7down – Dokumentation

Joachim Breitner, Manuel Holtgrewe, Lucas Lürich, Mathias Ziebarth
7down@lists.nomeata.de

12. Dezember 2007

Inhaltsverzeichnis

1 Einführung **4**

2 Die Lösungsidee **4**
2.1 Naives Backtracking 4
2.2 Backtracking mit Suchraum-Einschränkung 4
2.3 Idee – Zusammenfassung 6

3 Skizze der Umsetzung **7**
3.1 Umsetzung der Zahlen 7

3.2 Regeln . . . 7
3.3 Normalisieren von Regeln . . . 7
3.4 Behandlung von Operatoren . . . 7

4 Die Programmstruktur **8**
4.1 Modulstruktur . . . 8
4.2 Implementierung . . . 9

5 Der Lösungsalgorithmus im Detail **10**

6 Umsetzung der Regeln **11**
6.1 Prädikat-Regeln . . . 11
6.1.1 AB(X) – DescendingRule . . . 11
6.1.2 $PAL(X)$ – PalindromeRule . . . 12
6.1.3 $PZ(X)$ PrimeRule . . . 12
6.1.4 $Q(X)$ – SquareRule . . . 13
6.1.5 $K(X)$ – CubicRule . . . 14
6.1.6 $U(X)$ – UniqueDigitsRule . . . 14
6.1.7 $=(X, Y)$ – EqualityRule . . . 14
6.1.8 $V(X, Y)$ – MultipleOfRule . . . 15
6.2 Hilfs-Regeln . . . 16
6.2.1 NumberRule . . . 16
6.2.2 $NLZ(X)$ NoLeadingZeroRule . . . 17
6.2.3 $REMOVE(X, M)$ RemoveValuesFromDigitsRule . . . 18
6.3 Operatoren-Regeln . . . 18
6.3.1 $IstQuersumme(X, Y)$ – CrossSumRule . . . 18
6.3.2 $QuerProduktIst(X, Y)$ – CrossProductRule . . . 19
6.3.3 $IsMal(A, B, C)$ – MultOperatorRule . . . 19
6.3.4 $IsPlus(A, B, C)$ – AddOperatorRule . . . 19
6.3.5 $IsMinus(A, B, C)$ – SubtractOperatorRule . . . 19
6.3.6 $IsReverseOf(A, B)$ – FlipOperatorRule . . . 20

7 Graphische Benutzeroberfläche **20**
7.1 Übersicht über die Bedienelemente . . . 20
7.2 Laden eines Rätsels . . . 22
7.3 Den Lösungsvorgang beobachten . . . 22
7.4 In den Lösungsvorgang eingreifen . . . 22
7.5 Bearbeiten von Rätseln . . . 22
7.6 Implementierungs-Bemerkungen . . . 23

8 Softwaretechnisches Vorgehen **23**
8.1 Qualitätssicherung . . . 23
8.2 Entwicklungs-Werkzeuge . . . 24

9 Mathematische Abstrahierung **24**
9.1 Mathematische Modellierung . . . 24
9.2 Algorithmus . . . 25
9.3 Optimierungs-Strukturen . . . 25

10 Entwicklungsverlauf **26**

A Ein/Ausgabe des Programms **27**

B Installation und Aufruf **27**

1 Einführung

Dies ist die Dokumentation für unsere Lösung[1] der Aufgabe ,,Kreuzzahlenrätsel" für den Informaticup 2007.

Wir stellen zuerst unsere Lösungsidee und dann die Umsetzung der Idee vor. Dies machen wir zunächst unabhängig von der Implementierung. Danach erläutern wir die Implementierung der Lösung. Hierfür beschreiben wir Modul- und Klassenstruktur des Programms, um anhand daran auf die Umsetzung der Regeln einzugehen. Zu guter Letzt beschreiben wir noch eine mathematische Modellierung unserer Problemlösung.

Im Anhang befindet sich die Spezifikation der Ein- und Ausgabe des Programms sowie eine Installations- und Startanleitung.

2 Die Lösungsidee

Im Folgenden bezeichnen wir die ,,Prädikate" als ,,Regeln". Das hat den Grund, dass wir – wie in Abschnitt 3.4 beschrieben – auch die Operatoren als eine Art ,,Prädikat" umsetzen. Um die Begriffe besser abzugrenzen wählen wir den Namen ,,Regel".

2.1 Naives Backtracking

Da Kreuzzahlrätsel im Allgemeinen mindestens $\mathcal{NP}$-schwer sind, ist kein schneller und einfacher Lösungsalgorithmus zu erwarten (siehe [2]). Stattdessen kann man sich auf eine einfache erschöpfende Suche mittels *backtracking* beschränken: Dabei probiert man systematisch alle Kombinationen für Belegungen von Feldern und überprüft jeweils, ob eine der Regeln verletzt ist. Ist dies der Fall, so probiert man die nächste Belegung.

Hat das Feld eine Höhe von h, eine Breite von w und gibt es n Regeln, dann ist offensichtlich $O(10^{h \cdot w} \cdot T_r(n))$ die Worst-Case-Laufzeit des naiven Algorithmus. Dabei bezeichnet $T_r(n)$ die Gesamtlaufzeit aller Regeln. Unabhängig davon, wie lange die Regeln benötigen ist eine Laufzeit von $O(10^{100})$ bei Feldern, die bis 10 mal 10 groß werden können nicht akzeptabel (10^{80} ist in etwa die Anzahl der Atome im Universum, siehe [3]).

2.2 Backtracking mit Suchraum-Einschränkung

Bei genauerer Betrachtung merkt man schnell, dass man den Suchraum stark einschränken kann, wenn man die Regeln nicht nur zur Überprüfung der Zahlen benutzt. Stattdessen kann die Information der Regeln den Suchraum sehr weit einschränken. Wir geben hierzu ein Beispiel (Abbildung 1).

Einschränkungen auf Ziffer-Ebene. Betrachtet man etwa das Beispielfeld in Abbildung 1 so sieht man etwa, das viele Einschränkungen möglich sind. Diese Einschränkungen entsprechen auch, wie ein Mensch intuitiv vorgehen würde: Aus der Vielzahl der Möglichkeiten werden erst einmal die einfach auszuschließenden herausgetrichen, bevor Annahmen getroffen werden.

Die Zahlen in den Feldern geben jeweils an, welche Ziffern dort noch möglich sind. Die Felder sind dabei mit (x, y) angegeben, x und y beginnen bei 1.

- $(1, 1)$, $(3, 1)$ und $(1, 2)$ können nicht gleich 0 sein, da sie erste Stellen von Zahlen sind – dies ist in der Aufgabenstellung verboten.

[1] Der Name des Programms ist *7down*, da Bezeichner in Python nicht mit Zahlen beginnen dürfen benutzen wir dort den Namen *sevendown*.

A 1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0	C 1 2 3 4 5 6 7 8 9 0
B 1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0

Abbildung 1: Ein Beispielfeld. Die Regeln sind R1 $PZ(C_s)$, R2 $=(qs(A_w), 3)$, R3 $=(qs(B_s), 9)$ und R4 $V(B_w, A_w)$

- $(3, 2)$ kann keine geraden Zahl und die 5 nicht sein, da C_s dann keine Primzahl wäre (sondern Vielfaches von 2 oder 5).
- Die Felder von A_w können nicht größer sein als 3. Außerdem müssen $(1, 1)$ und $(1, 3)$ mindestens 1 sein. Also können die Ziffern von A_w maximal 2 sein.
- Ähnliches gilt auch für B_w. $(1, 2)$ muss mindestens gleich 1 sein, was für $(2, 2)$ und $(3, 2)$ die Zahl 9 ausschließt.

Wendet man diese Einschränkungen an, erhält man das Feld aus Abbildung 2(a).

A 1 2	1 2 0	C 1 2
B 1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 0	1 3 7

(a) Das Beispielfeld nach den ersten Einschränkungen.

A 1 2	1 2 0	C 1 2
B 1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 0	1 3 7

(b) Das Beispielfeld nach weiteren Einschränkungen.

Abbildung 2: Die ersten beiden Einschränkungen.

Man sieht nun außerdem, dass durch die Einschränkung von $(3, 2)$ die zweite Ziffer von B_w nicht mehr 8 sein kann, da die beiden anderen Ziffern mindestens 1 sind. Man erhält also das Feld aus Abbildung 2(b).

Einschränkungen auf Zahlen-Ebene. Bis jetzt wurde nur auf Ebene der Ziffern eingeschränkt. Man kann jedoch auch Einschränkungen auf der Ebene der Zahlen vornehmen. Im Beispielfeld aus Abbildung 2(b) gibt es nur noch relativ wenige möglichen Werte von A_w und C_s:

$$C_s \in \{11, 13, 17, 21, 23, 27\} \text{ und } A_w \in \{101, 102, 111, 112, 121, 122, 201, 202, 211, 212, 221, 222\}$$

Da 21 und 27 keine Primzahlen sind, können wir sie für C_s ausschließen. Die Primzahlregel lässt hier jedoch leider keine weiteren Schlüsse zu, da sowohl 1 als auch 2 in der ersten Ziffer der möglichen Primzahlen vorkommen. Das gleiche gilt auch für die 1 und 7 in der zweiten Ziffer.

Mit genug Papier und Rechenaufwand könnten nun alle $9 \cdot 8 \cdot 3 = 216$ mögliche Zahlen von B_w aufgeschrieben werden. Dann könnte man überprüfen, ob ein möglicher Wert von A_w sie teilt und die Werte ggf. ausschließen.

Dabei kommt heraus, das $(3, 1)$ und $(2, 1)$ nicht 2 sein können. Man erhält das Feld aus Abbildung 3(a). Für A_w bleiben dann noch 201 und 101 übrig, für C_w noch 11, 17 und 13. B_w wird auch weiter eingeschränkt. Andersherum könnte man auch für jeden möglichen Wert von A_w prüfen, ob er ein Teiler eines möglichen Wertes von B_w ist.

Der Suchraum wurde bis jetzt von 10^6 auf $3'888$ eingeschränkt. Nun wird der Wert einer Ziffer verfügt. Dabei hat man generell die Freiheit und man kann sich auch verschiedene Strategien überlegen.

Verfügt man etwa $(1, 1)$ zu 1 so erhält man für A_w den Wert 111. Dadurch kann B_w nur noch 111, 333 oder 777 sein. Nur 333 hat die Quersumme 9 und das Rätsel hat die Lösung wie in Abbildung 3(b).

A 1 2	1 0	C 1
B 1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 0	1 3 7

(a) Noch eine Einschränkung.

A 1	1	C 1
B 3	3	3

(b) Die Lösung

Abbildung 3: Eine weitere Einschränkung und die Lösung.

Die Zahl der Backtracking-Schritte konnte in diesem Beispiel also auf 1 gedrückt werden. Eine naive erschöpfende Suche, die Felder $0 \ldots 9$ verfügt, hätte zwar nicht den schlechtesten Fall von $1'000'000$ erreicht, aber immernoch etwa 300 Schritte.

Dies ist aber nur hier der Fall. Hätte $(1, 1)$ den Wert 9 annehmen müssen, hätte man diese Suche schon fast in den schlechtesten Fall getrieben.

2.3 Idee – Zusammenfassung

Zusammengefasst ist unsere Idee also:

1. Fasse Ziffern und Zahlen als Variablen mit möglichen Wertemengen auf.
2. Schränke diese Mengen auf Ziffernebene durch Regeln ein.
3. Sobald die Anzahl der möglichen Werte einer Zahl „klein genug" wird, berechne alle möglichen Werte der Zahl und wende die Regeln auch auf die Zahl an.
4. Versuche durch die verkleinerten Wertemengen für Zahlen auch die der Ziffern einzuschränken.
5. Wiederhole Schritte 2, 3 und 4 bis sich keine Veränderungen mehr ergeben.
6. Wähle eine Ziffer. Für alle möglichen Werte x der Ziffer: Verfüge den Wert zu x und gehe zu 1.

3 Skizze der Umsetzung

Wir wollen hier kurz auf die grundlegenden Punkte der Umsetzung aus Abschnitt 2 eingehen. Diese werden unabhängig von der genauen Implementierung beschrieben.

3.1 Umsetzung der Zahlen

Es stellt sich zunächst die Frage, wie die in Abschnitt 2.2 beschriebenen Zahlen und Ziffern umgesetzt werden sollen. In unserer Umsetzung gibt es dafür *Zellen*, *Zahlen* und *Ziffern*. In der Implementierung ist Zelle die Oberklasse für Zahlen und Ziffern. Diese Einschränkung hat jedoch Implementierungsgründe.

Eigentlich müssen Zahlen und Ziffern nur die Mengen von momentan noch möglichen Werten bereit stellen. Zahlen haben außerdem noch eine Sequenz von Ziffern. Außerdem müssen diese Mengen einschränkbar sein, etwa über Mengenschnitt und Mengendifferenz.

Konstanten, wie etwa die 42 in der Regel $=(A_w, 42)$ können auch einfach als Zahlen dargestellt werden. Von ihren Ziffern sind offensichtlich die – schon einelementigen – möglichen Werte bekannt.

3.2 Regeln

Regeln können einfach als ,,Einschränkungen auf Zellen" betrachtet werden. Findet die Regel einen Widerspruch, so kann sie in einer Zelle die Menge der möglichen Werte leeren.

Wenn der Programmteil, der die Regeln anwendet, dann eine leere Menge in Zellen findet, kann er den aktuellen Schritt wegen einer Regelverletzung abbrechen. Diese Normalisierung vereinfacht viele Abbruchbedingungen: Es gibt keine explizite Rückgaben oder Ausnahmen für ,,nicht lösbar". Stattdessen wird dies implizit durch leere mögliche Mengen signalisiert.

3.3 Normalisieren von Regeln

Um den Lösungs-Algorithmus bzw. seine Subroutinen für die Regeln möglichst einfach zu halten, kann es nützlich sein, komplexe Regeln durch einfachere zu ersetzen.

Wir nennen diesen Schritt *normalisieren von Regeln.*

Ein Beispiel ist etwa die Palindrom-Regel. Betrachtete man hier die Regel nur auf Zahlen, so müsste man warten, bis die Menge der möglichen Zahlen klein genug ist, um erzeugt zu werden. Da diese durch die möglichen Werte der Ziffern induziert wird, kann dies lange dauern. Außerdem möchte man bei einer 10 Ziffern langen Zahl die 0 in der letzten Ziffer ausschließen, ohne entweder 10^{10} int-Werte zu erzeugen oder zu warten, bis die Anzahl der möglichen Werte entsprechend gesunken ist.

Die Palindrom-Regel lässt sich einfach durch Gleichheits-Regeln ersetzen: Eine für die erste und die letzte Ziffer, eine für die zweite und zweitletzte usw. Wird die Menge der möglichen Werte einer Ziffer eingeschränkt, so kann gleich die Menge für die zugehörige auch eingeschränkt werden (wenn die Zahl nicht ein Palindrom ungerader Länge und die Ziffer die mittlere ist).

Die Palindrom-Regel muss nach dem Normalisieren nicht weiter betrachtet werden.

3.4 Behandlung von Operatoren

Bis hier wurden nur Regeln ohne Operatoren betrachtet. Diese haben eine einfache Form und die Zahl der Fälle für die Einschränkungen der möglichen Mengen ist überschaubar. Da Operatoren beliebig tief geschachtelt werden können, ist dies hier nicht mehr der Fall.

Da es unendlich viele Formen von Regeln gibt, kann nicht für jede ein optimierter Einschränkungsfall angegeben werden. Man betrachte etwa alle Regeln der Form

$$=(A_w, qp(qs(\dots qs(qp(B_w)))\dots)) \quad \text{oder}$$
$$=(A_w, qp(plus(A_w, \dots qp(plus(A_w, 1))\dots))).$$

Um dies zu umgehen, kann man Regeln mit geschachtelten Operatoren *ausflachen*. Für jeden Operator führt man eine Regel ein, die ihr erstes Argument über den Operator angewandt auf die restlichen Argumente definiert. Dabei wird aus einem k-stelligem Operator eine $k+1$-stellige Regel.

So gibt es für den Quersummen-Operator $qs(a)$ die Regel $IstQS(x, a)$, die aussagt, dass x die Quersumme von a ist.

Das Ausflachen von Formeln benötigt jedoch einen Datentyp, der diese Zwischenwerte x aufnehmen kann. Wir führen daher *Zwischen-Zahlen* ein. Diese haben das gleiche Verhalten wie Zahlen, nur dass sie unter Umständen keine Ziffern haben.

Der Fall, dass eine Zahl auf dem Feld mit einer Gleichheitsregel und einer Funktion definiert wird besonders behandelt. Die Regel $= (A_w, mal(B_w, C_w))$ aus der Eingabe würde umgesetzt auf die Regel $IstMal(A_w, B_w, C_w)$ statt auf die beiden Regeln $= (A_w, X_0)$ und $IstMal(X_0, B_w, C_w)$.

Diese Umformungen können auf der syntaktischen Ebene direkt nach dem Parsen der Eingabe geschehen.

4 Die Programmstruktur

Wir gehen zunächst kurz auf die Modulstruktur ein und erklären dann ausführlich die Klassen. Die Funktionsweise der Regeln wird in Abschnitt 6 beschrieben.

4.1 Modulstruktur

Alle Module des Programms sind Untermodule des Moduls *sevendown*.

- *config* – enthält Variablen, über die einige parametrisierte Teile des Programms kontrolliert werden können.
- *graph* – definiert die grundlegenden Klassen für das Spielfeld, Zahlen, Ziffern sowie die Superklasse für Regeln.
- *loader* – enthält den Parser für die Texteingabe.
- *logger* – enthält Klassen zum Kapseln der detaillierten Programmausgabe.
- *pretty* – Subroutinen zum Ausgabe des Spielfeldes in ASCII-Art.
- *rules* – Implementierung der Regeln.
- *solver* – Implementierung des Lösungsalgorithmus.
- *statistics* – Modul zum Sammeln von einfachen Statistiken während des Lösungsalgorithmus.

- *undo* – Dieses Modul erlaubt es, Veränderungen in Objekten zu speichern und wieder rückgängig zu machen. Damit kann nach einem Rücksprung aus einem Backtracking-Schritt eine Veränderung wieder rückgängig gemacht werden.
- *util* – Hilfsmethoden.
- *binlist* – Implementierung einer Liste für vorberechnete Zahlen, mit effizienter Ist-Element-Von-Operation.

4.2 Implementierung

In diesem Abschnitt gehen wir auf die Implementierung der Lösungsidee ein. Dabei beschreiben wir zunächst die Klassenstruktur. Danach diskutieren wir noch weitere benutzte Datenstrukturen.

Zahlen, Zellen, ... Wie schon in 3.1 erwähnt gibt es für Zellen, Zahlen, Ziffern, und Konstanten eigene Datentypen (Klassen in Python). Außerdem existiert noch eine Bequemlichkeits-Klasse für Zahlen auf einem Feld. Im Python Code heißen die Klassen naheliegend *Cell*, *Number*, *Digit*, *Fixed* und *FieldNumber*. Abbildung 4 zeigt die Beziehungen dieser Klassen-Hierarchie in UML.

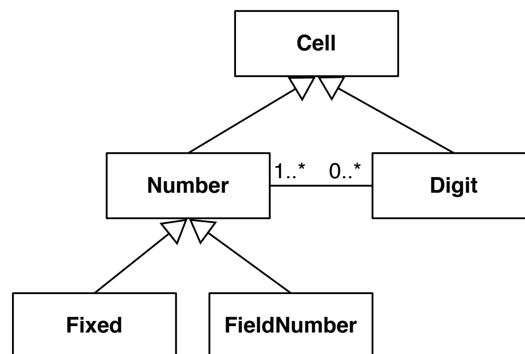

Abbildung 4: Die Beziehnungen in der Klassenhierarchie um *Cell*.

Feld. Das Spielfeld ist in der Klasse *Field* gekapselt. Es enthält ein zweidimensionales Feld von *Digit* Objekte sowie ein Feld von *Number* und eines von *Rule* Objekten. Abbildung 5 zeigt diese Beziehungen.

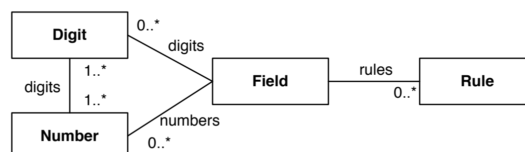

Abbildung 5: Die Beziehnungen um die Klasse *Field*.

Regel. Die Klasse *Rule* kapselt eine Regel. Ihre Unterklassen *UnaryRule* und *BinaryRule* arbeiten jeweils auf einem bzw. zwei Argumenten.

Dass gewisse Regeln nur auf *Digit* bzw. *Number* Objekten arbeiten, ist nicht durch Vererbung ausgedrückt. Stattdessen befinden sich in den entsprechenden Klassen Zusicherungen (Python hat mit dem Schlüsselwort *assert* dafür ein Sprachfeature), die dies sicherstellen. Da Python keine statisch getypte Sprache ist, hätte das Einführen von Klassen zu diesem Zweck keinen praktischen Mehrwert.

Die Klassenhierarchie um *Rule* ist in ausschnittsweise in Abbildung 6 dargestellt.

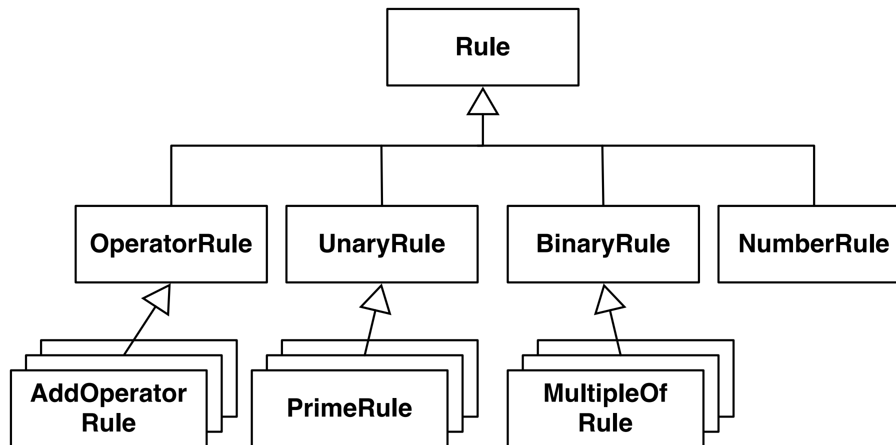

Abbildung 6: Die Vererbungen der Klassenhierarchie um *Rule* (Ausschnitt).

Weitere benutzte Datenstrukturen. Neben den eigenen Typen (bzw. Klassen) benutzen wir intern noch eingebaute Typen von Python: Felder von *Digit* Objekten werden in Python auf *list* Objekte abgebildet. An vielen Stellen werden Regeln in Python *set* Objekten abgebildet. Die Mengen von möglichen Werten werden auch auch auf Python *set* Objekte abgebildet.

Pythons *list* Objekte ähnlich wie C++ *vector* implementiert: Dynamisch wachsende Felder mit einer Hysterese für die Entscheidung. Zugriff über Indizes, Anhängen von Werten sind damit in $O(1)$ möglich, Löschen von Werten innerhalb von *list* Objekten in $O(n)$. Damit unterscheiden sich die Teile unseres Python Codes nur um konstante Faktoren von einer hochgradig optimierten Implementierung in einer maschinennäheren Sprache.

Pythons *set* Objekte sind wie C++ *tr1::unordered_set* Objekte über Hash-Tabellen implementiert. Betrachtet man nur Erwartungswerte, so gilt offensichtlich: Das Überprüfen auf Enthaltensein und das Entfernen von Elementen ist in $O(1)$ möglich. Das Bilden des Mengenschnitts in $O(n+m)$, wobei n und m die Kardinalitäten der zu schneidenden Mengen sind.

Hier könnte eine maschinennähere Implementierung mehr Kontrolle und durch genauere Analyse mehr Spielraum für Optimierung bringen. Zum Beispiel könnte es effizienter sein, Mengen als geordnete Sequenzen darzustellen. Da Programme in Python jedoch kompakter und schneller zu schreiben sind haben wir uns für diese dynamische Sprache und nicht etwa für C++ entschieden.

5 Der Lösungsalgorithmus im Detail

TODO

6 Umsetzung der Regeln

Zunächst beschreiben wir in Abschnitt 6.1, wie die Regeln umgesetzt wurden, die als Prädikate ins Programm eingegeben wurden. Danach beschreiben wir in Abschnitt 6.2 einige einfache Hilfs-Regeln, die zur Umsetzung der anderen Regeln benutzt werden, jedoch vom Programm generiert werden und nicht eingegeben wurde. Zuletzt beschreiben wir in Abschnitt 6.3 die Regeln, die die Operatoren umsetzen.

Für jede der Regeln beschreiben wir zunächst die abstrakte Funktion. Diese Beschreibung entspricht unserer Interpretation des Prädikats/Operator aus der Aufgabenstellung bzw. der Definition unserer eigenen Regeln.

Dann beschreiben wir, wie die Regel ggf. andere Regeln erzeugt, die ihr beim Durchsetzen ihrer Funktion hilft. So wird zum Beispiel die Regel fürs Prädikat „Palindrom" auf Gleichheits-Regeln ihrer Ziffern abgebildet.

Schließlich beschreiben wir noch, wie die Regel vorgeht, wenn sie angewandt wird.

Wenn dabei von den Ziffern einer Zahl die Rede ist, so gilt dieses nur für Zahlen, die auch Ziffern haben, was für Zwischenwerte nicht immer gegeben ist. Weiterhin ist mit „Menge der möglichen Zahlen werden gesetzt" gemeint, dass sie nur gesetzt werden, wenn die Menge nicht größer als die konfigurierte obere Schranke für explizite Mengen ist.

Wenn die Menge der möglichen Werte für eine Zahl nicht gesetzt ist – über sie ist also noch nichts bekannt – dann wird dies als Menge aller möglicher Zahlen interpretiert. Die Kardinalität ist „∞" (genauer $\aleph_0$).

Ist im Folgenden von einer Zahl X die Rede, dann seien ihre Ziffern mit $X_1, X_2, \ldots, X_k$ bezeichnet. Zu einer Ziffer oder Zahl z sind die $m(z)$ möglichen Werte.

6.1 Prädikat-Regeln

6.1.1 AB(X) – DescendingRule

Beschreibung. $AB(X)$ bedeutet, dass die Ziffern der Zahl X nicht-aufsteigend sortiert sein müssen. So sind etwa die Ziffern der Zahl 321 und 111 absteigend sortiert und die der Zahl 123 nicht.

Bei negativen Zahlen wird das Minuszeichen ignoriert. Eine negative Zahl z hat also genau dann nicht-aufsteigend sortierte Ziffern wenn ihr Betrag $|z|$ nicht-aufsteigend sortierte Ziffern hat. Wir beschreiben also o. B. d. A. nur DescendingRule auf positiven Zahlen.

Erzeugte Regeln. DescendingRule erzeugt keine neuen Regeln.

Funktionsweise auf X_i. Wird die Regel angewandt, so stellt sie zunächst sicher, ob die Bedingung „absteigend" auf den Ziffern von X durchgesetzt ist. Dies wird wie folgt sicher gestellt:

Sei h die höchste Zahl, die die nächste Ziffer noch annehmen kann. Bei der ersten Ziffer ist diese noch 9. Bei der zweiten Ziffer ist diese dann 9 wenn $\max m(X_1)$ gleich 9 ist, also der größte noch mögliche Wert der ersten Ziffer auch 9 ist. Dann werden aus der Menge der möglichen Ziffern der zweiten Ziffer, also $m(X_2)$ die Werte entfernt, die größer sind als h.

Insgesamt wird also von der ersten zur letzten Ziffer die Menge der für die Ziffer möglichen Werte eingeschränkt. Die Einschränkung erfolgt auf die Werte, die kleiner oder gleich des größten Wertes der vorherigen Ziffer sind.

In Pseudocode:

```
1  h ← 9
2  for i ← 1 to k
3          do m(X_i) ← {z | z ∈ m(X_i) ∧ z ≤ h}
4              h ← max m(X_i)
```

Funktionsweise auf X. Nachdem die möglichen Werte der Ziffern eingeschränkt wurden, kann die Regel außerdem noch die möglichen Werte von X einschränken. Für jeden möglichen Wert wird geprüft, ob seine Ziffern absteigend sortiert sind.

In Pseudocode:

```
1  m(X) ← {z | z ∈ m(X) ∧ Ziffern von z sind absteigend sortiert.}
```

Beachte, dass die Menge der möglichen Werte auf die leere Menge eingeschränkt werden kann. Dies ist dann ein Wiederspruch für den Lösungsalgorithmus.

6.1.2 $PAL(X)$ – PalindromeRule

Beschreibung. $PAL(X)$ bedeutet, dass X ein Palindrom ist. Es gilt also, dass die letzte Ziffer gleich der ersten ist, die vorletzte gleich der zweiten usw. Formal: $\forall 1 \leq i \leq k : X_i = X_{k-i+1}$.

Es reicht dies für i bis $\lfloor \frac{k}{2} \rfloor$ zu überprüfen um doppeltes Prüfen zu vermeiden und man bei ungerader Länge die mittlere Ziffer nicht mit sich selbst vergleichen muss.

Erzeugte Regeln. Wenn X explizite Ziffern hat, dann wird die Funktionsweise von PalindromRule durch mehrere EqualityRules durchgesetzt. Für jedes Paar von Ziffern, die gleich sein müssen, wird eine neue EqualityRule eingeführt.

In Pseudo-Code:

```
1  if X hat explizite Ziffern
2      then for i ← 1 to ⌊k/2⌋
3                  do Erzeuge neue Regel  =(X_i, X_{k−i+1})
```

Funktionsweise. Wenn X keine expliziten Ziffern hat, sondern nur Werte, dann muss die Funktionsweise (also das Testen auf ,,ist Palindrom") auf den möglichen Werten für die Zahl durchgesetzt werden: Für jeden möglichen Wert wird getest, ob er ein Palindrom ist. Ist dies nicht der Fall, so wird er entfernt.

In Pseudocode:

```
1  if X hat keine expliziten Ziffern
2      then m(X) ← {z | z ∈ m(X) ∧ z ist Palindrom}
```

6.1.3 $PZ(X)$ PrimeRule

Beschreibung. $PZ(X)$ bedeutet, dass X eine Primzahl ist.

Erzeugte Regeln. Die letzte Ziffer einer Primzahl kann kein Vielfaches von 2 sein, sonst wäre sie selbst ein Vielfaches von 2. Weiterhin kann die letzte Ziffer nicht 5 sein, sonst wäre die Zahl selbst ein Vielfaches von 5.

PrimeRule erzeugt eine RemoveValuesFromDigitsRule auf die letzte Ziffer, die dann die Werte aus der Menge der möglichen Werte für die letzte Ziffer entfernt.

In Pseudocode:

1 Erzeuge neue Regel REMOVE($X_k, \{0, 2, 4, 5, 6, 8\}$)

Funktionsweise auf X_i. PrimeRule selbst arbeitet nicht auf Ziffern. Dies überlässt sie der erzeugten RemoveValuesFromDigitsRule.

Funktionsweise auf X. Eine Liste von Primzahlen zu erzeugen ist zeitaufwändig, kann mit einem Sieb des Eratosthenes jedoch einfach implementiert werden. Um die Zahlen noch etwas schneller zu berechnen benutzen wir das Sieb des Atkin. Das Prüfen einer Zahl auf ,,ist Primzahl" ist mit dem Algorithmus von Rabin und Miller einfach möglich.

Wir haben alle Primzahlen mit 7 Stellen und weniger vorberechnet. Derer gibt es 664'579, als 4-Byte-Zahlen entspricht das etwa 2,6 MB, was vertretbar ist. Das heißt, dass unserer Programm schnell Sequenzen von Primzahlen zwischen zwei Zahlen bis 7 Stellen erzeugen kann indem es diese aus den vorberechneten abliest.

Die Regel arbeitet nun wie folgt:

Wenn die möglichen Werte von X noch nicht gesetzt sind und die Zahl vier oder weniger Ziffern hat, dann werden die vorberechneten Primzahlen mit der richtigen Stellenzahl als Menge der möglichen Werte für die Zahl gesetzt.

Wenn die möglichen Werte von X schon gesetzt sind dann werden alle Zahlen entfernt, die keine Primzahlen sind. Das Prüfen geschieht mit einem Rabin-Miller-Test

In Pseudocode:

1 **if** $m(X) = \text{NIL} \wedge$ Menge der möglichen Werte ist klein genug
2 **then if** $k \leq 4$
3 **then** $m(X) \leftarrow \{z \mid z \text{ ist prim} \wedge |z| = k\}$
4 **else** $m(X) \leftarrow \{z \mid z \in m(X) \wedge z \text{ ist prim}\}$

6.1.4 $Q(X)$ – SquareRule

Beschreibung. $Q(X)$ bedeutet, dass X eine Quadratzahl ist.

Erzeugte Regeln. SquareRule erzeugt keine neuen Regeln.

Funktionsweise auf X_i. SquareRule arbeitet nicht auf den Ziffern der Zahl.

Funktionsweise auf X. Wenn die möglichen Werte von X noch nicht gesetzt sind und nicht zu viele Ziffern hat dann kann die Regel alle Quadratzahlen der Länge k berechnen. Solche Quadratzahlen gibt es maximal $\lceil 10^{\frac{k}{2}} \rceil$, also relativ wenige.

Sind die möglichen Werte von X schon gesetzt, überprüft die Regel einfach für jeden möglichen Wert, ob er eine Quadratzahl ist. Die nicht-Quadratzahlen werden aus $m(X)$ entfernt.

In Pseudocode (LIMIT ist Parameter zur Leistungsoptimierung):

1 **if** $m(X) = \text{NIL} \wedge$ Menge der möglichen Werte ist klein genug
2 **then if** $k \leq \text{LIMIT}$
3 **then** $m(X) \leftarrow \{z \mid z \text{ ist Quadratzahl} \wedge |z| = k\}$
4 **else** $m(X) \leftarrow \{z \mid z \in m(X) \wedge z \text{ ist Quadratzahl}\}$

6.1.5 $K(X)$ – CubicRule

$K(X)$ bedeutet, dass X eine Kubikzahl ist.

CubicRule arbeitet analog zu SquareRule. (In der Implementierung erben beide von PotenceRule, und beide Klassen setzen nur entsprechende Parameter für den Exponenten 2 bzw. 3).

6.1.6 $U(X)$ – UniqueDigitsRule

Beschreibung. $U(X)$ bedeutet, dass die Ziffern von X eindeutig sind. Formal: $\forall 1 \leq i, j \leq k : X_i = X_j \Rightarrow i = j$.

Erzeugte Regeln. UniqueDigitsRule erzeugt keine eigenen, neuen Regeln.

Funktionsweise. Im ersten Schritt wird überprüft, ob es noch genug mögliche Ziffern gibt, um jeder Ziffer der Zahl eine unterschiedliche zuzuweisen.

In Pseudocode:

1 **if** $|X_1 \cup X_2 \times \cdots \cup X_n| < |X|$
2 **then** $X_1 \leftarrow \emptyset$

Danach werden alle Ziffern betrachtet, die sich seit dem letzten Regelaufruf geändert haben und inzwischen nur noch einen möglichen Wert haben, und dieser Wert bei allen andern Ziffern entfernt. Dies wird wiederholt, bis keine neuen Schlüsse mehr möglich sind:

1 **while** $\exists X_i : |X_i| = 1 \wedge X_i$ wurde noch nicht behandelt.
2 **do for** $j \in \{1, \ldots, n\} \setminus \{i\}$
3 **do** $m(X_j) \leftarrow m(X_j) \setminus m(X_i)$

6.1.7 $=(X, Y)$ – EqualityRule

Beschreibung $=(X, Y)$ bedeutet, dass die Ziffern oder Zahlen X und Y gleich sind. Es müssen entweder X und Y Zahlen oder X und Y Ziffern sein.

Erzeugte Regeln. Wird eine EqualityRule auf zwei Ziffern angewandt, so werden keine weiteren Regeln erzeugt.

Wird sie hingegen auf zwei Zahlen angewandt und beide Zahlen haben Ziffern, dann werden neue Regeln erzeugt: Für die Ziffernpaare an der gleichen Position in X bzw. Y wird dann eine neue EqualityRule auf Ziffern eingeführt.

In Pseudocode:

1 $\triangleright$ Vorbedingung: $|X| = |Y|$
2 **for** $i \leftarrow 1$ **to** k
3 **do** Create new rule $=(X_i, Y_i)$

Wird EqualityRule auf zwei Zahlen angewandt, dann müssen diese gleich viele Ziffern haben, sofern überhaupt beide Ziffern haben. Sonst wäre schon in der Rätseldefinition ein Widerspruch.

Funktionsweise auf Ziffern. Auf zwei Ziffern X und Y wird die Regel angewandt, indem sie die Menge der möglichen Werte von beiden Ziffern schneidet und den Schnitt dann in die Mengen der möglichen Werte setzt.

1 $S \leftarrow m(X) \cap m(Y)$
2 $m(X) \leftarrow S$
3 $m(Y) \leftarrow S$

Funktionsweise auf Zahlen. Wenn X und Y beide Ziffern haben, dann macht EqualityRule beim Anwenden nichts. Alles Eingrenzen von möglichen Werten kann dann auch über die erzeugten EqualityRules auf den Ziffern erfolgen.

Wenn eine oder beide der Zahlen X und Y keine Ziffern hat (weil sie Zwischenzahl sind) dann muss die Gleichheit auf den möglichen Werten sichergestellt werden. Bei jeder Anwendung der Regel wird daher die Menge der möglichen Werte für X und Y auf den Schnitt beider Mengen möglicher Werte gesetzt.

In Pseudocode:

1 **if** X hat keine Ziffern $\vee$ Y hat keine Ziffern
2 **then** $S \leftarrow m(X) \cap m(Y)$
3 $m(Y) \leftarrow S$
4 $m(X) \leftarrow S$

6.1.8 $V(X, Y)$ – MultipleOfRule

Beschreibung. $V(X, Y)$ bedeutet, dass X Vielfaches von Y ist.

Erzeugte Regeln. MultipleOfRule erzeugt selbst keine weiteren Regeln.

Funktionsweise. Ist der Wert u von X und der Wert v von Y jeweils eindeutig, so kann einfach geprüft, ob u ein Vielfaches von v ist: Genau dann, wenn gilt: $u \equiv 0 \mod v$. Ist dies nicht der Fall, werden die Menge der möglichen Werte auf die leere Menge gesetzt, um anzuzeigen, dass ein Widespruch vorliegt.

Ist der Wert v von Y bekannt, der von X jedoch nicht, so geht die Regel wie folgt vor: Wenn v gleich 0 ist dann wird die Menge möglicher Werte von X mit 0 geschnitten, da nur 0 ein Vielfaches von 0 ist.

Ansonsten kann die mögliche Menge von X eingeschränkt werden, wenn X explizite Ziffern hat: Wir berechnen den kleinsten und den größten durch die Ziffern induzierten Wert a bzw. b. Diese teilen wir nun durch v und erhalten damit $a' = \lfloor a/v \rfloor$ und $b' = \lfloor b/v \rfloor$. Dann ist die Menge der möglichen Werte für X die Menge $\{i \cdot v \mid a' \leq i \leq b + 1\}$. Diese wird mit $m(X)$ geschnitten um das neue $m(X)$ zu erhalten.

Hat sowohl X als auch Y explizite Ziffern, so kann man noch auf den letzten Ziffern Einschränkungen vornehmen: X_k wird mit allen möglichen Vielfachen von Y_k modulo 10 gesetzt. Umgekehrt, wird Y_k auf alle möglichen Faktoren von X_k modulo 10 gesetzt.

In Pseudocode:

```
 1  if m(X) = m(Y) = 1
 2     then if Wert(X) ≢ 0  mod Wert(Y)
 3             m(X) ← m(Y) ← ∅
 4             return
 5     else  if |m(X)| > 1 ∧ |m(Y)| = 1
 6     then if Wert(Y) = 0
 7                then m(X) ← m(X) ∩ {0}
 8                else  if X hat explizite Ziffern
 9                         then u ← Wert(Y)
10                              a ← kleinster von Ziffern induzierter Wert von X
11                              b ← kleinster von Ziffern induzierter Wert von Y
12                              a′ ← ⌊a/u⌋
13                              b′ ← ⌊b/u⌋
14                              S ← {i · v | a′ ≤ i ≤ b + 1}
15                              m(X) ← m(X) ∩ S
16  ▷ Sonderbehandlung der letzten Ziffern
17  if X und Y haben beide Ziffern
18     then vielfache ← {Y_i · i  mod 10 | i ∈ ℕ}
19             faktoren ← {z | z ist Faktor von z modulo 10}
20             m(X_i) ← m(X_i) ∩ vielfache
21             m(Y_i) ← m(Y_i) ∩ faktoren
```

6.2 Hilfs-Regeln

6.2.1 NumberRule

Beschreibung. NumberRule sort dafür, dass die möglichen Werte für die Ziffern einer Zahl mit den möglichen Werten der Zahl abgeglichen werden. Wird etwa die mögliche Menge der Zahlen eingeschränkt, so kann dies Auswirkungen auf die möglichen Werte für Ziffern haben und umgekehrt.

Kann die letzte Ziffer der Zahl etwa nur noch 0 sein, so kann man alle möglichen Werte der Zahl, die nicht 0 als letzte Stelle haben, ausschließen.

Für jede Zahl wird beim Parsen der Feldbeschreibung eine NumberRule erzeugt.

Erzeugte Regeln. NumberRule erzeugt selbst keine Regeln.

Funktionsweise auf X_i. Wenn sich die Menge der möglichen Werte für die Zahl verändert, so werden die Ziffern aktualisiert: Es wird jeder mögliche Wert von X betrachtet. Für jeden dieser Werte w werden die einzelnen Stellen w_i betrachtet und die möglichen Werte für jede Stelle in s_i gesammelt. Danach wird die Menge der möglichen Werte für jede Ziffer X_i mit s_i geschnitten.

In Pseudocode:

```
1  if m(X) hat sich verändert
2     then ▷ Initialisiere s_i
3          for i ← 1 to k
4              do s_i ← ∅
5          ▷ Sammle alle mögliche Werte für die Ziffern
6          for w ∈ m(X_i)
7              do for i ← 1 to k
8                     do s_i ← s_i ∪ {w_i}
9          ▷ Entferne nicht mehr mögliche Werte für die Ziffern
10         for i ← 1 to k
11             do X_i ← X_i ∩ s_i
12
```

Funktionsweise auf X. Wenn sich für eine der Ziffern von X die Menge der möglichen Werte geändert hat, dann muss auch die Menge der möglichen Werte von X aktualisiert werden. Für jeden dieser möglichen Werte w von X wird betrachtet, ob die Stellen von w in den zugehörigen Ziffern von X vorkommen.

Ist die Menge der möglichen Werte von X noch nicht gesetzt, dann wird sie aus den Ziffern erzeugt sofern diese nicht eine zu große Menge induzieren.

In Pseudocode:

```
1  if m(X) == NIL ∧ Ziffern induzieren keine zu große Menge
2     then m(X) ← von Ziffern induzierte Menge
3  for w ∈ m(X)
4      do for i ← 1 to k
5             do if w_i ∉ m(X_i)
6                  then m(X) ← m(X) \ {w}
```

6.2.2 $NLZ(X)$ NoLeadingZeroRule

Beschreibung. $NLZ(X)$ bedeutet, dass die erste Ziffer einer Zahl den Wert 0 nicht annehmen darf. Die Regel wird beim Laden des Feldes automatisch für jede Zahl auf dem Feld erzeugt. Zahlen auf dem Feld selbst haben offensichtlich Ziffern.

Erzeugte Regeln. NoLeadingZeroRule selbst erzeugt keine Regeln.

Funktionsweise auf X_i. Der Wert 0 wird aus der Menge der möglichen Werte von X_0 entfernt.

In Pseudocode:

```
1  ▷ Vorbedingung: X hat Ziffern.
2  m(X_0) ← m(X_0) \ {0}
```

Funktionsweise auf X. Die möglichen Werte von X selbst werden durch NoLeadingZeroRule nicht verändert.

6.2.3 $REMOVE(X, M)$ RemoveValuesFromDigitsRule

Beschreibung. $REMOVE(X, M)$ bedeutet, dass die Werte aus der Menge M von den möglichen Werten der Ziffer X entfernt werden sollen. Die wird von PrimeRule (siehe Abschnitt 6.1.3) erzeugt, um bestimmte Werte von der letzten Ziffer einer Zahl auszuschließen.

Erzeugte Regeln. RemoveValuesFromDigitsRule selbst erzeugt keine Regeln.

Funktionsweise auf X. Die Werte aus M werden aus den möglichen Werten für X entfernt.
In Pseudocode:

1 $m(X) \leftarrow m(X) \setminus M$

6.3 Operatoren-Regeln

6.3.1 $IstQuersumme(X, Y)$ – CrossSumRule

Beschreibung. $IstQuersumme(X, Y)$ bedeutet, dass X die Quersumme von Y ist.

Erzeugte Regeln. CrossSumRule selbst erzeugt keine Regeln.

Funktionsweise. Wenn der eindeutige Wert von Y bekannt ist ($m(Y) = 1$), dann wird von diesem Wert die Quersumme s berechnet. Danach wird $\{s\}$ mit den bisher möglichen Werten für X geschnitten. Dies ergibt entweder die Menge $\{s\}$ oder die leere Menge.

Ist Y nicht bekannt, der Wert w von X aber ($m(X) = 1$), dann wird versucht auf die möglichen Werte für die Ziffern von Y zu schließen:

Dann geht die Regel die Ziffern Y_1 bis Y_k durch. Für jede Ziffer Y_i summiert die Regel für alle anderen Ziffern jeweils den höchsten möglichen Wert zu s auf. Der kleinste mögliche Wert von Y_i ist dann größer oder gleich $w - s$, sonst könnte w als Quersumme nicht erreicht werden.

Danach geht die Regel noch einmal die Ziffern von Y_1 bis Y_k durch und führt den im vorherigen Absatz beschriebenen Algorithmus noch einmal durch um die kleinsten möglichen Werte zu bestimmen: Für jede Ziffer Y_i summiert die Regel für alle anderen Ziffern jeweils den kleinsten möglichen Wert zu s auf. Der größte mögliche Wert von Y_i ist dann kleiner oder gleich $w - s$, sonst könnte w als Quersumme nicht erreicht werden.

Im Pseudocode:

```
 1  if |m(Y)| = 1
 2     then s ← QUERSUMME(Y)
 3          m(X) ← m(X) ∩ {s}
 4     else if |m(X)| = 1
 5             then w ← QUERSUMME(X)
 6                  ▷ Finde untere Schranke für mögliche Werte.
 7                  for i ← 1 to k
 8                      do s ← Σ_{i≠j} max m(Y_i)
 9                         m(Y_i) ← {z | z ∈ m(Y_i) ∧ z ≥ s}
10                  ▷ Finde obere Schranke für mögliche Werte.
11                  for i ← 1 to k
12                      do s ← Σ_{i≠j} min m(Y_i)
13                         m(Y_i) ← {z | z ∈ m(Y_i) ∧ z ≤ s}
```

6.3.2 $QuerProduktIst(X, Y)$ – CrossProductRule

Beschreibung. $QuerProduktIst(X, Y)$ bedeutet, dass X das Querprodukt von Y ist.

Erzeugte Regeln. CrossProductRule erzeugt selbst keine weiteren Regeln.

Funktionsweise. Wenn der Wert von Y bekannt ist $|m(Y)| = 1$ dann wird das Querprodukt u von X berechnet und $\{v\}$ mit den möglichen Werten von X geschnitten.

Ist der Wert von Y nicht bekannt, jedoch der Wert u von X, dann wird wird die Menge der möglichen Werte von Y auf die Werte eingeschränkt, die das Querprodukt u haben.

In Pseudocode:

```
1  if |m(Y)| = 1
2     then u ← QUERPRODUKT(Y)
3          m(X) ← m(X) ∩ {u}
4     else if |m(X)| = 1
5             then v ← Wert von X
6                  m(Y) ← {z | z ∈ m(Y) ∧ QUERPRODUKT(z) = v}
```

6.3.3 $IsMal(A, B, C)$ – MultOperatorRule

Die Vorgehensweise der MultOperatorRule ist weitgehend identisch zu der der AddOperatorRule und SubtractOperatorRule, da sie alle von der OperatorRule ableiten.

Beschreibung. $IsMal(A, B, C)$ bedeutet, dass A das Produkt von B und C ist.

Erzeugte Regeln. MultOperatorRule erzeugt selbst keine weiteren Regeln.

Funktionsweise. Sofern die möglichen Werte von B und C nicht zu viele Kombinationen ergeben, werden diese Produkte berechnet und mit den möglichen Werten von A geschnitten. Dann werden, sofen es wiederum nicht zu viele werden, die Quotienten aus Werten aus A und C berechnet und mit den Werten von B geschnitten. Selbiges dann wiederum für Quotienten aus A und B und den Werten aus C. Dieser Dreischritt wird, wenn es Aussicht aus Erfolg hat, ein zweites mal ausgeführt.

6.3.4 $IsPlus(A, B, C)$ – AddOperatorRule

Beschreibung. $IsPlus(A, B, C)$ bedeutet, dass A die Summe von B und C ist.

Funktionsweise. Die Funktionsweise ist mit der der MultOperatorRule identisch, nur dass sich A als Summe von B und C bildet und B bzw. C entsprechend als Differenz von A und C bzw. B.

6.3.5 $IsMinus(A, B, C)$ – SubtractOperatorRule

Beschreibung. $IsMinus(A, B, C)$ bedeutet, dass A die Differenz von B und C ist.

Funktionsweise. Die Funktionsweise ist mit der der MultOperatorRule identisch, nur dass sich A als Differenz von B und C bildet, B als Summe von A und C und C wiederum als Differenz von B und A.

6.3.6 $IsReverseOf(A, B)$ – FlipOperatorRule

Beschreibung. $IsReverseOf(A, B)$ bedeutet, dass A aus B entsteht, indem man die Reihenfolge der Ziffern umdreht.

Erzeugte Regeln. In dem Spezialfall, dass beide Parameter Zahlen auf dem Feld sind (die ja nicht mit Null beginnen können), wird diese Regel durch eine Reihe von EqualityRules auf Ziffern ersetzt, die diese Regel effizienter durchsetzen können.

Funktionsweise. Im allgemeinen Fall ist diese Regel wiederum nur eine Instanz der OperatorRule, die im Abschnitt MultOperatorRule beschreiben ist, wobei hier A aus B wie auch B aus A durch Umdrehen der Ziffern berechnet wird. Dabei kann B aus A nur berechnet werden, wenn $|B|$ bekannt ist, denn es gilt $r(1) = r(10) = r(100)$, und somit r nicht eindeutig invertierbar, wenn die Länge des Operanden unbekannt ist.

7 Graphische Benutzeroberfläche

Zusätzlich zum Textinterface entwickelten wir eine graphische Benutzeroberfläche, die das Vorgehen unseres Lösungsprogramms visualisieren soll. Dabei liegt der Schwerpunkt auf dem Nachvollziehen der automatischen Lösungsschritte, und weniger auf einem manuellen Lösen des Problems.

Die graphische Version von 7down startet man mit

```
$> python bin/7down-gui.py
```

7.1 Übersicht über die Bedienelemente

Darauf öffnet sich ein Fenster, das, wie in Abbildung 7 zu sehen, in zwei Hälften geteilt ist. Links ist eine graphische Darstellung des Feldes zu sehen, wobei stets in jeder Zelle die Ziffern zu sehen sind, die das Lösungsprogram noch für möglich hält. Insbesondere stehen *blaue Ziffern* für solche, die durch das Backtracking vorerst ausgeblendet werden.

Darunter werden die vom Backtracking angestellten Annahmen aufgeschlüsselt und für alle Zahlen, ob auf dem Feld oder als Zwischenzahl (siehe 3.4), werden die ersten paar gültigen Werte aufgelistet, sofern sie für diese Zahl bereits explizit bekannt sind.

Rechts im Fenster ist ein Reiter, der drei mögliche Ansichten bietet. Die erste ist der Programm-Log, in dem in Textform die Schritte der Lösungsfindund erläutert werden.

Der nächste Reiter bietet eine Übersicht über die Regeln, wie sie unser Program versteht. Das heißt, dass einige Regeln wie in 3.3 beschrieben durch einfachere ersetzt wurden und automatisch generierte Regeln wurden hinzugefügt. Wählt man eine Regeln in der Liste aus, so wird ihre Bedeutung darunter ausführlich erklärt und die entsprechenden Zellen des Felds und der Zahlenliste darunter farblich hervorgehoben.

Die letzte Ansicht erlaubt dem Benutzer, die Rätselbeschreibung zu betrachten und zu bearbeiten.

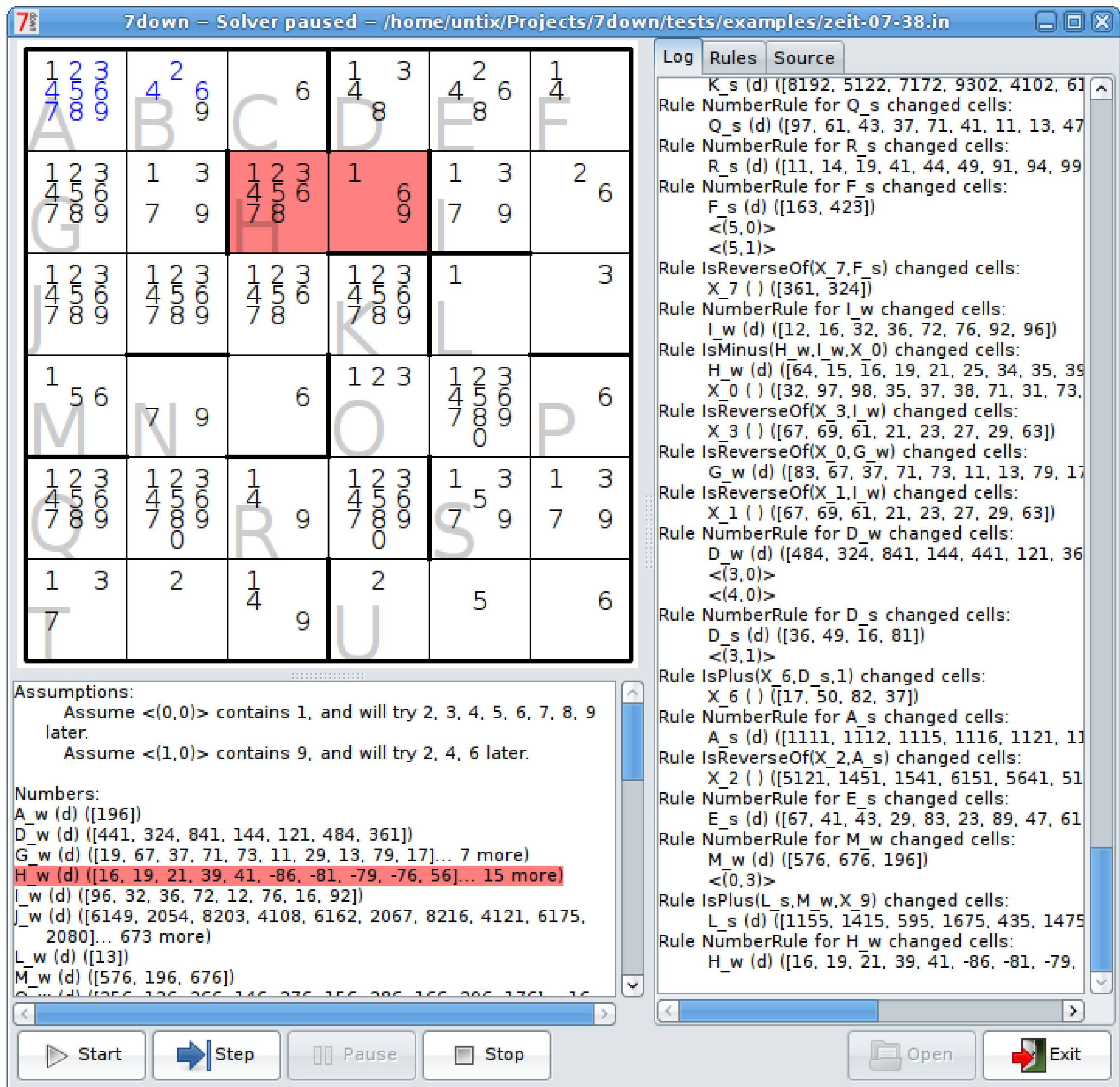

Abbildung 7: Die graphische Benutzeroberfläche in Betrieb

Darüberhinaus sind an der unteren Kante diverse Knöpfe aneinandergereiht, die im Folgenden erläutert werden. Die Titelleiste des Fenster gibt die aktuell geöffnete Datei an, sowie den Zustand des Programms.

7.2 Laden eines Rätsels

Um eine Datei mit einer Rätselbeschreibung zu laden, kann der „Load“-Knopf benutzt werden. Alternativ kann auch ein Dateiname über auf der Kommandozeile übergeben werden. Sofern der Bearbeiten-Tab geöffnet ist, liest das Programm die Datei auch direkt ein, und man kann sich etwa die Regeln genauer anschauen. Befindet sich der Benutzer jedoch im Bearbeiten-Modus, geschieht das Einlesen erst, wenn er diesen verlässt (siehe 7.5).

7.3 Den Lösungsvorgang beobachten

Sobald ein Rätsel geladen, lässt sich mit dem „Start“-Knopf der Lösungsprozess starten. Im Folgenden kann in der Log-Ansicht verfolgt werden, welche Schritte 7down unternimmt, um der Lösung näher zu kommen. Gleichzeitig wird auch die Anzeige der noch möglichen Ziffern aktualisiert, wie auch die gemachten Annahmen und die expliziten Werte der Zahlen. Hierbei werden die gerade geänderten Ziffern und Zahlen farblich hervorgehoben.

Der Pause-Knopf unterbricht den Lösungsvorgang. Jetzt kann mit dem „Step“-Knopf einen Schritt des Lösers ausgeführt werden. Nach diesem Schritt befindet sich das Programm wieder im Pause-Zustand. Der „Start“-Knopf lässt den Solver nach einer Pause weiterlaufen. Um die Ausführung abzubrechen und das System in den Ausgangszustand zu versetzten, kann der „Stop“-Knopf benutzt werden.

7.4 In den Lösungsvorgang eingreifen

Dem Benutzer werden beschränkt Möglichkeiten geboten, den Lösungsvorgang gezielt zu steuern. Dabei muss zwischen den Änderung vor und während des Lösungsvorgangs unterschieden werden.

Nachdem ein Rätsel geladen ist, aber bevor auf den Start- oder Step-Knopf gedrückt wurde, können durch einfache Klicks einzelne Ziffern im Feld entfernt werden. Diese werden dann im späteren Lösungsverlauf komplett ignoriert. Hält man während dem Klick die Umstelltaste gedrückt, verschwinden alle anderen Ziffern in der gewählten Zelle.

Während der Löser seine Arbeit verrichtet, bewirkt ein Klick auf eine Ziffer lediglich, dass diese blau markiert wird. Diese Ziffer wird vorerst ignoriert, sollte sich aber keine Lösung finden lassen, wird sie wieder als Kandidat angenommen. Dies ist auch gut unterhalb des Felders, in der Liste der Annahmen, nachzuvollziehen. Wie vor dem Start kann hier mit der Shift-Taste erreicht werden, dass alle anderen Ziffer der gewählten Zelle blau markiert werden.

7.5 Bearbeiten von Rätseln

Experimente mit Rätseln lassen sich direkt in der graphischen Oberfläche vornehmen, ohne diese verlassen zu müssen. Dazu muss man in der Quell-Ansicht das Häkchen bei „Edit Source“ setzen, damit das Feld bearbeitbar wird. Dazu darf der Löser nicht gestartet sein, also muss man diesen gegebenenfalls mit dem „Stop“-Knopf abbrechen.

Die Syntax der Rätsel ist die in der Aufgabenstellung vorgegebene. Um die Veränderungen zu übernehmen muss das Häkchen wieder entfernt werden. Nun wird der Text eingelesen und,

sofern kein Fehler auftritt, kann der Lösungsvorgang gestartet werden. Im Fehlerfall wird eine bedingt hilfreiche Fehlermeldung angezeigt und das Programm verbleibt im Bearbeiten-Modus.

Ist die Quellansicht geöffnet, werden auch noch zwei „Speichern“-Knöpfe eingeblendet, damit die auf diesem Weg erstellten Rätsel auch erhalten bleiben können.

7.6 Implementierungs-Bemerkungen

Für die graphische Oberfläche haben wir PyGTK [1], die Gtk-Anbindung für Python, gewählt. Ausschlaggebend war hier vor allem die im Team vorhandene Erfahrung damit, aber auch die Platformunabhängigkeit. Darüber hinaus bietet die eng mit Gtk verbundene Vektor-Graphikbibliothek Cairo schöne Ergebnisse mit Anti-Aliasing und Alpha-Transparenz bei geringem Aufwand. Auch kann Cairo nicht nur auf den Bildschirm zeichen. So konnte die gleiche Funktion, die in der GUI das Logo malt, von einem externen Skript aufgerufen werden und verwendet werden, um das Logo als Vektorgraphik in eine PDF-Datei zu zeichnen und etwa hier in diese Dokumentation einzubinden.

Die Zeichenfunktionalität des Feldes haben wir in einem eigenen Gtk-Widget gekapselt, da ursprünglich noch weitere Varianten der GUI geplant warnen, deren Funktionalitäten dann doch von der einen, jetzt vorhandenen, abgedeckt werden.

Der Programmablauf innerhalb der graphischen Oberfläche wird als Zustandsautomat modelliert und Knöpfe, die im aktuellen Zustand nicht sinnvoll sind, werden abhängig vom Zustand deaktiviert. So ist es etwa nicht möglich, während der Bearbeitung eines Rätsels den Löser zu starten: Vorher muss das Rätsel eingelesen werden, indem man den Bearbeiten-Modus verlässt.

8 Softwaretechnisches Vorgehen

Für das Projekt schien uns ein formales, monolithisches Vorgehen übertrieben komplex und schwerfällig – keiner der Beteiligten hat vorher schon einmal ein ähnliches Programm geschrieben.

Zuerst planten wir das generelle Vorgehen. Dabei wurde die Grobstruktur des Programms, also die in Abschnitt 4 beschriebenen Module und Klassen identifiziert und beschrieben. Als Programmiersprache wählten wir Python, der Konsistenz wegen benutzten wir für Bezeichner und Kommentare englische Begriffe.

Dann schrieben wir einen ersten Prototypen in Haskell, der kleine Felder mit einfachen Regeln lösen konnte. Der Prototyp probierte einfach alle Kombinationen durch und prüfte jeweils, ob die Lösung gültig war. Weiterhin waren die Probleme fest im Programm ,,verdrahtet”.

Danach implementierten wir iterativ und modulweise die geplanten Klassen. Begonnen beim Parser, den Klassen für das Feld und dananch iterativ die einzelnen Regeln. Wir schrieben Quelltext und Code im ,,Tests danach” Ansatz. Möglichst jeder Teil Code sollte, *nachdem* er geschrieben war, dokumentiert sein.

Für den zweiten Aufgabenteil planten wir wieder erst grob die GUI um sie dann iterativ umzusetzen.

8.1 Qualitätssicherung

Wir arbeiteten mit vier Personen am gleichen Quelltext (über ein Versionskontroll-System, siehe Abschnitt 8.2). Dieser war zwar modular, aber jeder Programmierer sollten auch an jedem Teil des Codes arbeiten können.

Damit dies nicht unerkannt Qualitätsprobleme mit sich bringen würde, einigten wir uns zum einen darauf, jede Methode und Klasse möglichst gut zu dokumentieren. Python bringt hierfür sog. *docstrings* mit, aus denen sich dann auch eine ansprechende API-Dokumentation generieren lässt.

Desweiteren sollten für möglichst jede Method bzw. Klasse soweit sinnvoll ein Unit Test existieren. Dies fing viele Fehler ab und diente gleichzeitig gut als Dokumentation.

Weiterhin, schrieben wir System-Tests, die wir auf Beispieleingaben los ließen. Eingaben waren etwa die vorgegebenen Beispiele, aber auch einige Kreuzzahlenrätsel aus dem Internetauftritt der Zeitung ,,Die Zeit".

Neben einer möglichst geringen Anzahl von Fehlern war uns auch die Geschwindigkeit des Lösens wichtig. Da als Implementierungs-Sprache Python gewählt war, erhofften wir uns kein Lösen von schweren Rätseln im Mikrosekundenbereich. Wir wollten jedoch den Effekt von Optimierungen beobachten können.

Dafür schrieben wir ein Werkzeug, was für jede Änderung automatisch alle Tests durchlaufen ließ und die Zeit stoppte sowie Zählte, wie oft eine Regel angewandt wurde oder ein Backtracking-Schritt unternommen wurde.

Der Parser für die Rätsel-Beschreibung wurde mit dem Fuzzer zzuf auf nicht abgefangene Randfälle überprüft.

8.2 Entwicklungs-Werkzeuge

Die gewählte Programmiersprache Python ermöglicht es, effizient Code zu schreiben ohne zwangsläufig auf große Entwicklungsumgebungen zurückgreifen zu müssen. Somit verwendeten wir jeweils einfache Texteditoren mit Syntax-Highlighting (etwa vim oder TextMate).

Um die gröbsten Fehler bei der Zusammenarbeit zu vermeiden benutzten wir das Versionskontroll-Sytem ,,Mercurial". Die hiermit erstellten Versionen benutzten wir auch um die in Abschnitt 8.1 beschriebene Historie von Laufzeiten zu erstellen.

Weiterhin benutzten wir das Werkzeug ,,Glade", um die Dialoge für die graphische Oberfläche zu erstellen.

Um festzustellen, welche Teile des Codes zu langsam laufen, benutzten wir den mit Python mitgelieferten Profiler.

9 Mathematische Abstrahierung

Ein Ziel unseres Ansatzes war es, von der konkreten Aufgabe soweit zu abstrahieren, dass die grobe Herangehensweise von der Logik her sehr überschaubar wird und auch andere Problemarten lösen kann.

Diese Abstraktion kennt nur zwei Objekttypen: Zellen, die Mengen im mathematischen Sinne sind, und Regeln, die diese diese Mengen modifizieren. Die konkrete Implementierung des Kreuzzahl-Problem muss dann nur noch geeignete Instanzen für Zellen und Regeln finden, dem gegebenem Feld und den zugehörigen Aussagen über die Felder entsprechen.

9.1 Mathematische Modellierung

Betrachen wir nun, welche Eigenschaften Zellen und Regeln erfüllen müssen. Sei dazu I eine Indexmenge für die Zellen und Ω_i der Grundraum der Werte, welche die Zelle mit Index i

aufnehmen kann. Ein Zustand des Problems ist ein Z im Zustandsraum $\mathcal{Z}$:

$$Z \in \mathcal{Z} := \Big\{ \prod_{i \in I} Z_i,\ Z_i \subseteq \Omega_i \text{ für } i \in I \Big\}$$

Ein Zustand Z heißt *genauer* als ein Zustand Z', geschrieben $Z \leq Z'$, wenn $Z_i \subseteq Z'_i$ für alle $i \in I$ gilt.

Die Regeln $R \in \mathcal{R}$ sind nun Abbildungen von Zuständen, also

$$\begin{aligned} &R^j : \mathcal{Z} \to \mathcal{Z} \\ &\prod_{i \in I}(Z_i) \mapsto R^j(\prod_{i \in I}(Z_i)) = \prod_{i \in I}(R_i^j(Z_i)) \end{aligned}$$

Wir nehmen an, dass die Aufgabe genau eine gültige Lösung $(l_i)_{i \in I}$ mit $l_i \in \Omega_i$ hat. Gesucht ist also ein Zustand $L \in \mathcal{Z}$ mit $L_i = \{l_i\}$ für alle $i \in I$.

Weiter fordern wir von den Regeln $R \in \mathcal{R}$, gewisse Eigenschaften:

- Korrektheit:
$$L_i \leq Z_i \implies L_i \leq R(Z)_i$$
- Monotonie
$$R(Z) \leq Z$$
- Fehler-Erkennung: Gilt $|Z_i| = 1$ aber $Z_i \neq \{l_i\}$ für ein $i \in I$, das Problem also nicht mehr lösbar ist, so gibt es eine Regel $R \in \mathcal{R}$, so dass $R(Z)_{i'} = \emptyset$ für ein $i' \in I$ gilt. (Üblicherweise gilt dann wohl $i = i'$, das ist aber nicht zwingend nötig).

9.2 Algorithmus

Der Algorithmus berechnet jetzt eine Folge $Z^n \in \mathcal{Z}$, $n = 0, \dots, N$ ausgehend von $Z^0 = \prod_{i \in I} \Omega_i$, so dass $Z^N = L$ gilt.

Dabei berechnet sich für $n > 0$ der neue Zustand Z^n aus Z^{n-1} durch Anwendung aller Regeln, also

$$Z^n = (\bigcirc_{R \in \mathcal{R}} R)(Z^{n-1})$$

sofern es eine Regel $R \in \mathcal{R}$ gibt, für die $R(Z^{n-1}) \neq Z^{n-1}$ gilt. Die Eigenschaften für Regeln garantieren uns, dass wir so der Lösung näher kommen und diese nicht verlieren.

Gibt es dagegen keine Regel, die den Zustand verändern kann, so wird Backtracking gemacht. Das heißt, man wählt ein $i \in I$ mit $|Z_i^n| > 1$ und eine Partitionierung $\mathcal{P}$ von Z_n mit mindestens zwei Partitionen. Für jede Partition $P \subset Z_n$ setzt man $Z_i^n = P$ und $Z_j^n = Z_j^{n-1}$, für $j \neq i$, und fährt mit dem Verfahren fort. Für genau eine Partition (nämlich jede mit $l_i \in P$) führt dies zu der gewünschten Folge von Zuständen.

9.3 Optimierungs-Strukturen

Diese Modellierung ist ausreichend, um gegebene Probleme zu lösen. Allerdings benutzt unser Solver-Code zur Optimierung weitere Konstrukte:

Zu jedem Zellindex $i \in I$ gibt es eine Liste von Regeln $B_i \subseteq \mathcal{R}$, auch nach erstmaliger Anwendung der Regel bei späteren Änderungen der Zelle i erneut angewandt werden möchten. Dabei sollte eine Regel $R \in \mathcal{R}$ in den Zellen $B_R := \{i \in I, R \in B_i\}$ registriert sein, dass für

alle $Z, Z' \in \mathcal{Z}$, die kleiner sind als ein Zustand in $R(\mathcal{Z})$ und für die $Z_i = Z'_i$, $i \in B_R$, gilt: $R(Z) = Z \iff R(Z') = Z'$.

Weiter ist für jede Regel und jeden Zustand bekannt, welche Zellen bei Anwendung der Regel auf diesen Zustand geändert werden:

$$\begin{aligned} C : \mathcal{Z} \times \mathcal{R} &\to \mathcal{P}(I) \\ (Z, R) &\mapsto \{i \in I : R(Z)_i \neq Z_i\} \end{aligned}$$

Dies wird verwendet, um in jedem Schritt $n = 0, \dots, N$ eine Liste von anzuwendenten Regeln $T_n \subseteq \mathcal{R}$ abzuarbeiten, beginnend mit $T_0 = \mathcal{R}$. Abweichend von oben ist die Vorschrift für einen Lösungsschritt im Falle $T_{n-1} \neq \emptyset$ für ein $R \in T_{n-1}$ gegeben durch

$$Z^n = R(Z^{n-1})$$

wobei im nächsten Schritt die Regeln

$$T_n := (T_{n-1} \cup \bigcup_{i \in C(Z^{n-1}, R)} B_i) \setminus \{R\}$$

angewandt werden.

Ist dagegen $T_{n-1} = \emptyset$, so führen wir Backtracking bezüglich einer Zelle $i \in I$ wie oben beschrieben durch, wobei im nächsten Schritt die Regeln

$$T_n = B_i$$

angewandt werden.

So werden Regelaufrufe vermieden, die sowieso keinen Effekt haben würden.

10 Entwicklungsverlauf

Das 7down-Team fand sich erst relativ spät zusammen – das Repository wurd em 26. Oktober angelegt. Davor wurde nur ein eingeschränkter Prototyp für das Problem in Haskell implementiert, der Kreuzzahlrätsel ohne Operatoren und mit nur gewissen Prädikaten löste. Dieser Code wurde später nicht weiter verwendet, beeinflusste aber möglicherweise die Designentscheidungen.

Die folgenden Wochen fanden wir uns regelmäßig einen Abend pro Woche zusammen, um das Programm voranzutreiben. Anfangs lag das Augenmerk vor allem auf der vollständigen Implementierung der gegebenen Regeln, und Optimierung oder Lösungs-Intelligenz wurde zurückgestellt. Diese Entwicklung lässt sich an den Namen verschiedener Implementierungen verfolgen: Auf den unvollständigen StupidSolver folgte bald der SimpleSolver. Erst Mitte November löste diesen dann der OrderedSolver ab, der erstmal verfolgte, welche Regeln überhaupt anzuwenden sind, da sich ihre Zellen verändert haben. Die graphische Oberfläche stellte weitere Anforderungen an die Zugänglichkeit des Zustand des Solvers, und so konzentrierte sich die Entwicklung ab Dezember auf den StackSolver.

Die graphische Oberfläche entstand während und nach den Weihnachtsferien. Währenddessen wurden auch die Optimierungsbemühungen intensiviert, welche die automatische Performance-Statistik oft gnadenlos als uneffektiv blosstellte. Trozdem gelang es uns, die Ausführungszeit der meisten Rätsel auf ein erträgliches Maß zu drücken.

Ein Problem waren die Unterschiede in der Programmiererfahrung innerhalb des Teams, insbesondere da einige von uns bisher nicht mit Python oder Gtk programmiert haben. Sofern

möglich, wurden die Aufgaben entsprechend verteilt und nicht erwartet, dass von allen Code-Beiträge in gleichen Maßen entstanden.

In den letzten sieben Tagen vor Abgabe haben wir uns dann dreimal getroffen, zuletzt am Montag abend – Programmieren bis zur letzten Minute wollten wir vermeiden. Echten Zeitdruck hatten wir dabei nicht, da alle relevanten Teile schon fertig waren, und so konzentrierten wir uns auf allgemeinen Feinschliff und diese Dokumentation.

Insgesamt ergibt eine Abschätzung Auswertung der Commit-Zeiten, dass 215 Mannstunden Programmierarbeit in das Projekt flossen. Dabei entstanden 5500 Zeilen Programmcode, 3200 Zeilen Testcode und diese Dokumentation.

A Ein/Ausgabe des Programms

Das Konsolenprogramm *bin/solve-puzzle.py* erwartet den Dateinamen der Eingabe als Parameter oder den Inhalt der Datei auf der Standardeingabe. Gibt es eine Lösung so wird sie ausgegeben. **Gibt es keine Lösung, so gibt das Programm nichts aus.**

B Installation und Aufruf

Systemanforderungen. Auf dem Zielsystem sollte die Version *2.4* oder *2.5* des *Python* Interpreters installiert sein. Weiter wurde das Programm nur auf *Linux* und *Mac Os X* getestet. Für die graphische Benutzeroberfläche ist *PyGTK* nötig. Getestet wurde dies mit Version 2.12. Es sollte auf weiteren Unix-änlichen Betriebssystemen laufen, unter Windows wurde es nicht getestet.

Installation. Das Programm kann einfach installiert werden, indem die Datei *sevendown.tar.gz* in ein beliebiges Verzeichnis entpackt wird.

Aufruf des Konsolen-Programms. Das Lösungsprogramm befindet sich im Unterverzeichnis *bin*. Die Aufgabendatei kann entweder auf die Standardeingabe geschrieben werden oder es kann der Dateiname als Parameter übergeben werden.

Beispiele:

```
$> python2.5 bin/solve-puzzle.py < datei.txt
$> python2.5 bin/solve-puzzle.py datei.txt
```

Aufruf der graphischen Benutzeroberfläche Der Aufruf der graphischen Benutzeroberfläche geschieht durch:

```
$> python bin/7down-gui.py
$> python bin/7down-gui.py datei.txt
```

Optional kann noch – wie in der zweiten Zeile des Listings – eine Datei mit übergeben werden. Diese wird dann beim Programmstart geladen (man kann natürlich dann noch andere Dateien öffnen). Die Bedienungsanleitung für die graphische Benutzeranleitung finden sie in Sektion 7.

Literatur

[1] The python gtk bindings.

[2] William Sit. On Crossnumber Puzzels And The Lucas-Bonaccio Farm. 1998.

[3] Wikipedia. Observable Universe.