# Aufgabe 5: Writer-Dokument

Team-ID: 00000

Team-Name: Name

Bearbeiter/-innen dieser Aufgabe:

Emilis Guzys

Leonard Kraus

C. V. Esteban

November 23, 2020

## Contents



## Lösungsidee

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

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

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

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 Anordung 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 pragmatish.
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 Anordungen gibt, die diese Eigenschaft besitzen, dann folgt natülich 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 Geschen 9 ist; damit man *überhaupt* das Geschenk 9 zu jemandem zuteilen könnte. Jetzt wollen wir mit der richtigen Umsetzung anfangen.

## Umsetzung

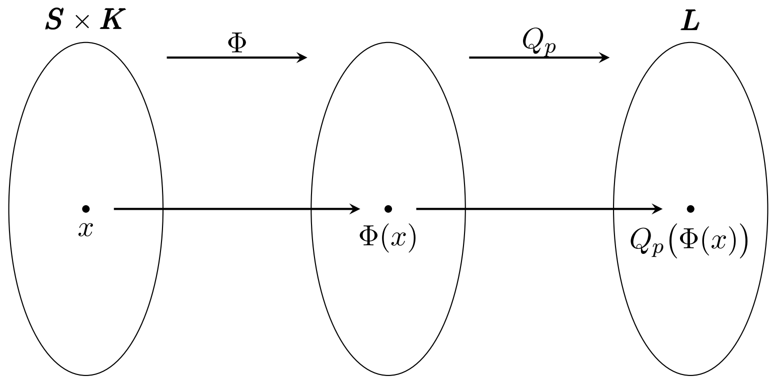
Jetzt geht es darum die oben erkläerten Ideen als informatische Datensctrukturen und *"Algorithmen"* zu übersetzen. Dennoch das Quellcode wird nicht hier "erklaert", aber im "Quellcode" Abteilung. Wie es schon erwähnt wurde, das Kern dieser Lösung ist in einer sogennante logischen Maschine: Ein "State" wird gegeben (die mögliche Kinder, die noch kein Geschenk bekommen haben bzw. ein Vektor/Kollektion dieser Form $->$ $[[k_i\ \ [a\ \ b\ \ c]]\ \ [k_j\ \ [x\ \ y\ \ z]]\ \ ...\ \ ]$ 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 $->$ $[a\ \ x\ \ ...]$. Diese Kollektion ist sehr wichtig. Dadrin sind alle erste-Wünsche und zwar so, dass z.B. $a$ auf der Position 0 ist und entspicht das erstes Wunsch von dem ersten Kind... so haben wir kind und Wünsche "verbinden". Am Ende machen wir aus dieser Kollektion ein sogennantes **Set**, so dass $[a\ \ b\ \ a\ \ ...]->\{a\ \ b\ \ ...\}$. Das machen wir um zu wissen, wie viele *verschiedene* Wünsche es gibt. Und jetzt das Punchline... diese letzte **Set** enthählt 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 $[a\ \ b\ \ a\ \ ...]$ und das Set $\{a\ \ b\ \ ...\}$, sodass $\forall_x \in \boldsymbol{S} : f(x) := \{(i_1;\ \ i_2;\ \ ...)\ /\ \boldsymbol{K}[i_j] = x\}$ wobei $\boldsymbol{K}[i_j]$ entspricht das element auf dem Index $i_j$ in der Kollektion $\boldsymbol{K}$. 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 Anordungen und ein *Parameter bzw. "Constraint"*, das die Suche regulieren könnte. Dann lautet das Ziel bzw. *Query* so: $\boldsymbol{L} := \{(l_1;\ \ l_2\ \ ...)\ /\ l_i \in \Phi[i]\}$. Die optimalste Lösung wäre dass man $\boldsymbol{L}$ 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 $\boldsymbol{Q}_p$ ein Parameter $p$, die entspricht die maximale Anzahl von Geschenke die "sich rütschen düerfen". Man fangt mit 0 an, und wenn es nicht klapp (wenn die Machine kein passendes $\boldsymbol{L}$ 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

Als erstes Beispiel wird das Lösungsprozess von *Beispieldatei wichteln1.txt* ganz genau erkärt. Erstmal lesen wir das *.txt* Dokument und erstellen wir das folgende Struktur:

$$
\begin{aligned}
([0\ \ (2\ \ 10\ \ 6)]\\
[1\ \ (2\ \ 7\ \ 3)]\\
[2\ \ (4\ \ 7\ \ 1)]\\
[3\ \ (3\ \ 4\ \ 9)]\\
[4\ \ (3\ \ 7\ \ 9)]\\
[5\ \ (4\ \ 3\ \ 2)]\\
[6\ \ (7\ \ 6\ \ 2)]\\
[7\ \ (10\ \ 2\ \ 4)]\\
[8\ \ (9\ \ 8\ \ 1)]\\
[9\ \ (4\ \ 9\ \ 6)])
\end{aligned}
$$

(a) Umgewandte Datenstruktur

$$
\boldsymbol{M}_0^{10\times 3} = \begin{pmatrix} 2 & 10 & 6 \\ 2 & 7 & 3 \\ 4 & 7 & 1 \\ 3 & 4 & 9 \\ 3 & 7 & 9 \\ 4 & 3 & 2 \\ 7 & 6 & 2 \\ 10 & 2 & 4 \\ 9 & 8 & 1 \\ 4 & 9 & 6 \end{pmatrix}
$$

(b) Abstrakte Beschreibung

Figure 2: Anfangssituation

An dieser Stelle müssen wir die erste Wünsche angucken und eine $\boldsymbol{L}$ösungsmenge daraus machen bzw. welche Geschenke zu welche Kinder als erstes Wunsch zugeteilt werden. An diesem Fall sieht es so aus ($S$ bedeutet Set und entspricht die Geschenke):

$$
\begin{aligned}
\{S\{7\ \ 4\ \ 3\ \ 2\ \ 9\ \ 10\}\\
(6\ \ 2\ \ 3\ \ 0\ \ 8\ \ 7)\}
\end{aligned}
$$

Dann müsser wir aus $\boldsymbol{M}_0^{10\times 3}$ alle $\{(M_{i(1;2;3)}; M_{j(1;2;3)}; .../(i; j; ...) \in \boldsymbol{L}\}$ substrahieren um zu finden, welche Kinder noch keine Geschenk bekommen haben.

$$
\begin{aligned}
&([1\ \ (2\ \ 7\ \ 3)]\\
&\ [4\ \ (3\ \ 7\ \ 9)]\\
&\ [5\ \ (4\ \ 3\ \ 2)]\\
&[9\ \ (4\ \ 9\ \ 6)])
\end{aligned}
\qquad\qquad
\boldsymbol{M}_1^{4\times 3} = \begin{pmatrix} 2 & 7 & 3 \\ 3 & 7 & 9 \\ 4 & 3 & 2 \\ 4 & 9 & 6 \end{pmatrix}
$$

Jetzt müssen wir entscheiden, welche Geschenke *als zweites Wunsch* zugeteilt werden. Aber in diesem Fall merkt man dass die Geschenke $7, 3$ und 9 schon zugeteilt wurden und deswegen sollten wir in dieser Runde *gar kein* Geschenk zuteilen... das ist genau das Antwort von dem Program:

$$
\begin{aligned}
&\{S\{\}\\
&\ \ nil\}
\end{aligned}
$$

Und selbsverändlich wenn man keine Geschenke zugeteilt hat, entspricht das Matrix $\boldsymbol{M}_1^{4\times 3}$ immer noch die Kinder die kein Geschenk bekommen haben. Also, brauchen wir kein weiteres Matrix zu presentieren. Dennoch jetzt werden die Geschenke definiert, die als drittes Wunsch zugeteilt werden:

$$
\begin{aligned}
&\{S\{6\}\\
&\ \ (9)\}
\end{aligned}
$$

Mit dem bloßen Augen hätte man auch merken können, dass das einzige Geschenke, die übrig geblieben ist, ist das Geschenk 6 und der einzelne Kind, das Kind an der Stelle 9 (*Das Kind 0 existiert!*). Das letztes, das man noch machen muss; ist die Kinder und Geschenke definieren; die zuällig verbunden werden:

$$
\begin{aligned}
&\{S\{1\ \ 5\ \ 8\}\\
&S\{1\ \ 4\ \ 5\}\}
\end{aligned}
$$

Und das Antwort lautet:

**"The kids who receive their first wish are:** [6 2 3 0 8 7] **those who receive their second wish are:** [] **, and the ones receiving their third/last wish are** [9] **And the kids** [1 4 5] **will receive one random gift out of the following gifts** [1 5 8]**"**

Als zweites Beispiel wird das Beispieldatei *wichteln2.txt* präsentiert. Aber diesmal ohne so viele Erklärungen um Papier zu sparen.

([0 (4 6 5)]
[1 (5 4 6)]
[2 (6 4 5)]
[3 (6 4 5)]
[4 (5 4 6)]
[5 (4 6 5)]
[6 (4 5 6)]
[7 (5 4 6)]
[8 (6 5 4)]
[9 (4 5 6)])

(a) Umgewandte Datenstruktur

$$\boldsymbol{M}_0^{10\times 3} = \begin{pmatrix} 4 & 6 & 5 \\ 5 & 4 & 6 \\ 6 & 4 & 5 \\ 6 & 4 & 5 \\ 5 & 4 & 6 \\ 4 & 6 & 5 \\ 4 & 5 & 6 \\ 5 & 4 & 6 \\ 6 & 5 & 4 \\ 4 & 5 & 6 \end{pmatrix}$$

(b) Abstrakte Beschreibung

Figure 4: Anfangssituation

Drei Kinder werden ausgewählt um ihre erstes Wunsch zu bekommen:

$$\begin{aligned}&\{S\{4\ \ 6\ \ 5\}\\ &\quad(0\ \ 2\ \ 1)\}\end{aligned}$$

Dann muss man die restliche Kinder *erkennen*:

$$\begin{array}{r}([3\ \ (6\ \ 4\ \ 5)]\\ [4\ \ (5\ \ 4\ \ 6)]\\ [5\ \ (4\ \ 6\ \ 5)]\\ [6\ \ (4\ \ 5\ \ 6)]\\ [7\ \ (5\ \ 4\ \ 6)]\\ [8\ \ (6\ \ 5\ \ 4)]\\ [9\ \ (4\ \ 5\ \ 6)])\end{array} \qquad \boldsymbol{M}_1^{7\times 3} = \begin{pmatrix} 6 & 4 & 5\\ 5 & 4 & 6\\ 4 & 6 & 5\\ 4 & 5 & 6\\ 5 & 4 & 6\\ 6 & 5 & 4\\ 4 & 5 & 6 \end{pmatrix}$$

Jetzt muss man merken, dass kein anderes Kind irgendein Wunsch kriegen könnte, weil alle gewünschte Geschenke schon zugeteilt wurden. Also, müssen wir tatsächlich kein Geschenk zuteilen:

$$\begin{aligned}&\{S\{\}\\ &\quad nil\}\end{aligned}$$

Deswegen arbeiten wir mit dem Matrix $\boldsymbol{M}_1^{7\times 3}$ weiter. Und jetzt fragen wir nach den dritte Wünsche. Nochmal ist einfach zu merken, dass kein drittes Wunsch erfüllt werden könnte. Also, wir müssen kein Geschenk zuteilen:

$$\begin{aligned}&\{S\{\}\\ &\quad nil\}\end{aligned}$$

Da wir *nur* 3 Geschenke zu 3 Kinder zugeteilt haben; sollten es 7 Kinder und auch 7 Geschenke übrig bleiben:

$$\{S\{7\ 1\ 3\ 2\ 9\ 10\ 8\}$$
$$S\{7\ 4\ 6\ 3\ 9\ 5\ 8\}\}$$

Deswegen, lautet das Antwort:

**"The kids who receive their first wish are:** [0 2 1]**,those who receive their second wish are:** []**, and the ones receiving their third/last wish are** [] **And the kids** [7 4 6 3 9 5 8] **will receive one random gift out of the following gifts** [7 1 3 2 9 10 8]**"**

* **Als Anmerkung** wollte ich auch sagen, dass *logic programming* und Ich Grenzen haben. Beispieldateien wie *wichteln7.txt* bieten ungeheuer viele Möglichkeiten, die von der logischen Engine überprüft werden müssen damit die hier erklärte Lösung funktioniert. Das macht die Laufzeit des Programm gar nicht optimal und das Programm selbst keine optimale Lösung für das verallgemeinertes Problem. Diese Grenzen zu überbinden hoffen wir als Team in der kommende Zeit zu schaffen.

## 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 $\boldsymbol{S} \times \boldsymbol{K}$.

```
(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 Φ. 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]
  (let [r (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])]
    (map #(if (empty? %1) {} %1) r)))
```

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-membero vars pool)), sodass wenn ich die Potenzmenge dieser Menge/Kollektion mir angucke; 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-membero 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).

```clojure
(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 grßer String, das aus verschiedenen Teilen besteht: Erst sind die Kinder, die sein erstes Wunsch bekommen und so und so fort.

```clojure
(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))
```