1 Vertiefung Datenstrukturen und Algorithmen

1.1 Zuordnungsprobleme in Graphen

1.1.1 Allgemeine Definitionen

- **Graph**: Skript S. 6 / Folie 7)
- **Bipartiter gewichteter Graph:** Skript S. 13 / Folie 21
- **Zuordnung**: Elne Zuordnung M für einen ungerichteten Graphen $G = (V, E)$ ist eine Teilmenge $M \subseteq E$ der Kanten (E), sodass keine Kanten denselben Endknoten haben
- **Nicht erweiterbare Zuordnung (Maximal Matching):** Eine Zuordnung M 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 Zurodnung (Maximum Matchin):** Eine nicht erweiterbare Zuordnung mit maximaler Größe $|M|$
- **Perfekte Zuordnng:** 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(mit $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$) zerlegen zu können, dass keine Kante zwei Knoten in V_1oder zwei Knoten in V_2verbindet. ($E \subseteq \{\{v_1, v_2\} | v_1 \in V_1, v_2 \in V_2\}$
- **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.

1.1.2 Maximale Zurodnungen

- **Gebundene Kante:** Kante $e \in M$
- **Freie Kante:** Kante $e \in E$ aber $e \notin M$
- **Gebundener Knoten:** Knoten der mit einer gebundenen Kante verbunden ist
- **Freier Knoten:** Knoten der mti einer freien Kante verbunden ist
- **Alternierender Pfad:** Kanten sind abwechselnd gebunden und frei
- **Vergrößernder Pfad:** Pfad, dessen Anfangs- und Endknoten freie Knoten sind
- **Blüte:** Eine Blüte ist eine Schleife in G, die aus 2n+1 Kanten besteht, von denen genau n Kanten gebunden sind (zur Zuordnung M gehören)
- **Kontrahierter Graph:** Ein kontrahierter Graph ist ein Graph, der aus G enststeht, indem Blüten zu Knoten v_B zusammengezogen werden.

1.1.3 Bestimmen einer maximalen Zuordnung (ungewichteter bipartiter Graph):

1. Suche der Reihe nach freie 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.

1.1.4 Bestimmen eienr maximalen Zuordnung (gewichteter bipartiter Graph):

Für gutes Beispiel siehe Skript S. 15-17 / Folien 25-30. Eine Zuordnung mit minimalen Gewicht kann erstellt werden, wenn man das Maximum m aller Zellen bestimmt und jeden Zelenwert 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, indem man in jeder Zeile i den Zellenwert c_{ij} von y_i abzieht.
3. Markiere alle Zellen, die 0 ergeben. Zeichne für all diese Zellen die Kanten von y_i nach y_j in einen sogenannten Gleichheitsgraphen
4. Bestimme die maximale Zuordnung
5. Streiche in der ehemaligen Überschuss-Matrix alle Spalten die eine 0 enthielten durch und ermittle aus aus den verbleibenden Zellen das Mimimum m. Berechne die neuen y zu $y'_i = y_i - m$, $y'_j = y_j + m$.
6. Berechne die neue Überschussmatrix durch $c'_{ij} = y_i + y_j - c_{ij}$ (wobei die c_{ij} die originalen Zellenwerte aus 2.) sind.
7. Fahre fort mit 3.) Ist die Zuordnung perfekt, ist die Lösung gefunden. Es gibt hingegen keine perfekte Zuordnung, wenn es eine Knotenüberdeckung gib, die alle Kanten in G überdeckt

1.1.5 Maximale Zuordnung in allgemeinen Graphen (ungewichtet):

- Für Vorgehen siehe Beispiel S. 19f. /Folien 34-36

1.2 Balancierte Bäume:

1.2.1 Definitionen:

- **Balancierter Baum:** Ein Baum, bei dem kein Blatt wesentlich weiter von der Wurzel entfernt ist, als ein anderes Blatt des Baumes (maximale Höhe des Baumes $c \cdot \log(n)$ mit n="Anzahl der Elemente des Baumes" und c eine beliebige Konstante)

- **Binärbaum:** Baum, bei dem jeder Knoten maximal zwei Kindnoten hat
- **Balance eines Binärbaumes:** Höhenunterschied zwischen dem linken (B_L) und dem rechten (B_R) Teilbaum eines Knotens bzw. der Wurzel (Balance $= H(B_L) - H(B_R)$)
- **Suchbaum:** Skript S. 22 / Folie 39
- **AVL-Baum:** AVL-Baum ist binärer Suchbaum, bei dem sich die Höhe der beiden Teilbäume maximal um 1 unterscheidet => die Balance ist -1, 0, +1.
- **Rot-Schwarz-Baum:** Seite 32 / Folie 60
- **B-Baum:** Seite 37 / Folie 70

1.2.2 Einfügen in einen AVL-Baum:

1. Einfügen, wie in einen Suchbaum, danach Umsortieren, entsprechend der Bilder auf den Seiten 25-27 / Folien 45-49.

1.2.3 Entfernen aus einem AVL-Baum

1. Wenn Knoten ein Blatt ist, entferne ihn
2. Hat der zu entfernende Knoten nur ein Kind, entferne den Knoten und ersetze ihn duch sein Kind
3. Sind beide Kinder des zu entfernenden Knotens Blätter, ersetze den Knoten durch eines seiner Blätter
4. Ist mindestens eines der beiden Kinder des zu entfernenden Knotens ein innterer Knoten:
   (a) wähle den symmetrischen Nachfolger (Balance des zu entfernenden Knotens < 0) bzw. Vorgänger (Balance des zu entfernenden Knotens ≥ 0) des Knotens „SR“
   (b) übertrage den Schlüssel von SR auf den aktuellen Knoten
   (c) und lösche den Knoten SR (hierbei können wieder die Fälle 1-4 auftreten)

Balanciere den Baum nach jedem Schritt erneut wie auf Seiten 25-27 / Folien 45-49.

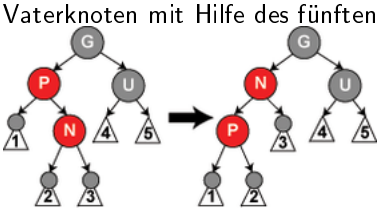
1.2.4 Eigenschaften eines Rot-Schwarz-Baums, die stets zu prüfen sind

- Jeder Knoten ist entweder rot oder schwarz
- Alle Blätter sind schwarz und enthalten keine Daten
- **Wurzel ist immer schwarz**
- **Jeder rote Knoten hat zwei schwarze Kinder**
- **Jeder Pfad von einem gegebenen Knoten zu einem Blattknoten enthält die selbe Anzahl schwarzer Knoten**
- Jeder innere Knoten hat zwei Kindknoten

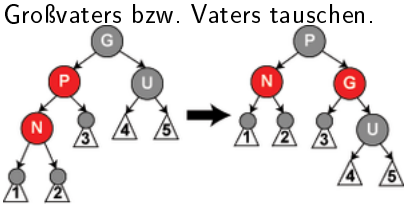
1.2.5 Einfügen in einen Rot-Schwarz-Baum

Neu eingefügte Knoten sind immer rot!

1. Der Baum ist leer: Füge den Knotenn 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.



1.2.6 Entfernen in einem Rot-Schwarz-Baum

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.

1.2.7 Einfügen in einen B-Baum

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

1.2.8 Löschen aus einem B-Baum:

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.

1.3 Suchalgorithmen

1.3.1 Binäre Suche

- Algorithmus: $m = m_u + \frac{1}{2}(m_o - m_u)$ S. 41 / Folie 77
- Der Suchbereich halbiert sich (näherungsweise) bei jedem Schritt
- Durchschnittliche Laufzeit: $O(log\, N)$

1.3.2 Interpolationssuche

- Algorithmus: $m = m_u + \frac{k - a[m_u].key}{a[m_o].key - a[m_u].key}(m_o - m_u)$ S 42 / Folie 79
- Durchschnittliche Laufzeit: $O(log_2 log_2 N)$ (bei gleichverteilten Schlüsseln)
- Worst Case Laufzeit: $O(N)$ (bei exponentiell ansteigenden Schlüsseln)

1.4 Hashverfahren

1.4.1 Ziel:

Der Sinn von Hashtabellen ist es die Position eines Schlüssels nach Möglichkeit aus der Schlüssel berechnen zu können, um somit mit einer Laufzeit von O(1) auf den Eintrag zugreifen zu können.

1.4.2 Definitionen:

- Siehe S. 45 / Folie 85
- **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 heißen Synonyme, falls $h(k) = h(k')$
- **Perfektes Hashing:** Bei einer perfekten Hashfunktion werden Kollisionen vollkommen vermieden (nur möglich, wenn $|K| \leq M$ und K fest und vorher bekannt ist)
- **Universelles Hashing:** S. 47f / Folie 90f.
- **Überläufer:** Ein zu speichernder Eintrag, der auf den gleichen Platz gehasht wird, wie ein anderer Eintrag
- **Primäre Häufung:** Sonierungsfolgen unterschiedlicher Hash-Adressen treffen aufeinander und verlaufen synchron (z.B. beim linearen Sondieren)
- **Sekundäre Häufung:** Schlüssel, die den gleichen Hashwert haben, durchlaufen dieselbe Sondierungsfolge

1.4.3 Kriterien für guters Hashing

1. Möglichst wenige Adresskollisionen
2. Adresskollisionen lassen sich effizient lösen
3. Die Hashfunktion lässt sich O(1) berechnen lässt

1.4.4 Geeignete Hashfunktion (Beispiele) (S. 46 / Folie 88)

- $h(k) = k\, mod(M)$ (mit M="größe der Hashtabelle" und idealerweise eine Primzahl)

1.4.5 Kollisionsbehandlung

Prinzipiell existieren zwei Verfahren, um Überläufer zu speichern:

1. Speicherung von Überläufern außerhalb der Hashtabelle, z.B. mit Hilfe von verketteten Listen (S. 49 / Folie 93f.)
   - Vorteile: Einfache Verwaltung
   - Nachteile: Erhöhter Speicherbedarf (heute allerdings kein Problem mehr)
2. Speicherung von Überläufern innerhalb der Hashtabelle unter Zuhilfenahme von Sondierungsfunktionen (S. 50-52 / Folie 95-99 [Allgemeines, Suchen, Einfügen, Löschen])
   - Vorteile: Geringerer Speicherbedarf, als bei 1.)
   - Nachteile: Entfernen von Einträgen aus der Hashtabelle führt zu Unterbrechung der Sondierungsfolgen => Effizient sinkt

1.4.6 Beispiele für Sondierungsfolgen

Allgemein setzen Hash-Funktionen inklusive Sondierung wie folgt zusammen: $s'_k(n) = (h(k) + s_k(n)) \, mod \, M$

- **Lineares Sondieren:** $s_k(n) = n$
- **Qaudratisches Sondieren:** $s_k(n) = (-1)^{n+1} \lceil \frac{n}{2} \rceil^2$
- **Doppeltes Hashing:** $s_k(n) = n \cdot h'(k)$
  - $h'(k) \neq 0 \forall k$
  - Beispiel: $h'(k) = 1 + k \, mod \, (M-2)$

1.5 Suchen in Texten

Naive Suchalgorithmen haben eine Laufzeit von

- Durchschnittlich: $O(M+N)$ (M: länge des Textes, N: Länge des zu suchenden Wortes)
- Wortst Cast: $O(M \cdot N)$ (M: länge des Textes, N: Länge des zu suchenden Wortes)

1.5.1 Knuth-Morris-Pratt Algorithmus:

Die Idee beim Knuth-Morris-Pratt Algorithmus besteht darin, bereits während des Vergleichs des Textes mit dem Suchstring zu prüfen, ob der Suchstringanfang mehrfach innerhalb des abgesuchten Textes vorkommt. Kommt es nämlich beim Vergleich des Textes mit dem Suchstring zu Unstimmigkeiten, so kann die neue Suche an der Stelle begonnen werden, wo der Suchstring das erste Mehrfachvorkommen gezeigt hat.

Durch diese Idee erreicht der Algorithmus maximal die folgende Komplexität:

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

Der Algorithmus funktioniert dabei wie folgt:

1. Baue eine Verschiebungstabelle (siehe Beispiel):

W	A	B	C	A	B	D	A	B	A
m	0	1	2	3	4	5	6	7	8
T[m]	-1	0	0	0	1	2	0	1	2

   (Setze stets an erster Stelle der Tabelle eine „-1“ und zunächst an allen weiteren Stellen eine Null. Gehe dann die Tabelle durch und prüfe, ob vor der aktuellen Position ein Präfix des Suchstrings vorkommt und gebe die Anzahl der Stellen an, die dieses Präfix umfasst)

2. Verschiebe den Suchstring schrittweise entlang des Textes und berechne bei Unstimmigkeiten den neuen Startwert anhand folgender Formel:

$i' = i + m - T[m]$, wobei i der alte Startwert, i' der neue Startwert und m die Stelle, an der die Unstimmigkeit auftrat, bezeichnet.

2 Optimierungsprobleme - und verfahren

2.1 Nützliche Rechenregeln:

- Eine Matrix ist genau dann positiv definit, wenn gilt:
  - $\mathbf{x^T \cdot A \cdot x > 0}$
  - Eine symmetrische Matrix ist genau dann positiv definit, wenn alle Eigenwerte größer 0 sind
- Rechenregeln für Matrizen:
  - $(A+B)^T = A^T + B^T$
  - $(AB)^T = B^T A^T$
  - $(A^T)^T = A$
- Allgemeine Normdefinition
  - $\mathbb{R}^n \to \mathbb{R}, \, x \mapsto \|\mathbf{x}\|_p = (\sum_i^n |x_i|)^{1/p}$
  - $\mathbb{R}^n \to \mathbb{R}, \, x \mapsto \|\mathbf{x}\|_\infty = \max_{1 \leq i \leq n} |x_i|$

2.2 Konvexe Mengen und Funktionen

- Eine Menge ist konvex, falls der Geradenabschnitt zwischen zwei beliebigen Punkten der Menge wieder in der Menge liegt:
  - $x_1, x_2 \in M, \, 0 \leq \theta \leq 1 \Rightarrow \theta x_1 + (1-\theta) x_2 \in M$
- Eine Funktion ist konvex, falls $\forall x_1, \, x_2 \in \mathbb{R}^n, \, \theta \in [0,1]$ gilt:
  - $f(\theta x_1 + (1-\theta) x_2) \leq \theta \cdot f(x_1) + (1-\theta) \cdot f(x_2)$
  - $f(x) \geq f(x_0) + \nabla f(x_0)^T (x - x_0), \, \forall x, x_0$
  - $f''(x) \geq 0$ (für $x \in \mathbb{R}^2$ zeige, dass die Hessematrix positiv semidefinit)
  - Rechenregeln im Skript S. 73
  - Bekante Konvexe Funktionen Skript S. 74
- Eine Funktion $f : S^n \subseteq \mathbb{R} \to \mathbb{R}$ ist quasikonvex, falls die Teilmenge S^nkonvex ist und $\forall x_1, x_2 \in S^n, \, \theta \in [0,1]$gilt:
  - $f(\theta x_1 + (1-\theta) x_2 \leq max(f(x_1), f(x_2))$
- Eine Funktion ist quasilinear, wenn sie sowohl quasikonvex, als auch quasikonkav ist.

2.3 Optimierungsprobleme

Liegt eine Maximierungsaufgabe vor, so kann diese durch das Minimieren von -f(x) für alle folgenden Fälle in eine Minimierungsaufgabe umgewandelt werden!

2.3.1 Allgemeines Konvexes Optimierungsproblem

$$\begin{array}{rcl} min & & f_0(x) \\ s.t.\, f_i(x) & \leq & 0,\, i = 1, ..., m \\ s.t.\, h_j(x) & = & 0,\, j = 1, ..., n \end{array}$$

2.3.2 Lineare Programmierung

$$\begin{array}{rcl} min & & \mathbf{c}^T\mathbf{x} + \mathbf{d} \\ s.t.\, \mathbf{G} \cdot \mathbf{x} & \leq & \mathbf{h} \\ s.t.\, \mathbf{A} \cdot \mathbf{x} & = & \mathbf{b} \end{array}$$

2.3.3 Quadratische Programmierung

$$\begin{array}{rcl} min & & \frac{1}{2}\mathbf{x}^T\mathbf{P}\mathbf{x} + \mathbf{q}^T\mathbf{x} + r,\, (P\, ist\, positiv\, semidefinit) \\ s.t.\, \mathbf{G} \cdot \mathbf{x} & \leq & \mathbf{h} \\ s.t.\, \mathbf{A} \cdot \mathbf{x} & = & \mathbf{b} \end{array}$$

2.4 Umwandlung von Problemen in andere / Modellalgemeinheit:

- Modell-Allgemeinheit: $LP < QP < QCQP < SOCP < SDP$
- Lösungseffizienz: $LP > QP > QCQP > SOCP > SDP$

2.5 Lagrange Multiplikatoren

- Lösungsaufgabe:

$$\begin{array}{rcll} max & f(\mathbf{x}) & & \mathbf{x} \in \mathbb{R}^n \\ s.t. & \mathbf{h}(\mathbf{x}) = 0 & & \mathbf{x} \in \mathbb{R}^n \end{array}$$

- Lösungsansatz:

$$\begin{array}{rcl} \Lambda(x, \lambda) & = & f(\mathbf{x}) + \lambda^{\mathbf{T}}\mathbf{h}(\mathbf{x}),\, x \in \mathbb{R}^n \\ \Rightarrow \nabla f(\mathbf{x_{opt}}) + \lambda^{\mathbf{T}}\nabla\mathbf{h}(\mathbf{x_{opt}}) & = & 0 \\ \Rightarrow \mathbf{h}(\mathbf{x_{opt}}) & = & 0 \end{array}$$

2.6 Karush-Kuhn-Tucker (KKT)

- Lösungsaufgabe:

$$\begin{array}{rcll} min & f(\mathbf{x}) & ,\mathbf{x} \in \mathbb{R}^n \\ s.t. & h(\mathbf{x}) = 0 & ,\mathbf{x} \in \mathbb{R}^m \\ s.t. & g(x) \leq 0 & ,\mathbf{x} \in \mathbb{R}^p \end{array}$$

- Lösungsbedingungen:
  - Stationarity: $\nabla f(\mathbf{x_{opt}}) + \sum_{i=1}^{m} \mu_i \nabla h_i(\mathbf{x_{opt}}) + \sum_{j=1}^{p} \lambda_j \nabla g_j(\mathbf{x_{opt}}) = 0$
  - Primal feasibility: $g_j(\mathbf{x_{opt}}) \leq 0,\, h_i(\mathbf{x_{opt}}) = 0,\, i = 1, ..., m,\, j = 1..p$
  - Dual feasibility: $\lambda_j \geq 0,\, j = 1..p$
  - Complementary Slackness: $\lambda_j \cdot g_j(\mathbf{x_{opt}}) = 0,\, j = 1..p$

2.7 Abstiegsverfahren / Gradientenverfahren

Das Gradientenverfahren ist dazu geeignet unbedingte Funktionen zu minimieren (d.h. $min\, f(x)$)

2.7.1 Voraussetzungen:

- $f(\mathbf{x})$ ist konvex und zweimal stetig diff'bar
- Es existiert ein Optimalwert $q_{opt} = inf_x\, f(\mathbf{x}) > -\infty$

2.7.2 Lösungsverfahren:

1. Wähle einen Startpunkt $\mathbf{x^{(0)}} \in D(f)$
2. Wiederhole folgende Schritte bis ein zuvor gewähltes Abbruchkriterium erreicht wurde (i.d.R. $\|\nabla f(\mathbf{x^{(k)}}\|_2 \leq \varepsilon$)
   (a) Bestimme eine Abstiegsrichtung Δx (siehe hierzu z.B. Unterpunkt Gradientenverfahren)
   (b) Wähle eine Schrittweite[1] $t > 0$
   (c) Bestimme das nächste $\mathbf{x^{(k+1)}} = \mathbf{x^{(k)}} + t^k \Delta\mathbf{x^{(k)}}$

2.8 Newton Verfahren

Das folgende Verfahren funktioniert nur für unbedingte Optimierungsprobleme. Bedingte Optimierungsprobleme lassen sich mit dem Verfahren auf S. 96 / Folie 187 lösen.

1. Wähle eibnen geeigneten Startpunkt $\mathbf{x^{(0)}} \in D(f)$
2. Wiederhole die folgenden Schritte, bis das Abbruchkriterium $\lambda^2 \leq \varepsilon$ ($\lambda(\mathbf{x}) := \sqrt{\frac{\nabla f(\mathbf{x})^T}{\mathbf{H}f(\mathbf{x})} \cdot \nabla f(\mathbf{x}))^2}$ erreicht ist:
   (a) Bestimme den Abstieg: $\Delta\mathbf{x}_{Nt}^{(k)} := -[\mathbf{H}f(x^{(k)})]^{-1} \cdot \nabla f(\mathbf{x}^{(k)}$
   (b) Bestimme das Dekrement: $\lambda^2(\mathbf{x^{(k)}}) := \nabla f(\mathbf{x}^{(k)})^T \cdot [\mathbf{H}f(x^{(k)})]^{-1} \cdot \nabla f(\mathbf{x})$
   (c) Berechne das nächste $\mathbf{x}^{(k+1)} = \mathbf{x^{(k)}} + \mathbf{\Delta} x_{Nt}^{(k)}$

2.9 Konvexe Relaxation

- S.99f. / Folie 193-196

2.10 Branch-and-Bound

- S.101f. / Folie 197-199

[1] eine Optimale Schrittweite ergibt sich z.B. aus $f(\mathbf{x^{(k+1)}}) = \min_{t>0} f(\mathbf{x^{(k)}} - t\nabla f(\mathbf{x^{(k)}})$

[2] Für genauere Informationen siehe S. 95 / Folie 185

2.11 Siumulated Annealing (grobe Beschreibung)

- S.102-104 / Folie 200-204
- Nicht konvexe Probleme haben mehrere Minima (viele Lokale und ein Globales)
- (Mit dem Algorithmus wird versucht eine Wahrscheinlichkeit zu ermitteln, ob das gefundene Minimum ein lokales oder globales ist)
- Hierzu wird an jedem Punkt die „Energie“ berechnet. Ist diese hoch, so kann die Suche in einer großen Umgebung von dem Punkt fortgesetzt werden, ist die Energie hingegen klein, so wird der Suchbereich stark eingeschränkt
- Dieses Verfahren wird solange durchgeführt bis die Energie einen statischen Zustand einnimmt

2.12 Genetische Algorithmen

- S. 105f. / Folie 205-210

3 Modellierung von Prozessen und Systemen

3.1 SystemC

3.1.1 Allgemein (S. 110-113 / Folien 215-221):

- System C wurde designed um die Lücke zwischen Software und Hardwareentwicklung zu schließen
- Basiert auf C++
- Ermöglicht die Modellierung von Hard und Software auf verschiedenen Abstraktionsebenen

3.1.2 Module (S. 113f / Folie 222 -223)

- Mögliche enthaltene Komponenten
  - Untermodule
  - Ports
  - Signale
  - Prozesse / Daten / Methoden
- Codefragmente

```
SC_MODULE(modul_name){
        SC_CTOR(modul_name){
        }
}
```

3.1.3 Ports (S. 114f. / Folien 224f):

- Schnittstelle eines Moduls zu seiner Umgebung
- Codefragmente

```
SC_Module(modul_name){
        sc_in   <int>    port_1
        sc_out  <byte>    port_2;
        sc_inout<sc_bit> port_3[];
}
```

- Lese/Schreibzugriff:

```
input = port_1.read()
input = port_1; //Alternative

port_2.write(output);
port_2 = output; //Alternative
```

3.1.4 Signale (S. 115-117 / Folien 226-229(

- Allgemein:
  - Stellen eine Art Datenleitungen zwischen Modulen und Prozessen dar
  - Werden an Ports angeschlossen
  - **Sind nur lokal innerhalb eines Moduls gültig**
  - Haben keine Richtung, **müssen aber den selben Typ haben, wie der Port, an die sie angeschlossen werden**

- Codefragmente:

```
SC_MODULE(module_name){
        sc_signal <int> signal_1;
        sc_signal <sc_bit> signal_2[];
}
```

- Lese/Schreibzugriff

```
input = signal_1.read();
signal_2.write(output);
```

- Verbinden von Ports und Signalen

```
//Anlegen eines vorher definierten Moduls
//mit einem Inputport in und Outputpot out
module_name module1;

sc_signal<bool> signal_1;
sc_signal<bool> signal_2;

module1.in(signal_1);
module1.out(signal_2);

//oder Alternativ
module1 << signal_1 << signal_2
```

3.1.5 Hierarchie/Untermodule

- Allgemein:
  - Module können Untermodule enthalten
  - Modul auf höchster Ebene heißt System
- Code-Fragmente:

```
SC_MODULE(sub_module){
  //Implementierung des Sub-Moduls
}

SC_MODULE(main_module){
  sub_module *inst;

  SC_CTOR(main_module)
    //Alternativ :inst("inst");
  {
    inst = new sub_module("inst");
    //Port-Signal-Zuweisung
  }
}
```

3.1.6 Prozesse (S. 118-121 / Folie 232 - 237):

- Allgemein:
  - Parallelausführung aktiver Prozesse (simulierte Zeit!)
  - Interprozesskommunikation über Signale
  - Nichts anderes als spezielle Methoden eines Moduls
- Prozessarten (Übersicht):
  - SC_METHOD:
    - Einmalige Ausführung nach Aktivierung, wird einmalig zu Beginn des Systems ausgeführt
    - Sequentielle, unendlich schnelle (simulierte Zeit!) Ausführung der Instruktionen
  - SC_THREAD
    - Dauerhaft aktiv, wird zu Beginn des Systems gestaret
    - Sequentielle, unendlich schnelle (simulierte Zeit!) Ausführung der Instruktionen
    - Ausführung bis zur nächsten **wait()**[3] Anweisung
    - Bei erneuter Akivierung Fortsetzung ab **wait()**-Anweisung
  - SC_CTRHEAD
    - Wie SC_THREAD, jedoch nur sensitiv auf **ein** Taktsignal
    - Nur wait() und wait(n) zugelassen
- Codefragment

```
SC_MODULE(modul_name){
  sc_in<bool> a;
  sc_in<bool> b;

  void process1(){
    //Implementierung
  }

  void process2(){
    while(1){
      //Implementierung
    }
  }

  SC_CTOR(module_name){
    SC_METHOD(process1);
        sensitive << a << b;

    SC_THREAD(process2);
        //Alternativ
    SC_CTHREAD(process, clk.pos());
  }
}
```

3.1.7 Simulation, Testbench (S. 131f. / Folien 257-259)

- Codefragment

```
#include <systemc.h>


int sc_main(int argc, char* argv[]){
  //TOP-LEVEL Signale definieren
```

[3] **Übersicht über Waitanweisungen auf Seite 120 / Folie 236**

```
    //TOP-LEVEL Modul(e)
    //Es ex. eine Implement. von module
    module *dut; //Design Under Test
    dut = new moduele("dut");

    //Verbinde Signale und Ports

    // Create the VCD trace file
    sc_trace_file *tf = sc_create_vcd_trace_file("waveform");
    // Specify which signals to trace
    sc_trace(tf, TestClk,     "clk");

    //Beginne die Simulation
    sc_start();
    ...
    sc_stop();//Kann auch woanders stehen

    // Close the trace file
    sc_close_vcd_trace_file(tf);
}
```

- Häufig nützlich ist die Verwendung einer Clock

```
sc_clock TestClk("TestClock", 10,
        SC_NS,0.5, 1, SC_NS);
```

3.1.8 Event-basierter Simulationskernel

- SystemC ermöglicht das scheinbare parallele Ausführung von Hard- und Softwareabläufen unter Verwendung des Konzeptes der Delta-Zyklen
- Hierfür wurde ein Scheduler eingeführt, der folgende Aufgaben übernimmt
  - Kontrolle über simulierte Zeit
  - Kontrolle der Reihenfolge der Prozessausführung
  - Handhabung der Event-Notifications
  - Aktualisierung von Kanälen
- Genaue Übersicht über Ablauf auf Seiten 142-144 / Folien 279-283
- Gewünschte Tabelle

Event-Nr.	Event	Simul. Zeit
1	Initialisierung	t=0ns
...	...	...

Evaluationszeilennr.	Updates	Triggered Events
...	...	...
.,..	...	...

- In der Initialisierungsphase alle Änderungen (sowohl in main, als auch in Modulen) berücksichtigen und **keine** Events auslösen.

3.2 UML-Diagramme

- Seite 153 / Folie 301/302

3.3 Petri-Netze:

3.3.1 Allgemein:

- Dienen der Modellierung von nebenläufigen, asynchronen, verteilten, parallelen, nichtdeterministischen und/oder Stochastischen Systemen
- Stellen ein graphisches und mathematisches Werkzeug dar

3.3.2 Petrinetz als 5-Tupel:

- $PN = (P, T, F, W, M_0)$
- $P = \{p_1, ..., p_n\}$:endliche Menge von Stellen (Plätzen)
- $T = \{t_1, ..., t_n\}$: endliche Menge von Transitionen
- $F \subseteq (P \times T) \cup (T \times P)$: Menge von Kanten
- $W : F \to \{0, 1, 2, ...\}$:Gewichtungsfunktion
- $M_0 : P \to \{0, 1, 2, ...\}$: Anfangsmarkierung

3.3.3 Defintionen:

- **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** Petrinetz: Petri-Netz ohne Schleifen
- **Gewöhnliches** Petrinetz, bei dem alle Kanten das Gewicht 1 haben.
- **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

3.3.4 Umwandlung in Petri-Netze unendlicher Kapazität

- **Strenge Transitionsregel**: „Eine Transition darf nur dann feuern, wenn dadurch die Kapazität der Ausgangsstellen nicht überschritten wird
- Ziel der Umwandlung ist es, die begrenzung der Kapazität aufzugeben. Gleichzeitig soll sich das Petinetz aber immer noch genauso verhalten, wie zuvor. Folgender Algorithmus ist hierbei anzuwenden:
  1. Füge für alle begrenzten Stellen p eine neue Stelle p' ein
  2. Wähle die Markierung $M_0(p') = K(p) - M_0(p)$

3. Verbinde den Ausgang einer jeden Transitionen t_1, die Token aus der Stelle p abzieht, mit der neuen Stelle p' so, dass die gleiche Anzahl Tokens, die aus p entfernt werden p' wieder hinzugefügt werden
4. Verbinde genauso den Stelle p' mit dem Eingang einer jeden Transition t_2, die Token zur Stelle p hinzufügt, sodass wieder die Anzahl der aus p' entfernten Stellen mit den zu p hinzugefügten Stellen übereinstimmt

3.3.5 Dynamische Eigenschaften von Petri-Netzen (S. 169-172 / Folien 333-339):

- **Erreichbarkeit**: Eine Markierung ist erreichbar, wenn es Transitionen gibt, die M_0 in M_n überführen
- **Beschränktheit:** Ein Petrinetz ist k-beschränkt, falls keine Markierung, die von M_0 aus erreicht werden kann, mehr als k Tokens hat. (Ein 1-beschränktes Petrinetz heißt sicher)
- **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)
- **Umkehrbarkeit:** Anfangsmarkierung 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.
- **Fairness:** Nicht behandelt

3.3.6 Überdeckbarkeitsbaum / Graph

- Eigenschaften, die sich aus dem Baum bzw. Graph erkenn lassen:
  - Beschränktheit, wenn kein ω in den Knotenbezeichnern vorkommen
  - Sicherheit, wenn nur Nullen oder Elinsen in den Knotenbezeichnern vokommen
  - Tod von Transitionen, wenn diese nicht als Kante auftreten
  - (wenn M vo M_0aus erreichbar ist, dann existiert in T ein Knoten M' für den $M \leq M'$ gilt.
- Anmerkungen:
  - Beim zeichnen des Baumes, darf man erst aufhören, wenn sich Markierungen wiederholen

3.4 Kahn Prozess Netzwerke (KPN) (S. 178-182 / Folie 352-360)

3.4.1 Allgemein:

- Dient der Modellierung von Systemen bestehend aus Prozessen

3.4.2 Aufbau und Eigenschaften:

- FIFO-Kanäle zwischen zwei Prozessen
- Prozesse werden sequentiell abgearbeitet (und zwar vollständig, nachdem sie einmal angestoßen wurden) und porduzieren und/oder konsumieren dabei eine festgelegte Anzahl von Token
- 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 (Ausführungsreihenfolge der Prozesse irrelevant)
- Es existiert ein Scheduler, der sich um das geplante Ausführen von Prozessen kümmert. (Bei begrenzten FIFOS muss der Scheduler außerdem verhindern, dass es zu Deadlocks kommen (**für Algorithmus siehe S. 182 / Folie 359f.**)
- Beispiel: Seiten 180f. / Folien 355-358

3.5 Synchrone Datenflussgraphen (SDF)

3.5.1 Allgemein:

- SDFs eigenen isch insbesondere zur Modellierung vieler Algorithmen in der digitalen Signalverarbeitung

3.5.2 Definitionen:

- Homogene SDFs: Alle Knoten konsumieren und produzieren jeweils nur ein Token an den Eingangs- und Ausgangskanten
- Inhomogene SDFs: Knoten können unterscheidlich viele Tokens konsumieren bzw. produzieren
- Für weitere Datenflussmodelle siehe S. 185 / Folie 365

3.5.3 Aufbau und Eigenschaften

- Spezialfall von KPNs
  - Knoten: Prozesse
  - Kanten: Datenpfade
- Besonderheit: Es kann **vor der Ausführung** festgelegt werden, wie viele Token produziert bzw. konsumiert werden, sodass die Anzahl der konsumierten und produzierten Token zur Ausführungszeit konstant ist. Damit lässt sich auch hier eine optimale, periodische Ausführungsreihenfolge festlegen, was verdeutlicht, dass auch SDFs deterministisch sind.

3.6 Touring-Machinen:

Die Touringmaschine stellt eine Erweiterung des endlichen Zustandsautomaten dar

3.6.1 Aufbau:

- 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)

3.6.2 Funktionsweise:

- Siehe Seite 187 / Folien 369

3.6.3 7-Tupel der Touringmaschine

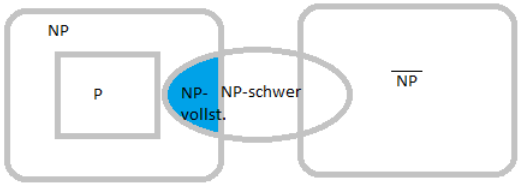
- S: endliche Menge von Zuständen
- Σ: Elngabealphabeth
- Γ: Bandalphabet, mit $\Sigma \in \Gamma$ und $\square \in \Gamma$
- $\delta : (S \backslash F \times \Gamma) \rightarrow (S \times \Gamma \times \{R/L\})$: Übergangsfunktion (R,L: Rechts- oder Linksshift des Lese/Schreibkopfes)
- $s_0 \in S$: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

3.6.4 Sinnvolle Touring-Tabelle

δ	lesen	schreiben	bewegn	n. Zustd.	Bem.

3.6.5 Weitere Definitionen:

- **Touring-berechenbar**: Es existiert eine Touring Machine, die für Eingaben aus dem Definitonsbereich anhält, sodass im Anschliuss der entsrechende Funktionswert auf dem Band steht.
- **Nicht deterministische Touring-Maschine:** Falls nicht eindeutige Übergangsrelationen von einem Zustand in den nächsten Zustand vorliegen, muss sich die Touring-Maschine verzweigen und parallel jeden dieser Wege weiterverfolgen. Kommt einer dieser Zweige zu einem Halt, so ist die Lösung akzeptiert
- **Komplexitätsklassen:** Geben an, wie schwer sich Probleme lösen lassen (genaue Darstellung auf S. 192f. / Folien 379-381)



4 Betriebsysteme (S. 194-232)

4.1 Allgemein

4.1.1 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

4.1.2 Aufgaben des Betriebssystems:

1. Speicherverwaltung
2. Programm- / Prozessverwaltung
3. Geräte- / Dateiverwaltung (hier bemüht sich das Betriebssystem um ein hohes Maß an Abstraktion, zum einfacheren Zugriff auf Geräte und Dateien)
4. Rechteverwaltung

4.2 Prozesse

Prozesse sind ablaufende Programme mit aktuellem Befehlszähler, Register und Variabeln und dienen dazu, nebenläufigkeit eines Systems zu ermöglichen (unabhängig von der Anzahl der Prozessoren)

4.2.1 Prozesszustände

- Prozess-Erzeugung:
  1. System Initialisierung
  2. Laufende Prozesse
  3. Benutzeranforderungen (user request)
  4. Initiierung durch Stapelverarbeitungs-Prozess (batch-job)
- Zustände zur Laufzeit
  1. rechnend (running)
  2. rechenbereit (ready)
  3. blockiert (blocked)
- Prozess-Terminierung
  1. Normale Terminierung (absichtlich)
  2. Fehlerhafter Ausgang (absichtlich)
  3. Fataler Fehler (unabsichtlich)
  4. Beendigung durch anderen Prozess (unabsichtlich)

4.2.2 Prozesskontext:

Der Prozesskontext beinhaltet sämtliche Informationen, die für den Ablauf und die Verwaltung von Prozessen von Bedeutung sind. Er wird häufig aufgeteilt in:

1. Benutzerkontext: Daten, Programm, Stack
2. Prozesskontrollblock (PCB) bestehend aus:
   (a) Hardware-Kontext: CPU-Register, Seitentabellen des Speichers (Paging), etc.
   (b) System-Kontext: Prozessnummer, geöffnete Dateien, Eltern-/Kindprozesse, Prioritäten, etc.

4.2.3 Prozesswechsel:

Ein Prozesswechsel geschieht durch folgende Arbeitsschritte: Speichern des aktiven Prozesskontextes, Laden des Prozesskontextes des ausgewählten Prozesses, eigentlicher Prozesswechsel = Kontextwechsel, Aktualisierung des Zustands beider Prozesse, Update der Scheduling Listen.

4.3 Threads:

Threads haben große Ähnlichkeiten mit Prozessen. Genauer gesagt wird ein Prozess, noch einmal in Teilprozesse zerlegt, die sich Betriebsmittel, wie Code und Datensegmente teilen:

4.3.1 Vorteile gegenüber Prozessen:

- Parallele Einheiten können den gleichen Adressraum verwenden
- Leichtgewichtiger als Prozesse (10-100 Mal schnellere Erzeugung und Zerstörung)
- Beschleunigen der Anwendung falls umfangreiche Ein- / Ausgabe auftritt
- Im Falle mehrer CPUs echte parallele Verarbeitung

4.4 Prozesse vs. Threads

Folgende Übersicht soll verdeutlichen, warum Threadwechsel sehr viel schlanker sind als Prozesswechsel:

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

4.5 Scheduling

Das Scheduling kümmert sich darum, wann welche Prozesse und Threads auf welcher CPU ausgeführt werden.

4.5.1 Optimierungskriterien:

Der Sinn von gutem Scheduling besteht darin, Prozesse und Threads möglichst effizeint auszuführen. Zwei Kriterien werden hierbei unterscheiden:

1. Optimierung der Performanz (Durchsatz maximieren, Latenz minimieren)
2. Optimierung der Ressoucenverwendung (Minimierung der verwendeten Prozessoren, Maximierung der Prozessorauslastung)

4.5.2 Einige Optimierungsalgorithmen:

1. **First-Come First-Served (FIFO) Algorithmus:** S.210 / Folie 415
2. **First-Come First-Served (FIFO) Algorithmus mit Prioritäten / Prioritätsbasiertes Scheduling:** S. 210 / Folie 416
3. **Round-Robin Algorithmus:** S. 211 / Folie 417
4. **Prioritätsbasiertes Scheduling mit Verdrängung (Preemtion):** S. 212 / Folie 419

Neben diesen Algorithmen gibt es noch viele weitere. Im Bereich der Real-Time-Systems können z.B. auch Zeitpunkte definiert werden, wann ein Prozess auf jeden Fall beendet sein muss.

4.6 Interprozess-Kommunikation:

Im Rahmen der Interprozess-Kommunikation wrid geklärt, wie Prozesse untereinander Informationen austauschen können, ohne dass bei gleichzeitigem Zugriff auf Ressoucen Inkonsistenzen entstehen.

4.6.1 Race Conditions:

- Race Conditions treten auf, wenn zwei oder mehr Prozesse auf gemeinsame Daten zugreifen (lesen oder schreiben), sodass das Endergebnis von der exakten Ausführungsreihenfolge abhängt.
- Dabei könne Racekonditions unabhängig davon auftreten, wie diese Daten ausgetauscht werden, also über Variabeln, gemeinsamen Speicher, Datein, etc.

4.6.2 Wechselseitiger Ausschluss

Race Conditions können durch wechselseitigen Ausschluss verhindert werden. Gängige Mechanismen sind dabei:

- Aktives Warten / Spinlocks
- **Semaphore / Mutex (binäres Semaphore):** S. 218 / Folien 431f.
- Monitor
- Message Passing
- Barriere
- Lock-Mechanismen

4.7 Memory Management

4.7.1 Relevante Speichertypen in einem PC

Von links nach rechts zunehmende Zugriffszeit und zunehmender Speicherplatz: On-Chip-Cache, Off-Chip-Cache, Hauptspeicher, Platten-Speicher (HDD, Disks, Bänder, etc.)

4.7.2 Speicherorganisation

I.d.R. ist es nicht möglich auf einzelne Bits im Speicher zuzugreifen. Der Grund hierfür besteht darin, dass mmer nur ganze Wörter $= n \cdot 1\,byte = n \cdot 8bit$ gelesen werden können. Jedem Speicherwort ist dabei eine ganze bestimme Adresse zugewiesen, über das es angesprochen werden kann.

4.7.3 Speicherverwaltung

Ziel der effizienten Speicherverwaltung besteht darin, dass von Programmen reservierter Speicher geschützt ist. Hierzu gehört insbesondere, dass der Betriebssystemkern (kernel) vor allen anderen Programmen geschützt ist. Um dies zu realisieren, weden im Wesentlichen zwei Verfahren der Speicherverwaltung angewandt:

1. Speicherverwaltung mittels fester Partitionen (Keine spezielle Hardware erforderlich) (S. 227 / Folien 449f.)
   - Funktionsweise:
     - Unterteilung des gesamten Speichers in gleich große Teilpartitionen (größe kann auch dynamisch sein).
     - Bei Speicheranforderungen werden diese Partitionen an Prozesse vergeben
   - Nachteile:
     - I.d.R. muss mehr Speicher allokiert werden, als wirklich benötigt wird.
     - Es kommt schnell zu einer Zerstückelung des Speichers, sodass die Sprünge innerhalb des Speichers immer größer werden (größerer Zeitaufwand!) und Prozesse nach kurzer Zeit nur noch wenig zusammenhängenden Speicher reservieren können
2. Memory Management Unit (MMU) mit Seiten (Paging) (Spezielle Hardware erforderlich) (S. 229 / Folien 453f.)
   - Funktionsweise
     - Unterteilung des Speichers in kleine (in Relation zur Gesamtgröße des Speichers) Teilpartitionen, sogenannte Frames
     - Allokiert ein Prozess speicher, so bekommt dieser einen zusammenhängenden virtuellen Speicherbereich zugewiesen
     - Dieser wird unterteilt in (virtuelle) Pages, die die selbe Größe haben wie die Frames
     - Jeder Page wird ein Frame zugeordnet

- Vorteile:
  - Maximale Speicherverschwendung bgerenzt durch Größe der Frames
  - Zerstückelung des Speichers (aus Sicht der Hardware) hat keine Auswirkungen auf die virtuellen Speicherbereiche der Prozesse, die (bliebig) groß sein können, aber trotzdem immer zusammenhängend sind.

4.8 Ein- und Ausgabe(geräte) (S. 230-232 / Folien 456-460)

4.8.1 Aufgaben des Betriebssystenms:

- Verwaltung der Ein- und Ausgabegeräte
- Anwendungen eine einheitliche, hardware- und herstellerunabhängige Schnittstelle zu den Geräten zur Verfügung stellen
- Abstraktion zur Vereinfachung der Verwendung von Eingabe- und Ausgabgeräten

4.8.2 Hardwaregrundlagen

- Siehe Skript S. 232 / Folien 459f.

5 Multi-Prozessorsysteme:

S. 234 - 262 / Folien 463 - 520

5.1 Einführung:

Multiprozessorsysteme beschreiben Systeme bestehend aus mehreren Prozessoren, die effizient zusammenarbeiten, um gewünschte Aufgaben schnell abzuarbeiten.

5.1.1 Komponentenübersicht:

Multiprozessorsysteme bestehen im Wesentlichen aus den folgenden Komponenten:

- Processing Elements (PE) (Ausführungseinheiten)
- Memory Architecture (Speicherarchitekturen)
- Communication Architecture (zwischen PE und PE oder PE und memory)

5.1.2 Sinn von Multiprozessorsystemen:

- Im Bereich der Performancesteigerung von Einzelprozessoren (Single Processors) ist man mittlerweile an die Grenzen der Renatabilität gestoßen:
  - Die CPU Frequenz (clock frequency) lässt sich nicht im selben Maße steigern, wie die Transistordichte (transistor density)
  - Ein wesentlicher Grund hierfür besteht in der stark erhöhten Stromaufnahme bei weiterer Steigherung der CPU Frequenz
  - Nur wenig neue Innovationen im Bereich des Instruction Level Parallelisms (mehrere Prozesse und Threads parallel auf einer CPU auszuführen)
- Daher Ansatz von Multiprozessorsystemen mit folgenden wesentlichen Verbesserungen:
  - Mehrere kleinere Prozessoren können effizient mit geringen CPU Frequenzen betrieben werden
  - Dynamisches Hin- und Herschalten zwischen den Kernen
  - Ungenutze Kerne können in einen Energiesparmodus versetzt werden un Energie zu sparen

5.1.3 Amdahl's Law (S. 240 / Folien 475f.)

Mehrere Prozessoren bringen nur dann einen Geschwindigkeitsvorteil (Speedup), wenn der Anteild es parallelisierbaren Anteils eines Programms groß ist:

- $S = \frac{T_s}{T_p}$ (S: Speedup, T_s: Ausführungszeit für serielle Implementierung, T_p: Ausführungszeit für parallele Implementierung
- $T_p = T_s(1 - p + \frac{p}{n})$ (p: parallelisierbarer Programmteil, n: Prozessoranzahl)
- $S = \frac{n}{n(1-p)+p}$ (Speedup)

5.2 Processing Elements (Ausführungseinheiten)

5.2.1 Allgemein:

- Ihre Aufgabe besteht im Ausführen von Berechnungen
- Sind untereinander verbunden und können miteinander kommunizieren

5.2.2 Arten von processing elements (S. 242-244 / Folien 479-483):

- Prozessoren (Funktionalität softwareseitig programmierbar, d.h. Software wird auf den Prozessoren ausgeführt)
  - General Purpose Processors (GPP) [z.B. Intel / AMD Standard-PC-Processoren],
  - Digital Signal Processors (DSP) [z.B. zu finden in Smartphones, HiFi-Geräten, etc.]
  - Application Specific Instruction-set Processors (ASIP) [z.B. auf Grafikkarten, Soundkarten, etc.]
- Field Programmable Devices (Programmierbare Logikschaltungen, deren Funktionalität softwareseitig festgelegt werden kann)
- Application Specific Integrated Circuit (ASIC) (festverdrahtete Logikschaltungen, bei denen ausschließlich vorher festgelegte Konfigurationsparameter eingestellt werden können)
- I/O protocol cores (dienen der Steuerung von Ein- und Ausgabegeräten)

5.2.3 Homogene vs. Heterogene Multiprozessorsysteme

- Homogene Multiprozessorsysteme bestehen nur aus Prozessoren vom gleichen Typ
- Heterogene Multiprozessorsysteme bestehen aus Prozessoren unterschiedlicher Typen

5.3 Speicherarchitekturen (S. 250 -253 / Folie 496 - 501)

5.3.1 Speichergröße/Geschwindigkeit (Grafik S. 251 / Folie 498)

5.3.2 Centralized Shared-Memory Architectures / Symmetric Shared-Memory Architecture

- Eigenschaften (**Bild**: **Skript S. 252 / Folie 499)**
  - Mehrere processor-cache subsystems teilen sich den selben physikalischen Speicher
  - Findet verwendung bei kleinen Prozessorzahlen
  - Einheitliche Zugriffszeit auf den aufgeteilten Speicher (shared Memory) von jedem Prozessor aus
  - I.d.R verfügr jeder Prozessor über einen kleinen weiteren Speicher, in dem er Daten Zwischenspeichern kann, um die Performance der Interprozesskommunikation zu erhöhen.

5.3.3 Distributed Memory Architectures

- Eigenschaften (**Bild: Skript S. 252 / Folie 500)**
  - Auch bei großen prozessorzahlen kann dieses konzept den Bandbreitenanforderungne gerecht werden
  - Der Grund hierfür besteht darin, dass das lokale (lokal traffic) Datenaufkommen vom entfernten Datenaufkommen (remote traffic) getrennt wird, sodass die Latenz für den lokalen Speicherzugriff drastisch reduziert wird
  - Erhöhte Komplexität der Interprozessorkommunikation

5.3.4 Cache Coherence Problem (S. 253 -256 / Folien 502 - 508):

- **Problem**: Wenn verschiedene Prozessoren ein bestimmtes Datum des Hauptspeichers in ihren Cache laden und ein Prozessor dieses in seinem lokalen Cache ändert, dann ist dieses Datum in allen anderen Caches veraltet.
- **Update des shared Memories:** Um die Daten konsistent zu halten ist es zum einen wichtig, dass geänderte Daten auch im Hauptspeicher geändert werden. Hierzu sind zwei verfahren üblich:
  - Write back: Das geänderte Datum wird zuerst im lokalen Cache gespeichert und dann vor dort dem shared memory bekannt gemacht
  - Write through: Das geänderte Datum wird gleichzeitig im lokalen Cache und dem shared Memory gespeichert.
- **Update der anderen lokalen Caches:** Damit alle Prozessoren mit den selben Daten arbeiten, ist es wichtig, dass auch die lokalen Caches aktualisiert werden. Auch hier gibt es im Wesentölichen zwei Methoden:
  - Snooping:
  - Directory based

5.3.5 Snooping based Cache Coherence Protocols

- Weit verbreitet in Multiprozessoren mit shared memory und **geringer** Prozessorzahl
- Statusaktualisierung lokaler Datensätze die Kommunikationsarchitektur indem alle Cache Controller am Bus / an der Kommunikationsarchitektur „schnüffeln“ (to snoop), ob ihre lokale Kopie des betroffenen Datums noch gültig ist.
- Genauere Funktionsweise siehe Seite 255 / Folie 506

5.3.6 Directory based Cache Coherence Protocols

- Geeignet auch für **hohe** Prozessorzahl
- An einem zentralisierten Ort, der sogenannten „Directory“, wird der Status der geteilten Daten festgehalten (hier muss ein extra Speicher vorhanden sein, der Aktualisierungen für jeden Speicher Block (memory block) aufnehmen kann)
- Status, die im Directory gespeichert seienn können:
  - Uncached: Der entsprechende Block ist von keinem Prozessor gechached
  - Shared: Der Block wird von mehreren Prozessoren gecached, ist aber gültig
  - Exclusive: Nur ein Prozessor hat den Block gecached und ist daher im Hauptspeicher und den Caches der anderen Prozessoren ungültig (es wird zusätzlich gespeichert, welcher Prozessor den aktuellen Wert gespeichert hat).
- Genauere Funkrionsweise siehe Seite 256 / Folie 507

5.4 Kommunikationsarchitekturen

5.4.1 Übersicht

- Punkt zu Punkt Verbindungen
- Shared Memory
- Busse (Ein Bus verbindet verschiedene Master (≥ 1) mit verschiedenen Slaves (≥ 1). Es kann immer nur ein Master gleichzeitig über den Bus mit einem Slave kommunizieren, Sind mehrere Master vorhanden wird ein Arbiter benötigt)
- Crossbar (Jeder Master kann auf einen Slave zugreifen, solange keine Konflikte beim Zugriff auftreten)
- Network on Chip (P2P-Links, Busse, Crossbars mit Bridges und Networkinterfaces beliebig zu einem Netzwerk verschaltet)

5.4.2 Shared Memory:

- Prozessoren kommunizieren über einen Speicher auf den alle prozessoren zugreifen können
- Je nach verwendetem Speicher kann nur ein Prozessor zur gleichen Zeit (single port memory) oder mehrere Prozessoren zur gleichen Zeit (multible port memories) zugreifen

5.4.3 Busse:

- Prozessoren (und andere Geräte) sind über einen Bus miteinander verbunden
- Nur Master können eine Kommunikation initiieren (bei mehreren Masters wird ein Bus Arbiter benötigt)
- Slaves antworten auf diese Anforderungen
- Vorteilhaft: es wird nur ein Satz von Leitungen benötigt
- Nachteilhaft: Nur eine Kommunikation zur gleichen Zeit.

5.4.4 Erweiterte Kommunikationsarchitekturen (S. 246-250 / Folien 488-495)

- Hier besteht die Möglichkeit mehrere Busse oder andere Verbindungen über einen Chip zu vernetzen
- Mögliche Realisierungen: Crossbars, Networks on Chip
- Benötigt werden hierfür Brücken (bridges), die die Verbindungen etablieren, sowie ein aktives Routing und Zwischenspeichern von Daten (Buffering)

5.5 Probleme der Parallelverarbeitung (S.257 - 262 / Folien 509 - 520)

5.5.1 Aufgaben:

1. Finden von Parallelismus in Programmen (manuell durch programmierer / automatisch durch Compiler) [1st partitioning issue]
2. Zusammenfassen von Aufgaben (Clustering), um scheduling und syncronisation overhead zu reduzieren [2nd partioning issue]
3. Zuweisung von Ressourcen [1st mapping issue]
4. Festlegen der Ausführungsreihenfolge (scheduling) [2nd mapping issue]
5. Synchronisierung (S. 260-262 / Folie 516-520)

6 Einführung Netzwerke

6.1 Definitionen:

- **Computer Netzwerk:** Ein Sammlung von verbundenen Computern, die in der Lage sind, untereinander Informationen auszutauschen mit dem Ziel ortsunabhängiger hoher Verfügbarkeit unter Ausnutzung ökonomischer Vorteile (Kostenersparnisse, geringerer Energieverbrauch, etc.)
- **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

6.2 OSI Referenz Modell (Übersicht)

6.2.1 Schichten

1. Physical Layer
2. Data link layer
3. Network Layer
4. Transport Layer
5. Session
6. Application

6.2.2 Grundprinzipien der Schichtung:

- Neue Layer werden erzeugt, um ein neues Abstraktionslevel zu erreichen
- Jedes Layer sollte eine genau präzisierte Aufgabe übernehmen
- Einhaltung internationaler Standards
- Schichtgrenzen sollen so definiert werden, dass das Datenaufkommen zwischen den Schichten minimiert wird
- Die Größe der Layer sollte so gewählt werden, dass sie groß genug sind, um klar abgetrennte Aufgaben übernehmen zu können, aber klein genug,dass sie nicht unübersichtlich werden.

6.3 Physical layer:

- Hat zur Aufgabe den rohen Bitstream von einer Hardwarekomponene zur anderen über einen physikalischen Kanal zu übertragen
- Definiert die Interfaces des physikalischen Mediums, die Signalübertragung, die Art der Datenübertragung (kabelgebunden, schnurlos) und die Art des hardwareseitigen Verbindungsaufbaus (**Circuit Switching, Packet Switching (S268f. / Folien 531 -534)**)

6.4 Data Link Layer

Das Data Link Layer lässt sich weiter unterteilen in den Medium Access Control (MAC) und Logical Link Control

6.4.1 Medium Access Control (MAC)

- Hauptaufgabe
  - Konflikte zu lösen, die dadurch entstehen, dass mehrere Komponenten versuchen, auf ein Übertragungsmedium zuzugreifen
  - Das Data Link Layer regelt daher die Art des Zugriffs auf den Übertragungskanal in Form von sogenannten MAC-Protokollen.
- Zwei typische MAC-Protokolle sind z.B:
  - pure/slotted **ALOHA** (Stichwort: keine Kanalprüfung, einfach Senden) (**S.272 / Folie 539**)
  - (1-persistent / p-persistent / non-persistent) **CSMA** (Stichwort: Erst Kanalprüfung, dann senden (**S. 272 / Folie 540)**
- Typische Unterscheidungsmerkmale von MAC-Protokollen finden sich auf **S. 271 / Folie 538**

6.4.2 Logical Link Control

- Hauptaufgabe: Garantieren einer fehlerfreien Datenübertragung:
  - Fehlererkennung und Fehlerkorrektur (**S. 276 / Folie 547)**
  - Flusskontrolle bei unterschiedlich schnellen Kommunikationspartnern
- Angebotene Services für das Netzwerk Layer:
  - Unaknowledged connectionless service (wird bei geringer Fehlerrate im Falle von Realtime-Datenübertragung verwendet)
  - Acknowledged connectionless service (wird häufig bei unzuverlässigen Kanälen (z.B. W-LAN eingesetzt)
  - Acknowledged connection-orientated service
- Weit verbreitetes Protokol: **PPP** (**S. 277 / Folie 550**)

6.5 Network-Layer

6.5.1 Allgemein:

- Hauptaufgabe (S.279 / 553):
  - Sicherstellung der ordnungsgemäßen Paketweiterleitung von der Quelle bis zum Ziel über alle Zwischenknoten (**ACHTUNG:** hier müssen Sender und Empfänger nicht mehr benachbart sein, sodass dies keine Aufgabe für das Data Link Layer darstellt)
  - Hierzu gehört auch das Zwischenspeichern und Weiterleiten von Paketen (Routing)
- Voraussetzungen zum Erfüllen dieser Aufgaben:
  - Subnetwork Topology (inkl. aller Router, Switche, etc.) muss den Kommunikationspartnern bekannt sein, damit die Pakete ordnungsgemäß versendet werden können
- Bereitgestellte Services für das Transport Layer (in beiden Fällen wird die eigentliche Netzwerktopologie versteckt und stattdessen ein einheitliches Adressschema verwendet, das globale Gültigkeit hat [Stichwort: IP-Adressen])
  - Connectionless (Internet)
  - Connection-Orientated (ATM)

6.5.2 Connectionless Service

- Zum Versenden von Paketen muss keine spezielle Verbindung aufgebaut werden, stattdessen beinhaltet jedes Paket Informationen über seine Absender und Zieladressen anhand derer die Weiterleitung berechnet werden kann
- Jeder Router besitzt darüber hinaus sog. Routing Tables, aus denen erkennbar ist, über welche Leitung ein Paket, das ein bestimmtes Ziel erreichen soll, gesendet werden muss

6.5.3 Connection-Orientated Service

- Da es Connection-Orientated Verbindungen im Internet so nicht gibt, wird dieser Service als Virtual Circuit realisiert
- Vor den Versenden eines Paketes muss zunächst eine Verbindung aufgebaut werden. Der gewählte Weg wird daher zu diesem Zeitpunkt bestimmt und in Routing tabellen gespeichert, wo während des gesamten Datenverkehrs gespeichert ist

6.5.4 Connectionless Service vs Connection-Orientated Service

- Siehe S. 280 / Folie 556

6.5.5 Routing Algorithmen

Beim weiterleiten von Paketen muss jeder Router wissen, welches Datenkabel er für die weiterleitung verwenden soll. Wichtige kriterien bei der Wahl dieses Datenkabels sind

- Richtigkeit
- Einfachheit
- Robustheit bei Fehlern
- Stabilität
- Fairness im Bezug auf Prioriäten und Lastverteilung
- Optimalität

Verschiedene eingesetze Verfahren sind dabei:

- **Shortest Path Routing** ((S.281 / Folie 558)
- **Flooding** (Stichwort: Einfach an alle Ausgangskanäle senden) (S.281 / Folie 558)
- **Distance Vector Routing** (Stichwort: Jeder Router hat eine Routingtabelle mit allen existierenden Knotenpunkten im Netzwerk) (S. 282 / Folie 559)
- **Link State Routing** (Stichwort: Jeder Router hat eine Routingtabelle, in der allerdings nur seine Nachbarn verzeichnet sind) (S. 282 / Folie 560)

6.5.6 Lastkontrolle

- Nicht behandelt (S. 283 / Folien 561 - 562)

6.5.7 Internetworking

- Üblicherweise sind sich in einem großen Netzwerk (z.B. Internet) verschiedene Netzwerktypen (z.B. LAN, WAN, W-LAN) mit unterschiedlichen Netzwerkprotokollen miteinander verbunden
- Probleme die dabei auftreten können sind z.B:
  - Connection-Orientated vs. Connectionless
  - Unterschiedliche maximale Paketgrößen
  - Unterschiedliche Fehler-, Fluss- und Überlastungskontrollen
- Lösungen:
  - Multiprotocol Router
  - Tunneling

6.6 Transport Layer

- Hauptaufgabe:
  - Sicherstellung einer zuverlässigen Datenübertragung von Quelle zu Ziel unabhängig von den zugrundeliegenden Netzwerktypen
- Stichwörter;:

  - TCP (S.288 / Folie 571)
  - UDP (S.287 / Folie 570)
  - Ports

- Abgrenzung zum Network-Layer:
  - Das Transport Layer läuft auf der User Hardware
  - Das Network Layer läuft i.d.R. auf Routern (hier sind Überschneidungen möglich)

6.7 Upper Layer

- Session Layer (Verbindungsaufbau (Dialogkontrolle, Token management, Synchronisation)
- Presentation Layer (Definition Abtrakter Datenstrukturen)
- Application Layer (HTTP, Email, FTP, ...)