Aufgabe 1
Wörter aufräumen

Tassilo Tanneberger 5. Oktober 2020

Theorie & Lösungsidee

Die Grundlegende ist ziemlich simpel wir suchen alle Wörter die zu einem Lückenwort passen raus und sammeln diese in einer Liste das führen wir für jedes Lückenwort aus. Danach beginnen wir mit dem Lückenwort, was die wenigsten möglichen passenden Wörter hat. Wir führen folgenden Schritt für jedes passende Wort aus. Wir löschen das Wort einmal aus den Möglichen Wörtern anderer Lückenwörter, weil das Wort nun genutzt wird. Nun beginnt der Prozess wieder bei der Suche des Lückenwortes mit minimaler anzahl von Auswahlmöglichkeiten.

Anmerkung zur Mathematischen Formulierung: l ist ein Lückenwort aus der Menge aller Lücken Wörter L und w ist ein mögliches angegebenes Wort aus W. Die Funktion $\Psi(l, w) \in \{0, 1\}$ trifft eine Aussage darüber ob ein Wort w zu einen Lückenwort l zugeordnet werden kann.

$$A(l) = \{w_i \in W | \, \Psi(l, w_i) \; i = (1, \ldots, |W|)\}$$

$A(l)$ ... *Alle Wörter die zu einen gebenen Lückenwort l passen*

Die Funktion $\Psi(l, w)$ schaut also ob einen gegebenes Wort zu einen gebenen Lückenwort. Die verwendete Syntax für ein Lückenwort ist das ein nicht gesetzter Buchstabe mit einem "_" gekennzeichnet ist. Die Funktion hinter Ψ iteriert einfach über die Buchstaben in dem Wort und dem Lückenwort und vergleicht die Buchstaben. Wenn er ein "_" findet wird der Buchstabe aus w ignoriert. Nur wenn jeder Buchstabe aus w mit jedem Symbol aus l übereinstimmt wird 1 zurückgegeben.

$A(l)$ besitzt eine wichtige Rolle da es alle Wörter aussortiert die nicht zu einen geben Lückenwort l passen. Somit kann man Lückenwörter priorisieren abhängig davon wie viele Wörter noch vorhanden sind die zu dem Lückenwort passen ($|A(l)|$).

Nun müssen den Algorithmus nur noch Formal definieren. Ich habe den Algorithmus in zwei Funktionen gespalten die erste Funktion (Find-Min-Size(W, L)) sucht das Lückenwort heraus was die geringste Auswahl an Wörtern hat und somit als nächstes ein Wort zugewiesen bekommt. Die zweite Funktion (Recursive-Search(W, L, C)) generiert über einen Backtracking Ansatz alle möglichen Sätze.

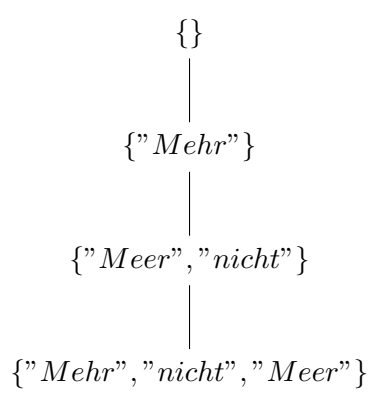
Algorithm 1 Finden möglicher Zuordnungen

```
1:  K ... Lösungs Menge
2:  function Find-Min-Size(W, L)
3:      λ ... Min Value
4:      I ... Index
5:      for i = (1, ..., |L|) do
6:          if |A(l_i)| < λ ∧ |A(l_i)| ≠ 0 then
7:              λ ← |A(l_i)|
8:              I ← i
9:          end if
10:     end for
11:     return I
12: end function
13:
14: procedure Recursive-Search(W, L, C)
15:     I ← Find-Min-Size(W, L)
16:     if |K_I| = 0 then                          ▷ Abbruch Bedingung
17:         if |C| ↔ |L| ∧ C ∉ K then
18:             K ← K ∪ {C}                        ▷ Mögliche Lösung gefunden
19:         end if
20:         return
21:     end if
22:     for w in R_I do
23:         P ← W \ {w}
24:         C ← C ∪ {w}
25:         Recursive-Search(P, L, C)              ▷ Rekursiver Neuaufruf
26:     end for
27: end procedure
```

Nehmen wir ein kleines Beispiel und führen den Algorithmus damit durch.

- Lückenwörter $L = \{"M\_h\_", "\_\_\_\_\_", "\_\_\_\_"\}$
- Wörter $W = \{"Meer", "nicht", "Mehr"\}$

Es entsteht eine Baumstruktur wir erkennen z.B auch das $|A("\_\_\_\_")| = 2$ somit es als letztes Lückenwort ein Wort zugewiesen bekommen. Eine verzweigung entsteht dann wenn der Algorithmus sich nicht sicher ist welches Wort er nun einsetzten soll bedeutet also $|A(l)| > 1$.



$A("M\_h\_") = \{"Mehr"\}$
$W = \{"Meer", "nicht", "Mehr"\}$

$A("\_\_\_\_\_") = \{"nicht"\}$
$W = \{"Meer", "nicht"\}$

$A("\_\_\_\_") = \{"Meer"\}$
$W = \{"Meer"\}$

$W = \{\}$

Weitere Ideen Für eine besonders Ambitionierte Lösung kann man auch die Funktion Ψ mit der Levenshtein-Distanz $d(l, w)$ ersetzen. Das würde den Vorteil mit sich bringen das man nicht mehr schaut ob ein gegebens Wort auf ein Lückenwort passt sondern wie ähnlich ein gegebens Wort zu den Lückenwort ist.

Implementierung

Das Program wurde in C++ umgesetzt. Wir werden in diesem Abschnitt uns ausgewählte Quellcodestücke nehmen und etwas genauer betrachten.

```
unsigned int index = 0;
unsigned int min˙value;

min˙value = std::numeric˙limits¡unsigned int¿::max();
for (unsigned int i = 0; i ¡ possible˙words.size(); i++) –
  if (min˙value ¿ possible˙words.at(i).size() and not possible˙words.at(i).empty())
      –
    min˙value = possible˙words.at(i).size();
    index = i;
  ˝
˝
```