

## 39. Bundeswettbewerb Informatik

### 1.Runde

C. V. Esteban

November 23, 2020

#### Aufgabe 5: Wichteln

#### Lösungsidee

**Am Anfang** denkt man direkt, dass es darum geht; einfach ein sogenanntes "*Sudoku*" zu lösen. Man müsste nur eine Menge  $\mathbf{L}$  (Lösungsmenge) mit den folgenden Eigenschaften erstellen bzw. mithilfe einer logischen Maschine wie z.B. Prolog:

$$\mathbf{M}^{n \times 3} = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 3 \\ \ddots & \ddots & \ddots \\ x_1a & x_2b & x_3c \end{pmatrix} \quad \text{wobei} \quad \forall x_i j : x_i \in [1 \mid I]$$

$$\mathbf{L} := \{(M_{1x}; M_{2y}; M_{3z} \dots M_{nr}) / M_{ab} \neq M_{cd}\} \wedge |\mathbf{L}| = n \wedge \mathbf{L} \equiv \{1; 2; 3 \dots n\}$$

Danach aber, nur nach den Beispieldatein angeschaut zu haben, scheint es komplizierter: Es kann ein oder mehrere Geschenke geben, die von keinem Kind gewünscht werden (*wichteln2.txt*). Also, muss man das Problem ein bisschen erweitern und sich selber andere Lösungswege denken. Aber eine Sache steht ganz klar: Man muss die **beste** Anordnung finden. Was passiert mit den Geschenken, die von niemandem gewünscht sind? Ich nehme

an, dass man frei ist, sie zu irgendeinem Kind zugeben. Und da man weißt, dass *jeder Kind ein Geschenk mitgebracht hat*; müssen am Ende genauso viele Geschenke übrig bleiben wie Kinder ohne Geschenke.

Die nächste wichtige Frage ganz genau zu beantworten ist: Was bedeutet dass eine Anordnung besser als eine andere ist? In der Aufgabestellung steht dass die *Parameter* nach den man eine Lösung bewerten muss sind die Anzahl der Kinder, die sein erstes Wunsch bekommen haben; dann die die ihre zweite Wunsch bekommen haben und am Ende natürlich die Anzahl der Kinder die sein Drittes Wunsch bekommen haben. Da wir die neue Möglichkeit betrachtet haben, dass ein Kind irgendein Geschenk bekommt; müssen wir auch diese Möglichkeit ein Wert geben; und zwar in einer intuitiver Weise. Sobald es ein Kind mehr gibt, die sein erstes Wunsch bekommen hat; ist diese Anordnung besser. Egal ob deswegen, bekommen andere 3 Kinder z.B. ein zufälliges Geschenk ... sehr pragmatisch. Mit diesen zwei Sachen sehr klar in Kopf (wie eine Anordnung besser als eine Anderer ist und, dass am Ende *jedes* Geschenk wird zu *einem* Kind zugeteilt) merkt man dass das Problem einfacher wird, indem man als Priorität auf die Wünsche passt. Egal was es f}ur erste Wünsche gibt, damit es keine andere bessere Lösung gibt; muss man erst alle mögliche erste-Wünsche erfüllen. Aber man muss nicht davon ausgehen, dass es *nur eine* Weise gibt, in der man *alle mögliche erste-Wünsche* erfüllt. Genau daran liegt das wichtigste Teil der Lösungsidee... keine Fälle einfach hängen lassen. Wenn es mehrere Anordnungen gibt, die diese Eigenschaft besitzen, dann folgt natürlich die Frage ob es einer inzwischen den gibt, die besser als die anderen ist (ein Beispiel dazu in dem Beispiel Abteilung). Ganz grob kann man sagen dass wenn es 2 Kinder gibt, die als erster Wunsch das Geschenk 8 wollen; aber einer von den will das Geschenk 9 als zweite Wunsch und *keiner anderer mehr*. Dann wäre es besser das Geschenk 8 dem Kind geben, dessen zweite Wunsch nicht das Geschenk 9 ist; damit man *überhaupt* das Geschenk 9 zu jemandem zuteilen könnte. Jetzt wollen wir mit der richtigen Umsetzung anfangen.

## Umsetzung

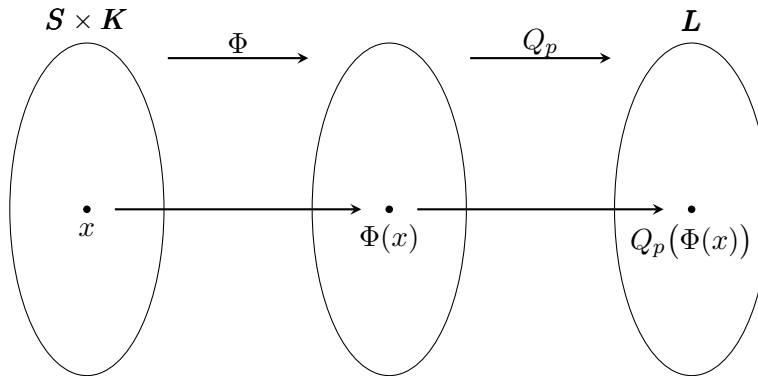
Jetzt geht es darum die oben erklärten Ideen als informatische Datenstrukturen und "Algorithmen" zu übersetzen. Dennoch das Quellcode wird nicht hier "erklärt", aber im "Quellcode" Abteilung. Wie es schon erwähnt wurde, das Kern dieser Lösung ist in einer sogenannte logischen Maschine: Ein "State" wird gegeben (die mögliche Kinder, die noch kein Geschenk bekommen haben bzw. ein Vektor/Kollektion dieser Form  $\rightarrow [[k_i [a \ b \ c]] [k_j [x \ y \ z]] \dots]$  wobei  $k_i$  entspricht das Kinder Nummer, was dasselbe als das Index von  $[a \ b \ c]$  ist). Dann, wie gesagt, guckt man nur auf die erste Wünsche aller Kinder und stellt man eine andere Kollektion  $\rightarrow [a \ x \dots]$ . Diese Kollektion ist sehr wichtig. Dadrin sind alle erste-Wünsche und zwar so, dass z.B.  $a$  auf der Position 0 ist und entspricht das erstes Wunsch von dem ersten Kind... so haben wir kind und Wünsche "verbinden". Am Ende machen wir aus dieser Kollektion ein sogenanntes **Set**, so dass  $[a \ b \ a \dots] \rightarrow \{a \ b \dots\}$ . Das machen wir um zu wissen, wie viele *verschiedene* Wünsche es gibt. Und jetzt das Punchline... diese letzte **Set** enthält die maximale Anzahl von erste-Wünsche (und auch genau welche) die erfüllt werden könnten. Wir machen fast dasselbe mit die zweite-Wünsche und dritte-Wünsche. Die einzelne Unterschied liegt daran, dass wir die Geschenke nicht betrachten wollen, die als erstes Wunsch zugeteilt werden. Also, nochmal um die Idee deutlich klar zu machen: Es geht darum zu wissen, welche Geschenke **muss** ich als erstes-Wunsch zuteilen; welche als zweites Wunsch; und welche als drittes Wunsch. Deswegen ziehen wir von dem zweiten Set alle Geschenke aus dem ersten Set. Und beim nächstes mal (bei Set 3) ziehen wir alle Geschenke die in den anderen zwei Sets mitendrin sind.

Die nächste Idee ist die Agrupierung von Kinder die diesselbe erste-Wunsch bzw. zweite-Wunsch/drite-Wunsch haben. Das machen wir so: Wir definieren eine Abbildung (Funktion)  $\Phi$  zwischen die Kollektion  $[a \ b \ a \dots]$  und das Set  $\{a \ b \dots\}$ , sodass  $\forall x \in \mathbf{S} : f(x) := \{(i_1; i_2; \dots) / \mathbf{K}[i_j] = x\}$  wobei  $\mathbf{K}[i_j]$  entspricht das element auf dem Index  $i_j$  in der Kollektion  $\mathbf{K}$ . Also, wir gucken uns an; welche Kinder gibt es, die dasselbe Geschenk als

dasselbe Priorit"at wollen (erstes, zweites, drittes -Wunsch). Und das machen wir nochmal mit jedem Wunsch. "Wozu?" mögen Sie fragen. Das Antwort wird sicherlich besser in der *Beispiele* oder *Quellcode* Abteilung erklärt werden. Aber das Antwort lautet ganz einfach dass wenn ich die Geschenke  $[a \ d \ e]$  zu den Kinder  $[x \ y \ z]$  gebe, würde ich gerne wissen ob es irgendein Geschenk gibt, die **nur** von  $[x \ z]$  gewollt wurde. Wenn das so ist, dann weiss ich *automatisch* dass dieses Geschenk  $f$  ganz am Ende zufällig zugeteilt wird.

In dieser Schritt kommt die logischen Maschine ins Spiel: Wir haben viele verschiedene mögliche Anordnungen und ein *Parameter* bzw. "Constraint", das die Suche regulieren könnte. Dann lautet das Ziel bzw. *Query* so:  $\mathbf{L} := \{(l_1; \ l_2 \ \dots) \mid l_i \in \Phi[i]\}$ . Die optimalste Lösung wäre dass man  $\mathbf{L}$  bestimmen könnte, sodass kein Geschenk, das als zweitens oder drittes Wunsch ist; sich "nach außen rütschen müsste". Aber manchmal wird das einfach nicht möglich. Deswegen, hat diese Query  $\mathbf{Q}_p$  ein Parameter  $p$ , die entspricht die maximale Anzahl von Geschenke die "sich rütschen dürfen". Man fangt mit 0 an, und wenn es nicht klapp (wenn die Maschine kein passendes  $\mathbf{L}$  findet), dann versucht man mit 1 und so weiter. Wenn alles gut läuft, dann kriegt man eine Kollektion von "kinder" bzw. Indexen. Die Zahlen die auf diesen Positionen sich befinden, sollten "gekreuzt" werden. Dieses Verfahren ist im Figur 1 deutlich zu erkennen.

Figure 1: Funktionelles Modell



Man wiederholt das ganze mit kleinen veränderungen, wie zum Beispiel dass beim zweiten mal muss man nicht auf dem ersten Wunsch passen, sonder auf dem dritten Wunsch. Aber im Grunde genommen ist das Verfahren dasselbe. Die Beispielen und das Quellcode werden alles besser bzw. genauer erklären.

## Beispiele

Aufgabe5sample.txt...

DATA

[0 (2 10 6)]

[1 (2 7 3)]

[2 (4 7 1)]

[3 (3 4 9)]

[4 (3 7 9)]

[5 (4 3 2)]

[6 (7 6 2)]

[7 (10 2 4)]

[8 (9 8 1)]

[9 (4 9 6)]

$$\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 3 \\ \ddots & \ddots & \ddots \\ x_1a & x_2b & x_3c \end{pmatrix}$$

FIRST-COLUMN-PAIRING

$$\{S\{7\ 4\ 3\ 2\ 9\ 10\}$$

$$(6\ 2\ 3\ 0\ 8\ 7)\}$$

After-first-column-pairing

$$([1\ (2\ 7\ 3)]$$

$$[4\ (3\ 7\ 9)]$$

$$[5\ (4\ 3\ 2)]$$

$$[9\ (4\ 9\ 6)])$$

second-column-pairing

$$\{S\{\}$$

$$nil\}$$

after-second-column-pairing

$$([1\ (2\ 7\ 3)]$$

$$[4\ (3\ 7\ 9)]$$

$$[5\ (4\ 3\ 2)]$$

$$[9\ (4\ 9\ 6)])$$

third-column-pairing

$$\{S\{6\}$$

$$(9)\}$$

free-spots

$$\{S\{158\}$$

$$S\{145\}\}$$

ANTWORT...

”The kids who receive their first wish are: [623087] those who receive their second wish are: [], and the ones receiving their third/last wish are [9] And the kids [145] will receive one random gift out of the following gifts [158]”

## Quellcode

Das Quellcode wird genauso wie bei der Umsetzung Abteilung beschrieben im Sinne Aussichtspunkt: Die wichtigsten Funktionen. Die erste Funktion ist **description**, was bisher keine Name zugegeben wurde, weil die eigentlich eine Mittelfunktion bzw. Hilfsfunktion ist. Mithilfe von *high-order-functions* ergibt diese Funktion ein *HashMap* wieder. In *first-wish* sind alle Geschenke die als erstes Wunsch zugeteilt werden **müssen**. Dasselbe passiert mit *second-wish* und *third-wish*. Damit sind wir im Raum  $S \times K$ .

```
(defn description
  [data]
  (let [per-column (map (fn [f]
                        (map (fn [v] [(first v) (f (second v))])
                          data))
    [first second last])
    uniques (map (fn [v] (set (map second v))) per-column)
    first-wish (first uniques)
    second-wish (set/difference (second uniques) (first uniques))
    third-wish (set/difference (last uniques) (set/union (second uniques)
                                                         (first uniques)))]
    {:per-column per-column :f first-wish :s second-wish :l third-wish}))
```

Jetzt kommt die Funktion  $\Phi$ . Diesmal wurde ein *Threading Macro* ( $->>$ ) und andere *high-order-functions* wie *filter* benutzt. Aus dieser Funktion kriegen wir welche Kinder

wollen welche Geschenke als erstes-, zweitens-, und drittens- Wunsch.

```
(defn structure
  [data]
  (map (fn [f k]
        (apply conj (map (fn [v] {v (->> (f (:per-column (description data)))
                                         (filter #(= (second %1) v))
                                         (map first))})
                      (k (description data))))))
    [first second last] [:f :s :l]))
```

Die Funktion **find-configuration** ist die logische Maschine (clojure.core.logic) die von David Nolen implementiert wurde. Die Ideen stammen aus Prolog; genauer gesagt aus Minikaren (eine minimalistische Implementation von *constraint programming*). Und die Regeln, die ich definiert habe sind folgende: Ich wähle  $n$  Elemente aus  $n$  Listen ( (rec-membro vars pool)), sodass wenn ich die Potenzmenge dieser Menge/Kollektion mir an-  
gucke; sollte kein Element dadrin sein, das ist auch bei *cond1* oder *cond2*. Also, ich will die Indexes so auswählen, dass es kein anderes Geschenk die genau nur auf den Indexen stehen kann.

```
(defn find-configuration
  [parameter pool cond1 cond2]
  (if (empty? pool)
      '()
      (let [vars (repeatedly (count pool) logic/lvar)
            S (map #(remove (fn [_] (= _ nil)) %1) (Superset [] vars))]
        (logic/run 1 [q]

          (rec-membro vars pool)
```

```

      (macro/symbol-macrolet [a (count (filter (fn [c] (membero-coll c S)) cond1))
                             b (count (filter (fn [c] (membero-coll c S)) cond2))]
        (fd/eq
          (<= a parameter)
          (<= b parameter)))

      (logic/== q vars))))))

```

Am Ende mache ich ein *Tester* um zu überprüfen ob alles in Ordnung gelaufen ist. Alles ist gut, wenn am Ende die erste-, zweite-, dritte- Wünsche und die sogenannte *free-spots* zu der gesamten Anzahl der Kinder addieren (dasselbe passiert mit den Geschenken).

```

(defn true-test []
  (let [answer (map (fn [f]
                     (apply set/union (map (fn [c]
                                             (f (first c)))
                                             [first-column-pairing
                                              second-column-pairing
                                              third-column-pairing
                                              free-spots])
                                           [first second]))
                    (= (count (first answer)) (count (second answer))))))

```

Da ich die Dateien aus einem .txt File gelesen habe, sollte ich auch das Antwort auf demselben File schreiben. Das Antwort ist ein großer String, das aus verschiedenen Teilen besteht: Erst sind die Kinder, die sein erstes Wunsch bekommen und so und so fort.

```

(defn answer-paolo []
  (let [output (if (true-test)

```

```

(str "\n The kids who receive their first wish are: "
  (into [] (first (vals first-column-pairing))) "\n"
  ",those who receive their second wish are: "
  (into [] (first (vals second-column-pairing))) "\n"
  ", and the ones receiving their third/last wish are "
  (into [] (first (vals third-column-pairing)))

  (if (empty? (first (vals free-spots)))
    ""
    (str "\n And the kids "
      (into [] (first (vals free-spots)))
      " will receive one andom gift out of the following gifts "
      (into [] (first (keys free-spots))))))
  "Something went wrong")
(with-open [wrtr (clojure.java.io/writer
  (io/resource "clojure_version/aufgabe5sample3.txt")
  :append true)]
  (.write wrtr output))
output))

```