Zusammenfassung

Zur Diplomprüfungsvorbereitung

„Parallele Algorithmen“

„Prof. Dr Amitava Datta“

Sommersemester 2002

Zusammengefasst von Markus Krebs und Andreas Horstmann

November 2002

Themenübersicht:

- Grundlagen
- PRAM-Modelle
- Netzwerkmodelle
- Speicherzugriffsarten
- Algorithmus: Präfixsummen
- Algorithmus: Präfixsummen (rekursiv)
- Algorithmus: Pointer Jumping
- Strategie: Partitioning und Merging
- Algorithmus: Berechnung des Maximums (3 verschiedene)
- Strategie: Accelerated Cascading
- Algorithmus: List Ranking (3 verschiedene)
- Algorithmus: Euler Tour Technik
- Algorithmus: Rooting a Tree (umgekehrte Euler Tour Technik)
- Die Rake Operation
- Algorithmus: Auswertung eines Ausdrucks dargestellt in einem Baum
- Algorithmus: Parallele Suche in einem sortierten Array
- Algorithmus: Verschiedene Merging-Algorithmen => Effizient sortieren
- Algorithmus: Verbundene Komponenten
    - Strategie: Pseudo-Forest
    - Algorithmus: „Verbundene Komponenten für dichte Graphen“
    - Algorithmus: „Verbundene Komponenten für dünne(lichte) Graphen“
    - Grafting
    - Algorithmus: Erkennen eines Sternen
- Algorithmus: 2D Convex Hull Problem
- Algorithmus: Halbebenenschnitt
- Algorithmus: 2 Variable Linear Programming Problem
- Die Klassen NC, P, NP
- Das Circuit Value Problem
- Beispiele für P-Vollständige Probleme

Vorlesung 1.1:

Beispiel: 8 Zahlen aufsummieren => Zeit log n, aber w(n) n log n

W(n) = Working Compelxity =T(n)x P(n)
T(n) = Ausführungszeit
P(n) = Anzahl der Prozessoren

Wann ist eine Par. Algorithmus Effizient ?

- ⇨ Wenn W1(n) in o(w2(n)) liegt
- ⇨ Wenn W1(n) = W2(n) dann der schnellere (kleineres T(n)

Problem: Wenn W1(n) > W2(n) aber T1(n)<T2(n).

Optimale Parallele Algorithmen

Work-optimal: Wenn die Workcomplexity in der Zeitkomplexitaet des zugehörigen sequentiellen Allgorithmus liegt. $W(n)=O(T_{seq}(n))$

Work-time-optimal: Wenn die Zeit des parallelen Alg. ($T_{par}(n)$) nicht verbessert werden kann.

Beispiel von oben verbessert: Statt Aufteilung von 2 Zahlen an jeden Prozessor unterteilung in n/log n. D.h. jeder Prozessor hat am Anfang log n Zahlen sequentiell zu bearbeiten in n Hierarchiestufen damit haben wir ein w(n)= O((n/log n) * log n) =O(n).

Vorlesung 1.2:

Warum Parallel-Computing ?

- ⇨ Riesige Rechengebiete, die mit normalen Seq-Rechnern nur in hoher Zeit berechnet werden könnten (Protein folding, Wetterbericht)
- ⇨ Wegen physikalische Beschränkungen im herkömmlichen VLSI Design
  - Minimisierung nur bis auf die größe eine Silizium Atoms theoretisch möglich. D.h. Grenze in ca. 15 Jahren erreicht, Mooresches Gesetz verliert dann seine Gültigkeit.
  - Lichtgeschwindigkeit: Elektronen brauchen Zeit um durch die Leiterbahnen zu laufen 3x10^8 m/s. Zeit um zwischen zwei Atomen „Daten“ auszutauschen ist 10^{-18} s bei vielen Tausen Transistoren steigt der Wert noch auf 10^{-15}s. => Grenze ist bei 1000 Teraflops erreicht. Aktuelle normale Rechner sind im Gigaflop bereich.

Vorteile von Parallel-Computing

- ⇨ Die. o.g. Limits überwinden
- ⇨ Schon heute werden viel „Paralleltechniken“ angewandt wie zB Pipelining

Probleme

- ⇨ Konventionelle von Neumann Architektur
- ⇨ Software und Hardware zu eng verflochten
- ⇨ Lösung durch Random Access Machine Modell
- ⇨ Viele verschiedene Paralelle Modell die alle anders funktionieren

- Spezifische Lösungen aber keine allgemeinen
- ⇨ Betriebssyteme sind schwer auf dieser Basis zu erstellen

Das PRAM Modell

- ⇨ N prozessoren greifen auf den gleichen Speicher zu und sollen zu jeder Zeit auf jede Zelel zugreifen können => evtl. Konflikte
- ⇨ Spezielle Lösung der Uni des Saarlandes => SB-PRAM. Lösung mittels zwischengeschalteter „Network“ Komponente
- ⇨ PRAM ist ein theoretische Modell, andere Modell zB Butterfly, hypercube und Vermaschung werden auch verwendet.
- ⇨ Aber Algorithmen für PRAM Modell ist einfach auf andere Systeme übertragbar
- ⇨ Wir reden hauptsächlich über Algorithmen für PRAM Modelle, andere System werden nur besprochen, wenn diese Lösungen (Algorithmen) komplett anders sind.

Vorlesung 2.1:

Parallelcomputer Modell

- ⇨ SIMD
  - Wichtigstes Modell zum Lösen von regulären Strukturen
- ⇨ MIMD
  - Generelleres Modell für Probleme ohne reguläre Struktur
- ⇨ SISD
  - Der normale sequentielle Rechner (KEIN Parallelrechner)
- ⇨ MISD
  - Macht keinen Sinn

SIMD:

N-Prozessoren

Charakteristik:

- ⇨ Jeder Prozessoren kann sowohl Programm als auch Daten in seinem lokalen Speicher speichern
- ⇨ Jeder Prozessor hat das gleiche Programm im Speicher
- ⇨ In jedem Zyklus führt jeder Prozessor die gleiche Instruktion aus, obwohl die Daten unterschiedlich sein können
- ⇨ Die Prozessoren Kommunizieren z.B. über Interconnection Netze oder „shared Memory“

Design Aspekte von SIMD

- ⇨ Modell ist ein Graph, die Knoten sind die Prozessoren, die Kanten die Verbindungen
- ⇨ Da jeder Prozessor nur ein Teilproblem der Gesamtaufgabe löst, ist es notwendig, dass die Prozessoren miteinander kommunizieren um das Gesamtproblem zu lösen
- ⇨ Die wichtigste Design Aspekte für Network SIMD Modelle sind:
  - Communication Diameter (Durchmesser)
  - Bisection Width

- Scalability (Skalierbarkeit)

Communication Diameter

- ⇨ Größter Abstand zwischen zwei Knoten
- ⇨ Wenn Diameter=d => Ω (d)

Bisection Width

- ⇨ Anzahl der Linkentfernungen zum Teilen des Graphen in zwei gleiche Teile
- ⇨ Je größer die Bisection Width, desto besser können die Knoten miteinander Kommunizieren, da es mehr Wege gibt

Scalability

- ⇨ Einfachheit des Erweiterns des Graphen um einen weiteren Prozessor
- ⇨ Je größer die Bisection Width, desto mehr Links müssen beim Hinzufügen eines neuen Prozessor angelegt werden
- ⇨ Extremfälle: Vollständig vermaschter Graph, linearer Graph

Zwei wichtige Netzwerkmodelle sind:

- ⇨ Mesh (Masche) (2-Dimensionaler Graph, Matrix)
  - Jeder Prozessor hat 4 Nachbarn
  - Wenn man etwas hinzufügt müssen nur die Randknoten verbunden werden => Sehr gute Skalierbarkeit
  - Bisecton Width und Diameter = O(Sqrt(n))
- ⇨ Hypercube
  - 0,1,2,3… Dimensionen
  - d-dimensional → 2^d Prozessoren
  - Direkt verbunden wenn die Bezeichnung nur um ein Bit differiert
  - Diameter = d
  - Bisection Width = 2^{d-1}
  - Skalierbarkeit: einfach zwei Hypercubes verbinden und man erhält d+1 Dimensionen
  - Verschiedene Varianten von Hypercubes: Butterfly, shuffle Exchange Network, Cube connected Cycles

Beispiel für beide: n Zahlen addieren:

- ⇨ Mesh => sqrt(n) Schritte
- ⇨ Hypercube => log(n) Schritte
- ⇨ Bei beiden ablaufen der Dimensionen

Vorlesung 2.2:

Verschiedene PRAM Modelle geordnet nach Mächtigkeit:

- ⇨ EREW (Exclusive Read, Exclusive Write)
- ⇨ CREW (Concurrent Read…)
- ⇨ CRCW
  - Common CRCW PRAM (Alle schreiben gleichen Wert auf Speicherzelle
  - Arbitrary CRCW PRAM (Willkürlicher Erfolg beim schreiben des Werts
  - Priority CRCW PRAM (Der Proz. Mit höchster Priorität gewinnt)

- ⇨ Ein Algorithmus eines schwächeren Modells läuft in gleicher Zeit und Komplexität auch auf einem mächtigeren Modell.

- ⇨ Ein Algorithmus eines mächtigeren Modells kann auf einem schwächeren mit Hilfe von meh Zeit bzw. mehr Prozessoren simuliert werden

Ein Modell A ist schwächer, wenn die Zeitkomplexität höher ist als bei einem Modell B. Sind die Zeitkomplexitäten gleich, wird nach der Workkomplexity entschieden. Je größer diese dann ist desto schwächer das Modell.

Beispiel mit n-Zahlen aufsummieren läuft auf einem EREW Modell

Beispiel Matrix Multiplikation

- ⇨ typisches Beispiel für CREW Modell (Zeitkompl.= O(log n), Work = $O(n^3)$
- ⇨ Kann umgewandelt werden für EREW Modell (kopieren der Speicherbereiche in exklusive Bereiche) Zeitkompl. = O(log n), Work = $O(n^3)$, ABER: Viel mehr Speicher als beim CREW Modell benötigt !

Vorlesung 3.1:

Berechnung der Präfixsumme

- ⇨ Gegeben eine Liste von geordneten Elementen (a_0,a_1,a_2,a_3…a_{n-1})
- ⇨ Anwenden eine binären, assoziativen Operators ⊗
- ⇨ Berechnen des Set (a_0, (a_0⊗a_1), (a_0⊗a_1⊗a_2),…,(a_0⊗a_1⊗…⊗a_{n-1}))

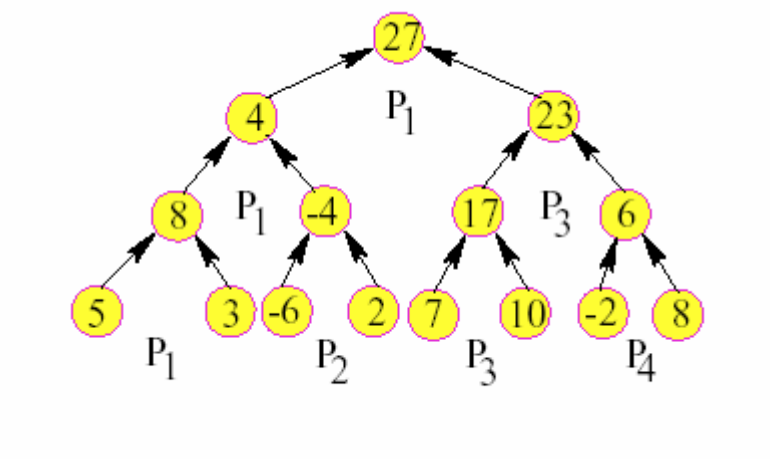
Bei + wäre das
bei folgendem Set (5,3,-6,2,7,10,-2,8) das Ergebnis (5,8,2,4,11,21,19,27)

Sequentielle Berechnung dauert O(n) Zeit

Parallele Berechnung in 2 Schritten (Bottom-Up und dannTop-Down)

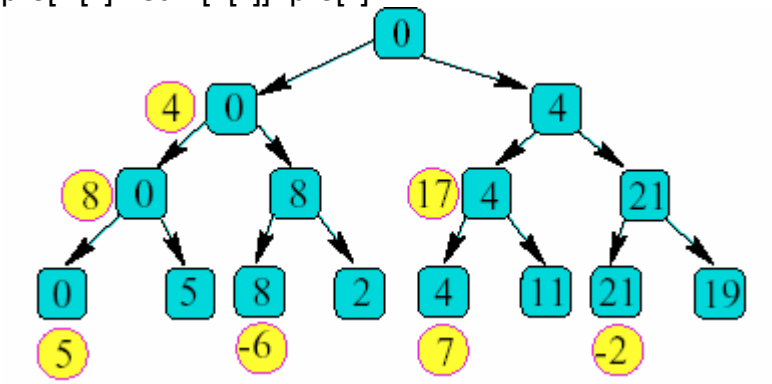
Schritt 1:

sum[v]:=sum[L[v]]+sum[R[v]], Daten warden in Array geschrieben.



Schritt 2:

pre[L[v]]:=pre[v]
pre[R[v]:=sum[L[v]]+pre[v]



gelbe Zahlen aus erstem Schritt (sum…)

Problem Ergebnis um eins nach rechts geshiftet, d.h. Summe über alles fehlt, aber diese steht in der Wurzel aus Schritt 1. => links shift und die Wurzel aus Schritt 1 anhängen.

Zeitkomplexität: O (log n)
Prozessoren: O(n)

Es ist möglich die Prozessoren auf O(n/log n) zu reduzieren.

Korrektheitsbeweis:
Nach Preorder prinzip durchlaufen
Lemma: Nach dem zweiten Schritt hat jeder Knoten die Summe seiner Blättervorgänger

Induktion von der Wurzel ausgehend

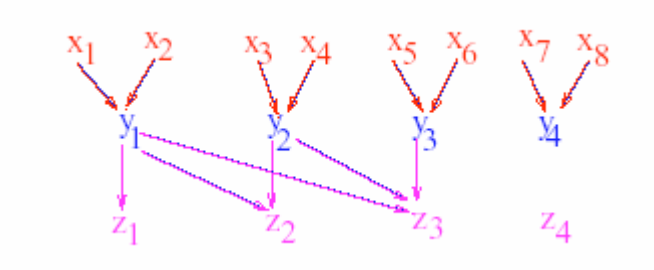
- ⇨ Indukionshypothese: Wenn ein Knoten die korrekte summe enthält so enthalten auch seine Kinder die korrekte Summe.
- ⇨ Basisfall: In der Wurzel zutreffend, da die Wurzel keine übergeordneten Knoten hat
- ⇨ Weiter: Betrachtung nach linkem und rechtem Kind

Sowohl Schritt 1 als auch Schritt zwei sind auf dem EREW Modell möglich

Vorlesung 3.2:

Rekursive Variante des Präfixsummen Algorithmus

Rekursive Berechnung der Präfix Summen in jeder Hierarchiebene, also immer zwei Knoten addieren in jeder Ebene



Unterscheidung zwischen geraden und ungerade Indices

- Bei den geraden Indices müssen nur die Vorgänger aufaddiert werden.
- Bei den ungeraden Indices muss jeweil auf das ungerade x_i zugegriffen werden, welche nicht in den Vorgängerwerten enthalten ist.

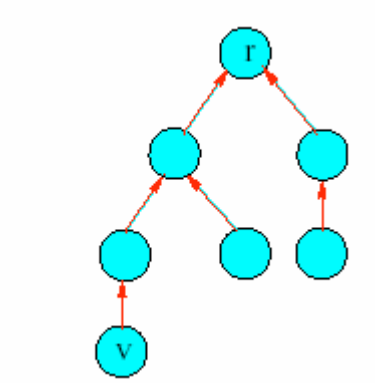
Hier folgt nun eigentlich der Korrektheitsbeweis für den Algorithmus

Zeitkomplexität: O(log n) (In jedem Schritt wird das Problem halbier!)
Work.kompl.: O(n) (wegen O(n/log n) Prozessoren)

Pointer Jumping Algorithmus

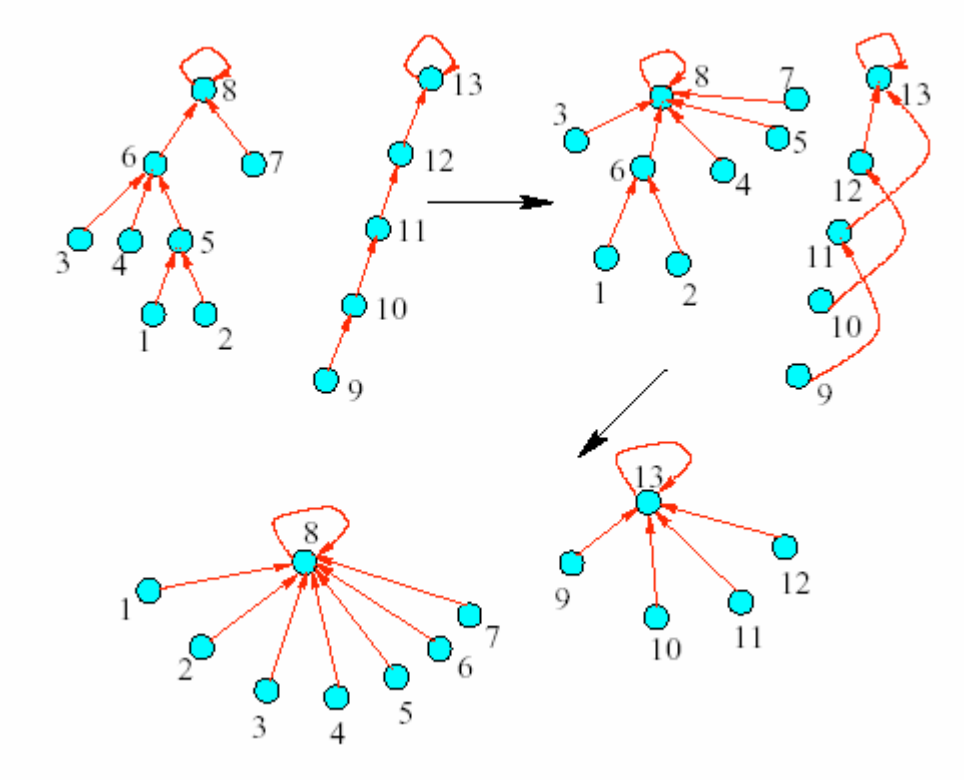
„Rooted directed Tree“
Def.:

- Alle Knoten ausser dem Wurzelknoten haben OutDeg =1 , der Wurzelknoten hat OutDeg=0
- Es existiert ein gerichteter Weg von jedem Knoten zum Wurzelknoten nicht andersrum.



Problemstellung: Zu einem Knoten in einem Wald von Bäumen den zugehörigen Wurzelknoten finden.

Algorithmus der aus jedem Baum ein Stern erzeugt, bei dem jeder Knoten direkt an seiner Wurzel hängt:



Prefix on rooted directed Trees

Jetzt geben wir jedem Knoten zusätzlich einer Wert, also z.B. ein Gewicht

Berechnung ist gleich wie eben, nur dass zusätzlich das Gewicht bis hin zur Wurzel in jedem Schritt noch aufaddiert wird.

Komplexität

Zeitkomplexität: O(log h) (h ist die maximale Tiefe des Baumes, in jedem Iterationsschritt verringert sich die Tiefe um 1/2)
Prozessoren: O(n)

CREW wird benötigt !

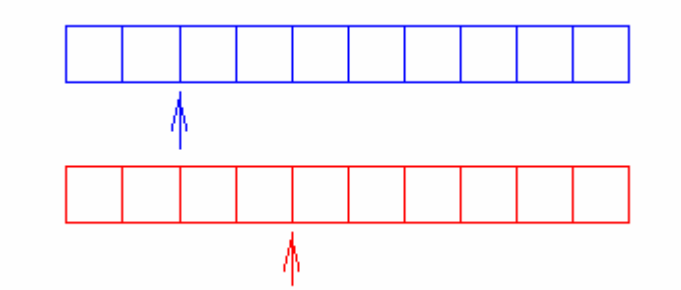
Vorlesung 4.1:

Partitioning und Merging Strategie

Partitioning: Einteilung in unabhängige Teilprobleme. Parallel lösen. => ähnlich wie Divide and Conquer bei sequentiellen Algorithmen, nur dass die Teilprobme beim Partitioning parallel gelöst werden, bei Divide an Conquer rekursiv.

Merging: Gegeben sind zwei geordnete Listen. Aufgabe: verschmelzen dieser beiden Listen, damit eine gemeinsame sortierte Liste entsteht.

Der sequentielle Algorithmus funktioniert so:



Zwei Pointer durchlaufen die Listen, die Elemente werden paarweise verglichen und der betreffende Pointer inkrementiert. Die Ergebnisliste wir in ein neues Array geschrieben. Wenn List 1 länge n hat und Liste 2 länge m, dann hat die Ergebnisliste entsprechende Länge n+m => Die Komplexität ist O(n+m).

⇨ Parallele Lösung durch Partioning Strategie

Bestimmen des Rangs einer Elementes einer Liste:

Rank(a_i:A) Anzahl der Elemente in A die kleiner oder gleich a_i sind
Rank(b_i:A) Anzahl der Elemente in A die kleiner oder gleich b_i sind

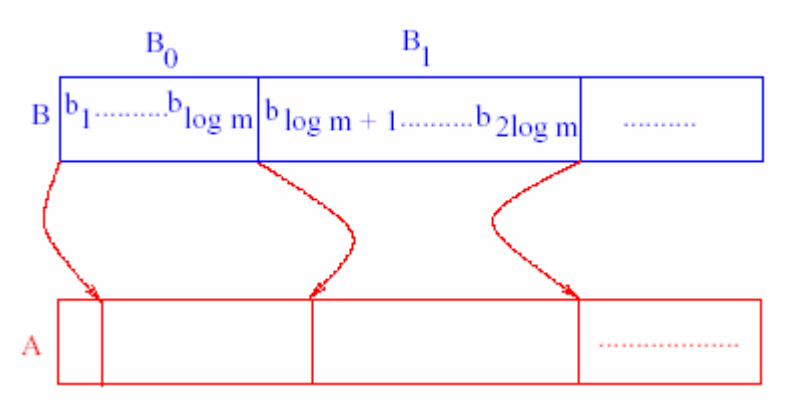
Die Position eines Elementes in der Ergebnisliste ist die Summer der beiden Teilränge, also rank(a_i:C)=rank(a_i:A)+ rank(a_i:B)

Paralleles Merging teilt nun das Gesamtproblem in viele kleine Teilproblem auf, die parallel verarbeitet werden können. Wenn die Problemgröße klein genug ist, wird das Problem sequentiell gelöst.

Schritt 1:
Die Liste B wird also in log m große Teile geteilt, es gibt also m/log m Teile. Das letzte Element eines Teils steht immer an Position i* log m, also ein vielfaches von log m.

Schritt 2:
Weise jedem Endelement der Liste B einen Prozessor zu. Dieser dursucht Liste A mittels Binärer Suche und teilt diese Liste gemäß dem Endelement von Liste B.

Das Suchen geht in O(log n) Zeit und die Liste wird in m/log m Teile geteilt. Zwei korrespondierende Blocks zwischen Liste B und A nennt man nun „matching Blocks“.



Wenn die Blöcke in Liste A größer als log m sind, dann werden diese in log m Blöcke unterteilt. Nun geschieht das gleiche mit der Teilliste von B wie eben mit Liste A, Sie wird mit binärer Suche durchsucht und „Matching Blocks“ zur Teilliste von A werden gesucht. Jeder Block in Liste A ist jetzt auf jeden Fall in O(log m) Größe.
Dies dauert O(log log n) Zeit .

Schritt 3:
Nun folgt der bereits erklärte sequentielle Merging Algorithmus auf die Matching Blocks Jeder zugewiesene Prozessor bearbeitet ein Paar von Matching Blocks in O(log m) Zeit.

Komplexität:

Schritt 1:
Prozessoren: m/log m
Zeit: O(1)

Schritt 2:
Prozessoren: m/log m (Bleibt natürlich gleich)
Zeit: O(log n)
Workin.komp.: O(m+n) (O((m/logm)*logn) ist in O(m+n)

Schritt 3:
Prozessoren: m/log m
Zeit: O(log m)
Working: koml: O(m)

Der Algorithmus benötigt das CREW Modell, da die Prozessoren beim Teilen der Liste gleiche Zellen lesen.

Theorem: Wenn 2 sortierte Listen der Größe n sind, dann können diese in O(log n) Zeit und in O(n) Operationen gemerged werden.

Vorlesung 4.2:

1. Algorithmus zur Berechnung des Maximums

Berechnung des Maximums in O(1)

- ⇨ Ein Array mit den p ungeordneten distincten Elementen
- ⇨ Ein zweites Boolean Array initialisiert mit „1“

Schritt 1:

- ⇨ Weise jedem Element der Liste p Prozessoren zu => p^2 Prozessoren werden benötigt.
- ⇨ Jeder Prozessor für genau einen Vergleich durch: If A(i) > A(j) then M(j)=0
- ⇨ Übrig bleibt dann genau eine „1“ im Boolean Array, diese markiert das größte Element

Schritt 2:

Suchen der „1“ im Boolean Array mit p Prozessoren in Konstanter Zeit, da jeder genau ein Element betrachtet. Der Prozessor der die „1“ findet gibt den Index k der Zahl zurück.

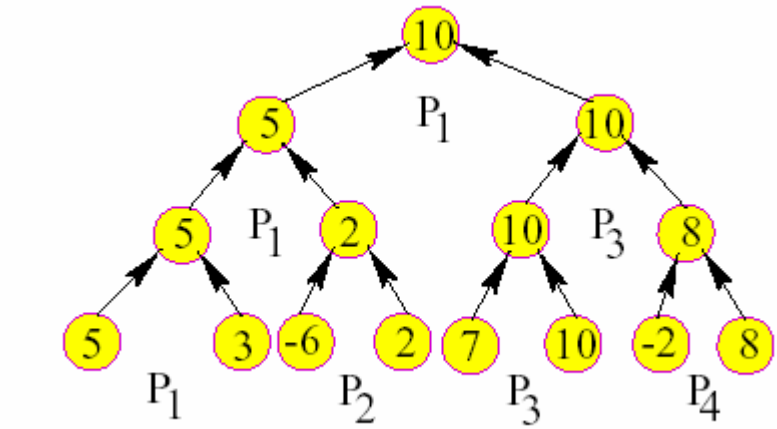
Komplexität:
Prozessoren: p^2
Zeit: O(1)

Der Algorithmus benötig das Common CRCW Modell, da alle Prozessoren, wenn sie auf die gleiche Speicherzelle schreiben, das gleiche schreiben, nämlich die „0“.

Der Algorithmus ist also optimal bezüglich Zeit, aber sehr komplex im Bezug auf Work.

2. Algorithmus zur Berechnung des Maximums

(Work-)Optimale Berechnung des Maximums

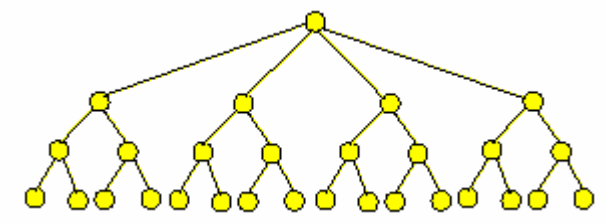


Komplexität:

Prozessoren: O(n) reduzierbar auf O(n/log n)
Zeit: O(log n)
Work.kompl.: O(n)

3. Algorithmus zur Berechnung des Maximums

Berechnung des Maximums in O(log log n) Zeit



Kein Binärbaum sonder komplexer. Die Wurzel des Baumes hat $2^{2^{k-1}} = \sqrt{n}$

Jeder Knoten im i-ten Level hat nun $2^{2^{k-i-1}}$ Kinder, wobei die Anzahl der Level. Level k hat immer 2 Kinder ! → k=O(log log n)

An den einzelnen Knoten sind nun allerdings keine paarweise vergleiche mehr möglich, da es mehr als 2 Kinder sind, deshalb nimmt man den O(1) Algorithmus von oben.

Komplexität:
Zeit: O(log log n)
Working.kompl.: O(n log log n)
Prozessoren: O(n)

Accelerated Cascading

Trick: Kaskadieren von zwei Algorithmen.

Beispiel mit Berechnung des Maximums:

Der zweite Algorithmus war workoptimal aber langsam
Der dritte Algorithmus war suboptimal aber sehr schnell

Wir kombinieren diese zwei Algorithmen mit der Accelerated Cascading Strategie. Wir beginnen mit dem optimalen Algorithmus und reduzieren das Problem bis auf eine gewisse Ebene, dann nehmen wir den suboptimalen aber sehr schnellen Algorithmus.

Phase 1:
Bis Level (log log log n) wird der workoptimale Algorithmus verwendet

Komplexität:
Work: O(n)
Zeit: O(log log log n)

Die Anzahl der zu berechnenden Elemente reduziert sich auf

$$\frac{n}{2^{\log\log\log n}} = \frac{n}{\log\log n}$$

Phase 2
Nun kommt der schnelle aber suboptimale Algorithmus zum Zug. Dieser bearbeitet die übrigen $n' = O(\frac{n}{\log\log n})$ Elemente

Komplexität:
Work.: O(n)
Zeit: O(log log n)

Theorem:
Das Maximum einer Liste von n Elementen kann in O(log log n) Zeit und O(n) work auf einer Common CRCW PRAM berechnet werden

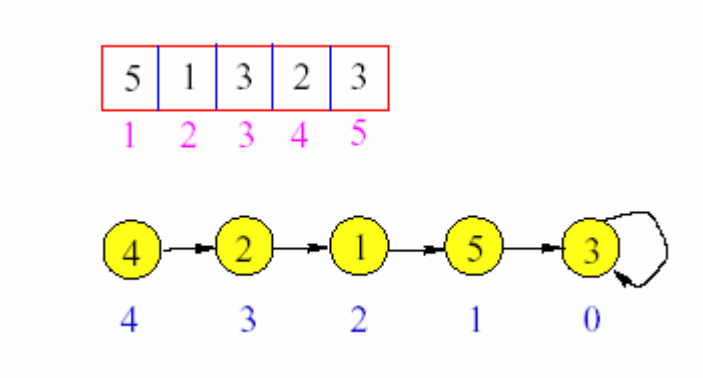
Vorlesung 5.1:

3 verschiedene List Ranking Algorithmen

1. O(log n) Zeit und O(n log n) Work
2. O(log n log log n) Zeit und O(n) Work
3. O(log n) Zeit und O(n) Work

List Ranking:

Lineare Liste, repräsentiert als Successor-Array



Input: Lineare Liste
Output: Distanzen der einzelnen Elemente bis zum Ende der Liste (Ranking)

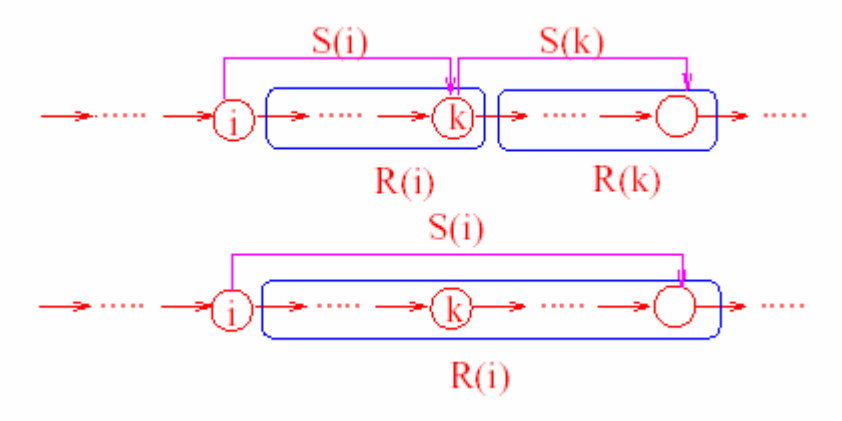
z.B. Der Nachfolger von 1 (index) ist 5 (inhalt an Position 1) und von 2 ist 1….Der Nachfolger von 3 ist 3 und ist damit das Ende der Liste.

Sequentiell ist das Problem in O(n) Zeit lösbar → Ein work-optimaler paralleler Algorithmus sollte das Problem in O(n) Work lösen können.

Erster Algorithmus:

O(n) Zeit und O(n log n) Work

- Der Algorithmus basiert auf dem Präfixsummenalgorithmus.
- Am Anfang werden alle Elemente mit „1“ initialisiert, ausser dem Endknoten, dieser wird mit „0“ initialisiert.
- Jetzt werden für jedes Element die Werte aufsummiert (Präfixsummen)
- Die Werte der einzelnen Elemente sind nun die Distanzen zum Endelement



Korrektheit:

- Untersuchung vor der Iteration.
- Untersuchung nach der Iteration
- Untersuchen ob der Algorithmus terminiert

Komplexität:

Iterationen:	O(log n)
Work,Kompl.:	O(n log n)
Zeitkompl.:	O(log n)
Prozessoren:	O(n)

Modell: CREW-PRAM Modell

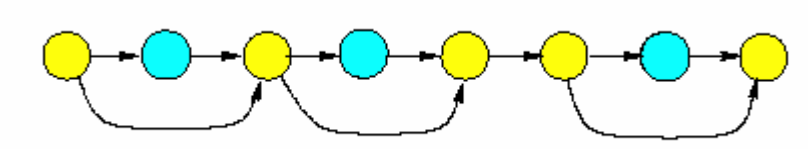
Zweiter Algorithmus

Verbesserung des ersten Algorithmus auf O(n) Work bei O(log n log log n) Zeit.

Strategie:

1. Aufteilen der List in Listen mit O(n/log n) Knoten
2. Pointer Jumping anwenden um den Wurzelknoten der nun erzeugten Teillisten zu finden
3. Die ursprüngliche Liste wiederherstellen und die weggelassenen Knoten „ranken“.

Schritt 1 ist sehr wichtig:



Ein Set i von Knoten ist unabhängig, wenn $i \in I, S(i) \notin I$.
(d.h. der Nachfolger jedes Knotens ist nicht im Set)

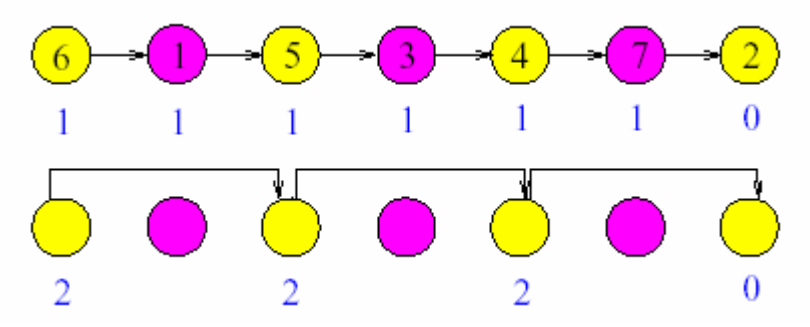
1. Phase:

2-Färbung der Liste:
Es existiert ein einfacher sequentieller-Algorithmus der das Problem in O(n) Zeit löst. Beim parallelen Algorithmus ist das komplizierter, aber wir gehen davon aus, dass:

Theorem: „Eine verkettetet Liste mit n Knoten kann in O(log n) Zeit und O(n) work zweigefärbt werden“

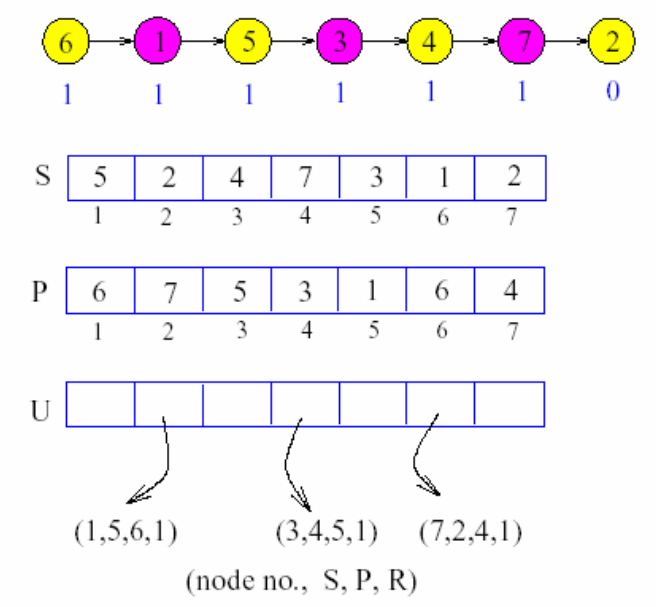
Im ersten Durchgang halbiert sich die Liste, wir wollen aber eine Liste mit n/log n Knoten, d.h. wir müssen den 2-färbe Algorithmus (log log n) mal ausführen.

Speichern der Informationen über die aus einem Set entfernten Knoten:
Entfernt wurden diese mit O(log log n) Iterationen, deshalb muss das Hinzufügen genauso schnell gehen.



oben: initiale Liste
untern: Knoten wurden entfernt und die Rankings der übrig gebliebenen Elementen um den Rank des entfernten Knotens erhöht.

Nun werden zu dem Ursprünglichen Array S (Succ. Array), ein Predesscor Array P und ein Array U, in welchem wir die entfernten Knoten, also Knotenindex, Vorgänger, Nachfolger und Rank speichern, erzeugt.



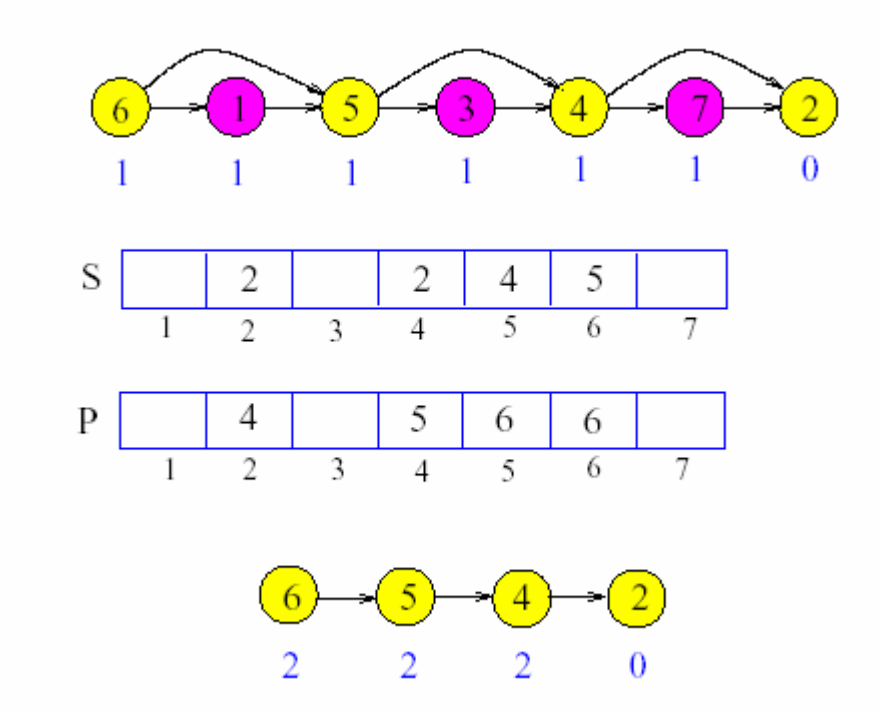
for all $i \in I$ pardo

Set $U(i) := (i, S(i), P(i))$

Set $R(P(i)) := R(P(i)) + R(i)$

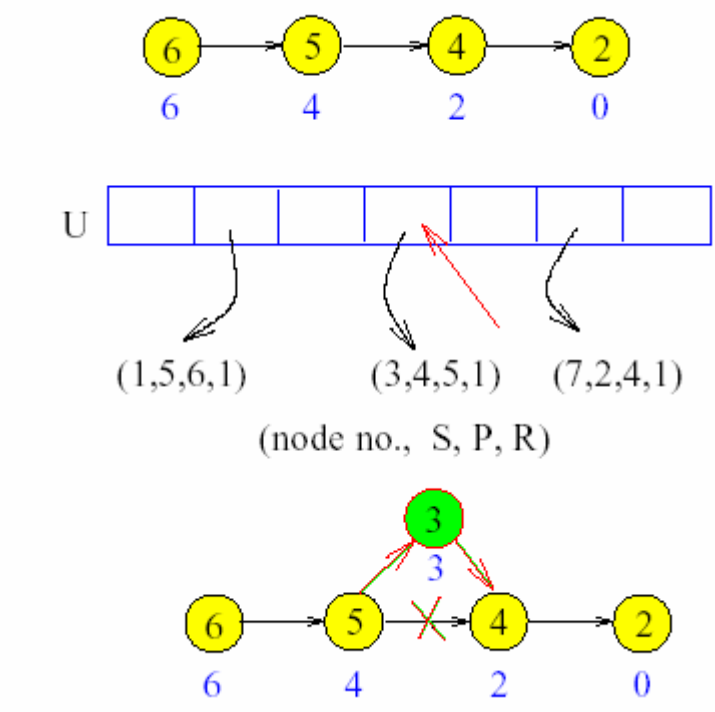
Set $S(P(i)) := S(i)$

Set $P(S(i)) := P(i)$



- Die Vorgänger und Nachfolgerpointer werden nun aktualisiert
- Die Ranks der Knoten direkt vor einem entfernten Knoten werden akualisiert
- Es gibt leere Arrayzellen
- P und S können dann aber in O(n) work un d O(log n) Zeit mit Hilfe der Präfixsummen komprimiert werden, allerdings nach JEDEM Entfernungsschritt
- Wir benötigen ein separates U Array für jeden Iterationsschritt

2. Phase



- Auf das auf n/log n reduzierte Teilarray wird nun in Phase 2 der suboptimale Algorithmus angewandt (Alg. 1)

- Ein entfernter Knoten kann nun mit Hilfe des U Arrays in O(1) wiedereingefügt werden.

Komplexität:

Phase 1:
Die 2-Färbung und die Präfixsummerberechnung:
Zeit: O(log n)
Work: O(n)
⇨ Gesamtzeit für Phase 1 = O(log n log log n)

Phase 2: In jeder Iteration halbiert sich Anzahl der Elemente in den Arrays. Jede Iteration benötigt:
Work O(n) wobei n aber nun die Größe des jeweiligen Arrays ist
⇨ Also Gesamt Work O(n), weil n+n/2+n/4+n/8+…. und das ganze log log n mal

Zeit Gesamt: O(log n log log n)
Work Gesamt O(n)

Vorlesung 5.2:

Dritter Algorithmus:

Optimaler, randomisierter List Ranking Algorithmus
Zeit O(log n), Work O(n), Prozessoren: O(n/ log n), EREW Modell

Las Vegas Algorithmus (Liefert immer das richtige Ergebnis, braucht aber in seltenen Fällen sehr lange, bzw. unendlich)

Monte Carlo Algorithmus (sehr schnell, aber das Ergebnis stimmt nur zu einer angegebenen Wahrscheinlichkeit. Durch merhfaches Ausführen kann man die Wahrscheinlichkeit erhöhren

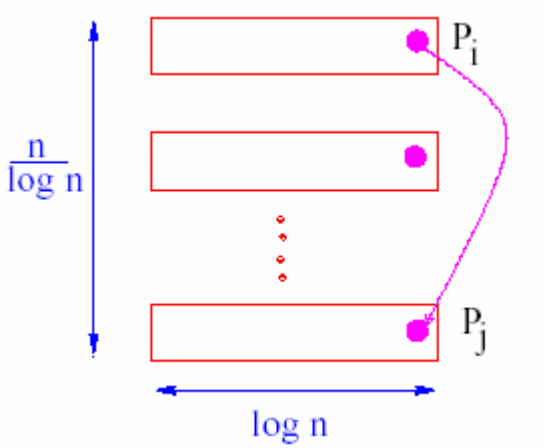
Unser jetzt besprochener Algorithmus ist ein Las Vegas Algorithmus

Die Laufzeit von O(log n) wird mit sehr hoher Wahrscheinlichkeit erreicht, ist aber nicht garantiert.

Der randomisierte Algorithmus ist im Prinzip identisch mit dem zweiten Algorithmus (der mit der Zweifärbung).

Folgendes ist anders am dritten Algorithmus:

Aufteilung der Listen zunächst in n/log n Listen mit log n Größe
Dann werden jeweils die Endelemente in eine neuen Liste geschrieben, aber:
⇨ Es darf nicht passieren, dass zwei in der Ursprungsliste aufeinanderfolgende Elemente zusammen kommen. Lösung bei einem Konflikt:
⇨ Die zwei konfligierenden Prozessoren erzeugen zufällig eine „1“ oder eine „0“. Der Prozessor, der die „1“ hat darf seine Zahl übergeben (die spielen Schere, Stein, Papier)



- ⇨ Es entstehen nun log n Listen mit jeweils n/log n Elementen !!! (gerade umgekehrt wie eben)
- ⇨ Nun werden die Listen wieder wie in Algorithmus zwei mit dem Suboptimalen Ranking Algorithmus zusammengefügt.

Komplexität:

- ⇨ Die erwartete Zeit bis eine log n große Liste leer ist, ist (4 log n), da mit
- ⇨ Wahrscheinlichkeit von ¼ ein Element entfernt wird. („1“ und „0“ ergeben 4 Möglichkeiten!!!)
- ⇨ Wie große ist nun die Wahrscheinlichkeit, das es viel länger geht ?
  - Die Wahrscheinlichkeit, dass es nach 16 log n Schritten noch nicht leer ist, ist nach Chernoffs Ungleicheit kleiner als 1/n
  - Also ist die Wahrscheinlichkeit, dass nach 16 log n Schritten überhaupt noch eine nicht-leere Liste vorhanden ist, kleiner als
    $$\frac{n}{\log n} \times \frac{1}{n} = \frac{1}{\log n}$$
  - Alle Knoten sind dann normalerweise nach O(log n) Schritten entfernt
- ⇨ Das Wiederzusammenfügen und die Berechnung der Ranks dauert O(log n)
- ⇨ Also terminiert der Algorithmus mit sehr hoher Wahrscheinlichkeit nach
  - Zeit: O(log n)
  - Work O(n)

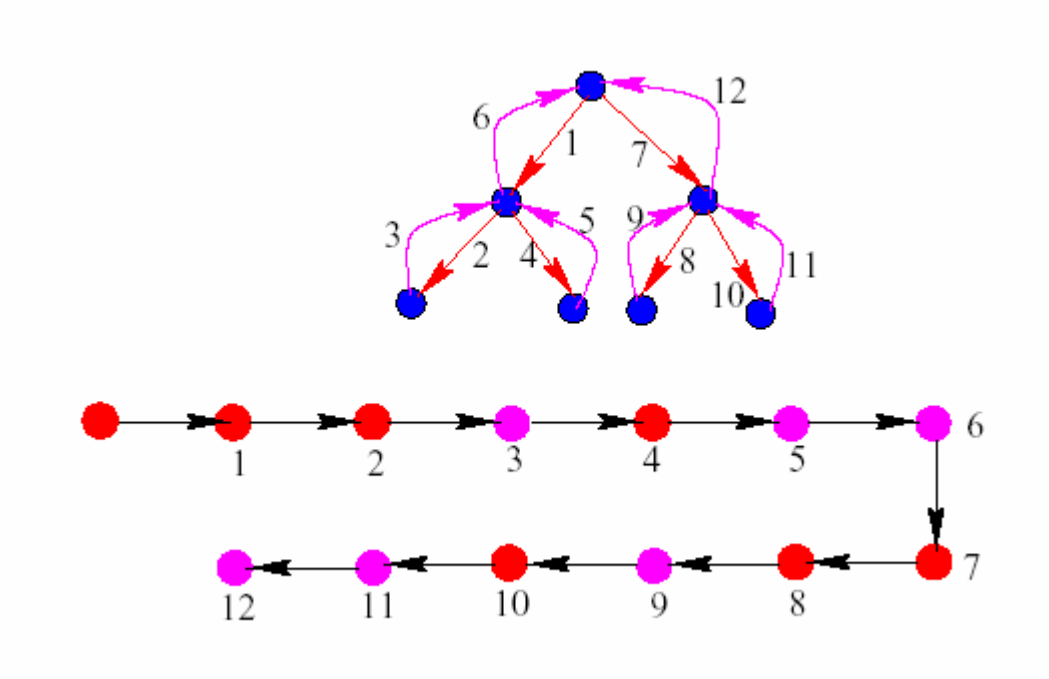
Vorlesung 6.1:

„Die Euler Tour Technik“

Parallele Tiefensuche in einem Baum ist schwierig. Man kann den Endknoten erst dann bestimmen, wenn man bereits die Vorgängerknoten besucht hat (sequentiell).

Lösung: Umwandlung des Baumes in eine Liste

- ⇨ Gute Algorithmen für Listen vorhanden !!!
- ⇨ Die Euler Tour erstellt eine Liste aus einem Baum, indem zusätzliche Kanten eingefügt werden (magentafarben !).



- ⇨ Die Liste wird dann in Tiefensuche erstellt, man erhält einen Zyklus, einen so genannten Eulerkreis
- ⇨ Hat der Baum n Knoten, dann erhalten wir eine Liste mit 2n-2 Knoten, wobei jede Kante des Baumes nun ein Knoten der Liste ist

Definitionen:

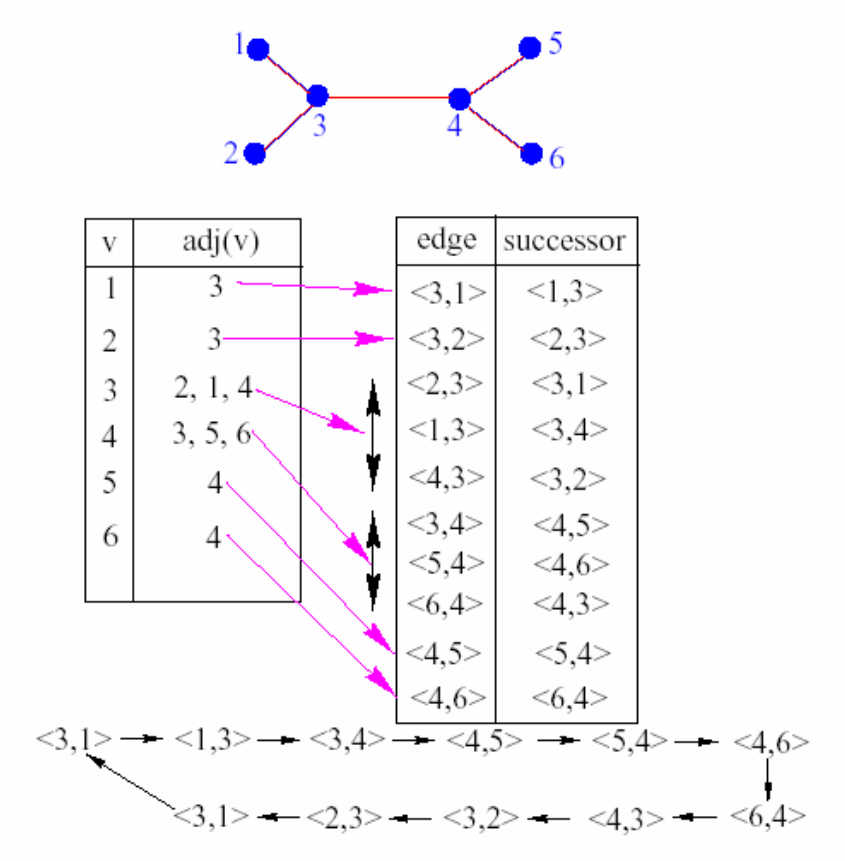
- ⇨ Für jeden Knoten v aus V ist p(v) der Vorgänger von v
- ⇨ Jeder Rote Knoten der Liste repräsentiert eine Kante der Art <p(v),v>
- ⇨ Zählt man die roten Knoten, so erhält man direkt die PreOder Nummerierung (den index erhält man indem man die roten Knoten mit „1“ bewertet und die magentafarbenene mit „0“ und darauf die Präfixsummen berechnet)
- ⇨ Unser Ursprünglicher Baum heisst T=(V,E) und unser neuer Eulergraph heisst jetzt T'=(V,E'). Jede Kante (u,v) aus E wird durch zwei Kanten (u,v) und (v,u) ersetzt
- ⇨ Indegree und Outdegree jedes Knotens ist nun gleich. Bei einem Blatt sind diese jeweils 1
- ⇨ T' ist nun unser Eulergraph
- ⇨ Ein Eulerkreis ist eine Tour den Baum entlang, bei dem jede Kante des Graphen genau einmal besucht wird und man am Ende wieder am Ausgangspunkt angelangt ist. Jeder Knoten wurde dann passiert.
- ⇨ Ein Eulerkreis ist dann möglich, wenn jeder Knoten gleichen Indegree wie Outdegree hat

Konstruktion des Eulerkreises

- ⇨ Jede Kante hat genau eine Nachfolgerkante
- ⇨ Speichere zu jedem Knoten seine adjazenten Knoten ab
- ⇨ $adj(v) =< u_0, u_1, \ldots, u_{d-1} >$ (u0,u1…) sind die Nachbarknoten von v und d ist die Anzahl derer
- ⇨ Der Nachfolger der Kante <u_i,v> ist

$$s(< u_i, v >) =< v, u_{(i+1) \bmod d} >, 0 \leq i \leq (d-1)$$

- ⇨ mod d erzeugt hierbei einen Kreis, also den letzten Schritt der Euler Tour

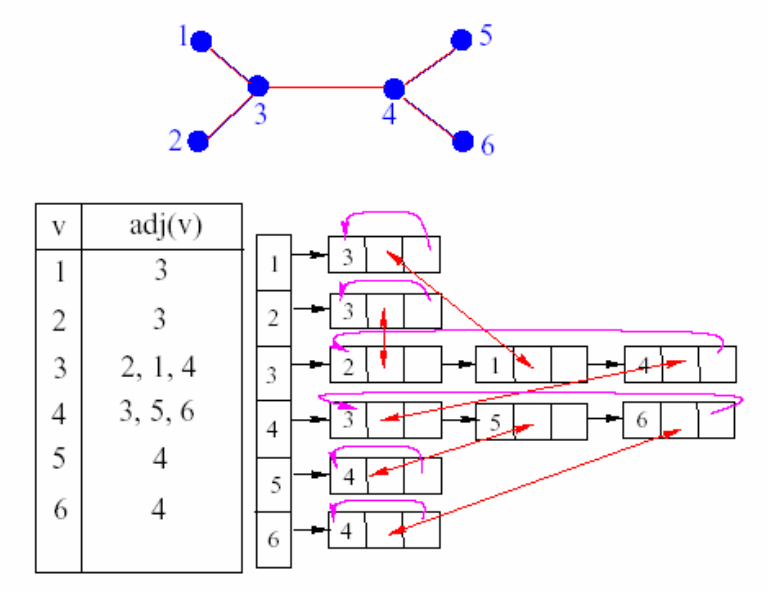


Korrektheit:

Lemma: Die Succesor Funktion s definiert nur einen Zyklus, kein Set von Zyklen

Diese Lemma wird nun mit Induktion bewiesen

Parallele Konstruktion der Euler Tour



⇨ Wir nehmen an, dass der Baum als eine Set von Adjazenzlisten gegeben ist

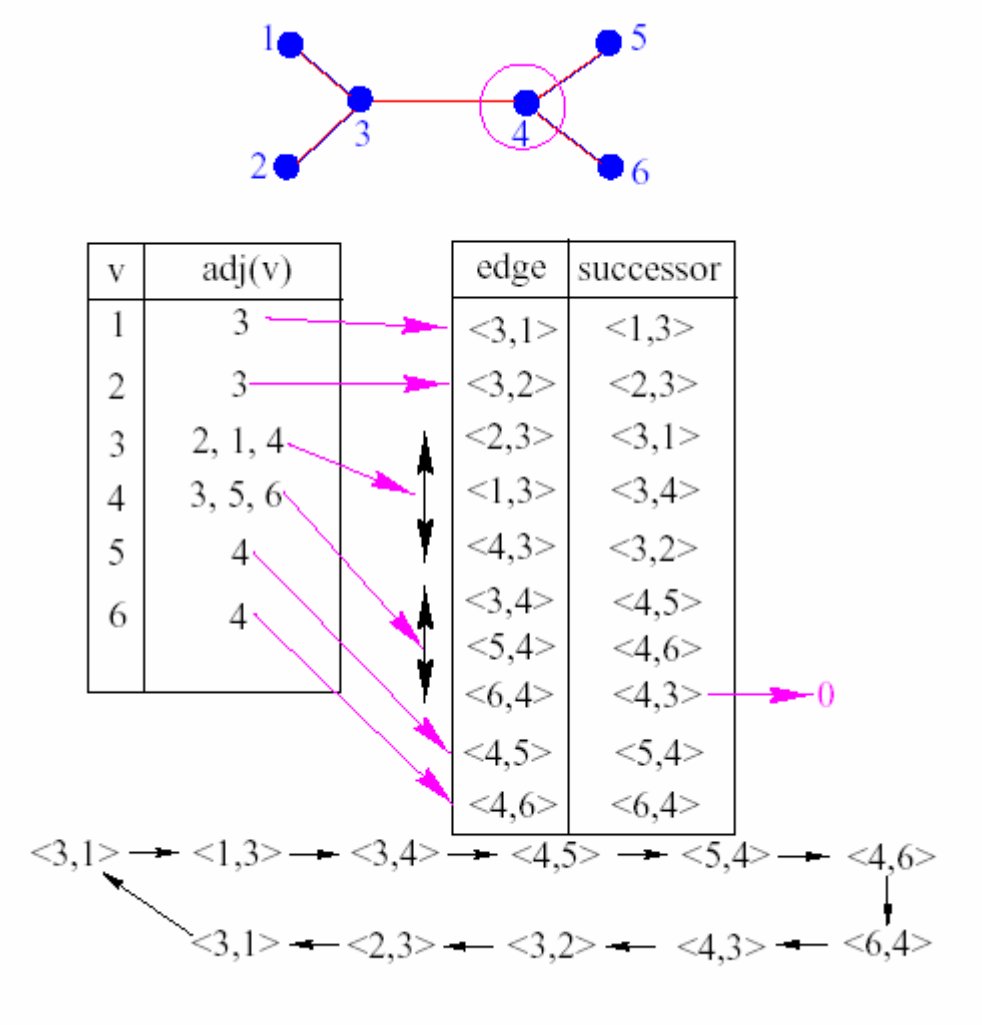
- ⇨ In dieser Liste müssen die Pointer auf die Nachfolgerknoten sein
- ⇨ Die Euler Tour kann man mit O(n) Prozessoren dann in konstanter Zeit O(1) berechnen
- ⇨ Jedem Knoten der Adjazenzliste wird ein Prozessor zugewiesen
- ⇨ Es wird keine Concurrend read oder Write benötigt
- ⇨ EREW Modell ist ausreichen

Vorlesung 6.2:

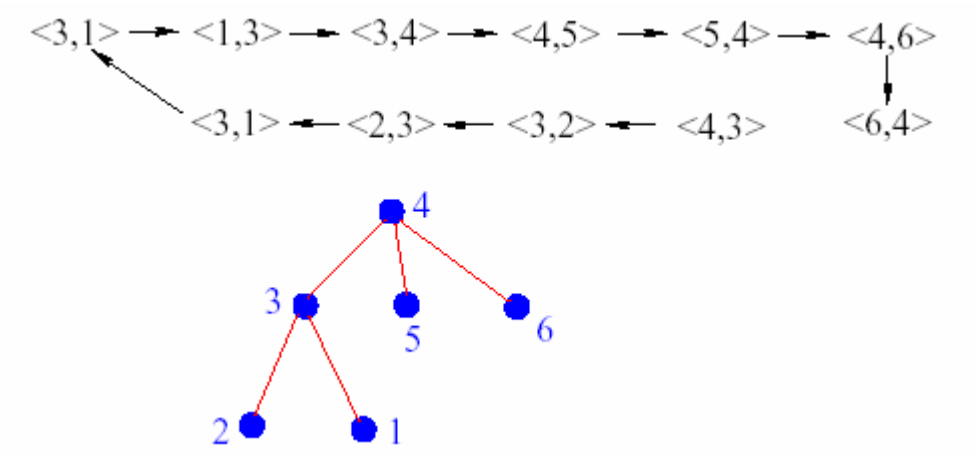
„Rooting a Tree“

Rücktransformation Euler-Tour → Baum, durch Angabe der gewünschten Wurzel

- ⇨ Aufsplitten des Euler Zyklus an der gewünschten „Wurzelstelle“



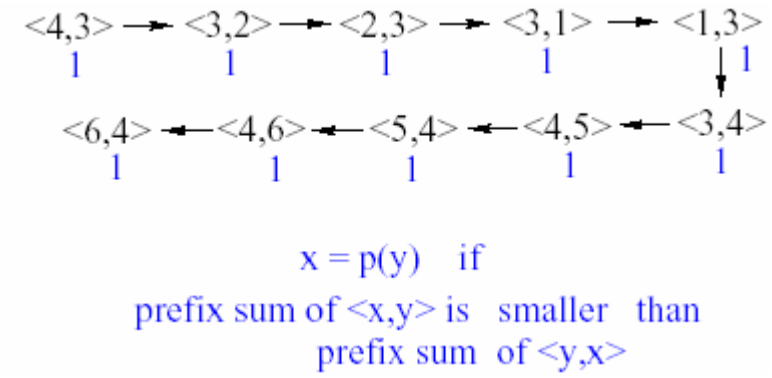
- ⇨ z.B. Wurzel=4, dann splitten der Kante zwischen <6,4> und dem Nachfolger <4,3>
- ⇨ Es entsteht nun wieder ein Baum



Algorithmus:

Eingabe: Die Euler Tour eines Baumes und ein spezieller Knoten r (Die neue Wurzel)
Ausgabe: Für jeden Knoten v (ohne r) den Vorgänger p(v).

1. Aufsplitten der Euler Tour in dem $s(< u, r >) = 0$, d.h. die zu löschende Kante wird „0“ gesetzt. U ist der letzte Knoten der Adjazenzliste von r
2. Weise jeder Kante den Wert „1“ zu und berechne die Präfixsummen parallel
3. Für jede Kante <x,y> setzen wir x =p(y), wenn die Präfixsumme von <x,y> kleiner ist als die von <y,x>



Berechnungen auf Bäume

- Berechnen der Euler Tour
- Rooting eines Baumes mit einem bestimmten Knoten (Rücktransformation)

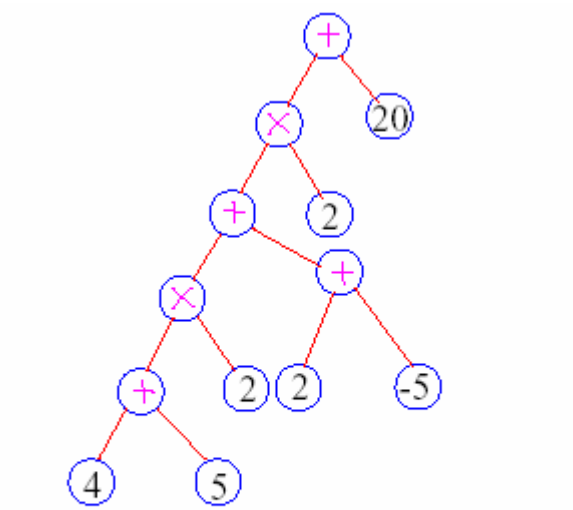
Wenn man diese zwei Schritte gemacht hat, kann man folgende Funktionen ausführen:

- Postoder berechnen
- Preorder berechnen
- Inorder berechnen
- Level jedes Knotens
- Anzahl der Nachfolger jedes Knotens

Im Prinzip geht alles dann über die Präfixsummenberechnung wobei den Knoten geschickt „1“ Werte zugewiesen werden.

Tree Contraction – Bäume Zusammenziehen

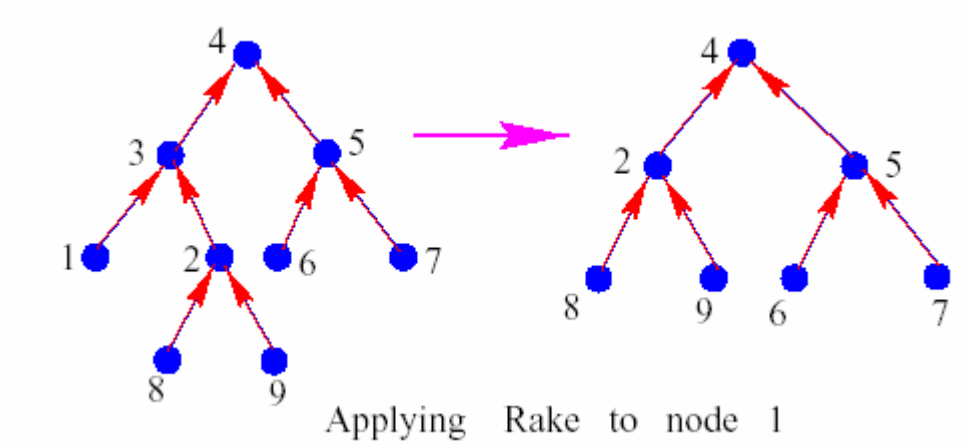
Manche Bäume sind für Euler Touren ungeeignet. Ein wichtiges Problem ist z.B. die Darstellung von arithmetischen Ausdrücken.



((4 + 5) * 2 + (-5 + 2)) * 2 + 20

Die Rake Operation

- T=(V,E) sein ein Binärbaum mit Wurzel und für jede Kante v ist p(v) der Vorgänger
- Sib(v) ist das Kind von p(v) und das Geschwisterkind von v
- Für ein Blatt u mit p(u) ungleich r tue folgendes:
  - Entferne u und p(u) von T
  - Verbinde sib(u) mit p(p(u))



Beim Baum Zusammenziehen, führen wir die Rake Operation mehrfach und parallel aus.

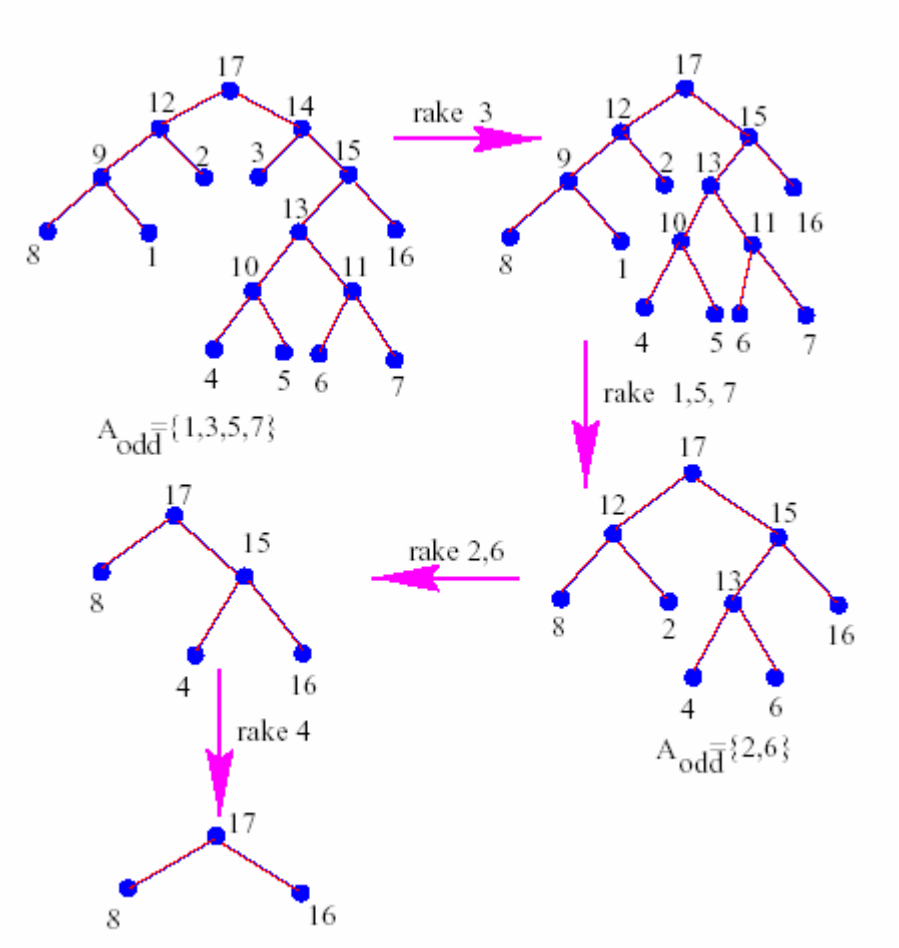
- Man kann die Rake Operation nicht ausführen, wenn die Vorgängerknoten zweier zu entfernender Blätter miteinander verbunden sind, z.B. geht hier 1 und 8 nicht!
- Man muss die Rake Operation also auf nicht zusammenhängende Blätter anwenden und zwar von links nach rechts, wie sie kommen.

Algorithmus:

1. Nummeriere die Blätter von links nach recht durch (in einem Eulerpfad erscheinen die Blätter gerade von links nach rechts)
2. Weise jeder Kante <v,p(v)> eine „1“ zu, wenn v ein Blatt ist
3. Der linkste und der rechteste Knoten werden ausgelassen, da diese beiden nach der vollständigen Kontraktion (3-Knoten Baum) als einzigste direkt mit der Wurzel verbunden sind
4. Präfixsummenberechnung auf die Ergebnisliste → damit sind die Blätter nun von links nach rechts durchnummeriert (Index)
5. Nun werden alls Blätter n in einem Array A gespeichert.
6. Im Subarray A_{odd} werden die Blätter mit den ungeraden Indexen gespeichert und in A_{even} die mit den geraden Indexen. Diese beiden Array können in konstanter Zeit O(1) und in O(n) Work erstellt werden

Nun kommt der Rake Algorithmusm zur Anwendung:

1. Führe folgendes (log n +1) mal aus:
    i. Paralleles Anwenden der Rake Operation auf alle linken Kinder die im Array A_{odd} gespeichert sind
    ii. Paralleles Anwenden der Rake Operaton auf den Rest der Elemente von A_{odd}.
    iii. Setze A auf A_{even}



Komplexität

Zeit: O(log n) Kontraktionszeit

Work: O(n) Erstellen der Euler Tour

Gesamtzahl der Iterationen:

$$O(\sum_i (\frac{n}{2^i})) = O(n)$$

Vorlesung 6.3:

Auswertung eines Ausdruckes in einem Baum

- Um einen Ausdruck bottom-up auszuwerten benötigt man O(n) Zeit für einen langen dünnen Baum. Darum wendet man die Tree Contraction an.
- Für jeden internen Knoten v weisen wir eine Bezeichnung (a_v,b_v) zu. Wobei a_v und b_v Konstanten sind.
- Der Wert eines Ausdrucks an einem Knoten ist dann: (a_vX+b_v), wobei X der unbekannte Wert des Unterbaumes von v ist.

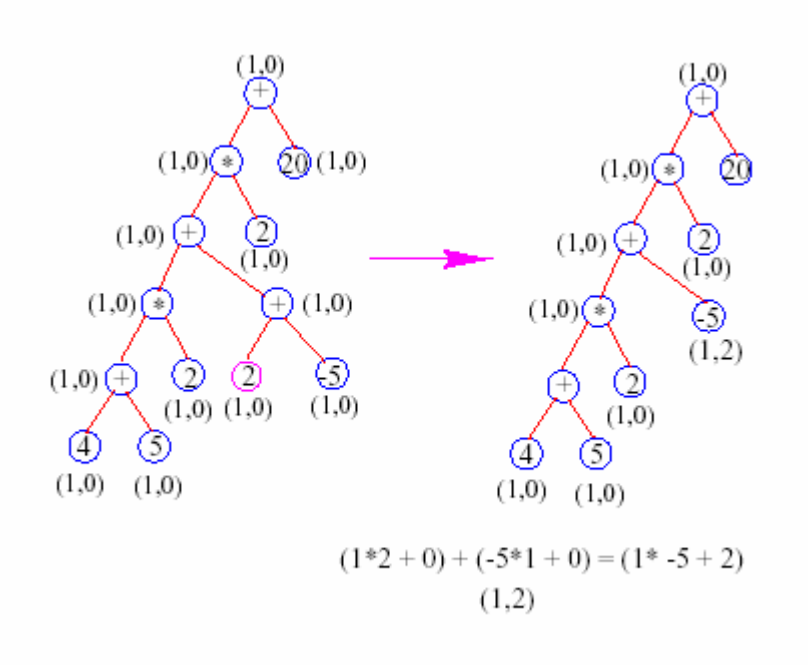
Invariante:

- Sei u der interne Knoten der den Operator $\oplus \in \{+, \times\}$ enthält.
- Seien v und w die Kinder von u mit den Bezeichnungen (a_v,b_v) und (a_w,b_w).

⇨ Dann ist der Wert bei u:

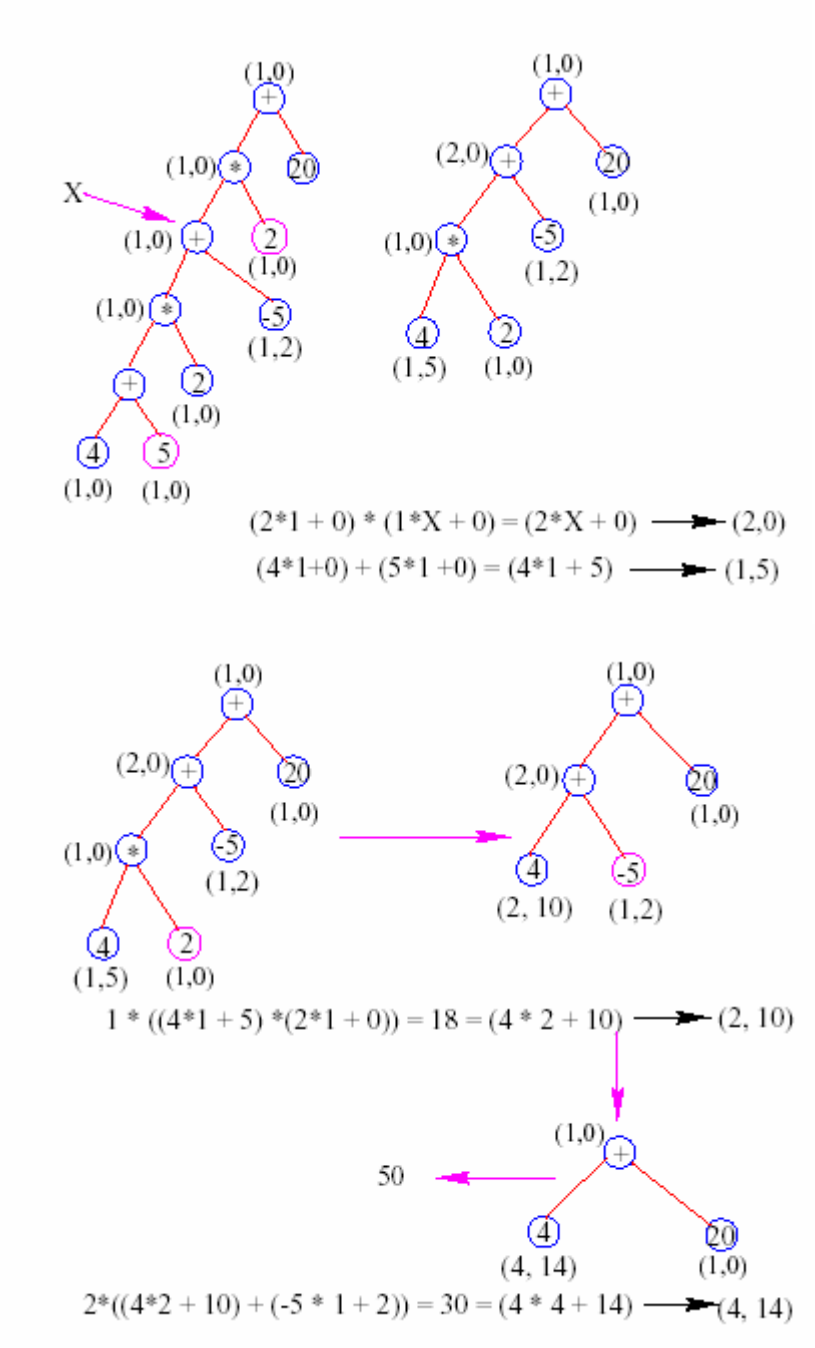
$$val(u) = (a_v val(v) + b_v) \oplus (a_w val(w) + b_w)$$

Beispiel



- Jeder Knoten wird mit (1,0) initialisiert
- Knoten „2“ markiert und der Level Hochgezogen
- Die erste Zahl in der Klammer bezeichnet den Multiplikator und die zweite Zahl den Summanden der auf die Multiplikation aufaddiert wird s.o.

- Gemäß der Rake Operation wird nun der ganze Baum zusammengezogen bis man das Endergebnis hat. (3er Baum aus dem man das Endergebnis direkt berechnen kann)



Das Gesamtergebnis des Terms ist nun 50 !

Komplexität:

Zeit: O(log n)
Work: O(n)

Vorlesung 7.1:

Paralleler Such-Algorithmus in einem sortierten Array

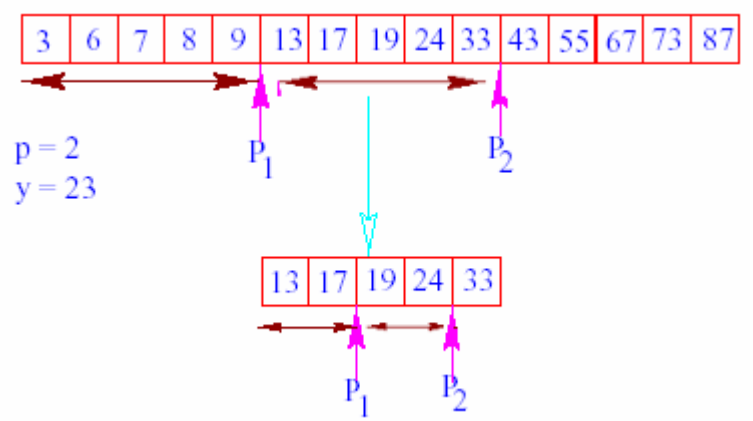
- n ist die Anzahl der Elemente des Arrays
- p (<n) ist die Anzahl der verwendeten Prozessoren
- Die Komplexität ist $O(\frac{\log(n+1)}{\log(p+1)})$ Zeit

Eingabe:

- Ein sortiertes Array S=(x1,x2,….xn)
- Eine Query Element y

Ausgabe:

- Zwei Elemente (x_i und x_{i+1})aus S zwischen denen y liegt ! $x_i \leq y \leq x_i+1$



(ACHTUNG; Fehler in der Folie, hier korrigiert!)

- Durch mehrere Iterationen wird das betreffende Array immer mehr verkleinert. Es wird in p+1 Teile unterteilt, wobei wir p mal Binärsuche anwenden um y zu finden. Die wird solange iteriert, bis die Größe des Arrays auf p gesunken ist. Dann folgt ein direkter Vergleich (jeder Prozessor hat nun ein Element und betrachtet seines und das des Nachfolgers)
- Jeder Prozessor betrachtet sein letztes Element und vergleicht ob es größer oder kleine y ist. Mit den beiden Ergebnissen wird das nächste Intervall festgelegt, in welchem die Iteration weiterläuft.
- Wenn das Array jetzt nur noch Größe p hat (entspricht der Anzahl der Prozessoren), dann wird wie folgt vorgegangen.
  - Wir weisen jedem Element einen Prozessor zu
  - Direkter Vergleich seine Elements und des Nachfolgers
  - In konstanter Zeit möglich !

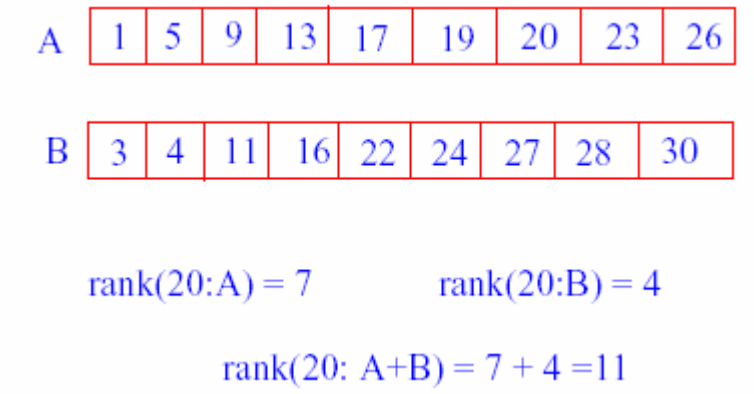
Komplexität:

- In der ersten Iteration wird die Größe des Arrays von n auf n/(p+1) reduziert
- In weiteren k Iterationen reduziert sich die Größe auf p
- Deshalb $n/(p+1^k)=p \rightarrow n=(p+1)^{k+1}$

⇨ $O(\frac{\log(n+1)}{\log(p+1)})$

Merging

Normales Ranking



- ⇨ Rankings von jedem Element berechnen
- ⇨ Daraus die Rankings der Ergebnisliste berechnen du damit das Ergebnisnaufbauen

Ranking einer kurzes Zahlensequenz in eine sortierte Liste

- ⇨ Sei X die sortierte Liste mit n Elementen
- ⇨ Sei Y eine willkürliche Liste von $m=O(n^s)$ Elementen wobei s zwischen 0 und 1 liegt (die Y Liste ist damit kleiner als die X Liste !!!)
- ⇨ Wenn wir $p = \lfloor \frac{n}{m} \rfloor = O(n^{1-s})$ Prozessoren verwenden, dann können wir jedes Element von Y in $O(\frac{\log n}{\log p}) = O(1)$ Zeit in X ranken.

Ein schneller Merging-Algorithmus

- ⇨ Wir haben nun zwei sortierte Arrays A und B mit n bzw. m Elementen.
- ⇨ Schnelles Merging ist ein wichtiger Bestandteil von Divide and Conquer Sortieralgorithmen

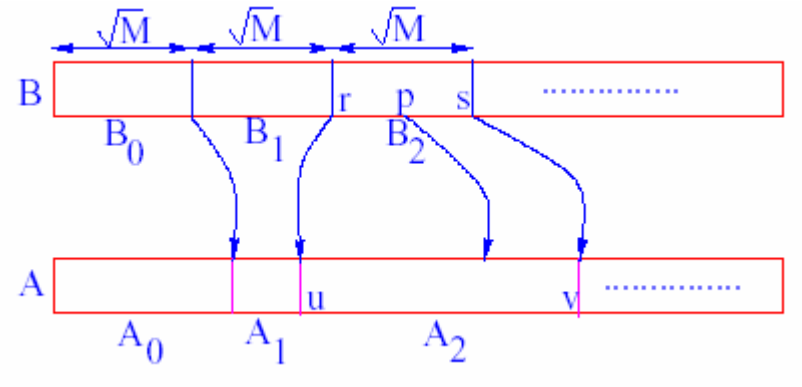
Eingabe:

- Zwei sortierte Listen A und B mit n bzw. m Elementen

Ausgabe:

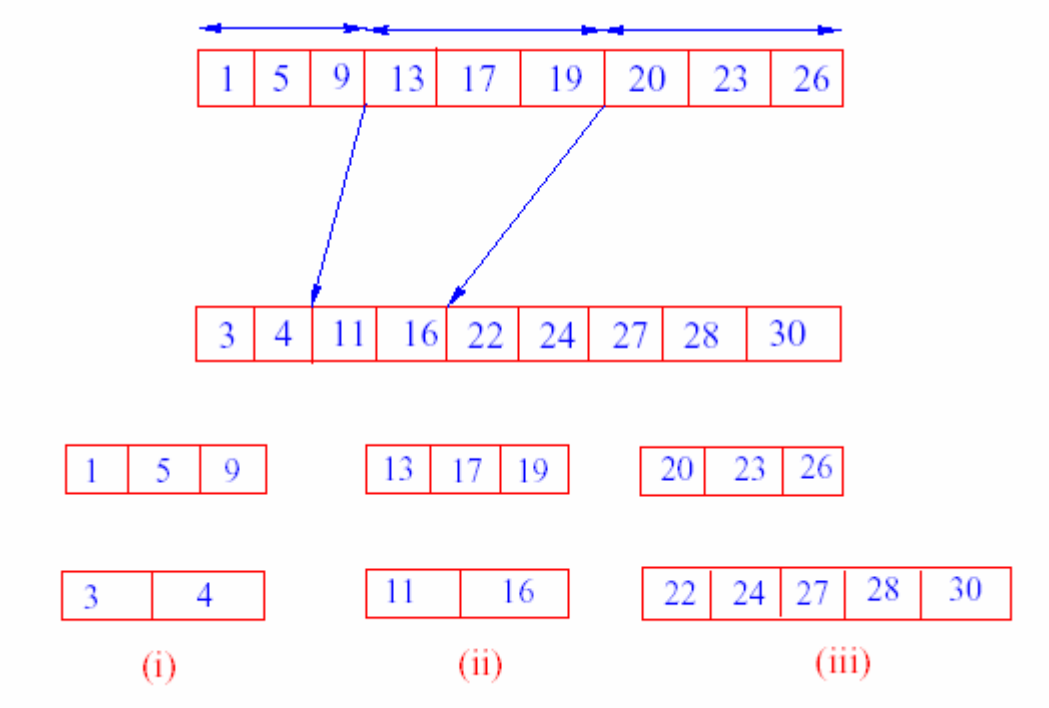
- rank(B:A) und rank(A:B)

- Wir teilen das Array B in SQRT(m) Teile auf mit SQRT(m) Elementen
- Wir ranken nun jeweils das letzte Element von B in die Liste A dadurch kann man die nun korrespondierenden Listen unabhängig und parallel mergen.

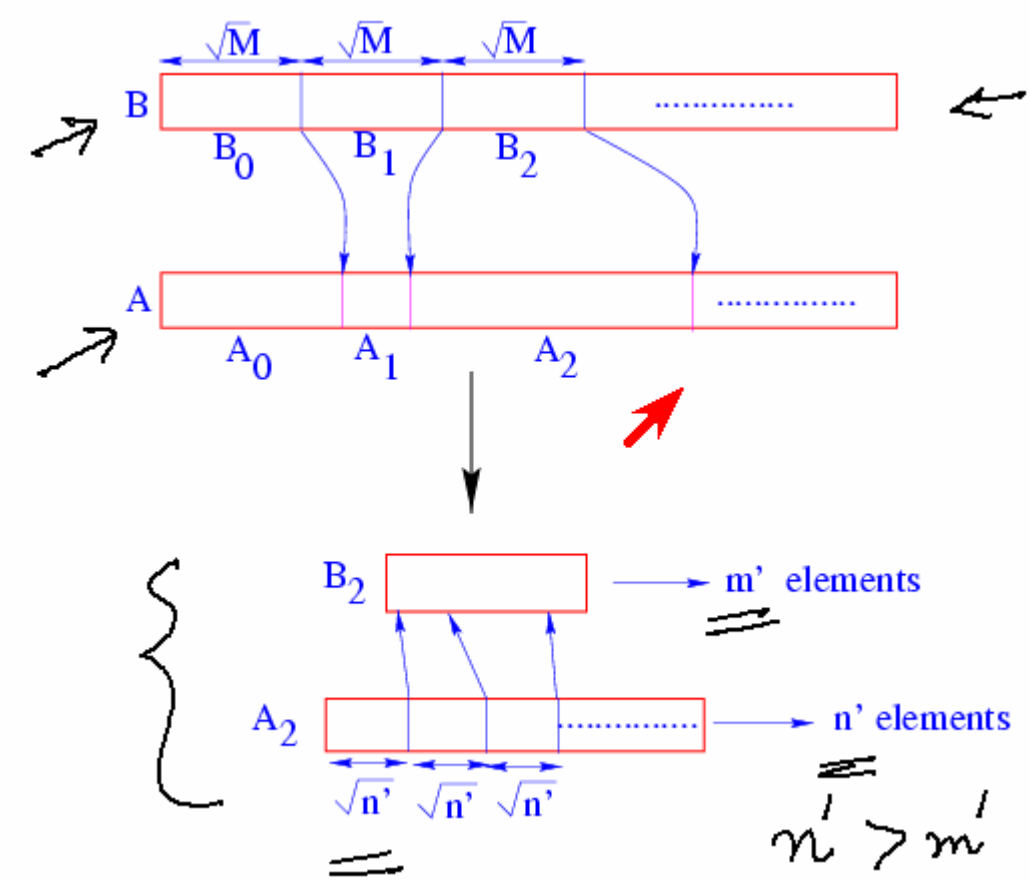


- Die SQRT(m) Elemente aus B können nun in O(1) in mittels paralleler Binärer Suche mit m Prozessoren in A gerankt werden.
- Die Liste A hat auch SQRT(m) Blöcke
- Die Blöcke können nun paarweise und rekursiv gemerged werden
- Diese beiden paarweisen Blöcke nennen wir B' und A' mit den Elementen m' und n'
- Wenn nun m' größer als n', dann wird B' unterteilt in SQRT(m') Blöcke
- Wenn n' größer als m', dann wird A' unterteilt in SQRT(n') Blöcke
- Die jeweils korrespondierende Liste muss natürlich dann auch wieder gemäß der Endelemente unterteilt werden → Rekursion

Beispiel:



Vorlesung 7.2:



- Die Rekursion zwischen allen Paaren von Blöcken läuft parallel ab
- Die Rekursion stoppt, wenn das Subproblem klein genug ist, dass es sequentiell in O(1) gelöst werden kann
- Am Ende des Algorithmus kennen wir rank(A:B) und rank(B:A), dann können wir die Elemente in sortierter Reihenfolge in eine andere Array schreiben

Komplexität:

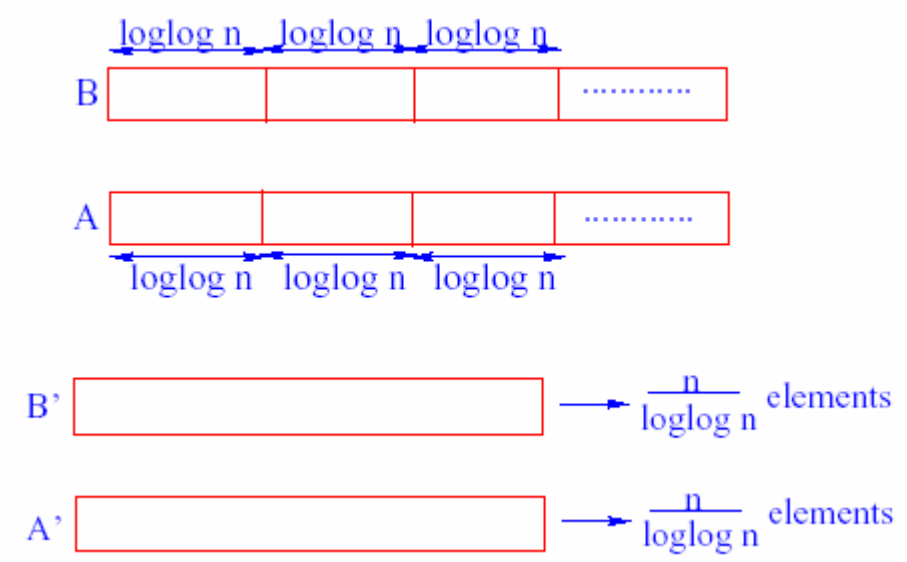
Zeit: O(log log m)
Work O((m+n)log log m)

Dieser Algorithmus ist zwar Zeitoptimal, aber noch nicht Work-optimal. Im folgenden machen wir den Algorithmus work-optimal

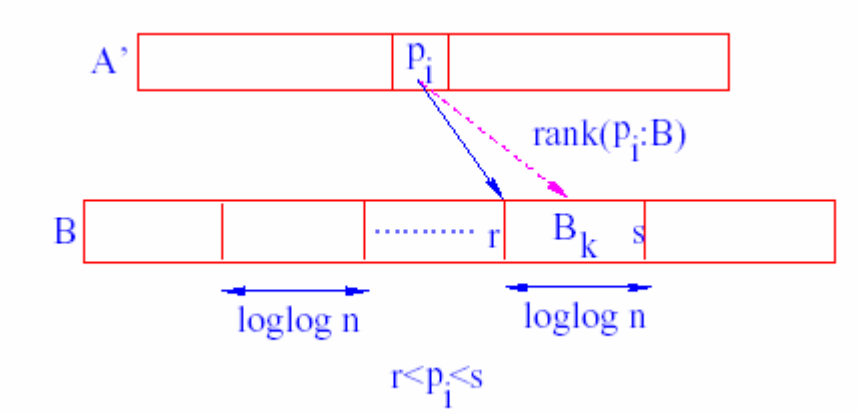
Work-optimaler Merging Algorithmus

Work-Optimal heisst, ihn auf Workingkomplexität von O(m+n) zu bringen

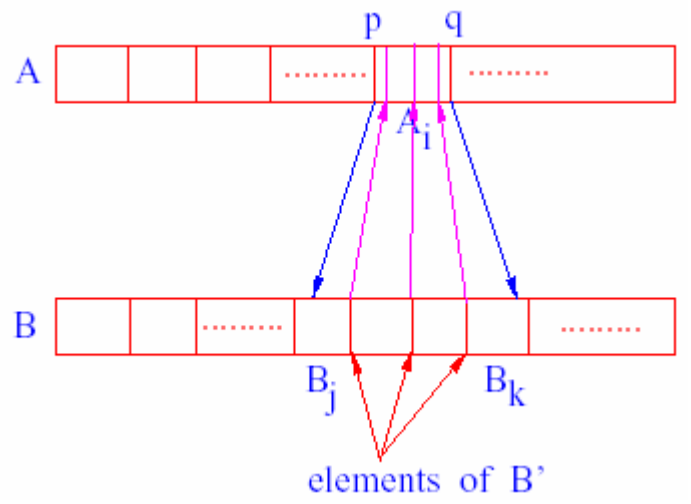
- Wir haben nun der Einfachheit n Elemente in jeder Liste
- Wir teilen in (n/log log n) Blöcke von log log n Elementen auf
- Wir nehmen das letzte Element jedes Blocks von A bzw. B und schreiben diese in eine Liste A' bzw. B' (A' und B' enthalten nur die Endelemente). Nun haben A' bzw. B' (n/log log n) Elemente.



- Nun berechnen wir rank(A':B') und rank(B':A'). Das dauert O(log log n) Zeit und letztendlich O(n) work (O((n/log log n) X log log n))
- Jetzt berechnen wir mittels eines Prozessors noch rank(A':B) um zu sehen in welchen Blöcken B die Boundary Elemente aus A' liegen



- Jetzt wird die genaue Position (ranking) von Pi in Bk mittels Binärer Suche und einem Prozessor ermittelt. Dies dauert wegen der (log log n) Elemente (log log log n) Zeit !
- Da alles parallel berechnet wird, benötigen wir O(n/loglog n) Prozessoren
- rank(B':A) geht analog



- Betrachten wir nun Ai, einen der log log n großen Blöcke in A
- Wir kennen rank(p:B) und rank(q:B) für die beiden Boundary Elemente p,q aus A
- Nun rufen wir unseren Algorithmus rekursiv auf mit Ai und den Elementen die zwischen rank(p:B) und rank(q:B) lin B liegen
- Möglicherweise sind aber zu viele Elemente dazwischen, und dadurch auch mehrere Blöcke von B. Also werden nun die Grenzen dieser Blöcke wieder in Ai gerankt
- Dann bekommen wir Paar von Blöcke n ein log log n großer Block aus B und ein kleinerer aus Ai
- Jetzt haben wir nur noch ein Teilproblem von O(log log n) Größe
- Es gibt O(n/log log n) Paare die von O(n/log log n) Prozessoren in O(log log n) Zeit gemerged werden.
- Die verschiedenen ranking Berechnungen dauern O(log log n Zeit) und benötigen O(n) work,. Auch der letzte Mergeschritt benötigt diese Zeit.
- Also können wir zwei sortierte Arrays der größe n in

Zeit O(log log n)
Work O(n)

Auf einem CREW PRAM Berechnen.

Effizient sortieren

- Mit dem eben vorgestellten Merging Algorithmus kann man wunderbar sortieren (Merge Sort)
- Man hat ein unsortiertes Array von Elementen und teilt dies rekursiv (DaC) in gleich große Teile bis auf Blattebene auf (ein Element pro Blatt)
- Dann werden die Kinder-Arrays gemerged (mit dem optimalen Mergealgorithmus) bis hin zur Wurzel
- Auf jedem Level des Binärbaumes haben wir n Elemente verteilt auf verschiedene Arrays
- Damit benötigen wir O(log log n) Zeit und O(n) work in jedem Level zum paarweisen Mergen. Der Binärbaum hat eine Tiefe von O(log n) also

Zeit O(log n log log n)
Work O(n log n)

- Dieser Algorithmus ist zwar workoptimal, aber nicht Zeitoptimal. Es gibt noch einen besseren genannt „Cole's pipelined merge sort algorithm" der nur O(log n) Zeit benötigt und sogar auf einem EREW PRAM läuft. S gitbt aber auch noch einen viel eingacheren aber gleich effizienten Algorithmus, der mit Randomisierung arbeitet

Vorlesung 8.1:

Parallele Graphen Algorithmen

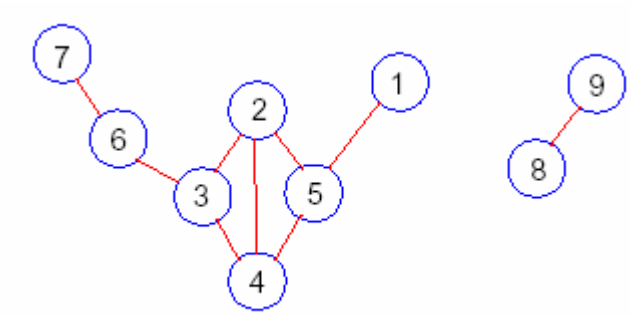
Verbundene Komponenten

Sei G=(V,E) ein ungerichteter Graph mit |V|=n und |E|=m

Definition: Zwei Knoten u und v sind miteinander verbunden:

- Wenn u und v gleich sind, oder
- Wenn es einen Pfad von u nach v gibt

Diese Relation ist eine Äquivalenzrelation (Transitiv, reflexiv, symmetrisch)
Jede Äquivalenzklasse wird dann verbundene Komponente aus G genannt



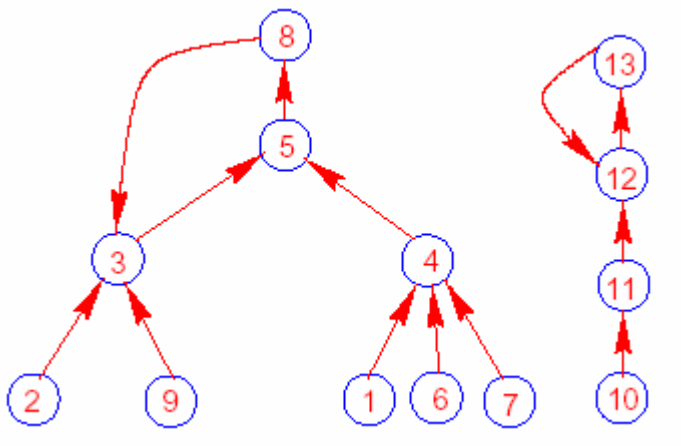
	1	2	3	4	5	6	7	8	9
1	0	0	0	0	1	0	0	0	0
2	0	0	1	1	1	0	0	0	0
3	0	1	0	1	0	1	0	0	0
4	0	1	1	0	1	0	0	0	0
5	1	1	0	1	0	0	0	0	0
6	0	0	1	0	0	0	1	0	0
7	0	0	0	0	0	1	0	0	0
8	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	1	0

Der Algorithmus bestimmt nun alle zusammenhängenden Knoten jedes Graphen

2 versch Algorithmen, Eingabe durch

- Adjazenzmatrix:
  - Time O(log²n)
  - Work O(n²) (Jeder Eintrag der Matrix muss mindestens einmal gelsen warden)
- Kantenliste
  - Time: O(log n)
  - Work: O((m+n)log n) (Sehr gut für einen dünnen Graphen)

Strategie mit Pseudoforest



- Ein Pseudoforest ist ein gerichteter Graph mit Outdeg=1
- Die Wurzel muss auch wieder mit einem Knoten verbunden werden (=> Genau 1 Zyklus)

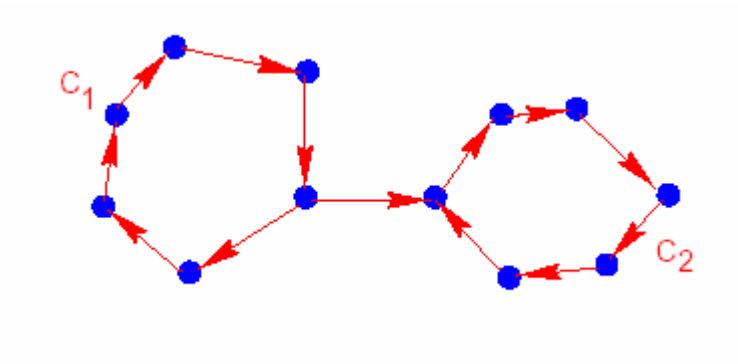
Allgemeine Strategie:

- Man definiert sich eine Funktion D:V→V die uns einen Pseudoforest (V,F) erzeugt. Wobei F={<v,D(v)>|v € V}. Die Rücktransformation geht nicht immer.

Eigenschaften des Pseudoforest

⇨ Jeder gerichtete Baum mit Wurzel in einem Pseudoforest enthält einen Zyklus

Beweis:



- Wenn es zwei Zyklen C1 und C2 in einem Baum gibt, dann müssen diese verbunden sein (s.o.) Wenn dies aber der Fall ist, dann gibt es automatisch einen Knoten mit Outdeg=2. → Es gibt nur einen Zyklus

Vorgehensweise zur Erstellung des Pseudoforest:

- Der Algorithmus ist Iterativ
- In jeder Iteration werden Gruppen von adjazenten Knoten in größere Gruppen gemerged
- Der Algorithmus terminiert wenn es nichts mehr zu mergen gibt
- Jede Gruppe wird nun repräsentiert durch einen gerichteten Graphen mit Wurzel, wobei die Wurzel der Repräsentant der Gruppe ist

Optimaler Algorithmus für dichte Graphen (Gespeichert in Adjazenzmatrix !!!)

Sei A eine nXn Adjanzenzmatrix eines ungerichteten Graphen G=(V,E)

- A(i,j)=1, wenn i,j € E
- C(v)=min{u|A(u,v)=1}
  - C(v)=v, wenn v ein isolierter Knoten ist (ohne Nachbarknoten)
  - C(v) ist der kleinste Knoten adjazent zu v

Lemma:

C definiert einen Pseudoforest F der die Knotenmenge V aufteilt in V1…Vs, wobei jede Vi ein Set von Knoten aus V des gerichteten Baumes Ti ist

1. Alle Knoten in jedem Vi gehören zur gleichen zusammenhängenden Komponente von G
2. Jeder Zyklus in F ist entweder ein Zyklus auf sich selbst oder er enthält nur zwei Kanten (zwischen zwei benachbarten Knoten)
3. Der Zyklus jedes Baumes Ti in F enthält den kleinsten Knoten in Ti

Die nun folgenden Beweise haben wir mit Mut zur Lücke weggelassen.

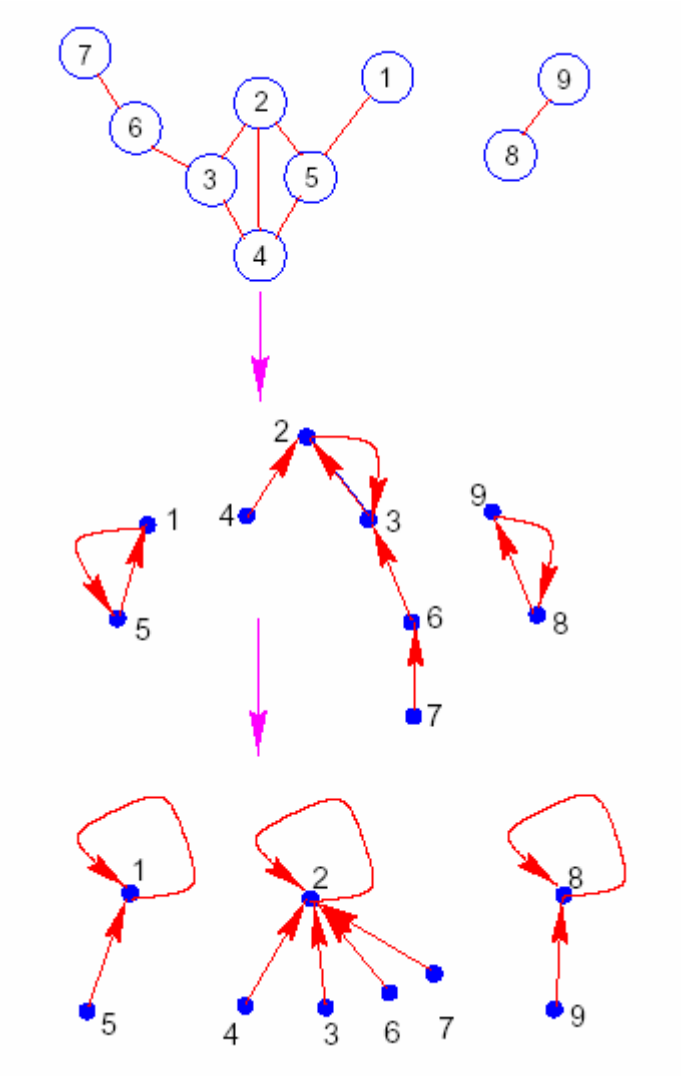
Vorlesung 8.2:

Die Iteration im Einzelnen:

1. Berechne die C Funktion für jeden Knoten aus jeder Zeile der Adjazenzmatrix
    - Für jeden Knoten v_i ist dies der kleinste Index der i-ten Zeile, welcher eine „1“ enthält.
2. Schrumpfe jedes Set Vi von Knoten zu einem „Super-Knoten“. (Stern machen)
3. Erstelle eine neue Adjazenzmatrix aus den „Super-Knoten“.

Schritt 2 genauer:

- Schrumpfen der Knotensets zu Sternen, d.h. alle Knoten sind die Kinder ihrer Wurzel (wir möchten damit von jedem Knoten die Wurzel wissen)
- Stichwort: „Pointer Jumping“ – Modifiziert zum Erkennen von Zyklen, sonst Endlosschleife.
- Danach ist jeder Knoten repräsentiert durch die Wurzel des Sternes

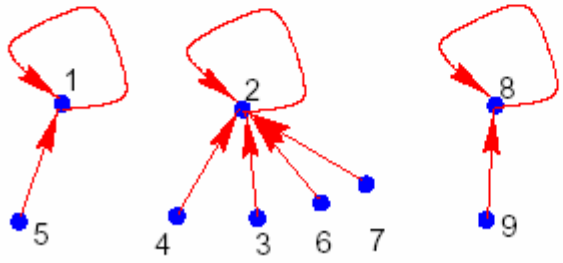


Schritt 3 genauer:
Erstellen der Adjazenzmatrix für die Superknoten (oben: Knoten 1,2,8)

- Vergleiche die Kindknoten und die Superknoten (Verknüpfungen zwischen zwei Superknoten gibt es natürlich nicht) der einzelnen Sterne paarweise miteinander und prüfe in der Ausgangsadjazenzmatrix ob diese eine gemeinsameKante besitzen. (oben sind das die Vergleiche: (5,4),(5,3)…(5,7),(4,9)…(7,9),(5,9) => die Sterne 1 und 2 sind z.B. über die Kante zwischen den Knoten 5 und 4 miteinander verbunden (siehe ursprüngliche Adjazenzmatrix))

	1	2	3	4	5	6	7	8	9
1	0	0	0	0	1	0	0	0	0
2	0	0	1	1	1	0	0	0	0
3	0	1	0	1	0	1	0	0	0
4	0	1	1	0	1	0	0	0	0
5	1	1	0	1	0	0	0	0	0
6	0	0	1	0	0	0	1	0	0
7	0	0	0	0	0	1	0	0	0
8	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	1	0

A_1

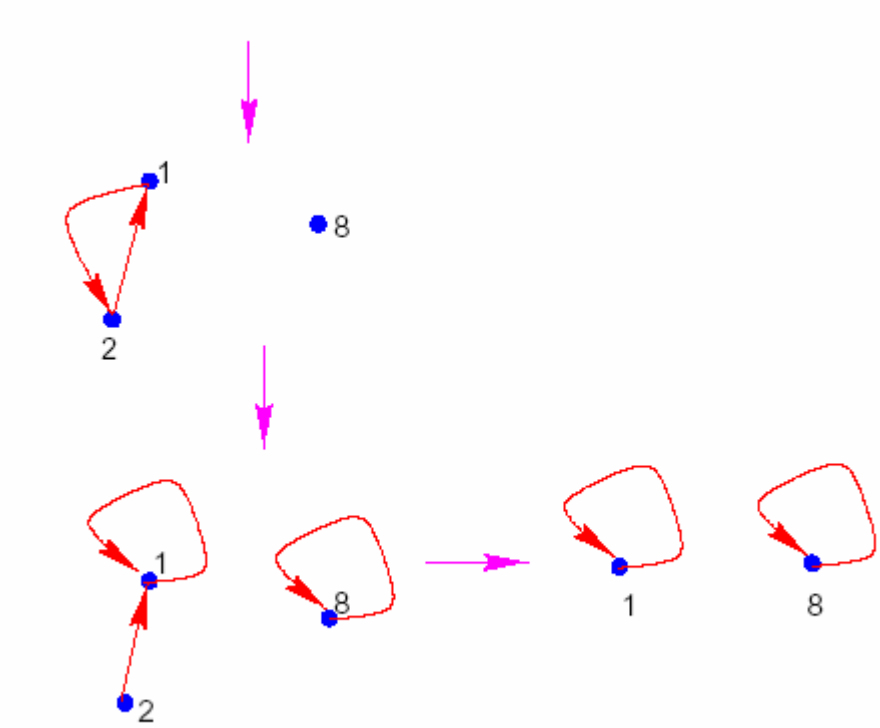


	1	2	8
1	0	1	0
2	1	0	0
8	0	0	0

A_2

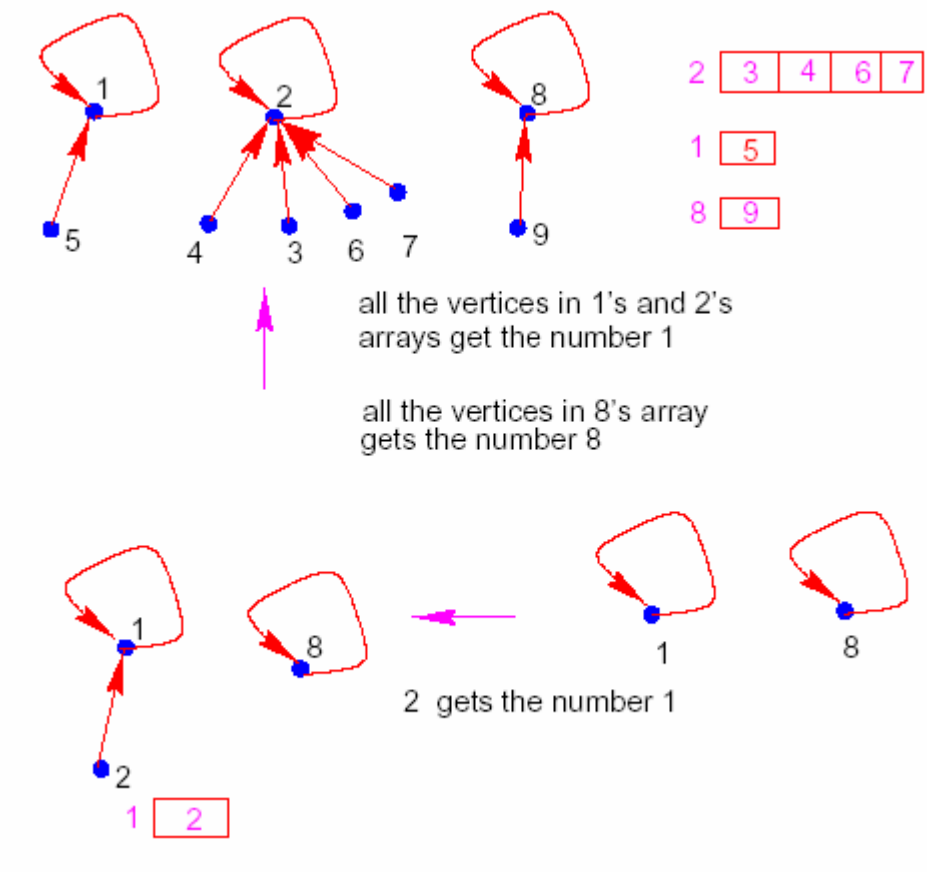
Als Ergebnis iseht man nun aus A_2 dass die beiden Sterne mit den Superknoten 1 und 2 miteinander verbunden sind

Nächste Iteration:



Am Ende haben wir nun in unserem Beispiel nur noch 2 „Connected Components“ („1“ und „8“)

- Nummerierung aller Knoten der einzelnen Komponenten mit ihrem „Superknoten“
- Um dies zu erreichen werden die Schritte des Algorithmus umgekehrt ausgeführt



Knoten 2 bekommt die Nummer „1“ zugewiesen und damit bekommen alle Kinder von „1“ und „2“ den Wert „1“ zugewiesen. Die Kinder von Knoten „8“ bekommen den Wert „8“ zugewiesen.

Komplexität

Schritt 1 der Iteration: Berechnen der C Funktion (n minimum Berechnungen):

Für jede Zeile benötigen wir

Work: O(n)
Time: O(log n)

Und damit Gesamt:

Work: $O(n^2)$
Time: O(log n) (wegen parallelberechnung)

Schritt 2 der Iteration: Zusamennziehen zu einem Stern mittels Pointer Jumping bzw. List Ranking

Work: O(n log n)
Time: O(log n)

Schritt 3 der Iteration: Berechnen der Adjazenzmatrix der Superknoten

Zeit: O(1)
Prozessoren $O(n^2)$

Work:

k=k-ter Iterationsschritt
n_k= Anzahl der zu bearbeitenden Elemente im k-ten Iterationsschritt.

$$\sum_k O(n_k^2) = \sum_k O(n^2/2^k) = O(n^2)$$

Insgesamte Komplexität

- Wir benötigen das „Common CRCW“ Modell (änderbar in CREW Modell)
- Workkomplexität: $O(n^2)$
- Zeitkomplexität: O(log n)

Die Workkomlexität könnte viel günstiger als $O(n^2)$ sein, wenn man einen „Sparsegraphen“ (wenig Kanten) hat, der anstatt in eine Adjazenzmatrix in eine Kantenliste gespeichert ist. Für „Densegraphen“ (viele Kanten) geht das nicht schneller.

Vorlesung 9.1:

Optimaler Algorithmus für dünne Graphen (Gespeichert in Kantenliste)

- Das problem bei dem Algorithmus für dichte Graphen eben war, dass wir in jeder Iteration den „rooted directed tree“ in einen Stern umwandeln mussten, was O(log n) Zeit benötigte.

- Die C Funktion war zu eng definiert, deshalb konnten wir die den „rooted directed trees“ nicht effizient zusammenfügen.

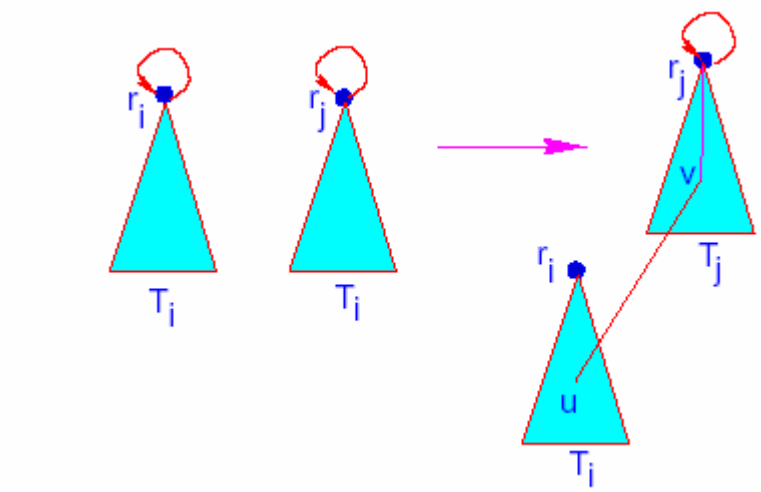
- Im nun folgenden Algorithmus werden wir eine neue, allgemeinere Funktion D verwenden um die Bäume zu mergen.

- Wir werden auch nicht in jeder Iteration den Baum in einen Stern umwandeln sondern wir werden den Pointer Jumping Algorithmus inkrementell verwenden

Grafting

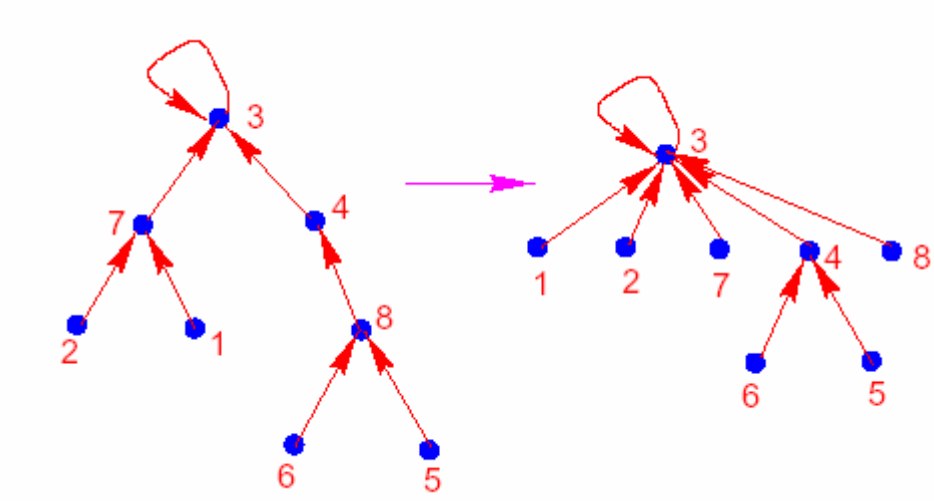
Sei D eine Funktion auf die Knotenmenge V

- D.h. D angewendet auf v ergibt einen neuen Knoten w
  - D(v)=w
- Initilisiert ist jeder Knoten v für D(v)=v



- Ti und Tj sind zwei Bäume des Pseudoforest der durch D definiert wird
- ri und rj sind die Wurzeln der beiden Bäume und v ist ein Knoten von Tj
- Das Grafting von Ti auf Tj wird mittels D(ri)=v gemacht

- Um nun die Höhe der entstandenen Bäume wieder zu reduzieren, wendet man Pointer Jumping an.

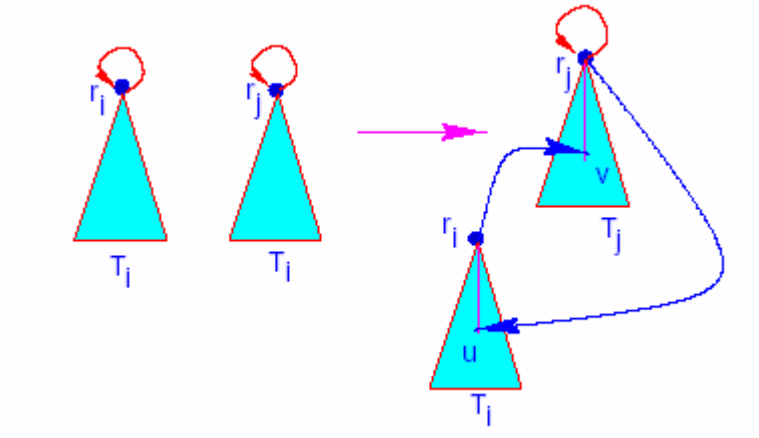


Das Pointer Jumping auf einen Knoten v hat den Effekt D(v)=D(D(v))

Gesamte Strategie

- Wir repräsentieren jeden Baum durch seine Wurzel, deshalb wissen wir nicht zu welchem Baum die Knoten u und v gehören !
- Wenn wir nun von einem Knoten die Wurzel kennen, dann können wir Grafting in O(1) anwenden. Entweder u oder v ist nun ein Kind seiner Wurzel.

- Wenn nun beide Knoten direkte Kinder ihrer Wurzeln sind, dann kann passieren, dass Ti nach Tj gegrafted wird und umgekehrt, damit erhalten wir einen ungewollten Zyklus.



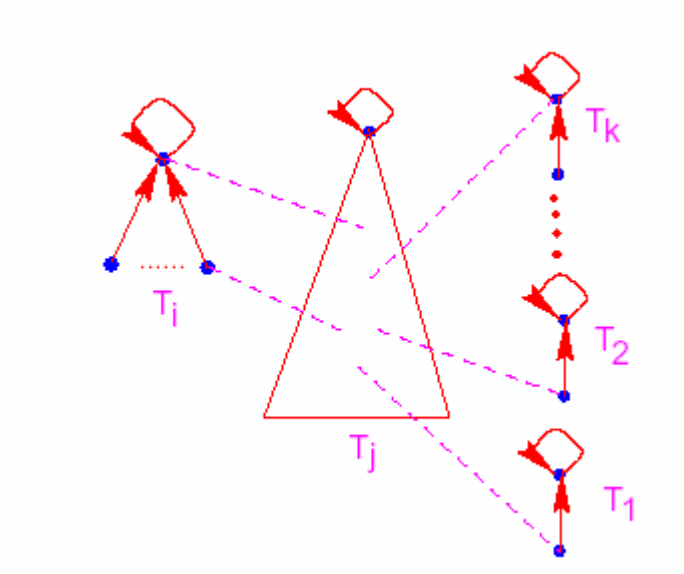
- Für jede Kante (u,v)€ E, wenn D(u)=D(D(u)) und wenn D(v)<D(u), dann graften wir Ti auf Tj . Wir setzen D(D(u))=D(v).
    - In anderen Worten: Wenn D(u) eine Wurzel ist (dann gilt D(u)=D(D(u))) und wenn d(v)<D(u) dann setzen wir die Wurzel von u unter v (Graften)

Problem:

Wir wollen sicherstellen, dass die Bäume um einen konstanten Faktor verkleinert werden, damit wir das problem in O(log n) Zeit lösen können, sonst klappt das nicht.

Die Grafting Operation oben stellt dies nicht sicher.

Mit diesem Conditional Grafting könnte es sogar Ω(n) dauern:
Beispiel (Worst Case):



- Die Knoten des Sterns Ti sind kleiner als alle adjazenten Knoten von Tj
- Der gerichtete Baum Tj hat kleinere adjazente Knoten in den Sternen T1…Tk

- Aber es gibt keine Kante zwischen zwei Bäume T1…Tk
- In diesem Fall wird Tj als erstes nach T1 gegrafted. Dann wird Tj U T1 nach T2 gegrafted usw.
- Somit wird am Ende(Tj U T1 U T2….U Tk) nach Ti gegrafted
- So dauert es Ω(n) Time um alle Bäume zu mergen

Also, wir benötigen eine andere Operation um Bäume schneller zu mergen

⇨ Unconditional Grafting

Nach jeder Iteration von Conditional Grafting versuchen wir nun die Sterne in andere Bäume zu mergen:

for all $(i,j) \in E$,

if i belongs to a star and $D(j) \neq D(i)$, then set

$D(D(i)) := D(j)$.

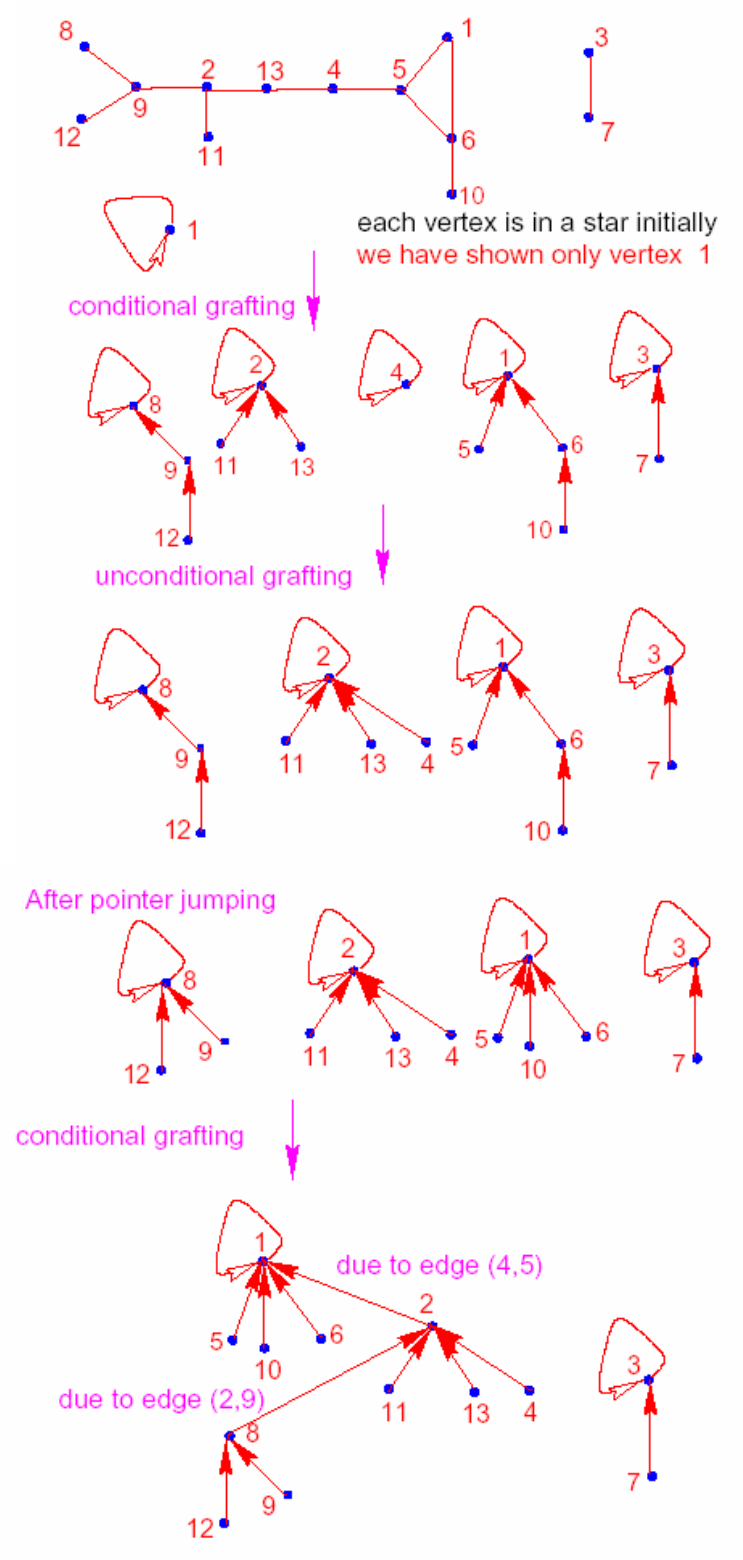
In unserem schlechten Beispiel eben, könnten wir nun alle Sterne T1…Tk in einer Iteration nach Tj mergen.

Beobachtungen:

- Sowohl nur conditional- als auch nur unconditional Grafting ist nicht ausreichend ! Verwenden wir nur unconditional Grafting, erhalten wir Zyklen.

- Also führen wir in jeder Iteration erst conditional Grafting und dann unconditional Grafting durch. (Macht man es andersherum, kann es wieder Zyklen geben)

Vorlesung 9.2:



Erkennen eines Sternes

Für unconditional Grafting müssen wir erkennen können, ob eine zusammenhängende Komponente ein Stern Ist.

Wir weisen jedem Knoten v einen Prozessor zu, welcher folgendes berechnet:

Set $Star(v) = true$

if $D(v) \neq D(D(v))$ then

Set $star(v), star(D(v)), star(D(D(v))) := false$

Set $star(v) := star(D(v))$

Star(v) ist am Ende „true“ wenn v zu einem Stern gehört, sonst nicht.

Korrektheit:

Behauptung 1:

Am Ende jeder Iteration sollte gelten:

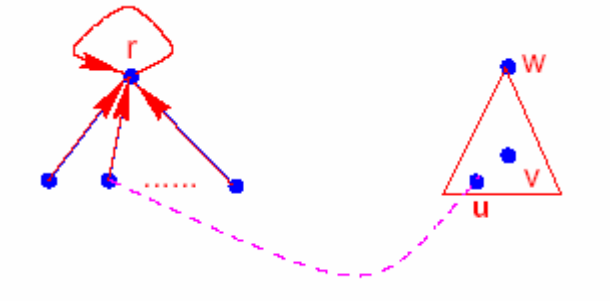
- Der durch D definierte Pseudoforrest besteht aus „Directed Trees“ mit Selbst-Loops an den Wurzeln
- Alle Knoten in einem Baum gehören zur gleichen verbundenen Komponente

⇨ Prüfung durch Induktion über die Iterationsnummer k
⇨ Basisfall k=0 → Erfüllt die Bedingung
⇨ Gilt für i-te Iteration → soll auch für die i+1-te Iteration gelten
⇨ Wir wenden nur conditional- unconditional Grafting und Pointer Jumping an. Die 3 Operationen verändern an der Bedingung nichts, also ist die Behauptung auch für den i+1-ten Schritt erfüllt

Behauptung 2:

- Sei r die Wurzel eines Sternes nach einer Iteration
- Alle Knoten v in der Zusammenhangskomponente von r hängen direkt an r (D(v)=r für alle v)

⇨ Beweis durch Gegenbeweis:
⇨ Sei v ein Knoten und D(v) != r;
⇨ v ist in einem „rooted directed tree“ mit Wurzel w



⇨ wenn w eine Wurzel ist und v direkt an w hängt, v aber auch in der Zusammenhangskomponente von r ist, dann kann r keine Wurzel eines Sternes sein.

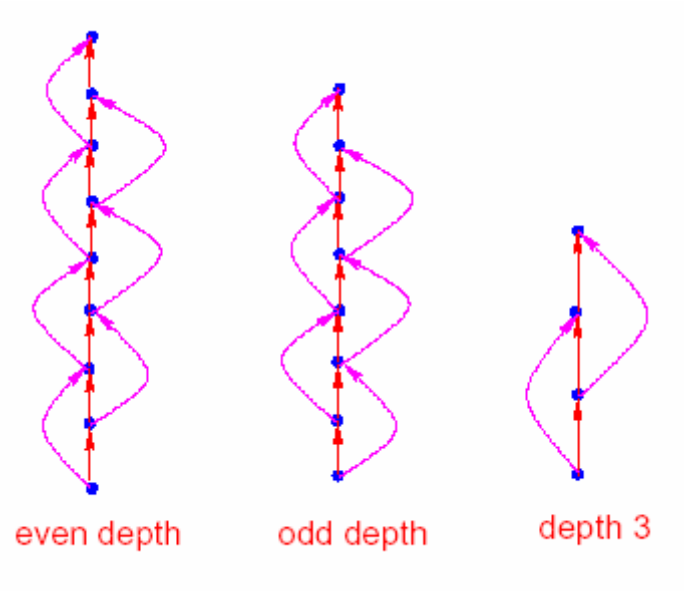
⇨ Der Knoten r muss also an den Baum (mit Wurzel) w gegraftet worden sein

Implikationen:

- ⇨ Wenn ein „rooted directed tree“ ein Stern ist am Ende einer Iteration, dann sind alle Knoten dieses Sternes in der gleichen Zusammenhangskomponente.
  - Deshalb können wir die Iteration stoppen, wenn alle „rooted directed trees“ nun Sterne sind.
  - Wie man einen Stern erkennen kann wissen wir bereits
- ⇨ Wir haben nun alle verbundenen Komponenten gefunden, wenn die Bäume alle Sterne sind

Behauptung 3:

- Sei d die Tiefe des Baumes T bevor Pointer Jumping in der Iteration angewendet wurde.
- Nach dem Pointer Jumping ist die Tiefe 2d/3.
- ⇨ Wenn d gerade ist, dann haben wir nach dem Pointer Jumping d/2 als Tiefe
- ⇨ Wenn d ungerade ist, dann haben wir danach (d+1)/2 als Tiefe
- ⇨ Am schlechtesten ist es bei d=3, dann haben wir eine Tiefe von 2 nach dem Pointer Jumping
- ⇨ Also reduziert sich die Tiefe auf mindestens 2d/3



Behauptung 4:

- Sei K eine zusammenhangskomponente von G
  - |K| ist die Anzahl der Knoten in K
  - Am Ende der k-ten Iteration ist $h_k(K)$ die Summe der Höhen der Bäume die die Knoten aus K enthalten
  - Wenn alle Knoten aus K noch keinen Stern gebildet haben, dann gilt: $h_k(K) \leq (2/3)^k |K|$

- ⇨ Wenn wir zwei Bäume aus K aneinander Graften, dann erhöht sich die Gesamthöhe der Bäume nicht
- ⇨ In jedem Schritt reduziert das Pointer Jumping die Höhe jedes Baumes aus K um 2/3
- ⇨ Nach k-Iterationen ist die Gesamthöhe dann $\leq (2/3)^k |K|$
- ⇨ Somit ist nach O(log n) Schritten die Gesamthöhe auf 1 reduziert, also, wenn alle Knoten in einem Stern sind

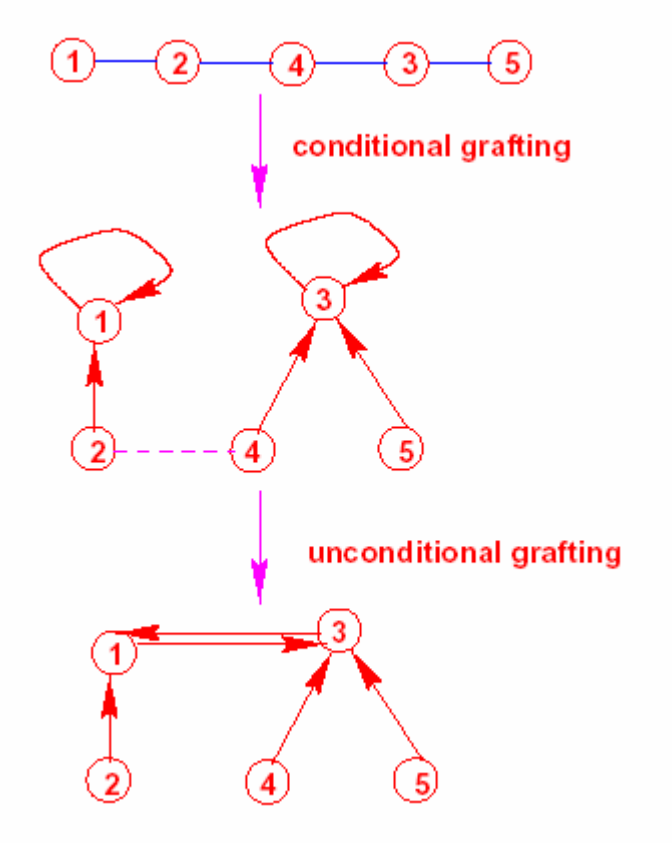
⇨ Das heisst also, dass wir alle Knoten in den Zusammenhangskomponenten in O(log n) Schritten zu einem Stern machen können

Komplexität:

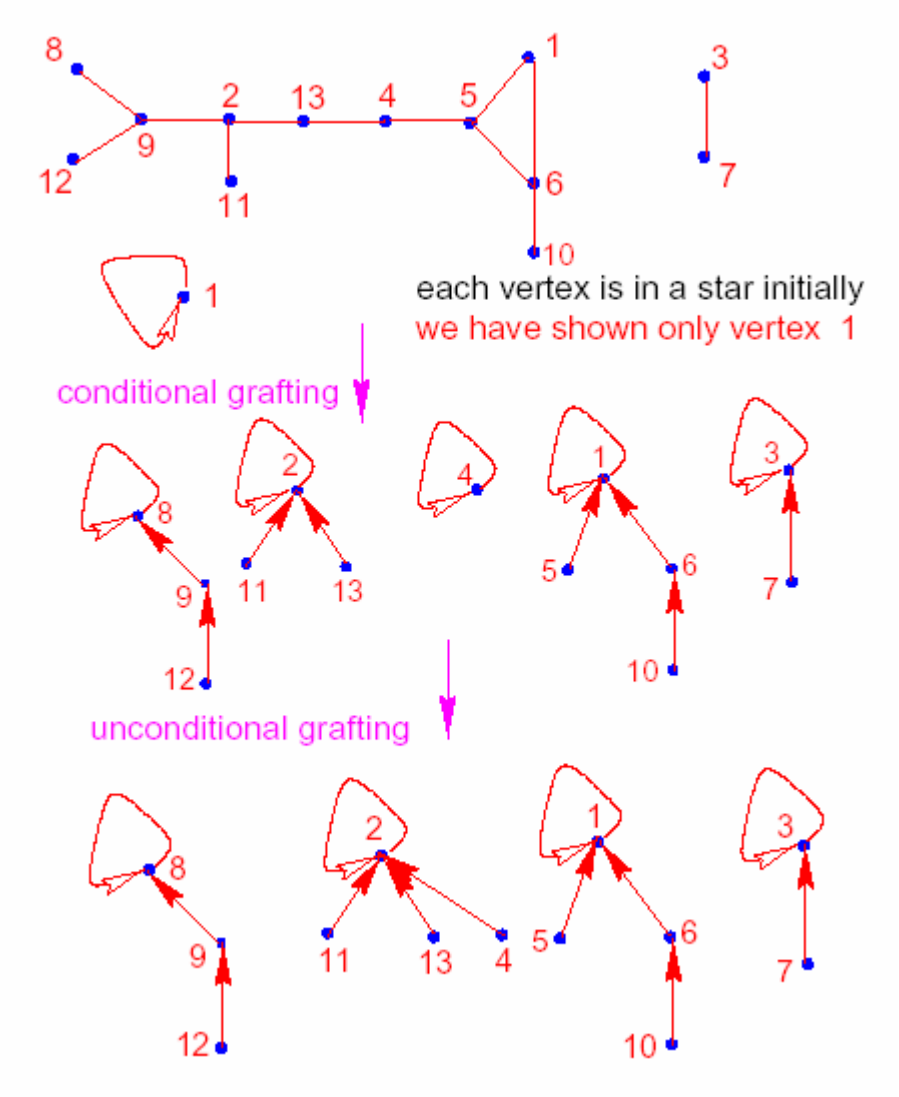
- Die zwei Grafting Operationen und das Pointer Jumping gehen in O(1) Zeit
  - Dazu weisen wir jeder Kante und jedem Knoten einen Prozessor zu
- Der Algorithmus macht O(m+n) work in jeder Iteration
- Es gibt O(log n) Iterationen → Gesamtwork O((m+n)*log n)) und O(log n) Zeit
- Beim uncondtional Grafting versuchen ggf. mehrere Prozessoren den gleichen Baum zu Graften → Das „Arbitrary CRCW PRAM Modell“ wird benötigt

Eine kleine Korrektur

Der Algorithmus ist ein wenig falsch, da er Zyklen im ersten Schritt erzeugen kann. Wir müssen ihn deshalb etwas modifizieren:



- Es kann passieren, dass hier „1“ und „3“ einen Loop bilden, was nicht erlaubt ist (keine eindeutige Wurzel mehr)
- Korrekt wäre, das entweder „1“ oder „3“ einen „Selbstloop“ hat
- Wir machen folgendes um dies zu verhindern:
  - Wir Graften nach dem Conditional Grafting im Ersten Schritt keine Sterne mehr im Unconditional Grafting
  - Wir Graften nun also nur noch Einzelknoten oder selbstloops, aber keine Sterne mit Sternen mehr
  - Diese Problem tritt in den weiteren Iterationsschritten nicht mehr auf

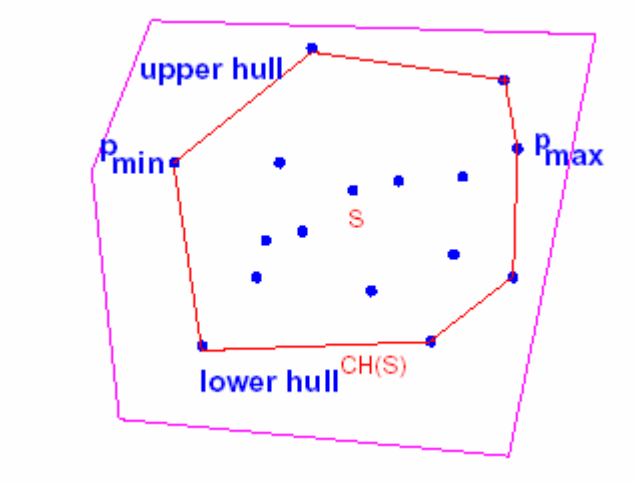


Vorlesung 10.1:

Ein optimaler paralleler Algorithmus zur Lösung des 2D Convex Hull Problems

Problemstellung:

- Eingabe: Eine Menge S von Punkten (p1…pn)
- Ausgabe: Die Konvexe Hülle CH(S) dieser Punkte
    - Die Konvexe Hülle ist das kleinste Polygon das alle Punkte aus S enthält
    - Jeder Knoten aus CH(S) wird Extrempunkt genannt. Die Konvexe Hülle ist also eine sortierte Liste von Extrempunkten



- Die Magentafarbene Hülle ist auch eine konvexe Hülle um die Punkte herum, aber nicht die kleinste ! Die kleinste ist die rote Hülle.

- P_{min} ist der Punkt mit der kleinsten x Koordinate
- P_{max} ist der Punkt mir der größten x Koordinate

- Diese Zwei Punkte teilen die Konvexe Hülle in eine obere Hülle und eine untere.

- Die Linie P_{min} nach P_{max} teilt die Konvexe Hülle in die zwei Teile
- Sei x(p) bzw. y(p) jeweils die x bzw. y Koordinaten des Punktes p
- Sei L eine Linie mit y=ax+b und eine Punkt q = (α,β)
- Wir sagen, q ist unter L, wenn $\beta < a\alpha + b$; q ist dann über L, wenn $\beta > a\alpha + b$

- Bei einem gegeben Set S von n Punkten könnne wir P_{min} und P_{max} in O(n) Zeit finden. Die Punkte über und unter der Linie von P_{min} und P_{max} können wir ebenfalls in O(n) Zeit finden

- Wir können nun die obere Hülle UH(S) berechnen bzw, die untere Hülle LH(S)

- Am Ende kann man die beiden Teilhüllen wieder zur konvexen Hüllen zusammenfügen.

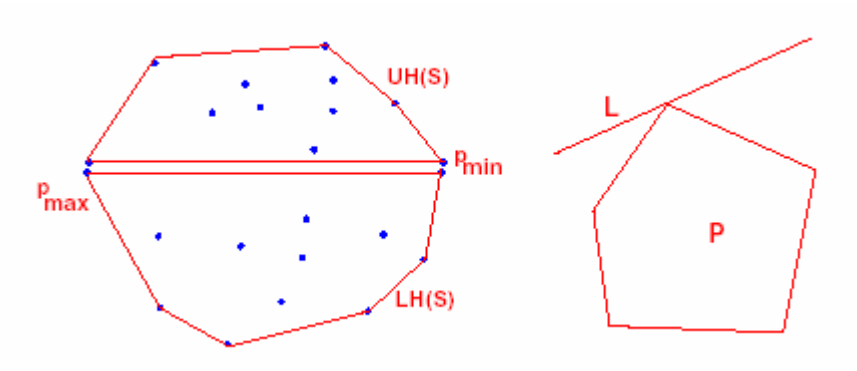
Komplexität beim sequentiellen Algorithmus

Zeit: $\theta(n \log n)$

Man kann diese untere Grenze Zeigen, wenn man zeigen kann, dass das Convex Hull Problem äquivalent zum Sortieren ist

⇨ Wir müssen einen O(n log n) work Algorithmus entwerfen um optimal zu sein

Berechnung der oberen Hülle (UH(S))

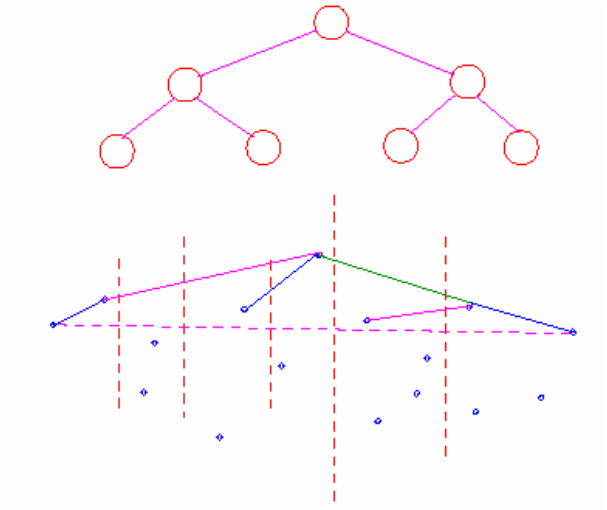


Der Algorithmus für die untere Hülle ist Äquivalent

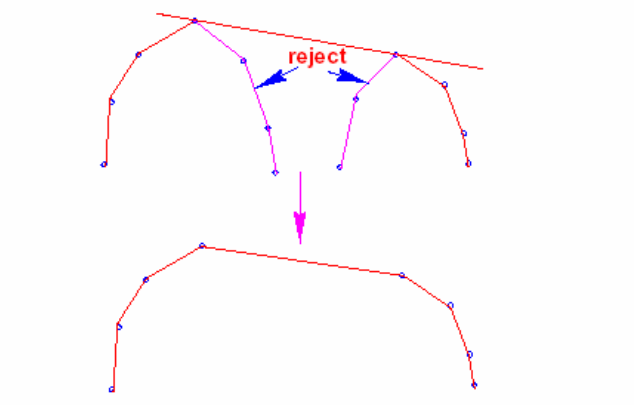
Eine Linie L ist eine Tangent eines Polygons P, wenn alle Knoten von P auf der gleichen Seite von L sind

Divide an Conquer Algorithmus

- Es gibt zwei Phasen: Top-Down und dann Bottom-Up
- Zuerst sortieren wir die Punkte nach den x-Koordinaten
- In der Top-Down Phase unterteilen wir das Set S rekursiv in zwei Teile und berechnen die Konvexe Hülle, wenn das Teilproblem klein genug ist
- In der Bottom-Up Phase mergen wir die Teilhüllen paarweise um am Ende die ober Hülle zu erhalten (Strategie entsprich dem sequentiellen Mergesort)

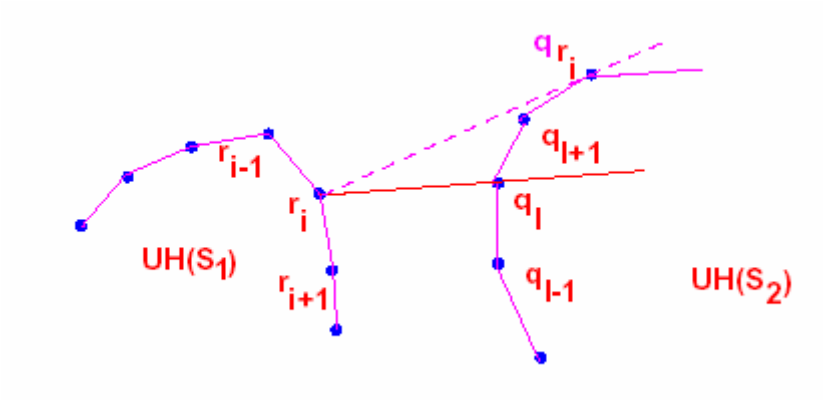


Zusammenfügen zweier oberer Teilhüllen



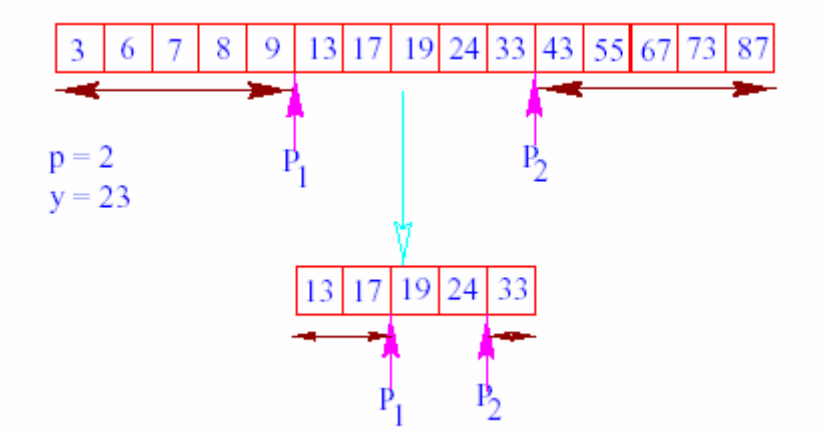
Hauptproblem ist das Berechnen einer gemeinsamen Tangente.

- Um das Problem am Ende in O(n log n) work zu lösen, benötigen wir einen Algorithmus, der das Mergen in O(1) Zeit erledigt.
- Wir finden zunächst eine Tangente von einem willkürlichen Punkt aus $UH(S_1)$ zu $UH(S_2)$



- Wir haben nun die Tangente r_i nach q_{ri}
- Nehmen wir nun eine zweite Linie von r_i zu irgendeinem Punkt q_l in $UH(S_2)$
- Wir versuchen nun herauszufinden, ob q_{ri} oberhalb oder unterhalb der Linie von r_i nach q_l liegt. Dies können wir in O(1) lösen.
- Wenn die Nachbarknoten von q_l (q_{l-1} und q_{l+1}) nicht beide auf der gleichen Seite der Linie r_i nach q_l liegen, dann ist q_{ri} oberhalb von q_l

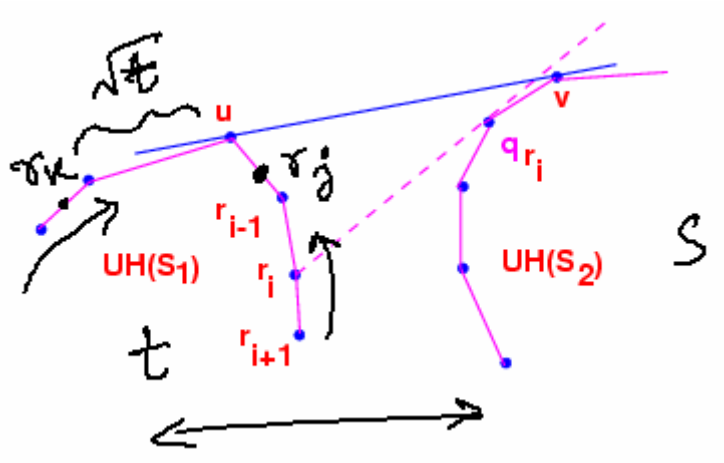
Zum Findem von q_{ri} benutzen wir den „Parallele Suche“ Algorithmus



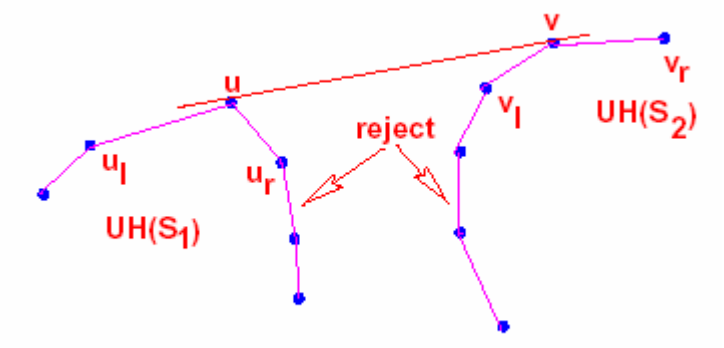
- Wir teilen Das Problem in P+1 Teile, wenn wir P Prozessoren verwenden. Nach jeder Iteration bleibt 1/(p+1)-stel Teil übrig in dem q_{ri} liegt → Rekursiv
- Wir benötigen das CREW PRAM Modell

Vorlesung 10.2:

- Wir haben nun die Linie r_i q_{ri} jetzt können wir feststellen ob u (ein Punkt der Common Tangente) oberhalb oder unterhalb der Linie r_iq_{ri} liegt und zwar in O(1)



- Nehmen wir an $UH(S_1)$ bestehe aus t Punkten
- Nehmen wir an $UH(S_2)$ bestehe aus s Punkten
- Wir teilen nun die t bzw s Punkt in SQRT(t) bzw SQRT(s) intervalle
- Wir machen nun folgende parallele Suche mit SQRT(t)*SQRT(s) Prozessoren
- Für jedes Boundary Element r_i aus $UH(S_1)$ berechnen wir das zugehörige q_{ri}. (Wurde vorhin erklärt) mit SQRT(s) Prozessoren geht das in O(1) Zeit
- Wir haben nun damit einen Bereich bestimmt in dem u liegen muss (zwischen r_j und r_k)
- Der Bereich $r_j r_k$ ist SQRT(t) groß
- Die Berechnung dauert O(1) und benötigt SQRT(s)*SQRT(t) = O(n) Prozessoren
- Auf der der anderen Seite geht das Ganze analog
- Wir erhalten nun also zwei Bereiche sowohl in $UH(S_1)$ als auch in $UH(S_2)$ in denen u und v liegen müssen.
- Nun werden alle möglichen Kombinationen von Linien durch die Elemente beider Seiten in den Bereichen analysiert → Eine davon ist dann letztendlich die „Common Tangente“
- Es gibt also SQRT(s)*SQRT(t)=O(n) Linien deshalb können wir das Ganze in O(1) Zeit berechnen wenn wir O(n) Prozessoren verwenden
- Alle Punkte liegen unterhalb der Common Tangente (eigentlich klar, weil das ist die Definition der Common Tangente)
- Aber woher weiss der Algorithmus nun welches die Common Tangente ist
- Eine Linie ist dann die Common Tangente, wenn die Nachbarn der zu untersuchenden 2 Punkte (linke und rechte Teil Konvex-Hüll) beide unterhalb dieser Linie liegen. O(n) Linien können wir nun wieder in O(1) Zeit berechnen und in O(n) Work



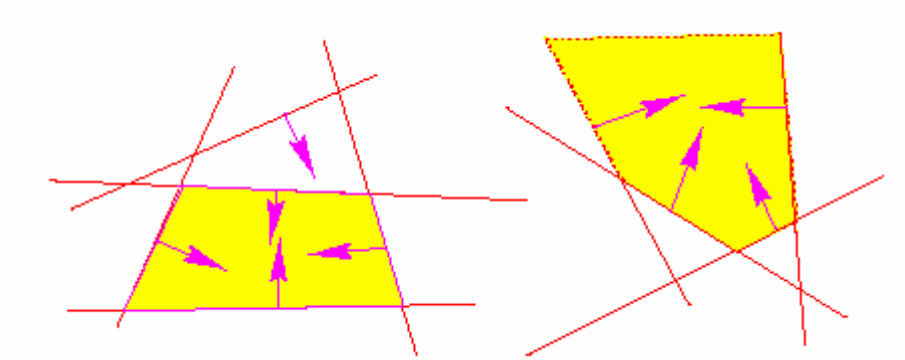
- Die Punkte in der mitte unterhalb der Common Tangente müssen zum vollständigen Verschmelzen noch entfernt werden (reject). Dies kann in O(1) Zeit durchgeführt werden.

Zusammenfassend:

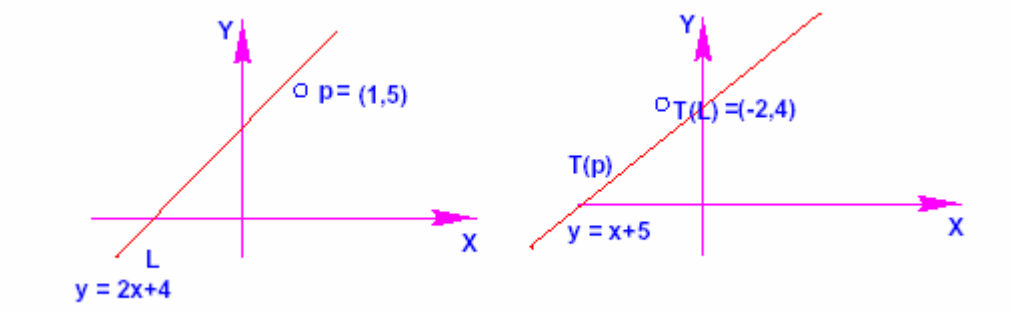
- Das ganze Problem wird mit Hilfe von Divide an Conquer gelöst
- Die Tiefe der Rekursion beträgt O(log n)
- Das Merging der Teilhüllen geht in jedem Rekursionslevel in O(1) bei O(n) Prozessoren und O(n) work
- Also ist die Gesamtzeit O(log n) und die Work O(n log n)
- Benötigt wird das CREW Modell (wegen der parallelen Suche). Es gibt auch ein Modell EREW, aber das ist wohl komliziert.

Intersection von Half-Planes (Schnitt von Halbebenen)

- Sei L eine Linie mit y=ax+b
- Seien $H^+(L)$ und $H^-(L)$ die zwei Halbebenen über und unterhalb von L
- In $H^+(L)$ sind alle Punkte (α,β) sodass gilt β≥aα+b
- In $H^-(L)$ sind alle Punkte (α,β) sodass gilt β≤aα+b
- Intuitiv ist $H^+(L)$ ein Set von Punkten über oder auf L
- Intuitiv ist $H^-(L)$ ein Set von Punkten unter oder auf L



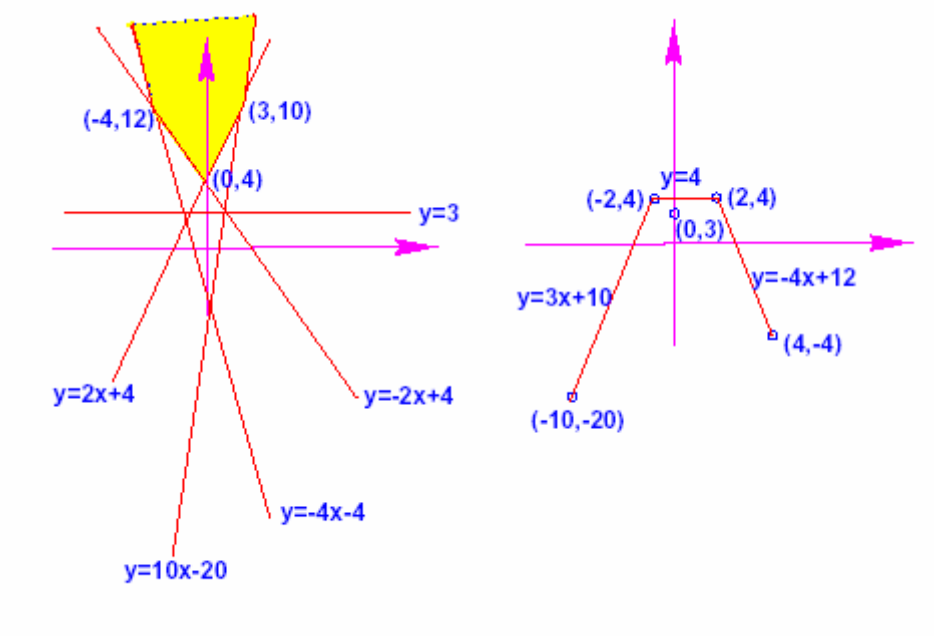
- Schneidet man nun verschiedene Halbebenen erhält man eine konvexe Fläche die aber nicht vollständig geschlossen sein muss (rechtes Bild)
- Wir wollen also nun diese Grenzen bzw die zugehörigen Schnittpunkte berechnen



Bezeichnung: links: Primal Space, rechts: Dual Space

- Sei T eine Funktion die einen Punkt p=(a,b) in eine Linie T(p); definiert durch y=ax+b, transformiert.

- Die Funktion T ist umkehrbar, sodass sie bei Eingabe einer Linie L (y=ax+b) einen Punkt T(L)=(-a,b) erzeugt.
-
- Wichtige Eigenschaft:
  - Ein Punkt p ist unter einer Linie dann und nur dann, wennT(p) unterhalb des Punktes T(L) ist
  - Betrachten wir ein Set von Linie $L_1,\ldots,L_n$ und die Region C ergibt sich durch den Schnitt aller $H^+(L_i)$. Die Region C besteht nun aus allen Punkten die über allen Linien sind.



In der Transformierten Domain (rechts), besteht
$T(C^+) = \{T(p) | p \in C^+\}$ aus allen Linien über den Punkten $T(L_i)$.
(Dies wurde in der Vorlesung irgendwie gar nicht erklärt !?)

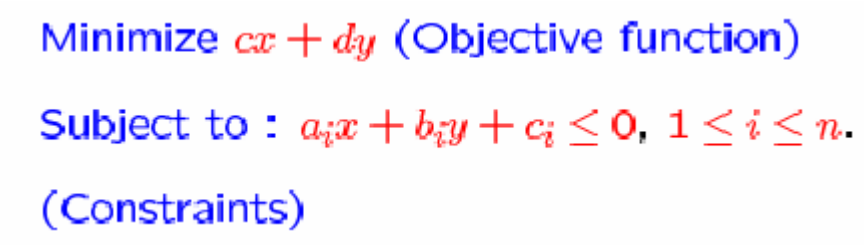
- In unserem DualSpace berechnen wir nun die konvexe Hülle
- Die daraus entstehenden Linien transformieren wir wieder zurück in Punkte und diese Punkte definieren die Schnittfläche.

Zusammenfassend:

- Um den Schnitt zweier Halbebenen zu berechnen, konvertieren wir zunächst die Linien in „Dualpoints“
- Dann berechnen wir die Konvexe Hülle dieser Punkte
- Letztendlich bekommen wir dann die Extrempunkte des Halbebenenschnitts durch konvertieren der Liniensegmente der konvexen Hülle in Punkte
- Die Transformation dauert O(1) Zeit, wenn wir jeder Linie einen Prozessor zuweisen
- Die Konstruktion der konvexen Hülle dauert O(log n) Zeit und O(n log n) work auf einem CREW PRAM

Mit Hilfe des Konvex Hull Algorithmus kann man auch das sogenannte „2 Variable Linear Programming Problem“ lösen.

Das Problem ist folgendermaßen definiert



- Jedes Constraint ist eine Halbeben. Die gesuchte Region ist ein Set von Punkten die alle Constraints erfüllt.
- Die Lösung ist es einen Punkt in dieser Region zu finden, der die Funktion (Objective Function) minimiert.
- Diese wird an einem bestimmten Extrempunkt der Region minimiert
- Wir können nun alle O(n) Extrempunkte der Region mit Hilfe des Halbebenenschnitt Algorithmus finden.
- Damit können wir denjenigen Extrempunkt finden der die Funktion minimiert.
- Der Algorithmus benötigt O(log n) zeit und O(n log n) work auf einem CREW PRAM.

Vorlesung 11.1:

Die Klasse NC und wann ist ei9n Problem parallelisierbar ?

Definition: Ein Poblem ist parallelisierbar, wenn es sehr schnell (in Polylogaritmischer Zeit, also $O(\log^k n)$) auf einer akzeptablem Menge von Prozessoren ($O(n^c)$) gelöst werden kann. (c und k sind Konstanten und n die Eingabegröße)

Die Klasse NC (Nick's Class) zur Ehren von Nick Pipenger:
Die Klasser NC besteht aus allen Problemn die auf einem PRAM in Polylogarithmischer Zeit mit polynomiell vielen Prozessoren, gelöst werden können:

Definition: Die Klasse NC ist also definiert als ein Set alle Sprachen L, die

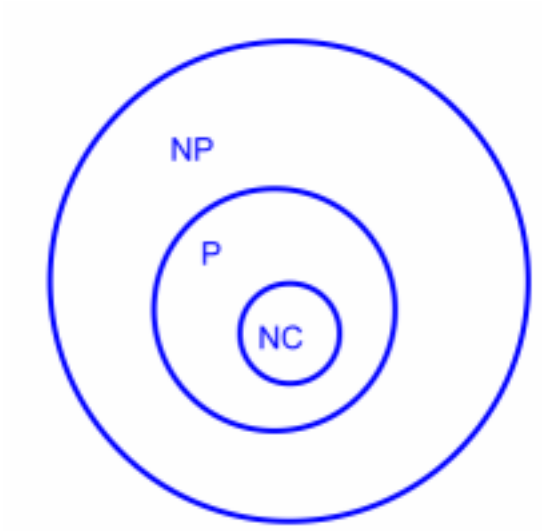
- Alle Eingaben von Größe n haben
- L kann auf einem PRAM in $O(\log^k n)$ Zeit erkannt werden

Die Klassen NC, P und NP

- P war die Klasse aller Probleme, die in polynomieller Zeit auf einer deterministischen Toruingmaschine gelöst werden können

- Offensichtlich ist NC eine Teilmenge von P, es ist aber nicht klar, ob NC auch eine Teilmenge von P ist, also beide Mengen gleich sind. (Die meisten Leute glauben, das sie nicht gleich sind.)

- Wenn die beiden Klassen nicht gleich sind, dann muss es Probleme aus P geben, die sich zwar auf einer RAM gut lösen lassen, aber nicht auf einem PRAM



- Die Klasse der P-Vollständigen Probleme sind besonders gute Kandidaten zur Prüfung ob sie in NC liegen.

- Das Problem ist das gleiche wie bei den Klassen P und NP

- Wenn jemand einen polynomiellen Algorithmus für ein NP-vollständiges Problem findet, dann können wir alle NP-Vollständigen Probleme in polynomieller Zeit lösen.

- Genau gleich ist das zwischen NC und P, wenn man einen polylogarithmischen Algorithmus für ein P-Vollständiges findet, dann können wir alle P-Vollständigen Probleme in polylogarithmischer Zeit lösen.

- NP → in nichtpolynomieller Zeit auf RAM lösbar
- P→ in polynomieller Zeit auf RAM lösbar
- NC→ in polylogarithmischer Zeit auf PRAM lösbar

- Wie können wir nun prüfen, ob ein Problem P-Vollständig ist ?

Definition von NC-Reduzierbar

- Seien L1 und L2 zwei Sprachen

- L1 ist NC-reduzierbar auf L2 wenn,
  - Es einen NC Algorithmus gibt, der einen beliebigen Input u1 für L1 in eine Eingabe u2 für L2 umwandelt, sodass
  - u1 € L1, dann und nur dann wenn u2 € L2

Definition von P-Vollständigkeit

Eine Sprache L ist P-Vollständig, wenn

- L € P
- Jede Sprache in P NC-Reduzierbar ist auf L

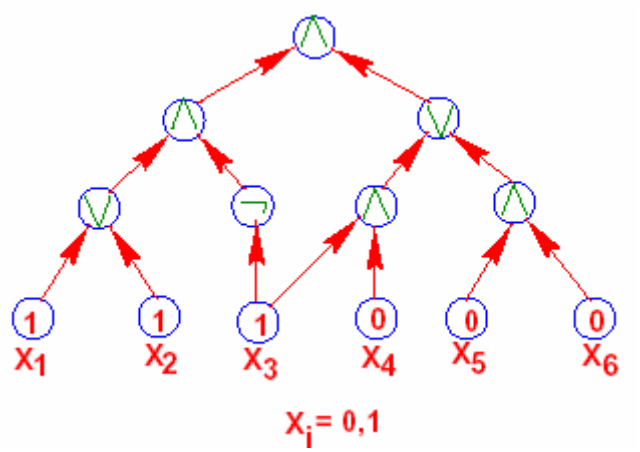
Die nächste Frage ist, ob es überhaupt ein P-Vollständiges Problem gibt.

Das Circuit Value Problem (CVP)

- Wir haben einen Boolschen Schaltkreis be idem „Not“, „Or“ und „And“ erlaubt sind. „Or“ und „And“ haben jeweils zwei Eingabewerte, „Not“ hat nur Einen

- **Problem:** Den Ausgabewert für eine bestimmt Eingabe zu finden

- Unser Boolscher Schaltkreis ist also definiert durch eine Menge von Gattern
  $C =< g_1, g_2, \ldots, g_n >$



- Jedes Gi aus C ist dann
  - Entweder ein Eingabewert, oder
  - $g_i = g_j \vee g_k$, or $g_i = g_j \wedge g_k$, or $g_i = \neg g_j$
  - mit j,k < i

CVP ist P-Vollständig ?

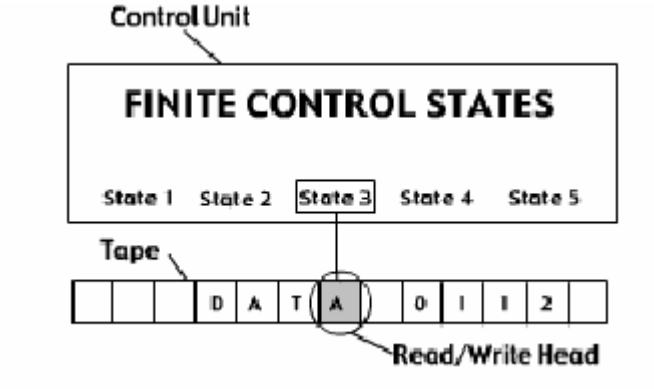
Wir müssen zunächst prüfen ob CVP überhaupt in P liegt

- Die ist eigentlich klar, denn wenn wir ein n-Gate CVP nehmen, dann könne wir die Gatter sequentiell in O(n) Zeit auswerten.

Nun müssen wir prüfen, ob eine beliebige Sprache L € P NC-Reduzierbar ist auf das CVP

Wir stellen dazu nun einen NC Algorithmus zur Verfügung, der eine willkürliche Instanz I_L von L nimmt, diese auf eine Instanz I_{CVP} von CVP abbildet und I_L dann „Yes“ antworten lässt, wenn I_{CVP} eine „1“ als Wert trägt.

Turing Maschine



- Wenn L€P, dann existiert eine „Determinischte Einband Touringmaschine“ M die L in polynomieller Zeit T(n) akzeptiert für eine Eingabemenge n.
- Wir können annehmen, dass die Eingabebits aufeinanderfolgend auf dem Band liegen, der Rest des Bandes ist leer. Der I/O Kopf der Touringmaschine scannt am Anfang Zelle 1 des Bandes und schreibt nach T(n) Zeit das Ergebnis wieder in Zelle 1.
- Q sei die endliche Menge der Zustände Q={q1,…,qs} der Touringmaschine M. q1 ist der Startzustand.
- Sei Σ(a1,…,am) das Band-Alphabet
- Die Überführungsfunktion ist: $\delta : Q \times \Sigma \to Q \times \Sigma \times \{L, R\}$
- Wir werden nun zeigen, wie man einen boolschen Schaltkreis C konstruiert ,so dass der Wert von C „1“ ist, dann und nur dann, wenn M L akzeptiert.
- Zu Beachten ist, das M nur auf T(n) Zellen in T(n) Zeit zugreifen kann. Die Indizes i sind somit zwischen 1 und T(n).
- Die Zeitschritte t sind zwischen 0 und T(n)
- Es gibt nur s Zustände in M, weil alle Zustände q_j zwischen q_1 und q_s liegen.

Hilfsfunktionen:
Der Schaltkreis C wird durch die folgenden boolschen Funktionen definiert:

- H(i,t), sodass
  - H(i,t)=1, nur wenn der Kopf gerade die Zelle i zum Zeitpunkt t scannt
- C(i,j,t), sodass
  - C(i,j,t)=1,nur wenn die Zelle i, das Zeichen a_j zum Zeitpunkt t enthält
- S(k,t), sodass
  - S(k,t)=1, nur wenn der Zustand von M q_k ist zum Zeitpunkt t.

Vorlesung 11.2:

- Unser Schaltkreis besteht aus T(n) Levels

- Für jeden Zeitschritt der Touringmaschine M, emuliert dieser Level das Verhalten der Touringmaschine M.
- Level l enthält die Gates, die die Boolschen-Funktionen H(i,l), C(i,j,l),S(k,l) berechnen für i zwischen 1 und T(n)
- Anmerkung: Wir wissen nicht genau wie M die Sprache L erkennt.Aber in unserem Schaltkreis C halten wir alle mögliche Schritte von M in jedem Level fest.

- Die Anzahl der Gatter in jedem Level i ist T(n)

- Zum Zeitpunkt t=0 gilt folgendes:
    - H(i,0)=1, wenn i=1 ist, 0 sonst
    - C(i,j,0)=1, wenn Zelle i zu Anfang aj enthält
    - S(k,0)=1, wenn k=1, 0 sonst

Nun definieren wir uns die Funktionen in Abhängigkeit aller möglichen Eingaben und I/O Kopfpositionen, also:

Für die Funktion H

$$H(i,t+1) =$$

$$H(i-1,t) \wedge (\vee_{(k,j)\in I_R} C(i-1,j,t) \wedge S(k,t))$$

$$\vee$$

$$H(i+1,t) \wedge (\vee_{(k,j)\in I_L} C(i+1,j,t) \wedge S(k,t))$$

Es gibt zwei Möglichkeiten auf die Position i zu kommen, und zwar von i-1 indem man nach rechts geht, oder von i+1 indem man anschließen nach links geht usw.

Komplexität zum Erstellen:
Zeit: O(1)
Work: $O(T(n)^2)$

Für die Funktion C:

$$C(i,j,t+1) =$$

$$\overline{(H(i,t)} \wedge C(i,j,t))$$

$$\vee$$

$$H(i,t) \wedge (\vee_\kappa (C(i,j',t) \wedge S(k,t)))$$

where,

$$\kappa = \{(k,j') | \delta(q_k, a_{j'}) = (*, a_j, *)\}$$

Komplexität zum Erstellen:
Zeit: O(1)
Work: O(T(n)²)

und für die Funktion S:

$$S(k, t+1) = \bigvee_{(i,(k',j))}(S(k', t) \wedge H(i, t) \wedge C(i, j, t))$$

Komplexität zum Erstellen:
Zeit: O(log n)
Work: O(T(n)²)

Den Endwert unseres Schaltkreise produziert die Funktion:

$$C(1, *, T(n))$$

- Es ist nun klar, dass unser Schaltkreis von einem NC Algorithmus konstruiert werden kann, weil T(n) polynomiell in n ist.

- Die Ausgabe unseres Schaltkreises ist nur dann „1“, wenn M die Sprache L erkennt.

- Weil L eine beliebige Sprache aus P ist, folgt daraus, dass CVP P-Vollständig ist

Implikationen:

- TRANSITIVITÄT: Wenn L1 NC reduzierbar auf L2 ist und L2 NC reduzierbar auf L3 ist, dann ist L1 NC reduzierbar auf L3
- Wenn L ein P-Vollständiges Problem ist und L in NC liegt, dann gilt P=NC. (offenes Problem)

Prüfung anderer P-Vollständiger Probleme:

- Sei eine Sprache L P-Vollständig. Wenn nun L NC reduzierbar auf eine andere Sprache L’ aus P ist. Dann ist auch L’ P-Vollständig.
- (genau das benötigt man um auch andere Probleme auf P-Volständigkeit zu prüfen)
    - Wenn wir also zB das CVP NC reduzieren können auf ein anderes Problem in P, dann ist auch dieses P-Vollständig.

Liste P-Vollständiger Probleme:
(nicht polylogarithmisch lösbar, also nicht in NC)

„Ordered depth-first search“
- Gegeben sei ein gerichteter Graph G=(V,E) in Form einer Adjazenzmatrix und
- gegeben seihen weiterhin drei Knoten s,u,v.
- Wir wollen nun wissen ob u vor v besucht wurde, wenn wir bei s starten.

„Maximum Flow“
- Gegeben sei ein Netzwerk mit gewichteten Kanten und zwei herausragenden Knoten (nämlich der Quelle und der Senke)

- Problem: Festzustellen ob der „Maximum Flow“ ungerade ist
- Applet unter: http://riot.ieor.berkeley.edu/riot/Applications/graal-flow/graal.html

„Linear inequalities“

- Gegeben sei eine n x m Matrix A und ein n-dimensionaler Vektor b.Alle Einträge sind ganze Zahlen.
- Problem: Gibt es einen rationalen Vektor x, sodass gilt $Ax \leq b$

Fragenkatalog:

1. Was ist die Working-Komplexität?
2. Wann ist ein paralleler Algorithmus Effizient?
3. Wann ist ein Algorithmus Work-Optimal bzw Work-Tine-Optimal?
4. Was ist ein Parallelrechner?
5. Welchen Sinn macht Parallelrechnen ? Wo liegen die Grenzen?
6. Was sind die Vorteile/Probleme von Parallelrechnern?
7. Erkläre das PRAM Modell?
8. Welche Parallelcomputer-Modelle gibt es?
9. Was sind die wichtigsten Design-Aspekte für Network SIMD Modelle? Erkläre diese 3?
10. Welche Netzwerkmodelle gibt es? (Mesh, Hypercube)
11. Welche verschiedenen PRAM Modelle gibt es bzgl. des Speicherzugriffs?
12. Sind diese untereinander kompatibel?
13. Wann ist ein solches Modell schwächer als ein Anderes? (Erst Zeit, dann Work)
14. Algorithmus „Präfixsumme“: Was berechnet dieser? Wie funktioniert er? Komplexität?
15. Algorithmus „Präfixsumme“: Es gibt eine rekursive Variante, wie funktioniert diese?
16. Algorithmus „Pointer Jumping“: Was berechnet dieser? Für was kann man den Algorithmus gut brauchen? Welche Komplexität hat er?
17. Was versteht man unter der „Partitioning und Merging Strategie“? Komplexität?
18. 3 Algorithmen zur „Berechnung des Maximums“: Welche verschiedenen Lösungsansätze gibt es? Welche Strategien verfolgen diese? Was ist Accelerated Cascading? Komplexitäten?
19. 3 Algorithmen zum „List Ranking“: Welche verschiedenen Algorithmen zum List Ranking gibt es? Welche Strategie verfolgen diese jeweils? Kompexität?
20. Was ist die „Euler Tour Technik“? Für was benötigt man diese? Was ist ein Eulerkreis? Was ist das besondere im Parallelen?
21. Wir haben für die Euler-Tour-Technik auch eine Umkehrung kennengelernt, „Rooting a Tree“ – Wie funktioniert diese Technik? Was ist Tree Contraction? Wie funktioniert die Rake-Operation bzw. der Rake Algorithmus?
22. Algorithmus zur „Auswertung eines Ausdruckes“, dargestellt als Baum: Wie funktioniert das?
23. Algorithmus „Parallele Suche in einem sortierten Array“: Welche Techniken werden hier angewandt? Welche Mergingverfahren haben wir dabei kennengelernt?
24. Wie kann man effizient sortieren? (vorheriger Mergingalgorithmus)
25. Parallele Graphen-Algorithmen: Was kann man verbundene Komponenten erkennen? Was ist ein Pseudo-Forrest?
26. Welche Eingabemöglichkeiten gibt es für Graphen? (Adjanzenzmatrix, Kantenliste)
27. Was ist ein dichter Graph, was ist ein lichter Graph? Wann nimmt man welchen Algorithmus?
28. Was ist Grafting? Wie kann man einen Stern erkennen? Was ist unconditonal Grafting, was ist conditional Grafting?

29. Was ist das 2D Konvexe-Hülle Problem? Kennen Sie einen optimalen parallelen Algorithmus zur Lösung? Was versteht man unter der Upper-Hull bzw. Lower-Hull? Wieso teilt man die Hülle? Wie erhält man die Teilhüllen?
30. Was ist die Common Tangente?
31. Was ist Halbebenen-Schnitt? Wie funktioniert der Algorithmus? Was versteht man unter „Primal Space“ und „Dual Space“?
32. Was ist das „2 Variable Linear Programming Problem“? Wie kann man dieses mit Hilfe des „Convex-Hull“ Algorithmus lösen?
33. Wann ist ein Problem parallelisierbar? (Definition)
34. Was ist die Klasse NC, wie ist sie definiert?
35. Parallelen zum P/NP Problem?
36. Was heisst NC Reduzierbarkeit?
37. Was ist P-Vollständigkeit?
38. Was versteht man unter dem „Circuit Value Problem“ CVP? Warum ist dieses P-Vollständig und wie weißt man das nach? (Touring Maschine)
39. Welche anderen P-Vollständigen Probleme wurden angesprochen?