

Notizen zu Algorithmen II

Jens Ochsenmeier, Carolin Beer

2. März 2020

INHALTSVERZEICHNIS

1 Algorithm Engineering	3
2 Fortgeschrittene Datenstrukturen	4
2.1 Adressierbare Prioritätslisten	4
2.2 Pairing Heaps	5
2.3 Fibonacci-Heaps	7
3 Kürzeste Wege	9
3.1 Allgemeine Definitionen	9
3.2 Dijkstras Algorithmus	9
3.3 Monotone ganzzahlige Prioritätslisten	10
3.4 All-Pairs Shortest Paths	11
3.5 Distanz zu Zielknoten	13
4 Anwendungen von DFS	14
4.1 Starke Zusammenhangskomponenten	14
5 Maximale Flüsse und Matchings	17
5.1 Definitionen	17
5.2 Cuts	17
5.3 Pfade augmentieren	18
5.4 Dinic-Algorithmus	20
5.5 Matchings	21
5.6 Preflow-Push	21
6 Randomisierte Algorithmen	23
6.1 Ergebnisüberprüfung von Sortieren	23
6.2 Hashing	23
7 Externe Algorithmen	24
7.1 Externe Stapel	24
7.2 Externes Sortieren	24
7.3 Mehrwegemischen, externes Mischen	24
7.4 Externe Prioritätslisten	25
7.5 Externe minimale Spannbäume	26
8 Approximationsalgorithmen	27
8.1 Job Scheduling	27
8.2 Turing-Reduzierbarkeit	27
8.3 Allgemeines Travelling Salesman-Problem	28
8.4 Entscheidungsproblem — Hamiltonkreis	28
8.5 Euler-Touren/-Kreise	28
8.6 Metrisches TSP	28
8.7 Pseudopolynomielle Laufzeit	29
8.8 Knapsack	29
8.9 (Voll) Polynomielle Approximationsschemata	30

9 Fixed-Parameter-Algorithmen	32
9.1 Fixed Parameter Tractable (FPT)	32
9.2 Vertex Cover/ Kantenabdeckung	32
10 Parallele Algorithmen	33
10.1 Einleitung	33
10.2 Nachrichtengekoppelte Parallelrechner	34
11 Stringology	37
11.1 Strings sortieren	37
11.2 Pattern Matching	38
11.3 Datenkompression	43
12 Burrows-Wheeler-Transformation	45
12.1 Konstruktion	45
12.2 Beobachtungen	45
12.3 Rücktransformation	46
12.4 Was bringt die BWT?	48
12.5 Kompression	48
12.6 Suche in der Burrows-Wheeler-Transformation	49
12.7 Wavelet Trees	50
13 A Compact Bit-Sliced Signature Index (COBS)	51
14 Geometrische Algorithmen	52
14.1 Grundlegende Definitionen	52
14.2 Streckenschnitte	52
14.3 Konvexe Hülle	54
14.4 Kleinste einschließende Kugel	55
14.5 Range Search	56
15 Online-Algorithmen	57
15.1 Übersicht	57
15.2 Job-Scheduling	59
15.3 Skiausleihe	60
15.4 Speicherverwaltung	60
Appendices	62
A Mathematische Grundlagen	63
A.1 Amortisierte Analyse	63
A.2 Komplexitätsklassen	63
A.3 Hyperwürfel	63
B Algorithm basics	64

1: ALGORITHM ENGINEERING

Algorithm Engineering ist ein interdisziplinäres Forschungsfeld welches Lücken zwischen Theorie und Praxis schließt. Es umfasst Theorie, Praxis und Anwendung. Theoretische Algorithmik ist ein Unterfeld des Algorithm Engineering. Die Aufgaben lassen sich unterteilen in:

- **Modellierung**
- **Design:** Einfachheit und Wiederverwendbarkeit
- **Analysis:** Berücksichtigung konstanter Faktoren und durchschnittlicher Laufzeiten
- **Implementierung**
- **Experimente:** Reproduzierbarkeit wichtig. Software Engineering wird miteinbezogen. Können bei der Analyse helfen.

Im Gegensatz zu rein theoretischer Algorithmik herrscht eine größere Methodevielfalt und größerer Bezug zu Anwendungen.

2: FORTGESCHRITTENE DATENSTRUKTUREN

Wir werden uns in diesem Kapitel mit Prioritätslisten beschäftigen. Es gibt noch viele weitere fortgeschrittene Datenstrukturen, z.B.

- monotone ganzzahlige Prioritätslisten (später im Kapitel “kürzeste Wege”)
- perfektes Hashing
- Suchbäume mit fortgeschrittenen Operationen
- externe Prioritätslisten (später im Kapitel “Externe Algorithmen”)
- Geometrische Datenstrukturen (siehe Kapitel “Geometrische Algorithmen”)

2.1 Adressierbare Prioritätslisten

Eine **adressierbare Prioritätsliste** muss folgende Funktionen implementieren:

BUILD ($\{e_1, \dots, e_n\}$)	$M := \{e_1, \dots, e_n\}$
SIZE	return $ M $
INSERT (e)	$M := M \cup \{e\}$
MIN	return $\min M$
DELETEMIN	$e := \min; M := M \setminus \{e\}; \text{return } e$
REMOVE ($h : \text{Handle}$)	$e := h; M := M \setminus \{e\}; \text{return } e \quad // \text{Handle ist ein Pointer}$
DECREASEKEY ($h : \text{Handle}, k : \text{Key}$)	$\text{key}(h) := k$
MERGE (M')	$M := M \cup M'$

Adressierbare Prioritätslisten haben viele Anwendungen, beispielsweise im *Dijkstra-Algorithmus* für kürzeste Wege oder in der Graphpartitionierung. Allgemein lassen sich adressierbare Prioritätslisten gut bei Greedy-Algorithmen verwenden, bei denen sich die Prioritäten (begrenzt) ändern.

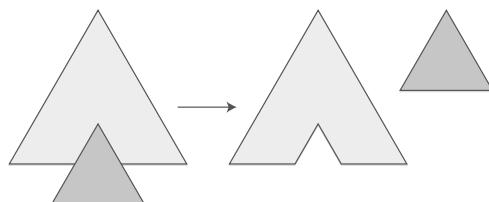
Datenstruktur

Als grundlegende Datenstruktur wird ein Wald heap-geordneter Bäume verwendet. Hier wird also der Binary Heap verallgemeinert

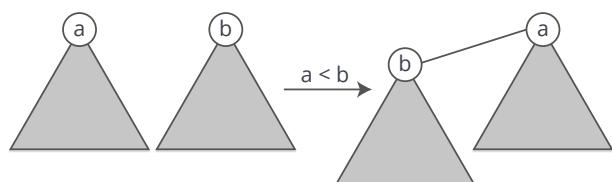
- Baum \rightarrow Wald
- zwei Kindknoten \rightarrow beliebig viele Kindknoten

Wir verwenden folgende grundlegende Operationen zur Bearbeitung solcher Wälder:

- **cut**: Teilbaum ausschneiden und als neuen Baum speichern



- **link**: Baum 2 mit größerem Wurzelknoten als Baum 1 an Baum 1 als Kindknoten des Wurzelknotens anhängen



- **union**: $\text{union}(a, b) = \text{link}(\min(a, b), \max(a, b))$

Dijkstras Algorithmus

Dijkstras Algorithmus kann die Distanz zwischen einem Startknoten s und jedem anderen Knoten des Graphen berechnen.

```

DIJKSTRA( $s$  : Node,  $T$  : Tree)
// Initialisieren: Distanz zu jedem Knoten ist  $\infty$ , zu Startknoten 0.
 $d = \langle \infty, \dots, \infty \rangle$ 
 $d[s] = 0$ 
// Startknoten zu PQ hinzufügen.
 $Q.insert(s)$ 
while  $Q \neq \emptyset$  do
     $u := Q.deleteMin$ 
    for all  $(u, n) \in E$  do
        if  $n \notin Q$  then  $Q.insert(n)$ 
        if  $d[n] > d[u] + c(u, n)$  then  $d[n] = d[u] + c(u, n)$ 

```

2.2 Pairing Heaps

Pairing Heaps müssen folgende Funktionen implementieren:

```

INSERTITEM( $h$  : Handle)
    newTree( $h$ )
NEWTREE( $h$  : Handle)
    forest := forest  $\cup \{h\}$ 
    if  $*h < \min$  then minPtr :=  $h$ 
DECREASEKEY( $h$  : Handle,  $k$  : Key)
    key( $h$ ) :=  $k$ 
    if  $h$  not a root then cut( $h$ ) else updateMinPtr( $h$ )
DELETEMIN() : Handle
     $m := \text{minPtr}$ 
    forest := forest  $\setminus \{m\}$ 
    foreach child  $h$  of  $m$  do newTree( $h$ )
    pairwiseRootUnion()
    updateMinPtr()
    return  $m$ 
PAIRWISEROOTUNION()
    // see picture
MERGE( $o$  : AdressablePQ)
    if  $*\text{minPtr} > *(o.\text{minPtr})$  then minPtr :=  $o.\text{minPtr}$ 
    forest := forest  $\cup o.\text{forest}$ 
     $o.\text{forest} := \emptyset$ 

```

Einige Funktionalitäten lassen sich gut veranschaulichen:

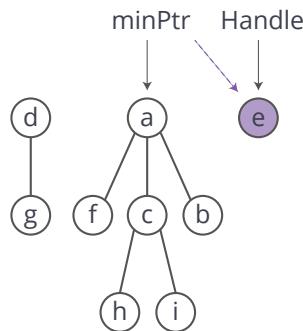


Abbildung 2.1. newTree: Element *e* wird hinzugefügt (ggf. auch ein ganzer Baum) und – falls nötig – der minPtr angepasst.

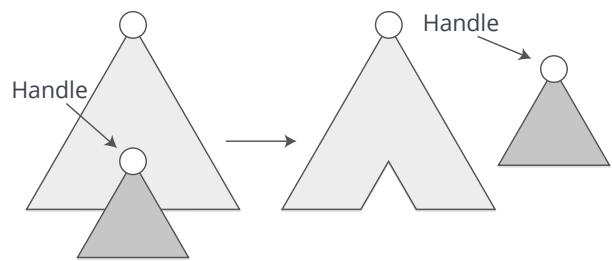


Abbildung 2.2. decreaseKey: Durch Herabsetzen des Keys wird eventuell die Heap-Eigenschaft verletzt, deswegen wird der Teilbaum ausgeschnitten und als neuer Baum hinzugefügt.

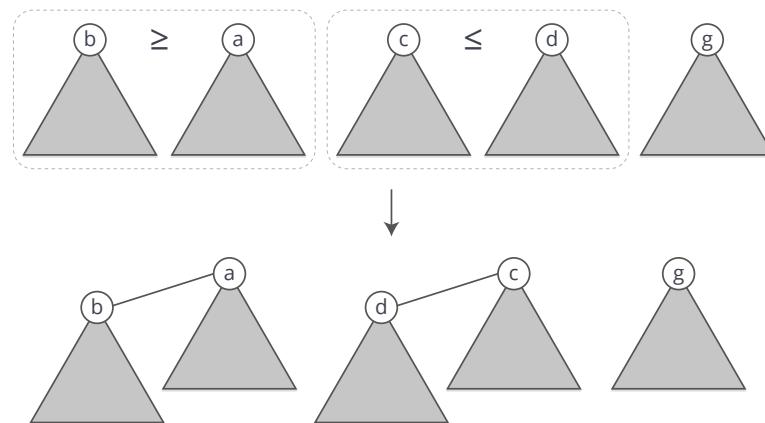


Abbildung 2.3. pairwiseRootUnion fügt jeweils zwei Bäume des Waldes zu einem größeren Baum zusammen, indem die beiden Wurzeln miteinander verglichen werden und eine der beiden Wurzeln Kindknoten der anderen wird.

Repräsentation

Meistens speichert man Pairing Heaps als doppelt verkettete Liste der Wurzeln. Die Baum-Items können beispielsweise folgendermaßen gespeichert werden:

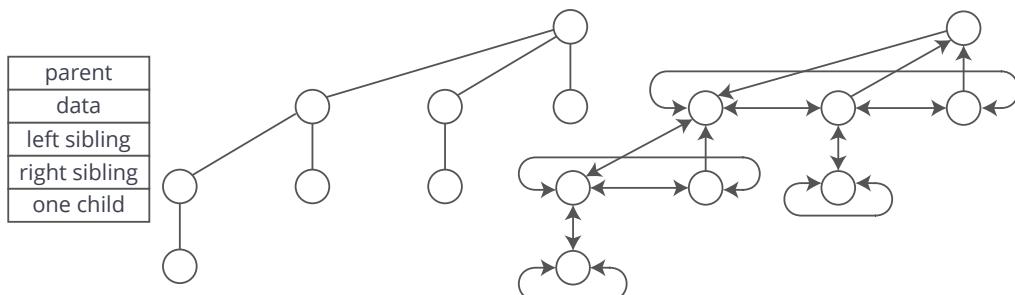


Abbildung 2.4. Speichern der Baum-Items. Man kann hier noch Speicherplatz einsparen, indem man *left sibling* und *parent* zusammenfasst. Allerdings muss man dann alle Geschwisterknoten traversieren, damit man zum Elternknoten kommen kann, da nur der linkeste Kindknoten den Elternknoten speichert.

Analyse

- `insert` und `merge` gehen in $O(1)$.
- `deleteMin` und `remove` gehen jeweils in $O(\log n)$ amortisiert.
- `decreaseKey` ist schwieriger zu analysieren, geht aber amortisiert in $O(\log \log n) \leq T \leq O(\log n)$ und ist in der Praxis sehr schnell.

Wir werden als nächstes *Fibonacci-Heaps* verwenden, um noch mehr Leistung rauszukitzeln.

2.3 Fibonacci-Heaps

Mithilfe von Fibonacci-Heaps erhalten wir eine amortisierte Komplexität von $O(\log n)$ für `deleteMin` und `remove` und $O(1)$ für alle anderen Operationen.

Fibonacci-Heaps speichern ein paar Zusatzinformationen pro Knoten ab, wodurch neue Hilfsfunktionen kreiert werden können, die diese Beschleunigung ermöglichen:

- **Rank** eines Knotens: Anzahl direkter Kinder
- **Mark**: Knoten, die ein Kind verloren haben, werden markiert
- **Vereinigung nach Rank**: Union nur für gleichrangige Wurzeln
- **Kaskadierende Schnitte**: Knoten, die beide markiert sind (also ein Kind verloren haben), werden geschnitten

Repräsentation

parent
data, rank, mark
left sibling
right sibling
one child

Die Repräsentation ist analog zu der von Pairing Heaps, die Wurzeln werden wieder als doppelt verkettete Liste gespeichert und die Baum-Items als Parameterliste:

Funktionalität

insert und merge werden wie gehabt implementiert. decreaseKey verwendet die neue cascadingCut-Methode und deleteMin Union-by-Rank. Wir beschleunigen Union-by-Rank, indem wir ein Feld pro Rank bereitstellen und Knoten in diese Felder eintragen. Sollte das passende Feld bereits belegt sein, so wird der neue Knoten mit dem Knoten, der sich im Feld befindet mittels union-by-rank gelinkt und entsprechend verschoben.

```

DELETEMIN(): Handle
  m := minPtr
  forest := forest \ {m}
  foreach child h of m do newTree(h)
  while  $\exists a, b \in \text{forest} : \text{rank}(a) \equiv \text{rank}(b)$  do
    union(a, b)
  updateMinPtr()
  return m
DECREASEKEY(h : Handle, k : Key)
  key(h) := k
  cascadingCut(h)
CASCADINGCUTH : Handle
  assert h is not a root
  p := parent(h)
  unmark(h)
  cut(h)
  if p is marked then
    cascadingCut(p)
  else mark(p)

```

Eine amortisierte Analyse von deleteMin ergibt $\Omega(\log n)$ für vergleichsbasiertes deleteMin.

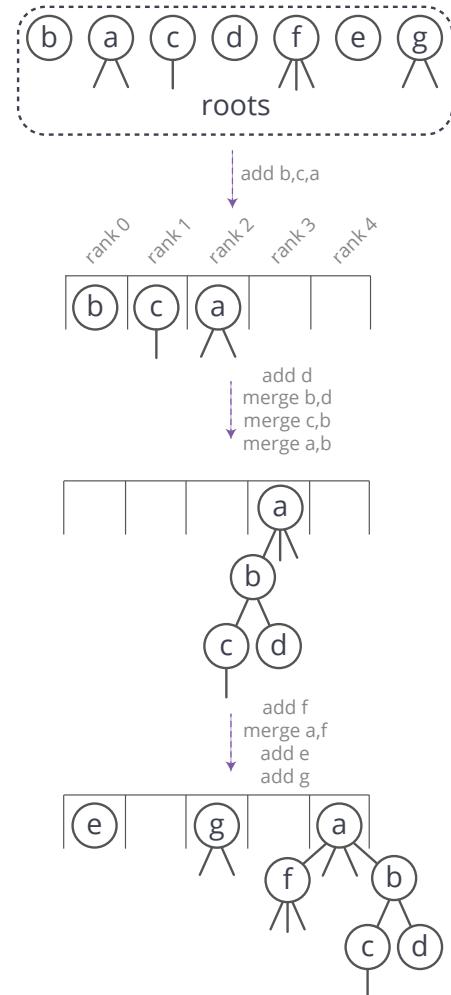


Abbildung 2.5. Fast Union-by-Rank.

3: KÜRZESTE WEGE

Wir betrachten einen Graph $G = (V, E)$ mit Kantengewicht $c : E \rightarrow \mathbb{R}$ und Anfangsknoten $s \in V$.

Gesucht ist die Länge $\mu(v)$ des **kürzesten Pfades** von s nach v für alle $v \in V$, wobei

$$\mu(v) := \min \{c(p) : p \text{ ist Pfad von } s \text{ nach } v\}$$

und

$$c((e_1, \dots, e_k)) := \sum_{i=1}^k c(e_i).$$

Oft suchen wir auch eine “geeignete” Repräsentation des kürzesten Pfades.

3.1 Allgemeine Definitionen

Wir benutzen im Allgemeinen zwei Knotenarrays:

- $d[v]$ = aktuelle (= vorläufige) Distanz von s nach v .
Invariante: $d[v] \geq \mu(v)$.
- $\text{parent}[v]$ = Vorgänger von v auf (vorläufigem) kürzesten Pfad von s nach v .
Invariante: Dieser Pfad bezeugt $d[v]$.

Initial ist

- $d[s] = 0$, $\text{parent}[s] = s$ und
- $d[v] = \infty$, $\text{parent}[v] = \perp$.

Kern ist das *Relaxieren* der Kanten $(u, v) \in E$:

```
if  $d[u] + c(u, v) < d[v]$  then // z.B. wenn  $d[v] \equiv \infty$ 
   $d[v] := d[u] + c(u, v)$ 
   $\text{parent}[v] := u$ 
```

Die oben genannten Invarianten werden dadurch nicht verletzt. $d[v]$ kann sich also problemlos mehrmals ändern.

3.2 Dijkstras Algorithmus

Dijkstras Algorithmus ist der wohl einfachste Algorithmus, um dieses Problem zu lösen. In Pseudocode:

```
// d und parent initialisieren
// alle Knoten als ungescannt setzen
while ∃ non-scanned node u with  $d[u] < \infty$  do
  u := non-scanned node v with minimal  $d[v]$ 
  relax all edges (u, v) out of u
  set u scanned
```

Ist $v \in V$ von s aus erreichbar, so wird v irgendwann gescannt. Wird v gescannt, so ist $\mu(v) = d[v]$.

Am Ende definiert d die optimalen Entfernungen und parent die zugehörigen Wege. Dieser Algorithmus wurde bereits in Algorithmen I ausführlich diskutiert.

Analyse

Es ist

$$T_{\text{Dijkstra}} = O(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))).$$

Nutzen wir Fibonacci-Heaps, so kriegen wir

$$T_{\text{DijkstraFib}} = O(m + n \log n).$$

Für Binary-Heaps ergibt sich die Laufzeit zu

$$T_{\text{BinaryHeap}} = O((m + n) \log n).$$

Hierbei sind die konstanten Faktoren jedoch kleiner, als bei Fibonacci-Heaps.

Da die Analyse zeigt, dass `decreaseKey` amortisiert lediglich $O(n \log \frac{m}{n})$ mal aufgerufen wird, ergibt sich die Laufzeit von Binary-Heaps für dichte Graphen ($m > n \log n \log \log n$) jedoch als linear in m .

3.3 Monotone ganzzahlige Prioritätslisten

Wir beobachten, dass Dijkstras Algorithmus die Prioritätsliste *monoton* benutzt – `insert` und `decreaseKey` benutzen nämlich Distanzen der Form $d[u] + c(e)$. Die Werte nehmen also ständig zu.

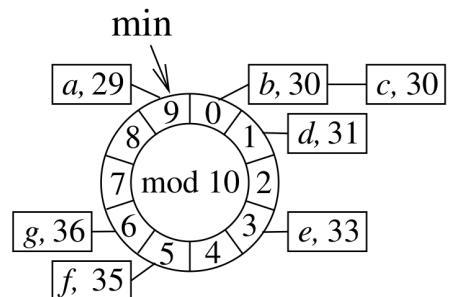
Sind alle Kantengewichte $\in [0, C]$, so gilt

$$\forall v \in V : d[v] \leq (n - 1)C.$$

Ist insbesondere \min der letzte Wert, der aus Q entfernt wurde, so sind in Q *immer* nur Knoten mit Distanzen im Intervall $[\min, \min + C]$.

Bucket-Queue

Eine **Bucket-Queue** ist ein zyklisches Array B von $C + 1$ doppelt verketteten Listen. Knoten der Distanz $d[v]$ werden in $B[d[v] \bmod (C + 1)]$ gespeichert.



Content=
 $\langle (a, 29), (b, 30), (c, 30), (d, 31), (e, 33), (f, 35), (g, 36) \rangle$

Abbildung 3.1. Bucket-Queue mit $C = 9$.

Folgende Operationen werden auf einer Bucket Queue implementiert:

- **Initialisierung:** $C + 1$ leere Listen werden angelegt, $\min = 0$.
- **insert (v):** fügt v in $B[d(v) \bmod (C + 1)]$ ein
 $\Rightarrow O(1)$
- **decreaseKey (v):** schiebt v von seiner Liste nach $B[d(v) \bmod (C + 1)]$
 $\Rightarrow O(1)$

- **deleteMin**: fängt bei $B[\min \bmod (C + 1)]$ an; falls leer, $\min := \min + 1, \cup$
 $\Rightarrow O(nC)$

Mit Bucket-Queues kriegen wir also den Dijkstra-Algorithmus auf $O(m + \text{maxPathLength})$ gedrückt.

Radix-Heaps

Radix-Heaps sind eine Variante von Bucket Queues, die Buckets von -1 bis K für $K = 1 + \lfloor \log C \rfloor$ benutzt. Wie vorhin schon betrachtet sei d^* die zuletzt aus Q entfernte Distanz und somit

$$\forall v \in Q : d[v] \in [d^*, \dots, d^* + C].$$

Wir betrachten die *binäre Repräsentation* der möglichen Distanzen in Q . Wir speichern v in Bucket $B[i]$, falls sich $d[v]$ und \min zuerst an der i -ten Stelle unterscheiden.

Das definiert die **Most Significant Digit** (kurz MSD) — das ist die Position der höchstwertigen Ziffer in der Binärdarstellung von a und b , an der sich die beiden unterscheiden. $\text{msd}(a, b)$ kann mit Maschinenbefehlen sehr schnell berechnet werden.

Wir nutzen folgende *Radix-Heap-Invariante*:

$$v \text{ ist gespeichert in Bucket } B[i], \text{ wo } i = \min \{\text{msd}(d^*, d[v]), K\}$$

Ausnahmen: Speichere in $B[K]$ falls $i > K$, speichere in $B[-1]$ falls sie sich nicht unterscheiden.

Wir können nun `deleteMin` folgendermaßen implementieren:

```
DELETEMIN() : Element
if  $B[-1] = \emptyset$  then
     $i := \min \{j \in 0 \dots K : B[j] \neq \emptyset\}$ 
    move  $\min B[i]$  to  $B[-1]$  and to  $d^*$ 
    foreach  $e \in B[i]$  do // exactly here the invariant is violated!
        move  $e$  to  $B[\min \{\text{msd}(d^*, d[e]), K\}]$ 
return  $B[-1].\text{popFront}()$ 
```

Die amortisierte `deleteMin`-Laufzeit ist $O(K)$, insgesamt lässt sich also die Laufzeit des Dijkstra-Algorithmus hierdurch auf

$$T_{\text{DijkstraRadix}} = O(m + n \log C)$$

drücken.

3.4 All-Pairs Shortest Paths

Herausforderung in diesem Abschnitt ist es, nicht die Abstände zu einem festgelegten Startknoten zu berechnen, sondern zwischen allen Knotenpaaren $(u, v) \in V^2$ für $G = (V, E)$. Zusätzlich erlauben wir negative Kantenkosten, allerdings keine negativen Kreise.

Wir werden zwei verschiedene Lösungen erhalten:

1. n mal den *Bellman-Ford-Algorithmus* ausführen
 $\Rightarrow O(n^2 m)$
2. *Knotenpotentiale* verwenden
 $\Rightarrow O(nm + n^2 \log n)$

Bellman-Ford-Algorithmus

Der **Bellman-Ford-Algorithmus** wurde bereits in Algorithmen I behandelt, deswegen hier nur eine kurze Wiederholung. Wie Dijkstras Algorithmus findet er den kürzesten Pfad zwischen einem festgelegten Startknoten und allen anderen Knoten des Graphen, unterstützt aber auch negative Kantengewichte.

1. Distanzen initialisieren: $d[s] = 0$, $d[v] = \infty$ für alle anderen Knoten
2. Von s ausgehend alle Knoten des Graphen durchgehen und pro Knoten v die ausgehenden Kanten betrachten.
 - Eintragen, ob v von s in $< \infty$ erreicht werden kann.
 - Ist $d[v] + c(v, u) < d[u]$ für einen Nachbar von v ? Wenn ja, $d[u]$ aktualisieren.

Der zweite Schritt wird maximal $|V| - 1$ mal wiederholt. Sollte sich schon davor bei einem Durchgang nichts mehr ändern, so kann man aufhören.

Die Laufzeit des Bellman-Ford-Algorithmus ist $O(nm)$, nutzt man ihn als Basis für All-Pairs Shortest Paths kriegt man also $O(n \cdot nm) = O(n^2m)$.

Knotenpotentiale

Jeder Knoten erhält ein Potential $\text{pot}(v)$. Mit diesen Knotenpotentialen lassen sich die **reduzierten Kosten** $\bar{c}(e)$ für eine Kante $e = (u, v) \in E$ als

$$\bar{c}(e) = \text{pot}(u) + c(e) - \text{pot}(v)$$

definieren.

Ist p ein Pfad von u nach v mit Kosten $c(p)$, dann ist

$$\bar{c}(p) = \text{pot}(u) + c(p) - \text{pot}(v).$$

Ist p' ein anderer u - v -Pfad, dann gilt

$$c(p) \leq c(p') \Leftrightarrow \bar{c}(p) \leq \bar{c}(p').$$

Wir berechnen die gewünschten Informationen nun so:

1. Wir fügen einen *Hilfsknoten* s zu G hinzu.
2. Wir fügen (s, v) für alle $v \in V \setminus \{s\}$ mit Kosten 0 hinzu.
3. Berechne $\mu(v) \forall v \in V$ als die kürzesten Pfade von s aus mit Bellman-Ford.
4. Definiere $\text{pot}(v) := \mu(v)$ für alle $v \in V$.

Die reduzierten Kosten sind jetzt alle nicht-negativ, also können wir Dijkstra benutzen und ggf. s wieder entfernen.

5. Für eine beliebige Kante $(u, v) \in E$ gilt

$$\mu(u) + c(e) \geq \mu(v) \text{ und deshalb } \bar{c}(e) = \mu(u) + c(e) - \mu(v) \geq 0.$$

```

neuen Knoten  $s$  und alle Kanten  $s, v$  hinzufügen //  $O(n)$ 
pot :=  $\mu := \text{BELLMANFORDSSSP}(s, c)$  //  $O(nm)$ 
foreach  $x \in V$  do
   $\bar{\mu}(x, \cdot) := \text{DIJKSTRASSSP}(x, \bar{c})$  //  $O(n(m + n \log n))$ 

// zurück zur ursprünglichen Kostenfunktion
foreach  $e = (v, w) \in V^2$  do //  $O(n^2)$ 
   $\mu(v, w) := \bar{\mu}(v, w) + \text{pot}(w) - \text{pot}(v)$ 

```

Die Gesamlaufzeit wird von n Dijkstra-Schritten dominiert und beträgt daher $O(n(m + n \log n))$.

3.5 Distanz zu Zielknoten

Wir haben bisher zwei Fälle diskutiert:

1. Abstände aller Knoten zu einem Startknoten s
2. Abstände zwischen allen Knotenpaaren $\{u, v\} \in V^2$

Als nächstes schauen wir uns an, wie man den kürzesten Pfad zwischen einem Startknoten s und einem Zielknoten t ermittelt.

Trick 0 – Dijkstra abbrechen

Am einfachsten ist es, einfach Dijkstra abzubrechen, wenn t aus Q entfernt wird. Das spart “im Schnitt” die Hälfte des Scans.

Bidirektionale Suche

Idee ist hier, abwechselnd von s und t aus zu suchen. Von s aus sucht man auf $G = (V, E)$, von t aus auf dem zugehörigen Rückwärtsgraphen $G^r = (V, E^r)$.

Die vorläufige kürzeste Distanz wird in jedem Schritt gespeichert:

$$d[s, t] = \min \{d[s, t], d_{\text{forward}}[u] + d_{\text{backward}}[u]\}$$

Abgebrochen wird, wenn die Suche einen Knoten scannt, der in die andere Richtung bereits gescannt wurde.

A^* -Suche

Idee der A^* -Suche ist es, “in die Richtung” des Ziels zu suchen. Dazu benötigen wir eine Funktion $f(v)$, die für alle $v \in V$ die eigentliche Funktion $\mu(v, t)$ schätzen kann.

Anschließend können wir $\text{pot}(v) = f(v)$ setzen und $\bar{c}(u, v) = c(u, v) + f(v) - f(u)$.

$f(v)$ muss diese Eigenschaften haben:

- **Konsistenz:** $c(e) + f(v) \geq f(u)$ ($\forall e = (u, v)$).
Die reduzierten Kosten dürfen also nicht negativ sein.
- $f(v) \leq \mu(v, t)$ ($\forall v \in V$).
Dann ist $f(t) = 0$ und wir können aufhören wenn t aus Q entfernt wird.

Ist p ein beliebiger Pfad von s nach t , so ist $d[t] \leq c(p)$ (alle Kanten auf p seien relaxiert). Wie finden wir jetzt aber so eine Funktion $f(v)$?

Wir benötigen eine Heuristik für $f(v)$.

- Betrachtet man eine Strecke im Straßennetzwerk, so kann $f(v)$ beispielsweise der euklidische Abstand $\|v - t\|_2$ sein. Damit erhält man eine deutliche, aber keine überragende Beschleunigung.
- **Landmarks** sind deutlich geeigneter, benötigen allerdings Vorberechnung.
Man wähle eine Landmarkmenge L . Berechne und speichere $\mu(v, l)$ für alle $l \in L, v \in V$.
Während einer Query suche man jetzt ein Landmark $l \in L$ “hinter” dem Ziel und benutze die untere Schranke

$$f_l(v) = \mu(v, l) - \mu(t, l)$$

Vorteile sind, dass Landmarks konzeptuell einfach sind, eine erhebliche Beschleunigung bringen (um den Faktor 20) und mit anderen Techniken kombinierbar sind.

Allerdings ist die Landmarkauswahl schwierig und der Platzverbrauch sehr groß (besonders für große V).

4: ANWENDUNGEN VON DFS

4.1 Starke Zusammenhangskomponenten

Zusammenhangskomponenten in einem ungerichteten Graph sind Teilgraphen, in denen es zwischen je zwei beliebigen Knoten einen Pfad gibt.

In gerichteten Graphen sind **starke Zusammenhangskomponenten** (SCCs) Teilgraphen $G \subseteq H$, in denen ebenfalls gilt: für jedes Knotenpaar $u, v \in G$ gibt es einen $u-v$ -Pfad und einen $v-u$ -Pfad.

Insbesondere werden starke Zusammenhangskomponenten durch Zyklen erzeugt (dann kann man einfach im Kreis laufen von einem Knoten zum anderen). Das bedeutet im Umkehrschluss, dass der **Schrumpfgraph** — das ist der Graph, den man erhält, indem man jede starke Zusammenhangskomponente als einen Knoten zusammenfasst — zyklusfrei ist.

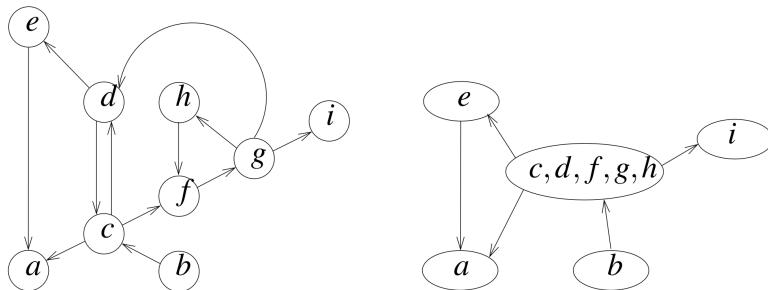


Abbildung 4.1. Gerichteter Graph G und zugehöriger Schrumpfgraph G_S .

Tiefensuchsschema

Um später SCCs ermitteln zu können konstruieren wir ein Tiefensuchsschema:

```

unmark all nodes
init()
foreach s ∈ V do
    if s is not marked then
        mark s
        root(s)
        DFS(s, s)

DFS(u, v : Node)
    foreach (v, w) ∈ E do
        if w is marked then
            traverseNonTreeEdge(v, w)
        else
            traverseTreeEdge(v, w)
            mark w
            DFS(v, w)
            backtrack(u, v)

```

Dieses Tiefensuchsschema kann auf unterschiedliche Graphtraversierungsprobleme angepasst werden.

Wir verwenden nun zwei Arrays zum Zwischenspeichern unserer Resultate:

- oNodes speichert die bereits besuchten Knoten,
- oReps speichert die Repräsentanten der einzelnen SCCs.

Beim Durchlaufen des Graphen werden die Knoten mit `dfsNum` inkrementell durchnummieriert.

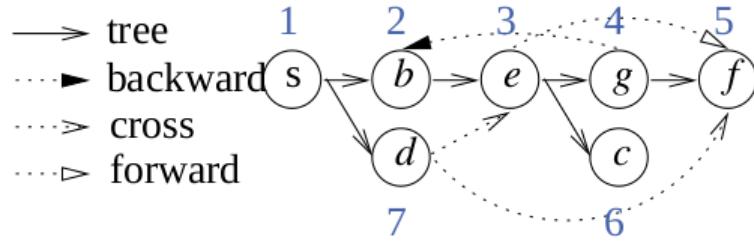


Abbildung 4.2. Illustration of numbered edges using DFS..

Außerdem gibt es drei Invarianten, die wir im Folgenden nicht verletzen dürfen:

1. Kanten von abgeschlossenen Knoten (d.h. alle Kanten exploriert) gehen zu abgeschlossenen Knoten
2. Offene Komponenten S_1, \dots, S_k bilden einen Pfad im Schrumpfgraph von c , G_C^s .
3. Jede Komponente besitzt einen Repräsentanten, welcher die offenen Komponenten bezüglich ihrer `dfsNum` partitioniert.

Für das Finden von SCCs brauchen wir folgende Implementierungen für die rot gekennzeichneten Prozeduren:

• **root(s):**

```
oReps.push(s)
oNodes.push(s)
```

Hierdurch wird eine neue offene Komponente gebildet und s als besucht gekennzeichnet.

• **traverseTreeEdge(v, w):**

```
oReps.push(w)
oNodes.push(w)
```

Hier wird $\{w\}$ als neue offene Komponente angelegt.

• **traverseNonTreeEdge(v, w):**

```
if w ∈ oNodes then
  while w.dfsNum < oReps.top.dfsNum do oReps.pop
```

Ist $w \notin oNodes$ ist w abgeschlossen und die Kante somit uninteressant. Ist w allerdings in $oNodes$, so werden die auf dem Kreis befindlichen SCCs kollabiert.

• **backtrack(u, v):**

```
if v ≡ oReps.top then
  oReps.pop
repeat
  w := oNodes.pop
  component[w] := v
until w = v
```

Damit haben wir alles was wir brauchen, um die Suche nach SCCs durchführen zu können. Wir kriegen sie sogar in $O(m + n)$, also in Linearzeit, hin!

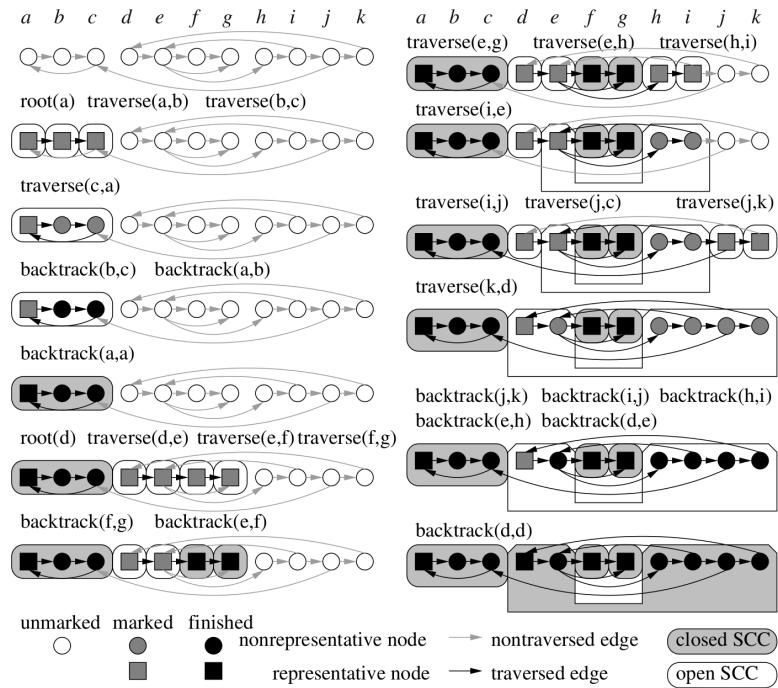


Abbildung 4.3. Kompletter Durchlauf des Algorithmus.

2-zusammenhängende ungerichtete Komponenten

Ein solcher Graph würde mit gerichteten Kanten zu einer zusammenhängenden Komponente werden.

5: MAXIMALE FLÜSSE UND MATCHINGS

5.1 Definitionen

Netzwerk

Ein **Netzwerk** ist ein gerichteter und gewichteter Graph mit zwei speziellen Knoten — einer **Quelle** und einer **Senke**. Unterschied zwischen s, t und dem Rest der Knoten ist, dass s keine eingehenden und t keine ausgehenden Kanten hat. Das Gewicht c_e der Kante e nennen wir die **Kapazität** der Kante. Diese muss nicht-negativ sein.

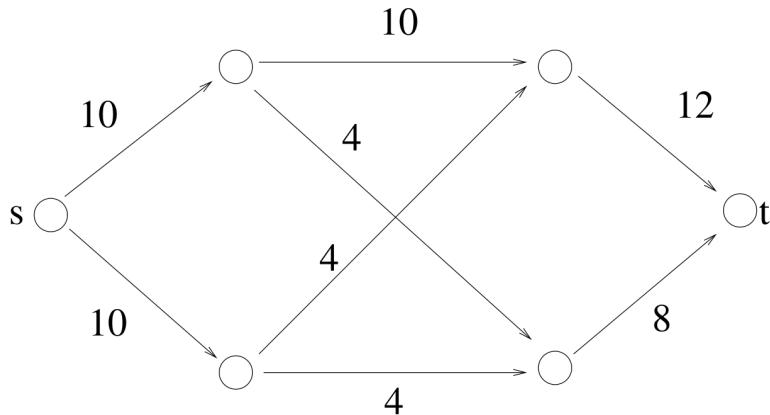


Abbildung 5.1. Beispiel für ein Netzwerk mit Quelle s (source) und Senke t (sink).

Fluss

Ein **Fluss** in einem Netzwerk ist eine Funktion f_e auf den Kanten des Netzwerks. Ein Fluss hat folgende Eigenschaften:

- $0 \leq f_e \leq c_e (\forall e \in E)$
- $\forall v \in V \setminus \{s, t\} : \text{Summe eingehender Flüsse} = \text{Summe ausgehender Flüsse}$

Wir definieren außerdem den **Wert** eines Flusses f als

$$\text{val}(f) = \Sigma \text{ von } s \text{ ausgehender Fluss} = \Sigma \text{ zu } t \text{ eingehender Fluss}$$

Ziel ist es in der Regel, einen Fluss in einem festgelegten Netzwerk mit *maximalem Wert* zu finden.

5.2 Cuts

Definition 5.2.1 (Cut). Ein $s-t$ -**Cut** ist eine Partitionierung eines Graphen G in zwei Mengen S und T , sodass $s \in S$ und $t \in T$.

Die **Kapazität** eines Cuts ist

$$\sum \{c_{(u,v)} : u \in S \wedge v \in T\},$$

also die Summe der Kapazitäten aller Kanten, die durch den Cut “durchgeschnitten” werden.

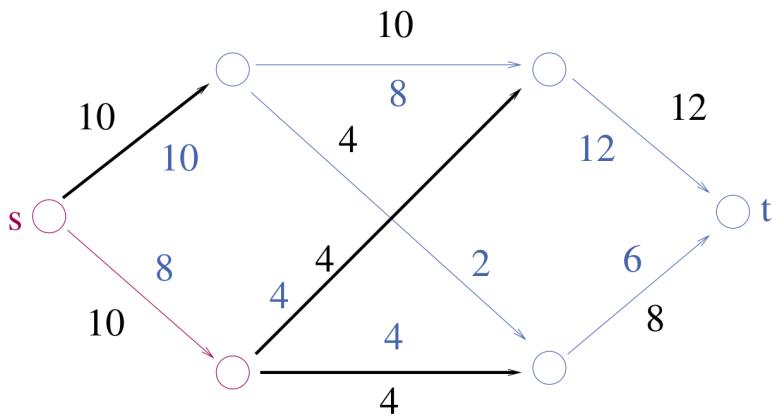


Abbildung 5.2. Die blauen Zahlen stellen einen möglichen Fluss in obigem Netzwerk dar. Die schwarzen Kanten sind ein möglicher Cut. Der Wert dieses Fluxes ist 18.

Es gilt folgender extrem praktischer Satz:

Satz 5.2.2 (Fluss-Cut-Dualität). Der Wert eines maximalen $s-t$ -Flusses ist die minimale Kapazität eines $s-t$ -Cuts.

Wie können wir nun diesen Satz nutzen, um einen maximalen Fluss zu finden? Eine Möglichkeit ist *Lineare Programmierung*, aber es gibt bessere Lösungen.

5.3 Pfade augmentieren

Idee ist folgende:

1. Wähle einen $s-t$ -Pfad, der noch Kapazität übrig hat.
2. Sättige die Pfadkante mit der kleinsten Restkapazität.
3. Korrigiere die Kapazitäten aller anderen Kanten mithilfe des *Residualgraphen*. Gehe wieder zu Schritt (1) und wiederhole den Prozess.

Residualgraph

Ist ein Netzwerk $G = (V, E, c)$ mit Fluss f gegeben, so erhalten wir den **Residualgraphen** $G_f = (V, E_f, c^f)$. Dabei gilt für jedes $e \in E$:

$$\begin{cases} e \in E_f \text{ mit } c_e^f = c_e - f(e) & \text{falls } f(e) < c(e) \\ e^{\text{rev}} \in E_f \text{ mit } c_{e^{\text{rev}}}^f = f(e) & \text{falls } f(e) > 0 \end{cases}$$

Dabei ist für $e = (u, v) \in E$ die Kante $e^{\text{rev}} = (v, u)$.

Die Kanten in die “normale” Richtung sind also diejenigen, die noch Restkapazität haben; die neue Kapazität ist die Restkapazität.

Die Kanten in umgekehrte Richtung sind die Kanten, wo der Fluss > 0 ist (das heißt insbesondere, dass auch beide Fälle eintreten können). Das Gewicht dieser Kanten entspricht dem Fluss der Kanten in normale Richtung.

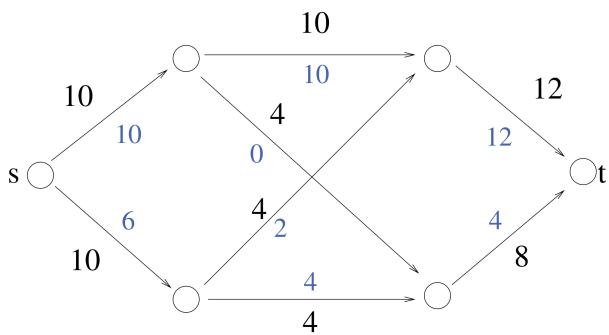


Abbildung 5.3. Nochmal der Fluss von oben.

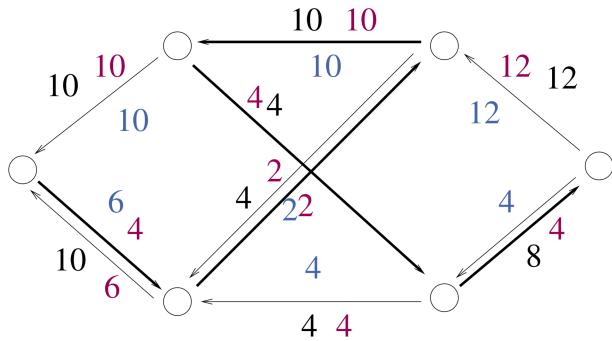


Abbildung 5.4. Der zu nebenstehendem Fluss gehörende Residualgraph. Schwarz die Kapazität, Blau der Fluss, Rot die residuale Kapazität.

Wir suchen jetzt nach einem $s-t$ -Pfad p , sodass jede Kante residuale Kapazität $c_e^f \neq 0$ hat:

```
 $\Delta f := \min_{e \in p} c_e^f$ 
foreach  $(u, v) \in p$  do
  if  $(u, v) \in E$  then  $f_{(u,v)} += \Delta f$  // forward edge
  else  $f_{(v,u)} -= \Delta f$  // backward edge
```

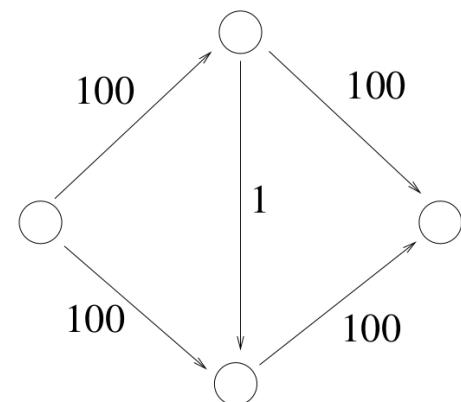
Wir können nun den **Ford-Fulkerson-Algorithmus** implementieren:

```
FFMAXFLOW( $G = (V, E)$ ,  $s, t, c : E \rightarrow \mathbb{N}$ ) :  $E \rightarrow \mathbb{N}$ 
 $f := 0$ 
while  $\exists$  path  $p = (s, \dots, t)$  in  $G_f$  do
  augment  $f$  along  $p$ 
return  $f$ 
```

Die Zeitkomplexität ist (bei terminierendem FF) $O(m \cdot \text{val}(f))$, da bei jedem Durchgang der Fluss um mindestens 1 erhöht wird und jedes Mal DFS $O(m)$ ausgeführt werden muss. Es gibt Netzwerke, in denen der Ford-Fulkerson-Algorithmus tatsächlich die längstmögliche Laufzeit braucht, obwohl die Lösung sehr trivial ist.

The maximum flow equals the minimum cut capacity.

Problem – Blocking Flows



In einigen Fällen konvergiert der Algorithmus nicht. Grund dafür sind sogenannte **Blocking Flows**.

f_b ist ein *blocking flow* in H , falls für jeden $s-t$ -Pfad p gilt:

$$\exists e \in p : f_b(e) = c(e).$$

Das bedeutet, dass jeder $s-t$ -Pfad eine vollständig ausgelastete Kante beinhaltet.

Abbildung 5.5. In diesem Netzwerk könnte der Ford-Fulkerson-Algorithmus stets einen Pfad über die mittlere Kante augmentieren und würde deswegen 200 Schritte brauchen.

5.4 Dinic-Algorithmus

Der **Dinic-Algorithmus** sieht so aus:

```
DINITZMAXFLOW( $G = (V, E)$ ,  $s, t, c : E \rightarrow \mathbb{N}$ ) :  $E \rightarrow \mathbb{N}$ 
 $f := 0$ 
while  $\exists$  path  $p = (s, \dots, t)$  in  $G_f$  do // an augmented edge exists
     $d = G_f.\text{reverseBFS}(t) : V \rightarrow \mathbb{N}$  // find shortest augmenting paths (distance) to source
     $L_f := (V, \{(u, v) \in E_f : d(v) = d(u) - 1\})$  // build layer graph
    find blocking flow  $f_b$  in  $L_f$  // with DFS from source on layer graph
    augment  $f += f_b$ 
return  $f$ 
```

Die rot gekennzeichneten Funktionen/Objekte müssen wir noch klären.

Der Ablauf lässt sich folgendermaßen zusammenfassen:

1. Berechne Distanz-Labels (Abstand zur Senke) für alle Knoten des Graphen.
(Rückwärtsgerichtete Breitensuche von t aus)
2. Stelle auf Basis der Distanz-Labels den Layer-Graph auf.
3. Suche auf dem Layer-Graph einen blockierenden Fluss zwischen s und t .
4. Führe den gefundenen blockierenden Fluss auf dem Residualgraph aus und aktualisiere diesen entsprechend.
5. Konstruiere den aktualisierten Layer-Graphen und gehe zu Schritt 3.
6. Breche ab, sobald kein weiterer Fluss im Layer-Graph existiert.

Abstandsfunktion

Die Abstandsfunktion d gibt den Abstand eines Knotens zur Senke t an.

Layer-Graph

Den **Layer-Graph** eines Netzwerks erhält man, indem man alle Kanten aus dem Residualgraphen entfernt, die nicht von einer Schicht in die vorherige führen. Es werden also alle Kanten

- innerhalb einer Schicht und
- zwischen Schicht i und $i + k$ ($k \in \mathbb{N}$)

entfernt. Formal ist das

$$L_f = (V, \{(u, v) \in E_f : d(v) = d(u) - 1\}).$$

Laufzeitanalyse

Blocking Flows ergibt sich mittels

$$n \cdot \#\text{breakthroughs} + \#\text{extends} + \#\text{retreats}$$

eine Laufzeit von $O(m + nm) = O(nm)$. Spezialfälle mit niedrigerer Komplexität sind Unit Flows (mit Einheitskantengewichten) und Dynamic Trees. Letztere verringern die Komplexität von Breakthroughs, erhöhen sie aber für Retreats und Extends. Dadurch sind in der Praxis in der Regel schlechtere Ergebnisse verbunden.

Die Laufzeit des Dinic-Algorithmus ist damit bei maximal n Durchgängen $O(mn^2)$ (stark polynomiell). Bei Verwendung von Dynamic Trees käme man auf $O(mn \cdot \log n)$.

Unit Trees lassen sich in $O((m + n)\sqrt{m})$ berechnen. Schränkt man Unit Trees weiter ein, und fordert dass alle Kanten v die Gleichung $\min(\text{indegree}(v), \text{outdegree}(v)) = 1$ erfüllen, so sprechen wir von einem Unit Network. Dieses ist durch $O(m + n\sqrt{n})$ beschränkt.

5.5 Matchings

Ein **Matching** in einem ungerichteten Graphen $G = (V, E)$ ist eine Teilmenge der Kanten, sodass es keine Kanten gibt, die einen gemeinsamen Knoten berühren.

Ein Matching ist *maximal*, wenn es keine Kante gibt, die man aus E zum Matching hinzufügen könnte, ohne die Matching-Anforderung zu verletzen.

Ein Matching hat *maximale Kardinalität*, wenn es kein Matching auf G gibt, das mehr Knoten abdeckt.

Wir werden Matchings zuerst einmal verwenden, um Flüsse zu berechnen. Dazu werden Source und Sink in den Graphen eingefügt und der Dinic-Algorithmus verwendet. Da dies ein Unit Flow Network ist, funktioniert dies in $O((n + m)\sqrt{n})$.

Eine Verallgemeinerung für gewichtete Graphen ist nur sehr begrenzt möglich, indem eine Kontraktion von SCCs mit hoher Kapazität vorgenommen wird.

5.6 Preflow-Push

Nachteil von pfadaugmentierenden Algorithmen ist, dass man Pfade mit hoher Kapazität aufgrund von späteren Knoten mit geringer Kapazität sehr häufig durchgehen muss.

Preflow-Push-Algorithmen lösen dieses Problem.

Definition 5.6.1 (Preflow). Ein **Preflow** ist ein Fluss f , bei dem die Summe der eingehenden Flüsse für einen Knoten höher sein darf als die Summe der ausgehenden Flüsse. Die Differenz dieser Summen nennen wir den **Excess** des Knotens:

$$\text{excess}(v) := \underbrace{\sum_{(u,v) \in E} f_{(u,v)}}_{\text{inflow}} - \underbrace{\sum_{(v,w) \in E} f_{(v,w)}}_{\text{outflow}} \geq 0.$$

Wir nennen einen Knoten $v \in V \setminus \{s, t\}$ **aktiv**, falls $\text{excess}(v) > 0$.

Wir definieren nun die push-Funktion:

```
PUSH(e = (v, w), δ)
  assert δ > 0
  assert excess(v) ≥ δ    // node is active
  assert residual capacity of e ≥ δ    // edge can handle flow increase
  excess(v) -= δ
  excess(w) += δ
  if e is reverse edge then f(reverse(e)) -= δ
  else f(e) += δ
```

Wir nennen einen Push

- **saturierend**, falls $\delta = c_e^f$,
- **nicht-saturierend**, falls $\delta < c_e^f$.

Level-Funktion

Idee ist nun, von s aus Richtung t zu pushen. Dazu benötigen wir eine Approximation $d(v)$ der BFS-Distanz von v zu t in G_f .

Wir können nun den Preflow-Push-Algorithmus implementieren:

```
GENERIC_PREFLOW_PUSH( $G = (V, E), f$ )
  forall  $e = (s, v) \in E$  do  $\text{push}(e, c(e))$ 
   $d(s) := n, d(v) = 0$  for all other nodes
  while  $\exists v \in V \setminus \{s, t\} : \text{excess}(v) > 0$  do
    if  $\exists e = (v, w) \in E_f : d(w) < d(v)$  then // eligible (descending) edge
      choose some  $\delta \leq \min\{\text{excess}(v), c_e^f\}$ 
       $\text{push}(e, \delta)$  // no new steep edges
    else  $d(v)++$  // relabel; no new steep edges
```

Wichtig ist hierbei die Invariante $\forall (v, w) \in E : d(v) \leq d(w) + 1$. Ist dies nicht erfüllt, so spricht man von einer steilen (Abwärts-)Kante.

Dieser generische Algorithmus hat eine Laufzeit von $O(n^2m)$, die Laufzeit wird dominiert durch nonsaturating pushes.

FIFO Preflow Push

Bei einem Knoten werden so lange wie möglich saturierende Pushes ausgeführt bis er deaktiviert oder relabelt wird. Dann gehe zum nächsten Knoten. Die Laufzeit beträgt hierfür $O(n^3)$. Ist daher vor allem für Graphen interessant, bei denen $m \gg n$.

Highest Level Preflow Push

Wählt man immer die aktiven Knoten aus, die $d(v)$ maximieren, so kann man die Laufzeit auf $O(n^2\sqrt{m})$ drücken. Die Implementierung erfolgt mit Bucket PQ.

Modified FIFO Selection Rule

Hierbei wird ein aktiverter ausgewählter Knoten solange weiter ausgewählt, bis er inaktiv wird.

Heuristiken

Mit heuristischen Mitteln lässt sich der practical case weiter reduzieren.

- **Aggressive local relabeling:** Hier wird nicht inkrementell um 1 relabeled wird, sondern um den minimalen Abstand + 1 einer inzidenten Kante.
- **Global relabeling:** Zu beginng und jede $O(m)$ Kanteninspektionen werden alle Kanten mittels reverse BFS gelabelt.
- **Returning Flow:** Sonderbehandlung von Kanten mit Level größer n (, da sie sowieso nur zurückfließen)
- **Gap Heuristiken:** Wenn ein Kantenlevel auf dem Pfad zu t fehlt, so kann sich kein Knoten damit verbinden.

6: RANDOMISIERTE ALGORITHMEN

Las Vegas Algorithmen: Ergebnis immer korrekt, die Laufzeit ist jedoch eine Zufallsvariable. Wir kennen dafür Hashing und Quicksort. **Monte Carlo Algorithmen:** Ergebnis bei k -facher Wiederholung mit Wahrscheinlichkeit $1 - p^k$ korrekt.

Einfach, effizient und oft bestes bekanntes Verfahren. Auch algebraische Methoden können hier verwendet werden. Häufig sind randomisierte Algorithmen jedoch schwer zu analysieren.

6.1 Ergebnisüberprüfung von Sortieren

Permutationseigenschaft. $\langle e_1, \dots, e_n \rangle$ ist eine Permutation von $\langle e'_1, \dots, e'_n \rangle$ gdw.

$$q(z) := \prod_{i=1}^n (z - \text{field}(\text{key}(e_i))) - \prod_{i=1}^n (z - \text{field}(\text{key}(e'_i))) = 0$$

wobei \mathbb{F} ein Körper ist, auf den alle Elemente injektiv abgebildet werden. Wenn dies keine Permutation ist, dann ist $q \neq 0$

Idee: Ein Polynom n -ten Grades hat maximal n Nullstellen. Wenn wir also an einer zufälligen Stelle $x \in \mathbb{F}$ auswerten, so ist $P[q \neq 0 \wedge q(x) = 0] \leq \frac{n}{|\mathbb{F}|}$. Dies ist ein Monte-Carlo-Algorithmus welcher sich in Linearzeit berechnen lässt.

Problem: Finden eines passenden Körpers \mathbb{F}

Sort Checking. Sei h eine zufällige Hashfunktion mit Werten aus $[0, U - 1]$ mit $h(S) := \sum_{e \in S} h(e)$. Dann überprüfe ob die Folge E eine Permutation der Folge E' ist indem die beiden Hashes auf Gleichheit überprüft werden.

Idee: Die Wahrscheinlichkeit, dass beide Folgen denselben Hashwert haben ist maximal $\frac{1}{U}$.

6.2 Hashing

Ziel: Suchen in worst case konstanter Zeit.

Perfektes Hashing. Idee: injektives h . Braucht allerdings $\omega(n)$ Bits Speicherplatz.

Fast Space Efficient Hashing. n Elemente mit Speicherplatz $(1 + \varepsilon)n$ implementieren.

Cuckoo Hashing. Table mit Speicherplatz $(2 + \varepsilon)n$ mit zwei Hashfunktionen. Beim Einfügen kann es zu Kollisionen kommen. Dann wird das andere Element verschoben. Diese Kettenoperationen führen mit hoher Wahrscheinlichkeit zu Erfolg. Andernfalls wird die komplette Tabelle rebuildet. Amortisiertes Einfügen in $O(1)$.

Kann als ein Graph modelliert werden. Knoten sind Speicherzellen. Eine ungerichtete Kante bedeutet, dass ein Element mittels der Hashfunktionen an den Endknoten gespeichert werden kann. Eine gerichtete Kante bedeutet, dass ein Element am Startknoten gespeichert wird. Jeder Knoten darf maximal eine ausgehende Kante besitzen, der Rest wird verdrängt.

Das Einfügen funktioniert gdw. die Komponente, die die neue Kante enthält, nicht mehr Kanten als Knoten enthält.

Kann gezeigt werden mittels Abstraktion in den Fall dass die Kante auf einen Tree oder einen Pseudotree zeigt.

Die Wahrscheinlichkeit für ein Rebuild liegt bei zufälligen Graphen in $O(1/n)$.

7: EXTERNE ALGORITHMEN

Greift man auf sehr große Datenbestände zu, so sind diese in der Regel auf Sekundärspeicher wie Platte oder Band gespeichert, weil diese sehr günstig sind. Problem ist, dass der Zugriff auf den Speicher sehr viel Zeit benötigt.

Externe Algorithmen nehmen an, dass die *interne Arbeit*, also die Rechenarbeit, kostenlos ist. Minimiert werden soll die Anzahl an Zugriffen auf den Sekundärspeicher.

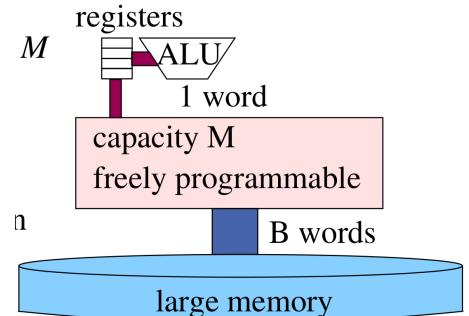


Abbildung 7.1. Sekundärspeicher-Modell mit schnellem internen Speicher der Größe M und einem beliebig großem externen Speicher, der über einen B Wörter breiten Bus angebunden ist.

7.1 Externe Stapel

Wir haben 2 interne Puffer mit Speicherplatz von je B Wörtern. Für das Lesen und Schreiben ist amortisiert jeweils eine Laufzeit von $O(1/B)$ nötig, da bei Über-/Unterlauf jeweils ein Puffer als Block auf den/vom externen Speicher geschrieben/gelesen wird.

7.2 Externes Sortieren

7.3 Mehrwegemischen, externes Mischen

Als nichttriviales Beispiel werden wir **Mehrwegemischen** betrachten:

```
MULTIWAYMERGE( $a_1, \dots, a_k, c : \text{File of Element}$ )
  for  $i := 1$  to  $k$  do  $x_i := a_i.\text{readElement}$ 
  for  $j := 1$  to  $\sum_{i=1}^k |a_i|$  do
    find  $i \in 1 \dots k$  that minimizes  $x_i$  // no IOs,  $O(\log k)$  time
     $c.\text{writeElement}(x_i)$ 
     $x_i := a_i.\text{readElement}$ 
```

Der Aufwand beträgt

- I/O: a_i lesen $\approx \frac{|a_i|}{B}$, c schreiben $\approx \sum_{i=1}^k \frac{|a_i|}{B}$

$$\Rightarrow \text{Insgesamt } \leq \approx 2 \frac{\sum_{i=1}^k |a_i|}{B}$$

Bedigungung: wir brauchen $k + 1$ Pufferblöcke.

Sortieren durch Mehrwegemischen mittels Run Formation

Ähnlich zu Merge Sort. Sortiere jeweils Eingabeportionen in sogenannten Runs der Größe M . Zunächst wurde das binäre Mischen mit $M/N = 2$ betrachtet. Mehrwegemischen ist eine Generalisierung dessen, welche eine bessere Laufzeit durch Verringerung der Mischphasen erzielt.

Wir können dann Mehrwegemischen verwenden, um zu sortieren:

1. Sortiere $\lceil \frac{n}{M} \rceil$ Runs mit je M Elementen.
2. Mische jeweils $\frac{M}{B}$ Runs, bis nur noch ein Run übrig ist.

Die Anzahl an I/Os setzt sich aus $2\frac{n}{B}$ I/Os für das Sortieren, $2\frac{n}{B}$ I/Os für das Mischen und $\lceil \log_{M/B} \frac{n}{M} \rceil$ Mischphasen zusammen, insgesamt also

$$\text{sort}(n) \cong \frac{2n}{B} \left(1 + \lceil \log_{M/B} \frac{n}{M} \rceil \right) \text{ I/Os.}$$

Die interne Arbeit setzt sich aus

- Run formation: $O(n \log M)$,
- Zugriffe auf die Prioritätsliste pro Phase: $O(n \log \frac{M}{B})$,
- Anzahl Phasen: $\lceil \log_{M/B} \frac{n}{M} \rceil$

zusammen, insgesamt ergibt dich die Laufzeit dann zu:

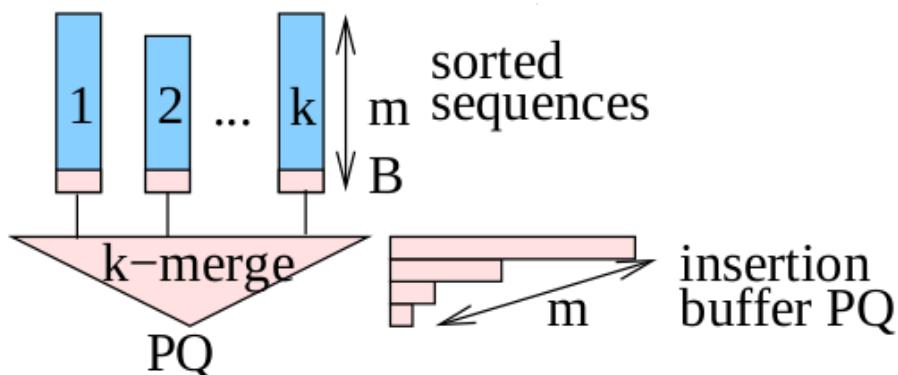
$$O\left(n \log M + n \log \frac{M}{B} \cdot \lceil \log_{M/B} \frac{n}{M} \rceil\right) = O(n \log n).$$

7.4 Externe Prioritätslisten

Wir beschränken uns nur auf nicht-adressierbare PQs (also nur `insert` und `deleteMin`, kein `decreaseKey`, etc.). Ziel: amortisierte Kosten von $\theta(\frac{1}{B} \log_{M/B} \frac{n}{M})$.

Mittelgroße PQs

Annahme: $k \cdot m \ll M^2 / B$ inserts



Internal Storage: ein insertion PQ-Puffer mit Größe m (konstanter Anteil von M) und eine deletion PQ die effektiv ein k -merge implementiert. Die deletion PQ speichert jeweils das kleinste Element einer Sequenz, zusammen mit dem Sequenzindex. Zudem sind jeweils bis zu B Sequenzen im internal Memory gespeichert, die verbleibenden Blöcke im external Memory.

Insertions werden nur bei Überlaufen in die Sequenzen geschrieben. Deletions erfolgen aus der PQ mit dem kleineren Minimum. Amortisiert $O(1/B)$ I/O-Zugriffe.

Die interne Analyse bezieht sich vor allem auf den Aufwand für das erneute Sortieren der bestehenden Strukturen und beläuft sich auf $O(\log m)$

7.5 Externe minimale Spannbäume

Annahme: $M = \Omega(n)$ konstant viele Worte pro Knoten. Also Knoten im Speicher, Kanten draußen.

Sortiere Kanten nach aufsteigendem Gewicht. Initialisiere Treekomponenten als jeweils ihren eigenen Knoten. Füge dann iterativ die Kanten hinzu, die nicht innerhalb derselben Komponente liegen.

8: APPROXIMATIONSALGORITHMEN

Idee: Für NP-harte Probleme, finde Lösungen welche in polynomieller Zeit garantiert "nahm Optimum dran sind.

8.1 Job Scheduling

- **Problem:** m Maschinen sollen n gewichtete Jobs abarbeiten, möglichst alle gleichzeitig fertig
 - Maschinen M_1, \dots, M_m
 - Jobs J_1, \dots, J_n
 - Lösung $S : \{1, \dots, n\} \rightarrow \{1, \dots, m\}$
 - Last von Maschine i : $L_i = \sum_{S(j)=i} t_j$
 - Zielfunktion: minimiere Makespan $L_{\max} = \max_i L_i$ (Wann ist letzte Maschine fertig?)
 - einfacher Fall: identische Maschinen, unabhängige Jobs, bekannte Ausführungszeiten
 - Problem ist **NP-hart**
- **Lösungsansatz:** Füge iterativ einen Job zu der aktuell am wenigsten ausgelasteten Maschine hinzu. Dann ist $L_{\max} \leq \sum_j \frac{t_j}{m} + \frac{m-1}{m} \cdot t_{\uparrow}$ für zuletzt beendeten Job \uparrow . Dies entspricht der durchschnittlichen Zeit für alle anderen Jobs plus der Zeit für t_{\uparrow} .
- **Approximationsfaktor:** Approximationsalgorithmus für ein Minimierungsproblem mit Zielfunktion f erzielt Approximationsfaktor ρ , falls er eine Lösung $x(I)$ findet sodass gilt:

$$\forall I \in M_I : \frac{f(x(I))}{f(x^*(I))} \leq \rho$$

- einfacher Fall: $\rho = 1 \rightsquigarrow A$ liefert stets optimale Lösung

Das hier vorgestellte ListScheduling erzielt einen Approximationsfaktor von $2 - 1/m$

8.2 Turing-Reduzierbarkeit

- **Definition:** Suchproblem Π **NP-schwer** oder **NP-hart**, falls ein **NP-vollständiges Entscheidungsproblem** L existiert mit $L \leq_T^p \Pi$, d.h.
 - Orakel-Turingmaschine für L mit Orakel für Π (eine Befragung = 1 Schritt) in polynomieller Laufzeit
 - ~> Turing-Reduktion in Polynomialzeit
 - ~> Wenn Π polynomiell lösbar ist, dann auch L

8.3 Allgemeines Travelling Salesman-Problem

- **Probleminstanz:** Graph $G = (V, E := V \times V)$, Längenfunktion $c : E \rightarrow \mathbb{Z}_+$
- **Zielfunktion** für Permutation π von V :

$$\begin{aligned} f(\pi) &= \sum_{i=1}^{n-1} c(\pi(i), \pi(i+1)) + c(\pi(n), \pi(1)) \\ &\rightsquigarrow f^*(G, c) = \min_{\pi} f(\pi) \end{aligned}$$

- **Gesucht:** Permutation π mit minimalem $f(\pi) = f^*(G, c)$

8.4 Entscheidungsproblem – Hamiltonkreis

- **Probleminstanz:** Graph $G = (V, E)$
- **Frage:** Gibt es Hamilton-Kreis in G ?
 \cong Permutation π derart, dass $\pi(1), \dots, \pi(n), \pi(1)$ Kreis
- Problem ist **NP**-vollständig

a -Approximation

- **Gegeben:**
 - $a \geq 1$
 - Probleminstanz (wie oben)
 - Zielfunktion (wie oben)
- **Gesucht:** Permutation π mit $f(\pi) \leq a \cdot f^*(G, c)$
- **Satz:** Für jedes $a \geq 1$ ist TSP- a -Approximations-Suchproblem **NP**-hart. Beweisansatz über implementieren des Hamiltonkreis-Problems.

8.5 Euler-Touren/-Kreise

Tour, welche jede Kante in einem Graphen genau einmal besucht. Dieses Problem lässt sich in Linearzeit $O(|E| + |V|)$ lösen.

Es lässt sich eine solche Tour finden gdw. G zusammenhängen ist und $\forall v \in V : \text{Grad}(v)$ ist gerade gilt.

8.6 Metrisches TSP

- **Definition:** Wie TSP, aber *Dreiecksungleichung* wird von c verlangt:

$$\forall x, y, z \in V : c(x, y) + c(y, z) \geq c(x, z)$$

- **Satz:** Für Instanzen des Problems kann man in Polynomialzeit eine 2-Approximation berechnen.
- **Lösungsansatz:** Bestimme MST, dupliziere alle Kanten, finde Eulerkreis und entferne Duplikate.

8.7 Pseudopolynomielle Laufzeit

- **Laufzeitabhängigkeit:** Laufzeit $t(I)$ für Eingabe I abhängig von Größe $n(I)$ der Repräsentation von I
- **Binäre Codierung:** Codierung von $k \in \mathbb{N}$ braucht $n(I) = n_2(I) = \Theta(\log_2 k)$ Bits
- **Unäre Codierung:** Codierung von $k \in \mathbb{N}$ braucht $n(I) = n_1(I) = k$ Bits
- **Polynomielle Laufzeit $t(n)$, wenn**

$$\exists \text{ Polynom } p(n) \forall I : t(I) \leq p(n_2(I))$$

- **Pseudopolynomielle Laufzeit $t(n)$, wenn**

$$\exists \text{ Polynom } p(n) \forall I : t(I) \leq p(n_1(I))$$

- **Achtung!**

- *Pseudopolynomielle Laufzeit*: Tatsächliche Form der Eingabe muss nicht unär erfolgen.
- Nicht verwechseln mit *quasipolynomieller Laufzeit* $t(n) = 2^{O((\log n)^c)}$ (Konstante $c > 0$)

Alle bekannten Pseudopolynomiellen Approximationsalgorithmen verwenden Dynamische Programmierung.

8.8 Knapsack

- **Probleminstanz:**

- Gegenstände $M = \{1, \dots, n\}$
 - Maximalgröße $W \in \mathbb{N}$ (Rucksackgröße)
 - Größen $w_i \in \mathbb{N}$ (oBdA jedes $w_i \leq W$, es passt also alles alleine in den Rucksack)
 - Profite $p_i \in \mathbb{N}$
- ↪ Gegenstand i hat Größe w_i und Profit p_i

- **Lösungen:** Teilmenge $M' \subseteq M$ mit

$$w(M') = \sum_{i \in M'} w_i \leq W$$

- **Gesucht:**

- Teilmengen mit möglichst großem Profit
- Zielfunktion $f(M') = p(M') = \sum_{i \in M'} p_i$
- Maximierungsproblem $f^*(I)$

- **Codierung** ($\hat{P} := \sum_i p_i$)

Bestandteil	$u_2(T)$	$u_1(T)$
$\{1, \dots, n\}$	$\log n$	n
W	$\log W$	W
$\langle w_1, \dots, w_n \rangle$	$n \log W$	nW
$\langle p_1, \dots, p_n \rangle$	$n \log \hat{P}$	$n\hat{P}$
insgesamt	$\Theta(n \log W + n \log \hat{P})$	$\Theta(nW + n\hat{P})$

- **Schwere:** NP-schwer

- bei “normaler” Messung der Eingabegröße
 - aber: pseudopolynomielle Laufzeit erreichbar
- ↪ schwach NP-schwer

- **Pseudopolynomielle Laufzeit** durch *dynamische Programmierung*:

$$C(i, P) := \min\{w(M') : M \subseteq \{1, \dots, i\} \wedge p(M') \geq P\}$$

falls ein solches M' existiert, ∞ sonst. D.h. die kleinste Kapazität für Gegenstände I mit Profit $\geq P$.

$$C(1, P) = \begin{cases} w_1, & \text{falls } p_1 \geq P \\ \infty, & \text{sonst} \end{cases}$$

$$C(i+1, P) = \min\{C(i, P), w_{i+1} + C(i, P - p_{i+1})\}$$

```
DYNPROGKNAPSACK(n, W, ⟨w1, …, wn1, …, pn⟩)
  for P ← 1 to P̂ do
    C(1, P) ← ...
    for i ← 1 to n - 1 do
      for P ← 1 to P̂ do
        C(i + 1, P) ← min{C(i, P), wi+1 + C(i, P - pi+1)}
  return max{P : C(n, P) ≤ W}
```

– Erweiterung: in $C(i, P)$ speichern, welche Objekte der $1, \dots, i$ benutzt werden

→ Skalarprodukt Objekt-Bitvektor und Profit ist maximaler Profit

8.9 (Voll) Polynomielle Approximationsschemata

- **Vorgaben:**

- $\left\{ \begin{array}{l} \text{Minimierungs} \\ \text{Maximierungs} \end{array} \right\}$ -Problem $\Pi = \{D, S, f\}$
 - Eingabemenge D
 - $S \ni S_I$ Menge der für Eingabe $I \in D$ gültigen Lösungen
 - Bewertungsfunktion $f : S_I \rightarrow \mathbb{N}$
- Algorithmus $\mathcal{A}(I, \varepsilon)$ mit $\varepsilon \in \mathbb{R}_+, I \in \Pi_D$

- **PTAS** \mathcal{A} (pol. Approx.-Schema), falls $\forall \varepsilon > 0 \exists$ Polynom $p(n) : \forall I \in \Pi_D :$

1. $f(\mathcal{A}(I, \varepsilon)) \leq \left(\frac{1+\varepsilon}{1-\varepsilon}\right) f^*(I)$
2. Laufzeit $t(I, \varepsilon) \leq p(n_2(I))$

- **FPTAS** \mathcal{A} (voll pol. Approx.-Schema), falls \exists Polynom $p(n, x) : \forall I \in \Pi_D, \varepsilon > 0 :$

1. $f(\mathcal{A}(I, \varepsilon)) \leq \left(\frac{1+\varepsilon}{1-\varepsilon}\right) f^*(I)$
2. Laufzeit $t(I, \varepsilon) \leq p(n_2(I), \frac{1}{\varepsilon})$

- Jedes FPTAS ist PTAS, aber nicht umgekehrt

Knapsack – FPTAS

- **Implementierung:**

```
EPSAPPROXKNAPSACK( $\varepsilon, n, W, w, p$ )
 $P \leftarrow \max_i p_i$ 
 $K \leftarrow \varepsilon \frac{P}{n}$  erlaubter Fehler pro Element, Skalierungsfaktor
each  $p'_i \leftarrow \lfloor \frac{p_i}{K} \rfloor$  // skaliertes Profits
 $x' \leftarrow \text{DYNPROGKNAPSACK}(n, W, w, p')$ 
return  $x'$ 
```

- **Analyse:**

- x^* optimale Lösung für ursprüngliches p , x' optimale Lösung für p'
 - $px^* = \sum_{i:x_i=1} p_i = \max.$ Profit des Originalproblems
 - *Frage:* Wie gut ist px' im Vergleich zu px^* ?
- $px' \geq (1 - \varepsilon)px^*$

- **Laufzeit:** $O(n^3 \frac{1}{\varepsilon})$

- dyn. Programmierung für p' Problem dominiert → Laufzeit $n\hat{P}'$
- $n\hat{P}' = n \sum i : x'_i = 1 p'_i \leq n^2 \max_i p'_i = n^2 \left\lfloor \frac{\hat{P}}{K} \right\rfloor = n^2 \left\lfloor \frac{\hat{P}n}{\varepsilon P} \right\rfloor \leq n^3 \frac{1}{\varepsilon}$

9: FIXED-PARAMETER-ALGORITHMEN

Für einfache Instanzen NP-harter Probleme können ggf. exakte Lösungen gefunden werden. Dazu führen wir den Parameter $k = \text{Ausgabegraph}$ ein. Wenn k klein ist, dann ist das Problem einfach.

9.1 Fixed Parameter Tractable (FPT)

$L \in FPT$ bzgl. Parameter k gdw. ein Algorithmus existiert mit einer Laufzeit in $O(f(k) \cdot p(n))$ mit f als von n unabhängiger Funktion und p einem von k unabhängigem Polynom (welches aber von n abhängen darf).

9.2 Vertex Cover/ Kantenabdeckung

- **Eingabe:** ungerichteter Graph $G = (V, E)$ mit $k \in \mathbb{N}$.
- **Problemstellung:** $\exists V' \subset V : |V'| = k \wedge \forall u, v \in E : u \in V' \vee v \in V'$. D.h. gibt es eine Teilmenge an k Knoten, sodass jede Kante inzident zu einem ausgewählten Knoten ist.
- **Lösungsansatz:**
 - Mittels Kernbildung (Kernelization): $\text{degree}(v) > k \Rightarrow v \in \text{Lösung} \vee \text{unlösbar}$. Füge also iterativ alle Knoten mit Grad größer k hinzu und reduziere k um 1.
 - Mittels tiefenbeschränkter Suche: Wähle eine Kante. Rufe den Algorithmus rekursiv mit Parametern $k - 1$ und jeweils einem der Endknoten aus G entfernt auf. Für $|E| = 0$ gib True zurück, für $k = 0$ gib False zurück. Der Algorithmus hat eine Laufzeit von $O((n + m) \cdot 2^k)$.

10: PARALLELE ALGORITHMEN

10.1 Einleitung

Indem man Parallelverarbeitung verwendet, kann man sowohl Ressourcen einsparen als auch Ressourcenrestriktionen brechen:

- **Zeitersparnis:** Arbeiten p Computer an einem Problem, so sind sie bis zu p mal so schnell.
- **Kommunikationsersparnis:** Fallen Daten verteilt an, so kann man sie auch verteilt (vor)verarbeiten.
- **Energieersparnis:** Zwei Prozessoren mit halber Taktfrequenz brauchen weniger Energie als ein voll getakteter Prozessor.
- **Speicherbeschränkung:** Mehr Prozessoren haben mehr Hauptspeicher, mehr Cache,...

Es gibt sehr viele Modelle der Parallelverarbeitung, wir werden hier allerdings nur zwei Standardmodelle diskutieren:

- **Prozessornetzwerke mit Nachrichtenkopplung.**

Prozessoren sind hier "normale CPUs" mit lokalem Speicher. Gemeinsam mit seinem lokalen Speicher wird eine lokale CPU auch *processing element* (PE) genannt. Der Datenaustausch passiert über ein Netzwerk in Form von Nachrichten zwischen zwei Prozessoren.

- **Parallele Registermaschinen mit Specherkopplung.**

Auch hier wird mit normalen CPUs gearbeitet, allerdings haben diese keinen lokalen Speicher, sondern sind über ein Netzwerk mit Speichermodulen verbunden, auf welche alle CPUs zugreifen können. Der Datenaustausch erfolgt über das Netzwerk zwischen einer CPU und einem Speicher.

Auf nachrichtengekoppelte Rechner werden wir genauer eingehen.

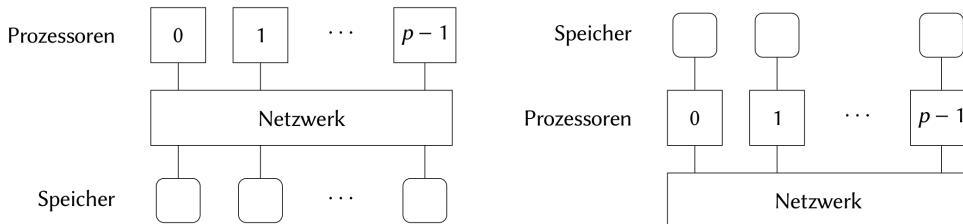


Abbildung 10.1. Vergleich zwischen *parallelen Registermaschinen mit Specherkopplung* (links) und einem *Prozessornetzwerk mit Nachrichtenkopplung* (rechts).

Nachrichtenkopplung vs. Specherkopplung

Neben den oben erläuterten Vorteilen von Parallelverarbeitung haben beide Modelle auch Nachteile:

- **Nachrichtenkopplung:**

- Zwei Prozessoren werden zum Datentransport benötigt. Das passt dem Empfänger nicht immer.
- Parallelismus muss explizit programmiert werden.

- **Specherkopplung:**

- *Skalierbarkeit*: Ist eine große Anzahl an Prozessoren sinnvoll?
- *Kostenmaß* bei Speicherzugriffskonflikten?

Als eine gute Strategie hat es sich erwiesen, den Entwurf für einen verteilten Speicher durchzuführen, da dieser einen viel breiteren Bereich abdecken kann. Die Implementierung erfolgt dann gegebenenfalls für einen gemeinsamen Speicher.

10.2 Nachrichtengekoppelte Parallelrechner

Modell

- **Netzwerk:** Vollständig verknüpftes Punkt-zu-Punkt-Netzwerk
 - voll-duplex
 - Nachrichten überholen sich nicht
- **Prozessoren:** RAMs, können jeweils maximal gleichzeitig
 - eine Nachricht an einen beliebigen Empfänger senden (`send(smsg,to)`)
 - eine Nachricht von einem beliebigen Sender empfangen (`rmsg := recv(from)`)
 - oder beides gleichzeitig (`rmsg := sendRecv(smsg, to, from)`)

Als *Kostenmodell* für das Senden oder Empfangen von l Bytes verwenden wir

$$T_{\text{comm}}(l) = T_{\text{start}} + l \cdot T_{\text{byte}},$$

wobei in der Praxis meist $T_{\text{byte}} \ll T_{\text{start}}$. Ignoriert wird hier unter anderem der “Abstand” zwischen Sender und Empfänger.

Als *Programmiermodell* verwenden wir **SPMD** (*single program multiple data*). Alle PEs führen hier dasselbe Programm aus, unterschieden wird (zur Symmetriebrechung) häufig durch “Ränge” der PEs (paarweise verschiedene PE-Nummern).

Analyse

- Ausführungszeit: $T(p)$.
- Arbeit: $W = p \cdot T(p)$. Ist ein Kostenmaß.
- Span: $T_\infty = \min_p T(p)$. Misst Parallelisierbarkeit. Für beliebig viele Prozessoren der kleinste Wert.
- Absoluter Speedup: $S = \frac{T_{\text{seq}}}{T(p)}$. Entspricht der Beschleunigung gegenüber dem besten sequenziellen Algorithmus.
- Relativer Speedup: $\frac{T(1)}{T(p)}$.
- Effizienz: $E = \frac{S}{p}$. Superlinearer Speeup für relative Beschleunigung über 1, idR. nicht möglich (außer wenn plötzlich alles in den Cache passt, usw.). Ziel ist es aber generell Werte nahe 1 oder in $\Theta(1)$ zu erreichen.

Parallele Reduktion

Im Folgenden gehen wir über die grundlegenden Werkzeuge, die wir benötigen, um parallele Programme analysieren zu können.

Definition 10.2.1 (Reduktion). Sei \otimes eine binäre, assoziative Operation auf einer Menge M .

Für $x = (x_0, \dots, x_{p-1}) \in M$ definieren wir

$$R_\otimes(x) = \bigotimes_{i < p} x_i = x_0 \otimes \cdots \otimes x_{p-1}.$$

Nun gilt folgender Satz:

Satz 10.2.2. Wenn \otimes eine assoziative Operation ist, welche sich in konstanter Zeit berechnen lässt, und p Elemente x_0, \dots, x_{p-1} auf p PEs verteilt sind, dann kann man $\bigotimes_{i < p} x_i$ in Zeit $O(\log p)$ auf PE 0 berechnen.

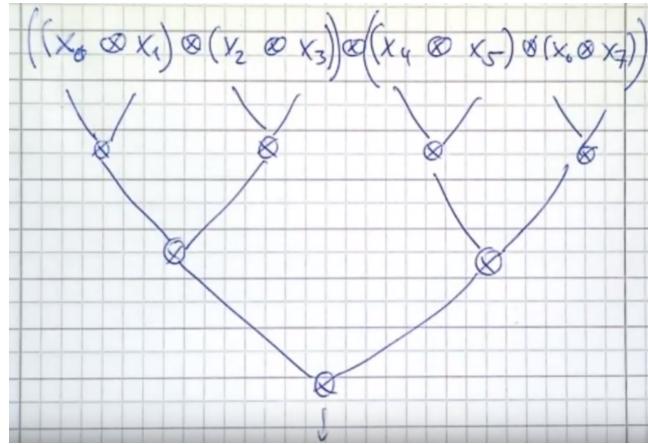


Abbildung 10.2. Idee hinter obigem Satz.

Sequenziell wäre die optimale Laufzeit $O(n)$. Auf $p = n$ PEs hätte man nach obigem Satz eine Laufzeit von $O(\log n)$ erreicht, also eine Beschleunigung von $O\left(\frac{n}{\log n}\right)$. „Ideal“ wäre eine Beschleunigung von $O(p) = O(n)$.

Wie kann man die Beschleunigung noch weiter verbessern?

Parallele Reduktion mit $p < n$

Verwenden wir nun $p < n$ viele PEs, um eine Reduktion auf n Elementen durchzuführen, so erhält jedes PE n/p Daten-elemente. Die PEs berechnen zuerst die Reduktion der lokalen Elemente, und anschließend wird auf diesen Reduktionen eine parallele Reduktion durchgeführt. Laufzeit hierfür ist $O(n/p) + O(\log p)$ und die Beschleunigung somit

$$\frac{O(n)}{O(n/p + \log p)}.$$

Ist $p \in O(n/\log n)$, so ist die Beschleunigung $O(p)$.

Man kann also durch Verringerung der Prozessorzahl p die Beschleunigung in die Nähe von p bringen. Ineffiziente Algorithmen werden also durch Verringerung der Prozessorzahl effizient. Dieses Prinzip nennt man **Brent's Prinzip**.

Parallele Präfixsummen

Wir verwenden \otimes wie oben definiert. Wir definieren nun

$$P_{\otimes}(x) = y = (y_0, \dots, y_{p-1})$$

mit $y_i = \bigotimes_{k \leq i} x_k$.

Es ist also $y_0 = x_0$ und $y_{i+1} = y_i \otimes x_{i+1}$.

Präfixsummen – Hyperwürfel-Algorithmus

Wir machen es uns hier einfach und legen fest, dass $p = n = 2^d$ (für $d \in \mathbb{N}$) und \otimes kommutativ.

Jede PE erhält nun eine „Koordinate“ $0 \leq i \leq 2^d - 1$. Diese wird als Bitvektor

$$i = (i_{d-1} \cdots i_0) \quad \text{mit} \quad i_j \in \{0, 1\}$$

repräsentiert. Hier ist i_k das k -te Bit von rechts in i .

Wir erlauben Kommunikation zwischen zwei PE i und i' nur, wenn die Hammingdistanz ihrer Bitvektoren 1 ist, sie sich also nur in einer Stelle unterscheiden. Wir erhalten so einen *Hyperwürfel* aus PEs.

Wir berechnen nun Präfixsummen auf einem solchen Hyperwürfel:

```

PREFIXSUM( $x, \otimes$ )
 $y := x$  // auf PE  $i$  liegt  $x_i$ 
 $s := x$  // für Summe von Elementen in Unterwürfel
for  $k := 0$  to  $d - 1$  do
     $s' := \text{SENDRECV}(s, i \oplus 2^k, i \oplus 2^k)$ 
     $s := s \otimes s'$ 
    if  $i_k \equiv 1$  then  $y := y \otimes s'$  // auf PE  $i$  liegt  $y_i = x_0 \otimes \dots \otimes x_i$ 

```

Wir erhalten eine Laufzeit

$$T_{\text{prefix}} \in O((T_{\text{start}} + l \cdot T_{\text{byte}}) \cdot \log p).$$

Diese Laufzeit ist nicht optimal für $l \cdot T_{\text{byte}} > T_{\text{start}}$. Wie man sie optimieren kann wird in der Vorlesung "Parallele Algorithmen" näher erläutert.

Paralleles Sortieren

Hier gibt es zwei verschiedene Aufgabenvarianten:

1. Alle n Elemente liegen zu Beginn auf PE 0. Deswegen muss jedes Element von PE 0 mindestens einmal angefasst werden. Die Laufzeit ist daher in $\Omega(n)$.
2. Je n/p Elemente liegen zu Beginn auf PE i . Dieser Fall ist wesentlich interessanter.

Zunächst betrachten wir den einfachen Fall $p = n$ (Prozessor i hat Eingabeelement x_i). Wir behalten die Grundidee von Quicksort bei:

- Wir wählen ein Element pv als Pivot.
- Elemente werden umverteilt:
 - kleiner als pv : auf Prozessoren mit kleineren Rängen
 - größer als pv : auf Prozessoren mit großen Rängen
- Parallele Rekursion.

Wir schauen uns den **Theoretiker-Quicksort** an:

```

// Teil 0: Vorbereitungen
THEOQSORT0( $x$ )
 $i \in 0, p - 1$  // hier hat PE  $i$  Element  $x_i$ 
 $p :=$  Anzahl aller PEs
theoQSort( $x, i, p$ )

// Teil 1: kleine Elemente zählen
THEOQSORT( $x, \dots$ )
if  $p = 1$  then return
 $r := \text{rand}(0, p - 1)$  // derselbe Wert in der gesamten Partition
 $pv = d@r$  broadcast value of pivot
 $small := (d \leq pv)$ 
 $j := \text{prefixSum}(small, 0, i)$ 
 $p' := \text{bcast}(j, p - 1)$   $p'$  ist der border index

// Teil 2: Datenumverteilung und Rekursion
if  $small \equiv 1$  then send( $d, j - 1$ )
else send( $d, p' + i - j$ )
 $d := \text{recv}()$ 
recursive theoQSort of "left"/"right" part

```

Die erwartete Rekursionstiefe ist in $O(\log p)$, die Zeit jeweils in $O(T_{\text{start}} \log p)$. Insgesamt ist die erwartete Zeit also in $O(T_{\text{start}}(\log p)^2)$.

11: STRINGOLOGY

Inhalt dieses Kapitels:

- Strings sortieren
- Patterns suchen
- Datenkompression

11.1 Strings sortieren

Naive Sortierverfahren, wie sie aus der Vorlesung “Algorithmen 1” bekannt sind, sind beim Sortieren von Strings ineffizient, deswegen gibt es für das Sortieren von Strings andere Algorithmen. Ein solcher ist der **Multikey Quicksort**-Algorithmus:

```
MKQSORT (S: String Seq, l:N) : String Seq
assert ∀e, e' ∈ S : e[1...l-1] = e'[1...l-1]
if |S| ≤ 1 then return S
pick p ∈ S randomly
return concatenation of
  MKQSORT ((e ∈ S : e[l] < p[l]), l),
  MKQSORT ((e ∈ S : e[l] = p[l]), l+1),
  MKQSORT ((e ∈ S : e[l] > p[l]), l)
```

Abbildung 11.1. Pseudocode-Implementierung des Multikey-Quicksort-Algorithmus.

Dieser Algorithmus sortiert eine String-Sequenz und nimmt an, dass die ersten $l-1$ Buchstaben bereits sortiert wurden. Zuerst wird ein zufälliges Pivotelement gewählt. Danach wird die übergebene Sequenz an Strings in drei Teilsequenzen geteilt:

1. Sequenz an Strings, deren l -ter Buchstabe kleiner ist als der l -te Buchstabe des Pivotelements.
2. Sequenz an Strings, deren l -ter Buchstabe derselbe ist wie der l -te Buchstabe des Pivotelements.
3. Sequenz an Strings, deren l -ter Buchstabe größer ist als der l -te Buchstabe des Pivotelements.

Auf die erste und dritte Teilsequenz wird der Algorithmus nun rekursiv mit dem selben Parameter l ausgeführt, da die Buchstaben an der l -ten Position nicht übereinstimmen (müssen) — auf die zweite Teilsequenz wird der Algorithmus rekursiv mit dem Parameter $l+1$ ausgeführt, weil hier die l -ten Buchstaben aller Wörter in der Sequenz gleich sind. Die Laufzeit des Algorithmus ist in $O(|S| \log |S| + d)$, wobei d die Summe der eindeutigen Präfixe der Strings in S ist.

Weiterhin gibt es das Verfahren **Most Significant Digit Radix Sort**. Wir müssen grundsätzlich eine minimale Anzahl von Zeichen in einer Liste von Strings ansehen, dies nennt sich das Distinguishing Prefix. Das Longest Common Prefix (LCP) ist die Anzahl der Zeichen, welche in einer sortierten Reihenfolge identisch zum vorherigen String sind. Kann in-place und out-of-place sortiert werden.

Vorgehen:

1. Zähle wie häufig jeder Buchstabe als erster in der Liste der Strings vorkommt.
2. Bilde die exklusive Präfixsumme für jeden String und sortiere anhand dessen die Strings in Buckets.
3. Rufe rekursiv jeden noch nicht beendeten Bucket auf.

11.2 Pattern Matching

Hinweis: In diesem Abschnitt sind Arrays 1-basiert.

In diesem Abschnitt wird es darum gehen, alle oder zumindest ein Vorkommen eines **Patterns** $P = p_1 \dots p_m$ in einem gegebenen **Text** $T = t_1 \dots t_n$ zu finden. Im Allgemeinen ist $n \gg m$, also der Text wesentlich länger als das Pattern, das wir in ihm suchen.

Naives Pattern Matching

Das naive Vorgehen ist, an jeder Position von T zu schauen, ob an dieser das gesuchte Pattern vorkommt. Offensichtlich ist dieser Algorithmus in $O(nm)$, da im schlimmsten Fall für jede Position des Textes das gesamte Pattern durchlaufen werden muss. Dieser Algorithmus kann folgendermaßen implementiert werden:

```
NAIVEPATTERNMATCH (P, T)
i, j := 1
while i ≤ n - m + 1
    while j ≤ m ∧ ti+j-1 = pj do j++
    if j > m then return "P occurs at pos i in T"
    i++
    j := 1
```

Abbildung 11.2. Pseudocode-Implementierung des naiven Pattern-Matching-Algorithmus.

Knuth-Morris-Pratt

Ein anderer Algorithmus zum Finden von Patterns in einem gegebenen Text ist der **Knuth-Morris-Pratt-Algorithmus**. Dieser hat sogar optimale Laufzeit, nämlich $O(n + m)$.

Idee dieses Algorithmus ist es, das Pattern eleganter nach vorne zu verschieben, wenn es einen Mismatch zwischen Text und Pattern gibt. Hierfür brauchen wir ein Hilfswerkzeug:

Für einen String S mit Länge k sei $\alpha(S)$ die Länge des längsten Präfixes von $S_{1\dots k-1}$, das auch Suffix von $S_{2\dots k}$ ist.¹

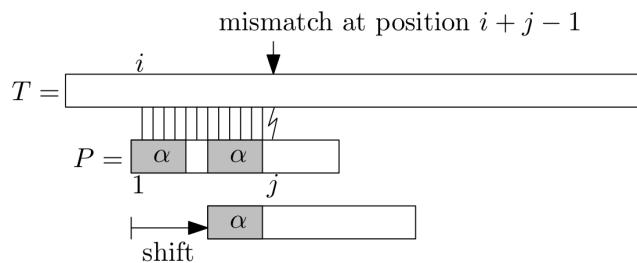


Abbildung 11.3. Idee beim Verschieben des Patterns: α wurde bereits gemitacht. Früher als mit dem bereits gemitachten Suffix kann das nächste Vorkommen von P nicht auftauchen, also kann man P direkt um $j - 1 - \alpha$ verschieben.

Der Algorithmus besteht aus zwei Teilen:

1. **Border-Array berechnen ($O(m)$)**. Damit die oben erläuterten Verschiebungen nachher effizient durchgeführt werden können, berechnen wir für das leere Wort *und* jeden Buchstaben in P einen α -Wert. Diese Werte ergeben das **Border-Array**:

¹ Wir lassen absichtlich bei Betrachtung des Präfixes den letzten und bei Betrachtung des Suffixes den ersten Buchstaben weg, damit $\alpha(S) = 0$ ist, wenn $|S| = k = 1$ ist.

$$\text{border}[j] = \begin{cases} -1, & \text{falls } j = 1 \\ \alpha(P_{1\dots j-1}), & \text{sonst} \end{cases}.$$

P	a	n	a	n	a	s	
border	-1	0	0	1	2	3	0

Abbildung 11.4. Beispiel für das Border-Array eines Patterns.

2. **Pattern matchen** ($O(n)$). Nun verwenden wir das erstellte Border-Array, um Vorkommnisse von P in T zu finden.

Wir starten sowohl im Text als auch im Pattern an Position 1 und fangen an zu matchen. Kommt es an Position $1 \leq j \leq m$ des Patterns zu einem Mismatch, so können wir P direkt um $j - \text{border}[j] - 1$ verschieben. In Pseudocode sieht das so aus:

```
KMPMATCH (P, T)
i, j := 1
while i ≤ n - m + 1
  while j ≤ m ∧ ti+j-1 = pj do j++
  if j > m then return "P occurs at pos i in T"
  i += j - border[j] + 1
  j := max {1, border[j] + 1}
```

Eine Ausführung des Algorithmus kann also folgendermaßen aussehen:

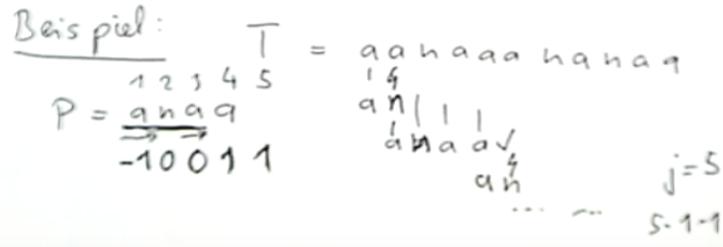


Abbildung 11.5. Beispiel für das Verwenden des Knuth-Morris-Pratt-Algorithmus.

Suffix-Arrays

Im Folgenden werden Arrays wieder mit Position 0 beginnen. Wir verwenden desweiteren folgende Festlegungen:

- Ein **String** ist ein Array von Buchstaben,

$S[0 \dots n] := S[0 \dots n-1] := [S[0], \dots, S[n-1]]$.

- Das **Suffix** S_i sei der Substring $S[i \dots n]$ von S .
- Wir setzen an das Ende jedes Strings ausreichend viele **Endmarkierungen**: $S[n] := S[n+1] := \dots := 0.0$ sei per Definition kleiner als alle anderen vorkommenden Zeichen.

Das **Suffix-Array** eines Strings lässt sich nun folgendermaßen konstruieren:

0	banana	5	a
1	anana	3	ana
2	nana	1	anana
3	ana	0	banana
4	na	4	na
5	a	2	nana

1. Bilde die Menge aller Suffixe S_i ($i = 0, \dots, n-1$) des Strings.
2. Sortiere die Menge aller Suffixe des Strings (z.B. mit Multikey Quicksort).

Abbildung 11.6. Beispiel für die Konstruktion des Suffix-Arrays des Strings “banana”.

Mithilfe dieses Suffix-Arrays lassen sich später viele Suchprobleme in Linearzeit lösen. Beispielsweise ist die Suche nach dem längsten Substring, der (eventuell mit Überschneidung) zweimal im Text vorkommt, linear — dafür muss nach Berechnung des Suffix-Arrays der längste String gefunden werden, der Präfix von zwei Strings im Suffix-Array ist (im Beispiel oben wäre das “ana”).

Berechnung des Suffix-Arrays in Linearzeit

Das Suffix-Array eines Strings lässt sich in Linearzeit berechnen.² Hier soll lediglich das Prinzip erläutert werden, genauere Angaben gibt es im Paper.

Wir betrachten den String

$$T[0, n) = \begin{matrix} x & a & b & b & a & d & a & b & b & a & d & o \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \end{matrix}$$

Unser Ziel ist das Suffix-Array

$$\text{SA} = (12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0).$$

Wir gehen wie folgt vor:

0. Suffixe wählen. Sei

$$B_k = \{i \in [0, n] : i \bmod 3 = k\}$$

und $C = B_1 \cup B_2$ sowie S_C die Menge der entsprechenden Suffixe. C ist also die Menge aller Positionen in T , an denen Suffixe mit einer nicht durch 3 teilbaren Länge beginnen. Hier ist $C = \{1, 4, 7, 10, 2, 5, 8, 11\}$.

1. Gewählte Suffixe sortieren. Wir fügen am Ende von T beliebig viele 0 hinzu und bilden zuerst für $k = 1, 2$ die Strings

$$R_k = [t_k t_{k+1} t_{k+2}] [t_{k+3} t_{k+4} t_{k+5}] \cdots [t_{\max B_k} t_{\max B_k + 1} t_{\max B_k + 2}].$$

Der Charaktere von R_k sind also Tripel. Das letzte Tripel ist immer eindeutig, weil $t_{\max B_k + 2} = 0$. Sei $R = R_1 \odot R_2$. Hier ist

$$R = [\text{abb}][\text{ada}][\text{bba}][\text{do}\text{o}][\text{bba}][\text{dab}][\text{bad}][\text{o}\text{o}\text{o}].$$

Die Ordnung der Suffixe von R stimmt mit der Ordnung der Suffixe S_i überein, deswegen genügt es, die Suffixe von R zu sortieren.

Wir sortieren R nun, indem wir die einzelnen Charaktere von R sortieren und durch ihren Rang in R ersetzen, sprich wir invertieren sie:

$$\text{SA}_R = (8, 0, 1, 6, 4, 2, 5, 3, 7).$$

Nun weisen wir jedem Suffix einen Rang zu. Dazu sei $\text{rank}(S_i)$ der Rang von S_i in C . Für $i \in B_0$ sei $\text{rank}(S_i)$ nicht definiert.

Hier ist $\text{rank}(S_i) = \perp 1 4 \perp 2 6 \perp 5 3 \perp 7 8 \perp 0 0$.

2. Restliche Suffixe sortieren. Jeder Suffix $S_i \in S_{B_0}$ sei dargestellt durch $(t_i, \text{rank}(S_{i+1}))$. Da wir alle anderen Suffixe oben schon sortiert haben ist $\text{rank}(S_{i+1})$ hier stets definiert.

Offensichtlich ist

$$S_i \leq S_j \Leftrightarrow (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1})),$$

² Kärkkäinen, Sanders, Burkhardt: Linear Work Suffix Array Construction

also lassen sich die Paare Radix-sortieren.

Hier ist

$$S_{12} < S_6 < S_9 < S_3 < S_0, \text{ weil } (0, 0) < (a, 5) < (a, 7) < (b, 2) < (x, 1).$$

3. Zusammenführen. Das Zusammenführen erfolgt vergleichsbasiert. Beim Vergleichen von $S_i \in S_C$ mit $S_j \in S_{B_0}$ unterscheiden wir zwei Fälle:

$$\begin{aligned} i \in B_1 : \quad S_i \leq S_j &\Leftrightarrow (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1})) \\ i \in B_2 : \quad S_i \leq S_j &\Leftrightarrow (t_i, t_{i+1}, \text{rank}(S_{i+2})) \leq (t_j, t_{j+1}, \text{rank}(S_{j+2})) \end{aligned}$$

Hier ist z.B. $S_1 < S_6$ weil $(a, 4) < (a, 5)$ und $S_3 < S_8$ weil $(b, a, 6) < (b, a, 7)$.

Die Suffixtabellenkonstruktion kann verallgemeinert werden mit Diff-Überdeckungen, um die Platzeffizienz der Implementierung zu steigern.

Suchen in Suffix-Arrays

Um ein Pattern in einem String zu finden, zu dem man das Suffix-Array konstruiert hat, muss man lediglich ein Suffix finden, das das gesuchte Pattern als Präfix hat. Man kann so beispielsweise mit binärer Suche in $O(m \log n)$ ein Vorkommen von P in T finden.

Nutzen wir eine zusätzliche Struktur, das **LCP-Array** — dieses speichert in $\text{LCP}[i]$ die Länge des längsten gemeinsamen Präfixes von $\text{SA}[i]$ und $\text{SA}[i - 1]$ — so können wir die Suchzeit auf $O(m + \log n)$ reduzieren.

0	banana	5	a	0	a
1	anana	3	ana	1	a na
2	nana	1	anana	3	a nana
3	ana	0	banana	0	banana
4	na	4	na	0	na
5	a	2	nana	2	n ana

Abbildung 11.7. Suffixe, Suffix-Array und LCP-Array des Strings “banana”.

Um das LCP-Array berechnen zu können brauchen wir das **invertierte Suffix-Array**. Dieses gibt Aufschluss darüber, wo im Suffix-Array ein bestimmter Suffix steht. Offensichtlich ist $\text{SA}^{-1}[\text{SA}[i]] = i$.

Der Algorithmus sieht folgendermaßen aus ($O(n)$):

```
CALCULATELCPARRAY (SA-1, SA)
h := 0, LCP[1] := 0
for i = 1, ..., n do
    if SA-1[i] ≠ 1 then
        while ti+h = tSA[SA-1[i]-1]+h do h++
        LCP[SA-1[i]] := h
    h := max(0, h - 1)
```

Suffix-Bäume

Noch anschaulicher, allerdings wesentlich platzverbrauchender, sind **Suffix-Bäume** von Strings. Sie sind formal der *kompaktierte Trie der Suffixe* und lassen sich (wenn auch sehr kompliziert) in $O(n)$ berechnen.

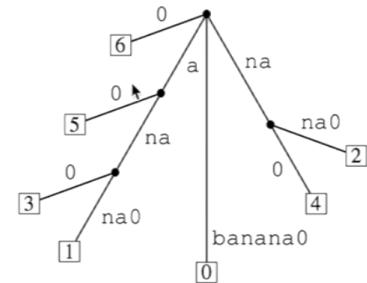


Abbildung 11.8. Beispiel für den Suffix-Baum des Strings “banana”.

“Naiv” ist die Erstellung des Suffixbaums in $O(n^2)$. Man kann ihn aber auch aus Suffix-Array und LCP-Array in Linearzeit konstruieren. Dazu hängt man die Suffixe sukzessive in der Tiefe ein, die ihr LCP-Wert angibt:

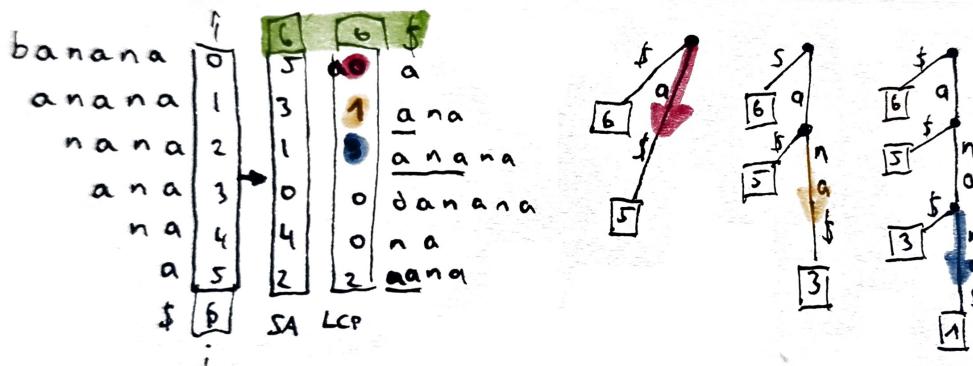


Abbildung 11.9. Sukzessive Konstruktion des Suffix-Baums aus Suffix- und LCP-Array. Zuerst hängt man $\$$ und das erste Suffix (dessen LCP-Wert immer 0 ist) an die Wurzel. Anschließend nutzt man den LCP-Wert des darauffolgenden Suffixes (hier 1), um festzulegen, wo der Suffix zum Baum hinzugefügt werden muss (durch Pfeile gekennzeichnet).

Die Suche in einem Suffix-Baum ist relativ simpel – man muss lediglich den entsprechenden Kanten entlanglaufen, alle Vorkommen des Patterns liegen im entsprechenden Teilbaum.

Zur Angabe der Komplexitäten sind zwei Fälle zu unterscheiden:

1. Die ausgehenden Kanten sind als Arrays der Größe $|\Sigma|$ gespeichert. Dann ist die Suchzeit in $O(m)$ und der Gesamtplatzbedarf in $O(n|\Sigma|)$.
2. Die ausgehenden Kanten sind als Arrays gespeichert, deren Größe proportional zur Anzahl der Kinderknoten ist. Dann ist die Suchzeit in $O(m \log |\Sigma|)$ und der Gesamtplatzbedarf in $O(n)$.

Suffix Arrays mit Präfix-Verdoppelung

Gegeben eines Strings, bilden wir eine Liste aller längstmöglichen Suffixe innerhalb des Strings. Das Suffix-Array gibt dabei jeweils an, wie viele Suffixe kleiner sind als das gegebene Suffix. Das Inverse Suffix ist die Inverse Funktion hierzu.

1. $n := 1$
2. Finde eine Ordnung für Suffixe der Länge 1 und berechne jeweils die Anzahl der Präfixe, die kleiner sind.
3. Analysiere die nächsten n Zeichen des Suffixes. Schaue nach, wie viele Präfixe dieser Form aus dem vorigen Schritt kleiner sind und ordne die Suffixe dementsprechend.
4. $n++$
5. Aktualisiere die Anzahl der kleinsten Präfixe für die Länge von n .
6. Gehe zu 2.

Der Output ist eine sortierte Liste mit Suffixen.

11.3 Datenkompression

Eine Anwendung der Suffix-Arrays und -Trees ist die **Datenkompression**. Inhalt dieser Vorlesung wird ausschließlich die *verlustfreie Textkompression* sein.

Wörterbuchbasierte Textkompression

Für besonders große Datenbestände bietet sich eine **wörterbuchbasierte Textkompression** an. Grundidee ist, $\Sigma' \subseteq \Sigma^*$ zu wählen und $S \in \Sigma^*$ durch $S' = \langle s'_1, \dots, s'_k \rangle \in \Sigma'^*$ zu ersetzen, sodass $S = s'_1 \dots s'_k$ ist. Problem ist der hohe zusätzliche Platzbedarf für das Wörterbuch.

Lempel-Ziv-Kompression

Die **Lempel-Ziv-Kompression** baut das Wörterbuch *on the fly* bei Codierung und Decodierung, sodass dieses nicht explizit gespeichert werden muss.

```
NAIVE LZCOMPRESS(⟨ $s_1, \dots, s_n$ ⟩,  $\Sigma$ )
 $D := \Sigma$  // init dictionary
 $p := s_1$  // current string
for  $i := 2$  to  $n$  do
  if  $p \cdot s_i \in D$  then  $p := p \cdot s_i$ 
  else
    output code for  $p$ 
     $D := D \cup p \cdot s_i$ 
     $p := s_i$ 
  output code for  $p$ 
```

```
NAIVE LZDECODE(⟨ $c_1, \dots, c_k$ ⟩)
 $D := \Sigma$ 
output decode( $c_1$ )
for  $i := 2$  to  $k$  do
  if  $c_i \in D$  then
     $D := D \cup \text{decode}(c_{i-1}) \cdot \text{decode}(c_i)[1]$  else
     $D := D \cup \text{decode}(c_{i-1}) \cdot \text{decode}(c_{i-1})[1]$ 
  output decode( $c_i$ )
```

Abbildung 11.10. Kompressions- und Dekodierungs-Algorithmus für die Lempel-Ziv-Kompression.

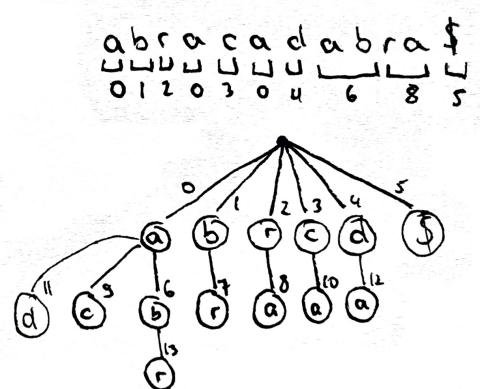


Abbildung 11.11. Beispiel für das Bilden der Lempel-Ziv-Kompression des Wortes “*abracadabra*”. Der Baum wird *on the fly* berechnet und nicht übergeben, sondern nur die komprimierte Information und das Alphabet.

12: BURROWS-WHEELER-TRANSFORMATION

Die **Burrows-Wheeler-Transformation** erzeugt eine sinnvolle Permutation des eingegebenen Strings; sie gruppiert Zeichen mit ähnlichem Kontext nahe beieinander. Die Struktur der Permutation beinhaltet alle Informationen, die benötigt werden, um eine Rücktransformation durchzuführen, es sind also keine Zusatzinformationen nötig. Hin- und Rücktransformation geht in $O(n)$. Sie wird hauptsächlich zur Vorverarbeitung statischer Texte genutzt, um sie komprimieren, indizieren und in ihnen suchen zu können.

12.1 Konstruktion

Sei $T = \text{lalalangng\$}$ der gegebene String (mit angehängtem \\$-Zeichen), $n = |T|$ und $T^{(i)}$ die i -te Permutation von T (durch i mal den vordersten Buchstaben nehmen und hinten anhängen). Man erhält die Burrows-Wheeler-Transformation von T so:

1. Schreibe $T^{(1)}$ bis $T^{(n)}$ untereinander.
2. Sortiere $T^{(1)}$ bis $T^{(n)}$.
3. Die letzte Spalte ist $T^{\text{BWT}} (= L)$, die Burrows-Wheeler-Transformation von T .

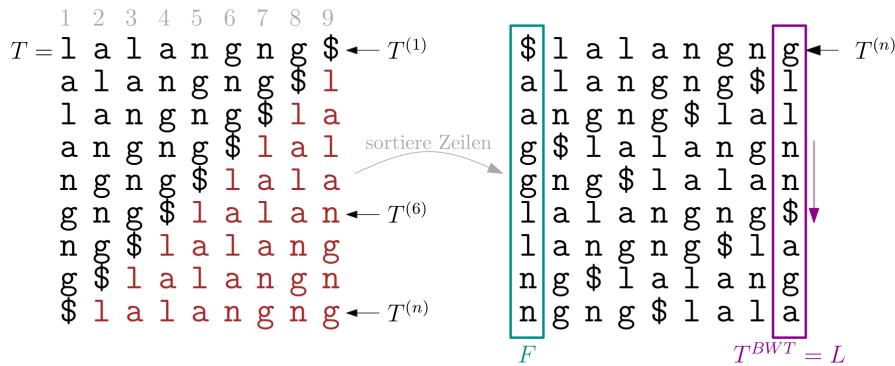


Abbildung 12.1. Konstruktion der Burrows-Wheler-Transformation von $T = \text{lalalangng\$}$, $T^{\text{BWT}} = \text{gllnn\$ aga}$. Da T^{BWT} die letzte Spalte ist schreibt man oft auch L stattdessen. Die erste Spalte wird auch F genannt.

Naiv benötigt die Berechnung von T^{BWT} $O(n^2 + n \log n)$ Schritte. Die Berechnungszeit lässt sich aber auf $O(n)$ reduzieren.

12.2 Beobachtungen

Folgende Eigenschaften lassen sich feststellen:

- Die Zeilen der oben konstruierten Matrix enthalten die sortierten Suffixe von T (vom Zeilenstart bis \\$ gehend).
 - Die Zeichen der letzten Spalte (also T^{BWT}) sind also die Zeichen, die vor dem zu ihrer Zeile gehörenden Suffix stehen.
- Formaler ist $T^{\text{BWT}}[i]$ das Zeichen vor dem i -ten Suffix in T :

$$T^{\text{BWT}}[i] = L[i] = T[\text{SA}[i] - 1] = T^{(\text{SA}[i])}[n]$$

Da wir mithilfe des DC3-Algorithmus das Suffix-Array in Linearzeit berechnen können, können wir auch die Burrows-Wheeler-Transformation in Linearzeit bestimmen.

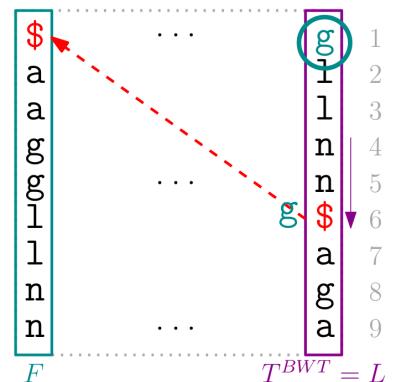
12.3 Rücktransformation

Wir können aus einer vorliegenden T^{BWT} einfach F – also die erste Spalte der Matrix – konstruieren, indem wir die Buchstaben von T^{BWT} sortieren. Hängen wir nun T^{BWT} und F hintereinander, so haben wir bereits Buchstabenpaare, die so auch in T auftreten. Sortieren wir nun die beiden Spalten (also die Buchstabenpaare) lexikographisch, so erhalten wir die auf F folgende Spalte. Durch diesen Prozess lässt sich die gesamte Matrix und somit T rekonstruieren.

$\begin{matrix} g \\ l \\ l \\ n \\ n \\ \$ \\ a \\ a \end{matrix}$	$\begin{matrix} \$ \\ a \\ a \\ g \\ g \\ l \\ l \\ n \\ n \end{matrix}$	$\begin{matrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{matrix}$
T^{BWT}	F	F

Diese Art der Rücktransformation benötigt $O(n^2 \log n)$ Schritte. Im Folgenden werden wir die Rücktransformation auf Linearzeit reduzieren. Dazu benötigen wir **Last-to-front mapping**:

$\text{LF}[i] :=$ Position in L , an der Vorgänger von $L[i]$ steht



Da die Spalten der BWT-Matrix zyklisch sind, ist der Vorgänger von $L[i]$ derjenige Buchstabe, der in $F[i]$ steht, also

$\text{LF}[i] =$ Position, an der $L[i]$ in F steht

Abbildung 12.2. Gesucht ist der Vorgänger von $\$$. Stellen wir uns T^{BWT} ein zweites Mal links von F vor, so sehen wir, dass es g ist.

Wir erhalten folgenden Zusammenhang:

$$\text{LF}[i] = j \Leftrightarrow T^{(\text{SA}[j])} = (T^{(\text{SA}[i])})^{(n)}.$$

Weitere Überlegungen zur Rücktransformation

Wir können desweiteren folgende Beobachtungen an T^{BWT} machen:

- Gleiche Zeichen haben gleiche Reihenfolge in F und L .
- Falls $L[i] = L[j]$ für $i < j$, dann ist $\text{LF}[i] < \text{LF}[j]$.

Grund dafür ist, dass die Zeilen der BWT-Matrix lexikographisch sortiert sind.

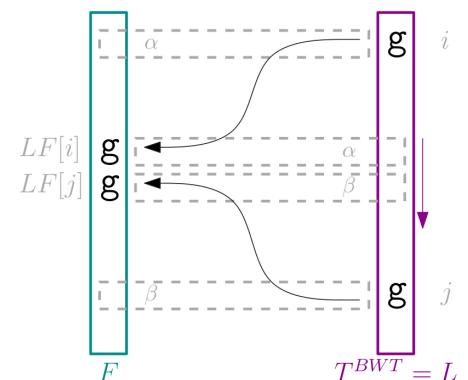


Abbildung 12.3. Präfixe α und β und wie sie in der Matrix vorkommen.

Wir können also LF rein aus T^{BWT} berechnen. Dazu brauchen wir nur zwei Hilfsfunktionen:

- $C(a) := \# \text{ Zeichen} < a$

- $\text{occ}[i] := \# \text{ Zeichen} = L[i] \text{ in } L[1 \dots i]$

Nun können wir $\text{LF}[i]$ darstellen als

$$\text{LF}[i] = C(L[i]) + \text{occ}[i]$$

und können somit LF in $O(n)$ berechnen, da sich C und occ in Linearzeit berechnen lassen.

Implementierung

Zuerst berechnen wir LF . Hier sieht die Implementierung so aus:

1. Initialisiere occ und h . h sei ein Array, das zählt, wie oft ein bestimmter Buchstabe vorkommt, damit wir nachher C gescheit berechnen können.
2. Laufe durch $L = T^{\text{BWT}} (i = 1 \dots n)$
 - $h(L[i])++$
 - $\text{occ}(L[i]) = h(L[i])$
3. Konstruiere C aus h : $C(\$) = 0, C(\alpha) = C(\alpha - 1) + h(\alpha - 1)$ (α ist ein Buchstabe, $\alpha - 1$ sein Vorgänger)
4. $\text{LF}[i] = C(L[i]) + \text{occ}[i]$

	T^{BWT}									LF
1	1	2	3	4	5	6	7	8	9	4
2	g	l	l	n	n	\$	a	g	a	6
3	1	1	2	1	2	1	1	2	2	7
4	n	n	\$	\$	a	a	g	a		8
5										9
6										1
7										2
8										5
9										3

	T^{BWT}									LF
\$										4
a										6
g										7
l										8
l										9
n										1

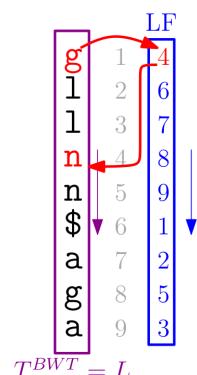
	T^{BWT}									LF
1										4
2										6
2										7
2										8
2										9
7										1
5										2
5										5
7										3
8										
9										

Abbildung 12.4. Beispiel des Algorithmus zur Berechnung von LF nach Durchführung.

Nun kann T von rechts nach links berechnet werden:

1. $T[n] = \$ \Rightarrow \text{LF}[\cdot] = 1$. Das ist unabhängig von T so.
2. $L[1] = g \Rightarrow T[n-1] = g \Rightarrow \text{LF}[1] = 4$
3. $L[4] = n \Rightarrow T[n-2] = n \Rightarrow \dots$

Also geht auch die Rücktransformation in $O(n)$.



12.4 Was bringt die BWT?

Die Vorteile der Burrows-Wheeler-Transformation sind nicht direkt erkennbar — sie nutzt dieselben Zeichen wie T und benötigt den gleichen Platz.

Allerdings wird die *Komprimierung stark vereinfacht*, weil Zeichen mit ähnlichem Kontext gruppiert werden. Besonders gut funktioniert sie auf Texten mit vielen gleichen Substrings, wie beispielsweise einem englischen Fließtext. Zur Vereinfachung von *Indexierung* und *Suche* steuert sie auch bei, weil Vorgänger von Suffixen einfach bestimmt werden können.

Im Folgenden werden wir uns die Burrows-Wheeler-Transformation im Kontext von *Kompression* und *Suche* anschauen.

12.5 Kompression

Wir schauen uns zwei Kompressionsmöglichkeiten an: die *move to front*-Kodierung und die Huffman-Kodierung

MTF-Kodierung

Idee der **MTF-Kodierung** ist es, lokale Redundanz zu nutzen und so kleine Zahlen für gleiche Zeichen, die nahe beieinander liegen, zu verwenden. Die Umsetzung funktioniert so:

1. Initialisiere Y mit Alphabet von T^{BWT} .
2. Durchlaufe T^{BWT} ($i = 1, \dots, n$)
 - Generiere $R[1, \dots, n]$, wobei $R[i]$ die Position von $T^{\text{BWT}}[i]$ in Y codiert.
 - Schiebe $T^{\text{BWT}}[i]$ an den Anfang von Y .

$T^{\text{BWT}} = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ g & l & l & n & n & a & g & a & \$ \end{matrix}$ $R = \begin{matrix} 3 & 4 \end{matrix}$	$\begin{matrix} 1 & 2 & 3 & 4 & 5 \\ \$ & a & g & l & n \end{matrix}$ $\begin{matrix} g & \$ & a & l & n \end{matrix}$ $Y = \begin{matrix} 1 & g & \$ & a & n \end{matrix}$
---	---

Abbildung 12.5. Es wurde hier gerade die 4 eingefügt und deswegen 1 in Y nach vorne genommen. Als nächstes muss 1 codiert ($\cong 1$) und Y anschließend nicht verändert werden, weil 1 ja eh schon ganz vorne steht.

Huffman-Kodierung

Die **Huffman-Kodierung** erzeugt präfixfreie Codes variabler Länge. Der Ablauf ist:

1. Notiere vorkommende Symbole und ihre jeweiligen Häufigkeiten. Sie sind die Blätter des (binären) Huffman-Baumes.
2. Verknüpfe die zwei seltensten Knoten in einem neuen Knoten. Die Häufigkeit des neuen Knotens ist die Summe der Häufigkeiten seiner Kinder. Dies erfolgt nun iterativ.
3. Die Wurzel hat relative Häufigkeit 1 (bzw. absolute Häufigkeit $|T|$).
4. Beschriffe die Kanten zwischen einem Knoten und seinen beiden Kindern mit 0 und 1. Der Pfad von der Wurzel zu einem bestimmten Blatt ergibt den Code des Symbols, zu dem das Blatt gehört.

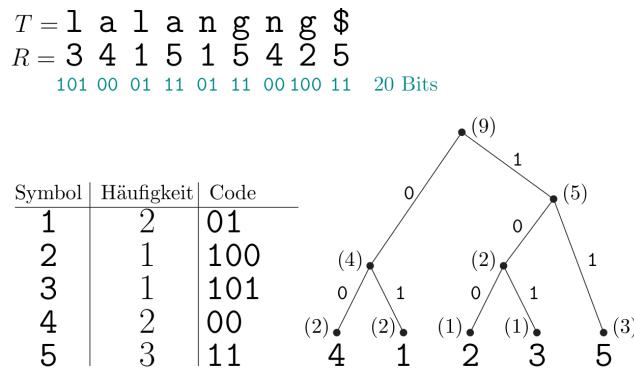


Abbildung 12.6. MTF-Kodierung R von T , die Häufigkeit der in R vorkommenden Symbole und die mit dem Huffman-Baum erzeugten Codes.

12.6 Suche in der Burrows-Wheeler-Transformation

Wir möchten nun in T^{BWT} nach einem Pattern P suchen. Hier sei

$$P = \text{bar} \quad \text{und}$$

$$T = \text{abracadabrabarbara\$} \quad \text{und somit}$$

$$\text{BWT} = \text{arrd\$rcbbraaaaaabba}$$

Wir benötigen dazu zwei Hilfsmittel:

- Das Array C beinhaltet für jeden eindeutigen Buchstaben in $t \in T$ die Position des ersten Suffixes im Suffix-Array, das mit t beginnt.
- $\text{rank}(i, X, \text{BWT})$ gibt an, wie oft ein Buchstabe X in $\text{BWT}[0, \dots, i - 1]$ vorkommt.

Wir suchen nun nach P in BWT . Wir suchen rückwärts, starten also mit ‘‘r’’. Dazu ermitteln wir alle Suffixe, die mit r starten. Wir nutzen dazu C und rank :

- Initiales Intervall: $[\text{sp}_0, \text{ep}_0] = [0, \dots, n - 1]$.
- Ermittle Intervall der Suffixe, die mit r starten:

$$\begin{aligned} - \text{sp}_1 &= C[r] + \text{rank}(\text{sp}_0, r, \text{BWT}) = 15 + \text{rank}(0, r, \text{BWT}) = 15 + 0 = 15 \\ - \text{ep}_1 &= C[r] + \text{rank}(\text{ep}_0 + 1, r, \text{BWT}) - 1 = 15 + 4 - 1 = 18 \end{aligned}$$

i	BWT	$\mathcal{T}[\text{SA}[i]..n - 1]$
0	a	\$
1	r	a\$
2	r	abarbara\$
3	d	abrabarbara\$
4	\$	abracadabrabarbara\$
5	r	acadabrabarbara\$
6	c	adabrabarbara\$
7	b	ara\$
8	b	arbara\$
9	r	bara\$
10	a	barbara\$
11	a	brabarbara\$
12	a	bracadabrabarbara\$
13	a	cadabrabarbara\$
14	a	dabrabarbara\$
15	a	ra\$
16	b	rabarbara\$
17	b	racadabrabarbara\$
18	a	rbara\$

Wir suchen nun analog nach ‘‘ar’’, indem wir sp_2 und ep_2 aus sp_1 und ep_1 berechnen. Das Ganze dann nochmal für ‘‘bar’’ und wir erhalten 9 und 10 als diejenigen Suffixe, die mit bar anfangen.

Abbildung 12.7. Intervall $[\text{sp}_1, \text{ep}_1]$.

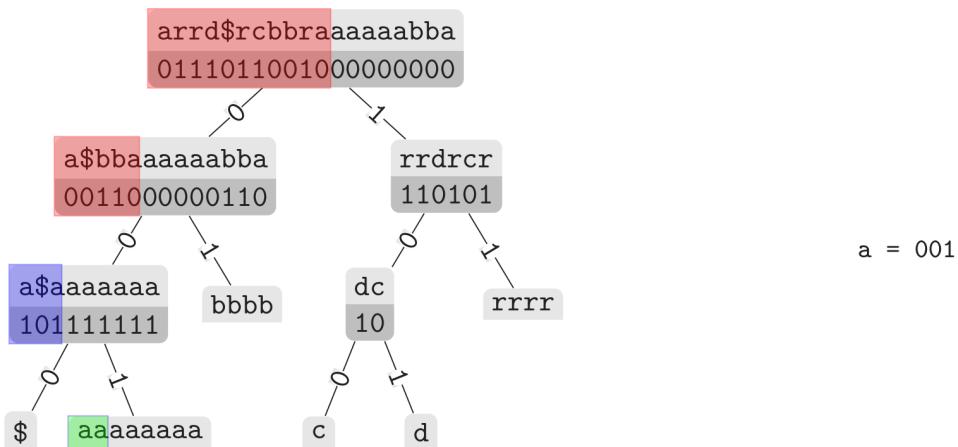
Zusammenfassung

Wir brauchen also nur C und R , um Abfragen zu Existenz und Anzahl eines Patterns machen zu können. Die Ausführungszeit ist in $O(m \cdot t_{\text{rank}})$, wobei t_{rank} die Ausführungszeit einer rank-Operation ist.

Als nächstes werden wir uns damit beschäftigen, wie wir die rank-Operation implementieren können. Wir werden dazu *wavelet trees*¹ verwenden.

12.7 Wavelet Trees

Wavelet Trees erlauben ein schnelles Berechnen der rank-Operation. Dazu wird in einem Baum codiert, ob ein bestimmter Buchstabe des Strings (hier des BWT) in der oberen oder der unteren Hälfte des Alphabets liegt. So werden die Buchstaben des BWT einem Kindknoten zugeordnet, wo auf dem jeweiligen Teilalphabet erneut eine Zweiteilung stattfindet. Dieser Prozess wiederholt sich so lange, bis in jedem Blatt nur Zeichen einer Art stehen.



$$\text{rank}(11, a, WT) = \text{rank}(\text{rank}(\text{rank}(11, 0, b_e) = 5, 0, b_0) = 3, 1, b_{00}) = 2$$

Abbildung 12.8. Wavelet Tree einer BWT und zugehörige Berechnung von $\text{rank}(11, a, \text{BWT})$.

Abfragen können auf einem Wavelet Tree in konstanter Zeit durchgeführt werden. Der Wavelet Tree selbst benötigt $o(n)$ viel Platz, genauer

$$O\left(\frac{n}{\log n} + \frac{n \log \log n}{\log n} + \sqrt{n} \log n \log \log n\right).$$

Die Konstruktion des Wavelet-Trees ist nicht zwingend an die Unterteilung des BWT zwei lexikographische Teilalphabete gebunden. Beispielsweise lässt sich eine Unterteilung in zwei Teilwörter auch durch die Häufigkeit der Buchstaben konstruieren, wodurch ein **Huffman-Wavelet-Tree** entsteht.

¹ Grossi & Vitter, 2003

13: A COMPACT BIT-SLICED SIGNATURE INDEX (COBS)

Motivation: Wir wollen approximative Matchings für Patterns in großen Datensätzen effizient finden.

Gegeben ist ein Arrays der Länge m worin n Items eingefügt wurden. Es gibt k paarweise unabhängige Hashfunktionen. Es kann überprüft werden, ob ein Item eingefügt wurde. Diese $\text{query}(x)$ -Funktion hat einen einseitigen Fehler - sie kann wahr zurückliefern, auch wenn sie nicht eingefügt wurden.

Die Wahrscheinlichkeit, dass ein bestimmtes Bit 1 ist beträgt $1 - (1 - \frac{1}{m})^{kn}$. Die Wahrscheinlichkeit, dass ein falsches Wahr zurückgegeben wird ist die Wahrscheinlichkeit, dass bei allen k Hashfunktionen eine 1 zurückgegeben wird.

Dementsprechend können wir durch die Anzahl der Hashfunktionen bzw der Größe des Arrays die Fehlerwahrscheinlichkeit wie gewünscht beschränken.

14: GEOMETRISCHE ALGORITHMEN

Inhalt dieses Kapitels:

- Plane-Sweep-Algorithmus
- Konvexe Hülle
- Kleinste einschließende Kugel
- Range Search

14.1 Grundlegende Definitionen

Wir nennen $p \in \mathbb{R}^d$ einen **Punkt**. $p.i$ stelle die i -te Komponente von p dar. Für $d \in \{2, 3\}$ schreiben wir $p.x, p.y, p.z$ statt $p.1, p.2, p.3$.

Für zwei Punkte a, b definieren wir

$$\overline{ab} := \{\alpha \cdot a + (1 - \alpha) \cdot b : \alpha \in [0, 1]\}$$

als das **Segment** zwischen a und b .

Ein **Polygon** ist eine Menge an Segmenten, gegeben als Punktemenge $P = p_1, \dots, p_n$ mit $p_i \in \mathbb{R}^d, p_n = p_1 \cdot \overline{p_i, p_{i+1}}$ für $i = 1, \dots, n - 1$ ist der **Umriss** des Polygons.

Ist für alle $a, b \in P$ auch $\overline{ab} \in P$, so nennen wir P **konvex**.

14.2 Streckenschnitte

Bei diesem Problem sind n Strecken $S = \{s_1, \dots, s_n\}$ gegeben und wir wollen alle Schnittpunkte dieser, also $\bigcup_{s, t \in S} s \cap t$ berechnen.

Naiv lassen sich diese Streckenschnitte in $O(n^2)$ berechnen:

```
foreach {s, t} ⊆ S do
    if s ∩ t ≠ ∅ then output {s, t}
```

Dieser Algorithmus ist für große Datenmengen offensichtlich zu langsam.

Idee ist nun, dass eine (waagerechte) **Sweep-Line** von oben nach unten läuft. Dabei speichern wir Segmente, die l schneiden, und finden deren Schnittpunkte. Invariante ist, dass Schnittpunkte oberhalb von l korrekt ausgegeben wurden.

Orthogonale Streckenschnitte

Zuerst betrachten wir die Vereinfachung, dass nur orthogonale Segmente (also parallel zur x - oder y -Achse existieren).

```

 $T := \langle \rangle$  SortedSequence of Segment
invariant  $T$  stores vertical segments intersecting  $I$ 
 $Q := \text{sort}(\{(y, s) : \exists \text{ hor-seg } s \text{ at } y \vee \exists \text{ ver-seg } s \text{ starting/ending at } y\})$ 
foreach  $(y, s) \in Q$  in descending order do
    if  $s$  is ver-seg and starts at  $y$  then  $T.\text{insert}(s)$ 
    elif  $s$  is ver-seg and ends at  $y$  then  $T.\text{remove}(s)$ 
    else // horizontal segment  $s = \overline{(x_1, y)(x_2, y)}$ 
        foreach  $t = \overline{(x, y_1)(x, y_2)} \in T$  with  $x \in [x_1, x_2]$  do output  $\{s, t\}$ 

```

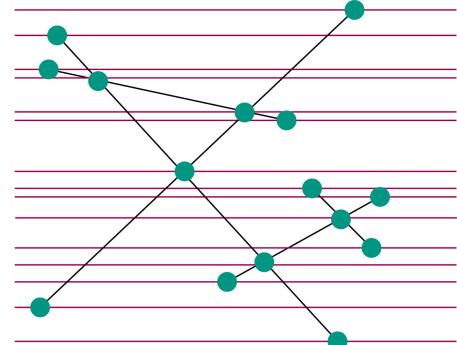
Hier sind T und Q die einzigen komplexen Datenstrukturen, die wir benötigen, also sortierte Listen an Segmenten (T geordnet nach x -Wert, Q nach y).

`insert` und `remove` gehen in $O(\log n)$, die `rangeQuery` für ein Segment in $O(\log n + k_s)$ (bei k_s Schritten mit horizontalem Segment s). Insgesamt haben wir also

$$O(n \log n + \sum_s k_s) = O(n \log n + k).$$

Verallgemeinerung

Wir verallgemeinern jetzt den Spezialfall von oben, verwenden allerdings folgende Vereinfachungen: Es gebe keine horizontalen Segmente und Überschneidungen sind immer nur zwischen zwei Segmenten, nicht mehr. Außerdem soll es keine Überlappungen geben, die Anzahl an Schnitten zwischen zwei Segmenten ist also immer entweder 0 oder 1.



Wir verwenden wieder T als nach x geordnete Liste der Strecken, die I schneidet. Außerdem verwenden wir *Ereignisse* – diese sind Änderungen von T , also das Starten und Enden von Segmenten sowie Schnittpunkte.

Einen Schnitttest müssen wir nur dann durchführen, wenn zwei Segmente an einem Ereignispunkt in T benachbart sind.

Abbildung 14.1. Die grünen Punkte stellen die Ereignisse dar. Außerdem ist I zum Zeitpunkt der Ereignisse dargestellt.

Zur Implementierung brauchen wir nun einige Zusatzmethoden:

FINDNEWEVENT ermittelt, ob es einen Schnitt zwischen zwei Segmenten s und t gibt.

```

FINDNEWEVENT( $s, t$ )
if  $s$  and  $t$  cross at  $y' < y$  then
     $Q.\text{insert}((y', \text{intersection}, (s, t)))$ 

```

Die Event-Handler werden kümmern sich um die Handhabung der drei möglichen Event-Types.

```
HANDLEEVENT( $y$ , intersection,  $(a, b)$ ,  $T$ ,  $Q$ )
output( $s \cap t$ )
 $T.\text{swap}(a, b)$ 
prev := pred( $b$ )
next := succ( $a$ )
findNewEvent(prev,  $b$ )
findNewEvent( $a$ , next)
```

```
HANDLEEVENT( $y$ , start,  $s$ ,  $T$ ,  $Q$ )
 $h := T.\text{insert}(s)$ 
prev := pred( $h$ )
next := succ( $h$ )
findNewEvent(prev,  $h$ )
findNewEvent( $h$ , next)
```

```
HANDLEEVENT( $y$ , finish,  $s$ ,  $T$ ,  $Q$ )
 $h := T.\text{locate}(s)$ 
prev := pred( $h$ )
next := succ( $h$ )
 $T.\text{remove}(s)$ 
findNewEvent(prev, next)
```

Nun können wir den Algorithmus implementieren.

```
 $T := \langle \rangle$  SortedSequence of Segment
invariant  $T$  stores relative order of segments intersecting  $l$ 
 $Q := \text{MaxPriorityQueue}$ 
 $Q := Q \cup \left\{ (\max \{y, y'\}, \text{start}, s) : s = \overline{(x, y)(x', y')} \in S \right\}$ 
 $Q := Q \cup \left\{ (\min \{y, y'\}, \text{finish}, s) : s = \overline{(x, y)(x', y')} \in S \right\}$ 
while  $Q \neq \emptyset$  do
     $(y, \text{type}, s) := Q.\text{deleteMax}$ 
    handleEvent( $y, \text{type}, s, T, Q$ )
```

Dieser Algorithmus benötigt $O(n \log n)$ zur Initialisierung und $O((n + k) \log n)$ für die Event-Schleife, insgesamt also $O((n + k) \log n)$.

14.3 Konvexe Hülle

Wir werden uns in diesem Abschnitt mit dem folgenden Problem beschäftigen:

Gegeben sei eine Punktmenge $P = \{p_1, \dots, p_n\} \subset R^2$. Gesucht ist ein konvexes Polygon C mit Eckpunkten $\in P$, sodass alle Punkte von P in C liegen.

Zuerst sortieren wir P lexikographisch. Das bedeutet, dass

$$p > q \Leftrightarrow p.x > q.x \vee (p.x = q.x \wedge p.y > q.y).$$

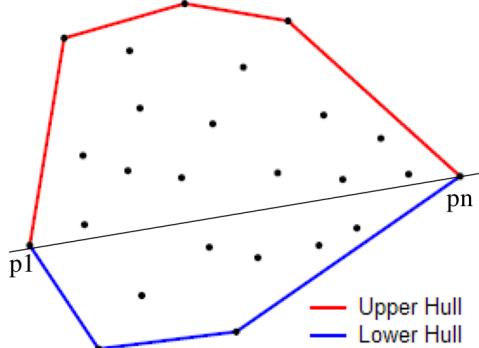


Abbildung 14.2. Obere Hülle.

Wir berechnen ohne Einschränkung nur die obere Hülle, also die Hülle um die Punkte oberhalb von $\overline{p_1 p_n}$.

Wir können beobachten, dass die obere Hülle ausschließlich Abbiegungen nach rechts macht (und die untere nur Abbiegungen nach links). Um damit arbeiten zu können müssen wir Abbiegungen definieren:

Definition 14.3.1 (Abbiegung). Für eine Punktemenge $P = \{p_1, \dots, p_n\}$ ist eine **Abbiegung nach rechts** an Stelle i vorhanden, falls p_{i+1} rechts von $\overline{p_{i-1} p_i}$ liegt.

Das konstruieren der oberen Hülle nennt sich auch **Graham's Scan**.¹

```
UPPERHULL( $p_1, \dots, p_n$ )
 $L := \langle p_n, p_1, p_2 \rangle$ : Stack of Point
invariant  $L$  is upper hull of  $\langle p_n, p_1, \dots, p_i \rangle$ 
for  $i := 3$  to  $n$  do
    while  $\neg$ rightTurn( $L$ .secondButLast,  $L$ .last,  $p_i$ ) do  $L$ .pop
     $L := L \circ \langle p_i \rangle$ 
return  $L$ 
```

Der Algorithmus selbst läuft in $O(n)$, weswegen das Sortieren dominiert und das ganze in $O(n \log n)$ liegt.

14.4 Kleinste einschließende Kugel

In diesem Abschnitt ist eine Punktmenge $P := \{p_1, \dots, p_n\} \subset \mathbb{R}^d$ gegeben und eine Kugel K mit minimalem Radius gesucht, sodass $P \subset K$. Wir verwenden einen Algorithmus in $O(n)$ nach Welzl.²

Q sei zu Beginn leer. Wir fügen Punkte derart zu Q hinzu, dass durch die Punkte in Q ein Ball aufgespannt wird, in dem alle $p \in P$ liegen.

```
sEB( $P, Q$ )
if  $|P| = 0 \vee |Q| = d + 1$  then return ball( $Q$ )
 $x := p \in P$  picked at random
 $B := sEB(P \setminus \{x\}, Q)$ 
if  $x \in B$  then return  $B$ 
return sEB( $P \setminus \{x\}, Q \cup \{x\}$ )
```

¹ Graham 1972, Andrew 1979

² Welzl, 1991

14.5 Range Search

Beim **Range Search** (Bereichssuche) haben wir wieder eine Menge $P = \{p_1, \dots, p_n\} \subset \mathbb{R}^2$ gegeben. Wir erhalten als Anfrage nun ein achsenparalleles Rechteck

$$Q := [x, x'] \times [y, y'].$$

Gesucht ist nun entweder $P \cap Q$ (*range reporting*) oder $k := |P \cap Q|$ (*range counting*).

Wir werden range counting in $O(\log n)$ und range reporting in $O(k + \log n)$ lösen. Dazu ist $O(n \log n)$ Vorverarbeitungszeit und $O(n)$ Platz notwendig.

Range Search im Eindimensionalen

Zuerst machen wir einen range search in einer Dimension. Dazu konstruieren wir im Voraus einen binären Suchbaum. Dieser codiert pro Blatt das größte Element des linken Teilbaums. So können die beiden Grenzen leicht gefunden werden.

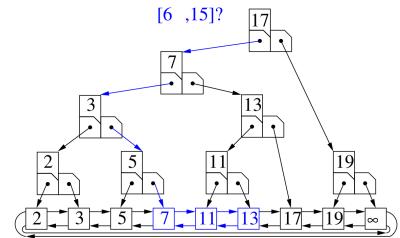


Abbildung 14.3. Ermittlung des ersten Elements, das größer als 6 ist.

Range Search im Zweidimensionalen – Erster Ansatz

Ein naives Verfahren ist es, die Punkte aus P in zwei Arrays A_x und A_y zu speichern, wobei die Punkte in A_x nach x -Wert und in A_y nach y -Wert sortiert sind.

Man bestimmt nun

$$k_x := \text{count}(x_0, x_1) \text{ in } A_x \quad (\text{alle Punkte mit } x_0 < x < x_1 \text{ und})$$

$$k_y := \text{count}(y_0, y_1) \text{ in } A_y \quad (\text{alle Punkte mit } y_0 < y < y_1).$$

Nun müssen noch $\min \{k_x, k_y\}$ Punkte gematcht werden, um zu überprüfen, dass der jeweilige Punkt sowohl im x - als auch im y -Wert im Rechteck liegt.

Insgesamt braucht dieser Ansatz also $O(\log n) + O(\min \{k_x, k_y\})$ Zeit, ist also nicht wirklich brauchbar.

Range Search im Zweidimensionalen – Zweiter Ansatz

Wir konstruieren nun einen balancierten Binärbaum mithilfe der x -Koordinaten. Anschließend berechnen wir die $O(\log n)$ Teilbäume, die zusammen alle Punkte mit $x_0 \leq x \leq x_1$ enthalten. Nun müssen diese nur noch nach y -Koordinate gefiltert werden. Wir speichern dazu in jedem Knoten die Punkte ab, die in seinem Teilbaum liegen, sortiert nach y -Wert.

Insgesamt können wir so den Algorithmus auf $O(\log^2 n + k)$ drücken.

Mithilfe von Wavelet Trees lässt sich die Zeit auf $O(\log n)$ reduzieren.

15: ONLINE-ALGORITHMEN

Inhalt dieses Kapitels:

- Einführung in Online-Algorithmen
- Beispiel: Job-Scheduling
- Beispiel: Skiausleihe
- Beispiel: Speicherverwaltung
- Beispiel: Auswahl von Experten

Online-Algorithmen werden verwendet, wenn Eingabegrößen nicht im Vornherein bekannt sind. Viele Algorithmen, die wir bisher diskutiert haben, benötigen Vorarbeitszeit, um optimale Ergebnisse liefern zu können, und sind daher nicht auf Probleme anwendbar, wo die Eingabe in serieller Natur vorliegt. Die bisher diskutierten Algorithmen werden daher auch **Offline-Algorithmen** genannt.

15.1 Übersicht

Wir werden uns im Folgenden Konzepte von Online-Algorithmen anhand von Beispielen anschauen. Zuerst definieren wir, was ein Online-Algorithmus formal ist.

- **Eingabe:** Folge von Anforderungen (*requests*)

$$\sigma = (r_1, \dots, r_n) \in R^n$$

- r_i muss auf Anforderung bearbeitet werden, es entsteht eine Antwort

$$a_i = g_i(r_1, \dots, r_i) \in A.$$

Es sind keine Informationen über die Zukunft verfügbar (r_{i+1}, \dots). Antworten sind unwiderruflich.

Der konkrete Algorithmus ist also durch g_1, \dots eindeutig festgelegt.

- **Kosten** $\text{cost}_n : R^n \times A^n \rightarrow \mathbb{R}_{>0}$
- **Ausgabe** für $\sigma \in R^n$ ist also

$$\text{ALG}[\sigma] = (g_1(r_1), g_2(r_1, r_2), \dots, g_n(r_1, \dots, r_n)) \in A^n$$

und die Kosten

$$\text{ALG}(\sigma) = \text{cost}_n(\sigma, \text{ALG}[\sigma])$$

Kompetitive Analyse

Um einschätzen zu können, wie gut ein Online-Algorithmus ist, vergleichen wir ihn mit einem optimalen Offline-Algorithmus — das ist die **Kompetitive Analyse** dieses Online-Algorithmus.

Definition 15.1.1 (c -kompetitiv). Ein Online-Algorithmus ist für ein Optimierungsproblem auf Input σ **c -kompetitiv**, falls es einen Offline-Algorithmus OPT gibt, der nach einem Kostenmaß $\text{OPT}(\sigma)$ eine Optimallösung berechnen kann, sodass für den betrachteten Online-Algorithmus ALG und alle Eingabesequenzen σ gilt:

$$\text{ALG}(\sigma) \leq c \cdot \text{OPT}(\sigma) + \alpha.$$

Ist die zusätzliche Konstante $\alpha \leq 0$, dann heißt ALG **strikt c -kompetitiv**.

Unterschied ist, dass man bei nicht-strikter Kompetitivität Ausnahmen erlaubt.

Wir können nun den **Wettbewerbsfaktor** (*competitive ratio*) für den Algorithmus ALG definieren als

$$c_{\text{ALG}} = \sup \left\{ \frac{\text{ALG}(\sigma)}{\text{OPT}(\sigma)} : \sigma \in R^+ \right\}.$$

Ist alg ein strikt c -kompetitiver Online-Algorithmus und

$$C = \{c : \text{ALG ist strikt } c\text{-kompetitiv}\},$$

so ist

$$c_{\text{ALG}} = \inf C \quad \text{und} \quad c_{\text{ALG}} \in C.$$

Im Folgenden geben wir eine kurze Übersicht über die Beispiele, die wir im Folgenden behandeln werden.

Job-Scheduling

Dieses Beispiel ist dem Beispiel im Kapitel “Approximationsalgorithmen” sehr ähnlich. Wir haben

- **Maschinen** M_1, \dots, M_m
- **Anfrage**: Job J_i , benötigt Zeit $t_i \geq 0$
- **Antwort**: Zuordnung von J_i zu M_j

Wieder stellt sich die Frage, wie sich der Makespan (also das Intervall zwischen Start und Fertigstellen der letzten Maschine) minimieren lässt. Diese Entscheidung muss hier für jedes J_i ohne Kenntnis über die zukünftigen Jobs passieren.

Skiausleihe

Die Situation ist hier folgende: Man befindet sich im Skilanglauf und solange das Wetter gut ist, lautet jeden Morgen die **Anforderung** “Ski ausleihen!”. Sobald das Wetter allerdings schlecht ist, lautet die **Anforderung** “Heimfahren!”.

Es sind zwei **Antworten** möglich:

- Ski für einen Tag ausleihen: Kosten k Euro
- Ski kaufen: Kosten K Euro ($K \gg k$)

Was muss nun getan werden, um die Gesamtkosten klein zu halten? Diese Entscheidung muss ohne Kenntnis des zukünftigen Wetters gemacht werden!

Speicherverwaltung

“Kosten minimieren” bedeutet im Speicherverwaltungskontext meist, die Anzahl an Cache Misses zu minimieren. Welche Verdränungsstrategie minimiert die Kosten hier am besten? Und wie kann man am besten zu verdrändende Seiten auswählen, ohne Kenntnis über zukünftige Anforderungen zu haben?

Auswahl von Experten

Hier gibt es mehrere Runden, jede läuft wie folgt ab:

1. Jeder von n Experten gibt zu einer Frage eine Ja/Nein-Empfehlung ab. Diese sind im Allgemeinen nicht richtig.
2. Man trifft seine eigene Ja/Nein-Entscheidung zu derselben Fragestellung.
3. Es wird mitgeteilt, welche Entscheidung richtig gewesen wäre.

Wie lässt sich hier die Anzahl an Fehlentscheidungen minimieren?

Selbstorganisierende Datenstrukturen

Hier haben wir eine einfach verkettete Liste und erhalten als **Anforderung** das Element x in der Liste. Die dabei entstehenden Kosten sind x . Als **Reaktion** kann entweder

- ohne weitere Kosten das angefragte Element weiter nach vorne gerückt werden, oder
- mit Kosten von 1 zwei aufeinanderfolgende Elemente vertauscht werden.

Welche Listen-Verwaltung minimiert hier die Kosten? Wie kann ohne Kenntnis über zukünftige Anforderungen sinnvoll umgeordnet werden?

15.2 Job-Scheduling

Wir nehmen den listScheduling-Approximationsalgorithmus und bauen ihn in einen Online-Algorithmus um:

```
LISTSCHEDULING( $n, m, t_1 \dots n$ )
each  $L_i := 0$  // load of machine  $1 \leq i \leq m$ 
each  $S_j := 0$  // machine for job  $1 \leq j \leq n$ 
for each  $j$  in range( $1, n$ ) do
    pick  $k$  from  $\{i : L_i \text{ is currently minimal}\}$ 
     $S_j := k$ 
     $L_k := L_k + t_j$ 
return  $S$ 
```

Frühere Analysen ergeben, dass der Wettbewerbsfaktor höchstens 2 ist. Der Online-Algorithmus sieht nun so aus:

```
LISTSCHEDULING( $m$ )
each  $L_i := 0$  // load of machine  $1 \leq i \leq m$ 

for each  $t_j$  in  $\sigma$  do
    pick  $k$  from  $\{i : L_i \text{ is currently minimal}\}$ 
     $a_j := k$ 
     $L_k := L_k + t_j$ 
    assign job to machine  $a_j$ 
```

15.3 Skiausleihe

Wir haben oben bereits diskutiert, was hier das Problem ist.

Die Optimalkosten sind offensichtlich

- Falls Urlaubsdauer $t \leq \frac{K}{k}$ Tage: t mal ausleihen \Rightarrow Kosten tk
- Falls Urlaubsdauer $t > \frac{K}{k}$ Tage: Ski sofort kaufen \Rightarrow Kosten K

Der Wettbewerbsfaktor ist hier 2. Optimal ist es, an Tag $\frac{K}{k}$ die Ski zu kaufen.

15.4 Speicherverwaltung

Wir werden uns die folgenden Speicherverwaltungsprobleme anschauen:

- *Offline* (lfd ist optimal)
- *Deterministisch* (bestenfalls k -kompetitiv)
- *Deterministisch*: lru ist k -kompetitiv
- *Resource Augmentation*: (h, k) -Seitenwechsel
- *Randomisiert* (randMark ist $2H_k$ -kompetitiv)

Wir betrachten hier stets einen Speicher der Größe K und einen Cache der Größe k ($K \gg k$).

Offline – lfd ist optimal

lfd (*longest forward distance*) verdrängt den Eintrag, der am weitesten in der Zukunft benötigt wird. Dieser Algorithmus ist offensichtlich optimal aber auch offensichtlich nicht möglich.

3	1	5	3	2	4	1
1	4	2				
3	1	5	3	2	4	1
1	3	2				

Abbildung 15.1. Eingabe und Cache vor und nach Bearbeitung der ersten Anforderung. Es wird die 4 ersetzt, weil sie erst am weitesten in der Zukunft wieder benötigt wird.

Deterministisch – bestenfalls k -kompetitiv

Die vier üblichen Algorithmen sind

- fifo (*first in first out*),
- lifo (*last in first out*),
- lru (*least recently used*),
- lfu (*least frequently used*).

lifo und lfu sind nicht kompetitiv, lru und fifo sind k -kompetitiv. In der Praxis wird üblicherweise lru verwendet.

Es gilt folgender Satz:

Satz 15.4.1. Jeder deterministische Online-Algorithmus für das Seitenwechselproblem mit Cachegröße k hat einen Wettbewerbsfaktor $c \geq k$. k ist also eine untere Schranke für den Wettbewerbsfaktor.

Deterministisch: lru ist k-kompetitiv

Wir können zeigen, dass lru k -kompetitiv ist.

Resource Augmentation

Wir betrachten das (h, k) -Seitenwechselproblem: hier wird der Online-Algorithmus alg_k mit der Cachegröße k mit lfd_h mit Cachegröße $h < k$ verglichen.

Wir benötigen folgende Definition:

Definition 15.4.2. Ein Online-Algorithmus heißt **konservativ**, falls er bei Andorderungsfolgen mit höchstens k verschiedenen Seiten höchstens k Cache-Misses hat.

Beispiele hierfür sind lru und fifo.

Es gilt:

Jeder konservative Online-Algorithmus ist $\frac{k}{k - h + 1}$ -kompetitiv. Dabei ist h die Größe des Caches.

Appendices

A: MATHEMATISCHE GRUNDLAGEN

A.1 Amortisierte Analyse

Amortisierte Laufzeiten stellen eine Worst-Case-Analyse für eine Sequenz von Operationen dar.

A_{Op} sind die amortisierten Kosten einer Operation $Op \in O$. T_{Op} sind die tatsächlichen Kosten einer Operation $Op \in O$. Die amortisierten Kosten sind korrekt, wenn für konstantes c gilt:

$$\sum_{1 \leq i \leq n} T_{Op_i} \leq c + \sum_{1 \leq i \leq n} A_{Op_i} \quad (\text{A.1})$$

Bankkonto- oder Potentialmethode

Finde ein Gehalt pro Operation, welches eine Obergrenze für die Kosten der Sequenz der Operationen darstellt.

Aggregatmethode

Schätzt $\sum_{1 \leq i \leq n} T_{Op}^i$ direkt ab.

Pseudotree

Ein zusammenhängender Pseudoforest, d.g. ein Graph der maximal einen Kreis enthält. Vom Kreis ausgehend sind es Trees.

A.2 Komplexitätsklassen

- P
- NP
- NP -vollständig: Menge der Probleme die in NP liegen und NP -schwer sind.
- NP -schwer oder NP -hart: Menge der Probleme auf die sich NP -Probleme reduzieren lassen.

A.3 Hyperwürfel

Ein Hyperwürfel der Dimension n entsteht durch Verbindung zweier $n-1$ -dimensionaler Würfel, wobei die äquivalenten Knoten jeweils miteinander verbunden werden. Ein Hyperwürfel in Dimension 0 ist ein einziger Knoten.

B: ALGORITHM BASICS

Komplexitätsanalyse:

- $O(n)$: Worst case
- $\Theta(n)$: Average case
- $\Omega(n)$: Best case

Präfixminima:

Für Permutationen in zufälliger Reihenfolge ist die Wahrscheinlichkeit, dass es k Präfixminima gibt gleich der Harmonischen Zahl H_k .

Durch

$$\ln k \leq H_k \leq 1 + \ln k$$

wissen wir, dass $H_k \approx \ln k$.