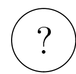
Zusammenfassung, der zu lernenden Dinge für die Informatik B Wiederholungsprüfung

erstellt von Christoph Tornau

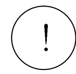
Habt Glauben an Gott! Markus 11,22

Bei der Prüfung ist es besonders wichtig, die Zusammenhänge zu erkennen. Es geht weniger darum genaue Dinge zu wissen, aber mehr darum den groben Überblick zu haben und Methoden, die wir für einige Probleme angewandt haben auch für andere Probleme auch aus dem Stehgreif zu übertragen. Weiterhin ist auch eine genaue präzise Sprache erwünscht.

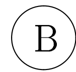
Erstellungsdatum dieser Version: 16. April 2002



Bedeutet: Noch einmal anschauen oder nicht verstanden



Bedeutet: Absolut wichtig



Bedeutet: Hier gibt es Beispiele zum üben

Inhaltsverzeichnis

I Informatik III **11**

1 Rekurrenzen **11**
1.1 Euklid (Bestimmung des ggTs) . . . 11
1.2 Binäre Suche . . . 11
1.3 Mergesort . . . 11
1.4 Quicksort . . . 12
1.4.1 Worst-Case . . . 12
1.4.2 Average Case . . . 12

2 Mastertheorem **12**
2.1 Übersicht . . . 12
2.2 Verständnis des Beweises . . . 13

3 Ableitung der untersten Schranke von Sortierverfahren $n \log(n)$ **13**

4 Weitere Sortierverfahren **13**
4.1 Countingsort . . . 13
4.2 Bucketsort . . . 13
4.3 Radixsort . . . 14

5 Graphen - Tiefensuche/Breitensuche **14**

6 Dijkstra-Beweis **14**

7 Bäume **14**
7.1 Definition . . . 14
7.2 Höhe und Tiefe . . . 15
7.3 Regeln . . . 15
7.3.1 Allgemeine . . . 15
7.3.2 Für Binär-Bäume . . . 15
7.4 Erzeugung von Wäldern aus Graphen . . . 15

8 Ungerichtete Graphen **15**
8.1 Allgemeine Eigenschaften . . . 15
8.2 Äquivalente Aussagen . . . 15

9 Minimumheap **16**
9.1 Darstellung durch Arrays . . . 16
9.2 Einfügen . . . 16
9.3 Löschen . . . 16
9.4 Heapsort . . . 16
9.4.1 Algorithmus . . . 16
9.4.2 Laufzeit . . . 16

10 Union-Find-Wälder **16**

11 Kruskal zur Erstellung des MST **17**
11.1 Abbruchkriterium . . . 17
11.2 Laufzeit . . . 17

12 Speichermöglichkeiten von Datenmengen mit dynamischer Größe **17**
12.1 Hashing 17
12.1.1 Verschiedene Hashfunktionen 17
12.1.2 Offenes Hashing mit Kollisionslisten 17
12.1.3 Geschlossenes Hashing mit offener Adressierung 17
12.1.4 Dynamisches Hashing 17
12.2 Balancierte Bäume 18
12.2.1 Höhe von Bäumen 18
12.2.2 Suchbäume 18
12.2.3 AVL 18
12.2.4 Blatt 18
12.2.5 B-Bäume 18

13 Greedy **19**
13.1 Huffman-Codes 19
13.2 Graph-Färbe-Problem 19
13.3 Rucksackproblem 19

14 Spielbäume **19**
14.1 15er-Puzzle 19
14.2 Tic-Tac-Toe 20

15 Dynamisches Programmieren **21**
15.1 Floyd 21
15.1.1 Beweis 21
15.1.2 Beispiel 21
15.1.3 Negative Zyklen 22
15.1.4 Konstruktion des Pfades 22
15.2 Warshall 22
15.3 TSP 22
15.3.1 TSP mit dynamischen Programmieren 23
15.3.2 Kosten TSP mit dynamischen Programmieren 23

16 Graphen **23**
16.1 DFS in Digraphen (Baumkante,Vorwärtskante,Rückwärtskante und Seitwärtskante) 23
16.2 Topologische Sortierung 24
16.3 SCCs - „strongly connected component" - Starke Zusammenhangskomponenten 24
16.3.1 Algorithmus - Aho/Hopcroft/Ullman 24
16.3.2 Beweis des SCC-Algorithmusses 24

17 Netzwerkflußproblem **25**
17.1 Ford&Fulkerson-Algorithmus 25
17.1.1 Algorithmus 25
17.1.2 Worst Case 26
17.1.3 Optimierung 26
17.2 Bipartites Matching 26

18 Mustererkennung **27**
18.1 Naives Verfahren 27
18.2 KMP - Knuth, Morris und Pratt 27

18.3 Boyer-Moore . . . 27

19 Scanlines **28**
19.1 Sichtbarkeitsproblem für horizontale Objekte . . . 28
19.2 Schnittproblem für horizontale und vertikale Objekte . . . 28
19.3 Geometrisches Divide&Conquer für horizontal und vertikal liegende Objekte . . . 28
19.4 Schnittproblem für (fast) beliebig liegende Objekte . . . 28
19.5 Voronoidiagramme (nur eine kurze Vorstellung) . . . 29

20 Randomisierte Algorithmen **29**
20.1 Las Vegas & Monte Carlo . . . 29
20.2 Randomisiertes Quicksort . . . 29
20.3 Mustererkennung - Karp&Rabin . . . 30
20.4 Primzahltest - Miller&Rabin . . . 30
20.5 Universelles Hashing . . . 30
20.6 Skip-Listen . . . 30

II Informatik IV **31**

0 Allgemeines **31**
0.1 Funktionen . . . 31
0.2 Äquivalenzrelation . . . 31
0.3 Formale Sprachen . . . 31

1 Berechenbarkeit **32**
1.1 Registermaschinen (kurz: RM) . . . 32
1.1.1 Aufbau und Befehle . . . 32
1.1.2 Kostenmaße . . . 32
1.1.3 Effiziente Algorithmen . . . 33
1.2 Turingmaschinen (kurz: DTM oder NTM) . . . 33
1.2.1 Einbandige „normale" Turingmaschinen (deterministisch) . . . 33
1.2.2 Mehrbandige Turingmaschinen . . . 34
1.2.3 Nichtdeterministische Turingmaschinen . . . 34
1.2.4 Kosten von Turingmaschinen . . . 34
1.2.5 Simulation von mehrbandigen Turingmaschinen durch einbandige Turingmaschinen . . . 34
1.2.6 Programmierung von DTMs . . . 35
1.3 Registermaschine ↔ Turingmaschine . . . 35
1.3.1 Simulation von Registermaschinen durch Turingmaschinen . . . 36
1.3.2 Simulation von Turingmaschinen durch Registermaschinen . . . 36
1.3.3 Simulation von Turingmaschinen mit Turingmaschinen - Universelle Turingmaschinen . . . 38

2 Entscheidbarkeit **38**
2.1 Wortproblem . . . 38
2.2 Charakteristische Funktion . . . 38
2.2.1 Totale charakteristische Funktion . . . 38
2.2.2 Partielle charakteristische Funktion . . . 38
2.3 Entscheidbarkeit . . . 38
2.3.1 Semientscheidbarkeit . . . 38

2.3.2 Entscheidbarkeit . . . 39
2.3.3 Nicht entscheidbar ↔ Unentscheidbar . . . 39
2.3.4 Nicht semientscheidbar . . . 39
2.4 Zusammenhang zwischen Entscheidbarkeit und Semientscheidbarkeit . . 39
2.4.1 L entscheidbar, dann auch $\overline{L}$ entscheidbar . . . 39
2.4.2 L und $\overline{L}$ semientscheidbar ⇔ L entscheidbar . . . 39
2.5 Rekursive Aufzählbarkeit . . . 40
2.5.1 Rekursiv aufzählbar → Semientscheidbar . . . 40
2.5.2 Semientscheidbar → Rekursiv aufzählbar . . . 40
2.6 Halteproblem . . . 40
2.6.1 Die Sprache des Halteproblems und des speziellen Halteproblems 40
2.6.2 Satz: Das Halteproblem ist unentscheidbar . . . 41
2.6.3 Informeller Beweis des Halteproblems mittels Halteproblemtabelle 41
2.6.4 Das Halteproblem ist Semientscheidbar . . . 41
2.6.5 Spezielles Halteproblem . . . 41
2.6.6 Reduktion . . . 42
2.6.7 Reduktionsprinzip . . . 43
2.6.8 Unentscheidbarkeit des Halteproblems . . . 43
2.6.9 Die Nicht-Semientscheidbarkeit des Komplements des Halteproblems: Satz . . . 43
2.6.10 Die Nicht-Semientscheidbarkeit des Komplements des Halteproblems: Informell . . . 43
2.6.11 Die Nicht-Semientscheidbarkeit des Komplements des Halteproblems: Formal . . . 44
2.7 Andere unentscheidbare Probleme . . . 44
2.7.1 Satz von Rice . . . 44
2.8 Nichtdeterminismus (Ergänzung) . . . 45
2.8.1 Nichtdeterministische Entscheidbarkeit . . . 45
2.8.2 NTM → DTM . . . 45

3 Komplexitätsklassen 45
3.1 Unterschiedliche Problemvarianten . . . 45
3.1.1 Vorstellung mit Beispiel . . . 45
3.1.2 Überführung ineinander am Beispiel des Cliquenproblems . . . 46
3.2 Zeitkomplexität . . . 46
3.2.1 SAT - Guess and Check Methode . . . 46
3.2.2 Primzahlen . . . 47
3.2.3 Cliquenproblem . . . 47
3.2.4 Zeitkomplexitätsklassen . . . 47
3.2.5 P,NP,EXPTIME . . . 47
3.2.6 NP ⊆ EXPTIME . . . 48
3.2.7 Polynomialzeit akzeptierende NTMs . . . 48
3.3 Platzkomplexität . . . 48
3.3.1 PSPACE . . . 48
3.3.2 SAT ∈ PSPACE . . . 48
3.3.3 In-Place Acceptance ∈ PSPACE . . . 48
3.3.4 NP ⊆ PSPACE . . . 49
3.3.5 PSPACE ⊆ EXPTIME . . . 49
3.3.6 Satz von Savitch: $PSPACE = NPSPACE$ . . . 49
3.3.7 Übersicht über die Komplexitätsklassen . . . 50

4 NPC Beweise **50**
4.1 Die große Frage der Informatiker P=NP oder P≠NP? 50
4.2 Polynomielle Reduzierbarkeit 50
4.3 NP-hart, NP-vollständig 50
4.3.1 NP-hart 50
4.3.2 NP-vollständig 50
4.3.3 Sätze 51
4.4 Aussagelogik 51
4.5 Satz von Cook 52
4.5.1 Satz 52
4.5.2 Beweis 52
4.5.3 Korrektheit 54
4.5.4 Kosten 54
4.6 Prinzipielle Techniken für NP-Vollständigkeitsbeweise 54
4.6.1 Spezialisierung 54
4.6.2 Lokale Ersetzung 55
4.6.3 Transformation mit verbundenen Komponenten 55
4.7 Einige NP-Vollständigkeitsbeweise 55
4.7.1 SAT $\leq_{poly}$ 3SAT (lokale Ersetzung) 55
4.7.2 3SAT $\leq_{poly}$ CP (Transformation) 56
4.7.3 3SAT $\leq_{poly}$ RP (Transformation sowie auch Spezialisierung) . . . 57
4.7.4 0/1 Integer Linear Programming (Transformation) 59
4.7.5 Weitere NPC-Beweise 59
4.8 coNP 60
4.9 PSPACE-Vollständigkeit 60
4.10 Zusammenfassung 61

5 Praktischer Einsatz von formalen Sprachen: Compiler **61**
5.1 Lexikalische Analyse: Scanner 61
5.2 Syntaktische Analyse: Parser 61
5.3 Semantische Analyse und Codeerzeugung 61

6 Grammatiken **62**
6.1 Definitionen 62
6.1.1 Notation 62
6.1.2 Definition Grammatik 62
6.1.3 Die durch eine Grammatik erzeugte Sprache 62
6.1.4 Äquivalenzen von Grammatiken 62
6.2 Chomsky Hierarchie 63
6.2.1 Tabellarische Übersicht 63
6.2.2 Beispiele 63
6.3 Inklusion der Chomskyhierarchie 63
6.4 Grafische Darstellung der Chomskyhierarchie 64
6.5 Transformation um die ϵ-Freiheit bis auf $S \rightarrow \epsilon$ in Typ 1 herzustellen . . 64
6.5.1 Algorithmus 64
6.5.2 Beispiel 65
6.6 Abschlußeigenschaften 65
6.6.1 Vereinigung 65
6.6.2 Konkatenation 66
6.6.3 Kleenabschluß 66
6.7 Sprachen vom Typ 0 66

6.7.1 Allgemein . . . 66
6.7.2 Nichtdeterministisches Semientscheidungsverfahren für das Wortproblem . . . 66
6.7.3 Abgeschlossenheit . . . 66
6.8 Kontextsensitive Sprachen - Typ 1 . . . 67
6.8.1 LBAs - Linear beschränkte Automaten . . . 67
6.8.2 Determinismus ↔ Nichtdeterminismus . . . 67
6.8.3 Wortproblem . . . 67
6.8.4 Abgeschlossenheit . . . 67
6.9 Wortproblem für Typ 0 und Typ 1 Sprachen . . . 67

7 Reguläre Sprachen **68**
7.1 Endliche Automaten . . . 68
7.1.1 Definition: Deterministischer endlicher Automat (DFA) . . . 68
7.1.2 Fangzustand . . . 69
7.1.3 Lauf . . . 69
7.1.4 Definition: Nichtdeterministischer endlicher Automat (NFA) . . . 69
7.2 Endliche Automaten und reguläre Grammatiken . . . 69
7.2.1 Übersicht . . . 69
7.2.2 DFA → reguläre Grammatik . . . 70
7.2.3 NFA → DFA (Potenzmengenkonstruktion) . . . 70
7.2.4 Effizienz von NFAs . . . 71
7.2.5 Reguläre Grammatik → NFA . . . 71
7.3 Verknüpfungen regulärer Sprachen . . . 72
7.3.1 NFAs mit ϵ-Übergängen . . . 72
7.3.2 Vereinigung - $L(M_1) \cup L(M_2)$ . . . 72
7.3.3 Durchschnitt - $L(M_1) \cap L(M_2)$ . . . 72
7.3.4 Komplement - $\overline{L(M)} = \Sigma^* \backslash L(M)$ . . . 72
7.3.5 Konkatenation - $L(M_1) \circ L(M_2)$ . . . 73
7.3.6 Kleenabschluß - $L(M)^* = L(M^*)$ . . . 73
7.4 Algorithmen zur Feststellung von Eigenschaften . . . 73
7.4.1 Das Wortproblem . . . 73
7.4.2 Leerheitstest . . . 73
7.4.3 Äquivalenztest . . . 73
7.4.4 Endlichkeitstest . . . 73
7.5 Pumping Lemma für reguläre Sprachen . . . 74
7.5.1 Definition . . . 74
7.5.2 Kernaussage . . . 74
7.5.3 Beweis . . . 74
7.5.4 Beispiele . . . 75
7.6 Reguläre Ausdrücke . . . 75
7.6.1 Syntax regulärer Ausdrücke . . . 75
7.6.2 Regulärer Ausdruck → NFA . . . 76
7.6.3 DFA → regulärer Ausdruck . . . 77
7.7 Syntaxdiagramme . . . 78
7.7.1 Regulärer Ausdruck → Syntaxdiagramm . . . 78
7.7.2 Syntaxdiagramm → DFA . . . 78
7.8 Zusammenfassung . . . 79
7.9 Minimierung von endlichen Automaten . . . 79
7.9.1 Satz von Myhill & Nerode . . . 79
7.9.2 Definition: Minimalautomat . . . 79

7.9.3 Beweis: Minimalisierungsalgorithmus . . . 79
7.9.4 Minimierungsalgorithmus . . . 79

8 Kontextfreie Sprachen (CFG) 80
8.1 Ableitungsbaum . . . 80
8.1.1 Rechtsableitung / Linksableitung . . . 80
8.1.2 Eindeutigkeit / Mehrdeutigkeit . . . 80
8.2 Nutzlose Variablen . . . 80
8.2.1 Definition . . . 80
8.2.2 Algorithmus . . . 80
8.3 Chomsky Normalform (CNF) . . . 81
8.3.1 Definition . . . 81
8.3.2 Algorithmus zur Eliminierung der Kettenregeln . . . 81
8.3.3 Algorithmus zur Erzeugung der Chomsky-Normalform . . . 81
8.3.4 Beispiel . . . 82
8.3.5 Größe der CNF . . . 83
8.4 CYK-Algorithmus für das Wortproblem . . . 84
8.4.1 Algorithmus . . . 84
8.4.2 Laufzeit . . . 84
8.4.3 Beispiele . . . 84
8.5 Pumping Lemma für kontextfreie Sprachen . . . 85
8.5.1 Satz . . . 85
8.5.2 Beweis . . . 85
8.6 Abschlußeigenschaften . . . 85
8.7 Griebach Normalform . . . 85
8.7.1 Satz . . . 85
8.7.2 Konstruktionsalgorithmus . . . 85
8.7.3 Beispiel . . . 86
8.8 Kellerautomaten . . . 86
8.8.1 Definition: Nichtdeterministischer Kellerautomat (NKA) . . . 86
8.8.2 Konfiguration, Notation der Übergangsfunktion und Konfigurationswechsel . . . 86
8.8.3 Akzeptanzverhalten . . . 87
8.8.4 Akzeptanz durch Endzustand → Akzeptanz durch leeren Keller . 87
8.8.5 Akzeptanz durch leeren Keller → Akzeptanz durch Endzustand . 87
8.8.6 Kontextfreie Grammatik → NKA . . . 87
8.8.7 NKA → kontextfreie Grammatik . . . 88
8.8.8 NKAs mit zwei Kellern . . . 88
8.8.9 reguläre Sprache ∩ kontextfreie Sprache = kontextfreie Sprache . 88

9 Deterministische kontextfreie Sprachen 88
9.1 Eigenschaften . . . 88
9.2 Deterministische Kellerautomaten (DKAs) . . . 88
9.2.1 Definition . . . 88
9.2.2 Eingliederung in die Chomskyhierarchie . . . 89
9.3 Akzeptanzbedingungen . . . 89
9.4 Präfixeigenschaft . . . 89
9.4.1 Definition Präfixeigenschaft . . . 89
9.4.2 Deterministische kontextfreie Sprachen mit Präfixeigenschaft werden von DKAs mit Leerer-Keller-Akzeptanz entschieden . . . 89
9.5 Beispiele . . . 90

9.6 Komplettes Schaubild der Chomskyhierarchie . . . 90
9.7 Abschlußeigenschaften . . . 91

10 LR(0)-Grammatiken **91**
10.1 Einleitung zu LR(k)-Grammatiken . . . 91
10.2 Eigenschaften von LR(0)-Grammatiken . . . 91
10.3 Begriffsdefinition: Reduktion . . . 91
10.4 Definition LR(0)-Grammatik . . . 91
10.5 Definition von Begriffen für eine verbale Definition von LR(0)-Grammatiken 92
10.5.1 Griff . . . 92
10.5.2 Rechtssatzform . . . 92
10.6 Verbale Definition von LR(0)-Grammatik . . . 92
10.7 Eindeutigkeit . . . 92
10.8 Präfixeigenschaft . . . 92
10.9 Nichtdeterministischer LR(0)-Parser . . . 93

11 LR(0)-Items **93**

A Tabellarische Aufführung der Abschlußeigenschaften verschiedener Sprachtypen **94**

B Prüfungsprotokolle **95**
B.1 Meine eigene Prüfung . . . 95
B.1.1 Allgemein . . . 95
B.1.2 Informatik III . . . 95
B.1.3 Informatik IV . . . 95
B.2 Meine Wiederholungsprüfung . . . 95
B.2.1 Info III . . . 95
B.2.2 Info IV . . . 96

Teil I
Informatik III

1 Rekurrenzen

1.1 Euklid (Bestimmung des ggTs)

Der Euklidische Algortihmus berechnet den ggT mittels rekursiven Aufrufen[1] folgendermaßen:

$$ggT(x, y) = ggT(y, x \ mod \ y)$$

Hierbei werden alle Faktoren, die nicht Primfaktoren beider Zahlen sind, herausgezogen, bis letztendlich $ggT(\sum Primfaktoren, 0)$ übrigbleibt.

Der **Worst-Case-Fall** des Euklidalgorithmus sind die Fibonaccizahlen. Die Fibonaccizahlen haben die Eigenschaft, daß wenn man den Rest von F_k/F_{k-1} bestimmt, F_{k-2} herauskommt, da $F_k = F_{k-1} + F_{k-2}$. Es müssen somit genau k Schritte durchgeführt werden, um darauf zu kommen, daß der ggT zweier Fibonaccizahlen, die im Wort-Case aufeinander folgen, 1 ist.
Für Fibonaccizahlen gilt

$$F_k \geq (\frac{3}{2})^{k-1}$$

Wir sehen, daß je größer die Zahlen werden, die Häufigkeit der Fibonaccizahlen logarithmisch abnimmt, da die Größe der Fibonaccizahlen exponentiell steigt. Somit gilt $O(\log n)$.

1.2 Binäre Suche

$$T(n) = \begin{cases} 0 & f\ddot{u}r\, n \leq 0 \\ 1 & f\ddot{u}r\, n = 1 \\ 1 + T(\lfloor \frac{n+1}{2} \rfloor - 1) & f\ddot{u}r\, n > 1 \end{cases}$$

$$T(n) = 1 + T(\lfloor \frac{n+1}{2} \rfloor - 1) \leq 1 + T(\lfloor \frac{n}{2} \rfloor)$$

1.Fall: n ist eine Zweierpotenz, etwa 2^k

$$\begin{aligned} T(n) &= T(2^k) \leq 1 + T(\tfrac{2^k}{2}) \\ &= 1 + T(2^{k-1}) \\ &\leq 1 + 1 + T(2^{k-2}) = 2 + T(2^{k-2}) \\ &\leq ... \leq r + T(2^{k-r}) =_{r=k} k + T(2^0) = k + 1 \\ &= log(n) + 1 = O(log(n)) \end{aligned}$$

2.Fall: n beliebig: Sei $2^{k-1} < n < 2^k (\rightarrow k = \lceil log(n) \rceil)$
$T(n) \leq T(2^k) \leq k + 1 = \lceil log(n) \rceil + 1 = O(log(n))$

1.3 Mergesort

$T(1) = 0$
$T(n) = n + T(\lfloor \frac{n+1}{2} \rfloor) + T(n - \lfloor \frac{n+1}{2} \rfloor) \leq n + 2 * T(\lfloor \frac{n+1}{2} \rfloor) \approx n + 2 * T(\lfloor \frac{n}{2} \rfloor)$
1. Fall $n = 2^k$:

[1] Es gibt auch ein iteratives Verfahren des Euklidischen Algorithmusses.

$$\begin{array}{ll} T(2^k) & = 2^k + 2T(\frac{2^k}{2}) \\ & = 2^k + 2T(2^{k-1}) \\ & \leq 2^k + 2(2^{k-1} + 2T(2^{k-2}) \\ & \leq 2 * 2^k + 2^2 T(2^{k-2}) \\ & \leq ... \leq r * 2^k + 2^r * T(0) \\ & = r * 2^k \\ & =_{r=k} k * 2^k \\ \Rightarrow & O(log(n) * n) \end{array}$$

1.4 Quicksort

1.4.1 Worst-Case

$$T(n) = (n-1) + \begin{array}{c} max \\ 1 \leq i \leq n \end{array} (T(n-i) + T(i-1)) = n - 1 + T(n-1) + T(0)$$

$$= n - 1 + n - 2 + n - 3 + ... + n - n = \sum_{k=1}^{k=n} (n-k)$$

$\Rightarrow O(n^2)$

1.4.2 Average Case

Für $n \geq 2$:

$$T(n) = n - 1 + \frac{1}{n} \sum_{i=1}^{n} (T(n-i) + T(i-1)) = n - 1 + \frac{2}{n} \sum_{i=1}^{n} T(i-1)$$

Durch Induktion kann man zeigen (ohne Beweis):

$$T(n) = 2(n+1)H(n) - 4n$$

Wobei $H(n)$ die harmonsische Reihe ist, für die gilt:

$$\int_1^{n+1} \frac{1}{x} dx \leq \sum_{i=1}^{n} \frac{1}{i} \leq \int_2^{n+1} \frac{1}{x-1} dx + 1$$

Hiermit läßt sich $H(n)$ nach $log(n+1)$ bzw. $log(n)+1$ abschätzen. Somit ist die Laufzeit im Average Case:

$$\Rightarrow \Theta(nlog(n))$$

2 Mastertheorem

2.1 Übersicht

$$T(n) = \begin{cases} O(1) & falls\, n = 1 \\ a * T(\frac{n}{b}) + c * n & falls\, n = b^k > 1 \end{cases}$$

1.Fall: $a < b : T(n) = \Theta(n)$
2.Fall: $a = b : T(n) = \Theta(n * log(n))$
3.Fall: $a > b : T(n) = \Theta(n^k)\, mit\, k = log_b(a)$

2.2 Verständnis des Beweises

Das Mastertheorem sind im Prinzip Regeln, die aussagen, welches Laufzeitverhalten der Algorithmus hat. Die Laufzeit wird abhänging von der Größe der einzelnen Teilprobleme, der Anzahl der entstandenen Teilprobleme und der Rechenzeit für die Erstellung bzw. Zusammenführung dieser Teilprobleme bestimmt.

- Dabei ist klar, daß wenn die Teilprobleme summiert kleiner sind als das Orignialproblem und die Erstellung der Teilprobleme auch nur linare Zeit beansprucht, der ganze Algorithmus in linearer Zeit ausgeführt werden kann.
- Wenn die Größe der Teilprobeme genau der Größe des Orignialproblems ist und des weiteren linearer Platz gebraucht wird, um diese Teilprobleme zu erstellen, benötigen wird $\Theta(n \log n)$ Zeit.
- Werden die einzelnen Teilprobleme in der Summe größer, als das Orignalproblem ist klar, daß wir noch mehr Laufzeit benötigen.[2]

3 Ableitung der untersten Schranke von Sortierverfahren $n\log(n)$

Wir können eine Entscheidungsbaum erzeugen, der anhand von Vergleichen auf die richtige Permutation der Eingabe schließt. Für n Zahlen gibt es $n!$ Permutationen.

Jeder Binärbaum der Höhe h hat höchstens 2^h Blätter. Es gilt

$$2^h \geq n! \Rightarrow h \geq \log(n!)$$

Da $n! \geq (\frac{n}{e})^n$ können wir $\log(n!)$ folgendermaßen abschätzen

$$\log n! \geq \log\left(\frac{n}{e}\right)^n = n\log\left(\frac{n}{e}\right) = n\log(n) - n\log(e)$$

$\Rightarrow \Omega(n\log(n))$
Somit haben wir eine Baumtiefe von mindestens $n\log(n)$, was die untere Schranke für Sortierverfahren ist.

4 Weitere Sortierverfahren

4.1 Countingsort

Das Vorkommen der Elemente im zu sortierenden Array wird einfach in einem Array gezählt und dann linear wieder geschrieben. Dieses Verfahren ist nur bei kleinen Universen anwendbar. Des weiteren dürfen hinter den Elementen keine weiteren Informationen mehr versteckt sein, die beim Sortieren mitsortiert werden (bei Datenbanken zum Beispiel der Fall).

4.2 Bucketsort

Die Elemente werden auf Buckets gelegt. Anders als bei Countingsort dürfen auch Informationen bei den Elementen enthalten sein.

[2] Diese Sache steht auf Seite 146 im Info III Skript

4.3 Radixsort

Einzelne Teile werden mit stabilen Verfahren vorsortiert. Z.B kann kann eine alphabetische Sortierung damit erzeugen, indem man die einzelnen Buchstaben mit einem stabilen Verfahren von hinten nach vorne sortiert.

5 Graphen - Tiefensuche/Breitensuche

Tiefensuche läuft mit einem Stack, die Breitensuche mit einer Queue. Sie laufen mit einer Laufzeit $O(|V|+|E|)$, weil jeder Knoten einmal auf den Speicher gelegt wird und jede Kante inspiziert wird.

Mittels Breitensuche lassen sich kürzeste Wege ermitteln, wobei das Kostenmaß der Kanten uniform, d.h $= 1$ ist, indem man für jeden unbesuchten Knoten, den man in die Queue einreiht die Kosten des Vorgängers um 1 erhöht und in den Knoten schreibt. Die Kosten kann man in einer Matrix ablegen.

Kosten für längste Wege lassen sich erzeugen, indem man die Kosten negiert und eine Algorithmus verwendet, der die kürzesten Wege berechnet.

6 Dijkstra-Beweis

Hinweis: Beim Daijkstra können wir gut eine Priorityqueue verwenden, um die Menge der noch nicht fixierten Knoten zu verwalten.

Startknoten ist der Knoten v. Wir haben eine Menge D, in welcher die Knoten enthalten sind, für die der kürzeste Weg von v aus schon bestimmt ist[3]

Von einem Knoten w dessen Abstand von v $d(v,w)$ ist, können wir einen Knoten x, welcher noch nicht in der Menge D enthalten ist, mit den minimalsten Kosten von v erreichen. Wir wählen diesen Knoten. Formalisiert gilt für alle Paare $w' \in D \backslash w$ und $x' \in (V \backslash D) \backslash x$

$$d(w,x) \leq d(w',x')$$

Würden wir nun einen Knoten $y \in V$ wählen, so würde gelten

$$d(v,w) + d(w,x) \leq d(v,y) + d(y,x)$$

Wenn $y \in D$ ist $d(v,y) + d(y,x)$ größer als $d(v,x)$, da $d(v,x)$ schon am kleinsten ist, weil in der Menge D schon alle kürzesten Pfade bestimmt sind.
Es ist unmöglich, daß $y \notin D$, da so der Weg $d(v,y) + d(y,x)$ garantiert größer ist, da über einen anderen Knoten aus D gegangen werden muß und $d(v,x)$ minimal ist.

7 Bäume

7.1 Definition

- Jeder Graph ohne Knoten und folglich auch ohne Kanten ist ein Baum
- Für genau einen Knoten gilt $Pre(v_0) = \emptyset$. Für alle anderen Knoten gilt: v ist genau über einen Pfad von v_0 erreichbar.
- $\Leftrightarrow$ Für genau einen Knoten gilt $Pre(v_0) = \emptyset$. Für alle anderen Knoten gilt $|Pre(v)| = 1$.

[3]Diese Menge können wir nach und nach herstellen. Wir starten dabei mit der leeren Menge, weil der Knoten v selbst mit der Weglänge 0 erreichbar ist und dies garantiert auch der kürzeste Weg ist.

7.2 Höhe und Tiefe

Die Höhe des leeren Baumes ist -1. Die Höhe des Baumes mit nur dem Wurzelknoten ist 0 usw.
Tiefe wird gesagt, wenn man die Entfernung eines Knoten zur Wurzel angibt. Der Wurzelknoten selbst hat die Tiefe 0.

7.3 Regeln

- $n = |V|$ Knotenanzahl
- $h = H(T)$ Höhe

7.3.1 Allgemeine

- Hat der Baum den Grad k, so ist $n \leq \frac{k^{h+1}-1}{k-1}$
- Für $k \geq 2$ gilt: $Höhe(T) \geq \log_k(n \cdot (k-1)+1) - 1$

7.3.2 Für Binär-Bäume

- $n \leq 2^{h+1} - 1$
- $Höhe(T) \geq \log_2(n+1) - 1$

7.4 Erzeugung von Wäldern aus Graphen

Wälder werden erzeugt, indem der Graph mittels DFS oder BFS durchgegangen wird. Dies geschieht, indem wir die Knoten, die in den Adjazenslisten stehen entweder in die Queue oder den Stack einreihen. Sollen wir einen schon besuchten Konten einreihen, so reihen wir diesen nicht ein.

8 Ungerichtete Graphen

8.1 Allgemeine Eigenschaften

- $n \geq 2$ und $|E| < n - 1 \Rightarrow G$ ist nicht zusammenhängend
- $E \geq n \geq 3 \Rightarrow G$ ist zyklisch.

8.2 Äquivalente Aussagen

Folgende Aussagen sind äquivalent

- G ist zusammenhängend und zyklenfrei
- G ist azykklisch und $|E| = |V| - 1$
- G ist zusammenhängend und $|E| = |V| - 1$
- Zu jedem Knotenpaar (v, w) gibt es genau einen Pfad von v nach w

9 Minimumheap

9.1 Darstellung durch Arrays

Array[1]	:	Wurzel
Array[2i]	:	linker Sohn des Knotens mit der Nummer i
Array[2i+1]	:	rechter Sohn des Knotens mit der Nummer i

9.2 Einfügen

Einfügen an der ersten möglichen Position. Danach wird der Knoten mit dem Vater solange getauscht, bis der Vater kleiner als der Vorgängerknoten ist.

9.3 Löschen

Löschen, in dem wir den zu löschenden Knoten mit dem letzten Knoten im Baum vertauschen und dann wegnehmen. Der vertauschte Knoten muß solange mit einem Sohn verglichen werden, bis er kleiner als beide Söhne ist. Dabei muß immer mit dem kleineren der beiden Söhne getauscht werden.

9.4 Heapsort

9.4.1 Algorithmus

Heapsort kann durchgeführt werden, indem wir immer das oberste Element entnehmen und danach die Minimumeigenschaften mit dem Algorithmus für das Löschen wiederherstellen.
Dabei kann der Heap zuerst geschrieben werden, indem die Hälfte der Zahlen direkt in den hinteren Teil des Arrays für den Heap kopiert werden. Dann kann immer das Array von hinten mit einem Vater aufgefüllt werden, der dann eventuell abgesenkt wird.

9.4.2 Laufzeit

Für das Absenken im Baum benötigen wir die Laufzeit $O(\log n)$. Da wir dies $n$mal für alle Elemente, die wir nacheinander dem Heapbaum in der sortierten Reihenfolge entnehmen, tun müssen, erhalten wir die Laufzeit $O(n \log n)$. Diese Laufzeit wird auch benötigt, um den Baum vorher aufzubauen. Also erhalten wir insgesamt $O(n \log(n))$.

10 Union-Find-Wälder

Union-Find-Wälder werden dazu benutzt, um die Zusammenhangskomponenten von ungerichteten Graphen zu bestimmen.
Wir starten mit trivialen Zusammenhangskomponenten, in der jeweils ein Knoten vorhanden ist. Wir verringern zwei getrennte Zusammenhangskomponenten zu einer, wenn es eine Kante zwischen zwei in diesen beiden Bäumen enthaltenen Knoten gibt. Um alle Kanten zu bestimmen, gehen wir die alle Adjazenzlisten durch.
Union-Find-Bäume werden vereiningt, indem der jeweils kleinere Baum an den größeren gehängt wird. Die Größe wird zur Laufzeitoptimierung bei jedem Baum mitgespeichert.
Einen Zyklus haben wir gefunden, wenn wir einen Baum des Waldes mit sich selbst vereinigen sollen.
Der Wurzelknoten eines Baumes kann mit $O(\log(n))$ gefunden werden, da der Baum

für jeden inneren Knoten mindestens 2 Söhne hat. Die Vereinigung geschieht in konstanter Zeit $O(1)$. Führen wir den Union-Find-Algorithmus n-mal aus, so erhalten wir als Kosten $O(n \log(n))$.

11 Kruskal zur Erstellung des MST

11.1 Abbruchkriterium

Sobald der Graph genau $|V|-1$ Kanten hat, können wir abbrechen, da sonst ein Zyklus sofort erzeugt wird.

11.2 Laufzeit

Die Laufzeit des alleinigen Kruskalalgorithmus ist wegen der Sortierung $O(|E| \log |E|)$. Dazu muß allerdings noch der Zyklentest kommen, der mittels BFS oder DFS in $O(|E|+|V|)$ $|V|-1$-mal läuft.

12 Speichermöglichkeiten von Datenmengen mit dynamischer Größe

12.1 Hashing

Beim Hashing wird mit Hilfe einer surjektiven Funktion das Universum auf einen kleineren Speicher abgebildet.

12.1.1 Verschiedene Hashfunktionen

- Modulofunktion: $x\ mod\ y$ wobei y die Größe des Hashspeichers ist
- Mittelquadrat: Die Speicheradresse wird quadriert. Danach werden aus der quadrierten Zahl in der Mitte Ziffern herausgenommen: Beispiel: $128^2 = 16\underline{38}4 \Rightarrow 38$
- Multiplikationsmethode: Die Zahl wird mit einer irrationalen Zahl zwischen $]0, 1[$ multipliziert. Danach erhalten wir eine Orignial+d-bittige Zahl. Die letzten d Bits werden als Speicheradresse für den Hash verwendet.

12.1.2 Offenes Hashing mit Kollisionslisten

Wenn es Kollisionen gibt, dann werden an den Hashspeicher Listen angehängt, in denen Elemente gespeichert werden, die nicht mehr in den normalen Speicher passten, weil diese schon besetzt war.

12.1.3 Geschlossenes Hashing mit offener Adressierung

Beim geschlossenen Hashing mit offener Adressierung wird bei belegten Hashspeicherplätzen mittels anderer Hashfunktionen auf andere Speicherplätze ausgewichen.

12.1.4 Dynamisches Hashing

Schon einmal berechnete Ergebnisse von Teilproblemen eines großen Problems werden gespeichert. Diese können aber problemlos gelöscht werden, wenn ein neuer Wert die Hashspeicherzelle überschreibt, da sie neu berechnet werden können.

12.2 Balancierte Bäume

Bei den Bäumen ist jeweils das Verhalten beim Einfügen und Entfernen wichtig.

12.2.1 Höhe von Bäumen

leerer Teilbaum:	-1
ein Knoten:	0
	usw.

12.2.2 Suchbäume

Bei einem INORDER-Durchlauf eines Suchbaumes sind alle Elemente sortiert.

12.2.3 AVL

Rotationen üben. Typen von Rotationen lernen!

- Wenn $bal(v) = -2$ und $bal(right(v)) \in \{-1, 0\}$ einfache Rotation nach links
- Wenn $bal(v) = -2$ und $bal(right(v)) = 1$ Doppelrotation rechts-links
- Wenn $bal(v) = 2$ und $bal(left(v)) \in \{1, 0\}$ einfache Rotation nach rechts
- Wenn $bal(v) = 2$ und $bal(left(v)) = -1$ Doppelrotation links-recht

Die Höhe eines AVL-Baumes ist $O(\log n)$. Schlechteste AVL-Bäume sind Fibonaccibäume. Bei diesen steht die Fibonaccizahl F_{n+1} in der Wurzel.

12.2.4 Blatt

In den inneren Knoten werden nur die Schlüssel, die zur Suche benötigt werden, gespeichert. Die eigentlichen Informationen stehen in den Blättern. Dabei sind die Blätter zum besseren sequentiellen Auslesen untereinander doppelt verknüpft.

12.2.5 B-Bäume

B-Bäume sind Speicherstrukturen für den externen Speicher.
Wenn ein B-Baum die Ordnung m hat, dann bedeutet das, daß jeder Knoten mindestens m aber höchstens $m \cdot 2$ Schlüsselwerte mit den dazugehörigen Daten beinhaltet. (Davon ist die Wurzel ausgenommen)
B-Bäume haben folgende Eigenschaften:

- Jeder Knoten, außer die Wurzel hat mindestens m und höchstens $2m$ Schlüsselwerte, wobei m die Ordnung ist.
- Die Wurzel hat höchstens $2m$ Schlüsselwerte. Nach unten gibt es jedoch keine Grenze.
- Jeder Knoten mit k Schlüsselwerten hat $k + 1$ Söhne, wenn dieser Knoten nicht ein Blatt ist.
- Alle Blätter haben dieselbe Tiefe bezüglich des Wurzelknotens.

Wir **fügen** Schlüssel zu dem Baum **hinzu**, indem wir sie in dem geeigneten Blattknoten einfügen. Wird dieser zu groß (OVERFLOW), müssen wir ihn splitten. Hierbei wandert der mittlere Wert dieses Knoten in den darüber liegenden Vater und aus dem einen Knoten werden zwei gemacht.
Beim **Löschen** kann es passieren, daß ein Knoten zu klein wird (UNDERFLOW). Wenn es möglich ist, borgen wir uns Schlüsselwerte von einem benachbarten Bruderknoten. Hierbei geht ein Wert aus dem Bruder in den gemeinsamen Vaterknoten, während ein Schlüsselwert aus dem Vaterknoten in den Knoten mit dem UNDERFLOW kommt. Geht dies nicht, so wenden wir Konkatenation an und zwei Bruderknoten werden zu einem Knoten zusammengefügt. Dieser Knoten nimmt des weiteren noch den Wert aus dem Vaterknoten auf, der nun auf keinen Sohn mehr zeigt.
Die Laufzeit von Suchen, Einfügen und Löschen auf B-Bäumen ist

$$O(Höhe(T)) = O(\log_{(m+1)}(N))$$

wobei m die Ordnung und N die Schlüsselanzahl.

13 Greedy

13.1 Huffman-Codes

!

Wir immer wieder für die beiden Knoten mit der geringsten Häufigkeitkeit einen Knoten mit der Summe beider Häufigkeiten hinzu, an den diese beiden Knoten angehängt werden.

13.2 Graph-Färbe-Problem

Beim Graphfärbealgorithmus überprüft man, ob man mit m Farben den Graphen färben kann. Wenn nicht erhöht man m um 1.

13.3 Rucksackproblem

Das Allgemeine Rucksackproblem ist mit einem Greedyalgorithmus optimal lösbar. Das {0,1}-Rucksackproblem nicht. Es ist sogar NPC.

14 Spielbäume

Für manche Spiele läßt sich der Spielbaum vollständig konstruieren. Für viele Spiele ist diesr jedoch zu groß. Man verwendet deshalb eine Pay-Off-Funktion, die die Güte einer Konfiguration berechnet und nach dieser Güte über den kommenden Spielzug entscheidet.

14.1 15er-Puzzle

Die Güte der Konfiguration wird berechnet mit

Anzahl der Plättchen, die an falscher Stelle sind + Baumtiefe (Anzahl schon verschobener Plättchen)

Je höher dieser Wert ist, desto schlechter ist es.
Es wird immer (LC-Methode) der Knoten expandiert, der momentan den optimalsten Wert hat. So ist sichergestellt, daß stets der kürzeste Weg dorthin verwandt wird.

Beim 15er-Puzzle ist es jedoch möglich, daß eine Konfiguration nicht in Zielkonfiguration zu bringen ist. Der Spielgraph besteht aus zwei unzusammenhängenden starken Zusammenhangskomponenten.

14.2 Tic-Tac-Toe

Die Pay-Offfunktion berechnet

$$\begin{array}{c}\text{Anzahl der möglichen Reihen des eigenen Spielers}\\ -\text{Anzahl der möglichen Reihen für den Gegner}\end{array}$$

Je höher dieser Wert ist, desto besser.
Es wird immer angenommen, daß der Gegner den besten möglichen Zug macht, also den für einen am schlechtesten.
Es wird ein Baum mittels Backtracking bis zu einer bestimmten Tiefe erstellt. Auf dieser Tiefe werden die Konfigurationen bewertet. Ist der darüberliegende Knoten ein MAX-Knoten, d.h. der Spieler selbst ist am Zug, so wird der Knoten mit dem größten Wert unter den Brüdern für den MAX-Knoten verwandt, um den größtmöglichen Gewinn zu haben.
Ist es ein MIN-Knoten, so wird der Knoten mit dem geringsten Wert genommen. Dies entspricht dem Wort-Case, daß der Gegner genau den für einen schlimmsten Zug macht.
Kommen wir zur Wurzel des Baumes, einem MAX-Knoten, so nehmen wir dann den Weg mit dem größten Gewinn.
Mittels α-β-Pruning können wir Knoten abschneiden, die sowieso schon übertrumpft oder untertrumpft sind.
Z.B. ist in einer Reihe von MINKNOTEN einer dieser Knoten maximal. Wenn durch Überprüfung der Ebene unter dem MIN-Knoten festgestellt wird, daß dort ein Knoten minimaler als schon der gefundene maximale Wert des MIN-Knoten ist, können wir diesen Knoten gleich wegschneiden.

α-cut:	Cutting der Söhne eines MAX-Knotens
β-cut:	Cutting der Söhne eines MIN-Knotens

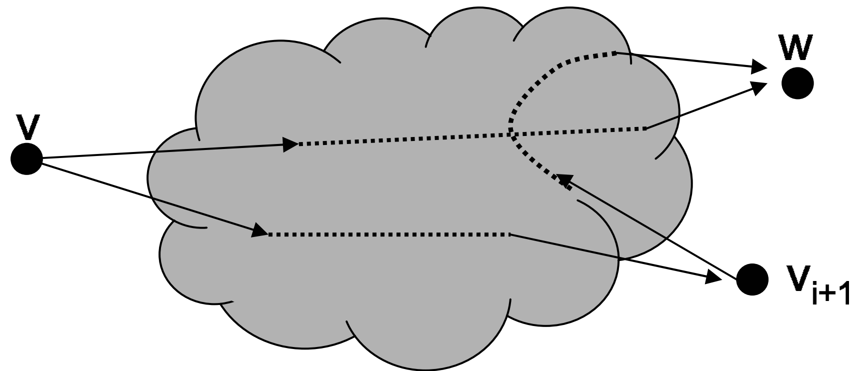
15 Dynamisches Programmieren

15.1 Floyd

15.1.1 Beweis

Die Idee dieses Algorithmusses ist es, daß für jeden Weg zu einem Knoten geprüft wird, ob er kürzer ist, wenn man direkt geht oder wenn man schon durch eine vorberechnete Menge D geht. Wird in der Menge D ein Knoten eingefügt, so wird versucht für jeden Knoten zu jedem Knoten einen Weg zu finden, der über diesen Knoten kürzer ist.

$$\Delta^{i+1}(v,w) = min\{\Delta^i(v,w), \Delta^i(v,v_{i+1}) + \Delta^i(v_{i+1},w)\}$$



15.1.2 Beispiel

$D_0 = \emptyset$:

	v_1	v_2	v_3	v_4	v_5
v_1	0	3	8	∞	-4
v_2	∞	0	∞	1	7
v_3	∞	4	0	∞	∞
v_4	2	∞	-5	0	∞
v_5	∞	∞	∞	6	0

$D_1 = \{v_1\}$:

	v_1	v_2	v_3	v_4	v_5
v_1	0	3	8	∞	-4
v_2	∞	0	∞	1	7
v_3	∞	4	0	∞	∞
v_4	2	**5**	-5	0	**-2**
v_5	∞	∞	∞	6	0

$D_2 = \{v_1, v_2\}$:

	v_1	v_2	v_3	v_4	v_5
v_1	0	3	8	**4**	-4
v_2	∞	0	∞	1	7
v_3	∞	4	0	**5**	**11**
v_4	2	5	-5	0	-2
v_5	∞	∞	∞	6	0

$D_3 = \{v_1, v_2, v_3\}$:

	v_1	v_2	v_3	v_4	v_5
v_1	0	3	8	4	-4
v_2	∞	0	∞	1	7
v_3	∞	4	0	5	11
v_4	2	**-1**	-5	0	-2
v_5	∞	∞	∞	6	0

$D_4 = \{v_1, v_2, v_3, v_4\}$:

	v_1	v_2	v_3	v_4	v_5
v_1	0	3	**-1**	4	-4
v_2	**3**	0	**-4**	1	**-1**
v_3	**7**	4	0	5	**3**
v_4	2	-1	-5	0	-2
v_5	**8**	**5**	**1**	6	0

$D_5 = \{v_1, v_2, v_3, v_4, v_5\}$:

	v_1	v_2	v_3	v_4	v_5
v_1	0	**1**	**-3**	**2**	-4
v_2	3	0	-4	1	-1
v_3	7	4	0	5	3
v_4	2	-1	-5	0	-2
v_5	8	5	1	6	0

15.1.3 Negative Zyklen

Im Graphen dürfen keine negativen Zyklen vorkommen. Dann werden Pfade gebildet, die unendlich klein sind und das schafft dieser Algorithmus auch nicht.

15.1.4 Konstruktion des Pfades

Es reicht, wenn wir uns bei jeder Veränderung merken, über welchen Knoten wir die Kosten der Kante verändert haben. Dann können wir den Pfad rekursiv zurückverfolgen.

15.2 Warshall

Beim Warshall geht es nur um die Erreichbarkeitsanalyse. Wir verzichten deshalb auf Kantenkosten und legen nur eine Wahrheitstabelle an, die aussagt, ob ein Knoten von einem anderen aus erreichbar ist. Wir verwenden ein ähnliches Schema wie das des Floydalgorithmusses. Wir nehmen immer einen Knoten aus der noch nicht in D enthaltenen Menge mit hinzu und bestimmen für diesen für alle j Knoten:

$$R^{k+1}(i,j) = R^k(i,j) \wedge (R^k(i,j) \vee R^k(k,j)$$

15.3 TSP

Das TSP (Traveling Salesman Problem - Problem eines Handlungsreisenden) ist das Problem den kürzesten Weg auf den Kanten durch alle Knoten zu finden, wobei jeder Knoten genau einmal besucht werden muß und auch demzufolge jede Kante nur einmal besucht werden darf. Das TSP-Problem ist das Problem einen **Hamiltonkreis** auf dem Graphen zu finden.

Auch das TSP läßt sich mit Hilfe des dynamischen Programmierens effinzienter lösen. Das naive Verfahren hat die Laufzeit $O(n!)$, da alle Permutationen ausprobiert werden müssen.

15.3.1 TSP mit dynamischen Programmieren

Die Laufzeit des TSP-Algorithmus läßt sich verringern, wenn man dynamsiches Programmieren anwenden. Hierbei werden angefangen von allen zweielementigen Teilmengen die kürzesten Wege in allen Teilmengen gesucht, die dann letztendlich zu immer größeren Hamiltonwegen zusammengesetzt werden, bis letztendlich nur noch ein Hamiltonweg übrigbleibt[4]. Das ganze läuft in einer Laufzeit von $O(n^2 \cdot 2^n)$ mit einem Platz von $O(n \cdot 2^n)$.

15.3.2 Kosten TSP mit dynamischen Programmieren

- Initialisieren kostet $\Theta(n)$ für jeden Knoten.
- Kosten pro l-elementige Teilmenge, die berechnet wird sind $\Theta(N_W)$

$$N_W = \sum_{v \in W} \sum_{w \in (W \setminus \{v\})} 1 = l \cdot (l-1)$$

- Kosten summiert für alle Teilmengen w von $v \setminus \{s\}$

$$\sum_{l=2}^{n} \sum_{w, wobei\ |w|=l} N_W =$$

$$\textstyle\sum_{l=2}^{n} \underbrace{\binom{n}{l}}_{=2^n} \underbrace{l \cdot (l-1)}_{\leq n^2}$$

$$\Rightarrow \Theta(2^n n^2) = \Theta(n^2 2n)$$

- Platzkomplexität ist

$$Theta(n \cdot 2^n),$$

  da für jede Teilmenge die Kosten des kürzesten Weg4es und der kürzeste Weg gespeichert werden müssen.

16 Graphen

16.1 DFS in Digraphen (Baumkante,Vorwärtskante,Rückwärtskante und Seitwärtskante)

Wir durchsuchen den Graphen mit einer Tiefensuche (DFS). Dabei gehen wir die Adjazenslisten der Knoten durch. Dabei sind die Knoten,

- die noch nicht besucht sind, blau
- die besucht worden sind, schwarz

[4]Wie das genau geht, haben wir nicht gemacht

- und die gerade noch auf dem Stack liegen, für welche die DFS also noch läuft, grün

Die dabei explorierten Kanten werden folgendermaßen genannt:

- **Baumkante**: Wenn der Zielknoten beim explorieren der Kante blau ist. (ganz normale Kante)
- **Vorwärtskante**: Wenn der Zielknoten beim explorieren der Kante schwarz ist. (d.h. der Index, der dem Zielknoten bei der Tiefesuche zugeteilt worden ist, ist größer als der Index von dem momentanen Knoten)
- **Rückwärtskante**: Wenn der Zielknoten beim explorieren der Kante grün ist. (d.h. das die Tiefensuche für diesen Knoten nicht abgeschlossen ist.) Der Knotenindex des Zielknotens ist kleiner als der Knotenindex des momentanen Knotens. **Wir haben in diesem Fall einen Zyklus gefunden.**
- **Seitwärtskante**: Alle übrigen Kanten. Die Tiefensuche kann vollständig abgeschlossen sein, ohne das ein Teilgraph besucht worden ist, wenn dieser von diesem Bereich nicht explorierbar ist. Alle Kanten, die von diesem Bereich dann in den alten schon abgeschlossenen Bereich führen, werden Seitwärtskanten genannt.

16.2 Topologische Sortierung

Eine topologische Sortierung kann nur dann vorliegen, wenn der Graph keine Zyklen hat. Der Knoten, der keine Vorgänger hat, bekommt die topologische Sortierungsnummer 1. Dann gehen wir nacheinander die Adjazenslisten mittels Breitensuche durch und numerieren die einzelnen Knoten durch, die wir in die Schlange einfügen.

16.3 SCCs - „strongly connected component" - Starke Zusammenhangskomponenten

16.3.1 Algorithmus - Aho/Hopcroft/Ullman

Der Algorithmus funktioniert wie folgt:

- Wir gehen den Graphen mit Hilfe einer Tiefensuche durch. Dabei bestimmen wir den Schwarzfärbungsindex eines jeden Knotens. Der Schwarzfärbungsindex ist die Reihenfolge in der die Knoten von der Tiefensuche wieder verlassen werden und schwarz gefärbt werden.
- Wir drehen alle Kanten um. D.h. wir erzeugen aus dem Graphen G den inversen Graphen G^{-1}
- Wir lassen wiederum die Tiefensuche laufen. Dabei starten wir diese immer wieder von dem Knoten mit dem **höchsten** Schwarzfärbungsindex, der noch nicht besucht ist. Heraus kommt dann ein DFS-Wald. Jeder Baum im Wald ist eine SCC.

16.3.2 Beweis des SCC-Algorithmusses

- Es ist klar, daß wenn man innerhalb einer SCC die Kanten umdreht immer noch eine SCC hat, weil eine SCC einen Zyklus bildet. Es geht so nur die andere Richtung herum.
- Nehmen wir an, im Originalgraphen G gab es eine Kante von

$$A \text{ nach } B.$$

Dann gibt es in G^{-1} eine Kante von

$$B \text{ nach } A.$$

Die Tiefensuche für die aktuelle SCC B würde Gefahr laufen auch andere SCCs in sich mit aufzunehmen, da es eine Kante gibt. Dies ist jedoch nicht der Fall, da die Tiefensuche im Originalgraphen G über die Kante gekommen ist und folglich beim Schwarzfärben die Knoten in den anderen SCCs einen höheren Schwarzfärbungsindex bekommen haben und folglich zuerst exploriert werden. Also darf die Kante in G^{-1} von B nach A gar nicht gegangen werden.

17 Netzwerkflußproblem

Das Netzwerkflußproblem ist das Problem des maximalen Flusses von einem Startknoten zu einem Zielknoten. Hierbei können durch die Kanten Flüsse fließen, wobei die Inschrift einer jeden Kante angibt, wie hoch der Fluss in dieser Kante momentan ist und maximal sein darf.

- Bei einem Knoten, der kein Start oder Zielknoten ist, gehen genausoviele Flüsse weg, wie reinkommen. Die Summe aller einkommenden Flüsse minus die Summe aller herausgehenden Flüsse ist also 0.

$$\sum_{w \in N \setminus \{v\}} f(w,v) - \sum_{w \in N \setminus \{v\}} f(v,w) = 0$$

- Wir können das Flußdiagramm über die Kanten an jeder Stelle in zwei Stücke schneiden. Sei S die Menge der Knoten die auf der einen Seite dieses Schnittes liegen. Der **Nettofluß** ist definiert mit

$$Flow(f,S) = \sum_{v \in S \;\; w \notin S} f(v,w) - \sum_{v \in S \;\; u \notin S} f(u,v)$$

- Für alle Schnitte gilt: Der Nettofluß ist für jeden Schnitt gleich.

- Die Kapazität ist der maximal Fluß, der über einen Schnitt geführt werden kann. Es gilt

$$cap(S) = \sum_{v \in S \;\; w \in post(v) \setminus S} c(v,w)$$

- Es gilt: Der maximale Fluß über das gesamte Netzwerk ist die minimalste Kapazität, die durch einen Schnitt gehen kann:

$$maxflow = mincut$$

17.1 Ford&Fulkerson-Algorithmus

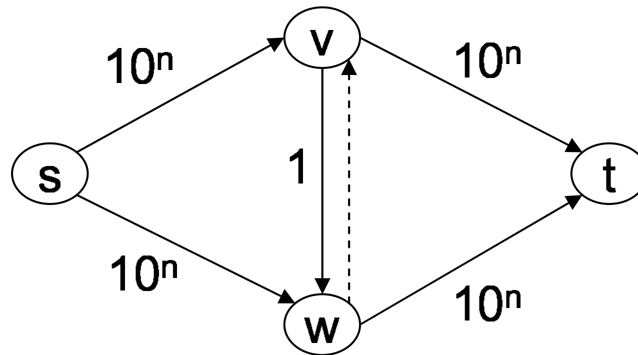
17.1.1 Algorithmus

- Wir schreiben an jede Kante ein Wertepaar $flow(v,w)_{aktuell}/flow(v,w)_{max}$. Zudem erzeugen wir für jede Kante eine Rückwärtskante, die zu Beginn mit 0 initialisiert wird.

- Wir suchen immer wieder einen **zunehmenden Pfad** in dem Netzwerk. Dies kann dadurch geschehen, daß wir eine Erreichbarkeitsanalyse (z.B. mit der Breitensuche gestartet mit dem Startknoten) auf dem Restgraphen durchführen, welches in der Laufzeit $O(|V| + |E|)$ geschehen kann.
- Wir erhöhen entlang des zunehmenden Pfades alle aktuellen Flußwerte um den gefundenen Flußwert des Pfades.
- Dies tun wir solange, bis kein zunehmender Pfad mehr gefunden werden kann. (d.h. der Schnitt mit dem geringsten Fluß (mincut) ist voll ausgelastet.)

17.1.2 Worst Case

Der Worst-Case ist gegeben, wenn eine Kante immer wieder vor und dann die Rückwärtskante wieder zurück gegangen werden muß. Folgendes Beispiel ist ein Worst Case bei dem 10^n Schritte von Nöten sind:



Zuerst wird die Kante von (s,v) genommen und dann über die 1-er-Kante (v,w) gegangen, um über (w,t) dann ins Ziel zu gehen. Im nächsten Schritt wird über (s,w) gegangen, dann über die Rückwärtskante (w,v) und dann über (v,t) zum Ziel. Durch ungeschickte Wahl der Kanten wird das Verfahren sehr ineffinzient, obwohl es schneller gehen könnte.

17.1.3 Optimierung

Die Laufzeit des Algorithmusses, wenn stets ein *kürzester* Pfad gewählt wird ist

$$O(|V| \cdot |E|^2).$$

17.2 Bipartites Matching

Beim Bipartiten Matching geht es darum möglichst viele Verbindungen zwischen Knoten auf der einen Seiten und Knoten auf der anderen Seite herzustellen, wobei ein Knoten höchstens auf einer mit 1 beschrifteten Kante liegen darf. Ein Beispiel für ein bipartites Matchingproblem ist zum Beispiel die Computerzuteilung für verschiedene Aufgaben. Nicht alle Aufgaben lassen sich auf allen Computern lösen. Von einem Aufgabenknoten ziehen wir eine Kante mit der Kapazität 1 zu einem Computer, der diese Aufgabe erfüllen könnte. Dieses Matchingproblem lösen wir, indem wir vor alle Aufgabenknoten einen Startknoten setzen und diesen mit einer 1-er-Kante mit allen Aufgabenknoten verbinden und von jedem Computerknoten auch wieder eine 1-er-Kante zu einem neu hinzugefügten Zielknoten hinzufügen. Das ganze kann man so als ein Netzwerkflußproblem betrachten.

18 Mustererkennung

Bei der Mustererkennung wird in einem **Text** ein **Muster** gesucht. Anwendung ist zum Beispiel eine Textverarbeitung, bei der ein String im Text gesucht wird. Aber es gibt auch andere Anwendungsbereiche, z.B. DNA Analyse oder Virenerkennung.

18.1 Naives Verfahren

Bei dem naiven Verfahren setzen wir das Muster immer um eine Position nach vorne und testen alle Stellen durch, ob kein Mismatch auftritt.
Dies erfordert dann $O(n \cdot k)$ Zeit.

18.2 KMP - Knuth, Morris und Pratt

Beim KMP-Algorithmus geht es darum, daß Muster, welches an den Text gelegt wird, um schon besuchte Zeichen im Text nach vorne zu schieben, bei denen man sicher gehen kann, daß sie nicht mit dem Wort übereinstimmen. Dafür benötigen wir eine Hilfsfunktion, welche einmalig für das Suchwort erstellt wird. In dieser Hilffunktion ist aufgezeichnet, wie groß der Rand des aktuellen schon verglichenen Präfixes des Suchwortes ist, wenn genau an der Stelle ein Fehler auftritt. Beispielsweise ist der Rand von *aba* *a*, da das *a* sowohl vorne als auch hinten vorhanden ist. Der Rand von *acbac* ist *ac*. Der Rand für *abcabfg* ist *ab*, falls nur bis zur 5. Position gelesen wird. In der Hilfsfunktion wird jeweils gespeichert, wie groß der Rand ist. Beispiel:

a	a	b	a	a	b
0	1	0	1	2	3

Der Text wird nun überprüft, indem das Muster darunter gelegt wird. Wenn ein Mismatch auftritt, verschieben wir das Muster um $h(i) + 1$ Stellen nach vorne. Also die Zahl in der Hilfsfunktionstabelle an der Stelle stehend $+1$.
Die Laufzeit dieses Verfahrens ist $O(n + k)$

18.3 Boyer-Moore

δ_1:

Fall	Aktion
1. X kommt in dem Muster rechts neben dem Zeiger vor / Die aktuelle Position hat einen Mismatch, da das Zeichen im Text X ist im Muster aber nicht	Eine Position nach rechts verschieben. Der Worst-Case tritt dann auf, wen wir beispielsweise einen Text aus nur a's haben und das Muster baa in diesem suchen. Jedesmal wird ein Mismatch a≠b erzeugt und es darf nur eine Position weiter gerutscht werden.
2. X kommt in dem Muster links neben dem Zeiger vor. / Im Text steht ein X aber im Muster nicht.	Das Muster wird so verschoben, daß das erste X im Suchstring an der Zeigerposition steht.
3. Es gibt gar kein X im Muster	Wir können das Muster um die ganze Länge des Musters nach rechts verschieben.

δ_2: Diese Heuristik ist dafür da, um den Fall 1 besser zu machen, d.h. um mehr Stellen nach rechts zu gehen.

19 Scanlines

19.1 Sichtbarkeitsproblem für horizontale Objekte

Aktive Linien verwalten wir in einem AVL-Blatt-Baum. Im AVL-Blatt-Suchbaum steht immer die y-Koordinate der Linien. Wir legen für jeden Anfang und jedes Ende einer Linie einen Scanlinestopp an. Kommen wir an den Anfang einer Linie, so aktivieren wir diese, indem wir sie in den AVL-Blatt-Baum einfügen. Wir können die doppelte Verknüpfung der Blätter nutzen, um festzustellen, ob wir direkte sichbare Nachbarn haben. Wenn wir Nachbarn finden, dann geben wir das Nachbarschaftspaar aus. Ähnlich verfahren wir mit dem Ende einer Linie. Wir entfernen sie aus dem AVL-Blatt-Baum und überprüfen danach, ob neue Nachbarschaften entstanden sind und geben diese ggf. aus.
Die Laufzeit hierfür ist $O(n \log n)$, für das Sortieren der Linien, jeweils $O(\log n)$ für das Einfügen und Löschen in den AVL-Blattbäumen an jedem Haltepunkt der Scanline. Macht zusammen die Laufzeit $O(n \log n)$

19.2 Schnittproblem für horizontale und vertikale Objekte

Wir verfahren ähnlich wie oben. Wir fügen Scanlinestopps für jede vertikale Linie ein. Bei jeder vertikalen Linie testen wir, welche Linie im AVL-Blatt-Baum vorhanden sind, die die Linie schneiden. Hierfür kann man wieder die Doppeltverkettung der Blätter gut nutzen.
Die Laufzeit hierfür ist $O(n \log n + k)$, wobei k die Anzahl der Schnitte ist.

19.3 Geometrisches Divide&Conquer für horizontal und vertikal liegende Objekte

Wir unterteilen den Raum, in welchem die Graden enthalten sind, in zwei ungefähr gleich große Räume und führen auf diesen wieder ein Devide und Conquer durch. In unserem Beispiel aus der Vorlesung haben wir den Divide-Schritt nur einmal durchgeführt und dann sofort die Teile „geconquert". Bei der Betrachtung der Teile können folgende Fälle auftreten:

- Die vertikale Gerade schneidet die horizontale Gerade nicht. ⇒ Es gibt keinen Schnittpunkt
- Die vertikale Gerade schneidet die horizontale Gerade. ⇒ Schnittpunkt ausgeben.
- Die vertikale Gerade schneidet die horizontale Gerade, wobei ein Endpunkt der horizontalen Geraden in dem anderen Teil des vorher in zwei Teile zerteilten Raumes ist. ⇒ Schnittpunkt ausgeben.
- Die vertikale Gerade schneidet die horizontale Gerade, wobei der eine Endpunkt noch nicht einmal in dem anderen Teil des Raumes ist, mit welchem der Raum vor dem Cutschritt noch zusammenhing. ⇒ Auch Schnittpunkt ausgeben.

19.4 Schnittproblem für (fast) beliebig liegende Objekte

Wir setzen bei diesem Algorithmus voraus, daß

1. Die Anfangs und Endpunkte der Geraden von Beginn des Algorithmusses an feststehen.
2. Keine Gerade parallel zur y-Achse also genau senkrecht verläuft.

3. Sich nicht mehr als zwei Geraden in ein und demselben Punkt schneiden.

Das Problem ist, daß sich die Sortierung der einzelnen Linien dynamisch ändert. Wir ermitteln beim aktivieren einer Geraden, ob sich diese mit einer benachbarten Gerade schneidet. Wenn wir eine Gerade wieder deaktivieren ermitteln wir, ob die nun neuen Nachbarn sich schneiden. So bekommen wir alle Schnittpunkte heraus, da sich keine zwei Geraden schneiden können, wenn sie keine direkten Nachbarn sind. Den Schnitttest können wir mit Hilfe eines linearen Gleichungssystems machen. Die Laufzeit dieses Verfahrens ist $O(n \log n)$, da die Sortierung der Haltepunkte die Laufzeit des Sortierverfahrens beansprucht ($O(n \log n)$) und an jedem Haltepunkt eine kostante Anzahl an Operationen, nämlich eine Lösch oder Einfügeoperation und ein oder zwei Schnitttests stattfinden.
Das hier vorgestellte Verfahren untersucht übrigens nur einen Schnittpunkt. Es kann sein, daß eine Gerade mehrmals andere Geraden schneidet. Um dies zu überprüfen benötigen wir eine Umsortierung des AVL-Blattbaumes.

19.5 Voronoidiagramme (nur eine kurze Vorstellung)

- Das Voronoidiagramm unterstützt die Suche nach dem Nachbarknoten mit der geringsten Entfernung.
- Die Kanten in diesem Diagramm liegen immer so, daß für benachbarte Knoten die Kante genau in der Mitte liegt. Für zwei benachbarte Knoten sind die Voronoi-Regionen Nachbarn.
- Außen liegende Knoten, für die es keine Nachbarn mehr gibt, heißen unbeschränkte Voronoiregionen. Die Punkte heißen zusammen „Konvexe Hülle”.
- Schnittpunkt die die Voronoikanten miteinander bilden, heißen Voronoiknoten.
- Mittes Divide&Conquer kann ein Voronoidiagramm erstellt werden.

20 Randomisierte Algorithmen

Randomisierte Algorithmen können für die selbe Eingabe unterschiedliche Laufzeit und unterschiedlichen Speicherplatz benötigen.

20.1 Las Vegas & Monte Carlo

Las Vegas	Monte Carlo
für „nein” immer korrekt	für „nein” immer korrekt
für „ja” immer korrekt	für „ja” manchmal[5]falsch
immer korrekt	für „ja” manchmal falsch

20.2 Randomisiertes Quicksort

Wir hatten gesehen, daß Quicksort im Worst Case eine sehr schlechte Laufzeit hat, da das Pivotelement falsch gewählt wird. Haben wir zum Beispiel eine schon vorsortierte Folge und wollen diese noch einmal mit Quicksort sortieren und nehmen dabei immer das erste Element, so haben wir den Worst Case.

[5]aber seltenst

Den Worst Case können wir abwenden, indem wir ein Element aus den möglichen wählbaren per Zufall auswählen. Hierbei kommen wir auf die Laufzeit

$$T_{average}(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}(T_{average}(i-1) + T_{average}(n-i))$$

was wieder auf $O(n \log n)$ hinaus läuft.

20.3 Mustererkennung - Karp&Rabin

Wir benutzen eine Fingerprinttechnik. Wir teilen das Muster durch eine Primzahl. Ebenso teilen wir auch den Text von der Länge des Musters fortlaufend immer durch eine Primzahl. Sobald bei den beiden Divsionen derselbe Rest herauskommt, ist die Wahrscheinlichkeit hoch, daß das Muster mit dem Text an dieser Stelle übereinstimmt. Dieses Verfahren ist aber ein Monte Carlo Verfahren. Es kann bei der Antwort „ja" auch sein, daß das Muster nicht gleich dem Text ist, wenn die beiden Zahlen ungünstig liegen. Um dieses Verfahren in ein Las Vegas verfahren umzuformen, müssen wir, wenn das Monte Carlo Verfahren ein „ja" zurückgibt dieses mit einem naiven Vergleichen noch überprüfen.
Das Monte Carlo Verfahren läuft in $O(n+k)$, wenn man voraussetzt, daß das Rechnen mit den Zahlen nur konstante Kosten verursacht.

?

Zu dem Las Vegas Verfahren haben wir auch eine Laufzeitabschätzung gemacht, in der wir gezeigt haben, mit welcher Wahrscheinlichkeit im Monte Carlo Verfahren ein Fehler auftritt und mit welcher Laufzeit wir dann den Fehler ausbügeln.

20.4 Primzahltest - Miller&Rabin

?

Dieser Primzahltest geht ziemlich weit in die Tiefen der Mathematik hinein.

20.5 Universelles Hashing

Beim Hashing kann es vorkommen, daß alle Werte im Worst-Case auf einen Wert abgebildet werden. Um dieses zu umgehen verwenden wir beim universellen Hashing eine Hashmenge, aus der wir zufällig eine passende Hashfunktion auswählen, die dann das Hashing optimal macht, so daß wir keine oder nur wenige Kollisionen haben.
Eine Hashmenge H heißt universell, wenn es in ihr mindestens soviele unterschiedliche Hashfunktionen gibt, wie es Schlüssel im Schlüsseluniversum gibt.

20.6 Skip-Listen

- Skiplisten sind normale Listen, denen „Überholspuren" aufgesetzt sind, um Listenglieder zu überspringen und so eine schnellere Suche zu fördern. Die „Höhe" eines Listenelementes ist damit definiert, wie viele Skiplisten über diesem Listenelement einen Stopp haben.
- Die perfekte Skipliste ermöglicht es sogar, in der Zeit $O(\log n)$ zu suchen.
- Randomisierte Skiplisten sind Skiplisten für die die Höhe der einzelnen Listenglieder zufällig festgelegt wird.
- Einfüge und Löschoperationen auf Skiplisten sind sehr aufwendig, weil die Liste rekonstruiert werden muß, weshalb nur weniger solcher Operationen auf solchen Listen durchgeführt werden sollten.

Teil II

Informatik IV

0 Allgemeines

0.1 Funktionen

Definition: Totale Funktion: Sei f eine Funktion $A \rightarrow B$. Wenn es für alle $a \in A$ ein $b \in B$ gibt, so heißt die Funktion f total.
Definition: Partielle Funktion: Sei f eine Funktion $A \rightarrow B$. Es existieren $a \in A$, für die kein $b \in B$ existiert: $f(a) = \perp$.[1]

0.2 Äquivalenzrelation

Eine Äquivalenzrelation ist reflexiv, transitiv und symetrisch:

Für die Relation $\Re$	
reflexiv	Für alle $x \in X$ gibt es $x \Re x$
transitiv	Für alle $x, y, z \in X$ gilt $x \Re y$ und $y \Re z \Rightarrow x \Re z$
symetrisch	Wenn für $x, y \in X$ $x \Re y$ gilt, dann gilt auch $y \Re x$

0.3 Formale Sprachen

Eine Sprache besteht aus Wörtern. Hierfür gibt es meistens Regeln, wie Wörter dieser Sprache gebildet werden, manchmal werden die Wörter aber auch explizit angegeben.

leere Sprache	$L = \{\}$ oder auch $L = \emptyset$
Länge des Wortes	Die Länge des Wortes ist die Anzahl der Symbole, die es aus Σ umfaßt.
Leeres Wort	$w_\epsilon = \epsilon$
Inverses Wort	w^R

Konkatenation: $L = L_1 \circ L_1$ bedeutet ($w_1 \in L_1$, $w_2 \in L_2$) $w = w_1 w_2$.
Konkatenation läßt sich mehrfach wiederholen. Beispielsweise bedeutet L^n n mal L:

$$\underbrace{LLL...L}_{n\,Mal}$$

Kleenabschluß: Unter dem Kleenabschluß versteht man

$$L^* = \bigcup_{n \geq 1} L^n$$

Für $\begin{array}{ll} n = 0: & L^0 = \{\epsilon\} \\ n = 1: & L^1 = L \end{array}$

L^+ steht für[2]

$$L^+ = \bigcup_{n \geq 1} L^n$$

[1] d.h. An einigen Stellen ist die partielle Funktion nicht definiert
[2] Genauso wie L^*, nur daß die Menge mit dem leeren Wort ausgeschlossen ist.

1 Berechenbarkeit

1.1 Registermaschinen (kurz: RM)

Registermaschinen-Berechenbar *RM-berechenbar*: Eine partielle Funktion $f : \mathbb{N} \rightarrow \mathbb{N}^k$ wird RM-berechenbar genannt, wenn es eine Registermaschine R gibt, so daß $f = f_R$.

1.1.1 Aufbau und Befehle

Eine Registermaschine besteht aus

Befehlszähler:	Dieser zeigt auf den aktuellen Befehl.
Programmspeicher:	Hier findet man die Befehlszeilen, welche RM-Befehle beinhalten.
Akkumulator:	Das Register, auf dem die RM Operationen ausgeführt werden. Der Akkumulator wird auch mit $c(0)$ bezeichnet.
unendlich viele Register:	Diese nehmen Werte (Variablen) auf Register werden mit $c(Nummer)$ bezeichnet.

Eine Registermaschine verfügt über folgende Befehle, die im Programmspeicher stehen können:

LOAD	Register in Akkumulator laden
STORE	Akkumulator in Register speichern
ADD	Zum Akkumulator ein Register hinzuaddieren
SUB	... subtrahieren
MULT	... multiplizieren
DIV	... dividieren
GOTO	Sprungbefehl. Der Befehlszähler wird verändert.
IF (...⋈...) THEN ...	Bedingte Anweisung
END	Der Befehlszähler wird auf ∞ gesetzt. Das Programm endet.

Zu jedem der fett geschriebenen Befehle kommt eine Variable[3], von denen es drei Arten gibt:

Konstante	#	k
direkte Variable	*nichts*	c(i)
indirekte Variable	$*$	c(c(i))

1.1.2 Kostenmaße

Es existiert sowohl das uniforme als auch das logarithmische Kostenmaß. Weiterhin existiert eine Zeit und eine Platzkomplexität:

t_R^u	uniforme Zeit
t_R	logarithmische Zeit
s_R^u	uniformer Platz
s_R	logarithmischer Platz

[3]Wir haben in Informatik IV nur Ein-Adress-Registermaschinen verwendet. Diese haben dieselbe Mächtigkeit wie Null, zwei oder drei Adressmaschinen.

Die **Zeitkomplexität** wird gemessen an den **abgearbeiteten Befehlen**. Die **Platzkomplexität** an den **besuchten Registerzellen**.
Für das logarithmische Kostenmaß wird die Wortlänge des zu bearbeitenden Wortes n betrachtet. Hierbei gilt für das logarithmische Kostenmaß:

$$L(n) = \begin{cases} 1 & f\ddot{u}r\, n = 0 \\ 1 + \lfloor \log n \rfloor & f\ddot{u}r\, n \geq 1 \end{cases}$$

1.1.3 Effiziente Algorithmen

Als effizient gelten Algorithmen, für die die Laufzeit polynomiell beschränkt ist:

$$T(n) = O(poly(n)) \Leftrightarrow T(n) \leq p(n), wobei\, p(n)\, ein\, Polynom$$

1.2 Turingmaschinen (kurz: DTM oder NTM)

1.2.1 Einbandige „normale" Turingmaschinen (deterministisch)

Eine DTM (deterministische Turingmaschine[4]) ist ein 7er-Tupel
$\tau = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$:

Q	Zustandsmenge
Σ	Eingabealphabet
Γ	Bandalphabet
δ	partielle Übergangsfunktion, der Form $\delta(q, a) = (q, a, move)$
q_0	Startzustand ($q_0 \in Q$)
$\Box$	Blanksymbol ($\Box \notin \Sigma, \Box \in \Gamma$)
F	Endzustandsmenge ($F \subseteq Q$)

Es gibt ein **Band** und einen **aktuellen Zustand** auf den der Zeiger der Turingmaschine zeigt. Ein Raum auf dem Band heißt **Bandzelle** oder **Bandquadrat**.
Eine partielle Übergangsfunktion hat die Form

$$\delta(q_1, a) = (q_2, b, N)$$

Eine Turingmaschine kann

akzeptieren Wenn $q_{aktuell} \in F$. Dabei ist es unerheblich, ob die partielle Funktion δ noch weiterrechnen könnte. Sobald ein Endzustand erreicht ist, ist Ende.

verwerfen Wenn $\delta(q_{aktuell}, Zeichen_{aktuell}) = \perp$. Verwerfen tut sie, wenn der aktuelle Zustand kein Endzustand ist und „es nicht mehr weiter geht".

endlos laufen Wenn sie niemals abbricht.

Fangzustand Die Turingmaschine kann in einen Fangzustand geraten. In einem Fangzustand arbeitet die Turingmaschine dauernd eine Übergangsfunktion folgender Form ab:

$$\delta(q_{aktuell}, Zeichen_{aktuell}) = (q_{aktuell}, Zeichen_{aktuell}, N)$$

Fangzustände können auch formal durch verwerfende Zustände ersetzt werden.

[4]Man schreibt leicht „Touringmaschine". Aber Turingmaschinen kommen gar nicht auf Touren, sondern sie sind einfach nur Turingmaschinen ohne o.

1.2.2 Mehrbandige Turingmaschinen

Bei mehrbandigen Turingmaschinen hat die Übergangsfunktion die Form

$$\delta(q,(a,b)) = (p, < c, L >, < d, R >)$$

Eine Konfiguration wird wie folgt dargestellt:

$$\begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_k \end{pmatrix} q_{aktuell} \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_k \end{pmatrix}$$

1.2.3 Nichtdeterministische Turingmaschinen

Übergangsfunktion:

$$\delta(q,a) = \{(q_1, b_1, X_1), \ldots, (q_k, b_k, X_k)\}$$

Die Konfiguration ist ein **Baum**.

1.2.4 Kosten von Turingmaschinen

Zeitkomplexität:
$t_\tau(w)$= Anzahl der Konfigurationswechsel, die τ bei der Eingabe w durchführt.
Platzkomplexität:
$s_\tau(w)$= Anzahl der Bandzellen, die τ bei der Eingabe w besucht.

Ferner gilt:

$$s_\tau(w) \leq t_\tau(w) + 1$$

,da die Turingmaschine auch sofort abbrechen kann. In diesem Fall würden zwar eine Bandzelle besucht aber keine Zeit verbraucht werden. Es würde stehen $1 \leq 0 + 1$. Man sieht, daß die 1 von Nöten ist.

1.2.5 Simulation von mehrbandigen Turingmaschinen durch einbandige Turingmaschinen

Jede Mehrbandturingmaschine kann durch eine Einbandturingmaschine simuliert werden und umgekehrt.

„$\Leftarrow$”: Ist trivial. Jede Einbandturingmaschine kann man auf einer Mehrbandturingmaschine simulieren, indem man nur ein Band der Mehrbandturingmaschine benutzt.

„$\Rightarrow$”: Man kann aus einer Mehrband-DTM folgendermaßen eine Einband-DTM machen:

- Man erweitert die DTM auf auf $2k$ Bänder. Jeweils auf dem $2k$-ten Band setht ein Zeichen, welches die Zeigerposition zeigt, auf dem $2k-1$-ten Band ist dsa normale Band zu finden.

- Nun erweitert man das Bandalphabet so, daß zu jeder Kombination an Zeichen auf den Bandzellen ein Zeichen in diesem Alphabet vorhanden ist:

  $$\Gamma' = \Gamma \cup (\Gamma \cup \{*\})^{2k}$$

  Weiterhin wird auch die Zustandsmenge Q in Q' verändert. In den Zuständen Q' ist für jeden Zustand Q eine Zustandsmenge vorhanden. In dieser werden die

Zeichen, die unter den Zeigern der Mehrbandturingmaschine stehen gespeichert und dann hinterher eine Bewegung auf allen virtuellen Bändern ausgeführt[5].

- Wir verfahren nun bei der Simulation wie folgt:
  1. Wir suchen alle Zeiger auf dem Band und speichern sie in dem Zustand ab.
  2. Wir verändern das Band gemäß der original Übergangsfunktion.
  3. Wir stellen den Zeiger der Einbandturingmaschine wieder vor den ersten im Band kodierten Zeiger.

Kosten für die Simulation:
Die Anzahl der Konfigurationswechsel der simulierenden DTM τ' ist beschränkt durch ein konstantes Vielfaches der Anzahl der Bandquadrate, die die zu simulierende DTM τ benutzt. Es gilt:

$$T_{\tau'}(n) = C \cdot M_w \cdot N_w$$

wobei M_w **die Konfigurationswechsel der Original-DTM sind und** N_w **die Anzahl der Bandquadrate, die besucht wurden, sowie** C **eine Konstante ist.** Es folgt:$\Rightarrow$

$$T_{\tau'}(n) = O(T_\tau(n)^2)\,,\; wegen\, M_w \cdot N_w$$

Es gilt auch

$$S_{\tau'} = O(S_\tau(n))\,,\; weil\, nur\, hin\, und\, her\, gespult\, wird.$$

1.2.6 Programmierung von DTMs

Hintereinanderschaltung $f_{\tau_1;\tau_2} = f_{\tau_1} \circ f_{\tau_2} = f_{\tau_2}(f_{\tau_1})$
Man vereinigt beide Zustandsmengen[6]. Alle Endzustände der ersten DTM τ_1 ersetzt man durch normale Zustände. Wird ein solcher Zustand erreicht, so leitet die DTM in die Zustandsmenge der zweiten DTM τ_2 über:

$$\delta(q_{F_{\tau_1}}, a) = (q_{0_{\tau_2}}, a, N)$$

Schleifen, bedingte Anweisungen usw.
Wir konstruieren eine DTM, die die Bedingung überprüft. Abhängig von der Ausgabe dieser DTM führen wir in die eine oder die andere DTM mit Hilfe des Zustandsmengenwechsels über.
Unterprogramme
Es gibt Zustände, in denen Kontrollinformationen mitgespeichert werden (z.B. Rücksprungadressen). Lokale Variablen von Unterprogrammen können auf einem Extraband gespeichert werden. Die wirft jedoch Probleme bei der Rekursion auf. Besser ist es, sie nacheinander an Stellen des zweiten Bandes zu speichern und sie mit entsprechenden Sonderzeichen von anderen Unterprogrammvariablen zu trennen.

1.3 Registermaschine $\leftrightarrow$ Turingmaschine

Man kann eine Registermaschine mit einer Turingmaschine simulieren und umgekehrt. **Hiermit führt man auch den Beweis, daß eine Registermaschine mit indirekter Adressierung die gleiche Mächtigkeit wie eine Registermaschine mit direkter Adressierung hat.**

[5] Die Zustände haben die Form $< q, b_1, b_2, \ldots, b_k, c, \ldots >$, wobei $b_i \in \Gamma \cup \{?\}$. Das „?" steht dafür, daß das Zeichen auf dem betreffenden Band noch nicht bekannt ist

[6] Dabei darf natürlich dann kein Zustand mit selben Namen einmal überschrieben werden.

1.3.1 Simulation von Registermaschinen durch Turingmaschinen

Die Zustandsmenge ist: $Q = Q_0 \cup Q_1 \cup \ldots \cup Q_p \cup Q_{p+1}$, wobei p die Anzahl der Programmzeilen der Registermaschine ist. Die Zustandsmenge Q_0 erzeugt die Ausgangskonfiguration auf den Bandzellen, die Zustandsmenge Q_{p+1} stellt die Ausgabe zur Verfügung.
Die Turingmaschine verwendet nun 4 Bänder:

1.Band	Eingabe
2.Band	Registerbelegung der Form $\#\#\#0\#bin(c(0))\#\#1\#bin(c(1))\#\#\ldots$
3.Band	Ausgabe
4.Band	Nebenrechnungen

Wird nun ein Befehl, der in einer Programmzeile steht, abgearbeitet, wird folgendes gemacht *(Am Beispiel von add*24)*:

1. Auf Band 2 wird zuerst an die Stelle 24 gespult, dann wird an die Stelle gespult, wo der Wert aus der Registerzelle 24 hinzeigt. Dieser Wert wird ausgelesen und auf das Nebenrechnungsband (Band 4) gelegt.

2. Der Akkumulator $c(0)$ auf Band 2 wird mit dem Nebenrechnungsband bitweise addiert.

3. Das Ergebnis wird zurück in den Akkumulator kopiert. Die Zustandsmenge wechselt in die nächste Zustandsmenge[7]: $Q_{neu} = Q_{p_{aktuell}+1}$.

Kosten
Jeder Befehl benötigt eine konstante Anzahl an Suchoperationen bzw. arithmetischen Operationen, die ausgeführt werden müssen. Diese laufen in polynomieller Zeit.
Es gilt

$$O(poly(N))$$

Für N gilt

$$N = O(t_R(n_1, \ldots, n_k) + \sum_{i=1}^{k} L(n_i))$$

was darauf zurückzuführen ist, daß die Registermaschine $t_R(n_1, \cdot, n_k)$ Zeit braucht und die Länge des Bandes $\sum_{i=1}^{k} L(N_i)$ ist, über welches gespult wird.
Insgesamt erhalten wir als Laufzeit der Simulation

$$O(poly(t_R(n_1, \cdots, n_k) + \sum_{i=1}^{k} L(n_i)))$$

Insbesondere, wenn die Laufzeit der Registermaschine polynomiell beschränkt ist, gilt für die gesamte Simulation

$$O(poly(m)).$$

1.3.2 Simulation von Turingmaschinen durch Registermaschinen

Allgemeiner Teil
Die Übergangsfunktion wird simuliert, indem wir für jede partielle Funktion eine IF-Abfrage programmieren:

[7] Sofern wir keine Gotobefehle haben

Wenn(Zustand=... und Zeichen=...)
dann{Zustand=...; Zeichen=...; Zeigerbewegung=...;}

1. Möglichkeit: **Durch indirekte Adressierung**
Man benutzt eine linksseitig beschränkte DTM. Diese ist äquivalent zu der unbeschränkten DTM.

Register 1:	aktueller Zustand
Register 2:	*Nummer* des jeweils gelesenen Zeichens (0 ist $\Box$, dann geht es für das Bandalphabet Γ mit 1 weiter)
Register 3:	Position des Zeigers auf dem simulierten Band
Register 4:	Wird freigehalten für Nebenrechnungen
Register 5 und höher:	Das Band der Turingmaschine[8]

Man kann nun die DTM mit Hilfe des „Semibandes" simulieren.

Kosten:
Die Rechenzeit der DTM ansich ist $t_\tau(w)$. Dabei werden maximal $t_\tau(w)+1$ Bandquadrate besucht. Die Operanden der Registermaschinen-Befehle sind deshalb aus dem Bereich $\{0,\ldots,N_w\}$, wobei

$$N_w = \max\{t_\tau(w)+6, |\Gamma|, |Q|\}$$

Somit kostet jeder Konfiguartionswechsel

$$O(\log N_w) = O(\log|Q| + \log|\Gamma| + \log t_\tau(w)) = O(\log t_\tau(w))$$

Es werden $t_\tau(w)$ Konfigurationswechsel ausgeführt. Deshalb sind die Gesamtkosten für das logarithmische Kostenmaß:

$$O(t_\tau(w)\cdot\log t_\tau(w))$$

unter dem uniformen:

$$O(t_\tau(w)\cdot 1)$$

Hinzu kommen unter dem uniformen Kostenmaß

$$O\left(\sum_{i=1}^{k} L(n_i)\right) = O(|w|)$$

und unter dem logarithmischen Kostenmaß

$$O\left(\sum_{i=1}^{k} L(n_i)^2\right) = O(|w|^2)$$

Schritte zur Erstellung der Ausgangskonfiguration.

Daraus folgt: Die gesamte Simulation läuft unter:

$$O(poly(t_\tau(w)+|w|))$$

2. Möglichkeit: **Mit zwei Stacks**
Wir benutzen zwei Keller. Auf den einen tun wir den linken Teil des Bandes, der links vom Schreib/Lesekopf zu finden ist, auf den anderen den rechten Teil. Weiterhin

[8] Das Register j enthält die $j-6$te Bandzelle

benutzen wir eine Variable für die aktuelle Bandzelle[9].
Wir können die Keller implementieren, indem wir Bitverschiebungen[10] machen. Wir können einen Wert des Stacks durch die Basis mit Rest teilen und erhalten die oberste Zelle oder aber wir können eine Zelle auf den Stack schieben, indem wir den Wert des Stacks mit der Basis multiplizieren und den Wert der Zelle addieren:
Rauf:

$$stack = stack \cdot basis + wert$$

Runter:

$$wert = stack \bmod basis$$

1.3.3 Simulation von Turingmaschinen mit Turingmaschinen - Universelle Turingmaschinen

Eine Turingmaschine kann man als ein einzelnes Wort kodieren. Universelle Turingmaschinen benutzen als Eingabe eine Turingmaschine, die sie simulieren.
Auf Band 1 der universellen Turingmaschine steht die kodierte Turingmaschine und das darauf angewandte Eingabewort. Folgendermaßen kann man die Übergangsfunktion kodieren:

$$\#\#bin(i)\#bin(j)\#bin(I)\#bin(J)\#bin(X)\#\#\ldots$$

Auf Band 2 speichern wir den aktuellen Zustand. Auf Band 3 wird der aktuelle Bandinhalt gespeichert. Band 4 benutzen wir für Nebenrechnungen. Eine solche universelle Turingmaschine kann die eingegebene Turingmaschine simulieren.

2 Entscheidbarkeit

2.1 Wortproblem

Gegeben ist eine Sprache $L \subseteq \Sigma^*$. Das Wortproblem ist die Frage, ob $w \in L$.

2.2 Charakteristische Funktion

2.2.1 Totale charakteristische Funktion

$\chi_L : \Sigma^* \to \{0, 1\}$,

$$\chi_L(w) = \begin{cases} 1 & falls\, w \in L \\ 0 & sonst \end{cases}$$

2.2.2 Partielle charakteristische Funktion

$\chi_L : \Sigma^* \to \{0, 1\}$,

$$\chi_L(w) = \begin{cases} 1 & falls\, w \in L \\ \perp & sonst \end{cases}$$

2.3 Entscheidbarkeit

2.3.1 Semientscheidbarkeit

Sei $L \subseteq \Sigma^*$ eine Sprache. L ist semientscheidbar, wenn es eine DTM τ gibt, für die gilt

$$L(\tau) = L$$

[9] Alternativ kann man auch auf diese verzichten. Es gibt Implementierungen, die direkt mit der Variable vom Stack arbeiten.
[10] Im Binärsystem. In anderen Systemen verschieben wir um andere Basen.

d.h. Für alle Worte der Sprache L akzeptiert die DTM τ. *Wenn die partielle charakteristische Funktion turingberechenbar ist, ist die Sprache semientscheidbar.*

2.3.2 Entscheidbarkeit

Sei $L \subseteq \Sigma^*$ eine Sprache. L ist entscheidbar, wenn es eine DTM τ gibt, für die gilt

$$L(\tau) = L$$

und für alle Worte $w \notin L$ verwirft die DTM τ. *Wenn die totale charakteristische Funktion turingberechenbar ist, ist die Sprache entscheidbar.*

2.3.3 Nicht entscheidbar $\leftrightarrow$ Unentscheidbar

Die Sprache $L \subseteq \Sigma^*$ wird unentscheidbar genannt, wenn sie nicht entscheidbar ist, d.h. es gibt keine DTM τ, die für alle $w \in L$ akzeptiert und für alle $w \notin L$ verwirft.

2.3.4 Nicht semientscheidbar

Eine Sprache $L \subseteq \Sigma^*$ wird nicht semientscheidbar genannt, wenn es keine DTM gibt, für die gilt $L(\tau) = L$.

2.4 Zusammenhang zwischen Entscheidbarkeit und Semientscheidbarkeit

2.4.1 L entscheidbar, dann auch $\overline{L}$ entscheidbar

L entscheidbar, dann auch $\overline{L}$ entscheidbar.
Wir benutzen eine DTM τ', die die DTM τ simuliert. Diese vertauscht die beiden Ausgaben „JA" und „NEIN".
O.b.d.A. können wir annehmen, daß für alle $q \in F$ (für die DTM τ) gilt $\delta(q, a) = \perp$
Wir definieren nun die partielle Übergangsfunktion von τ' folgendermaßen:

$$\delta'(q,a) = \begin{cases} \delta(q,a) & falls\, \delta(q,a) \neq \perp und\, q \notin F\, (normale\, \ddot{U}berg\ddot{a}nge) \\ \perp & falls\, q \in F\, (Verwerfen\, wenn\, die\, DTM\, \tau\, akzeptiert) \\ (q', a, N) & falls\, \delta(q,a) = \perp (Akzeptieren,\, wenn\, \tau\, verwirft) \end{cases}$$

Die Endzustandsmenge von τ' ist $F' = \{q'\}$.

2.4.2 L und $\overline{L}$ semientscheidbar $\Leftrightarrow$ L entscheidbar

Es genügt die Implikation „$\Rightarrow$" zu zeigen:
Wir nehmen die beiden Turingmaschinen, die L und $\overline{L}$ semientscheiden und schalten beide parallel. Eine der beiden Maschinen wird akzeptieren. Akzeptiert τ_L so geben wir „JA" aus. Akzeptiert $\tau_{\overline{L}}$ so geben wir „NEIN" aus.
Wir können die beiden Turingmaschinen parallel schalten, indem wir eine Zweibandturingmaschine verwenden, wobei jede Turingmaschine ein Band besitzt und die Zustandsmenge[11] immer hin und her schaltet.

[11] Anmerkung: Die Menge der Zustände der simulierenden Gesamt-DTM beträgt: $n * m$, wobei n die Menge der Zustände der ersten DTM ist und m die Menge der Zustände der zweiten DTM.

2.5 Rekursive Aufzählbarkeit

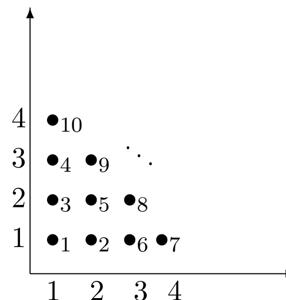
2.5.1 Rekursiv aufzählbar → Semientscheidbar

Rekursiv aufzählbar heißt, daß entweder die Sprache leer ist, oder daß es ein Verfahren gibt, welches die Sprache komplett aufzählt. Dabei können einzelne Elemente mehrfach aufgezählt werden.

Das Semientscheidungsverfahren kann man folgendermaßen auf die rekursive Aufzählbarkeit zurückführen: Wenn man fragt, ob ein Wort w in der Sprache enthalten ist, so kann man das Aufzählungsverfahren durchlaufen lassen. Wenn das Wort enthalten ist, so berechnet das Aufzählungsverfahren irgendwann das Wort. Dann können wir „JA” zurückgeben. Andernfalls läuft das Aufzählungsverfahren endlos und es wird eben halt nicht abgebrochen.

2.5.2 Semientscheidbar → Rekursiv aufzählbar

Die Sprache wird mit Hilfe des Semientscheidungsverfahrens rekursiv aufgezählt. Wir laufen allerdings in das Problem, daß, wenn für ein Wort w, daß nicht in der Sprache enthalten ist, das Semientscheidungsverfahren nicht terminiert, die Sprache nicht weiter aufzählt wird und der Algorithmus dann an dieser Stelle „hängt”.
Wir bedienen uns hierfür eines kleinen Tricks:
Wir stellen an das Semientscheidungsverfahren die Frage, ob nach k Schritten, die DTM, die semientscheidet, akzeptiert. Wenn nach k Schritten nicht akzeptiert wird, brechen wir einfach ab. Wir benutzen eine Funktion $\mathbb{N} \to \mathbb{N} \times \mathbb{N}$, die alle Tupel (n, k) aufzählt und zwar so, daß n und k gleichmäßig erhöht werden. Beispielsweise folgendermaßen[12]:



Die Zahlen an den Punkten geben hierbei die Reihenfolge an, mit der die Punkte „besucht” werden. Die x-Achse kann beispielsweise n sein, die y-Achse k. Wir erhalten bei diesem Verfahren beispielweise die Paare

$$(1,1),(2,1),(1,2),(1,3),(2,2),(3,1),(4,1),(3,2),(2,3),(1,4)\ldots$$

Nun bekommen wir für jedes Wort, welches in der Sprache enthalten ist, irgendwann ein „JA”, da die DTM ja nach endlich vielen Schritten (k) akzeptieren muß. Wir bekommen dann danach mehrmals für dieses Wort ein „JA” und können es in die Sprache einreihen. Dies ist aber egal, da die Worte ja auch mehrmals auftreten dürfen.

2.6 Halteproblem

2.6.1 Die Sprache des Halteproblems und des speziellen Halteproblems

Die Sprache H des Halteproblems ist

$$H = \{w\#x : w, x \in \{0,1\}^* : \tau_w \, hält \, für \, das \, Eingabewort \, x \, an\}$$

[12] Dieses Verfahren ist das Verfahren zum Beweis, daß die rationalen Zahlen abzählbar sind.

Die Sprache H' für das spezielle Halteproblem ist

$$H' = \{w; w \in \{0,1\}^* : \tau_w \, hält \, für \, die \, Eingabe \, w \, an\}$$

2.6.2 Satz: Das Halteproblem ist unentscheidbar

Die Sprache H des Halteproblems ist unentscheidbar, d.h. es gibt keine Turingmaschine die H entscheidet.

2.6.3 Informeller Beweis des Halteproblems mittels Halteproblemtabelle

Wir erstellen eine Tabelle.

- Auf der x-Achse tragen wir jede mögliche Eingabe auf, die ein Algorithmus haben kann.
- Auf der y-Achse tragen wir alle möglichen Algorithmen auf[13].

Jede Halteproblemtabellenzelle enthält ein

- **„JA”**: Wenn der angegebene Algorithmus für die Eingabe **terminiert**.
- **„NEIN”**: Wenn der angegebene Algorithmus für die Eingabe nicht terminiert, d.h **endlos läuft**.

Der Halteproblemalgorithmus muß sich komplementär zur Tabelle verhalten. Wenn

- Der Algorithmus terminiert, also ein „JA” dort steht, muß er endlos laufen.
- Der Algorithmus endlos läuft, also ein „NEIN” dort steht, muß er verwerfen.

Nun betrachten wir das Komplement der Diagonalen der Tabelle. Da der Halteproblemalgorithmus irgendwann selbst in der Tabelle auftaucht[14], muß er selbst entscheiden, ob er anhält. Wenn der der Halteproblemalgorithmus endlos läuft, muß er gleichzeitig terminieren, wenn er terminiert, muß er endlos laufen[15].

2.6.4 Das Halteproblem ist Semientscheidbar

Wir können eine universelle Turingmaschine τ konstruieren, die die Turingmaschine τ' simuliert. Der Algorithmus, den τ' ausführt, soll überprüft werden, ob er bei der enthaltenen Eingabe terminiert. Die Turingmaschine τ akzeptiert, sobald der τ' die Berechnung beendet hat. Wenn aber τ' endlos läuft, dann akzeptiert auch τ nie. $\Rightarrow$ Semientscheidbarkeit

2.6.5 Spezielles Halteproblem

Die Sprache H' des speziellen Halteproblems lautet:

$$H' = \{w; w \in \{0,1\}^* \tau_w \, hält \, bei \, der \, Eingabe \, w \, an\}$$

Definitionen:

- τ' ist die DTM von der angenommen wird, daß sie die Sprache H' entscheidet.

[13] Jeder Algorithmus ist formulierbar in endlich vielen Programmzeilen.

[14] Der Halteproblemalgorithmus ist ein Algorithmus und in der Tabelle sind alle Algorithmen aufgelistet.

[15] In der Tabelle müßten ein „J” und ein „N” gleichzeitig stehen.

- τ ist die DTM des Halteproblemalgorithmusses:
  - Läuft τ' endlos, so akzeptiert τ.
  - Akzeptiert τ', so läuft τ endlos.

Sei $\tau = \tau_w$. D.h. der Halteproblemalgorithmus wird auf sich selbst angewendet.

1.Fall:

	τ läuft endlos
$\Leftrightarrow$	τ' akzeptiert
$\Leftrightarrow$	$w \in H'$
$\Leftrightarrow$	$\tau_w = \tau$ hält an
	Widerspruch!

2.Fall:

	τ akzeptiert
$\Leftrightarrow$	τ' läuft endlos
$\Leftrightarrow$	$w \notin H'$
$\Leftrightarrow$	$\tau_w = \tau$ läuft endlos
	Widerspruch!

2.6.6 Reduktion

L heißt auf K reduzierbar ($L \leq K$), wenn es eine Funktion $f(x)$ gibt, so daß gilt

$$x \in L \Leftrightarrow f(x) \in K$$

d.h. daß es eine totale Übergangsfunktion gibt, die aus jeder Eingabe für L eine Eingabe für K macht und daß L und K für die gleiche (für K vorher transformierte) Eingabe akzeptieren, verwerfen oder endlos laufen.

$$L\,reduzierbar\,auf\,K \Leftrightarrow L \leq K$$

- Ein Algorithmus für L wird erstellt, indem ein Algorithmus K benutzt wird.
- Weiterhin kann auch L als ein Spezialfall von K betrachtet werden. Dies ist keine Reduktion mehr, sondern eine Einbettung in das Problem K.

Beispiele für den ersten Punkt:

- H wird mit Hilfe von H_ϵ gelöst (H wird so umgeformt, daß es H_ϵ ist):

$$H \leq H_\epsilon$$

- SAT wird auf $3SAT$ reduziert. Der SAT Algorithmus wird mit Hilfe von $3SAT$ gelöst. Jede aussagelogische Formel in SAT kann in eine aussagelogische Formel in $3SAT$ mit dem selben „Output" überführt werden:

$$SAT \leq_{poly} 3SAT$$

Beispiele für den letzten Punkt:

- Das spezielle Halteproblem wird in das allgemeine Halteproblem eingebettet. Hiermit zeigen wird, daß auch das allgemeine Halteproblem unentscheidbar ist.

$$H' \leq H$$

- $3SAT$ wird auf SAT zurückgeführt. $3SAT$ ist ein Spezialfall von SAT. Somit gilt diese polynomielle Reduktion, da wir sogar überhaupt gar keine Umformung durchführen müssen.

$$3SAT \leq_{poly} SAT$$

2.6.7 Reduktionsprinzip

$$L \leq K$$

K entscheidbar $\Rightarrow$ L entscheidbar
K semientscheidbar $\Rightarrow$ L semientscheidbar
K unentscheidbar $\Rightarrow$ L unentscheidbar

2.6.8 Unentscheidbarkeit des Halteproblems

Wir zeigen, daß das spezielle Halteproblem auf das Halteproblem reduzierbar ist $H' \leq H$. Offenbar kann man das spezielle Halteproblem mit der Funktion

$$f(x) = x\#x$$

in das allgemeine Halteproblem überführen (da die DTM τ ja auf sich selbst angewendet wird). Somit ist dann genauso wie das spezielle Halteproblem auch das allgemeine Halteproblem unentscheidbar, da das spezielle Halteproblem ein Spezialfall des allgemeinen Halteproblem ist[16] . Wir erhalten: Die Sprache H

$$H = \{w\#x : w, x \in \{0,1\}^* \, \tau_w \, hält\, für\, die\, Eingabe\, x\, an\}$$

ist unentscheidbar.

2.6.9 Die Nicht-Semientscheidbarkeit des Komplements des Halteproblems: Satz

Die Sprache $\overline{H}$ (das Komplement des Halteproblems)

$$\overline{H} = \{w\#x : w, x \in \{0,1\}^* : \, \tau_w \, hält\, für\, die\, Eingabe\, x\, nicht\, an\}$$

ist nicht semientscheidbar.

2.6.10 Die Nicht-Semientscheidbarkeit des Komplements des Halteproblems: Informell

- Zunächst beinhaltet die Komplementsprache $\overline{H}$ auch alle Worte, die nicht der Form $w\#x(w, x \in \{0,1\}^*)$ sind. Diese schließen wir jedoch aus.

[16] Wir machen hier nicht den Rückschluß, daß man ein unbekanntes Problem L auf ein bekanntes Problem K zurückführt, sondern wir sehen das spezielle Halteproblem als einen Spezialfall des Halteproblems an. Deshalb schreiben wir $H' \leq H$, obwohl H eigentlich noch gar nicht bekannt ist.

- Das spezielle Halteproblem war unentscheidbar (haben wir bewiesen), d.h. daß es keine DTM gibt, die H' entscheidet, mit anderen Worten, es gibt keine DTM, die alle Wörter $w \notin H'$ verwirft.
- Das Komplement des speziellen Halteproblems ist noch schärfer. Da das Verhalten der DTM[17] genau umgedreht sein muß, ist es nicht sicher, ob die DTM alle Wörter $w \in \overline{H}'$ akzeptiert.

$\Rightarrow$ nicht semientscheidbar

2.6.11 Die Nicht-Semientscheidbarkeit des Komplements des Halteproblems: Formal

Angenommen $\overline{H}'$ ist semientscheidbar. Wir definieren eine DTM τ, so daß gilt:

$$L(\tau) = \overline{H}'$$

Unsere DTM τ akzeptiert oder läuft nur endlos. Verwerfen tut sie nicht.
Sei $w = code(\tau)$.

Wir setzen weiterhin $\tau = \tau_w$: Es gilt:

	$w \in \overline{H}'$
$\Leftrightarrow$	$w \in L(\tau)$
$\Leftrightarrow$	Berechnung für $\tau = \tau_w$ akzeptierend
$\Leftrightarrow$	$\tau = \tau_w$ hält bei der Eingabe w an
$\Leftrightarrow$	$w \notin L(\tau)$

Widerspruch

2.7 Andere unentscheidbare Probleme

2.7.1 Satz von Rice

Sei S eine nichtleere, echte Teilmenge der Menge aller partiellen berechenbaren Funktionen $f : \{0,1\}^* \to \{0,1\}^*$. Dann ist die Sprache

$$L_S = \{w \in \{0,1\}^* : \tau_w \, berechnet \, eine \, partielle \, Funktion \, f \in S\}$$

unentscheidbar.

Beweis: Wir beweisen dies, indem wir zeigen:

$$\overline{H_\epsilon} \leq L_S \, und \, H \leq H_\epsilon$$

Kleine Vorbemerkung: H_ϵ ist das Problem, ob eine Turingmaschine bei leerem Wort anhält. Dies is also ein Spezialfall des Halteproblems. $\overline{H_\epsilon}$ ist das Komplementproblem. Alle Codewörter $w = code(\tau_w)$, bei der die Turingmaschine τ_w bei der Eingabe ϵ *nicht* anhält, sind in der Sprache enthalten.
Jetzt geht's los: Wir führen nur den Fall aus, daß $f_\perp \in S$.
$f_\perp$ ist die Funktion, die überall undefiniert ist.
1. $\overline{H_\epsilon} \leq L_S$**:**

Wir wählen eine beliebige Funktion $f_{f \notin S}$, die nicht in S liegt. Wir definieren eine Funktion f_w folgendermaßen:

[17] Diese gibt es eigentlich gar nicht, aber wir nehmen einfach mal an, daß es sie gibt.

$$f_w = \begin{cases} f_\perp & falls\, \tau_w\, bei\, der\, Eingabe\, \epsilon\, nicht\, anhält \\ f_{f \notin S} & sonst \end{cases}$$

Um die Funktion f_w zu berechnen, simulieren wir zuerst τ_w. Wenn τ_w anhält, dann kann es erst weitergehen. Das Problem aber, ob τ_w anhält ist alleine schon unentscheidbar (das beweisen wir erst weiter unten[18]), deshalb ist die ganze Geschichte unentscheidbar.

2. $H \leq H_\epsilon$**:**

Um mit dem Halteproblem für das leere Wort das Halteproblem zu lösen, bauen wir einfach das Eingabewort für das normale Halteproblem schon in die Turingmaschine hinein. Dies können wir zum Beispiel tun, indem die Turingmaschine für das Halteproblem mit leerem Wort vorerst das Wort auf dem Band erst erzeugt. Aus τ_x mit der Eingabe w wird so $\tau_{x\#w}$ mit der Eingabe ϵ für das Halteproblem mit leerem Wort.

2.8 Nichtdeterminismus (Ergänzung)

2.8.1 Nichtdeterministische Entscheidbarkeit

Eine NTM τ entscheidet eine Sprache L falls gilt:
1. $L(\tau) = L$ und
2. Der Berechnungsbaum für jedes mögliche Eingabewort $w \in \{0,1\}^*$ ist endlich, d.h. jedes Wort wird nach endlicher Zeit entweder akzeptiert oder verworfen.

2.8.2 NTM → DTM

Aus jeder NTM kann man eine DTM machen. Man geht dabei den Berechnungsbaum mit einer Breitensuche durch[19].

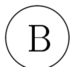
3 Komplexitätsklassen

3.1 Unterschiedliche Problemvarianten

3.1.1 Vorstellung mit Beispiel

Als Beispiel wird hier das CP[20] genommen.

1. Entscheidungsproblem: Gibt es eine Lösung mit dem Wert k? *Beispiel: Gibt es eine Clique der Größe k?*
2. Optimierungsproblem Typ 1: Bestimme den Wert k der optimalen Lösung *Beispiel: Wieviele Knoten umfaßt die größte Clique im Graphen?*
3. Optimierungsproblem Typ 2: Gebe die optimale Lösung zurück! *Beispiel: Rückgabe der Knoten der größtmöglichen Clique im Graphen*



Ein weiteres Beispiel: Rucksackproblem (RP)

1. *Gibt es eine Rucksackfüllung mit dem Gewinn k?*
2. *Was ist der Gewinn k einer optimalen Rucksackfüllung? (Gibt den maximalen erzielbaren Gewinn k zurück!)*
3. *Welche Gewichte sind in der optimalen Rucksackfüllung enthalten?*

[18] Halteproblem

[19] Hier darf man keine Tiefensuche verwenden. Das kann unter Umständen ins Auge gehen, da ein Berechnungsbaum ins Unendliche ragt, während ein anderer schon lange akzeptiert hat, man aber im unendlichen Berechnungsbaum immer tiefer geht und nie zum Ende kommt.

[20] Cliquenproblem

3.1.2 Überführung ineinander am Beispiel des Cliquenproblems

Es sieht so aus, als ob dies drei völlig unterschiedliche Probleme seien. Dennoch snd sie in vielen Fällen gleichschwierig. Beim CP sind sie gleichschwierig. Wir zeigen dies[21]:

- Es ist klar, daß man das Optimierungsproblem von Typ 2 auf Typ 1 zurückführen kann und Typ 1 auf das Entscheidungsproblem zurückgeführt werden kann. Dies tut man, indem man einfach „Teillösungen" der zurückgegebenen Lösung zurückgibt.

- Man kann das Entscheidungsproblem in das Optimierungsproblem vom Typ 1 umwandeln, indem man das k von 1 starten läßt und, solange das Entscheidungsproblem „JA" zurückgibt, um 1 erhöht. Sobald wir ein „NEIN" vom Entscheidungsproblem bekommen, wissen wir, daß das k des „JA"-s davor die größte Clique ist[22].

- Das Optimierungsproblem vom Typ 1 kann man in das Optimierungsproblem vom Typ 2 umwandeln, indem man den Graphen nimmt und immer eine Kante aus dem Graphen entfernt. Nachdem man diese entfernt hat, läßt man den Optimierungsproblemalgorithmus des Typs 1 noch einmal laufen. Gibt er immer noch das gleiche k zurück, so hat man die Clique nicht zerstört, und man kann die Kante dauerhaft entfernt lassen. Wenn das k kleiner wird, wird die Kante wieder dem Graphen hinzugefügt und als „besucht" markiert. Dies wird solange gemacht, bis alle Kanten im Graphen als „besucht" markiert sind. Nun haben wir die Knoten der Clique.

3.2 Zeitkomplexität

3.2.1 SAT - Guess and Check Methode

Guess and Check-Methode

Bei der Guess and Check-Methode wird eine Kombination mit Hilfe des „Orakels" einfach geraten. Dabei wird so geraten, daß direkt das richtige Ergebnis geraten wird. Gibt es kein richtiges Ergebnis, so wird ein falsches Ergebnis ausgegeben. Das von dem nichtdeterministischen Guess-Bereich erratene Ergebnis wird im deterministischen Check-Bereich noch einmal auf Richtigkeit überprüft. Hier stellt sich heraus, ob es wirklich richtig ist oder ob die Guess-Methode ein falsches Ergebnis zurückgegeben hat, weil es kein richtiges gibt.

SAT

α: Die zu überprüfende Formel, wobei z.B. $\alpha \in \{0, 1, \wedge, \neg, (,)\}$

Wir raten zunächst mit Hilfe des „Orakels" eine Belegung der Formel. Ist das SAT-Problem erfüllbar, bekommen wir eine erfüllbare Belegung zurück. Wenn nicht irgendeine Belegung.

Dann können wir in n Schritten überprüfen, ob die geratene Belegung tatsächlich eine wahre Belegung ist.

Wir benötigen polynomielle Laufzeit für beide Teilalgorithmen, da die Anzahl der Literale durch $|\alpha|$ beschränkt ist.

$$O(poly(|\alpha|)$$

[21] Hinweis: Für das RP ist nicht alles überführbar.

[22] Dies folgt daraus, daß jede l-er Clique auch eine $(l-1)$-er Clique ist.

3.2.2 Primzahlen

Unter dem deterministischen. logarithmischen Kostenmaß hat der Primzahltest die Laufzeit $\Theta(n \log n)$, unter dem nichtdeterministischen Kostenmaß die Laufzeit $O(n)$. Dies kommt dadurch zustande, daß wir mit Guess-Methode einen möglichen Teiler raten. Es wird ein Teiler der zu testenden Zahl geraten, wenn es einen gibt. Andernfalls wird irgendeine Zahl geraten. Nachdem wir den möglichen Teiler geraten haben, können wir testen, ob dieser geratene Teile wirklich Teiler der zu testenden Zahl ist.

3.2.3 Cliquenproblem

Kodierung des Graphen
Man kann den Graph G folgendermaßen codieren:

$$code(G) = \#\#bin(n)\#\#bin(von)\#bin(nach)\#\#$$

Kodierung des CP
Man kann das gesamte CP folgendermaßen kodieren:

$$code(G)\#\#\#bin(k)$$

Eingabegröße des CP
Die Eingabegröße ist also

$$\underbrace{|code(G)|}_{Graph} + \underbrace{3}_{\#} + \underbrace{\log k}_{bin(k)} \quad \underbrace{}_{nur\,der} = \underbrace{\Theta(m \log(n))}_{Graph\,ist\,relevant}$$

Laufzeit des CP

Für ein deterministisches Verfahren können wir für die untere Schranke die Laufzeit mit

$$\Omega\left(\binom{n}{k}\right)$$

festlegen. Dies beruht auf der Tatsache, daß wir $\binom{n}{k}$ Möglichkeiten haben, eine k-er Clique aus allen Knoten auszuwählen[23]. Wir testen jede Clique in konstanter Zeit.
Es gilt die Abschätzung:

$$\Omega\left(\binom{n}{k}\right) \geq 2^{\frac{n}{2}} = \sqrt{2}^n$$

$\Rightarrow$ exponentielle Laufzeit[24]

3.2.4 Zeitkomplexitätsklassen

Es gibt folgende Klassen im deterministischen und im nichtdeterministischen:

DTIME	NTIME
DSPACE	NSPACE

3.2.5 P,NP,EXPTIME

$$P = PTIME = \bigcup_{k \geq 1}(n^k)$$
$$NP = NPTIME = \bigcup_{k \geq 1}(n^k)$$
$$EXPTIME = \bigcup_{c \geq 1} \bigcup_{k \geq 1}(c^{n^k})$$

[23]Kombinatorik
[24]Das Nichtdeterministische Verfahren für das CP hat polynomielle Laufzeit.

3.2.6 NP $\subseteq$ EXPTIME

NP liegt in EXPTIME
Dies können wir beweisen, indem wir die NTM τ', die den NP-Algorithmus ausführt mit einer deterministischen DTM τ in EXPTIME simulieren.
Es liegt ein Berechnungsbaum vor, welchen wir mit Hilfe einer Breitensuche oder einem Preorderdurchlauf durchgehen. Da die nichtdeterministische Turingmaschine τ' polynomiell beschränkt ist, ist die maximale Tiefe des Baumes polynomiell beschränkt.
! Der Verzweigungsgrad c jedes Knotens dieses Baumes ist

$$c = |\delta(q,a)| \leq |\{L,N,R\}| \cdot |\Sigma| \cdot |Q|$$

somit gilt für die Knotenanzahl

$$O(c^{p(|x|)})$$

Also ist die Simulation von τ' exponentiell beschränkt
$\Rightarrow$ NP $\subseteq$ EXPTIME

3.2.7 Polynomialzeit akzeptierende NTMs

Auf wenn nur Wörter $w \in L$ von der Turingmaschine τ in $O(p(|x|))$ akzeptiert werden, können wir die gesamte Turingmaschine beschränken, indem wir alle irgendwann verwerfenden oder endlosen Berechnungen abschneiden, wenn sie über polynomielle Laufzeit hinaus laufen.

3.3 Platzkomplexität

3.3.1 PSPACE

Eine Sprache $L \subseteq \Sigma^*$ ist genau dann $\in PSPACE$, wenn gilt:

$$S_\tau(n) = O(poly(n))$$

3.3.2 SAT $\in$ PSPACE

Man kann mit Hilfe eines Backtrackingalgorithmus[25] die NTM für SAT simulieren. Die Simulation belegt nur polynomiellen Platz, nämlich soviel Platz wie die Länge der Formel[26]: $O(|\alpha|)$. *(Dieser Beweis ist ähnlich dem von $NP \subseteq PSPACE$)*

3.3.3 In-Place Acceptance $\in$ PSPACE

In-Place-Acceptance bedeutet, daß die Turingmaschine τ

- in $s_\tau(x) \leq |x|$ Platz
- akzeptiert oder verwirft.

Wir benutzen eine universelle Turingmaschine U, die mitzählt, wieviel Bandzellen τ benutzt:

- Wenn die Turingmaschine τ akzeptiert, akzeptiert die Turingmaschine U auch.

[25] Tiefensuche
[26] Wie ist die Länge der Formel noch einmal definiert? Raussuchen!

- Kommt sie über die erlaubte Anzahl $|x|$, der Länge des Eingabewortes, so verwirft sie.
- Kommt eine Konfiguration doppelt vor (daran kann man erkennen, daß sie Turingmaschine endlos laufen wird), so verwirft sie.
  Es liegt nun nahe, daß wir die alten Konfigurationen in der Turingmaschine einfach speichern. Dies würde allerdings zuviel Platz kosten. Statt dessen speichern wir die aktuelle Konfiguration und simulieren die gesamte Turingmaschiene τ noch einmal von Anfang an, generieren dabei alle alten Konfigurationen und vergleichen.

3.3.4 NP ⊆ PSPACE

Wir benutzen eine Tiefensuche, um den Berechnungsbaum von der Turingmaschine, welche in NP liegt zu durchgehen.
Die Rekursionstiefe ist durch ein Polynom $p(|x|)$ beschränkt (Definition von NP). Bei einem PTIME-Algorithmus darf der Berechnungsbaum höchstens polynomielle Tiefe haben, um die Rekursion zu speichern. Das ist hier erfüllt.

3.3.5 PSPACE ⊆ EXPTIME

Die Turingmaschine in $PSPACE$ ist im Platz durch ein Polynom $p(|x|)$ beschränkt. Er ergibt sich für die Gesamtanzahl der Konfigurationen[27] der Turingmaschine:

$$\underbrace{p(|x|)}_{Positionen\,Lese/Schreibkopf} \cdot \underbrace{|Q|}_{m\ddot{o}gl.\,Zust\ddot{a}nde} \cdot \underbrace{|\Gamma|^{p(|x|)}}_{m\ddot{o}gl.\,Bandinhalte}$$

Wir können eine Konstante c finden, so dass gilt:

$$c^{q(|x|)} > p(|x|) \cdot |Q| \cdot |\Gamma|^{p(|x|)}$$

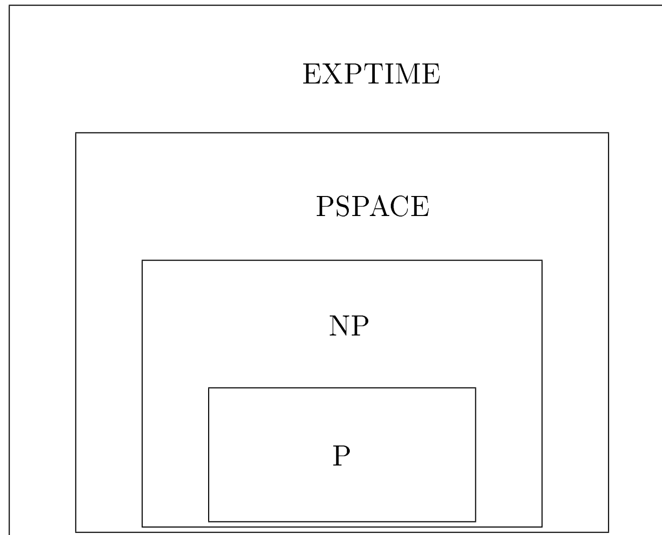
⇒ exponentiell beschränkt

3.3.6 Satz von Savitch: $PSPACE = NPSPACE$

Haben wir keinen Beweis (glücklicherweise) gemacht
Wir haben nur gesagt, daß nach diesem Satz gilt:

$$PSPACE = NPSPACE$$

3.3.7 Übersicht über die Komplexitätsklassen



[28]

4 NPC Beweise

4.1 Die große Frage der Informatiker P=NP oder P≠NP?

Läßt sich jedes Problem, daß sich **nichtdeterministisch** in polynomieller Zeit lösen läßt, auch in **deterministischer** polynomieller Zeit lösen?
Allgemein wird angenommen, daß $P \neq NP$. Dies ist allerdings nicht bewiesen.

4.2 Polynomielle Reduzierbarkeit

L heißt auf K polynomiell reduzierbar, wenn es eine total berechenbare Funktion f gibt, so daß gilt

- $x \in L \Leftrightarrow f(x) \in K$
- f ist in polynomieller Zeit berechenbar.

4.3 NP-hart, NP-vollständig

4.3.1 NP-hart

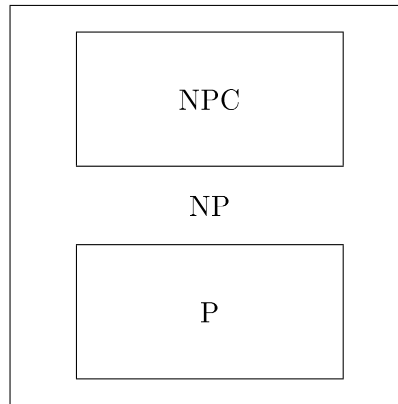
Eine Sprache K heißt NP-hart, falls für alle Sprachen $L \in NP$ gilt:

$$L \leq_{poly} K$$

4.3.2 NP-vollständig

Eine Sprache K heißt NP-vollständig ($K \in NPC$), falls sie NP-hart ist und $K \in NP$.

[27] Also Möglichkeiten, wie die Zeichen auf dem Band stehen können, Möglichkeiten wie der Zeiger stehen kann und Möglichkeiten in welchem Zustand sich die Turingmaschine momentan befindet.
[28] Die Sache mit den Kreisen funktionierte leider nicht so, wie ich es gerne wollte.



4.3.3 Sätze

- Sobald für ein NPC-Problem ein deterministischer Polynomialzeitalgorithmus angegeben werden kann, liegen wegen der NP-Härte alle Probleme aus NP in P!

$$NP = P$$

- Die NP-Härte eines Problems kann nachgewiesen werden, indem man ein schon bewiesenes NP-hartes Problem auf dieses Problem reduziert. Die NP-Vollständigkeit kann somit nachgewiesen werden, indem man einen nichtdeterministisches Polynomialzeitalgorithmus findet (Guess-and-Check-Methode) und die NP-Härte dann nachweist.

4.4 Aussagelogik

Sprache für eine Formel:

$$\alpha ::= true|x_i|(\alpha \wedge \beta)|(\neg\alpha)$$

Morgan-Regeln:

$$\alpha \vee \beta = \neg((\neg\alpha) \wedge (\neg\beta))$$
$$\alpha \wedge \beta = \neg((\neg\alpha) \vee (\neg\beta))$$

Zur Wiederholung:

$\wedge$ bedeutet „und”,
$\vee$ bedeutet „oder”.

Länge einer Formel: Die Länge einer Formel α ist die Anzahl der Opperatoren in α und wird mit $|\alpha|$ bezeichnet. Operatoren sind $\vee$,$\wedge$ und $\neg$.
Notationen:

$$\bigwedge_{1 \leq i \leq n} a_i = a_1 \wedge a_2 \wedge \ldots \wedge a_n$$

$$\bigvee_{1 \leq i \leq n} a_i = a_1 \vee a_2 \vee \ldots \vee a_n$$

Definitionen:

$$\bigwedge_{i=\emptyset} a_i = true$$

$$\bigvee_{i=\emptyset} a_i = false$$

Tautologie:
Eine Formel ist dann eine Tautologie, wenn sie für alle Belegungen erfüllt ist.

4.5 Satz von Cook

4.5.1 Satz

SAT ist NP-Vollständig

4.5.2 Beweis

1. $SAT \in NP$
   Mittels Guess-and-Check-Methode läßt sich eine Belegung von α bestimmen und anschließend in polynomieller Zeit überprüfen.

2. Jedes NP-Problem läßt sich auf SAT in polynomiell reduzieren:

$$L = L(\tau) = p(|x|)$$

Wir führen nur 2. durch. 1. ist trivial

Da τ in NP läuft ist der Berechnungsbaum für die akzeptierende Berechnung polynomiell beschränkt:

$$t(\tau) \leq p(|x|)$$

Somit ist auch die Anzahl der Bandquadrate, die besucht werden, polynomiell beschränkt. Wir müssen nur Bandquadrate betrachten, welche im folgenden Bereich sich befinden[29]

$$-p(|x|) \leq i \leq p(|x|)$$

Wir setzen nun eine Formel α für den SAT-Algorithmus aus der Turingmaschine τ folgendermaßen zusammen:

$$\alpha = A \wedge \ddot{U}b \wedge E \wedge R$$

wobei

A:	Anfangsbedingungen
Üb:	Übergangsfunktion
E:	Endbedingungen
R:	Randbedingungen

[30]

[29]Dies ist wichtig, da sonst unsere Formel α unendlich lang werden muß. Wir können also diese Beschränkung gut gebrauchen.

[30]Kurze Zusammenfassung der nun folgenden Definitionen:

A	Stellt Anfangsbedingungen her
Üb1	Die Übergangsfunktion
Üb2	Alle anderen Zellen werden beim Konfigurationswechsel nicht verändert
E	Abbruchbedingungen
R1	Immer nur ein Zustand aktuell
R2	Immer nur ein Bandquadrat aktuelles
R3	Immer nur ein Zeichen in einer Bandzelle

Anfangsbedingung A:

$$A = \underbrace{(zustand(0) = q_0}_{Startzustand\, q_0} \wedge \underbrace{(position(0) = 0)}_{Startpos.\, auf\, Zelle\, 0} \wedge \underbrace{\bigwedge_{1\le i\le |x|} (band(0, i-1) = a_i)}_{\substack{Eingabewort\, x \\ x = a_1 a_2 \ldots a_n}} \wedge$$

$$\underbrace{\bigwedge_{-p(|x|)\le i\le p(|x|);\, i\notin(1\le i\le |x|)} (band(0,i) = \Box)}_{Alle\, anderen\, Zellen\, leer}$$

Übergangsfunktion Üb:

$$\ddot{U}b = \ddot{U}b1 \wedge \ddot{U}b2$$

- **Üb1**: Beschreibt die Übergangsmöglichkeiten von Zeitpunkt t nach $t+1$.
- **Üb2**: Sagt aus, daß alle anderen Bandzellen unverändert bleiben.

Üb1:
Wir erweitern alle verwerfenden Konfigurationen auf die Laufzeit $p(|x|)$, so daß sie uns keine Probleme mehr bereiten. Dies tun wir, indem wir in der partiellen Übergangsfunktion einen Fangzustand für alle verwerfenden Übergänge hinzufügen[31].

$$\ddot{U}b1 = \bigwedge_{\substack{t < p(|x|) \\ -p(|x|) \le i \le p(|x|) \\ q,a}} (((zustand(t) = q) \wedge (position(t) = i) \wedge (band(t,i) = a)) \rightarrow \Delta(t,q,i,a))$$

wobei

$$\Delta(t,q,i,a) = \bigvee_{(q',b,X)\in\delta'(q,a)} ((zustand(t{+}1) = q') \wedge (position(t{+}1) = i{+}X) \wedge (band(t{+}1,i) = b))$$

(Anmerkung:δ' ist die neue Übergangsfunktion, die wir oben aus δ erzeugt haben.)
dabei ist $X = \{-1, 0, 1\}\, f\ddot{u}r\, L, N, R$

Üb2:

$$\ddot{U}b2 = \bigwedge_{i<p(|x|));\ i,a} ((position(t) \neq i) \wedge (band(t,i) = a) \rightarrow (band(t+1,i) = a))$$

Endbedingungen:
Die Endbedingungen sagen, daß wenn ein akzeptierender Zustand erreicht wird „true" zurückgegeben wird für „akzeptieren":

$$E = \bigvee_{q\in F} (zustand(p(|x|)) = q)$$

[31] Wir würden sonst mit der Teil-Formel, die wir jetzt definieren, bei einer verwerfenden Berechnung nicht terminieren, sondern weiterrechnen und vielleicht dann doch irgendwann aktzeptieren, was nicht geht.

Randbedingungen:

$$R = R1 \wedge R2 \wedge R3$$

R1: Garantiert, daß zu jedem Zeitpunkt nur ein Zustand aktuell ist.

$$R1 = \bigwedge_t \bigvee_q ((zustand(t) = q) \wedge \bigwedge_{q' \in Q \setminus \{q\}} (zustand(t) \notin q'))$$

R2:[32] Garantiert, daß der Schreibkopf auf genau nur einem Bandquadrat steht.

$$R2 = \bigwedge_t \bigvee_{i \in I} ((position(t) = i) \wedge \bigwedge_{i' \in (-p(|x|) \leq i \leq p(|x|)) \setminus \{i\}} (position(t) \notin i'))$$

R3: Garantiert, daß zu jedem Zeitpunkt nur ein Zeichen $a \in \Gamma$ in der Bandzelle i steht.

$$R3 = \bigwedge_t \bigwedge_{i \in I} \bigvee_{a \in \Gamma} ((band(t,i) = a) \wedge \bigwedge_{a' \in \Gamma \setminus \{a\}} (band(t,i) \neq a'))$$

4.5.3 Korrektheit

Bei dieser Reduktion „stecken" wir die ganze Turingmaschine in eine SAT-Formel. Wir berechnen so die ganze Turingmaschine quasi gleichzeitig. Die Berechnung geht anschaulich durch die ganzen Minterme hindurch. Wenn wir eine akzeptierende Berechnung haben, so haben wir in allen Mintermen verknüpft mit $\wedge$ „true".

4.5.4 Kosten

Teil der Formel α	Kosten
A	$p(n)$
E	$p(n)^0 = 1$
Üb1	$p(n)^2$
Üb2	$p(n)^2$
R	$p(n)^3$
all in all	$p(n)^3$

Die Gesamtkosten der Konstruktion sind $p(n)^3$ und somit polynomiell beschränkt.

4.6 Prinzipielle Techniken für NP-Vollständigkeitsbeweise

Man halte sich vor Augen:

$$L \leq_{poly} K$$

4.6.1 Spezialisierung

Spezialisierung ist die einfachste Form des Nachweises. Hierbei ist das zu reduzierende Problem L ein Spezialfall des Problems K und kann in K „eingebettet" werden[33]. Ein Beispiel hierfür ist $3SAT \leq_{poly} SAT$.

[32] R2 und R3 ähneln R1 sehr

[33] Diese Argumentationsweise funktioniert manchmal nicht. Mir ist es schleierhaft, warum man durch Spezialisierung polynomielle Reduzierbarkeit nachweisen kann, da zum Beispiel 2SAT auch ein Spezialfall von 3SAT ist, aber 2SAT $\in$ P und 3SAT $\in$ NP. 2SAT läßt sich reduzieren, indem man einfach ein Literal verdoppelt und schon hat man 3SAT. Man kann aber getrost ein Spezialproblem L auf ein Problem K reduzieren und damit die Unentscheidbarkeit zeigen.

4.6.2 Lokale Ersetzung

Die Eingabe von L wird in kleinere Einheiten von K „zerstückelt" und der Algorithmus K mehrmals laufen gelassen. Beispielsweise wird bei der Reduktion[34] SAT $\leq_{poly}$ 3SAT, SAT in 3SAT Formeln umgeformt, in dem die Formel mit Hilfe der De-Morgan-Regeln verändert wird.

4.6.3 Transformation mit verbundenen Komponenten

Diese Art des Beweises ist die am schwierigsten zu verstehende. Die Eingabe für K wird transformiert. Aus einer Eingabe für K entsteht also eine ganz andere Eingabe für L[35]. Eine solche Transfortmation ist z.B $L(\tau) \leq_{poly} SAT$ (Satz von Cook) oder $3SAT \leq_{poly} CP$.

4.7 Einige NP-Vollständigkeitsbeweise

4.7.1 SAT $\leq_{poly}$ 3SAT (lokale Ersetzung)

3SAT ist das Erfüllbarkeitsproblem mit der Voraussetzung, daß in der Formel α höchstens 3 Literale pro Klausel sein dürfen.

Es ist klar, daß man für den NP-Vollständigkeitsbeweis genauso wie bei SAT die Guess-And-Check-Methode bei 3SAT anwenden kann.

Wir können SAT auf 3SAT polynomiell reduzieren:

$$SAT \leq_{poly} 3SAT$$

1.Schritt
Mit Hilfe der De-Morgan-Regeln

$$\neg(a \wedge b) \equiv (\neg a) \vee (\neg b)$$

$$\neg(a \vee b) \equiv (\neg a) \wedge (\neg b)$$

und der Regel der doppelten Verneinung

$$a = \neg\neg a$$

erstellen wir einen Syntaxbaum aus der Eingabeformel α, in dem die Negationen nur in den Blättern vorhanden sind. Der so entstandene Syntaxbaum ist offensichtlich äquivalent zu der Eingabeformel α.
2.Schritt
Wir formen den so entstandenen Syntaxbaum um.

Dafür geben wir einem jeden Verknüpfungsknoten einen Namen, wobei v_0 der Wurzelverknüpfungsknoten ist.

Nun kommt ein Äquivalenzoperator $\leftrightarrow$ ins Spiel. Wir definieren denselben folgendermaßen[36]:
$v \leftrightarrow v_L\, op\, v_R$ bedeutet:
v ist 1, wenn $v_L\, op\, v_R \equiv 1$
v ist 0, wenn $v_L\, op\, v_R \equiv 0$
Wir können auch folgendes Konstrukt erstellen:

$$(v \leftrightarrow v_L\, op\, v_R)$$

[34]Das sehen wir später genauer.
[35]Natürlich gibt es auch hier gewisse Transformationsregel
[36]op ist ein Operator also entweder $\vee$ oder $\wedge$

Dies bedeutet, daß wenn der rechte und der linke Teil äquivalent sind, die Formel 1 ist, wenn beide Teile nicht äquivalent sind, dann ist der Wert der Formel 0. Wir können uns dies an einem Beispiel[37] klar machen:

v	v_L	v_R	$(v \leftrightarrow v_L \wedge v_R)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

!

Aufgrund dieser Wahrheitstabellen[38] können wir nun die Formeln α'_v konstruieren[39]:

- $(v \leftrightarrow v_L \vee v_R) \equiv (v \vee \neg v_L) \wedge (\neg v \vee v_L \vee v_R) \wedge (v \vee \neg v_R)$
- $(v \leftrightarrow v_L \wedge v_R) \equiv (\neg v \vee v_L) \wedge (\neg v \vee v_R) \wedge (v \vee \neg v_L \vee \neg v_R)$

Wir können für jeden Verknüpfungsknoten v des Syntaxbaumes eine solche Formel α'_v erstellen. Diese fügen wir zusammen:

$$\alpha' = \left(\bigwedge_v \alpha'_v \right) \wedge v_0$$

In den obigen Formel bemerken wir, daß wir höchstens immer 3 Literale pro Klausel in der Formel stehen haben[40]. Die Formeln α'_v sind also 3KNF Formeln. Somit ist auch α' eine 3KNF Formel. Wir können uns klar machen, daß $\alpha' \equiv \alpha$.

B

Beispiel hierfür noch machen

4.7.2 3SAT $\leq_{poly}$ CP (Transformation)

3SAT $\rightarrow$ CP:
Wir erzeugen den Graphen für das CP folgendermaßen:

- Für jede Klausel mit jeweils 3 Literalen erzeugen wir auch 3 Knoten.
- Wir zeichnen jeweils eine Kante zwischen allen Literalen zweier Klauseln. Einzige Ausnahme ist hierbei, daß die beiden Literale komplementär zueinander sind.
- Wir zeichnen niemals eine Kante zwischen den Literalen ein und derselben Klausel.

Wenn es eine Clique der Größe k gibt, wobei k die Anzahl der Klauseln in 3SAT ist, so ist die Formel erfüllbar.

Beispiel

---

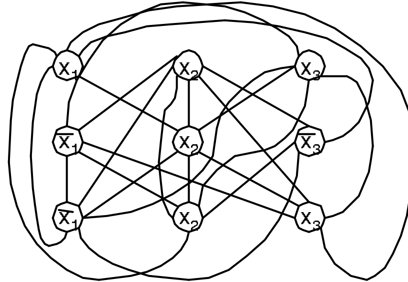
[37] Dies steht auf Seite 944 im Cormen: Introduction to Algorithms

[38] Für $(v \leftrightarrow v_L \vee v_R)$ gibt es ja auch noch eine. Die zweite Formel ist übrigends die Formel, für die wir oben das Beispiel gemacht haben.

[39] Dies sind nicht die einzigsten beiden Formeln. Es sind nur die für das Beispiel im Skript(???). Auf jeden Fall ist die Wahrheitstabelle richtig.

[40] Wenn wir weniger als drei haben, können wir ein weiteres Literal ergänzen, indem wir ein Literal der Klausel verdoppeln.

$$
\begin{array}{llllll}
 & x_1 & \vee & x_2 & \vee & x_3 \quad \text{1.Klausel} \\
\wedge & \overline{x_1} & \vee & x_2 & \vee & \overline{x_3} \quad \text{2.Klausel} \\
\wedge & \overline{x_1} & \vee & x_2 & \vee & x_3 \quad \text{3.Klausel}
\end{array}
$$



Da es eine Dreier-Clique gibt, ist die aussagelogische Formel für eine Belegung wahr.

4.7.3 3SAT $\leq_{poly}$ RP (Transformation sowie auch Spezialisierung)

Gliedert sich in 2 Teile:

- SUBSUM $\leq_{poly}$ RP (Spezialisierung)
- 3SAT $\leq_{poly}$ SUBSUM (Transformation)

Teil 1:
Für die NP-Vollständigkeit von RP genügt es nachzuweisen, daß $SUBSUM \in NPC$, da $SUBSUM$ ein Spezialfall von RP ist. Um $SUBSUM$ auf RP zu reduzieren[41] setzen wir das Gewicht für jedes Element gleich 1.
Die Frage von $SUBSUM$ ist es nun:

Gibt es eine Teilmenge aller Elemente mit dem Gewinn $p = q$?

Teil 2:
Es gilt zu beweisen[42]:

$$3SAT \leq_{poly} SUBSUM$$

Sei die $3SAT$-Formel von der folgenden Form:

$$\alpha = \bigwedge_{1 \leq i \leq k} (A_{i,1} \vee A_{i,2} \vee A_{i,3})$$

Die i-te Klausel hat folgende Form:

$$\gamma_i = A_{i,1} \vee A_{i,2} \vee A_{i,3}$$

Wir setzen den zu erzielenden Gewinn gleich[43]

$$q = \underbrace{44\ldots44}_{\substack{Anzahl\\ der\\ Klauseln}} \underbrace{11\ldots11}_{\substack{Anzahl\, der\\ verschiedenen\, Literale\, x_j}}$$

[41] $SUBSUM \leq_{poly} RP$
[42] Wir erstellen einen Algorithmus für $3SAT$ mit Hilfe von $SUBSUM$.
[43] Der zu erzielenden Gewinn für 3 Klauseln und 3 Literale wäre also 444111 (In Worten: vierhundertvierundvierzigtausendeinhundertundelf)

Wir erstellen für jedes Literal x_j vier Gewinne für $SUBSUM$[44]:

$$v_j = b_{j,1}b_{j,2}\dots b_{j,i}\dots b_{j,k} \quad \underbrace{00\dots}_{\substack{j-1\\ mal}} \quad 1 \quad \underbrace{\dots 00}_{\substack{k-j-1\\ mal}}$$

Dabei stellt $b_{j,k}$ das Vorkommen von x_j in der k-ten Klausel da. $b_{j,k} \in \{0,1,2,3\}$
Dasselbe für $\overline{x_j}$:

$$\overline{v_j} = \overline{b_{j,1}}\,\overline{b_{j,2}}\dots\overline{b_{j,i}}\dots\overline{b_{j,k}} \quad \underbrace{00\dots}_{\substack{j-1\\ mal}} \quad 1 \quad \underbrace{\dots 00}_{\substack{k-j-1\\ mal}}$$

v_j und $\overline{v_j}$ garantieren, daß kein Literal $true$ und $false$ gleichzeitig ist.
Nun erstellen wir noch die beiden Gewinne c_i und d_i für jedes Literal x_j zum Auffüllen, welche an j-ter Stelle eine 1 bzw. eine 2 stehen haben:

$$c_i = 00\dots010\dots0 \;\; 00\dots0\dots0$$

$$d_i = 00\dots020\dots0 \;\; 00\dots0\dots0$$

Aus den so entstandenen Gewinnen läßt sich nun genau der Rucksack voll ausschöpfen ($\sum p_i = q$), wenn die $3SAT$-Formel erfüllt ist.

B

Beispiel
Gegeben sei folgende aussagelogische Formel:

$$\begin{array}{rcccccl} & x_1 & \vee & x_2 & \vee & \overline{x_3} & \text{1.Klausel} \\ \wedge & x_1 & \vee & \overline{x_2} & \vee & x_3 & \text{2.Klausel} \\ \wedge & x_1 & \vee & \overline{x_2} & \vee & \overline{x_3} & \text{3.Klausel} \\ \wedge & \overline{x_1} & \vee & x_2 & \vee & x_3 & \text{4.Klausel} \end{array}$$

Der SUBSUM-Gewinn für 4 Klauseln und 3 verschiedene Literale ist

4444111

Wir erzeugen folgende Gewinne

Klauselnr.	1234		Im Rucksack
$v_1 =$	1100	001	•
$\overline{v_1} =$	0011	001	
$v_2 =$	1001	010	•
$\overline{v_2} =$	0110	010	
$v_3 =$	0101	100	
$\overline{v_3} =$	1010	100	•
$c_1 =$	1000	000	•
$d_1 =$	2000	000	
$c_2 =$	0100	000	•
$d_2 =$	0200	000	•
$c_3 =$	0010	000	•
$d_3 =$	0020	000	•
$c_4 =$	0001	000	•
$d_4 =$	0002	000	•

Eine gültige Belegung mit der wir genau den richtigen Gewinn treffen ist also

$$v_1 + v_2 + \overline{v_3} + c_1 + c_2 + d_2 + c_3 + d_3 + c_4 + d_4$$

[44] Also, wenn wir 3 unterschiedliche Literale haben (x_1, x_2, x_3), dann haben wir $4 \cdot 3 = 12$ Gewinne

4.7.4 0/1 Integer Linear Programming (Transformation)

Bei 0/1 Linear Programming ist ein lineares Gleichungssystem $A \cdot x \geq b$ mit ganzzahligen Koeffizienten gegeben. Gefragt ist, ob es einen Lösungsvektor x gibt, dessen Komponenten 0 oder 1 sind[45], so daß die Gleichung $A \cdot x \geq b$ wahr ist.

1. 0/1 ILP ist in NP
   Mittel Guess-and-Check-Verfahren läßt sich der Vektor x ermitteln und anschließend überprüfen.

2. $3SAT \leq_{poly} 0/1 - ILP$
   Wir reduzieren $3SAT$ in polynomieller Zeit auf $0/1ILP$, d.h wir erzeugen für $3SAT$ einen Algorithmus mit Hilfe von $0/1ILP$.

Sei die Formel α von $3SAT$ wieder der Form

$$\alpha = \bigwedge_{1 \leq i \leq k} (A_{i,1} \vee A_{i,2} \vee A_{i,3})$$

Die Matrix A ist nun $2n$ Zellen breit und $2n + k$ Zellen hoch, wobei n die Anzahl der unterschiedlichen Literale und k die Anzahl der Klauseln in α ist. Wir verwenden für das O/1-ILP in den einzelnen Zeilen der Matrix für „true" eine 1 für „false" eine 0.

- Die ersten $2n$ Ungleichungen stellen sicher, daß ein und dasselbe Literal nicht gleichzeitig wahr und falsch sein kann, bzw. auch eines der beiden sein muß und nicht gar nichts sein kann[46]:
  $(2 \cdot j - 1)$te Formel:
  $$x_j + \overline{x_j} \geq 1$$
  $(2 \cdot j)$te Formel:
  $$-x_j - \overline{x_j} \geq -1$$

- Die letzten k Ungleichungen sagen, daß pro Klausel ein Literal wahr sein muß:
  $$A_{i,1} + A_{i,2} + A_{i,3} \geq 1$$

Es ist leicht einsichtig, warum dieses $0/1ILP$-Problem genau dann auch erfüllt ist, wenn die $3KNF$ von $3SAT$ auch erfüllt ist.

4.7.5 Weitere NPC-Beweise

Weitere NPC-Beweise gibt es auf Aufgabenzettel 6.

[45] D.h. Maskierung der Zeilen in der Matrix mit dem Vektor x. Nur dort wo eine 1 vorkommt, wird auch die Spalte genommen.

[46] x_j und $\overline{x_j}$ können, wie man an den Formeln sieht nicht gleichzeitig 1 oder 0 sein, sondern müssen paarweise verschieden sein.

4.8 coNP

Sei C eine Komplexitätsklasse

$$coC$$

besteht aus allen Sprachen L, deren Komplement $\overline{L}$ in C liegt.
Für alle deterministischen Komplexitätsklassen gilt

$$coC = C$$

also z.B.

$$coP = P$$

$$coPSPACE = PSPACE$$

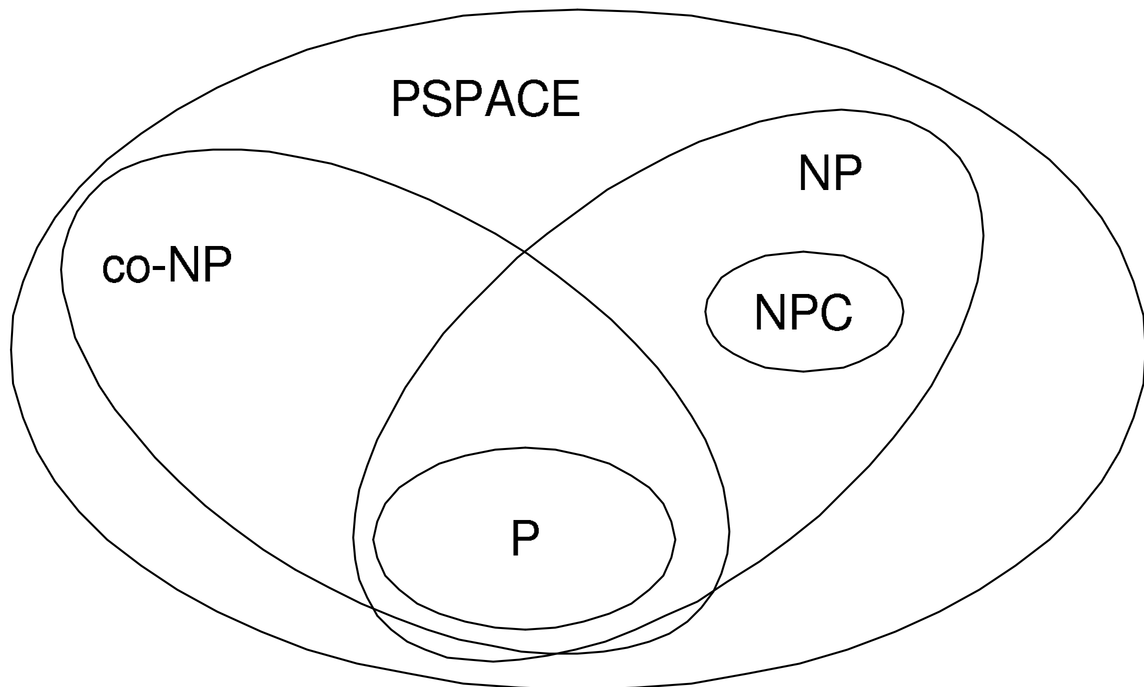
, da sich die Ausgabe einfach vertauschen läßt.
Dies läßt sich für nichtdeterministische Sprachen nicht durchführen, da das „Raten” der Belegung nicht mitspielt. Wir können zum Beispiel das komplementäre Problem von SAT $\overline{SAT}$ konstruieren, daß immer dann „JA” zurückgibt, wenn die eigegebene Formel nicht erfüllbar ist und „NEIN” zurückgibt, wenn sie erfüllbar ist. Schmeißen wir nun unser „Orakel” an, so gibt es uns eine Belegung, wo die Formel nicht erfüllbar ist, falls es eine gibt. Wir wissen dann jedoch nicht, ob die Formel für alle Belegungen unerfüllbar ist. Wir können also mit der einfachen Umkehr des Verfahrens kein korrektes Verfahren bekommen.

4.9 PSPACE-Vollständigkeit

Definition ähnlich der NP-Vollständigkeit:

- Das Problem muß in PSPACE liegen.
- Alle anderen in PSPACE liegenden Probleme müssen polynomiell auf dieses Problem reduzierbar sein.

4.10 Zusammenfassung



5 Praktischer Einsatz von formalen Sprachen: Compiler

Ein Compiler besteht aus den hier angeführten Bestandteilen:

5.1 Lexikalische Analyse: Scanner

Die Eingabe wird in Grundsymbole unterteilt. Hierbei gibt es 4 verschiedene Arten:

Schlüsselwörter	z.B: IF THEN ELSE GOTO...
Spezialsymbole	z.B: $\{, \}, ;, (,), <, > \ldots$
Identifier	Vom Benutzer definierte Namen für Variablen, Programmteile ...
Literale	Werte für Variablen z.B. 123 oder "Hallo"

5.2 Syntaktische Analyse: Parser

Die Ausgabe der lexikalischen Analyse wird in einem Syntaxbaum umgewandelt. Hierbei fallen auch die Fehler in der Syntax auf, wie zum Beispiel vergessene Klammern. Die Sprache der getestet wird ist eine (deterministische) kontextfreie Sprache. Z.B.:

$$\begin{array}{lcl} \text{Zahl} & \rightarrow & \text{Ziffer}|\text{Ziffer Zahl} \\ \text{Ziffer} & \rightarrow & „0“|„1“|„2“|\ldots|„9“ \end{array}$$

5.3 Semantische Analyse und Codeerzeugung

In der semantischen Analyse werden viele Dinge gemacht, auf die wir hier nicht konkret eingehen. Es werden maschinenspezifische Anpassungen vorgenommen und hinterher nach der Analyse wird dann auch der Code erzeugt.

6 Grammatiken

6.1 Definitionen

6.1.1 Notation

Jeder Identifier besteht aus einem Buchstaben gefolgt von beliebig vielen Buchstaben oder Ziffern.

$$\Leftrightarrow$$

$$L_{Idf} = L_{Buchstabe} \circ (L_{Buchstabe} \cup L_{Ziffer})^*$$

$$\Leftrightarrow$$

Idf	→	Buchstabe|Buchstabe A
A	→	Buchstabe|Buchstabe A|Ziffer|Ziffer A
Buchstabe	→	„A"|„B"|...|„Z"|„a"|...|„z"
Ziffer	→	„0"|...|„9"

Man kann die Ableitung eines Wortes mit Hilfe der Symbole $\Rightarrow$ und $\Rightarrow^*$ darstellen. Dabei steht

- $\Rightarrow$ für die Anwendung von nur einer Regel und
- $\Rightarrow^*$ für die Anwendung beliebig vieler Regeln.

6.1.2 Definition Grammatik

Eine Grammatik ist ein 4er-Tupel $G = (V, \Sigma, P, S)$ wobei

- V: endliche Menge von Nichtterminalen[47]
- Σ: endliche Menge von Terminalsymbolen. Dabei gilt $V \cap \Sigma = \emptyset$.
- P: Produktions bzw. Regelsystem. Einzelne Teile des Systems heißen „Produktion" oder „Regel".
- S: Das Startsymbol ($S \in V$)

6.1.3 Die durch eine Grammatik erzeugte Sprache

Die Sprache enthält alle durch die Grammatik G erzeugbaren Wörter:

$$L(G) = \{w \in \Sigma^* : S \Rightarrow^* w\}$$

6.1.4 Äquivalenzen von Grammatiken

Zwei Grammatiken G_1 und G_2 heißen äquivalent, wenn sie dieselbe Sprache erzeugen:

$$L(G_1) = L(G_2)$$

[47] Nichtterminale kann man auch Variablen nennen.

6.2 Chomsky Hierarchie

6.2.1 Tabellarische Übersicht

Typ	Bezeichnung	Eigenschaften
Typ 0	keine Bezeichnung	beliebig
Typ 1	kontextsensitiv	Für jede Relation $u \rightarrow v$ gilt $\|u\| \leq \|v\|$, d.h. daß das Wort nicht wieder abnehmen darf. Ausnahme ist hier die ϵ-Sonderregel. $S \rightarrow \epsilon$ ist erlaubt
Typ 2	kontextfrei	Links muß ein einzelnes Nonterminal stehen.
Typ 3	regulär	Die Regeln im Regelsystem müssen die folgende Form haben • $A \rightarrow \epsilon$ • $A \rightarrow a$ • $A \rightarrow aA$ wobei A Nonterminal und a Terminal ist.

6.2.2 Beispiele

- **Aussagelogische Formel für SAT** *Typ 2 - kontextfrei*

$$S \rightarrow true|x_i|(S \wedge S)|(\neg S)$$

  Mittels De-Morgan kann aus „oder" „und" gemacht werden.

- **Grammatik für die Sprache** $L = \{a^n b^n c^n\}$ $n \geq 1$ *Typ 1 - kontextsensitiv*

S	→	aSBC|aBC	Aufbau des Wortes
CB	→	BC	umsortieren
aB	→	ab	umwandeln in Terminale
bB	→	bb	umwandeln in Terminale
bC	→	bc	umwandeln in Terminale
cC	→	cc	umwandeln in Terminale

B

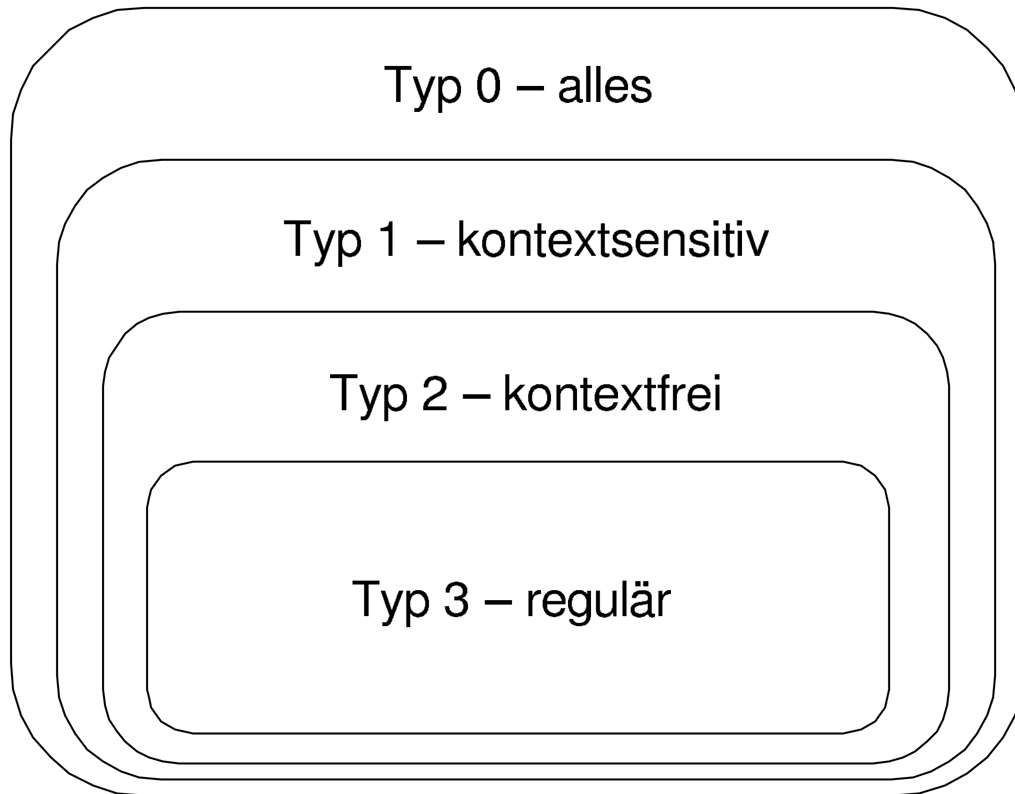
Hier gibt es einiges an interessanten Beispielen auf Aufgabenzettel Nr 7. Es ist gut diese sich noch einmal anzuschauen. In der Prüfung sind Beispiele für die jeweiligen Sprachtypen zu kennen.

6.3 Inklusion der Chomskyhierarchie

Eine Sprache eines höheren Typs ist auch immer gleichzeitig eine Sprache eines Typs darunter.

$$Typ\,3 \subset Typ\,2 \subset Typ\,1 \subset Typ\,0$$

6.4 Grafische Darstellung der Chomskyhierarchie



Hinweis: Die Grafik ist nicht ganz richtig. Typ 0 Sprachen sind nicht alle Sprachen, sondern nur fast alle Sprachen. Beispielsweise sind unentscheidbare Sprachen, wie zum Beispiel das Halteproblem, nicht Typ 0 Sprachen.

6.5 Transformation um die ϵ-Freiheit bis auf $S \rightarrow \epsilon$ in Typ 1 herzustellen

Es gibt eine kleine technische Panne bei

$$Typ\,2 \subset Typ\,1$$

$$kontextfrei \subset kontextsensitiv$$

Bei Typ 2 darf auf der rechten Seite in jeder beliebigen Regel das ϵ stehen. Bei Typ 1 jedoch nicht. Wir können jedoch jede kontextsensitive Grammatik in der die Regel $|u| \leq |v|$ durch das ϵ verletzt ist, in eine äquivalente regelkonforme Grammatik umformen.

6.5.1 Algorithmus

Markierungsalgorithmus:

- Markiere alle Non-Terminale A mit $A \rightarrow \epsilon$
- Solange es Regeln gibt, so daß die rechte Seite aus lauter markierten Non-Terminalen besteht, markiere das Non-Terminal auf der linken Seite.
- Gebe alle markierten Non-Terminale als V_ϵ aus.

Transformationsalgorithmus:

- Falls das Nonterminal S in irgendeiner Regel auf der rechten Seite vorkommt, füge $S' \to S$ ein.
- Führe den oben beschriebenen Markierungsalgorithmus durch.
- Entferne alle Regeln $A \to \epsilon$ aus der Grammatik
- Wenn $S' \in V_\epsilon$ füge $S' \to \epsilon$ ein.
- Solange es eine Regel $B \to xAy$ mit $|x| + |y| \geq 1$ und $A \in V_\epsilon$ gibt, füge $B \to xy$ ein, wenn diese Regel noch nicht existiert.

6.5.2 Beispiel

B

Transformation von folgender Grammatik G in eine äquivalente ϵ-freie CFG:

$$\begin{array}{lcl} S & \to & aCb|ACB \\ A & \to & aAA|DDDD|aab \\ B & \to & AAC|b \\ C & \to & SB|\epsilon \\ D & \to & AacbS|CE \\ E & \to & C|bca \end{array}$$

Markiert werden nacheinander folgende Nichtterminale (letztendlich alle):

$$C, E, D, A, B, S$$

Wir streichen die Regel

$$C \to \epsilon$$

S ist Element von V_ϵ. Wir müssen ein Startsymbol S' hinzufügen:

$$S' \to S$$

Folgende Regeln entstehen nun bei dem Durchlauf des Transformationsalgorithmusses:

$$\begin{array}{lcl} S' & \to & S \\ S & \to & aCb|ACB|ab|AB|CB|B|AC|A|C \\ A & \to & aAA|DDDD|aab|DDD|DD|D|aA|a \\ B & \to & AAC|b|AA|AC|C|A \\ C & \to & SB|S|B \\ D & \to & AacbS|CE|E|C|acbS|Aacb|acb \\ E & \to & C|bca \end{array}$$

6.6 Abschlußeigenschaften

Ein Sprachtyp i heißt *abgeschlossen* unter $\bowtie$, falls für die Sprachen L_1 und L_2 vom Typ i gilt:

$$L_1 \bowtie L_2\, ist\, auch\, vom\, Typ\, i$$

6.6.1 Vereinigung

Die Sprache $G_\uplus$ wird gebildet, indem man G_1 und G_2 vereinigt[48] und ein neues Startsymbol einfügt, von welchem auf die Startsymbole von G_1 und G_2 gezeigt wird:

$$S \to S_1$$

[48] $G_1 \uplus G_2$

$$S \rightarrow S_2$$

Die Sprache $L(G_{\uplus})$ enthält dann alle Wörter die die beiden Sprachen auch enthalten haben.

6.6.2 Konkatenation

Mit der Konkatenation $(G_1 \circ G_2)$ werden die beiden Grammatiken hintereinandergeschaltet. Zuerst kommt ein Wort der Sprache $L(G_1)$, dann eins der Sprache $L(G_2)$. Dies erreicht man indem man ein neues Startsymbol S einführt und für dieses folgende Regel definiert:

$$S \rightarrow S_1 S_2$$

6.6.3 Kleenabschluß

Beim Kleenabschluß (G^*) kommen beliebig viele Wörter der Grammatik G hintereinander[49].

6.7 Sprachen vom Typ 0

6.7.1 Allgemein

Zu jeder Grammatik G gibt es eine NTM τ mit $L(G) = L(\tau)$.
Zu jeder DTM τ gibt es eine Grammatik G mit $L(\tau) = L(G)$.[50]

6.7.2 Nichtdeterministisches Semientscheidungsverfahren für das Wortproblem

Eingabe: P (Regeln von G)
S (Startsymbol von G)
w (zu überprüfendes Wort $w \in \Sigma^*$, wobei Σ Terminalalphabet)

- Solange $w \neq S$
  - Wähle nichtdeterministisch eine Regel $u \rightarrow v$ aus P
  - Wenn v ein Teilwort von w ist, ersetze es durch u[51]
- Wir geben „JA" aus, wenn die Schleife fertig ist.

6.7.3 Abgeschlossenheit

Typ 0 Sprachen sind unter

- Vereinigung
- Konkatenation
- Kleenabschluß

[49] Klar, wie man das macht: $S' \rightarrow SS'$

[50] Der zweite Teil ohne Beweis.

[51] Der Baum zur Erzeugung von w wird rückwärts verfolgt, bis wir das Wort auf das Startsymbol reduziert haben, so daß wir sagen können, daß $w \in L(G)$. Wenn dies nicht gilt, so „hängen" wir in der Schleife.

- sowie dem Durchschnitt $(L_1 \cap L_2)$[52]

abgeschlossen.

Unter dem Komplement ist Typ 0 nicht abgeschlossen. Z.B. sind H und $\overline{H}$ nicht beide semientscheidbar.

6.8 Kontextsensitive Sprachen - Typ 1

6.8.1 LBAs - Linear beschränkte Automaten

Linear beschränkte Automaten sind NTMs, die mit linear beschränktem Platz auskommen. Sie sind durch zwei Begrenzungssymbole $ und # links und rechts beschränkt.

Zu jeder kontextsensitiven Grammatik G gibt es einen LBA τ mit

$$L_{LBA}(\tau) = L(G)$$

Zu jedem LBA τ gibt es eine Grammatik G mit

$$L(G) = L_{LBA}(\tau)$$

6.8.2 Determinismus $\leftrightarrow$ Nichtdeterminismus

Die Frage, ob deterministsche LBAs und nichtdeterministische LBAs die selben Sprachen akzeptieren können, ist noch nicht gelöst.

6.8.3 Wortproblem

Der Algorithmus kann als nichtdeterministisches Zusammenschieben bezeichnet werden.

- Solange w $\neq$ S
  - Wähle nichtdeterministisch eine Regel $u \to v$ aus P.
  - Wenn v ein Teilwort von w ist, ersetze es durch u.
  - Schiebe den LBA um $|v| - |u|$ Stellen zusammen. Das sind genau die Stellen, die nun nicht mehr benötigt werden.
- Wenn die Schleife erfolgreich abgelaufen ist, geben wir „JA” aus.

6.8.4 Abgeschlossenheit

Typ 1 Sprachen sind zusätzlich zu der Abgeschlossenheit von Typ 0 Sprachen auch unter dem *Komplement* abgeschlossen.

6.9 Wortproblem für Typ 0 und Typ 1 Sprachen

Das Wortproblem für Sprachen vom Typ 0 ist unentscheidbar.

Dies ist im wesentlichen eine Folgerung aus der Unentscheidbarkeit des Halteproblems.

Das Wortproblem für Sprachen vom Typ 1 ist $PSPACE$-vollständig.

[52] Wir können die Semientscheidungsverfahren der beiden Sprachen parallel oder hintereinander laufen lassen. Wenn beide akzeptieren, dann akzeptiert auch die Durchschnittssprache.

Wir können aufgrund dieser Erkenntnis ein Verfahren angeben, welchen Worstcaselaufzeit $EXPTIME$ hat. Sei

$$T_n^0 = \{S\}$$

und

$$T_n^{m+1} = T_n^m \cup \{x \in (V \cup \Sigma)^* : |x| \leq n \wedge t \Rightarrow x \, f\ddot{u}r \, ein \, t \in T_n^m\}$$

Wenn wir uns den Berechnungsbaum vorstellen, so ist T_n^m die Menge der Wörter, die wir bis Ebene m gebildet haben und die maximal die Länge n haben.

Algorithmus zur Berechnung, ob ein Wort w in der Sprache enthalten ist:

- $n = |w|$
- $m = 0$
- $T_n^0 = \{S\}$
- Wiederhole
  - Berechne T_n^{m+1}
  - $m = m + 1$
- Solange bis
  $\underbrace{w \in T_n^m}_{\text{Wort enhalten}} \vee \underbrace{T_n^{m+1} = T_n^m}_{\text{Es kommen keine neuen Wörter mehr hinzu} \rightarrow \text{Wort nicht enhalten}}$

Dieser Algorithmus hat exponentielle Laufzeit, da im schlimmsten Fall die Anzahl der Wörter in T_n^m $\Theta(|V| \cup |\Sigma|)^n$ ist.

7 Reguläre Sprachen

7.1 Endliche Automaten

7.1.1 Definition: Deterministischer endlicher Automat (DFA)

Ein DFA ist ein 5er-Tupel $M = (Q, \Sigma, \delta, q_0, F)$ bestehend aus

Q:	Endlicher Zustandsmenge
Σ:	endliches Alphabet, welches für das Eingabewort und bei den Übergängen benutzt wird
δ:	Übergangsfunktion $\delta : Q \times \Sigma \rightarrow Q$
q_0:	Startzustand aus Q
F:	Endzustandsmenge $F \subseteq Q$

- Gibt es keine passende Übergangsfunktion mehr und das Eingabewort ist noch nicht ganz gelesen, so verwirft der Automat.
- Ist der Automat, wenn das Eingabewort komplett gelesen ist in einem Endzustand, so akzeptiert er[53].

[53]Dies ist anders als bei Turingmaschinen. Turingmaschinen akzeptieren sofort, wenn sie in einen Endzustand kommen. Automaten akzeptieren nur, wenn das Wort zuende ist und sie in einem Endzustand sind. Sie laufen über den Endzustand hinweg, wenn das Wort noch nicht zuende ist.

- Ist er nicht in einem Endzustand bei Ende des Wortes, so verwirft er.

Automaten gehen auf dem Band mit jedem Zustandwechsel immer eine Zelle nach rechts.

7.1.2 Fangzustand

Manchmal benötigt man einen DFA, der solange läuft, bis das Eingabewort abgearbeitet ist. Mann kann hierfür einen Fangzustand generieren, in den alle nicht definierten Übergangsfunktionen gehen, wenn also an dieser Stelle eigentlich der Automat schon verwerfen müßte. Dieser Fangzustand ist nicht in der Endzustandsmenge, weshalb dann verworfen wird, wenn das Wort zuende ist.

7.1.3 Lauf

Man kann die Zustände, durch die ein DFA beim akzeptieren oder verwerfen eines Wortes läuft hintereinander auflisten. Eine solche Liste nennt man Lauf. Man kann einen akzeptierenden oder verwerfenden Lauf haben. Haben wir einen verwerfenden Lauf so schreiben wir an das Ende des Laufes ein $\perp$, also z.B

$$q_0 q_1 \dots q_i \perp$$

7.1.4 Definition: Nichtdeterministischer endlicher Automat (NFA)

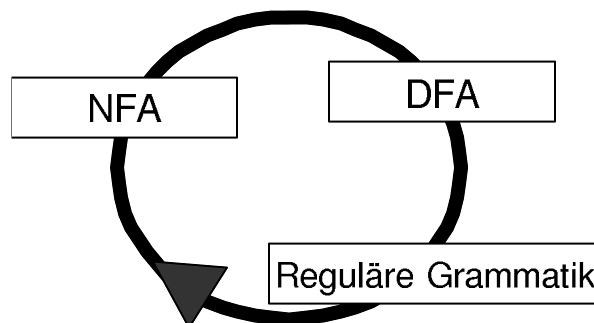
- Ein NFA ist ein Automat, der in mehr als einen Zustand mit dem gleichen Zeichen im Eingabewort wechseln kann. Grafisch gesprochen gibt es also mehrere Pfeile vom selben Zustand mit der selben Inschrift zu unterschiedlichen Folgezuständen.
- Des weiteren kann ein NFA mehr als einen Startzustand haben.
- Aus den zur Auswahl stehenden Zustandswechseln und Startzuständen wird nichtdeterministisch einer gewählt, so daß - wenn möglich - der NFA einen akzeptierenden Lauf hat.

$\delta : Q \times \Sigma \rightarrow 2^Q$	Übergangsfunktion
$Q_0 \subseteq Q$	Menge der Startzustände

2^Q bezeichnet hierbei die Potenzmenge. Eine Potenzmenge ist die Menge aller Teilmengen einer Menge, also hier alle Teilmengen von Q.

7.2 Endliche Automaten und reguläre Grammatiken

7.2.1 Übersicht



7.2.2 DFA → reguläre Grammatik

Wir können aus einem DFA eine reguläre Grammatik machen.

Dazu verwenden wir folgende Regeln:

Wenn $q_0 \in F$:	$q_0 \to \epsilon$
Wenn $\delta(q, a) = p$:	$q \to ap$
Wenn $\delta(q, a) = p$ und $p \in F$:	$q \to a$

7.2.3 NFA → DFA (Potenzmengenkonstruktion)

Beim Überführen von einem NFA in einen DFA generieren wir die Potenzmenge der Zustände des NFAs. D.h. das wir aus allen Untermengen der Menge Q einen neuen Zustand machen.
Wir generieren den neuen DFA $M' = (Q', \Sigma, \delta', q_0, F')$ wie folgt:

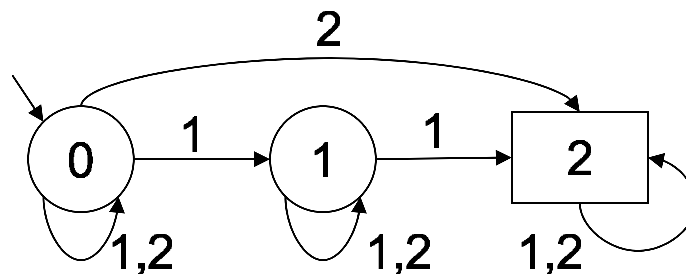
$$F' = \{P \subseteq Q : P \cap F \neq \emptyset\}$$

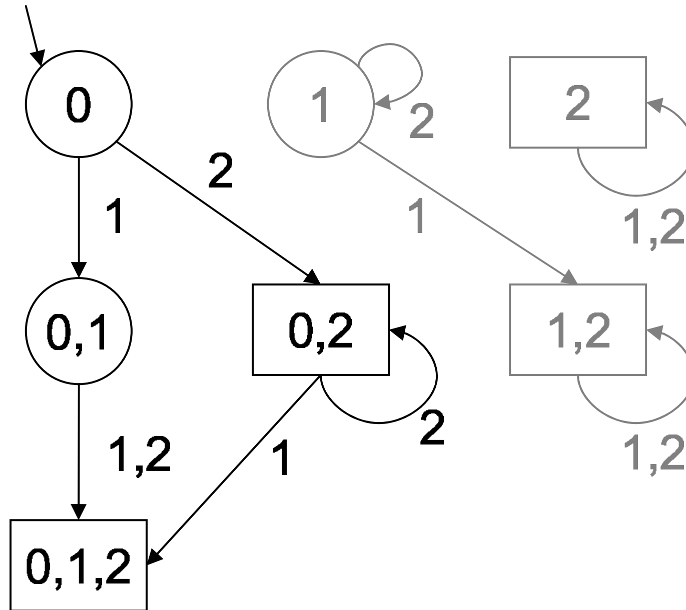
$$\delta'(P, a) = \bigcup_{p \in P} \delta(p, a)$$

Da dies niemand verstehen kann, hier noch einmal eine Konstruktionsvorschrift für den DFA. Zur Konstruktion müssen wir folgendes tun:

- Potenzmenge der Zustände des NFAs bilden und alle Zustände aufmalen. Dabei darauf achten, daß die Potenzzustände, in denen ein Endzustand vorkommt auch Endzustände werden. (erste Regel)
- Nun die Übergangsfunktionen in Form von Pfeilen mit Inschriften einzeichnen. Dabei betrachten wir für jeden Potenzzustand alle Symbole des Eingabealphabetes. Sei nun das aktuelle Symbol gerade a.
  - Wir legen uns eine Menge Z an.
  - Nun gehen wir jeden Zustand, der in unserem Potenzzustand vorhanden ist, durch. Können wir von diesem Zustand in dem NFA einen anderen Zustand oder vielleicht sogar mehrere mit Hilfe von a erreichen? Dann kommen diese Zustände in die Menge Z.
  - Wenn wir alle Zustände im vorhergehenden Schritt abgearbeitet haben, dann haben wir einen Potenzzustand in der Menge Z stehen, zu dem wir nun einen Pfeil mit der Inschrift a malen.

B





Dies ist das Beispiel aus dem Skript von Frau Bayer. Ich habe jedoch in dem konstruierten Graphen einige Fehler gefunden. Die grauen Knoten und Kanten kann man auch weglassen, da sie nie erreicht werden. Formal werden sie aber auch bei der Konstruktion erzeugt.

7.2.4 Effizienz von NFAs

Der DFA zu einem NFA hat im schlechtesten Falle exponentiell viele Zustände wie der NFA. Es gibt Sprachen, die sich durch NFAs viel besser darstellen lassen, wie zum Beispiel

$$L_n = \{w \in \{0,1\}^* : das\, n - letzte\, Zeichen\, von\, w\, ist\, eine\, 1\}$$

7.2.5 Reguläre Grammatik → NFA

Wir können aus einer regulären Grammatik einen NFA machen:

- Jedes Non-Terminal wird zu einem Zustand konvertiert.
- Dazu kommt der Zustand q_F, welcher Endzustand ist.
- Existiert in der Grammatik eine Regel $S \rightarrow \epsilon$, wobei S Startzustand ist, so ist im NFA S ein Endzustand. Dadurch wird erreicht, daß auch das leere Wort akzeptiert wird.
-
  - Wenn es eine Regel $A \rightarrow a$ gibt, dann kreiren wir eine Übergangsfunktion δ (zeichnen einen Pfeil) von dem Zustand A nach q_F mit der Inschrift a.
  - Wenn es eine Regel $A \rightarrow aB$ gibt, dann kreiren wir eine Übergangsfunktion (zeichnen einen Pfeil) von dem Zustand A in den Zustand B mit der Inschrift a.

7.3 Verknüpfungen regulärer Sprachen

7.3.1 NFAs mit ϵ-Übergängen

Wir können in NFAs ϵ-Übergänge[54] definieren. Bei ϵ-Übergängen verbraucht der NFA keine Eingabe. Er kann ϵ-Übergänge spontan nichtdeterminitisch ausführen (dann wenn es so erforderlich ist, damit das Eingabewort möglichst akzeptiert).

Das jeder NFA in einen NFA mit ϵ-Übergängen überführt werden kann, ist klar. Wir brauchen nichts zu machen.

Jeder NFA mit ϵ-Übergängen kann auch in einen NFA ohne ϵ-Übergänge umgeformt werden. Dazu „überbrücken" wir sozusagen die ϵ-Übergänge:

Sei

$$A \xrightarrow{\epsilon} B \xrightarrow{a} C$$

der ϵ-Übergang, noch gefolgt von einem anderen Übergang.

- Wir entfernen die Regel $A \xrightarrow{\epsilon} B$
- Wir fügen $A \xrightarrow{a} C$ hinzu.
- Falls es eine Übergangsfunktion $B \xrightarrow{b} B$ gegeben hat, dann fügen wir $A \xrightarrow{b} B$ und lassen $B \xrightarrow{b} B$. Des weiteren lassen wir auch $B \xrightarrow{a} C$ stehen.

7.3.2 Vereinigung - $L(M_1) \cup L(M_2)$

Wir fassen beide NFAs als einen auf. Es wird nichtdeterministisch entschieden, welcher Startzustand verwendet wird. Entweder der von Automat M_1 oder M_2.[55]

7.3.3 Durchschnitt - $L(M_1) \cap L(M_2)$

- Wir bilden einen Produktautomaten, indem wir einen jeden Knoten des Automatens M_1 mit allen Knoten des Automatens M_2 multiplizieren. In einem Zustand des Produktautomaten stehen dann also zwei Zustände.
- Wir zeichnen eine Übergangsfunktion zwischen den beiden Zuständen mit der Inschrift a, wenn es zwischen den einzelnen Zuständen in den Produktzuständen auch zwei Übergangsfunktionen mit Inschrift a gegeben hat. Also:

$$(P_1Q_1) \xrightarrow{a} (P_2Q_2) \Leftrightarrow P_1 \xrightarrow{a} P_2 \wedge Q_1 \xrightarrow{a} Q_2$$

7.3.4 Komplement - $\overline{L(M)} = \Sigma^* \setminus L(M)$

Wir gehen von einem Automaten hierbei aus, der eine totale Übergangsfunktion hat, d.h. das wir alle Übergänge, die nicht mehr „weitergehen" in einen Fangzustand leiten. Wir erhalten die Komplementsprache, indem wir alle Endzustände zu normalen Zuständen machen und alle normalen Zustände zu Endzuständen.

[54] Bei Professor Martini hießen diese ϵ-Moves

[55] Vereinigen wir zwei DFAs, so können wir auch die Morgansche Regel anwenden, um die Potenzmengenkonstruktion, die wir anwenden müssen, um aus dem entstehenden NFA wieder einen DFA zu machen, zu umgehen:

$$L(M_1) \cup L(M_2) = \overline{\overline{L(M_1)} \cap \overline{L(M_2)}}$$

7.3.5 Konkatenation - $L(M_1) \circ L(M_2)$

Wir können zwei NFAs hintereinanderschalten, indem wir aus dem Endzustand von M_1 einen normalen Zustand machen und von diesem mit einem ϵ-Übergang in M_2 überleiten.

7.3.6 Kleenabschluß - $L(M)^* = L(M^*)$

- Wir gehen vom Endzustand wieder in den Startzustand mit einem ϵ-Übergang. Hiermit akzeptieren wir beliebig oft Wörter, die mit M gebildet worden sind.
- Desweiteren fügen wir zusätzlich noch einen Endzustand hinzu, der im NFA auch gleichzeitig Startzustand wird. Hiermit akzeptieren wir das leere Wort.

7.4 Algorithmen zur Feststellung von Eigenschaften

7.4.1 Das Wortproblem

Das Wortproblem ist die Frage ob ein Wort w in der Sprache des DFAs enthalten ist. Wir simulieren hierfür den DFA. Das Wortproblem kann in linearer Zeit entschieden werden, nämlich in $\Theta(|w|)$. Für NFAs kann das Wortproblem in exponentieller Zeit gelöst werden, da vorerst der Automat in einen Potenzautomaten umgeformt werden muß, da ein Potenzautomat immer 2^m Zustände hat, wobei m die Anzahl der Zustände des Orignial-DFAs.

7.4.2 Leerheitstest

Man prüfe, ob der Automat Endzustände hat, die erreichbar sind. Hat er davon keine, so ist die Sprache leer. Andernfalls ist sie es nicht. Beachte, daß wenn der Startzustand Endzustand ist, die Sprache das Wort ϵ enthält und somit nicht leer ist.
Die Laufzeit hierfür ist $O(|Q| \cdot |\Sigma|)$.

7.4.3 Äquivalenztest

Es gilt

$$L(M_1) = L(M_2) \Leftrightarrow L(M_1) \subseteq L(M_2) \textit{ und } L(M_1) \supseteq L(M_2)$$

Für den Inklusionstest $L(M_1) \subseteq L(M_2)$ können wir folgendes ausnutzen:

$$L(M_1) \subseteq L(M_2) \Leftrightarrow L(M_1) \cap \overline{L(M_2)} = \emptyset \Leftrightarrow L(M_1 \times \overline{M_2}) = \emptyset$$

Wir bilden also zwei Mal den Produktautomaten aus

- einmal der einen Sprache normal und der anderen Sprache negiert und
- einmal aus der einen Sprache negiert und aus der anderen normal

und schauen, ob beide Automaten die leere Sprache erzeugen. Wenn ja, sind sie äquivalent.
Die Laufzeit für dieses Verfahren ist $O(|Q_1| \cdot |Q_2| \cdot |\Sigma|)$

7.4.4 Endlichkeitstest

Um zu testen, ob eine Sprache endlich ist, wenden wir auf dem Automaten ähnliche Algorithmen, wie schon aus Informatik III bekannt an.

- Zuerst entfernen wir alle Zustände und die dazugehörigen Kanten, die nicht vom Startzustand aus erreicht werden können.
- Dann bilden wir den inversen Graphen G^{-1}.
- Wir prüfen, ob von einem Endzustand dieses Graphens irgendein Zyklus erreicht werden kann. Die Zyklen sind genau dieselben, wie in dem Graphen G. Wir können Zyklen mit Hilfe der DFS-Kantenklassifizierung finden. Ein Zyklus exisitert dann, wenn es Rückwärtskanten gibt.

7.5 Pumping Lemma für reguläre Sprachen

7.5.1 Definition

Sei $L \subseteq \Sigma^*$ eine reguläre Sprache. Dann gibt es eine ganze Zahl $n \geq 1$, so daß jedes Wort $|x| \geq n$ wie folgt zerlegt werden kann:

$$x := uvw \text{ mit Wörtern } u, v, w \in \Sigma^*, \text{ so daß}$$

1. $|v| \geq 1$
2. $|uv| \leq n|$
3. $uv^k w \in L$ für alle $k \in \mathbb{N}$

7.5.2 Kernaussage

Die Kernaussage des Pumping Lemmas ist es, daß Autmaten keinen unbeschränkten Speicher haben, mit welchem sie zählen könnten. Es ist nicht möglich Sprachen von Typ (oder ähnlichen Typs)

$$L = \{a^n b^n : n \in \mathbb{N}\}$$

mit einem DFA zu entscheiden, da er nicht festhalten kann, wie groß n ist.

7.5.3 Beweis

Sei $x = a_1 a_2 \ldots a_m$ mit $|x| = m > n$ ein Wort in L, wobei n die Anzahl der Zustände des DFAs.

Dann ist $q_0 q_1 \ldots q_m$ der Lauf im DFA, wobei $q_m \in F$.

Da die Anzahl der Zustände n ist und der Lauf größer ist, müssen Zustände mehrfach besucht werden. D.h. wir gehen eine Schleife im Graphen für das Teilwort v.

Es gibt Indizes, so daß gilt $(0 < i < j < n)$

$$u = a_1, a_2, \ldots, a_i \;\; v = a_{i+1}, \ldots, a_j \;\; w = a_{j+1}, \ldots, a_n$$

1. Wegen $i < j$ gilt $|v| \geq 1$
2. Wegen $j \leq n$ gilt $|uv| = |a_1, \ldots, a_j| = j \leq n$
3. Wir können das Wort in $uv^k w$ unterteilen, da der Zustand zu dem Zeichen a_{i+1} gehörig, mehrfach besucht wird und wir beliebig häufig diesen besuchen können.

Wegen $q_m \in F$ ist jedes dieser Worte in der Sprache enthalten.

7.5.4 Beispiele

1.
$$L = \{a^k : k \in \mathbb{N}\}$$
Diese Sprache ist regulär, da wir
$$u = a \;\; v = a \;\; w = a$$
wählen können und Regel 1 erfüllt ist, daß $|v| \geq 1$, Regel 2 erfüllt ist, da $|uv| \leq n$ (jedes Wort hat mindestens 3 Zeichen) und Regel 3 auch erfüllt ist, da wir das a beliebig wiederholen dürfen und das zusammengesetzte Wort immer noch in der Sprache ist.

2.
$$L = \{a^k b^l : k, l \in \mathbb{N}\}$$
Diese Sprache ist regulär, da wir
$$u = a \;\; v = a \;\; w = b^l$$
wählen können und alle Regeln wiederum nicht verletzt sind, ganz besonders die Regel 3 nicht verletzt ist.

3.
$$L = \{a\}$$
Diese Sprache ist auch regulär, jedoch ist das einzige Wort welches gebildet werden kann nur 1 Zeichen lang, weshalb diese Sprache nicht mit dem Pumping Lemma überprüft werden kann, ob sie denn wirklich regulär ist.

4.
$$L = \{a^n b^n : n \in \mathbb{N}\}$$
Diese Sprache ist nicht regulär, da wir sie nicht richtig unterteilen können.

Sei n die Zahl aus dem Pumping Lemma. Da $|uv| \leq n$ kann u nur aus as bestehen. Dann muß $w = b^n$ sein. Da aber auch $v = a$, können auch Wörter $a^{n+l}b^n : l > 0$ gebildet werden. Dies ist ein Widerspruch zu den Eigenschaften der oben definierten Sprache L.

7.6 Reguläre Ausdrücke

7.6.1 Syntax regulärer Ausdrücke

Induktive Definition:

- $\emptyset$ und ϵ sind reguläre Ausdrücke
- für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck (alle Zeichen sind in sich selbst ein regulärer Ausdruck)
- Mit α und β sind auch $(\alpha\beta)$,$(\alpha + \beta)$ und (α^*) reguläre Ausdrücke.
- Nichts sonst ist ein regulärer Ausdruck

CFG:
$$\alpha = \emptyset \,|\, \epsilon \,|\, a \,|\, \alpha\alpha \,|\, \alpha + \alpha \,|\, \alpha^*$$

Syntaxdiagramm:
Beispiele für Syntaxdiagramme sehen wir gleich noch.

7.6.2 Regulärer Ausdruck → NFA

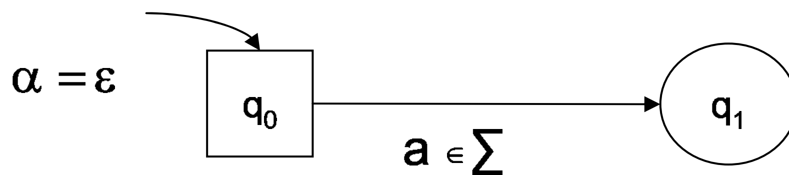
Zu jedem regulären Ausdruck α gibt es einen NFA M, so daß gilt $L(\alpha) = L(M)$

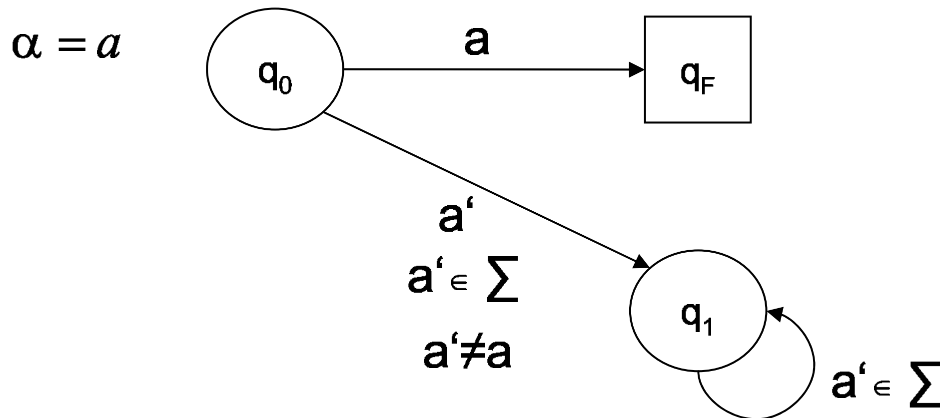
1.Verfahren:

Zu einem regulären Ausdruck α gehört eine Sprache $L(\alpha)$ die wie folgt definiert ist:

$$\begin{array}{ll} L(\emptyset) = \emptyset & L(\epsilon) = \epsilon \\ L(a) = \{a\} & L(\alpha\beta) = L(\alpha) \circ L(\beta) \\ L(\alpha) + L(\beta) = L(\alpha) \cup L(\beta) & L(\alpha^*) = L(\alpha)^* \end{array}$$

Dies läßt uns erkennen, wie wir einen Automaten aus einem regulären Ausdruck bilden können. Wir können dies, indem wir triviale Automaten für Teilausdrücke zur Verfügung stellen und mit diesen Operationen für die Vereinigung, den Durchschnitt und den Kleenabschluß durchführen:

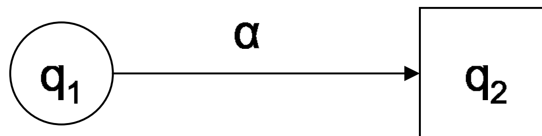




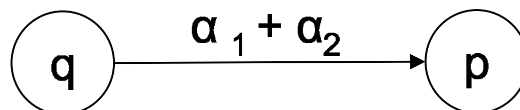
2.Verfahren:

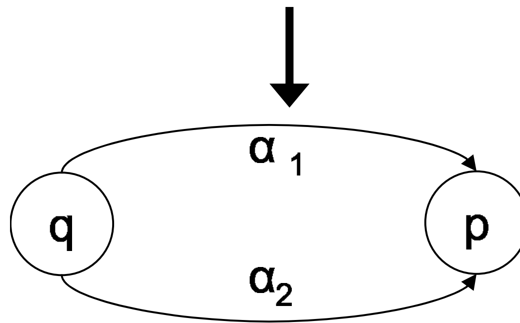
Wir können aus einem initialen NFA einen äquivalenten NFA zum regulären Ausdruck konstruieren, indem wir die Kanten rekursiv ersetzen:

Initialer NFA:

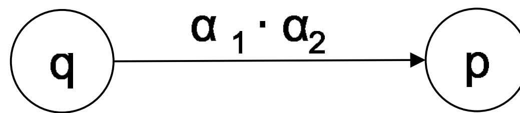


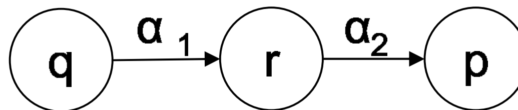
Wenn $\alpha = \alpha_1 + \alpha_2$:



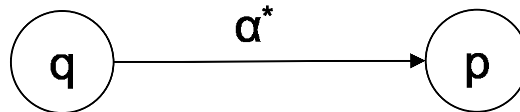


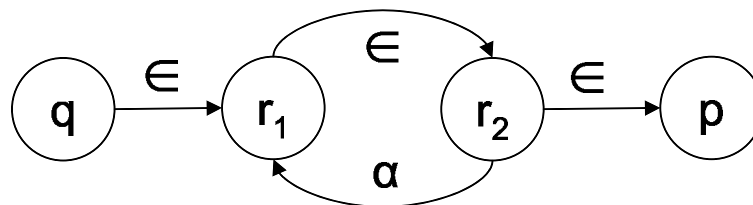
Wenn $\alpha = \alpha_1 \cdot \alpha_2$:





Wenn $\alpha = \alpha^*$:





7.6.3 DFA → regulärer Ausdruck

Zu jedem DFA M gibt es einen regulären Ausdruck α, so daß gilt $L(M) = L(\alpha)$
Wir wenden das Verfahren des dynamischen Programmierens an. Dabei erstellen wir für Teilmengen an Zuständen des DFAs reguläre Ausdrücke und fügen diese zu immer größer werdenden Teilmengen zusammen. Wir starten dabei bei Teilmengen aus zwei oder einem Zustand, wobei bei zwei Zuständen diese Zustände direkt verbunden sein müssen.
Es gilt für $1 \leq i$ und $j \leq n$:

$$L_{i,j}^0 = \{a \in \Sigma : \delta(q_i, a) = q_j\}$$

$$L^0_{i,i} = \{\epsilon\} \cup \{a \in \Sigma : \delta(q_i, a) = q_i\}$$

Wir können folgende reguläre Ausdrücke erzeugen:

$$\alpha^0_{i,j} = a_1 + a_2 + a_3 \ldots + a_n \quad f\ddot{u}r\ L^0_{i,j}$$

$$\alpha^0_{i,i} = \epsilon + a_1 + a_2 + a_3 \ldots + a_n \quad f\ddot{u}r\ L^0_{i,i}$$

Diese Ausdrücke können wir nach folgender Formel zusammenfügen

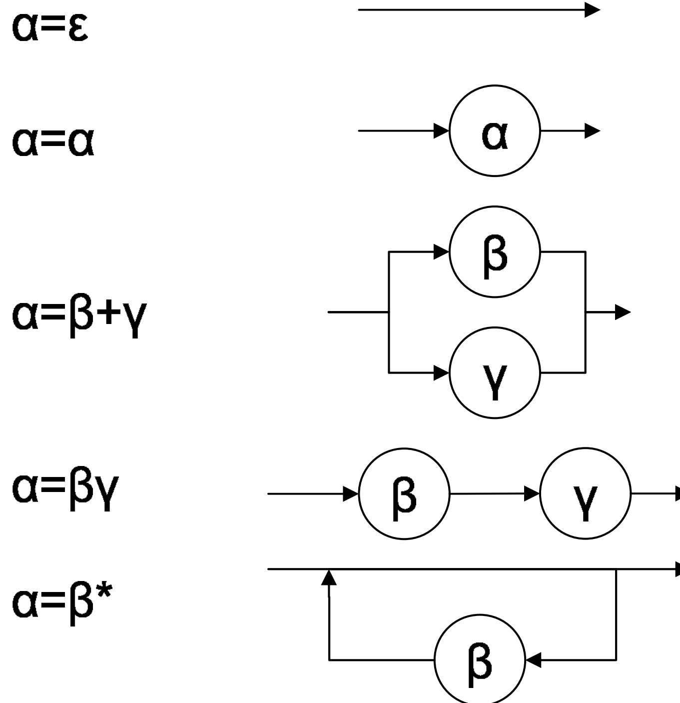
$$L^{k+1}_{i,j} = L^k_{i,j} \cup L^k_{i,k+1} \circ (L^k_{k+1,k+1})^* \circ L^k_{k+1,j}$$

Wieder der reguläre Ausdruck

$$\alpha^{k+1}_{i,j} = \alpha^k_{i,j} + \alpha^k_{i,k+1}(\alpha^k_{k+1,k+1})^*\alpha^k_{k+1,j}$$

7.7 Syntaxdiagramme

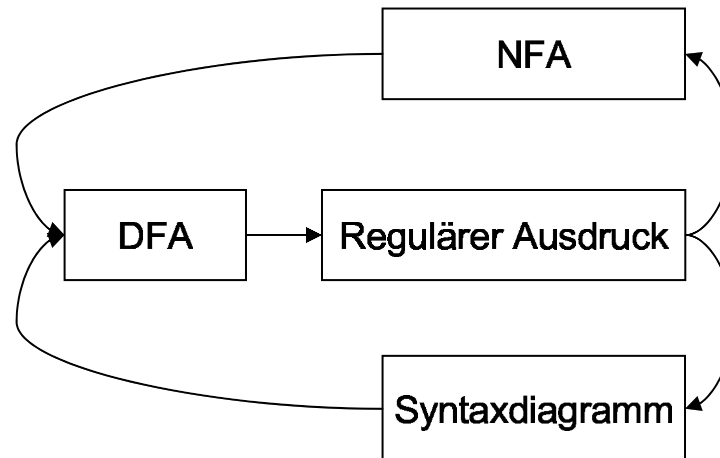
7.7.1 Regulärer Ausdruck → Syntaxdiagramm



7.7.2 Syntaxdiagramm → DFA

Siehe Prof. Martinis Vorlesung

7.8 Zusammenfassung



7.9 Minimierung von endlichen Automaten

7.9.1 Satz von Myhill & Nerode

Eine Sprache ist genau dann regulär, wenn der Index ihrer Äquivalenzklassen endlich ist[56].

7.9.2 Definition: Minimalautomat

Der Automat einer Sprache hat mindestens so viele Zustände, wie die Sprache Äquivalenzklassen hat. Einen Automaten mit minimal vielen Zuständen nennt man Minimalautomat.

Der Minimalautomat eines DFAs unterscheidet sich höchstens in der Benennung der Zustände. Der Minimalautomat eines NFAs kann sich auch in der Art seiner Übergänge unterscheiden.

7.9.3 Beweis: Minimalisierungsalgorithmus

?

noch machen

7.9.4 Minimierungsalgorithmus

1. Wir erstellen eine Tabelle, in der wir jedes Zustandspaar (q_i, q_j) eintragen, wobei $i \neq j$ gelten muß. (Also nicht einen Zustand mit sich selbst.)

q_3	□			
q_2	□	□		
q_1	□	□	□	
q_0	□	□	□	□
	q_4	q_3	q_2	q_1

2. Wir markieren in dieser Tabelle alle Zustandspaare, bei denen ein Zustand Endzustand ist.

[56] Der Beweis hierfür wird für den Beweis des Minimalisierungsalgorithmusses gebraucht

3. Wir markieren weiterhin alle Zustandspaare, für die gilt

$$(\delta(q,a), \delta(q',a))$$

ist schon markiert. (Das heißt für ein a ist schon das Folgezustandspaar markiert) Dabei reicht es, wenn dies für ein $a \in \Sigma$ gilt.
Das testen die einzelnen Zustandspaare solange, bis sich in der gesamten Tabelle nichts mehr ändern kann.

4. Wir können die nicht markierten Zustandspaare zusammenfügen.

8 Kontextfreie Sprachen (CFG)

8.1 Ableitungsbaum

Man kann einen Ableitungsbaum für ein Wort einer kontextfreien Sprache erstellen. Anders als bei regulären Sprachen, wo das Wort mit einer Regelfolge erzeugt wird, hat bei einer CFG die Ableitung Baumstruktur.

8.1.1 Rechtsableitung / Linksableitung

Die Rechtsableitung $\Rightarrow_R$ bedeutet, daß das am rechtesten stehende Nonterminal zuerst abgeleitet wird.
$\Rightarrow_R^*$ bedeuted, daß beliebig oft nach rechts abgeleitet wird.
Analog hierzu ist die Linksableitung definiert.

8.1.2 Eindeutigkeit / Mehrdeutigkeit

G heißt eindeutig, wenn es zu jedem Wort $w \in L(G)$ nur genau einen Ableitungsbaum gibt.

G heißt mehrdeutig, wenn es zu einem Wort w mehr als einen Ableitungsbaum gibt.

Eine Grammatik heißt inhärent mehrdeutig, falls es keine Möglichkeit gibt eine eindeutige CFG für die Sprache zu erstellen[57].

8.2 Nutzlose Variablen

8.2.1 Definition

Nutzlose Variablen sind solche Variablen, von denen kein Wort abgeleitet werden kann. Sie tragen nicht zur Erzeugung der Sprache bei.

8.2.2 Algorithmus

Die Entfernung von nutzlosen Variablen erfolgt, indem wir alle Non-Terminale markieren, die nicht nutzlos sind, also aus denen wir ein Wort machen können:

1. Durch Inspektion aller Regeln ermitteln wir die Menge

$$N = \{A \in V : A \rightarrow w \, f\ddot{u}r \, ein \, w \in \Sigma^*\}$$

(Also alle Nonterminale der Regeln $A \rightarrow a$ nehmen wir sofort auf.)

[57]Beispielsweise: $L = \{a^i b^j c^k : i = j \vee j = k\}$

2. Gibt es eine Regel
$$B \rightarrow x_1 A_1 x_2 A_2 \dots x_n A_n$$
in G, wobei
$$alle\, A_i \in N$$
füge B in N ein. Wiederhole Schritt 2 solange, bis keine Regel mehr diese Eigenschaft hat, daß alle Nontermials auf der rechten Seite Element in N sind und das Nonterminal auf der linken Seite noch nicht.

3. Die Menge $V \backslash N$ ist die Menge der nutzlosen Variablen.

8.3 Chomsky Normalform (CNF)

8.3.1 Definition

In der CNF gibt es nur Regeln der Form

$$A \rightarrow a$$

$$A \rightarrow BC$$

8.3.2 Algorithmus zur Eliminierung der Kettenregeln

1. Finde alle Regel-Ketten der Form
$$A_1 \rightarrow A_2,\, A_2 \rightarrow A_3,\, \dots\, , A_n \rightarrow A_1$$
und entferne sie und ersetze sie durch $A_1 \rightarrow A_1$.
Numeriere die Nicht-Terminale, so daß $\forall A_i \rightarrow A_j\;\, i < j$ gilt[58].

2. Reduzierung von Regeln
$$A_i \rightarrow A_j$$
indem das Nonterminal A_j gestrichen wird und die Regel
$$A_j \rightarrow x$$
in A_i integriert werden.

8.3.3 Algorithmus zur Erzeugung der Chomsky-Normalform

1. Algorithmus zur Eliminierung der Kettenregeln ausführen

2. Wir fügen für alle $a \in \Sigma$ eine Regel $A_a \rightarrow a$ ein und ersetzen alle Terminale in der ursprünglichen Grammatik durch A_a.
Also wird zum Beispiel eine Regel $A \rightarrow aBc$ zu
$$A \rightarrow A_a B A_c \quad A_a \rightarrow a \quad A_c \rightarrow c$$

[58] Dies läßt sich erledigen, indem man die starken Zusammenhangskomponenten des Digraphen (V, E) (wobei $(A, B) \in E$ genau dann, wenn $A \rightarrow B$) berechnet, den azyklischen Digraphen der starken Zusammenhangskomponenten erstellt und anschließend von diesem eine Topologische Sortierung macht. *Wozu dies gehört, weiß ich nicht*

3. Wir zerteilen die Regeln mit mehr als 2 Nonterminalen auf der rechten Seite, in dem wir neue Non-Terminals einfügen. Aus

$$A \rightarrow B_1 B_2 B_3 B_4 B_5$$

wird also

$$A \rightarrow B_1 C_1$$
$$C_1 \rightarrow B_2 C_2$$
$$C_2 \rightarrow B_3 C_3$$
$$C_3 \rightarrow B_4 B_5$$

8.3.4 Beispiel

B

Originalgrammatik:

$$S \rightarrow A|aB|aC$$
$$A \rightarrow B|C|cAd$$
$$B \rightarrow S|Ba$$
$$C \rightarrow D|c$$
$$D \rightarrow d|dDD$$

Algorithmus Nr. 1: Elimination der Kettenregeln

1. Wir finden die Kette

$$S \rightarrow A \quad A \rightarrow B \quad B \rightarrow S$$

und eliminieren diese, indem wir A und B durch S in den übrigen Regeln ersetzen und diese Regeln streichen. Dabei können wir auch die entstehende Regel $S \rightarrow S$ streichen.
Wir erhalten

$$S \rightarrow aS|aC| \underbrace{C|cSd}_{Aus\,A} | \underbrace{Sa}_{Aus\,B}$$
$$C \rightarrow D|c$$
$$D \rightarrow d|dDD$$

2. Ersetzung der alleinstehenden Nonterminale
Wir ersetzen $C \rightarrow D|c$ durch

$$C \rightarrow d|dDD|c$$

und $S \rightarrow aS|aC|C|cSd|Sa$ durch

$$S \rightarrow aS|aC|d|dDD|cSd|Sa$$

Von diesem Algorithmus ausgegebene Grammatik ist nun:

$$S \rightarrow aS|aC|d|dDD|cSd|Sa$$
$$C \rightarrow d|dDD|c$$
$$D \rightarrow d|dDD$$

Erstellung der CNF:

1. Führe die Elimination der Kettenregeln mit dem obigen Algorithmus durch
2. Erzeuge für alle Terminale $a \in \Sigma$

$$A_a \rightarrow a$$

und ersetze alle Terminale, die nicht alleinstehend sind:

$$A_a \rightarrow a$$

$$A_c \rightarrow c$$

$$A_d \rightarrow d$$

$$S \rightarrow A_aS|A_aC|d|A_dDD|A_cSA_d|SA_a$$

$$C \rightarrow d|A_dDD|c$$

$$D \rightarrow d|A_dDD$$

3. Wir splitten die Regeln mit mehr als 2 Nonterminalen auf der rechten Seite
$S \rightarrow A_dDD$ wird zu

$$S \rightarrow A_dX_1 \quad X_1 \rightarrow DD$$

$S \rightarrow A_cSA_d$ wird zu

$$S \rightarrow A_cX_2 \quad X_2 \rightarrow SA_d$$

$C \rightarrow A_dDD$ wird zu

$$C \rightarrow A_dX_3 \quad X_3 \rightarrow DD$$

$D \rightarrow A_dDD$ wird zu

$$C \rightarrow A_dX_4 \quad X_4 \rightarrow DD$$

Die Ausgabegrammatik ist

$$A_a \rightarrow a$$

$$A_c \rightarrow c$$

$$A_d \rightarrow d$$

$$S \rightarrow A_aS|A_aC|d|A_dX_1|A_cX_2|SA_a$$

$$X_1 \rightarrow DD$$

$$X_2 \rightarrow SA_d$$

$$C \rightarrow A_d|A_dX_3|A_c$$

$$X_3 \rightarrow DD$$

$$D \rightarrow A_d|A_dX_4$$

$$X_4 \rightarrow DD$$

8.3.5 Größe der CNF

$$size(G_{CNF}) \leq size(G_{normal})^2$$

Im wesentlichen passiert dies durch die Eliminierung der Kettenregeln.

8.4 CYK-Algorithmus für das Wortproblem

8.4.1 Algorithmus

Der CYK-Algorithmus benutzt das Grundprinzip des dynamischen Programmierens. Die CFG muß dabei in CNF vorhanden sein.

- Wir erstellen eine Tabelle für den Algorithmus in folgender Form

	$v[1,n]$	$\star$	$\star$	$\star$	$\star$
	$v[1,n-1]$		$\star$	$\star$	$\star$
					[59]
	$v[1,2]$				$\star$
	$v[1,1]$	$v[2,1]$			$v[n,1]$
Das Wort					

- Die unterste Zeile der Tabelle füllen wir auf, indem wir Regeln $Nonterminal \rightarrow Terminal$ aus der CNF benutzten.
- Wir füllen die einzelnen Zellen der Tabelle auf, indem wir für $v[i,j]$ folgende Regel anwenden:

$$v[i,j] = \bigcup_{l=1}^{j-1}\{A \in V : \exists A \rightarrow BC\, mit \quad \underbrace{B \in v[i,l]}_{in\, der\, Spalte} \quad \wedge \quad \underbrace{C \in v[i+l, j-l]}_{die\, Diagonale} \quad \}$$

  wobei i:Spalten und j:Zeilen.

- Wenn $S \in v[1,n]$, so ist $w \in L_{CNF}$.

8.4.2 Laufzeit

Der Algorithmus läuft in der Laufzeit $O(n^3)$ und hat einen Platzbedarf von $O(n^2)$[60].

8.4.3 Beispiele

Grammatik in CNF:

$$S \rightarrow AC|AB$$
$$C \rightarrow SB$$
$$A \rightarrow a$$
$$B \rightarrow b$$

Für das Wort $w_1 = aabb$ wird die Tabelle wie folgt von unten nach oben aufgebaut:

$\{S\}$			
$\{\}$	$\{C\}$		
$\{\}$	$\{S\}$	$\{\}$	
$\{A\}$	$\{A\}$	$\{B\}$	$\{B\}$
a	a	b	b

Für das Wort $w_2 = aaabbb$ wird die Tabelle wie folgt von unten nach oben aufgebaut:

[59] Die mit $\star$ gekennzeichneten Zellen werden nicht belegt.
[60] n ist die Länge des Eingabewortes

$\{S\}$					
$\{\}$	$\{C\}$				
$\{\}$	$\{S\}$	$\{\}$			
$\{\}$	$\{\}$	$\{C\}$	$\{\}$		
$\{\}$	$\{\}$	$\{S\}$	$\{\}$	$\{\}$	
$\{A\}$	$\{A\}$	$\{A\}$	$\{B\}$	$\{B\}$	$\{B\}$
a	a	a	b	b	b

Beide Wörter sind in der Sprache enthalten.

8.5 Pumping Lemma für kontextfreie Sprachen

8.5.1 Satz

Sei L eine kontextfreie Sprache. Dann gibt es eine natürliche Zahl n, so daß sich jedes Wort $z \in L$ der Länge $\geq n$ wie folgt zerlegen läßt: $z = uvwxy$, wobei

1. $|vx| \geq 1$
2. $|vwx| \leq n$
3. $uv^k wx^k y \in L$ für alle $k \in \mathbb{N}$.

8.5.2 Beweis

noch machen

8.6 Abschlußeigenschaften

Die Klasse der Kontextfreien Sprachen ist unter Vereinigung, Konkatenation und Kleenabschluß abgeschlossen, nicht aber unter Durchschnitts[61] oder Komplementbildung[62].

8.7 Griebach Normalform

8.7.1 Satz

Sei G eine CFG. G ist eine Griebach Normalform, falls eine Produktionen in G von der Form

$$A \rightarrow aB_1B_2 \ldots B_k$$

wobei $k = 0$ sein kann und auch $a = \epsilon$ möglich ist.

8.7.2 Konstruktionsalgorithmus

1. Wenn bei der Regel $A_i \rightarrow A_j x$ das i größer als das j ist, wird A_i überbrückt. $A_2 \rightarrow A_1 x$ $A_1 \rightarrow A_3|A_4$ wird zu $A_2 \rightarrow A_3 x|A_4 x$.

[61]

$$L_1 = \{a^n b^n c^m : n, m \geq 1\} \quad L_2 = \{a^m b^n c^n : n, m \geq 1\}$$

Obwohl beide Sprachen kontextfrei sind, ist

$$L_1 \cap L_2 = \{a^n b^n c^n : n \geq 1\}$$

nicht kontextfrei

[62] Ist unter der Komplementbildung nicht abgeschlossen, weil sie unter dem Durchschnitt nicht abgeschlosse sind. Dies gilt wegen der Regel von De-Morgan

2. Wenn die Regel der Form $A_i \rightarrow A_i x$ ist, wird diese entfernt und stattdessen $B_i \rightarrow xB_i$ und $B_i \rightarrow x$ eingefügt.
   Es werden für alle Regeln $A_i \rightarrow y$ (wobei $y = a \in \Sigma$ oder $aA_j...$) $A_i \rightarrow yB_i$ eingefügt, damit die Kette nicht zerstört wird.
   Am Ende findet man dann keine linksrekursiven Regeln mehr. Alle Terminale sind nach links gerutscht.

3. Eliminierung der Regeln, in denen ein Terminal nicht an erster Stelle steht:
   Beginn mit den größten i-Werten:
   Für alle Regeln $A_i \rightarrow A_j x$ füge für alle Regeln $A_j \rightarrow \beta$ eine Regel $A_i \rightarrow \beta x$ ein und entferne die Originalregel $A_i \rightarrow A_j x$. Am Ende wird kein β mehr vorne steht.

4. Tue dasselbe wie für die A-Regeln für die B-Regeln.

8.7.3 Beispiel

noch machen

8.8 Kellerautomaten

8.8.1 Definition: Nichtdeterministischer Kellerautomat (NKA)

Ein NKA ist ein Tupel

$$K = (Q, \Sigma, \Gamma, \delta, q_0, \#, F)$$

bestehend aus

- endlichen Menge Q von Zuständen
- Eingabealphabeth Σ
- Kelleralphabet Γ
- Anfangszustand $q_0 \in Q$
- Kellerstartsymbol $\#$ (unterstes Kellerzeichen)
- Menge $F \subseteq Q$ von Endzuständen
- Übergangsfunktion δ

8.8.2 Konfiguration, Notation der Übergangsfunktion und Konfigurationswechsel

Konfiguration:
Eine Konfiguration hat die Form

$$K = (q, x, \xi)$$

wobei q der aktuelle Zustand, x das noch zu bearbeitende Band und ξ der Keller ist, wobei das oberste Kellersymbol ganz links steht und das unterste ganz rechts.
Notation der Übergangsfunktion[63]:

$$\delta(q_{aktuell}, x_{erstes}, \xi_{oberstes}) = \{(q_{neu\,1}, \xi_{neu\,1}), \ldots, (q_{neu\,n}, \xi_{neu\,n})\}$$

[63] Das oberste Kellerelement wird immer gepoppt. Will man es auf dem Stack liegen lassen, muß man es wieder pushen. Hier kann nun ϵ stehen, wenn es entfernt werden soll.

Bei einem NKA kann in mehrere Folgezustände und/oder Belegungen des Stacks überführt werden, bei DKAs ist es immer nur einer. Leider kann man die Übergangsfunktion und damit den ganzen Automaten nicht mehr zeichnen wie bei DFAs oder NFAs.

Konfigurationswechsel z.B.:

$$(q_1, aaa, \#) \vdash (q_2, aa, a\#) \vdash^* (q_F, \epsilon, \epsilon)$$

Genaueres über die Akzeptanz siehe unten.

8.8.3 Akzeptanzverhalten

Wir haben bei Kellerautomaten zwei Möglichkeiten für die Akzeptanz:

Akzeptanz durch Endzustände:
Nach dem Lesen des *kompletten* Eingabewortes wird akzeptiert, wenn ein *Endzustand* erreicht ist. Der Kellerinhalt ist hierbei egal.

Die durch die Endzustände akzeptierte Sprache ist

$$L(K) = \{w \in \Sigma^* : (q_0, w, \#) \vdash^* (q, \epsilon, \xi) \, für \, beliebiges \, \xi \, und \, q \in F\}$$

Akzeptanz durch leeren Keller:
Nach dem Lesen des *kompletten* Eingabewortes wird akzeptiert, wenn der *Keller* leer ist. Es gibt bei dieser Akzeptanzform keine Endzustände.

Die durch den leeren Keller akzeptierte Sprache ist

$$L(K) = \{w \in \Sigma^* : (q_0, w, \#) \vdash^* (q, \epsilon, \epsilon) \, für \, ein \, beliebiges \, q \in Q\}$$

8.8.4 Akzeptanz durch Endzustand → Akzeptanz durch leeren Keller

Wir fügen von jedem ehemaligen Endzustand eine Übergangsfunktion mit ϵ-Übergang zu dem Zustand q_{leeren} ein. Im Zustand q_{leeren} leeren wir den Keller vollständig und akzeptieren somit. Es kann sein, daß der Keller einmal während der Berechnung leer wird und wir sozusagen ausversehen akzeptieren. Deshalb fügen wir ein Kellersymbol $\perp$ an die unterste Position des Kellers ein. Dieses entfernen wir erst, wenn wir in den Zustand q_{leeren} kommen. Wir benötigen dazu noch einen anderen Startzustand des NKAs, damit $\perp$ zuerst in den Keller gelegt wird und dann erst die Berechnung gestartet wird.

8.8.5 Akzeptanz durch leeren Keller → Akzeptanz durch Endzustand

Wieder führen wir ein Symbol $\perp$ ein, welches ganz unten auf dem Stack liegt, um zu verhindern, daß der Automat vorzeitig verwirft, obwohl er eigentlich akzeptieren soll, damit wir noch die Möglichkeit haben, in einen Endzustand zu überführen. Sobald wir nun in eine Konfiguration $(q, \epsilon, \perp)$ geraten, überführen wir in einen Zustand q_F der Endzustand ist. Zuvor entfernen wir dann auch wieder $\perp$.

8.8.6 Kontextfreie Grammatik → NKA

Die Grammatik muß in Greibach-Normalform vorliegen. Die Regeln sind dann der Art: $A \to aB_1B_2 \ldots B_n$. Zu Beginn steht schieben wir das Startsymbol S auf den Kellerspeicher. Wenn nun A als oberstes Element auf dem Kellerspeicher steht, können wir

- im Wort einen nach rechts gehen, wenn das aktuelle Zeichen a ist (ansonsten verwerfen wir)

- und dann $B_1 B_2 \ldots B_n$ auf den Keller schieben, so daß B_1 oben steht.

Für eine solche NKA benötigen wir nur einen Zustand, der zugleich Endzustand ist, und nehmen die Verschiebungen auf dem Stack mit ϵ-Moves vor. Wir akzeptieren bei leerem Keller.

8.8.7 NKA $\rightarrow$ kontextfreie Grammatik

Hierzu gibt es leider im Skript keinen vollständigen Beweis, weshalb ich ihn auch nicht mache. Auf jeden Fall ist eine NKA und eine kontextfreie Grammatik in beiden Richtungen überführbar und somit äquivalent.

8.8.8 NKAs mit zwei Kellern

NKAs mit zwei Kellern haben die Mächtigkeit von Turingmaschinen, da man mit einer Registermaschine mit zwei Kellern eine Turingmaschine simulieren kann. Man kann sich klar machen, daß dies auch für NKAs funktioniert, so daß NKAs eine Turingmaschine simulieren können.

8.8.9 reguläre Sprache $\cap$ kontextfreie Sprache $=$ kontextfreie Sprache

Ist L_1 eine reguläre Sprache und L_2 eine kontextfreie Sprache, so ist $L_1 \cap L_2$ kontextfrei. Wir können uns vorstellen, daß das Produkt eines NKAs mit einem DFA maximal ein NKA sein wird. *Wir haben auch hierfür keinen Beweis gemacht.*

9 Deterministische kontextfreie Sprachen

9.1 Eigenschaften

Im Gegensatz zu nichtdeterministischen kontextfreien Sprachen kann das Wortproblem bei deterministischen kontextfreien Sprachen statt in kubischer Zeit in linearer Zeit gelöst werden. Die DKA (Deterministischen Kellerautomaten) entscheiden die deterministischen kontextfreien Sprachen.

9.2 Deterministische Kellerautomaten (DKAs)

9.2.1 Definition

Ein DKA ist ein NKA

$$K = (Q, \Sigma, \Gamma, \delta, q_0, \#, F)$$

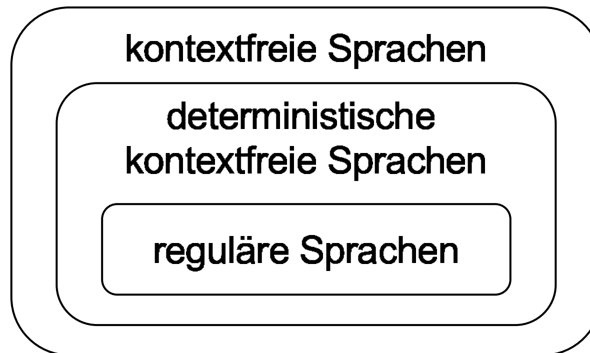
mit folgenden Einschränkungen

- $|\delta(q, a, A)| \leq 1$:
  Von jedem Zustand darf höchstens ein Übergang mit derselben Inschrift für dasselbe oberste Kellersymbol stattfinden (dasselbe wie bei deterministischen Automaten)

- $|\delta(q, \epsilon, A)| \leq 1$:
  Es darf von jedem Zustand höchstens ein ϵ-Move für dasselbe oberste Kellersymbol ausgehen.

- $|\delta(q, a, A) \cup \delta(q, \epsilon, A)| \leq 1$:
  Wenn ein ϵ-Übergang von einem Zustand ausgeht, so darf von diesem kein weiterer Übergang mit demselben obersten Kellersymbol ausgehen.

9.2.2 Eingliederung in die Chomskyhierarchie



9.3 Akzeptanzbedingungen

Bei NKAs für kontextfreie Sprachen sind die Akzeptanzbedingungen durch leeren Keller oder Endzustand äquivalent. Bei DKAs ist dies nicht so. Leerer Keller ist *schwächer* als Endzustand.
Dies ist deshalb so, weil die Berechnung abbricht, wenn der Keller leer ist. Habe wir ein Wort, dessen echtes Präfix[64] auch ein Wort der Sprache ist, so hält die Berechung nach dem Durchlauf des Präfixes verwerfend an, da der Keller leer ist, da ja das Präfix akzeptiert wird, kommt es alleine vor. Mit der Endzustandsakzeptanz könnten wir bis zum Ende laufen.

9.4 Präfixeigenschaft

Weil der Keller bei DKAs leer läuft, ist das Präfix des zu untersuchenden Wortes schon selbst ein Wort der Sprache, benötigen wir die Präfixeigenschaft für alle Wörter der Sprache. Das bedeutet, daß in der Sprache ein Wort niemals echtes Präfix eines anderen Wortes ist.

9.4.1 Definition Präfixeigenschaft

Sei $L \subseteq \Sigma^*$. L hat die Präfixeigenschaft, wenn für alle Wörter $w \in L$ gilt:

$$\text{Ist } x \text{ ein echtes Präfix von } w \text{ so gilt } x \notin L$$

9.4.2 Deterministische kontextfreie Sprachen mit Präfixeigenschaft werden von DKAs mit Leerer-Keller-Akzeptanz entschieden

Sei K ein DKA

1. $L_\epsilon(K)$ ist deterministisch kontextfrei, d.h. $L = L(K')$ für einen DKA K'.
2. $L_\epsilon(K)$ hat Präfixeigenschaft

1. Wenn wir aus einem DKA eine Grammatik machen und dann aus dieser wieder einen NKA, so ist die auch ein DKA. *Keine Ahnung, ob das richtig ist.*
2. Falls die Sprache keine Präfixeigenschaft hat, also $w = xy$ und $w, x \in L_\epsilon(K)$, so verwirft die DKA bei leerer Kellerakzeptanz in der Konfiguration

$$(q, y, \epsilon)$$

[64] Echtes Präfix bedeutet, daß das Wort der Gestalt xy ist, wobei x das Präfix ist und $y \neq \epsilon$

wenn der Keller leer ist. Wenn aber die Sprache Präfixeigenschaft hat, so läuft die DKA mit Leerer-Keller-Akzeptanz bis zum Ende.

Also akzeptieren DKAs mit Leerer-Keller-Akzeptanz nur deterministisch kontextfreie Sprachen mit Präfixeigenschaft.

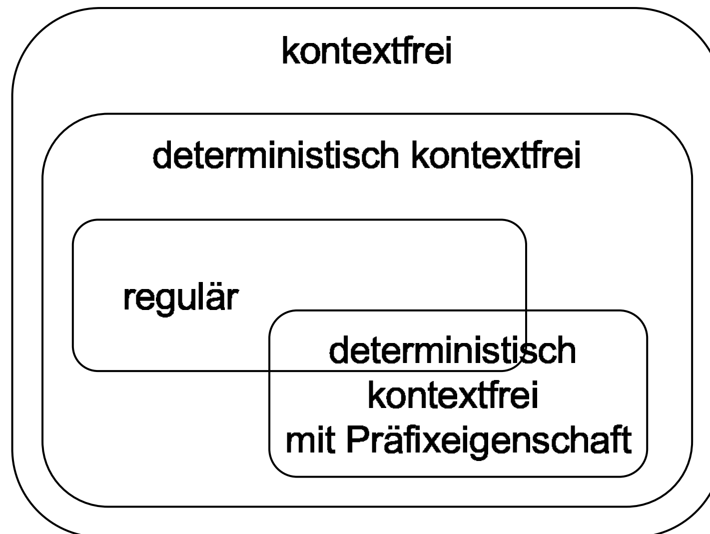
Des weiteren gibt es auch zu jeder deterministischen kontextfreien Grammatik mit Präfixeigenschaft einen DKA mit Leerer-Keller Akzeptanz.

Um dies zu beweisen simulieren wir den DKA mit einem DKA mit Leerer-Keller-Akzeptanz. Wir legen zu Beginn das Symbol $\perp$ auf den Stack. Geraten wir ein einen Endzustand, so entleeren wir den Stack ganz, so daß der DKA akzeptiert. Es ist klar das ein solcher simulierender DKA eine deterministische kontextfreie Grammatik mit Präfixeigenschaft akzeptiert, da der Keller niemals ganz leer wird, sondern das nur am Ende tut.

9.5 Beispiele

Sprache	Art
$L = \{w \in \Sigma^* : a^n b^n \; n \geq 1\}$	deterministisch kontextfrei mit Präfixeigenschaft
$L = \{w \in \{0,1,\#\} : x_1\$x_2 \; x_1 \in \{0,1\}^* \; x_2 \in \{0,1\}^*\}$	deterministisch kontextfrei ohne Präfixeigenschaft, da sobald das Dollarzeichen gelesen wurde es noch weitergehen kann, aber dieses Teilwort auch schon akzeptiert werden könnte. Diese Sprache ist des weiteren auch noch regulär
$L = \{xx^R\}$	nicht deterministisch kontextfrei
$L = \{x\$x^R\}$	deterministisch kontextfrei mit Präfixeigenschaft
L deterministisch kontextfrei $w \in L$ $L\$ = \{w\$: w \in L\}$	deterministisch kontextfrei mit Präfixeigenschaft

9.6 Komplettes Schaubild der Chomskyhierarchie



9.7 Abschlußeigenschaften

Deterministisch kontextfreie Sprachen sind nur unter dem Komplement abgeschlossen.

L_1 regulär und L_2 deterministisch kontextfrei $\Rightarrow L_1 \cap L_2$: deterministisch kontextfrei

10 LR(0)-Grammatiken

10.1 Einleitung zu LR(k)-Grammatiken

LR(k)-Grammatiken heißen LR(k)-Grammatiken, weil

L	die Eingabe von *links* nach rechts gelesen wird
R	immer *Rechtsableitungen* stattfinden
k	ein Lookahead von der Länge k benutzt wird

10.2 Eigenschaften von LR(0)-Grammatiken

LR(0)-Grammatiken sind mit deterministisch kontextfreien Sprachen äquivalent.

Es ist möglich, daß eine Grammatik, die die LR(0)-Bedingungen verletzt, eine äquivalente LR(0)-Grammatik besitzt.

10.3 Begriffsdefinition: Reduktion

Gibt es in der Grammatik die Regel $A \rightarrow y$, so kann

$$xyz_?$$

(wobei $z_?$ der noch nicht gelesene Teil des Wortes ist) nach

$$xAz_?$$

reduziert werden.

Hier wird uns anschaulich die Eigenschaft einer LR(0)-Grammatik klar. Eine LR(0)-Grammatik muß so aufgebaut sein, daß solche Reduktionen klappen. D.h. es muß immer eine Regel eindeutig bestimmt sein, mit der reduziert wird, so daß wir letztendlich in linearer Zeit überprüfen können, ob da Wort in der Sprache ist.

10.4 Definition LR(0)-Grammatik

Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik. G heißt LR(0)-Grammatik falls für alle $z, z', z'' \in \Sigma^*$ $y, x, x' \in (\Sigma \cup V)^*$ und $A, A' \in V$ gilt:

$$\left.\begin{array}{l} S \Rightarrow_R^* xAz \Rightarrow_R xyz \\ S \Rightarrow_R^* x'A'z' \Rightarrow_R x'y'z' = xyz'' \end{array}\right\} \Rightarrow x = x',\ \ A = A',\ \ z' = z''$$

Weiterhin heißt G nur LR(0)-Grammatik, wenn sie keine nutzlosen Variablen enthält und das Startsymbol niemals rechts steht. Zur Vereinfachung nehmen wir des weiteren an, daß unsere LR(0)-Grammatiken keine ϵ-Regeln enthalten.

10.5 Definition von Begriffen für eine verbale Definition von LR(0)-Grammatiken

10.5.1 Griff

Jedes Wortpaar (xy, y) mit $x, y \in (V \cup \Sigma)^*$ für das es eine Rechtsableitung der Form

$$S \Rightarrow_R^* xAz \Rightarrow_R xyz$$

gibt, heißt Griff von xyz. Wenn klar ist, an welcher Stelle die Regel $A \to y$ greift, so kann man auch nur Griff y von xyz sagen.

10.5.2 Rechtssatzform

Sei G eine CFG. Alle Wörter

$$t \in (V \cup \Sigma)^*$$

die aus dem Startsymbol durch eine Rechtsableitung generiert werden können ($S \Rightarrow_R^* t$), heißen Rechtssatzform. Eine Rechtssatzform ist also eine Menge.

10.6 Verbale Definition von LR(0)-Grammatik

Eine CFG G ist eine LR(0)-Grammatik, wenn

- der Griff von t und die zugehörige Regel eindeutig bestimmt ist.
- wenn (xy, y) der Griff von t und xyz Rechtssatzform mit $z \in \Sigma^*$ ist, dann ist (xy, y) zugleich Griff von xyz.
  D.h. also, daß es keine unterschiedlichen zwei Griffe gibt, wenn der Anfang der Wörter xyz und t gleich ist.

10.7 Eindeutigkeit

Jede LR(0)-Grammatik ist eindeutig.

Sei G eine LR(0)-Grammatik. Zu jeder Rechtssatzform $t \neq S$ gibt es ein eindeutig bestimmtes Tupel (x, y, z, A), so daß folgende Bedingungen erfüllt sind:

- $x, y \in (V \cup \Sigma)^*$ $z \in \Sigma^*$ und $A \in V$
- $A \to y$ ist eine Regel in G
- $t = xyz$
- xAz ist eine Rechtssatzform

Aus der Definition der LR(0)-Bedingungen folgt, daß $A \to y$ einzigste Regel ist, um y wieder rückwärts abzuleiten. Aus der Definition folgt, daß xAz einzigster Griff von (xy, y) ist. Somit gibt es nur einen einzigen Rückwärtsableitungspfad.
$\Rightarrow$ Jede LR(0)-Grammatik ist eindeutig

? Damit ist gezeigt, daß LR(0)-Sprachen von einem deterministischen Kellerautomaten entschieden werden können.

10.8 Präfixeigenschaft

Für jede LR(0)-Grammatik G hat die erzeugte Sprache $L(G)$ die Präfixeigenschaft.
? *Hierfür habe ich den Beweis nicht richtig verstanden*

10.9 Nichtdeterministischer LR(0)-Parser

Der Parser dient dazu zu erkennen, ob ein Eingabewort w in der Sprache enthalten ist. Wir zerlegen w wie folgt

$$w = w_1 z_?$$

wobei w_1 der schon vom Parser gelesene Teil ist und $z_?$ der noch nicht gelesene.
Der aktuelle Kellerinhalt ist das Wort v ($v \in (V \cup \Sigma)^*$, welches durch das Zurückverfolgen der Rechtsableitung ermittelt wurde:

$$v \Rightarrow_R^* w_1$$

Der Parser entscheidet nun nichtdeterministisch[65], welche der folgenden Aktionen ausgeführt werden:

- **ACCEPT**: Sobald das im Keller gespeicherte Wort S ist, kann akzeptiert werden.
- **REDUCE**: Eine Reduktion für die Regel $A \rightarrow y$ wird angewendet, so daß das auf dem Keller stehende gekellerte Wort xA anstatt xy ist.
- **SHIFT**: Ein weiteres Zeichen wird aus dem noch unbekannten Wort gelesen und auf den Keller gelegt.
- **ERROR**: Es wird erkannt, daß das Wort nicht in der Sprache enthalten ist und es gibt eine Fehlermeldung.

11 LR(0)-Items

Wenn Muße ist. Frau Bayer ist wahrscheinlich nicht so weit gekommen, da gerade ein wenig weiter auch die Übungszettel enden und keine Prüfung dieses beinhaltet.

[65] Später wollte Frau Bayer auch noch Verfahren angeben, wie dies deterministisch entschieden werden kann.

Anhang

A Tabellarische Aufführung der Abschlußeigenschaften verschiedener Sprachtypen

Sprache	Komplement	Vereinigung	Durchschnitt	Konkatenation	Kleeneabschluß	Wortproblem	Konstrukt
Typ 0	nein	ja	ja	ja	ja	unent.	DTM&NTM
Typ 1	ja	ja	ja	ja	ja	PSP.C	LBA
Typ 2	nein	ja	nein	ja	ja	kubisch	NKA
det. kontextfr.	ja	nein	nein	nein	nein	linear	$\text{DKA}_{Endzustand}$
det. kontextfr. mit Präfixeig.	”	”	”	”	”	linear	$\text{DKA}_{leerer\ Kel.}$ $=\text{DKA}_{\epsilon}$
regulär	ja	ja	ja	ja	ja	linear	DFA&NFA

B Prüfungsprotokolle

Im folgenden findet man einige Prüfungsprotokolle. Diese sind als Emails über die Mailingliste gegangen. Latex hatte etwas Probleme, da einige Strings in ihnen vorkommen, die von Latex interpretiert werden. Diese sind bis auf einige nun alle herausgenommen. Der Rest der Emails ist im Original belassen.
Sorry, die Prüfungsprotokollsammlung habe ich hier rausgenommen, da ich nicht weiß, ob diejenigen, die in dieser Sammlung enthalten sind, mit der Veröffentlichung Ihrer Protokolle einverstanden sind.

B.1 Meine eigene Prüfung

B.1.1 Allgemein

In der Prüfung ist immer der Weitblick, wie das Detail gefragt. Es kann auch vorkommen, daß in der Prüfung Dinge gefragt werden, die so in der Vorlesung gar nicht dran gekommen sind, die sich aber aus dem Zusammenhang erschließen lassen. Man muß wissen, was wir in dem ganzen Jahr, in dem wir die Vorlesung von Frau Baier gehört haben, gemacht haben. Beispielsweise kann man nach einem Greedyalgorithmus für das Graphfärbeproblem gefragt werden, welcher eine Färbung des Graphen ermitteln soll. Man muß wissen, daß dies mit Backtracking, LC und LIFO-Methode funktioniert, welches wir eigentlich nur für das 0/1-Rucksackproblem gemacht haben.

B.1.2 Informatik III

Man sollte alle Algorithmen/Datenstrukturen kennen und benutzen können. Man muß aber nicht die einzelnen Algorithmen auswendig aufschreiben können. Weiterhin ist gefordert, zu wissen, warum man den gerade die und die Datenstruktur verwendet und nicht eine andere. Zum Beispiel wird der AVL benutzt, um im Speicher einen Suchbaum zu haben. Der B-Baum wird für die Speicherung auf der Festplatte verwendet. Dann sind noch ganz wichtig die Laufzeitbetrachtungen, die wir am Beispiel von der Binären Suche, von Mergesort und Quicksort hatten. Hierbei ist die vollständige Induktion zu können. Des weiteren muß man das Master-Theorem nicht nur anwenden können, sondern auch den Beweis verstanden haben.

B.1.3 Informatik IV

Aus Informatik IV ist vor allen Dingen die NP-Vollständigkeit zu kennen. Man muß wissen, wie dieselbe definiert ist, wie man sie beweisen kann (SAT-Problem, Satz von Cook) und wie man ein beliebiges NP-Problem auf ein anderes mit polynomieller Reduktion zurückführen kann. Dabei sind unbedingt auch die Beispiele aus dem Skript zu lernen. Weiterhin ist das Halteproblem und die Entscheidbarkeit, Semientscheidbarkeit, Unentscheidbarkeit und Nicht-Semientscheidbarkeit wichtig. Danach kommen ganz bestimmt noch die ganzen Sprachen. Soweit bin ich aber nicht gekommen.

B.2 Meine Wiederholungsprüfung

B.2.1 Info III

- Mergesort Rekurrenz: Diesmal war die Rekurrenzgleichung nur ungefähr aufzuschreiben, danach Lösung dieser Gleichung
- Mastertheorem: Wieder eine Rekurrenzgleichung, die nicht ganz in dieses Theorem paßt.

- Ungerichtete Bäume: Was sind die Eigenschaften von ungerichteten Bäumen. (Knoten und Kantenanzahl)
- Netzwerke: Ford und Fulkerson und maximale Flußerhöhungsschritte
- Scanlines, hierzu AVL-Blatt-Bäume

B.2.2 Info IV

- Def. Entscheidbarkeit
- Halteproblem informell: Halteproblemtabelle
- Polynomielle Reduzierbarkeit
- Waruzm ist K in NPC, wenn Guess+Checkverfahren angewandt werden kann und $L \leq_{poly} K$ gilt.
- O/1 ILP

Note war insgesamt eine 1,7. Ich muß sagen, daß ich viele Dinge gar nicht richtig konnte. Ich habe mich dann so durchgehangelt und konnte die Professorin doch irgendwie überzeugen, daß ich was kann. Es ist mehr der Überblick als das Detail gefragt.