Lukas Kesch
lukas.kesch@gmail.com
Teilnahme-ID: 57621
Team-ID: 00579

Aufgabe 1: Wörter aufräumen

39. Bundeswettbewerb für Informatik - Runde 1 (23.11.2020)

1. Aufgabenstellung

Gegeben ist eine spezielle Art von Lückentext. Zu jedem Wort in diesem ist die Länge des Wortes und eventuell ein Buchstabe mit dessen Index bekannt. Des Weiteren ist eine Liste von Wörtern vorhanden, die den Lückentext eindeutig lösen. Ziel ist es ein Programm zu entwickeln, das solche Lückentexte löst.

2. Lösungsidee

Um ein korrektes Ergebnis garantieren zu können, müssen alle Wortreihenfolgen in Betracht gezogen und mit dem Lückentext abgeglichen werden. Dies wird mit einem rekursiv operierenden Backtracking-Algorithmus verwirklicht. Um eine schnellere Berechnung der Lösungswortreihenfolge zu ermöglichen, werden zwei gültige Einschränkungen getroffen.

Erstens werden bereits vor dem Ablauf des Algorithmus für jede Wortlücke alle passenden Wörter abgespeichert. Der rekursive Algorithmus darf dann nur diese in die Lücken setzen. Damit werden bereits vor der eigentlichen Berechnung sehr viele ungültige Wortreihenfolgen ausgeschlossen.

Desweiteren ist es oftmals der Fall, dass ein Wort zu mehr als einer Lücke passt. Deshalb wird innerhalb des rekursiven Algorithmus ein Pruning-Verfahren angewandt. Sobald ein Wort bereits in der potenziellen Lösung vorkommt, darf dieses nicht erneut verwendet werden. Müsste es erneut gesetzt werden, so wird der aktuelle Lösungszweig verworfen.

3. Umsetzung

Das Programm wurde in C++ geschrieben und mit dem MSVC Compiler kompiliert. Im Verlauf des Programms wird von der *Standard Libary* ausgiebig gebrauch gemacht, um die Übersichtkichkeit zu bewahren.

Damit sich der Buchhaltungsaufwand in Grenzen hält, werden zwei Klassen erstellt und verwendet. Ein Objekt der erste Klasse *given_word* repräsentiert jeweil ein Wort in der Wörterliste. Es speichert dessen eigentliches Wort als string, die Wortlänge als int und eine boolean Variable, die angibt, ob das Wort bereits verwendet wurde. Ein Objekt der zweiten Klasse *unkown_word* repräsentiert ein Wort in dem Lückentext. Es speichert dessen Länge, eine Liste an Zeigern auf aller für die Lücke zulässigen *given_word* Objekte, ein Zeiger auf den Index und den gegebenen Buchstaben, falls einer vorhanden ist, und eventuell vorkommende Satzzeichen nach dem Wort.

Das Programm beginnt, indem die Beispieldatei eingelesen wird. Währenddessen werden die benötigten *given_word* und *unkown_word* Objekte konstruiert und in zwei Listen eingetragen. Im Anschluss werden die Listen der Wortlänge nach sortiert, um darauf laufzeitschonend die Liste eines jeden *unkown_word* Objektes zu erstellen, in der auf die möglichen *given_word* Objekte verwiesen wird. Damit sind die Vorbereitungen für den Lösungsalgorithmus abgeschlossen.

Der rekursive Algorithmus beginnt damit, in die erste Lücke ein passendes Wort zu setzen. Im Anschluss wird in die zweite Lücke ein Wort gesetzt. Dabei wird überprüft, ob das zu setzende Wort bereits in der aktuellen Zusammensetzung vorkommt. Sollte dies der Fall sein, so wird ein anderes passende Wort, welches ebenfalls noch nicht verwendet wurde, in die Lücke gesetzt. Dieser Vorgang wird von Lücke zu Lücke wiederholt, bis eine Lücken nicht besetzt werden kann, da alle theoretisch in die Lücke passenden Wörter bereits verwendet werden. Nun wird Lücke für Lücke zurückgegangen, bis in eine Lücke ein weiteres passendes und noch nicht verwendetes Wort gesetzt werden kann. Dann wird wieder Lücke für Lücke befüllt.

Der Algorithmus terminiert, sobald alle Möglichkeiten durchprobiert wurden. Da die Aufgabenstellung aussagt, dass die Wörter den Lückentext eindeutig lösen, wird immer genau ein Ergebnis gefunden. Diese richtige Wortreihenfolge wird im Anschluss an den Benutzer ausgegeben.

4. Beispiele

4.1 Beispiel 0

file: examples/raetsel0.txt

oh je, was für eine arbeit!

4.2 Beispiel 1

file: examples/raetsel1.txt

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

4.3 Beispiel 2

file: examples/raetsel2.txt

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

4.4 Beispiel 3

file: examples/raetsel3.txt

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

4.5 Beispiel 4

file: examples/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.

4.6 Auswertung

Es stellt sich heraus, dass alle Lückentexte, wie in der Aufgabenstellung angegeben, gelöst werden können und laufzeittechnisch überhaupt kein Problem darstellen. Die durchschnittliche Laufzeit beträgt 0,32 Milisekunden mit einer Standardabweichung von 0,46 Milisekunden.

5. Laufzeitüberlegung

Da die gesuchten Wörter der Länge nach sortiert wurden und von der solve-Methode in dieser Reihenfolge abgearbeitet werden, ist das Backtracking faktorisiert. Es findet also für jede Wortlänge ein eigenes, von den anderen Längen unabhängiges Backtracking statt, was die Laufzeit drastisch reduziert von n! auf k * (n/k)! mit einem geeigneten k (irgendwo zwischen 5 und 10). Ohne diese Faktorisierung wäre die Laufzeit für längere Texte nicht mehr akzeptabel. Durch die vorausgehende einmalige Ermittlung der möglichen Wörter, die zu den vorgegebenen Buchstaben passen, wird n/k nochmals drastisch reduziert, so dass man für typische Texte k * m! erhält mit einem sehr kleinen m (im niedrigen einstelligen Bereich).

6. Denkbare Erweiterung

Das Programm kann momentan für eine Wortlücke nur einen gegebenen Buchstaben mit dessen Index in der Lücke speichern. Dies genügt für alle Beispiele. Um aber Beispiele anderer Arten zu lösen, müssten Listen verwendet werden, sodass mehrere gegebene Buchstaben abgespeichert werden können.

7. Quellcode

(siehe Anhang)

```h
#pragma once

#include <string>
#include <vector>
#include <iostream>
#include <sstream>
#include <fstream>
#include <algorithm>
#include <chrono>  // for high_resolution_clock
#include "given_word.h"
#include "unkown_word.h"

using namespace std;

//Variables
auto start_time = std::chrono::high_resolution_clock::now();
auto finish_time = std::chrono::high_resolution_clock::now();
const int number_of_tests = 5;
vector<given_word> given_words;
vector<unkown_word> unkown_words;
vector<wstring> solution;

//Methods
void print_user_greetings();
void read_input(ifstream& input_file_stream);
void find_possible_words();
void solve(int number);
bool check_solution();
void save_solution();
void print_solution();
void cleanup();
int main();

bool valid_index(int index, pair<int, int>* index_given_word_length);

```

```cpp
#include "BwInf39Runde1Aufgabe1.h"

void print_user_greetings() //O(1)
{
    cout << "Hello there!" << endl;
    cout << "This programm will loop through all the given test cases and print their solution." << endl;
}

void read_input(ifstream& input_file_stream) //O(n lg n)
{
    int number = 0;
    string line, word;

    getline(input_file_stream, line);
    stringstream input_stream_line_1(line);
    while (input_stream_line_1 >> word)
    {
        unkown_words.push_back(unkown_word(word, number++));
        solution.push_back(L"");
    }

    sort(unkown_words.begin(), unkown_words.end());

    getline(input_file_stream, line);
    stringstream input_stream_line_2(line);
    while (input_stream_line_2 >> word)
    {
        given_words.push_back(given_word(word));
    }
    sort(given_words.begin(), given_words.end());
}


void find_possible_words() //Worst: O(n^2)  Expected: O(n)
{
    int current_index = 0;
    pair<int, int> index_given_word_length;
    index_given_word_length.first = current_index;
    index_given_word_length.second = given_words[current_index].get_length();

    for (int index1 = 0; index1 < given_words.size(); index1++)
    {
        given_word* given_word_object = &(given_words[index1]);

        bool change_in_word_length = index_given_word_length.second != (*given_word_object).get_length();
        if (change_in_word_length)
        {
            index_given_word_length.first = index1;
            index_given_word_length.second = (*given_word_object).get_length();
        }

        int index2 = index_given_word_length.first;
        for (index2; valid_index(index2, &index_given_word_length); index2++)
```

```
54          {
55              unkown_word* unkown_word_object = &(unkown_words[index2]);
56
57              char given_char = (*unkown_word_object).get_given_char();
58              int index_given_char = (*unkown_word_object).get_index_given_char
                  ();
59
60              string word = (*given_word_object).get_word();
61
62              bool no_given_char = given_char == ' ';
63              if (no_given_char)
64              {
65                  unkown_words[index2].possible_words.push_back
                      (given_word_object);
66                  continue;
67              }
68
69              bool char_matches = given_char == word[index_given_char];
70              bool is_umlaut = given_char == '$';
71
72              if (char_matches && is_umlaut)
73              {
74                  int umlaut_code1 = (*unkown_word_object).get_umlaut_code
                      (index_given_char);
75                  int umlaut_code2 = (*given_word_object).get_umlaut_code
                      (index_given_char);
76                  bool umlaut_matches = umlaut_code1 == umlaut_code2;
77
78                  if (umlaut_matches)
79                  {
80                      unkown_words[index2].possible_words.push_back
                          (given_word_object);
81                  }
82              }
83              else if (char_matches)
84              {
85                  unkown_words[index2].possible_words.push_back
                      (given_word_object);
86              }
87          }
88      }
89  }
90
91  bool valid_index(int index, pair<int, int>* index_given_word_length) //O(1)
92  {
93      bool out_of_bounce = index >= unkown_words.size();
94      if (out_of_bounce)
95      {
96          return false;
97      }
98
99      bool change_in_word_length = (*index_given_word_length).second !=
          unkown_words[index].get_length();
100     if (change_in_word_length)
101     {
102         return false;
```

```
103     }
104
105     return true;
106 }
107
108 void solve(int number) //O(n!)
109 {
110     bool reached_end = number >= unkown_words.size();
111     if (reached_end && check_solution())
112     {
113         save_solution();
114         return;
115     }
116     else if (reached_end)
117     {
118         return;
119     }
120
121     unkown_word* word = &unkown_words[number];
122
123     for (int i = 0; i < (*word).possible_words.size(); i++)
124     {
125         given_word* possible_word = (*word).possible_words[i];
126
127         bool already_assigned = (*possible_word).assigned;
128
129         if (already_assigned)
130         {
131             continue;
132         }
133         else
134         {
135             (*possible_word).assigned = true;
136             (*word).solution = possible_word;
137             solve(number + 1);
138             (*word).solution = nullptr;
139             (*possible_word).assigned = false;
140         }
141     }
142 }
143
144 bool check_solution() //O(n)
145 {
146     for (given_word& word : given_words)
147     {
148         bool not_used = !(word.assigned);
149         if (not_used)
150         {
151             return false;
152         }
153     }
154     return true;
155 }
156
157 void save_solution() //O(n)
158 {
```

```
159     for (unkown_word& word : unkown_words)
160     {
161         int index_in_solution = word.index_in_output;
162         solution[index_in_solution] = word.print();
163     }
164 }
165
166 void print_solution() //O(n)
167 {
168     wstringstream output;
169     for (int i = 0; i < solution.size(); i++)
170     {
171         output << solution[i] << L" ";
172     }
173     wcout << output.str() << endl;
174 }
175
176 void cleanup() //O(n)
177 {
178     given_words.clear();
179     unkown_words.clear();
180     solution.clear();
181 }
182
183 int main()
184 {
185     print_user_greetings();
186     unkown_word::preparation();
187
188     for (int i = 0; i < number_of_tests; i++)
189     {
190         string file_name = "examples/raetsel";
191         file_name.append(to_string(i));
192         file_name.append(".txt");
193         cout << endl << "file: " << file_name << endl;
194         ifstream input_file_stream(file_name);
195
196         read_input(input_file_stream);
197         find_possible_words();
198         solve(0);
199         print_solution();
200         cleanup();
201     }
202
203     string dummy;
204     cin >> dummy;
205
206     return 0;
207 }
```