# 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

Man kann dieses Problem zu vereinfachen versuchen. Erstmal sieht so aus, als ob man musste ein Subgraph finden: Die kanten entsprechen die Möglichkeit 2 Dreiecken zu verbinden und natürlich sind die Knoten die Dreiecke. Aber das reicht nicht um ein effizientes Algorithm zu programmieren. Wenn man auf dieser Ebene geblieben wäre, dann müsste

man einfach **alle** Kombinationen bzw. Wege durchlaufen und gucken ob es einer gibt, die bestimmte Eigenschaften hat. Das klingt wie eine logische Maschine wie bei der Aufgabe 5. Dennoch glaubten wir dass es eine besser Lösung geben muss.
Vielleicht wäre es hilfreich die Suche in Teilen machen: Man kann das großes Dreieck nicht nur als eine Sache betrachten, aber viele verschiedene suzammen... ein Hexagon und 3 kleine Dreiecke mit 4 Teilen. Das wäre ein besseres Ausgangspunkt. Wir haben versucht, diese komplexe Methode zu implementieren aber unseres *amateur* Programmierensniveau und andere Sachen haben es uns leider verhindert. Core.logic ist immer noch **an entwickeln** und *constraint programming* ist ein nicht gewöhnliches Verfahren alltgliche Sachen zu programmieren (außer manche Aussnahmen, wie *Datomic queries*). Also, halten wie uns lieber daran, was machtbar ist.

## Umsetzung

Es gibt nicht so viel über die Umsetzung zu sagen. Der Leser weisst bescheid dass es um eine logische Maschine sich handelt, und deswegen werden eher die Regeln bzw. *Constraints* erklärt. Wenn man das großes Dreieck anguckt, merkt man dass es 9 *Verbindungen* gibt. Eine Verbindung wird definiert als 2 Zahlen, die zu 2 **verschiedene** Dreiecke gehören und miteinander zu eine bestimme Zahl addieren. Warum? Weil in dieser Lösung keine negative Zahlen benutzt werden dürfen (eine komische Sache von core.logic). So sind die Zahlen $-1$ und 1 eine Verbindung weil $(-1 + a) + (1 + a) = 2a = k$ wobei $a$ das erste Nummer ist, die uns gegeben wird bzw. das interval in dem die Nummern sich befinden könnten.
Die nächste Schritt ist eigentlich die Dreiecke einstufen: Es gibt 3 und nur 3 Dreiecken, die sich mit 3 anderer verbinden. Es gibt 3 und nur 3 Dreiecken, die sich mit 2 anderen Dreiecken verbinden. Und zum Schluss gibt es 3 und nur 3 Dreiecken, die sich mit *ein anderes* Dreieck verbindet. Die waren alle wichtige Eigenschaften der Lösung, die wir suchen. Jetzt zu dem Code.

## Beispiele

Um die Idee genauer zu erkären, wird ein vereinfachtes Beispiel *"gelöst"*: Es geht einfach darum, 4 Dreiecken anzuordnen.

```
(def test0 [[7 1 5] [2 4 6] [6 5 3] [-1 -3 -2]])
```

Das Code um dieses Beispiel zu lösen ist *nicht dasselbe* wie das Code, mit dem die Aufgabe tatsčhlich zu lösen versucht (in einem Fall gibt es nur 4 Teilen und in dem anderen gibt es 9... das sollte keine große Unterschied, aber bei ist doch). In der *Quellcode* Abteilung findet man das Code um diese *"subaufgabe"* zu lösen. Also, das Program bzw. die Idee funktioniert und ergibt folgendes als Antwort.

```
(comment
  {:A (14 8 12),  ;; [7 1 5]
   :B (13 12 10), ;; [6 5 3]
   :C (9 11 13),   ;; [2 4 6]
   ;; D [-1 -3 -2]
   :details {(6 4 5) {(14 8 12) {:c1 6,  ;; ... {-1 1}
                                 :x2 8},
                      (13 12 10) {:x6 10, ;; ... {3 -3}
                                  :c2 4},
                      (9 11 13) {:x7 9,   ;; ... {2 -2}
                                 :c3 5}}}})
```

Das nächstes Beispiel ist das *nächstes Niveau*. Dennoch verwenden wir hier auch ein eigenes Beispiel, weil *die Beispieldateien* zu schwer sind bzw. zu lange dauern. Das ist der Hauptgrund, warum seit ganz Anfang eine **"schlaue"** Art und Weise versucht wird; die Suche zu minimalisieren... bessere **constratints**. Alleine dieses naives Beispiel dauert 56 Sekunden zu lösen und das ist auf jeden Fall nicht optimal.

```
(def test2
  [[-1 -1 -1] [1 1 1] [-1 -1 -1]
   [-1 -1 -1] [1 1 1] [-1 -1 -1]
   [-1 -1 -1] [1 1 1] [-1 -1 -1]])

({:c [(2 2 2) (2 2 2) (2 2 2)],
  :b [(0 0 0) (0 0 0) (0 0 0)],
  :a [(0 0 0) (0 0 0) (0 0 0)],
  :details {:B1-C1 {2 0},
            :B1-C2 {2 0},
            :B2-C2 {2 0},
            :B2-C3 {2 0},
            :B3-C1 {2 0},
            :B3-C3 {2 0}}})
```

Also, direkt nach der Beendung der Lösungsidee und Umsetuzung als Code merkten wir dass es etwas gab, das nicht völlig stimmt. Deswegen wurde unser Hauptaufgabe zu **beweisen** ob dieses Verfahren/Algorithm überhaupt funktioniert oder nicht. Mit den oben gezeigte Beispielen ist es bewiesen dass das Problem nicht an dem Verfahren selbst liegt, sonder an unsere Fähigkeit und Herrschaft über das Paradigm von Logischen Programmierung.

## Quellcode

Die Funktion *test−todo* ist für das Beispiel mit 4 Klein Dreiecken gedacht; und die Funktion *start* für die wirkliche Aufgabe.

```
(defn test-todo []
  (logic/run 1 [q]
```

```
(logic/fresh [A B C D]

  (logic/permuteo [A B C D] (map #(map (fn [v] (+ v 7)) %1) test0))

  (logic/fresh [x1 x2 x3
                x4 x5 x6
                x7 x8 x9]

    (logic/permuteo [x1 x2 x3] A)
    (logic/permuteo [x4 x5 x6] B)
    (logic/permuteo [x7 x8 x9] C)

    (logic/fresh [c1 c2 c3]
      (logic/permuteo [c1 c2 c3] D)

      (fd/in x2 x6 x7
             c1 c2 c3
             (fd/interval 0 14))

      (fd/eq
       (= (+ c1 x2) 14)
       (= (+ c2 x6) 14)
       (= (+ c3 x7) 14))

      (logic/== q {:A A
                   :B B
                   :C C
```

```
                        :details {D {A {:c1 c1
                                        :x2 x2}
                                    B {:x6 x6
                                        :c2 c2}
                                    C {:x7 x7
                                        :c3 c3}}}}}}}))
```

```
(defn start
  [triangles shift]
  (let [h (* 2 shift)]
   (logic/run 1 [q]
     (logic/fresh [n1 n2 n3
                   n4 n5 n6
                   n7 n8 n9

                   C1 B1 A1
                   C2 B2 A2
                   C3 B3 A3]

       (logic/permuteo [C1 B1 A1
                        C2 B2 A2
                        C3 B3 A3]
                       (map #(map (fn [v] (+ v shift)) %1) triangles))

       (logic/membero C1 [[n1 n2 n3]
                          [n3 n1 n2]
                          [n2 n3 n1]])
       (logic/membero C2 [[n4 n5 n6]
```

```
                    [n6 n4 n5]
                    [n5 n6 n4]])
(logic/membero C3 [[n7 n8 n9]
                    [n9 n7 n8]
                    [n8 n9 n7]])

(logic/fresh [i1 i2 i3
              i4 i5 i6
              i7 i8 i9]


  (macro/symbol-macrolet [_ (logic/lvar)]
                         (logic/permuteo B1 [[i1 i4 _]
                                             [i1 i4 _]
                                             [i1 i4 _]])
                         (logic/permuteo B2 [i5 i8 _]
                                            [i1 i4 _]
                                            [i1 i4 _])
                         (logic/permuteo B3 [i2 i7 _]
                                            [i1 i4 _]
                                            [i1 i4 _])

                         (logic/permuteo A1 [i3 _ _])
                         (logic/permuteo A2 [i6 _ _])
                         (logic/permuteo A3 [i9 _ _]))
  (macroexpand-1 '(macro/symbol-macrolet [_ (logic/lvar)]
                         (logic/permuteo B1 [i1 i4 _])
```

```
                        (logic/permuteo B2 [i5 i8 _])
                        (logic/permuteo B3 [i2 i7 _])

                        (logic/permuteo A1 [i3 _ _])
                        (logic/permuteo A2 [i6 _ _])
                        (logic/permuteo A3 [i9 _ _])))

(fd/in i1 i2 i3
       i4 i5 i6
       i7 i8 i9

       n1 n2 n3
       n4 n5 n6
       n7 n8 n9

       (fd/interval 0 2))

;; B1
(logic/membero i1 B1)
(logic/membero i4 B1)
;; B2
(logic/membero i5 B2)
(logic/membero i8 B2)
;; B3
(logic/membero i3 B3)
(logic/membero i7 B3)
```

```
;; A1
(logic/membero i3 A1)
;; A2
(logic/membero i6 A2)
;; A3
(logic/membero i9 A3)

(fd/eq
 ;; B1
 (= (+ n1 i1) h) ;; - C1
 (= (+ n4 i4) h) ;; - C2
 ;; B2
 (= (+ n5 i5) h) ;; - C2
 (= (+ n8 i8) h) ;; - C3
 ;; B3
 (= (+ n3 i3) h) ;; - C1
 (= (+ n7 i7) h) ;; - C3

 ;; A1
 (= (+ n3 i3) h)
 ;; A2
 (= (+ n6 i6) h)
 ;; A3
 (= (+ n9 i9) h))

(logic/== q {:c [C1 C2 C3]
             :b [B1 B2 B3]
```

```
:a [A1 A2 A3]
:details {:B1-C1 {n1 i1}
          :B1-C2 {n4 i4}

          :B2-C2 {n5 i5}
          :B2-C3 {n8 i8}

          :B3-C1 {n3 i3}
          :B3-C3 {n7 i7}}})))))))
```

**Das ganzes Quellcode** kann man unter das GitHub Repository *BundeswettbewerbInformatik* von dem User *steve0el0crack* finden.