praxisphase – ss 2019

nico pistel

Bericht

PRAXISPHASENBERICHT

Classcon Consulting GmbH

D.VELOP AG

Nico Pistel

Sommersemester 2019

WESTFÄLISCHE HOCHSCHULE
WIRTSCHAFT UND INFORMATIONSTECHNIK

ABSTRACT

Nope.

INHALTSVERZEICHNIS

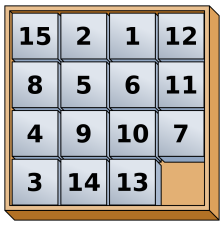
ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

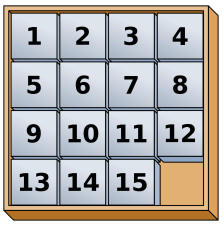
LISTINGS

1 EINLEITUNG

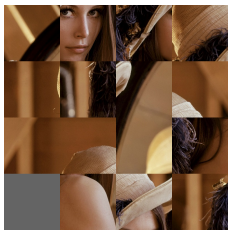
Schiebepuzzles wie das 15-Puzzle haben zum Ende des 19. Jahrhunderts das Interesse vieler amerikanischer Puzzle-Enthusiasten geweckt[2, 1] Heutzutage gibt es solche Schiebepuzzles in vielen Varianten. Dazu gehört die weitverbreitete Variante, bei der es nicht das Ziel ist, Zahlen aufsteigend zu sortieren, sondern bei dem das Puzzle aus einem Bild besteht, welches in quadratische Stücke aufgeteilt und durcheinander gemischt wurde. Das Originalbild lässt sich erst dann komplett erkennen, nachdem die Puzzlestücke in die richtige Reihenfolge gebracht wurden.



(a) 15-Puzzle mit Zahlen (gemischt)



(b) 15-Puzzle mit Zahlen (gelöst)



(c) 15-Puzzle mit Bild (gemischt)

(d) 15-Puzzle mit Bild (gelöst)

Abbildung 1.1: Zwei Arten des 15-Puzzle

In der Informatik ist das Lösen von Schiebepuzzles ein klassisches Problem der künstlichen Intelligenz und ein übliches Beispielproblem für die Modellierung und Illustration von Suchalgorithmen. Dabei beschränkt sich die meiste Literatur auf das Lösen des klassischen Schiebepuzzles mit Zahlen.

Um diese Lösungsverfahren auf ein Schiebepuzzle mit Bild zu übertragen, muss also zuvor seperat das Originalbild rekonstruiert werden.

In dieser Arbeit wird genau solch ein zweischrittiger Ansatz erläutert und analyisert.

1.1 PROBLEMSTELLUNG UND ZIELSETZUNG

In dieser Arbeit wird das Problem anhand des allgemeinen N-Puzzles betrachtet mit $N = m \times n - 1$ (für das 15-Puzzle gilt somit $m = n = 4$). Die N Puzzlestücke sind dabei rechteckige Teile eines Bildes. Diese müssen nicht zwingend quadratisch sein, sollten aber alle das gleiche Seitenverhältnis aufweisen, damit diese als Teil eines Schiebepuzzles auch wirklich alle verschiebbar sind und sich somit in den Zustand eines Schiebepuzzles zusammensetzen lassen.

m: Anzahl der Zeilen
n: Anzahl der Spalten

Die Position des leeren Feldes ist dabei nicht zwingend fest (so wie in 1.1 z. B. immer unten-rechts) und kann variieren. Diese Position ist außerdem unbekannt und muss damit anhand der Zusammensetzung der restlichen N Puzzleteilen und den bekannten Dimensionen des Puzzles ausgemacht werden.

Außerdem wird noch eine $m \times n$ Anordnung der N Puzzlestücke (und dem leeren Feld) als Anfangszustand vorgegeben.

Folgende Fragen sollen dann (in dieser Reihenfolge) beantwortet werden:

1. Wie sah das originale Bild aus und wo befindet sich das leere Feld im Ausgangsbild (wo fehlt also ein Stück des Bildes)?

2. Ist das Puzzle lösbar, lässt sich also der gegebene Anfangszustand unter einer endlichen Sequenz von legalen Zügen (also ausschließlich durch das hin- und herschieben von Puzzlestücken) in den zuvor ermittelten Endzustand transformieren?

3. Wie viele Schritte (Verschiebungen) sind mindestens nötig, um das Puzzle zu lösen und wie sieht ein optimaler Lösungsweg (ein Lösungsweg mit der kleinstmöglichen Anzahl an Verschiebungen) aus?

Dazu wird ein vollautomatischer Schiebepuzzlelöser programmiert, der genau diese Schritte abarbeitet und am Ende bei einer gefundenen (optimalen) Lösung das Ergebnis dem Benutzer präsentiert.

Programmiert wird der Puzzlelöser in der Programmiersprache C++. Zur Analyse und Verarbeitung des Bildes wird die Open-Source Computer-Vision Bibliothek OpenCV verwendet.

1.2 AUFBAU DER ARBEIT

Zunächst werden einige Grundlagen angesprochen, die den Umgang mit OpenCV erläutern und unter anderem aufklären, wie Bilder in OpenCV dargestellt und bearbeitet werden können.

Da zur Rekonstruktion Informationen wie die Kompatibilität (oder auch Ähnlichkeit) zweier Puzzlestücke notwendig ist, werden daraufhin einige Metriken vorgestellt, die diese Information repräsentieren.

Daraufhin wird die Rekonstruktion des Bildes weiter erläutert und ein Greedy Verfahren vorgestellt, welches dieses Problem mithilfe der zuvor berechneten Metriken löst.

Im nächsten Schritt wird das Puzzle mit seinem Anfangs- und Endzustand in ein äquivalentes Schiebepuzzle mit Zahlen übersetzt. Damit werden die nächsten Schritte anhand dem klassischen Zahlenpuzzle (jedoch weiterhin mit beliebigen Dimensionen und beliebigem Anfangs- und Endzustand) gezeigt.

Es wird dann erläutert, wie sich die Lösbarkeit eines Schiebepuzzles bestimmen lässt. Zusätzlich wird geklärt, wie sich zufällige Schiebepuzzle so generieren lassen, dass diese wahlweise immer lösbar oder immer unlösbar sind.

Zum Schluss werden Suchalgorithmen aus der künstlichen Intelligenz vorgestellt, die das eigentliche Lösen des Puzzles übernehmen. Dazu werden sowohl uninformierte Suchalgorithmen, als auch informierte Suchalgorithmen (welche Heuristiken in ihre Suche mit einbeziehen) untersucht und verglichen. An dieser Stelle werden somit auch verschiedene Heuristiken für Schiebepuzzle vorgestellt.

2

GRUNDLAGEN

Help

3 PROGRAMMIERUNG

Um sich mit der AWS API vertraut zu machen, wurde zunächst eine Windows-Forms Anwendung entwickelt, die synchrone Texterkennungs- und auch Dokumentanalyse-Anfragen an Amazon stellt. Damit werden zunächst nur Bilder unterstützt. Das Bild wird in der Anwendung dargestellt und die Ergebnisse von Amazon (in Form von Block-Objekten) werden in entsprechenden Tabellen aufgelistet. Eine Interaktion mit einem Block-Objekt (z. B. das Markieren einer Zelle einer Tabelle) führt dazu, dass der Inhalt des entsprechenden Blockes im Bild farblich markiert wird.

3.1 SYNCHRONE TEXTERKENNUNG

Die Dateien, die synchron mit Amazon Textract analysiert werden sollen, können entweder in Amazon S3-Buckets abgelegt werden und dann mit dem entsprechenden Namen in der Anfrage referenziert werden oder man überträgt ein Bild als Base64-codiertes Byte-Array. Dieses Byte-Array kann dann direkt als Eingabeparameter der Anfrage mitgegeben werden. Da zunächst nur die Funktionsweise des Dienstes getestet werden sollte, werden die Dokumente nicht nach Amazon S3 hochgeladen, sondern einfach als Byte-Array nach Amazon gesendet.

Als Anfrage wird `AnalyzeDocument` verwendet, womit neben dem reinen Text (Zeilen und Wörter) auch noch Formulardaten (Key-Value-Paare) und Tabellen (mit ihren Zellen) als Antwort zurückgegeben werden (dazu wird beim `AnalyzeDocumentRequest` als `FeatureTypes` `FORMS` und `TABLES` angegeben).

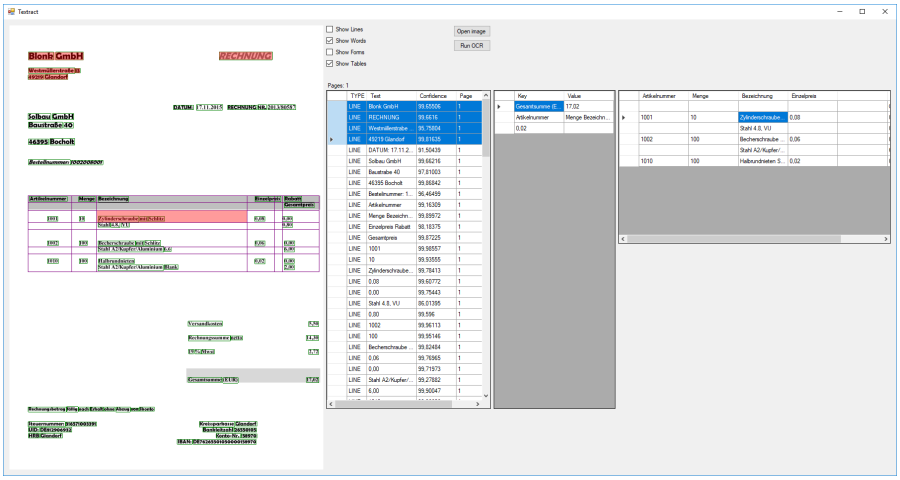


Abbildung 3.1: Auswertung eines Dokumentes mit Textract API

Die Abbildung zeigt die Auswertung eines Dokumentes in der selbst programmierten Textract GUI. Die AABBs der vier verschiedenen Blocktypen (Zeilen, Wörter, Formulardaten und Tabellen) lassen sich per Checkboxen anzeigen. Um die Geometrie einzelner markierter Blöcke hervorheben zu können, werden die Blöcke bei der Auswertung der OCR-Antwort in ein `Dictionary` gespeichert (so wird in C# eine generische `HashMap` bezeichnet), welches die Block-Id (ein String) auf das entsprechende Block-Objekt abbildet. Die jeweilige Block-Id eines Objektes wird in der GUI versteckt hinterlegt (als Eigenschaft einer Zeile oder Zelle der Tabellen). Damit lässt sich das Objekt bei Markierung in durchschnittlich konstanter Zeit direkt farbig hervorheben.

3.2 ASYNCHRONE TEXTERKENNUNG

Da der zu programmierende Dienst natürlich auch mehrseitige Dokumente unterstützen soll, wurde sich noch mit der asynchronen Texterkennung beschäftigt. Dazu wurde zunächst die eigene GUI soweit erweitert, dass diese asynchrone Anfragen an Amazon stellt.

Da nicht alle Dokumente der Classcon Consulting GmbH im PDF-Format sind (das einzige mehrseitige Format, was von Amazon Textract unterstützt wird), sondern viele im TIF-Format sind, wurde zunächst eine Methode geschrieben, welche TIF-Dateien in PDF-Dateien umwandelt.

Dokumente, die asynchron verarbeitet werden sollen, müssen in Amazon S3-Buckets liegen. Damit muss jedes Dokument vor der Texterkennung in einen solchen Bucket hochgeladen werden und am Ende sollte das Dokument auch wieder aus dem Bucket entfernt werden. Die asynchronen Klassen der Textract API weisen den Präfix `Start` auf, somit ist der asynchrone Request auf eine Texterkennung mit der Klasse `StartDocumentTextDetectionRequest` möglich. Sobald dieser Request fertig abgearbeitet wurde von Amazon, wird eine entsprechende Nachricht in ein (zuvor angelegtes) Amazon SNS Topic bekannt gegeben. Dieses Topic lässt sich mit einer Amazon SQS Queue überwachen (die Queue *abonniert* das Topic). Die Nachricht lässt sich dann aus der Queue herauslesen. Bei einer erfolgreichen Verarbeitung findet sich in

den Metadaten dann eine eindeutige `JobId`, welche als Parameter für einen `GetDocumentTextDetectionRequest` verwendet wird. Damit lässt sich das eigentliche Ergebnis der Texterkennung abfragen.

Folgender Ausschnitt an C# Code zeigt die grobe Vorgehensweise bei der asynchronen Texterkennung mit Amazon Textract.

```
using (var textractClient = new AmazonTextractClient(
    ACCESS_KEY_ID,
    SECRET_ACCESS_KEY,
    RegionEndpoint.EUWest1
))
using (var sqsClient = new AmazonSQSClient(
    ACCESS_KEY_ID,
    SECRET_ACCESS_KEY,
    RegionEndpoint.EUWest1
))
{
    var startResp = textractClient.StartDocumentTextDetection(
        new StartDocumentTextDetectionRequest {
            DocumentLocation = new DocumentLocation {
                S3Object = new S3Object {
                    Bucket = BUCKET_NAME,
                    Name = fileToOCR
                }
            }, NotificationChannel = new NotificationChannel {
                RoleArn = ROLE_ARN,
                SNSTopicArn = TOPIC_ARN
            }
        }
    );

    bool jobFound = false;
    do
    {
        foreach (var msg in
            sqsClient.ReceiveMessage(QUEUE_URL).Messages)
        {
            var body =
                JsonConvert.DeserializeObject<dynamic>(msg.Body);
            var message =
                JsonConvert.DeserializeObject<dynamic>(body.Message);
            var jobId = message.JobId;
            var status = message.Status;

            if (jobId == startResp.JobId)
            {
                jobFound = true;

                if (status == "SUCCEEDED")
                {
                    string token = null;
                    do
                    {
                        var getResp =
                            textractClient.GetDocumentTextDetection(
```

```
                        new GetDocumentTextDetectionRequest {
                            JobId = jobId,
                            NextToken = token
                        }
                    );

                    foreach (var block in getResp.Blocks)
                    {
                        // Process block
                    }

                    token = getResp.NextToken;
                } while (token != null);
            }

            sqsClient.DeleteMessage(QUEUE_URL,
                msg.ReceiptHandle);
            break;
        }
    }
} while (!jobFound);
}
```

Listing 3.1: Asynchrone Texterkennung in Amazon Textract

3.3 CLOUD-DIENST

Die Aufgabe des Dienstes ist es, für ausstehende Dokumente eine OCR durchzuführen und das Ergebnis der Texterkennung in eine XML-Struktur zu schreiben, welche dann später zur Klassifizierung wieder eingelesen und ausgewertet wird. Der Dienst arbeitet dabei mit mehreren Threads (jeder Thread übernimmt die Texterkennung eines Dokumentes), deren Anzahl konfigurierbar ist und standardmäßig zwischen (einschließlich) 2 und 5 liegt.

Die betriebsinterne XML-Struktur, welche den erkannten Text eines Dokumentes speichert, ist dabei hierarchisch aufgebaut: Unter der Parent-Node befindet sich für jede Seite eine Page-Node mit Metadaten zur Seite (Seitenzahl, Höhe und Breite der Seite) und zwei weiteren Attributen, die den Inhalt der Seite beschreiben. Das erste Attribut enthält eine Aufzählung von Unicode Werten, welche letztendlich den kompletten Text der Seite ergeben. Das zweite Attribut beschreibt für jedes Unicode Zeichen das Rechteck (Left, Top, Width, Height), welches die Position des Zeichens auf der Seite angibt.

Die Vorgehensweise des Dienstes (bzw. jedes Threads des Dienstes) kann dabei folgendermaßen beschrieben werden: Sofern ein Dokument zur Texterkennung aussteht, wird zunächst sichergestellt, dass die Datei vom Dateityp PDF ist (im Falle einer TIF-Datei wird diese in eine PDF-Datei umgewandelt). Daraufhin wird die Datei mit einem Globally Unique Identifier (GUID) versehen, damit sichergestellt wird, dass der Dateiname eindeutig ist. Die Datei wird in einen Amazon S3-Bucket hochgeladen und es wird eine asynchrone Texterkennungsanfrage (keine Dokumentanalyse, da wir nur Seiten, Zeilen und Wörter benötigen) an Textract gestellt. Es werden für jeden Page-Block in der XML-Struktur ein neues Page-Element angelegt und danach jeder

Line-Block der Seite durchgegangen und der Unicode-Wert jedes Zeichens der Zeile notiert. Die Geometrie (das Rechteck) der Zeile wird gleichmäßig auf die Anzahl der Zeichen in der Zeile aufgeteilt, da Textract leider keine Möglichkeit bietet, das Rechteck eines einzelnen Zeichens abzufragen. Am Ende wird das Dokument im S3-Bucket wieder gelöscht und das Ergebnis der OCR (in Form der XML-Struktur) wird lokal gespeichert.

4 STATISTISCHE AUSWERTUNG

Da Amazon Textract als Alternative zu ABBYY FineReader oder Tesseract OCR eingesetzt werden soll, soll sowohl die Geschwindigkeit als auch die Erkennungsqualität des Dienstes ausgewertet und bewertet werden.

4.1 GESCHWINDIGKEIT

Da es zum Zeitpunkt des Praktikums leider nicht möglich ist, Dokumente in Textract parallel auszuwerten (da die Anzahl asynchroner Jobs auf 1 limitiert ist), lässt sich noch keine Aussage über die Skalierbarkeit des Dienstes treffen. Die durchschnittliche Auswertungszeit eines Dokumentes mit nur einer Seite (inklusive dem Hochladen nach Amazon S3 und dem Erstellen der XML-Struktur) beträgt etwa 7 Sekunden. Diese Zeit ist zwar schlechter als die von ABBYY FineReader (etwa 4 Sekunden) und Tesseract OCR (etwa 5 Sekunden), gilt aber trotzdem als akzeptabel. Dies ist damit zu begründen, dass die anderen beiden Dienste lokal (also im auf Servern im betriebsinternen Netzwerk) laufen, während die Kommunikation mit Textract über entfernte Amazon Server geht. Da Textract zudem in der Testphase in EU nur in Dublin (Irland) unterstützt wird, ist der Kommunikationsweg noch länger als er im normalen Fall (mit Frankfurt als Region) sein sollte.

4.2 ERKENNUNGSQUALITÄT

Um die Erkennungsqualität von Amazon Textract bewerten zu können, werden 1.157 Beispielrechnungen eingelesen und deren Rechnungsnummern extrahiert. Die extrahierte Rechnungsnummer wird mit der korrekten Rechnungsnummer (welche sich in einer Datenbank befindet) abgeglichen. Es wird sich dabei zunächst auf die Auswertung der Rechnungsnummer beschränkt, da Amazon Textract zum aktuellen Zeitpunkt noch keinen deutschen Wörterbuchabgleich durchführt und lediglich deutsche Beispieldokumente zum Testen vorhanden sind. Damit würde Amazon Textract bei deutschen Texten aktuell sicherlich schlechtere Ergebnisse liefern, als ABBYY FineReader oder Tesseract OCR (welche beide einen Abgleich mit bekannten deutschen Wörtern durchführen). Da die Rechnungsnummern der Dokumente haupt-

sächlich aus zusammenhangslosen Zeichenketten (Buchstaben, Zahlen und gelegentlich auch Sonderzeichen) bestehen, würde ein Wörterbuchabgleich an dieser Stelle keine Vorteile geben.

Zum Vergleich der ausgelesenen Rechnungsnummern mit den korrekten Rechnungsnummern ist eine Metrik notwendig, die angibt, wie ähnlich oder unterschiedlich zwei Zeichenketten voneinander sind. Wir betrachten dafür die Editierdistanz.

4.2.1 *Editierdistanz*

Die Editierdistanz ist eine Metrik, die angibt wie unterschiedlich zwei Strings (Zeichenketten) voneinander sind, indem die kleinste Anzahl der nötigen Editierungsoperationen gezählt wird, welche einen String in einen anderen String transformieren. Die genaue Definition der Editierdistanz hängt davon ab, welche Art von Operationen zur Editierung zugelassen sind. Unter solchen Operationen zählen das Entfernen von einem Zeichen (`Deletion`), das Einfügen von einem Zeichen (`Insertion`), das Ersetzen von einem Zeichen durch ein anderes Zeichen (`Substitution`) und auch das Vertauschen zweier aneinanderliegender Zeichen (`Transposition`). Wir betrachten im Folgenden hauptsächlich die Levenshtein-Distanz, welche als Operationen das Entfernen, Einfügen und Ersetzen von einem Zeichen im String zulässt.

Wir legen dabei die Kosten jeder Operation auf 1 fest, womit die minimale Distanz also die minimale Anzahl an nötigen Operationen wird. Eine allgemeinere Definition würde jede Operation mit eigenen (nicht negativen) Kosten versehen ($C_{del}, C_{ins}, C_{sub}$).

4.2.2 *Levenshtein-Distanz*

Für zwei Zeichenfolgen a, b mit entsprechenden Längen $|a|, |b|$ weist die Levenshtein-Distanz $d(a, b)$ dann unter anderem folgende Eigenschaften auf:

- $a = b \iff d(a, b) = 0$, da jeder String ohne Operationen in sich selber transformierbar ist.
- $a \neq b \iff d(a, b) > 0$, da mindestens eine Operation (mit nicht negativen Kosten) nötig ist, um a in b zu transformieren.
- $d(a, b) = d(b, a)$, da die Kosten zum Entfernen und zum Einfügen eines Zeichens gleich sind und die beiden Operationen die inverse Operation der jeweils anderen darstellen.
- $d(a, b) + d(b, c) \geq d(a, c)$ (Dreiecksungleichung), da b im besten Fall ein Zwischenschritt in der Transformation von a nach c ist, kann die linke Seite der Ungleichung nicht kleiner als $d(a, c)$ sein.
- $d(a, b) \geq ||a| - |b||$, da bei der Transformation a auf die Länge von b gebracht werden muss und dabei mindestens $||a| - |b||$ Operationen (Entfernen oder Einfügen, je nachdem welcher String länger ist) nötig sind.

- $d(a, b) \leq \max(|a|, |b|)$, da im schlechtesten Fall (keine Zeichen aus a lassen sich zu b zuordnen) $\min(|a|, |b|)$ Zeichen durch andere ersetzt werden müssen und die weiteren $\max(|a|, |b|) - \min(|a|, |b|)$ Zeichen entweder entfernt oder eingefügt werden müssen (je nachdem welcher String länger ist).

Die Levenshtein-Distanz zwischen dem Wort `levenshtein` und dem Wort `meilenstein` (bei uniformen Operationskosten von 1) ist 4. Eine mögliche Umwandlung von `levenshtein` nach `meilenstein` könnte zum Beispiel folgendermaßen aussehen:

1. **`l`**`evenshtein` → **`m`**`evenshtein` (Ersetzen von `l` durch `m`)
2. `me`**`v`**`enshtein` → `me`**`i`**`enshtein` (Ersetzen von `v` durch `i`)
3. `meienshtein` → `mei`**`l`**`enshtein` (Einfügen von `l`)
4. `meilens`**`h`**`tein` → `meilenstein` (Entfernen von `h`)

Mit den Strings $a = a_1 \dots a_{|a|}$ und $b = b_1 \dots b_{|b|}$ folgt die rekursive Definition der Levenshtein-Distanz. Dabei steht $d(i, j)$ für die Levenshtein-Distanz zwischen den ersten i Zeichen von a und den ersten j Zeichen von b.

$$d(i,j) = \begin{cases} j & i = 0 \\ i & j = 0 \\ \min \begin{cases} d(i-1, j-1) + [a_i \neq b_j] \\ d(i, j-1) + 1 \\ d(i-1, j) + 1 \end{cases} & 1 \leq i \leq |a| \land 1 \leq j \leq |b| \end{cases} \tag{4.1}$$

Die Levenshtein-Distanz der beiden kompletten Strings ist dann gegeben durch $d(|a|, |b|)$. Außerdem sei angemerkt, dass das erste Element im Minimum der Substitution eines Zeichens entspricht (bzw. der Zuordnung zweier Zeichen, sofern diese gleich sind), wobei $[a_i \neq b_j]$ für das Iverson-Bracket steht, welches zu 1 evaluiert, sofern der Ausdruck in den Klammern wahr ist und zu 0 wenn der Ausdruck unwahr ist. Das zweite Element im Minimum steht für das Einfügen eines Zeichens in a (wir betrachten die Operationen weiterhin von a nach b) und das dritte Element steht für das Entfernen eines Zeichens aus a. Die Basisfälle der Rekursion berechnen die Distanz mit einem leeren String. Da wir für das Einfügen und für das Entfernen eines Zeichens jeweils die Kosten 1 haben gilt für jeden String s: $d(s, \varnothing) = d(\varnothing, s) = |s|$. Anschaulich ist dies damit zu begründen, dass ein leerer String durch das Einfügen von exakt $|s|$ Zeichen in einen String s transformiert werden kann und umgekehrt kann jeder String s durch das Entfernen von exakt $|s|$ Zeichen in den leeren String transformiert werden.

An dieser Stelle kann mithilfe einer weiteren Eigenschaft der Editierdistanz eine erste Optimierung vorgenommen werden. Wenn die Strings a und b einen gemeinsamen Präfix besitzen, dann ist dieser Präfix für die Berechnung der Editierdistanz $d(a, b)$ nicht relevant. Für jeden Präfix u gilt somit $d(ua, ub) = d(a, b)$. Für unseren rekursiven Algorithmus bedeutet dies, dass im Falle von $a_i = b_j$ die Distanz $d(i, j)$ direkt als $d(i-1, j-1)$ berechnet werden

kann, wodurch an dieser Stelle zwei rekursive Aufrufe gespart werden. Der Algorithmus sieht dann folgendermaßen aus:

$$d(i,j) = \begin{cases} j & i = 0 \\ i & j = 0 \\ \begin{cases} d(i-1,j-1) & a_i = b_j \\ 1 + \min \begin{cases} d(i-1,j-1) \\ d(i,j-1) \\ d(i-1,j) \end{cases} & a_i \neq b_j \end{cases} & 1 \leq i \leq |a| \wedge 1 \leq j \leq |b| \end{cases} \tag{4.2}$$

Der Algorithmus kann direkt nach der obigen Formulierung rekursiv implementiert werden.

Es folgt eine Implementation in C#, wobei der eigentliche Algorithmus als private Methode implementiert wurde (damit diese durch die Indizes rekursionsfähig ist). Eine weitere öffentliche Methode stellt die Schnittstelle zum Programmierer dar. Diese ruft den Algorithmus mit $i = |a|$ und $j = |b|$ auf und berechnet somit mit $d(|a|, |b|)$ die Editierdistanz zwischen allen Zeichen von a und allen Zeichen von b.

```
private static int LevenshteinDistance(string a, string b,
                                       int    i, int    j)
{
    if (i == 0) return j;
    if (j == 0) return i;

    // Subtraktion von 1 im Index, da nullbasierte Strings
    if (a[i - 1] == b[j - 1])
    {
        return LevenshteinDistance(a, b, i - 1, j - 1);
    }

    return 1 + Math.Min(
        LevenshteinDistance(a, b, i - 1, j - 1), // Substitution
        Math.Min(
            LevenshteinDistance(a, b, i, j - 1), // Insertion
            LevenshteinDistance(a, b, i - 1, j)  // Deletion
        )
    );
}

public static int LevenshteinDistance(string a, string b)
{
    return LevenshteinDistance(a, b, a.Length, b.Length);
}
```

Listing 4.1: Rekursiver Algorithmus zur Berechnung der Levenshtein-Distanz

4.2.3 *Dynamische Programmierung*

Leider ist dieser rekursive Ansatz nicht sehr effizient. Um dies zu verdeutlichen betrachten wir die ersten beiden Rekursionstiefen des Algorithmus bei

der Berechnung einer Distanz $d(i, j)$. Wir gehen dabei von dem schlechten Fall aus, dass keine Zeichen übereinstimmen und wir immer alle drei Fälle für das Minimum auswerten müssen. Die Reihenfolge, in der die drei Fälle ausgewertet werden bleiben weiterhin zunächst Substitution, dann Insertion und zuletzt Deletion (dies dient jedoch nur zur weiteren Erklärung).

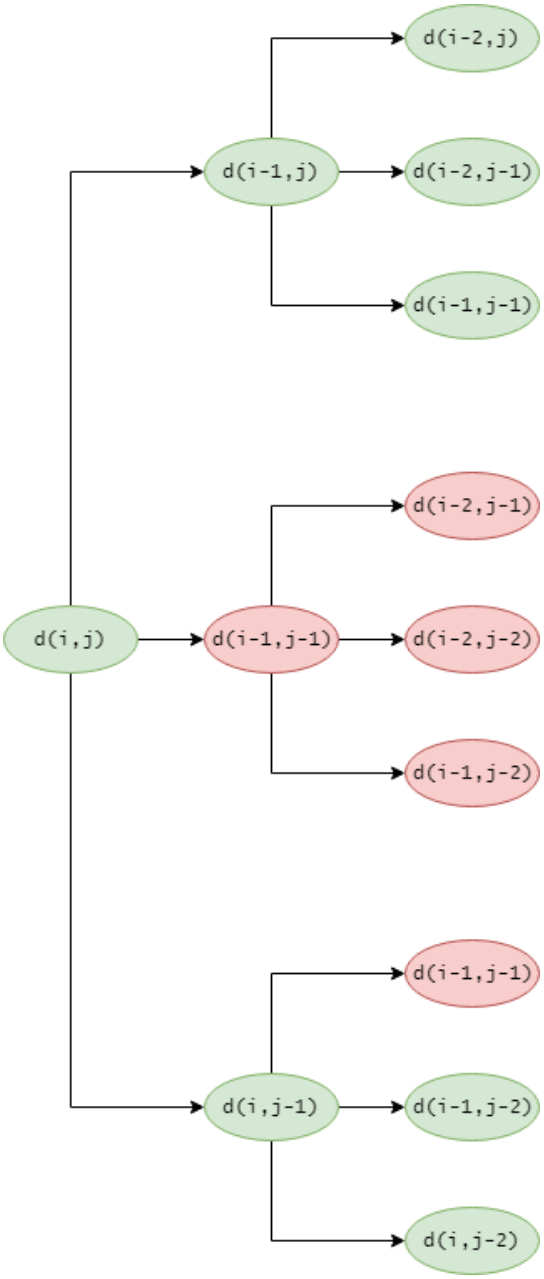


Abbildung 4.1: Rekursionsbaum bei der Berechnung der Levenshtein-Distanz

Wie zu erkennen ist, werden die gleichen Teilprobleme von $d(i, j)$ mehrmals berechnet ($d(i-1, j-1)$ würde dreimal berechnet werden). Diese kombinatorische Explosion führt zu einem exponentiellen Laufzeitverhalten von $\Omega(3^{\min(|a|,|b|)})$ und $\mathcal{O}(3^{|a|+|b|})$. Außerdem sei angemerkt, dass der Algorithmus (auch wenn nicht explizit Speicher angelegt wird) durch den Aufbau

des Stacks bei der Rekursion eine Speicherkomplexität von $\Theta(|a|+|b|)$ besitzt, da die längsten Wege durch den Rekursionsbaum von $d(|a|, |b|)$ nach $d(0, 0)$, $d(0, 1)$ oder $d(1, 0)$ gehen ohne dabei den Rekursionsschritt $d(i-1, j-1)$ zu benutzen, wobei exakt $|a| + |b|$ Funktionsaufrufe des Algorithmus auf dem Stack liegen.

Dies ist ein klassisches Problem aus der dynamischen Programmierung (DP). Der erste Lösungsansatz wäre es, den rekursiven Algorithmus so anzupassen, dass jedes Teilproblem nur einmal berechnet wird, gespeichert wird und im Verlauf der weiteren Rekursion jederzeit wieder abgerufen werden kann. Somit wird nicht unnötig wieder tiefer in die Rekursion gegangen und das Ergebnis kann direkt zurückgegeben werden. Diese Technik wird in der Literatur als Memoisation bezeichnet. Als Datenstruktur für die Memoisation benötigen wir hier ein zweidimensionales Array der Dimension $|a| \times |b|$, für alle möglichen Ergebnisse von $d(i, j) : 1 \leq i \leq |a|, 1 \leq j \leq |b|$.

Da Rekursion zunächst in die Tiefe geht, bevor das nächste Teilproblem eines Rekursionslevels abgearbeitet wird, ergibt sich eine Arbeitsweise wie in der Abbildung 4.1 gezeigt. Die grünen Aufrufe werden berechnet und für den weiteren Verlauf zwischengespeichert. Die roten Aufrufe sind Teilprobleme, die bereits berechnet wurden und gehen damit nicht weiter in die Rekursion, sondern geben lediglich den bereits berechneten Wert zurück. In der Abbildung wird $d(i-1, j-1)$ zunächst als Teilproblem von $d(i-1, j)$ berechnet (was wiederum ein Teilproblem von unseren initialen Problem $d(i, j)$ ist). Nachdem $d(i-1, j)$ komplett berechnet wurde, würde der Algorithmus auf dem höchsten Rekursionslevel zurückkehren und probieren mit $d(i-1, j-1)$ weiter zu machen. Da dieses Problem jedoch bereits berechnet wurde, wird der komplette Rekursionszweig abgeschnitten und es wird nur der zuvor berechnete Wert zurückgegeben.

Eine mögliche Umsetzung in C# könnte folgendermaßen aussehen:

```
private static int LevenshteinDistance(string a, string b, int
    i, int j, int[,] memo)
{
    if (memo[i, j] > -1)
    {
        // Rueckgabe des bereits berechneten Wertes
        return memo[i, j];
    }

    // Sonst Wert normal berechnen, speichern und zurueckgeben
    if (i == 0) return memo[i, j] = j;
    if (j == 0) return memo[i, j] = i;

    if (a[i - 1] == b[j - 1])
    {
        memo[i, j] = LevenshteinDistance(a, b, i - 1, j - 1,
            memo);
    }
    else
    {
        memo[i, j] = 1 + Math.Min(
            LevenshteinDistance(a, b, i - 1, j - 1, memo),
            Math.Min(
```

```
                LevenshteinDistance(a, b, i, j - 1, memo),
                LevenshteinDistance(a, b, i - 1, j, memo)
            )
        );
    }

    return memo[i, j];
}

public static int LevenshteinDistance(string a, string b)
{
    // (|a| + 1) x (|b| + 1) Array um die Basisfaelle
        mitzuspeichern
    // Vereinfacht auch den Umgang mit den einsbasierten Indizes
    int[,] memo = new int[a.Length + 1, b.Length + 1];

    // Memotable mit Wert fuellen, der als "nicht berechnet" gilt
    for (int i = 0; i <= a.Length; i++)
    {
        for (int j = 0; j <= b.Length; j++)
        {
            memo[i, j] = -1;
        }
    }

    return LevenshteinDistance(a, b, a.Length, b.Length, memo);
}
```

Listing 4.2: Levenshtein-Distanz mit Memoisation

Der Ansatz mit Memoisation weist nun zwar eine höhere Speicherkomplexität von $\Theta(|a| \cdot |b|)$ auf (dieser Speicher wird benötigt, um die Ergebnisse der Teilprobleme zwischenzuspeichern), besitzt aber dafür auch eine Laufzeitkomplexität von $\Theta(|a| \cdot |b|)$, da jedes der $|a| \cdot |b|$ Teilprobleme nur noch einmal berechnet werden muss und nicht mehr als 3 mal sein Ergebnis zurückgeben muss.

Da dieser Top-down-Ansatz (wir starten beim Ausgangsproblem $d(|a|, |b|)$ und gehen in der Rekursion bis zu den Basisfällen) weiterhin auf Rekursion basiert, wird für die Erhaltung des Rekursionsstacks zusätzlicher Speicher und Leistung (Auf- und Abbau des Stacks) aufgebracht. Wir betrachten damit noch eine finale Lösung, welche auch auf DP basiert, aber iterativ arbeitet.

Alternativ zur vorherigen Lösung füllen wir unser DP-Array nun nach dem Bottom-up-Ansatz: Wir tragen die Werte für die Basisfälle $d(0, j)$ und $d(i, 0)$ direkt ein und gehen die Teilprobleme dann so durch, dass diese sofort mit den Ergebnissen der vorherigen Teilprobleme berechnet werden können. Dazu berechnen wir zunächst die Teilprobleme $d(1, j)$ für $j = 1 \,..\, |b|$. Die aufsteigende Reihenfolge für j ist dabei wichtig, da wir für die Berechnung von $d(1, j)$ das Ergebnis von $d(1, j - 1)$ benötigen (für $j = 1$ ist dies kein Problem, da wir $d(i, 0)$ für alle i bereits eingetragen haben). Da wir auch die Ergebnisse aller $d(0, j)$ schon haben, können wir damit nun alle $d(1, j)$ berechnen. Mit allen $d(1, j)$ können wir dann aber alle $d(2, j)$ berechnen. Allgemein benötigen wir alle $d(i - 1, j)$ um alle $d(i, j)$ zu berechnen. Für alle $i = 1 \,..\, |a|$ (wieder aufsteigende Reihenfolge) berechnen wir also mit

$j = 1 \,..\, |b|$ alle $d(i, j)$ ohne dabei auf Teilprobleme zu stoßen, die wir für $d(i, j)$ benötigen aber noch nicht berechnet haben. Am Ende ist $i = |a|$ und $j = |b|$ und wir berechnen das Ergebnis des Ausgangsproblems $d(|a|, |b|)$.

Dieser Algorithmus zur Berechnung der Levenshtein-Distanz wird auch als Wagner-Fischer-Algorithmus bezeichnet. Der Algorithmus wurde auch folgendermaßen in C# implementiert, um die Editierdistanz zwischen zwei Rechnungsnummern zu finden.

```
public static int WagnerFischer(string a, string b)
{
    int m = a.Length;
    int n = b.Length;

    int[,] d = new int[m + 1, n + 1];

    // Basisfaelle eintragen
    for (int i = 0; i <= m; i++) d[i, 0] = i;
    for (int j = 0; j <= n; j++) d[0, j] = j;

    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            if (a[i - 1] == b[j - 1])
            {
                d[i, j] = d[i - 1, j - 1];
            }
            else
            {
                d[i, j] = 1 + Math.Min(
                    d[i - 1, j - 1],
                    Math.Min(
                        d[i, j - 1],
                        d[i - 1, j]
                    )
                );
            }
        }
    }

    // Ergebnis steht in d(m, n) = d(|a|, |b|)
    return d[m, n];
}
```

Listing 4.3: Wagner-Fischer-Algorithmus

Die Laufzeitkomplexität des Wagner-Fischer-Algorithmus beträgt wie auch beim Top-down-Ansatz $\Theta(|a| \cdot |b|)$. Der abgebildete Algorithmus besitzt auch weiterhin eine Speicherkomplexität von $\Theta(|a| \cdot |b|)$. Dies lässt sich jedoch noch verbessern, wenn man beachtet, dass für die Berechnung von $d(|a|, |b|)$ nicht die komplette Matrix aufgebaut werden muss. Stattdessen reicht es nur zwei Zeilen zu speichern (die zuvor berechnete Zeile und die aktuelle Zeile). Dies lässt sich damit begründen, dass für eine Zeile i der Matrix, zur Berechnung aller Distanzen $d(i, j)$ dieser Zeile nur die Werte aus den vorherigen Berechnungen $d(i - 1, j)$ benötigt werden (für Substitution und Deletion)

und der Wert direkt links (in derselben Zeile) vom aktuell berechneten Wert (für Insertion). Die Distanzen $d(u, j)$ mit $u < i - 1$ sind also für die weiteren Berechnungen nicht mehr relevant und müssen nicht gespeichert werden. Wählt man nun die Länge des kürzeren Strings als Dimension dieser Zeile werden exakt zwei Arrays der Größe $\min(|a|, |b|)$ benötigt. Damit ergibt sich eine verbesserte Speicherkomplexität von $\Theta(\min(|a|, |b|))$. Dies lässt sich noch weiter verbessern auf nur eine Zeile und eine Variable, da für die Berechnung von $d(i, j)$ auch alle $d(i - 1, v)$ mit $v < j - 1$ irrelevant sind. Wir überschreiben also die Werte der alten Zeile mit den neu berechneten Werten. Dabei steht (der bereits neue Wert) $d(i, j - 1)$ direkt links in der Zeile, der alte Wert $d(i-1, j)$ steht noch an der Stelle, wo jetzt der neue Wert $d(i, j)$ reingeschrieben wird. Die zusätzliche Variable speichert $d(i-1, j-1)$, dessen Wert zuvor links vom aktuellen Wert stand, jedoch durch den Wert $d(i, j - 1)$ überschrieben wurde. Vor jedem Überschreiben eines Wertes müssen wir also den alten Wert zwischenspeichern, da dieser Wert im nächsten Schritt noch mal gebraucht wird.

Die speicheroptimierte Version des Wagner-Fischer-Algorithmus sieht folgendermaßen in C#-Code aus:

```
public static int WagnerFischerSpaceOptimization(string a,
                                                  string b)
{
    int m = a.Length;
    int n = b.Length;

    // Speicherkomplexitaet von O(min(|a|,|b|)) sicherstellen
    if (m < n) return WagnerFischerSpaceOptimization(b, a);

    int[] d = new int[n + 1];
    for (int j = 0; j <= n; j++) d[j] = j;

    for (int i = 1; i <= m; i++)
    {
        int diag = d[0]; d[0] = i;
        for (int j = 1; j <= n; j++)
        {
            int tmp = d[j];
            if (a[i - 1] == b[j - 1])
            {
                d[j] = diag;
            }
            else
            {
                d[j] = 1 + Math.Min(
                    diag,
                    Math.Min(
                        d[j - 1],
                        d[j]
                    )
                );
            }
            diag = tmp;
        }
```

```
    }

    return d[n];
}
```

Listing 4.4: Wagner-Fischer-Algorithmus mit Speicheroptimierung

Der Nachteil bei der speicheroptimierten Version ist, dass am Ende des Algorithmus nicht noch mal die Matrix durchlaufen werden kann, um nachzuvollziehen, welche Operationen zu der minimalen Editierdistanz führen. Wir betrachten dazu noch mal das Beispiel mit $a = "levenshtein"$ und $b = "meilenstein"$ und einer Editierdistanz von 4 (vgl. Abb. 4.2).

Möchten wir nun zusätzlich zu der Editierdistanz noch wissen, welche Menge an Operationen den String a innerhalb von 4 Schritten zu b transformiert, dann müssen wir vom Endergebnis aus (die untere rechte Ecke der Matrix), den Weg durch die Matrix finden, der an jeder Stelle eine minimale Entscheidung getroffen hat (dabei können auch Abzweigungen und damit mehrere Wege entstehen). Der erste Schritt von der unteren rechten Ecke aus geht z. B. nach links-oben, da für `n=n` keine Substitution nötig ist und wir somit bei Kosten von 4 bleiben. Die anderen beiden Alternativen würden beide auf Kosten von $5+1 = 6$ kommen. Die entsprechende Operation, die an dieser Stelle stattgefunden hat, ist also das Matching von `n` und `n`. Führen wir das bis zur oberen linken Ecke durch, so ergeben sich in unserem Beispiel zwei Wege durch die Matrix. Die beiden Wege unterscheiden sich dabei lediglich in einer Operation (in der Tabelle als grün und gelb dargestellt) und haben sonst gleiche Operationen (in der Tabelle als blau dargestellt). Die beiden Wege entsprechen folgenden Operationen (das Matching zweier Zeichen wird nicht aufgeführt):

Weg über grün:

1. Ersetzen von `l` durch `m`
2. Ersetzen von `v` durch `i`
3. Einfügen von `l` hinter `i`
4. Entfernen von `h`

Weg über gelb:

1. Ersetzen von `l` durch `m`
2. Einfügen von `i` hinter `e`
3. Ersetzen von `v` durch `l`
4. Entfernen von `h`

	∅	m	e	i	l	e	n	s	t	e	i	n
∅	0	1	2	3	4	5	6	7	8	9	10	11
l	1	1	2	3	3	4	5	6	7	8	9	10
e	2	2	1	2	3	3	4	5	6	7	8	9
v	3	3	2	2	3	4	4	5	6	7	8	9
e	4	4	3	3	3	3	4	5	6	6	7	8
n	5	5	4	4	4	4	3	4	5	6	7	7
s	6	6	5	5	5	5	4	3	4	5	6	7
h	7	7	6	6	6	6	5	4	4	5	6	7
t	8	8	7	7	7	7	6	5	4	5	6	7
e	9	9	8	8	8	7	7	6	5	4	5	6
i	10	10	9	8	9	8	8	7	6	5	4	5
n	11	11	10	9	9	9	8	8	7	6	5	4

Abbildung 4.2: Backtracking im Wagner-Fischer-Algorithmus

4.2.4 *Relative Editierdistanz*

Um die Erkennungsqualität von mehreren Rechnungsnummern (mit unterschiedlichen Längen) bewerten zu können, muss die Editierdistanz relativ zu den Längen der Strings betrachtet werden. Dazu bilden wir die Distanz auf das Intervall $[0, 1]$ ab. Da die Distanz zwischen zwei Strings a, b maximal $\max(|a|, |b|)$ ist und minimal 0 (die Strings sind identisch), lässt sich die relative Editierdistanz zu

$$\frac{d(a,b)}{\max(|a|,|b|)}$$

berechnen. Die relative Ähnlichkeit der Strings berechnet sich dann zu

$$1 - \frac{d(a,b)}{\max(|a|,|b|)} = \frac{\max(|a|,|b|) - d(a,b)}{\max(|a|,|b|)}$$

.

Um die relative Ähnlichkeit von n erkannten Rechnungsnummern a_k zu n Sollwerten b_k zu berechnen, wird das (gewichtete) arithmetische Mittel der relativen Ähnlichkeiten aller Paare (a_k, b_k) berechnet.

$$\begin{aligned}
&\frac{\sum_{k=1}^{n} \max(|a_k|, |b_k|) \cdot \frac{\max(|a_k|,|b_k|) - d(a_k,b_b)}{\max(|a_k|,|b_k|)}}{\sum_{k=1}^{n} \max(|a_k|, |b_k|)} \\
&= \frac{\sum_{k=1}^{n} \max(|a_k|, |b_k|) - d(a_k, b_k)}{\sum_{k=1}^{n} \max(|a_k|, |b_k|)} \\
&= \frac{\sum_{k=1}^{n} \max(|a_k|, |b_k|) - \sum_{k=1}^{n} d(a_k, b_k)}{\sum_{k=1}^{n} \max(|a_k|, |b_k|)} \\
&= 1 - \frac{\sum_{k=1}^{n} d(a_k, b_k)}{\sum_{k=1}^{n} \max(|a_k|, |b_k|)}
\end{aligned} \tag{4.3}$$

Die Auswertung ergab bei den Rechnungsnummern eine Erkennungsquote von etwa $86\,\%$ (es wird also im Durchschnitt $86\,\%$ einer Rechnungsnummer richtig erkannt). Bei der Erkennung von Textattributen (z. B. Orte oder Namen) sinkt diese auf etwa $68\,\%$. Da Amazon Textract keine Umlaute kennt, wurden die Tests noch mal mit einer Vorbearbeitung der Attribute (Umlaute werden bei den Sollwerten durch die entsprechenden Vokale ersetzt) durchgeführt. Die Erkennungsrate steigt damit auf $76\,\%$. Ein Abgleich der erkannten Wörter mit einem deutschen Wörterbuch würde diese Quote sicherlich (wie bei den Rechnungsnummern) auf über $80\,\%$ bringen, was als eine akzeptable Erkennungsrate gilt.

Das entwickelte Framework zum Testen der Erkennungsrate verschiedenster Attribute wurde darüber hinaus auch zum Auswerten verschiedener Klassifizierungstechniken benutzt.

5 FAZIT

Das Praktikum bei der Classcon Consulting GmbH hat mir wertvolle Einblicke in den beruflichen Alltag eines Kleinunternehmens im IT-Bereich gegeben. Besondern die Zusammenarbeit in kleinen Teams und die Absprache mit anderen Entwicklern hat sich als interessant und abwechslungsreich herausgestellt. Neben meiner Hauptaufgabe (Amazon Textract), welche hauptsächlich von mir alleine bearbeitet wurde, durfte ich an vielen Stellen in kleineren und auch größeren Projekten mithelfen. Die Teilnahme an regelmäßigen Entwicklerbesprechungen hat gezeigt, dass die Kommunikation im Team auch in kleineren Projekten von großer Bedeutung ist.

Meine erworbenen Fähigkeiten und Kenntnisse aus dem Studium konnte ich vielseitig anwenden, einbringen und in vielen Bereichen auch vertiefen. Darüber hinaus konnte ich an viel neues Spezialwissen gewinnen, was so im Studium nicht behandelt wird. Neben den erworbenen Fachkenntnissen konnte ich auch einen groben Einblick in die Berufsbilder verschiedenster Berufe (Web-Entwicklung, Consultant, Backend-Programmierung) bekommen.

Insgesamt hat sich die Praxisphase bei der Classcon Consulting GmbH als ein interessantes, abwechslungsreiches, forderndes und durchaus empfehlenswertes Praktikum herausgestellt.

LITERATURVERZEICHNIS

[1] Dic Sonneveld Jerry Slocum. *The 15 Puzzle Book: How it Drove the World Crazy.* Slocum Puzzle Foundation, 2006.

[2] Yakov Isidorovich Perelman. *Fun with Maths and Physics : Brain Teasers Tricks Illusions.* MIR Publishers (Moscow), 1988.