechners, mit dem die Berechnungen durchgeführt wurden sind:

- Prozessor: Intel i7 6700hq 2,6-3,1 GHz
- Hauptspeicher: 8 GB RAM

Das SETLX-Programm lief in 17,4 Sekunden, während das Python-Skript 46,6 Sekunden für die Berechnung benötigte. Auffällig ist, dass in SETLX die Ausführung über doppelt so schnell wie bei der Python-Implementierung ist. Grund hierfür ist, dass die virtuelle Maschine, in der Java ausgeführt wird, etwas effizienter als die virtuelle Maschine von Python ist. Da SETLX auf Java basiert, wird die Effizienz von Java zu Python verglichen. Außer dem Unterschied bei den virtuellen Maschinen, unterstützt die Programmiersprache Java zusätzlich eine statische Typisierung. Python hingegen unterstützt, wie es für Skriptsprachen üblich ist, eine dynamische Typisierung, die etwas ineffizienter ist. Eine statische Typisierung ist effizienter da der Rechenaufwand für eine Typüberprüfung wegfällt. Der allgemeine Leistungsunterschied der Sprachen ist unter <https://benchmarkgame.alieth.debian.org/u64q/python.html> zu sehen. Python schneidet in fast allen Tests deutlich schlechter ab als Java. Eine unterschiedliche Implementierung der für diese Arbeit entwickelten Mengen und der Mengen, die in SETLX verwendet werden, kann nicht die Ursache für diese Abweichungen in der Performanz sein. Die Sets des Python-Moduls „lecture“ wurden basierend auf die Implementierung, wie die in SETLX verwendet wird, umgesetzt.

4.3.2 Watson

Oftmals werden Rechner für einfachere Rechenoperationen verwendet, für die ein Mensch bereits die Überlegungen zur Logik getätigt hat. Für die Entwickler wird es interessant, wenn der Rechner auch komplexe Zusammenhänge erkennen soll. Dieses Umfeld ist als künstliche Intelligenz bekannt und stellt neue Herausforderungen dar.

Die Aufgabe Watson soll anhand von gegebenen Tatsachen den oder die Täter aus drei Verdächtigen herausfinden.

Die Methode `evaluate` wird bereits am Anfang des Programms definiert und gibt an, wie verschiedene Ausdrücke ausgewertet werden sollen. Es sind ein paar kleine Unterschiede in beiden Umsetzungen zu erkennen. In der Python-Implementierung werden `True` und `False` nicht als Ausdrücke behandelt. Das liegt daran, dass die Schlüsselwörter „True“ und „False“ in der Match-Implementierung nicht eingetragen sind und es somit zu Problemen kam, dass alle Variablen in diese Fälle gelaufen sind. Außerdem hat das Herauslassen dieser beiden Fälle keine negativen Auswirkungen auf die Ausführung des Skripts. Wenn der Ausdruck $g \Rightarrow h$ gelesen wird, wird in SETLX die eingebaute Implikation aufgerufen. Da Python keinen Implikations-Operator besitzt, wird an dieser Stelle die Umschreibung für diesen Junktor verwendet. Die Implikation von g und h kann auch als $\neg g \vee h$ geschrieben werden.

Die Methode `createValuation` gibt eine Menge mit Tupel, die für jedes Element in v angeben, ob es sich auch in m befindet. In `create_valuation` wird das gleiche Resultat geliefert, allerdings befinden sich die Tupel in einer Liste. Die Struktur wurde so gewählt, da zu einem späteren Verlauf aus einer solchen Menge/Liste der Wahrheitswert (x in m) durch das Aufrufen aus der Liste, anhand der Bezeichnung des Elements abgefragt wird. In Python ist das mit einem Dictionary möglich, jedoch unterstützt Python eine Sortierung von Dictionaries nicht, weshalb diese nicht in die Sets eingefügt werden können. Das heißt dass keine Dictionaries in Sets möglich sind.

```

1  evaluate := procedure(f, i) {
2      match (f) {
3          case true:      return true;
4          case false:     return false;
5          case ^variable(p): return i[p];
6          case !g:        return !evaluate(g, i);
7          case g && h:     return evaluate(g, i) && evaluate(h, i);
8          case g || h:     return evaluate(g, i) || evaluate(h, i);
9          case g => h:     return evaluate(g, i) => evaluate(h, i);
10         case g <==> h:   return evaluate(g, i) == evaluate(h, i);
11         default:        abort("syntax error in evaluate($f$, $i$)");
12     }
13 };
14

```

Abbildung 4.14: evaluate in Watson (SETLX)

```

1  def evaluate(f, i):
2      match = Match()
3      if match.is_variable(f):
4          return i[f]
5      elif match.match('!g', f):
6          return not evaluate(match.values['g'], i)
7      elif match.match('g && h', f):
8          return evaluate(match.values['g'], i) and
9                 evaluate(match.values['h'], i)
10     elif match.match('g || h', f):
11         return evaluate(match.values['g'], i) or
12                evaluate(match.values['h'], i)
13     elif match.match('g => h', f):
14         return not(evaluate(match.values['g'], i)) or
15                evaluate(match.values['h'], i)
16     elif match.match('g <==> h', f):
17         return evaluate(match.values['g'], i) ==
18                evaluate(match.values['h'], i)
19     else:
20         raise SyntaxError('Syntax error in evaluate(%s,%s)' %
21                             (f, i))
22

```

Abbildung 4.15: evaluate in Watson (Python)

```

1  createValuation := procedure(m, v) {
2      return { [ x, x in m ] : x in v };
3  };

```

Abbildung 4.16: createValuation in Watson (SETLX)

Die Struktur, Paare (zweistellige Tupel) in Listen zu hinterlegen, können allerdings über den `dict()`-Befehl in ein Dictionary konvertiert werden. Somit wird die Liste zum Abspeichern in

```

1  def create_valuation(m, v):
2      return [(x, x in m) for x in v]

```

Abbildung 4.17: `create_valuation` in Watson (Python)

einem Set verwendet und sobald die Valuation wieder ausgelesen wird, wird diese als Dictionary weiterverwendet.

Nun werden die Tatsachen, die zu dem Mordfall bekannt sind, als Strings von booleschen Ausdrücken eingelesen. In SetlX werden diese über den `parse`-Befehl in Variablen gelesen, während in Python die Strings direkt in die Variablen `f1` bis `f6` gespeichert werden.

```

1  fs := { f1, f2, f3, f4, f5, f6 };
2  v  := { "a", "b", "c" };
3  p := 2 ** v;
4  print("p = ", p);
5  b := { createValuation(m, v) : m in p };
6  s := { i : i in b | forall (f in fs | evaluate(f, i)) };
7  print("Set of all valuations satisfying all facts: ", s);
8  if (#s == 1) {
9      i := arb(s);
10     offenders := { x : x in v | i[x] };
11     print("Set of offenders: ", offenders);
12 }

```

Abbildung 4.18: Lösung von Watson (SETLX)

```

1  fs = [f1, f2, f3, f4, f5, f6]
2  v = Set('a', 'b', 'c')
3  p = 2 ** v
4  print('p = ', p)
5  b = Set(create_valuation(m, v) for m in p)
6  s = [dict(i) for i in b if all(evaluate(f, dict(i)) for f in fs)]
7  print('List of all valuations satisfying all facts: ', s)
8  if len(s) == 1:
9      i = s[0]
10     offenders = [x for x in v if i[x]]
11     print('List of offenders: ', offenders)

```

Abbildung 4.19: Lösung von Watson (Python)

Die Tatsachen werden alle in einer Liste/Menge gespeichert und die Täter, als Buchstaben „a“, „b“ und „c“, in einer Menge abgelegt. Daraufhin wird die Potenzmenge der Täter gebildet, in der alle möglichen Täter-Kombinationen enthalten sind und Wahrheitswerte den Variablen zugewiesen. Bei der Ermittlung, welche Täterkombinationen möglich sind, wird in SETLX die Funktion `forall` und in Python `all` verwendet. Beide Funktionen geben nur dann `True` zurück, wenn alle Elemente der angegebenen Liste/Menge ebenfalls den Wahrheitswert `True` besitzen. In Python werden die Variable-Boolean-Paare in Dictionaries gespeichert, um den Aufruf für `evaluate` und die Ausgabe zu erleichtern. Zuletzt wird das Ergebnis noch ausgegeben.

4.3.3 Wolf Ziege Kohl

Bei der Wolf-Ziege-Kohl-Aufgabe besteht die Herausforderung darin, dass ein Bauer einen Wolf, eine Ziege und einen Kohl von einer Seite eines Flusses auf die andere bringen möchte. Das Problem dabei ist, dass er nur ein Element auf einmal transportieren kann. Wenn er den Wolf mit der Ziege alleine lässt, so frisst der Wolf die Ziege und wenn er die Ziege mit dem Kohl alleine lässt so frisst die Ziege den Kohl. Diese Problemstellung sollen die Studenten anhand eines Programms lösen. Die Aufgabe wird auch als wolf-goat-cabbage (wgc) bezeichnet.

```
1  findPath := procedure(x, y, r) {
2      p := { [x] };
3      while (true) {
4          oldP := p;
5          p := p + pathProduct(p, r);
6          found := { l : l in p | l[-1] == y };
7          if (found != {}) {
8              return arb(found);
9          }
10         if (p == oldP) {
11             return;
12         }
13     }
14 }
```

Abbildung 4.20: findPath in wgc (SETLX)

```
1  def find_path(x, y, r):
2      p = Set([x])
3      while True:
4          old_p = p
5          p = p + path_product(p, r)
6          found = Set(l for l in p if l[-1] == y)
7          if found:
8              return found[0]
9          if p == old_p:
10             return
```

Abbildung 4.21: find_path in wgc (Python)

Die Funktion `find_path` ermittelt, ob es einen Pfad vom Knoten `x` zum Knoten `y` innerhalb von `r` gibt und gibt diesen zurück, sofern er vorhanden ist. In der Variable `p` befinden sich alle möglichen Pfade, die gefunden werden. Pro Schleifendurchgang werden die nächst mögliche Pfadschritte berechnet und geprüft, ob die Lösung sich danach bereits in `p` befindet. Die erste Abbruchbedingung tritt ein, wenn die Lösung gefunden wurde. In dem Fall wird die Lösung zurückgegeben. Die zweite Abbruchbedingung ist, wenn bei der Ermittlung des nächsten Schritts keine neuen Pfade entdeckt wurden. In diesem Fall gibt die Methode keinen Wert zurück.

Die Abläufe sind in beiden Programmiersprachen nahezu identisch. Die erste Abbruchbedingung wird syntaktisch anders formuliert. Während in SETLX geprüft wird, ob die Menge `found` nicht leer ist (Abbildung 4.20 Zeile 7), wird in Python geprüft ob `found` Elemente enthält (Abbildung 4.21 Zeile 7). Diese beiden Aussagen sind logischerweise dieselben. In der nächsten Zeile wird daraufhin in SETLX ein beliebiges Element, während die Python-Implementierung immer

das erste Element der Menge zurückgibt. Es wäre auch möglich die, in den Python Sets implementierte, Funktion `arb()` aufzurufen, allerdings gibt diese auch nur das erste Element zurück. Um das Skript einfach und übersichtlich zu halten, wurde das direkte Aufrufen des ersten Elements bevorzugt.

```

1  pathProduct := procedure(p, q) {
2      return { add(x,y) : x in p, y in q |
3          x[-1] == y[1] && noCycle(x,y) };
4  };

```

Abbildung 4.22: `pathProduct` in `wgc` (SETLX)

```

1  def path_product(p, q):
2      return Set(add(x, y) for x in p for y in q
3          if x[-1] == y[0] and no_cycle(x, y))

```

Abbildung 4.23: `path_product` in `wgc` (Python)

In `path_product` werden die nächsten Schritte, die von `p` aus nach `q` erreichbar sind zurückgegeben.

Wie sich leicht erkennen lässt, sind diese beiden Implementierungen gleich. Das einzige, das zu beachten ist, ist, dass das erste Element einer Liste in SETLX den Index 1 besitzt, während es in Python unter dem Index 0 zu finden ist.

```

1  noCycle := procedure(l1, l2) {
2      return #({ x : x in l1 } * { x : x in l2 }) == 1;
3  };

```

Abbildung 4.24: `noCycle` in `wgc` (SETLX)

```

1  def no_cycle(l1, l2):
2      length = len(Set(x for x in l1) * Set(x for x in l2))
3      return length == 1

```

Abbildung 4.25: `no_cycle` in `wgc` (Python)

Die Funktion `no_cycle` prüft, ob es zu Zyklen kommen kann, indem geprüft wie viele gleiche Elemente in `l1` und `l2` enthalten sind. Nur wenn es nur ein Element, das in beiden Listen enthalten ist, gibt, tritt kein Zyklus auf.

Hier fällt auf, dass es nur syntaktische Unterschiede zwischen den beiden Implementierungen gibt.

Die Rückgabe von `add` ist die Summe der Mengen `p` und `q`, wobei das erste Element von `q` das letzte von `p` sein muss.

An dieser Stelle muss wieder auf die unterschiedliche Indexierung der beiden Sprachen verwiesen werden. Beide Methoden greifen auf das zweite Element von `q` zu, benötigen aber dafür unterschiedliche Indizes.

```

1  add := procedure(p, q) {
2      return p + [ q[2] ];
3  };

```

Abbildung 4.26: add in wgc (SETLX)

```

1  def add(p, q):
2      return p + [q[1]]

```

Abbildung 4.27: add in wgc (Python)

Wgc ist in drei Bereiche aufgeteilt. Der erste Teil, der aus den eben beschriebenen Funktionen besteht, beinhaltet die Methoden, die zur Problemlösung benötigt werden. Der zweite Teil, der im Nachfolgenden beschrieben wird, beinhaltet den problemspezifischen Code. Der letzte Teil behandelt die visuelle Darstellung der Lösung. Auf diesen Teil wird nicht genauer eingegangen, da nur die Lösung von Problemen in der theoretischen Informatik im Vordergrund steht.

```

1  problem := procedure(s) {
2      return !("farmer" in s) &&
3          ("goat" in s && "cabbage" in s ||
4          "wolf" in s && "goat" in s);
5  };

```

Abbildung 4.28: problem in wgc (SETLX)

```

1  def problem(s):
2      return not 'farmer' in s and \
3          (('goat' in s and 'cabbage' in s) or
4          ('wolf' in s and 'goat' in s))

```

Abbildung 4.29: problem in wgc (Python)

Die Methode `problem` prüft, ob es bei dem Pfad `s` zu Problemen kommen kann. Mit Problemen sind die Restriktionen in der Aufgabenstellung, wie beispielsweise der Wolf und die Ziege dürfen nicht unbeaufsichtigt alleine sein, gemeint.

Hierzu ist nicht viel zu sagen, da beide Implementierungen bis auf die Syntax komplett gleich sind.

Da nun alle benötigten Funktionen definiert sind, kann das Problem in Angriff genommen werden. Die Menge `wgc_all` wurde anders benannt als in SETLX, da `all()` bereits eine eingebaute Funktion in Python ist. In der Menge befinden sich die Bezeichnungen der Elemente, die in der Problemstellung beschrieben wurden. In `p` werden daraufhin alle möglichen Kombinationen der Elemente in `wgc_all`, die zu keinen Problemen führen, festgehalten. In `r1` findet man alle Möglichkeiten, in denen der Bauer eine Situation in `p` mit einem weiteren Element oder alleine verlässt. Die Menge `r` wird aus einer Vereinigung von `r1` und `r2` (dem Inversen von `r1`) gebildet. Nun kann ein möglicher Lösungspfad mit `find_path()` ermittelt werden, indem als Start `wgc_all`, als Ziel eine leere Menge und als mögliche Pfade `r` übergeben werden.

```

1  all := { "farmer", "wolf", "goat", "cabbage" };
2  p   := { s : s in 2 ** all | !problem(s) && !problem(all - s) };
3  r1  := { [s, s - b] : s in p, b in 2 ** s
4         | s - b in p && "farmer" in b && #b <= 2
5         };
6  r2  := { [y, x] : [x, y] in r1 };
7  r   := r1 + r2;
8
9  start := all;
10 goal  := {};
11
12 path  := findPath(start, goal, r);

```

Abbildung 4.30: Lösung des wgc-Problems (SETLX)

```

1  wgc_all = Set('farmer', 'wolf', 'goat', 'cabbage')
2  p       = Set(s for s in 2 ** wgc_all if not problem(s)
3             and not problem(wgc_all - s))
4  r1      = Set([s, s - b] for s in p for b in 2 ** s
5             if (s - b) in p and 'farmer' in b and
6             len(b) <= 2)
7  r2      = Set([y, x] for [x, y] in r1)
8
9  r       = r1 + r2
10 start  = wgc_all
11 goal   = Set()
12 path   = find_path(start, goal, r)

```

Abbildung 4.31: Lösung des wgc-Problems (Python)

Im Allgemeinen kann gesagt werden, dass dieses Programm mit Hilfe der Set-Implementierung sehr gut in Python nachkonstruiert werden konnte. Im Python-Skript sind nur wenige Abweichungen auffindbar.

4.3.4 8 Damen Problem

Das 8 Damen Problem beschäftigt sich mit der Frage, wie acht Damen auf ein Schachbrett gestellt werden können, ohne sich gegenseitig schlagen zu können. Als Hinführung zu diesem Problem wird den Studenten zuvor das Davis-Putnam-Verfahren vorgestellt. Nachdem das Davis-Putnam-Verfahren implementiert wurde, soll anhand dessen das 8 Damen Problem gelöst werden.

Zu Beginn der `davis_putnam`-Funktion wird die `saturate`-Methode auf die Menge `clauses` ausgeführt. Das Ergebnis wird in die Variable `s` gespeichert, während in SETLX dafür die Variable `clauses` überschrieben wird. Daraufhin wird geprüft, ob die ermittelte Menge eine leere Menge enthält, da in diesem Fall das Ergebnis ein Falsum wäre. Dadurch würde eine leere Menge in einer leeren Menge zurückgegeben werden. Wenn in `s` nur Unit-Klauseln enthalten sind, so ist das Ergebnis auch bereits ermittelt und es wird `s` zurückgegeben. Auch wenn beide if-Abfragen leicht unterschiedlich aussehen führen sie jedoch die gleiche Abfrage aus. Danach wird ein beliebiges Literal aus `s`, das sich nicht in `literals` befindet gewählt und in einer weiteren Variable negiert. Nun folgt ein rekursiver Aufruf, bei dem `s` mit 1 addiert und `literals` mit sowohl 1 als auch `not_1` addiert, als Parameter übergeben werden. Wenn das Ergebnis keine Menge einer leeren

```

1  davisPutnam := procedure(clauses, literals) {
2      clauses := saturate(clauses);
3      if ({} in clauses) {
4          return { {} };
5      }
6      if (forall (c in clauses | #c == 1)) {
7          return clauses;
8      }
9      l := selectLiteral(clauses, literals);
10     notL := negateLiteral(l);
11     r := davisPutnam(clauses + {{l}}, literals + {l, notL});
12     if (r != { {} }) {
13         return r;
14     }
15     return davisPutnam(clauses + {{notL}}, literals + {l, notL});
16 };
```

Abbildung 4.32: davisPutnam in Davis Putnam (SETLX)

```

1  def davis_putnam(clauses, literals):
2      s = saturate(clauses)
3      if Set() in s:
4          return Set(Set())
5      if all(len(c) == 1 for c in s):
6          return s
7      l = select_literal(s, literals)
8      not_l = negate_literal(l)
9      r = davis_putnam(s + Set(Set(l)), literals + Set(l, not_l))
10     if r != Set(Set()):
11         return r
12     return davis_putnam(s + Set(Set(not_l)), literals +
13                          Set(l, not_l))
```

Abbildung 4.33: davis_putnam in Davis Putnam (Python)

Menge ist, wird es zurückgegeben. Anderen Falls wird das Ergebnis des rekursiven Aufrufs mit `not_l` anstelle von `l` zurückgegeben.

Zwischen den beiden Implementierungen ist kein bedeutender Unterschied zu bemerken. Das Einzige, das bei der Übersetzung bedacht werden muss ist, dass Funktionen wie `len` oder `all` bzw. einzeilige for-Schleifen in beiden Sprachen unterschiedlich dargestellt werden.

In `saturate` werden alle Klauseln, die mit Unit-Schnitten aus `s` ableitbar sind zurückgegeben. Hierbei werden die Klauseln, auf denen ein Unit-Schnitt bereits durchgeführt wurde, aus `s` entfernt.

Hier ist auch wieder zu erkennen wie groß die Ähnlichkeit der beiden Implementierungen ist. Bis auf syntaktische Unterschiede, wie beispielsweise der Aufruf von `arb` in Zeile 5 beider Umsetzungen, gibt es keine Unterschiede. Die Methoden `_reduce`, `select_literal`, `is_positive` und `negate_literal` sind ebenfalls nahezu identisch. Die enthaltenen Unterschiede, wie die Match-Syntax, wurden bereits bei vorherigen Code-Schnipseln gezeigt und somit an dieser Stelle nicht wiederholt durchleuchtet. Die Methode `_reduce` hat einen Unterstrich als erstes Zeichen, das in es in Python bereits eine Funktion mit der Bezeichnung `reduce` gibt.

Da damit der Davis-Putnam-Algorithmus implementiert ist, kann dieser nun in Queens (8

```

1  saturate := procedure(s) {
2      units := { k : k in s | #k == 1 };
3      used := {};
4      while (units != {}) {
5          unit := arb(units);
6          used += { unit };
7          l := arb(unit);
8          s := reduce(s, l);
9          units := { k : k in s | #k == 1 } - used;
10     }
11     return s;
12 };

```

Abbildung 4.34: saturate in Davis Putnam (SETLX)

```

1  def saturate(s):
2      units = Set(k for k in s if len(k) == 1)
3      used = Set()
4      while len(units) != 0:
5          unit = units.arb()
6          used += Set(unit)
7          l = unit.arb()
8          s = _reduce(s, l)
9          units = Set(k for k in s if len(k) == 1) - used
10     return s

```

Abbildung 4.35: saturate in Davis Putnam (Python)

Damen Problem) verwendet werden.

```

1  atMostOne := procedure(s) {
2      return { { !p, !q } : p in s, q in s | p != q };
3  };

```

Abbildung 4.36: atMostOne in Davis Putnam (SETLX)

```

1  def at_most_one(s):
2      return Set(Set("!" + p, "!" + q) for p in s for q in s
3                  if p != q)

```

Abbildung 4.37: at_most_one in Davis Putnam (Python)

Bei der Betrachtung der Methode `at_most_one` fällt auf, dass in Python Strings verwendet werden, während in SETLX eine einfache Negierung stattfindet. Dies liegt daran, dass in SETLX für die Feldpositionen Variablen erzeugt werden, was jedoch in Python nicht möglich ist. Die Lösung mit den Strings erschwert zwar teilweise die Schreibweise, erfüllt jedoch denselben Zweck wie die Lösung in SETLX.

```

1  atMostOneInRow := procedure(row, n) {
2      return atMostOne({ Var(row, column) : column in [1 .. n] });
3  };

```

Abbildung 4.38: atMostOneInRow in Davis Putnam (SETLX)

```

1  def at_most_one_in_row(row, n):
2      return at_most_one(Set("varr%sc%s" % (str(row), str(column))
3                          for column in range(n)))

```

Abbildung 4.39: at_most_one_in_row in Davis Putnam (Python)

In der Funktion `at_most_one_in_row` kann gesehen werden, wie die „Variablen“ in Python dargestellt werden. Das Präfix „var“ steht für Variable, nach dem „r“ kommt die Reihenummer (row) und nach dem „c“ die Zeilennummer (column). An dieser Stelle wird wieder, wie bereits zuvor, die automatische Generierung einer Liste in SETLX über den `range`-Befehl in Python gelöst.

Die Funktion `one_in_column` ist sehr ähnlich zu `at_most_one_in_row` und gleich aufgebaut. Deshalb wird die Methode an dieser Stelle nicht dargestellt. Anstatt die Rückgabe von `at_most_one` zurückzugeben, wird hier eine Menge zurückgegeben.

```

1  atMostOneInLowerDiagonal := procedure(k, n) {
2      s := { Var(r, c) : c in [1..n], r in [1..n] | r - c == k };
3      return atMostOne(s);
4  };

```

Abbildung 4.40: atMostOneInLowerDiagonal in Davis Putnam (SETLX)

```

1  def at_most_one_in_lower_diagonal(k, n):
2      return at_most_one(
3          Set(
4              "varr%sc%s" % (str(row), str(column))
5              for row in range(n)
6              for column in range(n)
7              if row - column == k
8          )
9      )

```

Abbildung 4.41: at_most_one_in_lower_diagonal in Davis Putnam (Python)

In `at_most_one_in_lower_diagonal` lässt sich wieder erkennen, dass, bis auf die Eigenart des Strings als Variable, beide Implementierungen derselben Struktur folgen. Die Implementierungen sind sehr gut gegenüberstellbar.

In `at_most_one_in_upper_diagonal` ist die Umsetzung analog zu der in `at_most_one_in_lower_diagonal`.

Beide Ausführungen von `all_clauses` weisen wieder große Ähnlichkeiten auf. Allerdings wird zur Ermittlung der Summe einer Menge in SETLX der Operator `+`, während in Python die Methode `sum` verwendet.

```

1  allClauses := procedure(n) {
2      return +/ {atMostOneInRow(row, n)      : row in {1..n}}
3      + +/ {atMostOneInLowerDiagonal(k, n) : k in {-(n-2) .. n-2}}
4      + +/ {atMostOneInUpperDiagonal(k, n) : k in {3 .. 2*n - 1}}
5      + +/ {oneInColumn(column, n)          : column in {1 .. n}};
6  };

```

Abbildung 4.42: allClauses in Davis Putnam (SETLX)

```

1  def all_clauses(n):
2      return Set(at_most_one_in_row(row, n)
3                 for row in range(n)).sum() + \
4                 Set(at_most_one_in_lower_diagonal(k, n)
5                     for k in range(-(n-2), n-2)).sum() + \
6                 Set(at_most_one_in_upper_diagonal(k, n)
7                     for k in range(3, 2*n - 1)).sum() + \
8                 Set(one_in_column(column, n)
9                     for column in range(n)).sum()

```

Abbildung 4.43: all_clauses in Davis Putnam (Python)

```

1  solve := procedure(n) {
2      clauses := allClauses(n);
3      print(clauses);
4      solution := davisPutnam(clauses, {});
5      if (solution != { {} }) {
6          printBoard(solution, n);
7      } else {
8          print("The problem is not solvable for $n$ queens!");
9          print("Try to increase the number of queens.");
10     }
11 };

```

Abbildung 4.44: solve in Davis Putnam (SETLX)

Da nun alle benötigten Methoden definiert wurden, kann die Lösung ermittelt werden. Zuerst werden alle Klauseln mit dem `all_clauses`-Befehl erstellt. Diese werden daraufhin ausgegeben und dann das Problem anhand von `davis_putnam` gelöst. Zuletzt wird, sofern ein Ergebnis vorhanden ist, dieses ausgegeben, oder eine Fehlermeldung ausgegeben.

Die Zeit für *queens.py* wird, wie beim Schiebepuzzle, gemessen. Die nachfolgenden Messwerte wurden mit demselben Rechner, wie er in 4.3.1 beschrieben wurde, ermittelt. Während das SETLX-Programm in ca. 500-600 Millisekunden durchläuft, werden beim Python-Skript Zeiten von 34-36 Sekunden gemessen. Prüfungen mit einem Profiler ergaben, dass die meiste Zeit an Aufrufen von `davis_putnam` verloren geht. Es wird vermutet, dass die Sortierung der Mengen zudem eine Rolle bei der Effizienz spielen.

```
1  def solve(n):
2      clauses = all_clauses(n)
3      print(clauses)
4      solution = davis_putnam(clauses, Set())
5      if solution != Set(Set()):
6          print_board(solution, n)
7      else:
8          print("The problem is not solvable for %s queens!"
9                % str(n))
10         print("Try to increase the number of queens.")
```

Abbildung 4.45: solve in Davis Putnam (Python)

Kapitel 5

Fazit

Prinzipiell kann gesagt werden, dass in den meisten Fällen die SETLX-Programme, mit Hilfe des lecture-Moduls, in Python gleich programmiert werden können. Die Syntax beider Sprachen unterscheidet sich stellenweise und manchmal muss ein Ausdruck nach einem anderen Schema aufgebaut werden. Jedoch bieten beide Sprachen für viele Aufgaben geeignete Methoden die eine einfache Problemlösung ermöglichen.

Eine Umstellung der Programmiersprache in der Vorlesung wäre durchaus möglich. Die Performance-Differenzen, die in zwei der umgesetzten Implementierungen aufgetreten sind, sind zwar ärgerlich, jedoch befinden sich die Laufzeiten noch in akzeptablen Bereichen. Der Wechsel der Programmiersprache würde einer der Intentionen von SETLX, den Studenten eine Sprache näherbringen, die eine sehr ähnliche Syntax wie die Sprachen Java und C besitzt, nicht mehr verfolgen. Allerdings wäre Python eine stärker verbreitete Programmiersprache und die Studenten würden somit auch die möglichen Unterschiede zwischen Programmiersprachen erkennen. Das wäre auch eine gute Vorbereitung auf die Praxis, da dort häufig eine Vielzahl an verschiedenen Programmiersprachen Einsatz findet. Ein weiteres Problem wäre, dass die sehr mathematische Notation von SETLX nicht mehr vorhanden wäre. In Python werden häufig Wörter als Operatoren anstatt mathematischen Operatoren verwendet. Im Gegenzug dazu könnten sich Anfänger, die noch nie Kontakt mit einer Programmiersprache den Code leichter lesen, da anfangs weniger mathematische Operatoren gelernt werden müssen um den Code verstehen zu können.

Literaturverzeichnis

- [Guo14] Philip Guo. Python is now the most popular introductory teaching language at top u.s. universities, 07.07.2014.
- [SH15] Karl Stroetmann and Tom Herrmann. Setlx - a tutorial: Version 2.4.0. Tutorial for programming language, 30.07.2015.