Ludwig-Maximilians-Universität München
Institut für Informatik
Prof. Dr. Thomas Seidl
Anna Beer, Florian Richter

München, 29.06.2018

Algorithmen und Datenstrukturen

SS 2018

Übungsblatt 13: Paradigmen

Tutorien: 10.07-13.07.2018

Aufgabe 13-1 *Backtracking*

In der Vorlesung haben Sie als Beispiel für ein Backtracking Problem das 4 Damenproblem kennengelernt.

(a) Geben Sie nun eine Lösung für das 5 Damenproblem an. Berechnen Sie diese mittels Backtracking. Sie müssen nicht alle Schritte explizit darstellen. Sie können die Damen mit x kennzeichnen und in die folgende Tabelle eintragen:

	1	2	3	4	5
A					
B					
C					
D					
E					

Lösungsvorschlag:

Eine mögliche Lösung ist zum Beispiel:

	1	2	3	4	5
A		X			
B				X	
C	X				
D			X		
E					X

(b) Wie groß ist die Komplexität für das n Damenproblem, wenn ein naiver Algorithmus für die Berechnung der Lösung verwendet wird?
Es reicht eine obere Abschätzung in O-Notation anzugeben. Dabei kann man vernachlässigen, dass Damen auch diagonal gehen können.

Lösungsvorschlag:

Die Komplexität ist eine Teilmenge von $n!$ und damit $O(n!)$. Sie berechnet sich durch die Anzahl der möglichen Positionierungen der Damen. Für die 1. Zeile sind es n mögliche Positionen, für die 2. $n-1$ mögliche Positionen (etwas weniger eigentlich, allerdings vernachlässigen wir hier die Diagonalen, da es reicht, die Obermenge anzugeben), ..., für die n-te Zeile gibt es nur noch (maximal) eine Position. Damit $n \cdot (n-1) \cdot \ldots \cdot 1 = n!$. Mittels verschiedener mathematischer Eigenschaften lassen sich jedoch effizientere Algorithmen finden.

(c) Weiterhin Sei eine modifizierte Version eines Sudokus mit jeweils nur 4 anstelle von 9 Feldern gegeben. Wie bei dem originalen Sudoku gilt, dass in jeder Zeile, in jeder Spalte und in jedem Block (2x2 Unterquadrat) die Zahlen 1 bis 4 nur einmal vorkommen dürfen.

Gegeben Sei nun folgendes Sudoku:

1	3		
2	4	3	
3	2	1	4
4			

(i) Geben Sie generell grob einen naiven Algorithmus in Pseudocode an, der mittels Backtracking eine Lösung berechnen kann. Tipp: Es muss nicht über alle Spalten und Zeilen iteriert werden, sondern es reicht eine Betrachtung der freien Felder.

Lösungsvorschlag:

Ein möglicher Brute-Force Algorithmus (alle Möglichkeiten durchprobieren) ist folgender:

```
Algorithmus sudoku(int[][] sudokuBrett, Feld[] freieFelder, int freiesFeldZahl))
        if istZahlAufBrettDoppelt(sudokuBrett)
                return false
        if freiesFeldZahl == freieFelder.length
                return true
        for zahl = 1 bis 4
                freiesFeld = freieFelder[freiesFeldZahl]
                sudokuBrett[freiesFeld.zeile][freiesFeld.spalte] = zahl
                if sudoku(sudokuBrett, freiesFeldZahl + 1)
                        return true
        return false
```

Die Ausführung startet dann mit sudoku(sudokuBrett, freieFelder, 0)

(ii) Berechnen Sie die (eindeutige) Lösung des obigen Sudokus mit Hilfe des erstellten Algorithmus. Geben Sie dabei alle Zwischenschritte an, wobei es reicht, die anfangs noch leeren Felder anzugeben. Im folgenden sollen die freien Felder von links nach rechts und dann von oben nach unten betrachtet werden, also wie in gewohnter deutscher Schreib-/Leserichtung (wobei jede beliebige Abfolge dieser Felder möglich wäre).

Lösungsvorschlag:

1	3		
2	4	3	
3	2	1	4
4			

⇒ Wahl von Zahl 2:

1	3	2	
2	4	3	
3	2	1	4
4			

⇒ Keine Lösung:

1	3	2	X
2	4	3	
3	2	1	4
4			

⇒ Keine Möglichkeit für X, daher Backtracking zu letzter Wahlmöglichkeit und einsetzen von 4:

1	3	4	
2	4	3	
3	2	1	4
4			

⇒ Eindeutig 2:

1	3	4	2
2	4	3	
3	2	1	4
4			

⇒ Eindeutig 1:

1	3	4	2
2	4	3	1
3	2	1	4
4			

⇒ Eindeutig 1:

1	3	4	2
2	4	3	1
3	2	1	4
4	1		

⇒ Eindeutig 2:

1	3	4	2
2	4	3	1
3	2	1	4
4	1	2	

⇒ Eindeutig 3. (Eindeutige) Lösung gefunden:

1	3	4	2
2	4	3	1
3	2	1	4
4	1	2	3

Aufgabe 13-2 *Dynamische Programmierung*

(a) Berechnen Sie die Edit-Distanz der Wörter `ALGORITHMEN` und `DATENSTRUKTUREN` mit Hilfe dynamischer Programmierung.

(b) Geben Sie die Komplexität der dynamischen Programmierung für dieses Beispiel im Gegensatz zu einer naiven rekursiven Lösung zur Berechnung der Edit-Distanz an.

Lösungsvorschlag:

(a)

		A	L	G	O	R	I	T	H	M	E	N
	0	1	2	3	4	5	6	7	8	9	10	11
D	1	**1**	2	3	4	5	6	7	8	9	10	11
A	2	**1**	2	3	4	5	6	7	8	9	10	11
T	3	**2**	2	3	4	5	6	6	7	8	9	10
E	4	**3**	3	3	4	5	6	7	7	8	8	9
N	5	4	**4**	4	4	5	6	7	8	8	9	8
S	6	5	5	**5**	5	5	6	7	8	9	9	9
T	7	6	6	6	**6**	6	6	6	7	8	9	10
R	8	7	7	7	7	**6**	7	7	7	8	9	10
U	9	8	8	8	8	**7**	7	8	8	8	9	10
K	10	9	9	9	9	8	**8**	8	9	9	9	10
T	11	10	10	10	10	9	9	**8**	9	10	10	10
U	12	11	11	11	11	10	10	9	**9**	10	11	11
R	13	12	12	12	12	11	11	10	10	**10**	11	12
E	14	13	13	13	13	12	12	11	11	11	**10**	11
N	15	14	14	14	14	13	13	12	12	12	11	**10**

(b) Dynamische Programmierung:
Wie in der Vorlesung gezeigt, liegt die Komplexität für zwei Zeichenketten s und t bei $O(|s| \cdot |t|)$. Für unser Beispiel müssen demnach maximal ca. $11 \cdot 15 = 165$ Operationen bei dynamischer Programmierung durchgeführt werden.

Rekursive Lösung:
Für die rekursive Lösung müssen im schlechtesten Fall in jeder Iteration 3 rekursive Aufrufe getätigt werden, um das Minimum dieser Aufrufe berechnen zu können. Dabei verringert sich immer die Länge mindestens einer der Zeichenketten s oder t um 1. D.h. können maximal $|s| + |t|$ Rekursionsschritte gemacht werden. Insgesamt ergibt sich damit also eine obere Schranke für die Laufzeit von $O(3^{|s|+|t|})$. Das bedeutet für unser Beispiel eine Durchführung von maximal $3^{11+15} = 3^{26} \approx 2.5 \cdot 10^{12}$