Zusammenfassung Grundgebiete Informatik 3

Steffen Vogel
steffen.vogel@rwth-aachen.de

11. März 2012

Inhaltsverzeichnis

Teil I
Definitionen

1 Vertiefung Datenstrukturen und Algorithmen

1.1 Zuordnungsprobleme in Graphen

- **Graph**: Ein (gerichteter) Graph $G = (V, E)$ besteht aus einer Menge V von Knoten (vertices) und einer Menge E von Kanten (edges) mit $E \subseteq V \times V$.
  (u, v) bezeichnet eine Kante von Knoten u zu Knoten v, $(u, v) \in E$
- **Ungerichteter Graph**: Gilt für alle Kanten (v, u) eines Graphen $G = (V, E)$:
  $(u, v) \in E \Rightarrow (v, u) \in E$ so heißt G ungerichtet. Schreibweise:$\{u, v\} \in E$
- **Zuordnung**: Eine Zuordnung M für einen ungerichteten Graphen $G = (V, E)$ ist eine Teilmenge $M \subseteq E$ der Kanten E, so dass keine zwei Kanten in M den selben Endknoten haben. Äquivalent gilt, dass kein Knoten $v \in V$ mit mehr als einer Kante indiziert (verbunden) ist.
- **Nicht erweiterbare Zuordnung (Maximal Matching)**: Eine Zuordnung M für einen ungerichteten Graphen ist nicht erweiterbar (maximal matching), wenn es keine Kante $e \in E$, $e \notin M$ gibt, die hinzugefügt werden könnte, sodass $M \cup e$ eine Zuordnung wäre.
- **Maximale Zuordnung (Maximum Matching)**: Eine nicht erweiterbare Zuordnung mit maximaler Größe $|M|$
- **Perfekte Zuordnung**: Eine maximale Zuordnung M, in der alle Knoten $v \in V$ mit genau einer Kante $e \in M$ verbunden sind.
- **Bipartiter Graph**: Bei einem bipartiten Graphen $G = (V, E)$ ist es möglich die Knotenmenge V so in zwei disjunkte Teilmenegen V_1 und V_2 zerlegt werden kann (mit $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$), dass keine Kante zwei Knoten in V_1oder zwei Knoten in V_2verbindet (d.h. $E \subseteq \{\{v_1, v_2\} | v_1 \in V_1, v_2 \in V_2\}$.
- **Gebundene Kante**: Eine in der Zuordnung verwendete Kante $e \in M$ wird als gebundene Kante bezeichnet.
- **Gebundener Knoten**: Ein Knoten, der mit einer gebundenen Kante indiziert ist, ist ein gebundener Knoten.
- **Freier Knoten**: Ein nicht gebundener Knoten, d.h. ein Knoten der nur mit freien Kanten indiziert ist, ist ein freier Knoten.
- **Alternierender Pfad**: Ein Pfad in G wird als alternierender Pfad (alternating path) bezeichnet, falls die Kanten abwechselnd gebunden und frei sind.
- **Vergrößernder Pfad**: Ein vergrößender Pfad (**augmenting** path) ist ein alternierender Pfad, dessen Anfangs- und Endknoten freie Knoten sind.
- **Gewichteter bipartiter Graph**: Ein gewichteter/bewerteter (weighted) bipartiter Graph ist ein bipartiter Graph $G = (V, E)$ in dem jede Kante $e_{ij} = \{v_{1,i}, v_{2,j}\}$ ein Gewicht (eine Bewertung) c_{ij} hat. Das Gewicht der Zuordnung M ist die Summe der Gewichte der Kanten in M.
- **Blüte**: Eine Blüte ist eine Schleife in G, die aus $2n + 1$ Kanten besteht, von denen genau n Kanten gebundene Kanten sind (zu M gehören).
- **Schleifen**: Eine Schleife entsteht in einem bipartiten Graphen dann, wenn man es schafft über $2n$ Kanten von einem Startknoten M wieder zu diesem Startknoten zurückzukommen.
- **Kontrahierender Graph**: Ein kontrahierender Graph bezeichnet einen Graphen, der aus G entsteht, indem die Blüte in einem neuen Knoten v_B zusammengezogen wird.

1.2 Balancierte Bäume

- **Blattsuchbaum**: Ein Suchbaum, dessen Blätter auch Schlüssel enthalten.
- **Balancierter Baum**: Ein balancierter Baum ist ein Baum, bei dem kein Blatt wesentlich weiter von der Wurzel entfernt ist, als irgendein anderes Blatt des Baums. Formal: die maximale Höhe des Baums ist $c \cdot log(n)$, wobei n die Anzahl der Elemente des Baums und c eine von n unabhängige Konstante ist.
- **Binärbaum**: Ein Baum, bei dem jeder Knoten maximal zwei Kindknoten besitzt.
- **Balance eines Binärbaumes**: Höhenunterschied zwischen dem linken (B_L) und dem rechten (B_R) Teilbaum eines Knotens bzw. der Wurzel: Balance = $H(B_L) - H(B_R)$
- **Binärer Suchbaum**: Sei $B = (r, B_L, B_R)$ ein Binärbaum und bezeichne $key(n)$ den Suchschlüssel für jeden Knoten n, r bezeichnet den Wurzelknoten. B heißt Suchbaum, falls gilt:
  - Für alle $n \in B_L : \ key(n) < key(r)$
  - Für alle $n \in B_R : \ key(n) > key(r)$
  - $B_L$und $B_R$sind Suchbäume (Rekursion)
- **AVL-Baum**: Ein AVL-Baum ist ein binärer Suchbaum, bei dem sich die Höhe der beiden Teilbäume eines Knoten maximal um 1 unterscheidet.
  Die Balance in einem AVL-Baum ist entwerder 0, oder ± 1
- **Rot-Schwarz-Baum**: Ein Rot-Schwarz-Baum ist ein annähernd balancierter Suchbaum, der ein zusätzliches Bit (bzw. die Farben rot und schwarz) verwendet um die Balance zu erhalten. Kein Blatt des mehr als doppelt so weit von der Wurzel entfernt, wie irgendein anderes Blatt.
  - Eigenschaften
    - Jeder Knoten ist entweder rot oder schwarz
    - Die Wurzel des Baumes ist schwarz
    - Alle Blätter sind schwarz und enthalten keine Schlüssel
      - Jeder Pfad von einem gegebenen Knoten zu einem Blattknoten enthält die selbe Anzahl von schwarzen Knoten
    - Ist ein Knoten rot, so sind beide Kinder schwarz
- **B-Baum**: Ein B-Baum ist ein balancierter Suchbaum, bei dem jeder Knoten (außer der Wurzel) zwischen $\frac{m}{2}$ und m Kindknoten hat. m ist eine positive ganze Zahl ($m > 1$). m ist die Ordnung des B-Baums.
  - Eigenschaften
    - Die Wurzel hat zwischen 2 und M Kind-Knoten
    - Jeder Knoten hat genau einen Schlüssel zwischen zwei Kind-Knoten
    - Die Blätter besitzen keine Schlüssel
    - Der Baum ist immer perfekt balanciert
    - Ein B-Baum besitzt die Suchbaum Eigenschaften

1.3 Suchalgorithmen

- **Binäre Suche**: $m = m_u + \frac{1}{2}(m_o - m_u)$
  - Durchschnittliche Laufzeit: $O(log\, N)$
- **Interpolationssuche**: $m = m_u + \frac{k - a[m_u].key}{a[m_o].key - a[m_u].key}(m_o - m_u)$
  - Durchschnittliche Laufzeit: $O(log_2 log_2 N)$ (bei gleichverteilten Schlüsseln)
  - Worst Case Laufzeit: $O(N)$ (bei exponentiell ansteigenden Schlüsseln)
- m wird immer auf die nächstliegende ganze natürliche Zahl gerundet!

1.4 Hashverfahren

- **Hashverfahren**: Hashverfahren sind eine effiziente Möglichkeit Schlüssel in einer Hashtabelle zu suchen. Mit einer effizienten Hashfunktion kann so auf einen Datensatz mit $O(1)$ zugegriffen werden (vgl. $O(log(n))$ für binäre Suche).
- **Hashfunktion**: Eine Hashfunktion $h(k) : \mathfrak{K} \to \{0, \ldots, M-1\}$ ordnet jedem Schlüssel k einen Index $h(k)$ mit $0 \leq h(k) \leq M-1$ zu. I.d.R. gilt $|\mathfrak{K}| > M$.
- **Hashadresse**: $h(k)$ wird als Hashadresse bezeichnet.
- **Hashtabelle**: Die gehashten Datensätze werden anhand ihrer Hashadresse in einem linearen Feld mit den Indizes $0, \ldots, M-1$ gespeichert. M ist die Größe der Hashtabelle.
- **Perfektes Hashing**: Bei einer perfekten Hashfunktion werden Kollisionen vollkommen vermieden (nur möglich, wenn $|\mathfrak{K}| \leq M$ und $\mathfrak{K}$ fest und vorher bekannt ist) .
- **Universelles Hashing**: Sei $\mathfrak{K}$ eine endliche Klasse von Hashfunktionen, die jeweils jedem Schlüssel aus $\mathfrak{K}$ auf die Menge $\{0, \ldots, M-1\}$ abbilden, dann heißt $\mathfrak{K}$ universell, wenn für zwei verschiedene Schlüssel $x, y \in \mathfrak{K}$ gilt:
  $\frac{|\{h \in \mathfrak{K} : h(x) = h(y)\}|}{|\mathfrak{K}|} \leq \frac{1}{M}$
  Die Wahrscheinlichkeit, dass ein beliebiges Paar von zwei verschiedenen Schlüsseln x, y aus $\mathfrak{K}$durch eine zufällig aus der universellen Klasse $\mathfrak{K}$ ausgewählte Hashfunktion $h(k)$ auf die selbe Hashadresse abgebildet wird, ist höchstens $\frac{1}{M}$.
- **Adresskollision**: Da die die Größe der Hashtabelle M i.d.R. viel kleiner ist als die Menge der möglichen Schlüssel, kann es passieren dass die Hashfunktion (keine bijektive Funktion) mehrere Schlüssel auf den selben Wert abbildet. Sollen mehrere solcehr Schlüssel in der Hashtabelle gespeichert werden, so tritt eine Adresskollision auf, die eine Sonderbehandlung erfordert.
- **Synonyme**: Zwei Schlüssel k, k' heißen Synonyme, falls $h(k) = h(k')$ gilt.
- **Überläufer**: Ein zu speichernder Eintrag, der auf die gleiche Hashadresse gehasht wird, wie ein anderer Eintrag.
- **Primäre Häufung**: Sondierungsfolgen unterschiedlicher Hash-Adressen treffen aufeinander und verlaufen synchron (z.B. beim linearen Sondieren).
- **Sekundäre Häufung**: Schlüssel, die den gleichen Hashwert haben, durchlaufen dieselbe Sondierungsfolge.
- **Sondierungsfolgen**:
  - Lineares Sondieren: $s_k(n) = n$
  - Doppeltes Hashing: $s_k(n) = n \cdot h'(k)$ bsp. mit $h'(k) = 1 + k \, mod \, (M-2)$

1.5 Suchen in Texten

1.5.1 Naive Zeichenketten-Suche

- Durchschnittliche Laufzeit: $O(M+N)$
- Worst-case Laufzeut: $O(M \cdot N)$

1.5.2 Knuth-Morris-Pratt Algorithmus

- Komplexität
  - Aufbau der Verschiebungstabelle: $O(M)$
  - Suchalgorithmus: $O(n)$
  - Gesamtlaufzeit (auch im Worst-case): $O(M+N)$

2 Optimierungsprobleme und -verfahren

- **Affine Menge**: Die Gerade durch zwei beliebige, verschiedene Punkte der Menge liegen in der Menge.
- **Konvexe Menge**: Der Geradenabschnitt zwischen zwei beliebigen, verschiedenen Punkten der Menge liegt in der Menge.
  - $x_1, x_2 \in C, \theta \in [0,1] \Rightarrow \theta x_1 (1-\theta) x_2 \in C$
  - Schnittmenge zweier konvexer Menge ist konvex
- **Konvexe Funktion**:
  - falls $f(\theta \underline{x}_1 + (1-\theta)\underline{x}_2) \leq \theta f(\underline{x}_1) + (1-\theta) f(\underline{x}_2)$ $(\divideontimes)$
  - falls f diffbar: $f(\underline{x}) \geq f(\underline{x}_0) + \nabla f(\underline{x}_0)^T (\underline{x} - \underline{x}_0)$ $\forall \underline{x}, \underline{x}_0 \in C$
  - falls Hessematrix positiv semidefinit: $H(f(x)) \succeq 0$ $\forall \underline{x} \in C$
  - f ist konkav $\Rightarrow -f$ ist konvex
  - strikt Konvex, falls $(\divideontimes)$ strikt erfüllt ist
- **Hessematrix**:
  $$H(f) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{pmatrix}$$
- **Positive semidefinite Matrix**: $A \succeq 0$
  - falls $\underline{x}^T \cdot A \cdot \underline{x} > 0$ $\forall \underline{x} \in \mathbb{R}^n$
  - falls die Determinante aller Untermatrizen > 0
  - falls die Matrix reel und symmetrisch ist, und alle Eigenwerte positiv sind
  - falls A eine Diagonalmatrix ist und alle Einträge > 0
- **Quasikonvexe Funktion**:
  - Bsp: $f(x) = \sqrt{|x|}$

2.1 Konvexe Optimierungsprobleme

- **Konvexes Optimierungproblem**:
  $$\begin{cases} \min\limits_{x} & f_0(\underline{x}) \\ s.t. & f_i(\underline{x}) \leq 0,\ i = 1, \ldots, m \end{cases}$$
  - Rechenzeit ist grob propotional zu $max\{n^3, n^2, m, F\}$
- **Linear Programming (LP)**:
  $$\begin{cases} min & \underline{c}^T \underline{x} + d \\ s.t. & G\underline{x} \leq \underline{h} \\ & A\underline{x} = \underline{b} \end{cases}$$
- **Quadratic Programming (QP)**:
  $$\begin{cases} min & \frac{1}{2} \underline{x}^T P \underline{x} + \underline{q}^T \underline{x} + d \\ s.t. & G\underline{x} \leq \underline{h} \\ & A\underline{x} = \underline{b} \end{cases}$$
  - P ist positiv semidefinite Matrix: $P \succeq 0$

- **Semidefinite Programming (SDP)**:
$$\begin{cases} min & \underline{c}^T \underline{x} \\ s.t. & x_1 F_1 + x_2 F_2 + \ldots + x_n F_n \succeq 0 \\ & A\underline{x} = \underline{b} \end{cases}$$
- **Modell-Allgemeinheit**: LP < QP < ... < SDP
- **Lösungeffizienz**: LP > QP > ... > SDP

2.2 Lagrange-Multiplikatoren

- Im Allgemeinen nicht lineares Problem mit Gleichheitsbedingungen
$$\begin{cases} min & f(\underline{x}) \\ s.t. & \underline{h}(\underline{x}) = 0 \end{cases}$$
- Lagrange Funktion: $\Lambda(\underline{x}, \underline{\lambda}) = f(\underline{x}) + \underline{\lambda}^T \underline{h}(\underline{x})$
- Bedingung: $\nabla\Lambda = \nabla f + \underline{\lambda}^T \nabla \underline{h} = 0$ (Gradienten sind parallel)

2.3 Karush-Kuhn-Tucker (KKT) Bedingungen

- Stationarity: $\nabla f(\underline{x}_{opt}) + \sum_{i=1}^{m} \mu_i \nabla h_i(\underline{x}_{opt}) + \sum_{i=1}^{p} \lambda_i \nabla g_i(\underline{x}_{opt}) = 0$
- Primal feasability: $g_j(\underline{x}_{opt}) \leq 0, \quad h_i(\underline{x}_{opt}) = 0, \quad i = 1, \ldots, m \quad j = 1, \ldots, p$
- Dual feasability: $\lambda_j \geq 0, \quad j = 1, \ldots, p$
- Complementary slackness: $\lambda_j g_j(\underline{x}_{opt}) = 0, \quad j = 1, \ldots, p$

3 Modellierung von Systemen und Prozessen

3.1 Hardwaremodellierung

- **SystemC**: Systembeschreibungssprache (SLDL, engl. System Level Design Language)

3.2 Petri-Netze

3.2.1 Allgemein

- Auch P/T-Netze und S/T-Netze genannt (Plätze, Stellen, Transitionen)
- Dienen der Modellierung von nebenläufigen, asynchronen, verteilten, parallelen, nichtdeterministischen und/oder stochastischen Systemen
- Stellen ein graphisches und mathematisches Werkzeug dar
- Nebenläufigkeit ist nicht Gleichzeitigkeit, sondern Unabhängigkeit: Petri-Netze haben keine Zeitdimension!

3.2.2 Defintionen

- **Strenge Transitionsregel**: „Eine Transition darf nur dann feuern, wenn dadurch die Kapazität der Ausgangsstellen nicht überschritten wird“
- **Quellen-Transitionen**: Eine Transition ohne Eingangsstellen (immer aktiv)
- **Senken-Transitionen**: Eine Transition ohne Ausgangsstellen
- **Schleifen**: Ein Paar aus Stelle p und Transition t, bei dem p sowohl Eingangs- als auch Ausgangsstelle von t ist.

- **Pures Petri-Netz**: Ein Petri-Netz, dass keine Schleifen besitzt.
- **Gewöhnliches Petri-Netz**: Alle Kanten haben das Gewicht 1.
- **Sicheres Petri-Netz**: Ein Petri-Netz, das ist 1-beschränkt ist.
- **Kapazität $K(p)$:** Anzahl der Tokens die in einer Stelle gespeichert werden können.
- **Petri-Netz endlicher Kapazität:** Nur endlich viele Tokens können in p gespeichert werden.
- **Petri-Netz unendlicher Kapazität:** Unendlich viele Tokens können in p gespeichert werden.
- **Nebenläufigkeit**: Zwei Transitionen heißen nebenläufig, wenn beide aktiviert sind und sie keine gemeinsamen Aus- und Eingangsstellen besitzen.
- **Konflikt**: Zwei Transition enstehen im Konflikt, wenn beide aktiviert sind und (wenigstens) eine gemeinsame Vor- oder Nachbedingung erfüllt ist.
- **Konfusion**: Konfusion ist ein potentieller Konflikt, der durch die willkürliche Reihenfolge des Feuerns zweier nebenläufiger Transitionen entstehen könnte. Konfusion liegt immer dann vor, wenn die Reihenfolge des Feuerns zweier nebenläufiger Transitionen für den weiteren Ablauf des Systems wesentlich ist.
- **Deadlock**: Nebenläufigkeit kann zum Stillstand führen, d.h., zu einem Zustand, in dem keine Transition feuern kann.

3.2.3 Formale Darstellung

- als 5-Tupel: $PN = (P, T, F, W, M_0)$
- endliche Menge von Stellen (Plätzen): $P = \{p_1, p_2, \ldots, p_m\}$
- endliche Menge von Transitionen: $T = \{t_1, t_2, \ldots, t_n\}$
- Menge von Kanten: $F \subseteq (P \times T) \cup (T \times P)$
- Gewichtungsfunktion: $W : F \to \{1, 2, 3, \ldots\}$
- Anfangsmarkierung: $M_o : P \to \{0, 1, 2, 3, \ldots\}$
- Es gilt ferner: $B \cap T = \emptyset$ und $P \cup T = \emptyset$ (Graph ist bipartit)

3.2.4 Dynamische Eigenschaften von Petri-Netzen

von Anfangmarkierung M_0 Abhängig

- **Erreichbarkeit**: Eine Markierung M_n ist erreichbar, wenn es eine Folge von Transitionen gibt, die M_0 in M_n überführen kann.
  - $M_n$ist von M_0 erreichbar, wenn es im im Überdeckbarkeitsbaum einen Pfad von M_0 nach $M_n$exisiert für den man nicht in Richtung der Wurzel laufen muss.
- **Beschränktheit:** Ein Petri-Netz ist k-beschränkt, falls keine Markierung, die von M_0 aus erreicht werden kann, mehr als k Tokens pro Stelle hat.
  - wenn die Knotenbezeichner des Überdeckbarkeitsbaums kein ω enthalten.
- **Lebendigkeit:** Petri-Netz ist lebendig, wenn es - ungeachtet der gerade erreichten Markierung - stets möglich ist jede Transition durch eine Reihe von Feuerungen zu aktivieren (Nachweis ist schwer).
  - Eine Transition ist „tot“ wenn sie nicht als Kante im Überdeckbarkeitsbaums exisistiert.
- **Umkehrbarkeit:** Die Anfangsmarkierung M_0 kann von jeder Markierung M aus erreicht werden (Wichtig: M musste zuvor von der Anfangsmarkierung aus erreichbar sein)

- **Überdeckbarkeit**: Eine Markierung M ist Überdeckbar, wenn eine andere Markierung M' existiert, sodass für jede Stelle gilt: $M'(p) \geq M(p)$
- **Beständigkeit:** Für jedes paar von aktivierten Transitionen gilt, dass das feuern einer dieser Transitionen die anderen nicht deaktiviert.
- **Sicherheit**:
  - wenn die Knotenbezeichner des Überdeckbarkeitsbaums nur Nullen oder Einsen enthalten.
- **Fairness:** Nicht behandelt

3.3 Kahn-Prozessnetzwerke (KPN)

- KPN dienen der Modellierung von Systemen bestehend aus Prozessen
- Prozesse
  - kommunizieren ausschließlich über FIFOs („message passing“ vgl. Erlang)
  - bestehen aus sequentiellen Anweisungen
  - produzieren und/oder konsumieren dabei eine festgelegte Anzahl von Tokens
- Nicht-blockierendes Schreiben
  - Produzent muss nicht warten, bis Konsument liest
  - FIFO-Länge unbegrenzt
- Blockierendes Lesen
  - Konsument muss bei leerem FIFO warten bis nächstes Token vorliegt
  - Prozesse können nur an einem Kanal gleichzeitig warten
  - Kein Lese-Zugriff auf einen Kanal, ohne dass ein Token konsumiert wird
- Kahn-Prozess-Netzwerke sind deterministisch
  - Folge der Ausgabetokens ist nicht von der Ausführungsreihenfolge abhängig $\Rightarrow$
  - Ausführungsreihenfolge der Prozesse ist irrelevant
- Scheduling der Prozesse
  - Anhäufung von Tokens muss vermieden werden
  - Bei begrenzenten FIFO Längen können Deadlocks entstehen
    - wenn Ausgangs-FIFO voll
    - wenn Eingangs-FIFO leer
  - Algorithmus vin Park (1995)
    - vermeidet Deadlocks
    - Scheduling für begrenzte FIFOs mit möglichst kleiner Länge

3.3.1 Synchrone Datenflussgraphen (SDF)

- SDF sind ein Spezialfall von KPN
  - Knoten: Prozesse
  - Kanten: Datenpfade
- Die Anzahl der gefeuerten/konsumierten Tokens ist im Vorhinein festgelegt (vgl. Petri-Netze) $\Rightarrow$ Scheduling kann vorher festgelegt werden
- SDF eigenen sich zur Modellierung vieler Algorithmen in der digitalen Signalverarbeitung (DSP)
- **Homogene SDF**: Alle Knoten konsumieren und produzieren jeweils nur ein Token an den Eingangs- und Ausgangskanten
- **Inhomogene SDF**: Knoten können unterscheidlich viele Tokens konsumieren bzw. produzieren

3.4 Zustandsautomaten, Turing Maschinen, Komplexitätsklassen

3.4.1 Turing Maschine

Die Turingmaschine stellt eine Erweiterung des endlichen Zustandsautomaten (FSM, Mealy & Moore, vgl Info2) dar

- Endliche Menge an einnehmbaren Zuständen
- Ein (einseitig) unendliches Band
- Bewegbarer Lese/Schreib-Kopf zuum Lesen und Schreiben von Symbolen auf dem Band
- Zustandsbasierte Steuerung (Programm)

Turing-Tabelle:

δ	read	write	move	new state	note

- **Curch-Turing-These**: Die Klasse der Turing-berechenbaren Funktionen ist genau die Klasse der intuitiv berechenbaren Funktionen.
- **Turing-berechenbar**: Es existiert eine Turing Machine, die für Eingaben aus dem Definitonsbereich anhält, sodass im Anschliuss der entsrechende Funktionswert auf dem Band steht.
- **Turing-vollständig**: Ein Modell, dass ebenfalls alle Turing-berechenbaren Funktionen berechnen kann.
- **Nicht deterministische Turing-Maschine (NTM):** Falls nicht eindeutige Übergangsrelationen von einem Zustand in den nächsten Zustand vorliegen, muss sich die Turing-Maschine verzweigen und parallel jeden dieser Wege weiterverfolgen. Kommt einer dieser Zweige zu einem Halt, so ist die Lösung akzeptiert

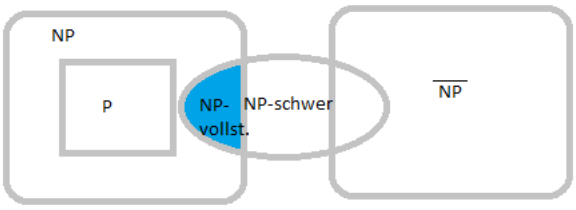
3.4.2 7-Tupel der Touringmaschine

- S: endliche Menge von Zuständen
- Σ: Elngabealphabeth
- Γ: Bandalphabet, mit $\Sigma \in \Gamma$ und $\square \in \Gamma$
- $\delta : (S \backslash F \times \Gamma) \rightarrow (S \times \Gamma \times \{R, L\})$: Übergangsfunktion (R,L: Rechts- oder Linksshift des Lese/Schreibkopfes)
- $s_0 \in S$: Anfangszustand
- $\square$: Leeres Symbol
- F: Menge der Endzustände mit $F \subseteq S$

3.4.3 Komplexitätsklassen

Klassifizieren Probleme anhand ihrer Komplexität. Übliche Metriken sind **Laufzeit**, **Speicherbedarf**, etc..

- **Komplexitätsklasse P**: Ist in Polynomialzeit von deterministischer TM lösbar. Probleme dieser Klassen gelten als praktisch (effizient) berechenbar.
- **Komplexitätklasse NP**: Ist in Polynomialzeit von nicht deterministischer TM lösbar. $P \subseteq NP$, jedes Problem in P ist auch Teil von NP.
- **NP-schweres Problem**: Ein Problem ist NP-schwer, wenn sich jedes Problem in NP in polyniomaler Zeit auf dieses Problem zurückführen lässt.
- **NP-vollständiges Problem**: Ist ein Problem NP-schwer und selbst in NP enthalten, ist es NP-vollständig.



4 Betriebssysteme

- **Definition**: Das Betriebssystem ist eine softwareseitige Erweiterung der Hardware, welche diese abstrahiert und einfache Zugriffsmechanismen zur Verfügung stellt. Daher verwaltet das Betriebssystm zur Verfügung stehende Ressourcen und deren Zuteilung.
- **Aufgaben**:
  - Speicherverwaltung
  - Programm-Verwaltung / Prozess-Verwaltung
  - Geräte- und Dateiverwaltung / Ein- Ausgabe
  - Rechteverwaltung / Sicherheit
  - Abstraktion / Kompatibilitätschicht

4.1 Prozesse & Threads

- **Prozesse**: Prozesse sind ablaufende Programme mit eigenem Befehlszähler, Register und Speicherbereich und dienen dazu (Pseudo-)Nebenläufigkeit eines Systems zu ermöglichen. Sie besitzen in ihrem Adressraum nur einen Ausführungsfaden (**thread-of-control**).
- **Threads**: Mehrere Ausführungsfäden in einem Prozess, die sich einen gemeinsamen Adressraum teilen. Threads sind in ihrer Erzeugung, Scheduling und der Kommunikation untereinander leichtgewichtiger als Prozesse.
- Vergleich

Prozess Elemente	Thread Elemente
Adressraum	Befehlszähler
Globale Variabeln	Register
Offene Dateien/Verbindungen	Stack
Kindprozesse	Zustand
Ausstehende Signale	
Signale und Signalroutinen	
Verwaltungsinformationen	

4.2 Scheduling/Ablaufplanung

- **Definition**: Die Ablaufplanung (engl. Scheduling) bestimmt die räumliche und zeitliche Zuordnung von Aktivitäten zu auszuführenden Instanzen.
- **Echtzeitbetriebssystem/RTOS**: Scheduling mit Sollzeitpunkten/Deadlines.

4.2.1 Algorithmen

Unterscheiden sich hauptsächlich durch die Aufrufreihenfolge der Prozesse und deren Verdrängung/Preemption

- **First-Come First-Serve (FIFO)**
  - Reihenfolge: Abarbeitung in Ankunftsreihenfolge (FIFO-Liste)
  - keine Verdrängung: Prozess wird abgearbeitet bis er sich selbst unterbricht oder beendet.
- **Round-Robin**
  - Reihenfolge: Abarbeitung in Ankunftsreihenfolge (FIFO-Liste)
  - Verdrängung: Prozesswechsel mindestens nach vorgeschriebener/fester Zeit δ.
- **Prioritäts-basiert (ohne Verdrängung)**
  - Reihenfolge: nach Prioritäten sortierte Liste
  - keine Verdrängung: Prozess wird abgearbeitet bis er sich selbst unterbricht oder beendet.
- **Prioritäts-basiert (mit Verdrängung)**
  - Reihenfolge: nach Prioritäten sortierte Liste
  - Verdrängung: Prozess wird abgearbeitet bis er sich selbst unterbricht, beendet oder ein Prozess mit höherer Priorität eintrifft.

4.3 Interprozess-Kommunikation

- **Race Conditions**: Wettlaufsituationen können auf treten, wenn zwei oder mehr Prozesse auf gemeinsame Daten zugreifen (lesen oder schreiben), so dass das Endergebnis von der exakten Ausführungsreihenfolge abhängt (vgl. Konfusion der Petri-Netze).
- **Mutual exclusion**: Race Conditions können durch wechselseitigen Ausschluss verhindert werden. Gängige Mechanismen sind dabei:
  - Aktives Warten / Spinlocks
  - **Semaphore / Mutex (binäres Semaphore)**
  - Monitor
  - Message Passing
  - Barriere
  - Lock-Mechanismen

4.4 Speicher Management

siehe auch Speicherarchitekturen

- **Memory Management Unit**:
  - Bildet logische Speicheradressen auf physikalische Adressen ab.
  - Versteckt dadurch Prozess-fremde Speicherbereiche.
  - Implementierung mittels Mase/Limit Paar oder Seiten-basiert (Paging)

5 Multi-Prozessorsysteme

- **Homogenes Multi-Prozessorsystem**: viele gleichartige PE, hohe Flexibilität, geringe Performence
- **Heterogenes Multi-Prozessorsystem**: verschiede PE, anwendungsspezifische Konfiguration, gute Energieeffizienz & Performence
- **Geschwindigkeitsgewinn (Speedup) durch Parallelisierung**: $S = \frac{T_S}{T_P}$
- **Amdahl's Law**: $T_p = T_S(1 - p + \frac{p}{n}) \Rightarrow S = \frac{1}{1-p+\frac{p}{n}}$
  - mit n Prozessoren, p Anteil an parallelisierbarem Code
  - Maximaler Geschwindigkeitsgewinn: $S(n \to \infty) = \frac{1}{1-p}$
  - Optimaler Geschwindigkeitsgewinn: $S(p = 1) = n$
  - Realistischer Geschwindigkeitsgewinn: $S(p = 0.9) = 10$ (Beispiel)

5.1 Prozessorarten/Processing Elements (PE)

- General Purpose Processor (GPP)
- Digital Signal Processor (DSP)
- Application Specific Instructionset Processor (ASIP)
- Application Specific Integrated Circuit (ASIC)
- Field Programmable Gate Array (FPGA)

5.2 Kommunikationsarchitekturen

- Direkte Verbindung
- Geteilter Speicherbereich
- Bus-System
- Crossbar
- Network on Chip (NoC)

5.3 Speicherarchitekturen

- **Speicherhierachie/pyramide**: zunehmende Kapazität $\longleftrightarrow$ abnehmende Zugriffszeit
  - Prozessor-Register
  - L1/L2-Cache
  - Hauptspeicher
  - Flash-Speicher
  - Hard-Disks
  - Bänder/CDs
- **Cache Kohärenz Problem:** Durch die Sicherstellung von Cache-Kohärenz wird bei Mehrprozessorsystemen mit mehreren CPU-Caches verhindert, dass die einzelnen Caches für die gleiche Speicheradresse unterschiedliche (inkonsistente) Daten zurückliefern.
- Cache Kohärenz Protokolle:
  - Snooping-basiert: alle Cache Controller lauschen auf einem gemeinsamen Interconnect und verwalten den Status der Blöcke im lokalen Cache.
  - Verzeichnis-basiert: zentrales Verzeichnis aller gecachten Blöcke und die darauf zugreifenden Prozessoren mit ihrem Status (exlusive, shared).

6 Einführung Netzwerke

- **Computer Netzwerk:** Ein Sammlung von verbundenen Computern, die in der Lage sind, untereinander Informationen auszutauschen.
- **Ziele**:
  - Ortsunabhängigkeit
  - hohe Verfügbarkeit/Redunanz
  - hohe Energie- und Kosteneffizienz für Mainframe Rechner
- **Service:** Sammlung von primitiven Operationen von unteren Schichten oberen Schichten verfügbar gemacht werden
- **Protocol:** Satz von Regeln, die die Kommunikation innerhalb einer Schicht definieren.
- **Circuit Switching:** Vor der Datenübertragung muss eine fest definierte Verbindung zwischen beiden Kommunikationspartnern aufgebaut werden, die erst nach dem Ende der Übertragung wieder geschlossen wird
- **Packet Switching:** Ohne einen expliziten Verbindungsaufbau werden die Daten in Form von kleinen Paketen, die in ihrem Header die Zieladresse enthalten über spontan gewählte Übertragungswege vom Absender zum Empfänger versandt

6.1 OSI Referenz Modell

7. Application-Layer (Anwendungsschicht)

- Verschafft Anwendungen Zugriff auf das Netz
- **Protokolle**: Telnet, SMTP, HTTP, LDAP, FTP, NFS, SSH

6. Presentation-Layer (Darstellungsschicht)

- Kommunikation zwischen Systemen
- Definition Abstrakter Datenstrukturen
- Datenkompression, Verschlüsselung, Übersetzer zwischen Datenformaten
- **Protokolle**: ISO 8822/X.216, ASN.1

5. Session-Layer (Sitzungsschicht)

- Prozesskommunikation, Dialogkontrolle, Token management, Synchronisation
- Widerherstellung nach Sitzungszusammenbruch durch bereitgestellte Dienste
- **Protokolle**: RPC, ISO 8306/X.215

4. Transport-Layer (Transportschicht)

- Segmentierung des Datenstroms
- Ende-zu-Ende-Kontrolle
- Bietet den anwendungsorientierten Layern 5-7 einen einheitlichen Zugriff
- **Protokolle**: TCP, UDP, SCTP, ISO 8073/X.224

3. Network-Layer (Vermittlungsschicht)

- Schalten von Verbindungen und Vermittlung von Paketen
- Wegsuche / Routing
- Bereitstellen netzwerkübergreifender Adressen
- **Protokolle**: IP, IPsec, X.25, ISO 8208, CLNP, ESIS, ICMP
- **Hardware**: Router, Layer-3-Switch (BRouter)

2. Data-Link-Layer (Sicherungsschicht)

- lässt sich gliedern in Medium Access Control (MAC) und Logical Link Control (LLC)
- Gewährleistung fehlerfreier Übertragung
- Aufteilen des Datenstroms in Blöcke/Frames
- Hinzufügen von Folgenummern und Prüfsummen
- Daten Fusskontrolle
- **Protokolle**: Ethernet, CSMA/CD, WLAN, Token Bus, Token Ring, FDDI
- **Hardware**: Bridge, Switch (Multiport-Bridge)

1. Physical-Layer (Bitübertragungsschicht)

- Hat die Aufgabe rohen Bitstream über einen physikalischen Kanal zu transportieren
- Definiert die Interfaces des physikalisches Mediums (kabelgebunden, schurlos), die Signalübertragung und die Art des hardwareseitigen Verbindungsaufbaus (**Circuit Switching, Packet Switching)**
- **Protokolle**: V.24, V.28, X.21, RS 232 (Seriell), RS 422, RS 423, RS 499
- **Hardware**: Modem, Hub, Repeater

Teil II
Algorithmen

1 Zuordnungsprobleme in Graphen

1.1 Bestimmen einer maximalen Zuordnung (ungewichteter bipartiter Graph)

1. Wähle einen freien Knoten
2. Suche ausgehend von diesem Knoten alle alternierenden Pfade
   (a) Ist der gefundene Pfad ein vergrößerdner Pfad vergrößere die Zuordnung durch Austausch von freien und gebunden Kanten
   (b) Tritt eine Schleife auf, wähle anderen Pfad. Gibt es keinen solchen, fahre fort mit 1.)
3. Wurden alle Knoten geprüft, ist die Zuordnung maximal.

1.2 Bestimmen einer maximalen Zuordnung (gewichteter bipartiter Graph)

Eine Zuordnung mit minimalen Gewicht kann gefunden werden, wenn man das Maximum m aller Zellen bestimmt und jeden Zellenwert c_{ij} von diesem Wert abzieht: $c'_{ij} = m - c_{ij}$

1. Wähle y_i als das Maximum einer **Zeile** und $y_j = 0$
2. Erzeuge **Überschuss-Matrix**: $e_{ij} = y_i + y_j - c_{ij}$ (Alle Spalten, die im vorherigen Durchlauf durchgestrichen wurden, ändern sich nicht)
3. Markiere alle Zellen, für die gilt: $e_{ij} = 0$. Zeichne für all diese Zellen die Kanten von y_i nach y_j in einen sogenannten **Gleichheitsgraphen** G.
   (a) Bestimme die **maximale Zuordnung** M des Gleichheitsgraphen G, ist diese perfekt wurde die Lösung gefunden.
   (b) Finde eine **Knotenüberdeckung** W mit der Größe $|M|$ im Gleichheitsgraphen G
   (c) **Satz von König**: Überdeckt die Knotenüberdeckung W alle Kanten des Graphen, gibt es hingegen keine perfekte Zuordnung!
4. Streiche in der ehemaligen Überschuss-Matrix alle Spalten die eine 0 enthielten durch und ermittle aus aus den verbleibenden Zellen das Mimimum m.
   (a) Berechne die neuen y_i zu $y'_i = y_i - m \in V_2 \setminus W$ und $y'_j = y_j + m \, \forall y_j \in V_1 \cap W$.
   (b) Gehe wieder zu 2.) solange keine perfekte oder maximale Zuordung im Gleichheitsgraphen exisitiert.

1.3 Bestimmen einer maximalen Zuordnung (in allgemeinen Graphen, Algorithmus von Edmonds)

2 Balancierte Bäume

2.1 AVL-Baum

2.1.1 Einfügen

1. Einfügeposition mit binärer Suche bestimmen und Knoten einfügen.
2. Ausgehend von Einfügeposition nach unbalancierten Teilbäumen suchen und diese balancieren.

2.1.2 Entfernen

1. Wenn Knoten ein Blatt ist, entferne ihn
   (a) Ist der zu entfernende Knoten ein Blatt, entferne ihn.
   (b) Hat der zu entfernende Knoten nur ein Kind, entferne den Knoten und ersetze ihn duch sein Kind.
   (c) Sind beide Kinder des zu entfernenden Knotens Blätter, ersetze den Knoten durch eines seiner Blätter (beliebig).
   (d) Ist mindestens eines der beiden Kinder des zu entfernenden Knotens ein innerer Knoten:
      i. wähle den symmetrischen Knoten SR
         A. als Nachfolger (Balance des zu entfernenden Knotens < 0)
         B. als Vorgänger (Balance des zu entfernenden Knotens ≥ 0)
      ii. übertrage den Schlüssel von SR auf den zu entfernenden Knoten
      iii. und lösche das Blatt SR

Balanciere den Baum nach jedem Schritt erneut!

2.1.3 Balancieren

Es werden 4 Fälle unterschieden:

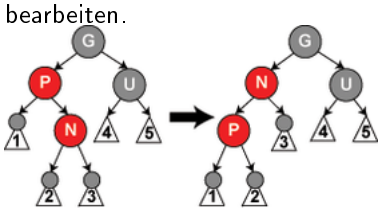
- left-of-left: Rechts-Rotation nötig
- right-of-left: Links-Rotation mit anschliessender Rechts-Rotation nötig
- right-of-right: Links-Rotation nötig
- left-of-right: Rechts-Rotation mit anschliessender Links-Rotation nötig

2.2 Rot-Schwarz Baum

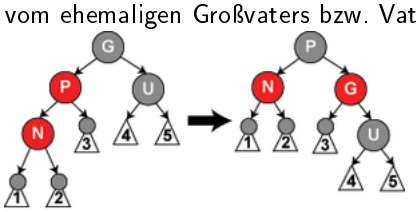
2.2.1 Einfügen

Neu eingefügte Knoten sind immer rot!

1. Der Baum ist leer: Füge den Knoten direkt ein und färbe ihn schwarz.
2. Der Vaterknoten ist schwarz: Füge den Knoten direkt ein
3. Vater-Knoten und Onkel-Knoten sind beide rot: Ändere die Farben dieser beiden Knoten in schwarz. Färbe den Großvaterknoten rot und betrachte ihn wie einen neu eingefügten Knoten bezüglich seiner Vorgänger
4. Der neue Knoten hat einen schwarzen oder keinen Onkel und ist das rechte Kind seines roten Vaters während der Vater jedoch links am Großvater sitzt: Führe eine linksrotation um den Vater aus, welche die Rolle des einzufügenden Knotens und seines Vaters vertauscht. Danach kann man den ehemaligen Vaterknoten mit Hilfe des fünften Falles bearbeiten.



5. Der neue Knoten hat einen schwarzen oder keinen Onkel und ist das linke Kind seines roten Vaters; der Vater hängt auch links am Großvater: Zunächst rechtsrotation um den Großvater, wodurch der ursprüngliche Vater nun der Vater von sowohl dem neu einzufügenden Knoten als auch dem ehemaligen Großvater ist. Im Anschluss Farben vom ehemaligen Großvaters bzw. Vaters tauschen.



2.2.2 Entfernen

1. Knoten ist rot und hat nur Blätter als Kindknoten: Lösche den Knoten
2. Knoten ist schwarz und hat nur einen roten Kind-Knoten: Ersetze ihn durch seinen Kindknoten und färbe diesen schwarz
3. Knoten ist schwarz und hat nur Blätter als Kind-Knoten: Entferne den Knoten. Anschließend folgen Rotationen und Farbänderungen (Drehen & Umfärben, falls noch weitere Unterebenen vorhanden, sonst nur Umfärben)
4. Knoten (N) hat zwei innere Knoten als Kindknoten: Bestimme den symmetrischen Vorgänger oder Nachfolger und verschiebe die Werte von R nach N. Das Entfernen von R entspricht den Fällen 1-3.

2.3 B-Baum

2.3.1 Einfügen

1. Solange Einfügen, bis Anzahl Elemente in Knoten gleich m-1 ist
2. Dann überfüllten Knoten in linke und rechte Hälfte aufteilen und inneren Knoten hochschieben
3. Führt dies zu einer Überfüllung des oberen Knoten wiederhole für diesen Knoten Schritte 1-3

2.3.2 Löschen

1. Ist der Knoten aus dem ein Element gelöscht wird nach dem Löschen nicht leer, so werden die linken und rechten Kindknoten des gelöschten Knotens zusammengefasst.
2. Ist der Knoten nach dem Löschen leer, so wandert das Vaterelement in den leeren Knoten und wird mit seinem Geschwisterknoten zusammengefasst. Hierbei werden die Regeln des Einfügens beachtet.

3 Suchalgorithmen

Tabelle:

#	m_u	m_o	m	$a[m].key$	$a[m].key \overset{?}{\lessgtr} k$

3.1 Binäre Suche

- Algorithmus: $m = m_u + \frac{1}{2}(m_o - m_u)$
- Der Suchbereich halbiert sich (näherungsweise) bei jedem Schritt
- Durchschnittliche Laufzeit: $O(log\, N)$

3.2 Interpolationssuche

- Algorithmus: $m = m_u + \frac{k - a[m_u].key}{a[m_o].key - a[m_u].key}(m_o - m_u)$
- Durchschnittliche Laufzeit: $O(log_2 log_2 N)$ (bei gleichverteilten Schlüsseln)
- Worst Case Laufzeit: $O(N)$ (bei exponentiell ansteigenden Schlüsseln)

4 Hashverfahren

- Kollisionsbehandlung
  - Überläufer in verketteter Liste speichern
  - Überläufer durch Sondierungsreihe verschieben

5 Suchen in Texten

5.1 Naive Zeichenketten-Suche

1. Beginne mit dem ersten Zeichen des Textes und fahre fort bis zum Zeichen a_{N-M}
2. Vergleiche das Muster zeichenweise mit dem Text.
   (a) Tritt ein Mismatch auf, verschiebe das Muster auf das Nächste Zeichen und beginne wieder bei 2.)
   (b) Wurde das Ende des Musters erreicht, wurde ein Vorkommen gefunden.

5.2 Knuth-Morris-Pratt Algorithmus

1. Aufbau einer Verschiebungstabelle: $m, W, T[m]$
   (a) m: Index Muster
   (b) W: Muster
   (c) $T[m]$: Anzahl der Zeichen, die ein Teil des Musters bereits mit dem Musteranfang übereinstimmt.
2.

6 Optimierungsprobleme & -verfahren

6.1 LP $\rightarrow$ SDP

$$\begin{cases} min & \underline{c}^T \underline{x} \\ s.t. & A\underline{x} \leq \underline{b} \end{cases} \Rightarrow \begin{cases} min & \underline{c}^T \underline{x} \\ s.t. & -x_1 diag(A_1) - \ldots - x_n diag(A_n) + diag(\underline{b}) \succeq 0 \end{cases}$$

mit $A_k$als Spaltenvektoren von A

6.2 QP $\rightarrow$ SDP

skalarer Fall $\underline{x} = x_1$

1. Einführen einer zusätzlichen Optimierungsvariablen x_2
$$\begin{cases} min & x_2 \\ s.t. & \frac{1}{2} x_1 P x_1 + q x_1 \leq x_2 \\ & G_1 x_1 \leq h_1 \\ & A_1 x_1 = b_1 \end{cases}$$
2. Beschreibung der Ungleichheitsbedingungen durch eine positiv semidefinite Matrix
3. Darstellung in SDP Standardform

6.3 Lagrange-Multiplikatoren

6.4 Gradientenvefahren

6.5 Newton'sches Verfahren

6.6 Branch-and-Bound

7 Event-basierter Simulationskernel

Tabelle:

#	Event Auslöser	Simulierte Zeit	Evaluation Zeilen Nr.	Updates	ausgelöste Events

- Positive/steigende Flanke: `clk.pos()`
- Negative/fallende Flanke: `clk.neg()`

8 Petri-Netze

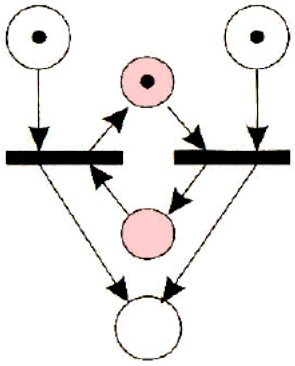
8.1 Komplementäre-Stellen Transformation

Die Komplementäre-Stellen Transformation beschreibt eine Umwandlung eines pures Petri-Netz **endlicher** Kapazität in ein Petri-Netz **unendlicher** Kapazität.

Ziel der Umwandlung ist es, die Begrenzung der Kapazität aufzugeben. Gleichzeitig soll sich das Verhalten des Netzes dabei nicht ändern.

1. Einführen einer komplementären Stelle p' für jede Stelle p mit $M_0'(p') = K(p) - M_0(p)$
2. Einführen von Kanten (t, p') bzw. (p', t) zwischen jeder Transition und einigen komplementären Stellen p', so dass die Summe der Tokens in p und p' vor und nach dem Feuern von t gleich $K(p)$ ist. Dabei gilt: $w(t, p') = w(p, t)$ bzw. $w(p', t) = w(t, p)$.

8.2 Lösung von Konflikten durch zusätzliche Eingabestellen



Rot markierte Stellen zwingen das Petri-Netz zwischen beiden Transitionen zu togglen.

8.3 Konstruktion eines Überdeckbarkeitsbaum

1. Anfangsmarkierung M_0ist Wurzel und wird als „**neu**“ gekennzeichnet
2. Solange „**neue**“ Markierungen existieren:
   (a) Wähle eine neue Markierung M
   (b) Wenn M identisch ist zu einer Markierung auf dem Pfad von der Wurzel zu M, dann kennzeichne M als „alt“ und wähle eine andere Markierung.
   (c) Solange es in M aktivierte Transitionen gibt, tue folgendes für jede aktivierte Transition t:
      i. Ermittele Markierung M', die aus Feuern von t resultiert
      ii. Wenn auf dem Pfad von der Wurzel zu M eine Markierung M” mit $M'(p) \geq M$”(p) für jede Stelle p und $M' \neq M$” existiert, ersetze $M'(p)$ mit ω für jedes p mit $M'(p) > M$”(p)
      iii. Führe einen Knoten für M' sowie eine Kante t von M nach M' ein und kennzeichne M' als „neu“

Beim Zeichnen des Baumes, darf man erst aufhören, wenn sich die Markierungen wiederholen.

8.4 Konstruktion eines Überdeckbarkeitsgraphen

1. Konstruiere gerichteten Graphen $G = (V, E)$ mit
   - Knotenmenge V: Menge aller verschiedenen Knoten im Überdeckbarkeitsbaum
   - Kantenmenge E: Menge aller Kanten, die einzelne Transition t_k repräsentieren, für die $M_i[t_k > M_j$gilt, wobei $M_i$und M_j Knoten aus V sind