Aufgabe 1: Störung

Team-ID: 00988

Team / Bearbeiter: Finn Rudolph

19. November 2022

Inhaltsverzeichnis

1 Lösungsidee

Das Problem wird durch Durchsuchen des großen Texts nach Vorkommnissen des Lückensatzes gelöst. Um das zu erleichtern, soll der Text in einem Vorverarbeitungsschritt in Wörter aufgespalten werden. So muss sich während der Suche nicht um Satz- oder Leerzeichen gekümmert werden. Auch der Lückensatz wird derartig zerlegt. Im Folgenden sei n die Anzahl an Wörtern im großen Text und m die Anzahl an Wörtern oder Lücken im Lückensatz. Mithilfe der Vorverarbeitung muss nur noch überprüft werden, ob für ein $i : 0 \leq i < n - m$ gilt, dass alle Wörter im Intervall $[i, i + m)$ im großen Text gleich den Wörtern im Lückensatz sind. Falls das Wort im Lückensatz eine Lücke ist, wird der Vergleich übersprungen. Außerdem können an Stellen mehrerer aufeinanderfolgender Lücken sofort alle gemeinsam übersprungen werden. Ein Beispiel für solch einen Fall ist *stoerung4.txt*:

```
1 ein _ _ tag
```

Wenn Index 1 (beginnend bei Index 0) erreicht wird, kann gleich zu Index 3 gesprungen werden. Bei größeren Lückensätzen kann das einen erheblichen Geschwindigkeitsvorteil bieten.

2 Laufzeitanalyse

Seien n und m wie oben definiert und ρ die durchschnittliche Wortlänge. Im Folgenden wird angenommen, dass die maximale Wortlänge innerhalb eines konstanten Faktors der durchschnittlichen Wortlänge liegt. Es werden $\Theta(n)$ Vergleiche des Lückensatzes mit einem String an Wörtern durchgeführt (für jedes $i : 1 \leq i < n - m$). Jeder dieser Vergleiche benötigt $\Omega(1)$ und $O(m \cdot \rho)$ Zeit. Die untere Schranke von $\Omega(1)$ wird erreicht, wenn sich der erste Buchstabe des ersten Worts des Lückensatzes vom ersten Buchstaben des Worts bei i unterscheidet. Dagegen benötigt das Feststellen einer vollständigen Übereinstimmung des Lückensatzes $O(m \cdot \rho)$ Zeit. Somit beträgt die Laufzeit des gesamten Programms $\Omega(n)$ und $O(n \cdot m \cdot \rho)$ Zeit.

Die Speicherkomplexität beträgt $\Theta(\rho(n + m))$, da jeder der beiden Texte in einem Format abgespeichert wird, dessen Größe sich nur um einen konstanten Faktor vom Originaltext unterscheidet.

Anmerkung: Mit Hashing könnte man den Faktor ρ aus der Laufzeit eliminieren. Bei Texten in menschlicher Sprache ist dieser jedoch nur ein kleiner konstanter Faktor (ca. 10 im Deutschen). Die Berechnung von Hashwerten (z. B. durch einen Polynomial Hash) benötigt meist teure Operationen wie Modulo und wäre deshalb nicht gewinnbringend.

3 Implementierung

Der Algorithmus wird in C++ implementiert, das Programm kann auf Linux x86-64 Systemen ausgeführt werden. Zum Kompilieren kann in der Kommandozeile im Ordner von Aufgabe 1 einfach *make aufgabe1* eingegeben werden. Das Programm liest die Eingabe aus der Standardeingabe. Die Implementierung ist in drei Teile gegliedert: Im ersten Teil (Z. 90-109 in *main*) werden der große Text und der Lückensatz eingelesen und in Wörter zerteilt. Der zweite Teil (Z. 111-143 in *main*) ist die eigentliche Suche nach Übereinstimmungen. Zuletzt werden die Ergebnisse der Suche ausgegeben (Z. 145-156 in *main*). Der große Text, in dem gesucht werden soll, ist standardmäßig *alice.txt*, kann aber als Argument in der Kommandozeile angegeben werden (Z. 97-99).

Das Einlesen des großen Texts übernimmt die Funktion *read_text*. Sie wandelt den Text in einen Vektor von Wörtern um, wobei ein Wort als Struktur *word* gespeichert wird. In diesem wird zu jedem Wort seine Zeile *l* und Stelle *w* in der Zeile abgespeichert, um bei der späteren Ausgabe von gefundenen Übereinstimmungen präzise Angaben machen zu können. Noch eine Anmerkung zur Klassifizierung nach Satzzeichen, Anführungszeichen und Wort: Normale Satzzeichen wie . oder , werden durch *is_punctuation* durch einen simplen Vergleich erkannt. Das Anführungszeichen » oder « zu erkennen ist allerdings etwas schwieriger, da es nach UTF-8-Codierung zwei Bytes einnimmt. Gelöst wird dieses Problem durch die

Funktion *is_quotation*, die zwei aufeinanderfolgende Bytes auf eine Übereinstimmung mit dem UTF-8-Code von » oder « überprüft. Nun zur Umwandlung des Texts: Für jede Ausführung der äußeren while-Schleife (Z. 50) wird eine Zeile eingelesen und verarbeitet. Für jede Zeile, gespeichert in *line*, iteriert der Iterator *it* mithilfe der zweitinnersten while-Schleife über alle ihre Zeichen (Z. 55-80). In jeder Iteration dieser while-Schleife wird zunächst ein Wort eingelesen (Z. 59-67) und anschließend ihm folgende Satz- und Leerzeichen übersprungen. Um das Ende des Worts zu erkennen, wird die Funktion *is_word* (Z. 21-25) verwendet, die 1 zurückgibt, wenn der gegebene Iterator weder zu einem Satzzeichen, noch zu einem Anführungszeichen oder Leerraum zeigt. Beim Überspringen von Satzzeichen (Z. 72-82) muss wieder darauf geachtet werden, dass Anführungszeichen zwei Bytes einnehmen.

Das Einlesen des Lückensatzes (Z. 104-116) geschieht durch eine for-Schleife, in der alle eingelesenen Wörter in eine Struktur *pattern_elem* umgewandelt werden und dem Vektor *pattern* hinzugefügt werden. Lücken werden dadurch angezeigt, dass *is_gap* auf 1 gesetzt wird. Falls direkt aufeinanderfolgende Lücken auftreten, wird kein neues *pattern_elem* hinzugefügt, sondern die Lückenlänge *gap_len* im bereits vorhandenen erhöht.

Das Finden von Übereinstimmungen geschieht mithilfe einer while-Schleife, gefundene Übereinstimmungen werden als Paar aus Zeile und Wortnummer in *matches* gespeichert. *i* ist der Anfangsindex der aktuellen Übereinstimmung in *text*. *j* ist die Länge der aktuellen Übereinstimmung, gemessen an der Anzahl Wörtern und *k* der Index des nächsten zu überprüfenden Elements in *pattern* (Z. 115). *j* kann von *k* abweichen, da mehrere Lücken als ein *pattern_elem* zusammengefasst werden können. In der while-Schleife werden drei Fälle unterschieden: Ist das aktuelle Wort im Lückensatz eine Lücke, werden die nächsten *gap_len* Wörter übersprungen (Z. 119-123). Der zweite Fall ist, dass das nächste Wort in Text und Lückensatz übereinstimmen, sodass *j* und *k* erhöht werden (Z. 124-128). Im dritten Fall müssen sich die Wörter in Text und Lückensatz unterscheiden, sodass die aktuelle Übereinstimmung abgebrochen und *i* erhöht wird (Z. 129-134). Zuletzt wird überprüft, ob die aktuelle Übereinstimmung vollständig ist, sodass sie zu *matches* hinzugefügt werden kann (Z. 136-142).

Im letzten Teil des Programms werden die Ergebnisse der Suche ausgegeben. Die Angabe von Zeile und Stelle des ersten Worts bei einer Übereinstimmung ist deshalb sinnvoll, da Bücher (und Textdateien) üblicherweise nach Zeilen gegliedert sind und so das Auffinden der Stelle besonders einfach wird.

4 Beispiele

Die Richtigkeit der Ausgabe wurde bei allen gezeigten Beispielen mithilfe der Suchfunktion eines Texteditors überprüft. Dazu wurde nach einem im Lückensatz vorkommenden Wort gesucht und alle Vorkommnisse dessen angesehen, um Übereinstimmungen zu finden. Alice im Wunderland beinhaltet ca. 130 000 Zeichen ohne Leerzeichen. Um das Programm noch an einem anderen Text zu testen und zu sehen, wie es sich bei größeren Texten verhält, sind auch Beispiele mit Kants *Kritik der reinen Vernunft* als Suchtext (ca. 1 060 000 Zeichen) angeführt. Indem *kritik.txt* als Kommandozeilenargument angegeben wird, kann diese als Suchtext verwendet werden.

4.1 Alice im Wunderland

4.1.1 stoerung0.txt

```
das _ mir _ _ _ vor
```

```
1 Übereinstimmung(en) (Zeile, Wort):
440, 2
```

4.1.2 stoerung1.txt

```
ich muß _ clara _
```

```
2 Übereinstimmung(en) (Zeile, Wort):
425, 2
441, 10
```

4.1.3 stoerung2.txt

```
fressen _ gern _
```

```
3 Übereinstimmung(en) (Zeile, Wort):
213, 10
214, 3
214, 7
```

4.1.4 stoerung3.txt

```
das _ fing _
```

```
2 Übereinstimmung(en) (Zeile, Wort):
2319, 4
3301, 10
```

4.1.5 stoerung4.txt

```
ein _ _ tag
```

```
1 Übereinstimmung(en) (Zeile, Wort):
2293, 6
```

4.1.6 stoerung5.txt

```
wollen _ so _ sein
```

```
1 Übereinstimmung(en) (Zeile, Wort):
2185, 2
```

4.1.7 stoerung6.txt

```
_
```

```
25554 Übereinstimmungen (Zeile, Wort):
1, 1
1, 2
1, 3
3, 1
3, 2
5, 1
7, 1
...
3684, 10
3684, 11
3685, 1
3685, 2
```

Dieses Beispiel wurde als Extremfall hinzugefügt, wegen der großen Zahl an Übereinstimmungen können nicht alle abgedruckt werden.

4.2 Kritik der reinen Vernunft

Wie zu erwarten, zeigte sich aufgrund der großen Zeichenanzahl eine leicht erhöhte Laufzeit im Vergleich zu Alice im Wunderland. Während alle Beispiele von Alice im Wunderland nach maximal ca. 80 ms terminierten, lag die Grenze bei diesen Beispielen bei ca. 300 ms (Prozessor: AMD Ryzen 3700U Mobile).

4.2.1 kritik0.txt

```
formula _ _ 0
```

```
1 Übereinstimmung (Zeile, Wort):
8237, 7
```

Dieses Beispiel wurde gewählt, um zu überprüfen, ob das Programm auch Lückensätze mit Zahlen finden kann. Die gesamte Stelle lautet *"formula 3 - 3 = 0"*. Da = und - als Satzzeichen behandelt werden, sind zwei Lücken nötig.

4.2.2 kritik1.txt

```
human _ _ _ _ that
```

```
2 Übereinstimmungen (Zeile, Wort):
1971, 7
12029, 4
```

Dieses Beispiel testet den Fall von mehreren aufeinanderfolgenden Lücken.

4.2.3 kritik2.txt

```
mathematics _ _ _ _ _ _ space _ _ dimensions
```

```
1 Übereinstimmung (Zeile, Wort):
7776, 6
```

4.2.4 kritik3.txt

```
i _ think
```

```
13 Übereinstimmungen (Zeile, Wort):
744, 2
756, 7
1370, 2
4024, 4
4027, 6
4032, 7
4503, 7
4815, 9
5064, 9
8572, 11
8617, 11
8657, 7
10639, 5
```

Mit diesem Beispiel wurde überprüft, ob das Programm wirklich alle Vorkommnisse eines Lückensatzes findet. Durch Überprüfen aller Vorkommnisse von *"think"* mithilfe der Suchfunktion kann die Ausgabe bestätigt werden.

4.2.5 kritik4.txt

```
semblance _ _ hypothesis _ _ _ _ _ _ another occasion _ _ _ _ _ _ _ _ seem that _ _ _ _ _ _ _ _ _ _ _ opinion
```

```
1 Übereinstimmung (Zeile, Wort):
225, 5
```

Dieses Beispiel testet mit einem sehr langen Lückensatz einen Extremfall.

5 Quellcode

```cpp
#include <vector>
#include <string>
#include <iostream>
#include <fstream>

inline bool is_punctuation(char c)
{
    return c == '.' || c == ',' || c == ';' || c == '(' || c == ')' ||
           c == '[' || c == ']' || c == '{' || c == '}' || c == '"' ||
           c == '=' || c == ':' || c == '-' || c == '_' || c == '!' ||
```

```
            c == '?' || c == '$' || c == '*' || c == '\'';
}

// Ob der gegebene Iterator zu einem » oder « Zeichen (UTF-8) zeigt.
inline bool is_quotation(std::string const &s, std::string::iterator const &it)
{
    return s.end() - it >= 2 && *it == (char)0xc2 &&
            (*(it + 1) == (char)0xab || *(it + 1) == (char)0xbb);
}

inline bool is_word(std::string const &s, std::string::iterator const &it)
{
    return !is_punctuation(*it) && !is_quotation(s, it) &&
            *it != ' ' && *it != '\n';
}

struct word
{
    std::string s;
    size_t l, w;
};

// Beinhaltet entweder ein Wort oder eine Anzahl aufeinanderfolgender Lücken.
struct pattern_elem
{
    std::string s;
    bool is_gap;
    size_t gap_len;
};

std::vector<word> read_text(std::string const &fname)
{
    std::ifstream fin(fname);
    fin.tie(0);

    std::vector<word> text;
    std::string line;
    size_t l = 1;

    while (!fin.eof())
    {
        std::getline(fin, line);
        size_t w = 1;

        auto it = line.begin();
        while (it != line.end())
        {
```

```
58
59              std::string s;
60              while (it != line.end() && is_word(line, it))
61              {
62                  // Wandle alle Buchstaben in Kleinbuchstaben um.
63                  if (*it >= 'A' && *it <= 'Z')
64                      *it = (*it - 'A') + 'a';
65                  s.push_back(*it);
66                  it++;
67              }
68
69              if (!s.empty())
70                  text.push_back({s, l, w});
71
72              // Überspringe Satz- und Leerzeichen.
73              while (it != line.end() && !is_word(line, it))
74              {
75                  if (is_quotation(line, it))
76                      it++;
77                  it++;
78              }
79              w++;
80          }
81
82          l++;
83      }
84
85      return text;
86  }
87
88  int main(int argc, char **argv)
89  {
90      std::string fname = "alice.txt";
91      if (argc >= 2)
92          fname = argv[1];
93
94      std::vector<word> text = read_text(fname);
95      size_t n = text.size();
96
97      std::vector<pattern_elem> pattern;
98      size_t m = 0;
99
100     while (std::cin.peek() != EOF)
101     {
102         std::string s;
103         std::cin >> s;
104         if (pattern.empty() || !pattern.back().is_gap || s[0] != '_')
```

```
            pattern.push_back({s, s[0] == '_', s[0] == '_'});
        else
            pattern.back().gap_len++;
        m++;
    }

    std::vector<std::pair<size_t, size_t>> matches;

    // i, j: Indizes im Text (Wörter im Intervall [i, i + j) stimmen überein)
    // k: Index im Lückensatz
    size_t i = 0, j = 0, k = 0;

    while (i + j < n)
    {
        if (pattern[k].is_gap)
        {
            j += pattern[k].gap_len;
            k++;
        }
        else if (text[i + j].s == pattern[k].s)
        {
            j++;
            k++;
        }
        else
        {
            j = 0;
            k = 0;
            i++;
        }

        if (j == m && i + j < n)
        {
            matches.push_back(std::make_pair(text[i].l, text[i].w));
            j = 0;
            k = 0;
            i++;
        }
    }

    if (matches.empty())
        std::cout << "Lückensatz konnte nicht gefunden werden.\n";
    else
    {
        std::cout << matches.size();
        if (matches.size() == 1)
            std::cout << " Übereinstimmung (Zeile, Wort):\n";
```

```
152         else
153             std::cout << " Übereinstimmungen (Zeile, Wort):\n";
154         for (auto const &[l, w] : matches)
155             std::cout << l << ", " << w << '\n';
156     }
157 }
```

6 Quellen

- Kant, Immanuel: *Kritik der reinen Vernunft*. http://www.textfiles.com/etext/AUTHORS/KANT/c_of_p_r.txt