Aufgabe 1: Wörter aufräumen

Team-ID: 00537

Team-Name: LuC++

Bearbeiter dieser Aufgabe:
Lucas Schwebler

14. November 2020

Inhaltsverzeichnis

1 Lösungsidee

Aus den zwei gegebenen Mengen (Menge der Wörter W und Menge der Lücken L) lässt sich ein bipartiter Graph $G = (W \cup L, E)$ konstruieren: Eine Kante verbindet ein Wort und eine Lücke genau dann, wenn das Wort in die Lücke passt. Gesucht ist nun ein maximales bipartites Matching.

2 Umsetzung

Um dieses bipartite Matching zu finden, lässt sich ausnutzen, dass nach Aufgabenstellung nur eine Zuordnung existiert. Allerdings lässt sich das Problem auch mit einem klassischen Algorithmus zum Bestimmen eines bipartiten Matchings lösen:

Wir versuchen nacheinander allen Wörtern eine Lücke zuzuordnen. Dazu testen wir alle möglichen Lücken (Nachbarknoten im bipartiten Graphen). Ist die Lücke frei, also noch nicht zugeordnet, haben wir eine passende Zuordnung gefunden. Sonst müssen wir das Wort, das aktuell für diese Lücke vorgesehen ist, neu zuordnen. Wir können die Funktion also Rekursiv mit dem aktuell zuzuordnenden Wort aufrufen. Damit wir dies für keine Wörter mehrfach tun (und vielleicht eine Endlosschleife verursachen), wird gespeichert, ob wir diesen Knoten (dieses Wort) bereits besucht haben:

```
def findMatch(wort):
  for lücke in mögliche_lücken[wort]:
    if not besucht[lücke]:
      besucht[lücke] = True
      if (not zuordnung[lücke]) or findMatch(zuordnung[lücke]):
        zuordnung[wort] = lücke
        return True
  return False
```

Bei der Implementation ist darauf zu achten, dass Unicodezeichen wie ä, ö, ü oder ß richtig eingelesen werden. In Python geht dies indem man die Inputdatei im utf-8 Format öffnet:

```
open("words.in", "r", encoding="utf-8")
```

Außerdem ist es wichtig, bei den Lücken Satzzeichen zu ignorieren. Da Satzzeichen in der Regel hinter dem Wort stehen und auch nur eines vorhanden ist, wird einfach überprüft, ob eine Lücke mit einem Satzzeichen endet und wenn ja, wird dieses Entfernt und als Suffix für diese Lücke gespeichert, der dann bei der Ausgabe wieder angehängt wird.

3 Beispiele

```
raetsel0.txt:

oh je, was für eine arbeit!


raetsel1.txt:

Am Anfang wurde das Universum erschaffen. Das machte viele Leute sehr wütend und wurde
    allenthalben als Schritt in die falsche Richtung angesehen.


raetsel2.txt:

Als Gregor Samsa eines Morgens aus unruhigen Träumen erwachte, fand er sich in seinem
    Bett zu einem ungeheueren Ungeziefer verwandelt.


raetsel3.txt:

Informatik ist die Wissenschaft von der systematischen Darstellung, Speicherung,
    Verarbeitung und Übertragung von Informationen, besonders der automatischen
    Verarbeitung mit Digitalrechnern.


raetsel4.txt:

Opa Jürgen blättert in einer Zeitschrift aus der Apotheke und findet ein Rätsel. Es ist
    eine Liste von Wörtern gegeben, die in die richtige Reihenfolge gebracht werden
    sollen, so dass sie eine lustige Geschichte ergeben. Leerzeichen und Satzzeichen
    sowie einige Buchstaben sind schon vorgegeben.
```

4 Quellcode

Zunächst wird der bipartite Graph aus den Inputdaten erstellt:

```
def init_adj():
    global adj
    for word in woerter:
        x = []
        for j in range(len(luecken)):
            luecke = luecken[j]
            if len(word) == len(luecke):
                valid = True
                for c in range(len(word)):
                    if luecke[c] != '_' and luecke[c] != word[c]:
                        valid = False
                if valid:
                    x.append(j)
        adj.append(x)
```

Danach wird ein Matching gefunden:

```
def find_match(i, visited, match):
    for j in adj[i]:
        if not visited[j]:
            visited[j] = True
```

```
            if match[j] == -1 or find_match(match[j], visited, match):
                match[j] = i
                return True
    return False

def find_bipartite_matching():
    n = len(adj)
    match = [-1] * n
    for i in range(n):
        visited = [False] * n
        find_match(i, visited, match)

    return match
```