Effiziente Algorithmen im WS 2021/22

Prof. Martin Dietzfelbinger

Master Informatik, 1. Sem., und Interessent/inn/en, 5 LP

Vorlesung: Freitag, 11:00–12:30 Uhr, LdV-Hs 1, Prof. Dietzfelbinger

Übungen (b.a.w. MD oder online):
Donnerstag, 17:00–18:30 Uhr, Sr HU 210, (Beginn: 21.10.2020)

Kommunikation erfolgt über Moodle

Materialien: Skript (kapitelweise auf Moodle)

Medien: Tafelanschrieb und Projektion

Prüfung: Mündliche Prüfung, 30 Minuten.

Inhalt

1. **Flussprobleme und ihre Algorithmen:** Ford-Fulkerson-Methode, Algorithmus von Edmonds/Karp, Sperrflussmethode (Algorithmus von Dinitz), Preflow-Push
2. **Matchingprobleme und ihre Algorithmen:** Kardinalitätsmatching, Lösung über Flussalgorithmen, Algorithmus von Hopcroft/Karp; gewichtetes Matching: Auktionsalgorithmus, Ungarische methode; Stabile Paarungen: Satz von Kuhn/Munkres, Algorithmus von Gale/Shapley.
3. **Amortisierte Analyse von Datenstrukturen:** Ad-Hoc-Analyse, Bankkontomethode, Potenzialmethode.
4. **Adressierbare Priority Queues:** Fibonacci-Heaps.
5. **Textsuche:** Randomisiertes Verfahren; Algorithmus von Knuth/Morris/Pratt, Algorithmus von Aho/Corasick, Algorithmus von Boyer/Moore.

Literatur

Neben Vorlesungsskript:

- J. Kleinberg, E. Tardos: Algorithm Design, Pearson Education, 2005
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: Algorithmen – eine Einführung, 4. Auflage 2017 (Auch auf Englisch)
- M. Dietzfelbinger, K. Mehlhorn, P. Sanders: Algorithmen und Datenstrukturen – Die Grundwerkzeuge, Springer, 2014
- T. Ottmann, P. Widmayer: Algorithmen und Datenstrukturen, 6. Auflage., Springer-Vieweg 2017
- S. Dasgupta, C. Papadimitriou, U. Vazirani: Algorithms, McGraw-Hill 2007
- V. Heun, Grundlegende Algorithmen: 3. Auflage, Springer 2015 (E-Book)
- R. Sedgewick, K. Wayne: Algorithmen – Algorithmen und Datenstrukturen, 4. Auflage, Pearson 2014
- R. K. Ahuja, T. L. Magnanti, J. B. Orlin: Network Flows, Prentice Hall, 1993
- S. O. Krumke, H. Noltemeier: Graphentheoretische Konzepte und Algorithmen, Teubner, 2005
- M. Crochemore, W. Rytter: Jewels of Stringology, World Scientific, 2003

(M. Dietzfelbinger, aktualisiert 6. November 2021)

1 Maximierung von Flüssen in Netzwerken

Situation 1: Wir betrachten ein System von Leitungen für Flüssigkeiten (z. B. Wasser). Jede Leitung ist eine Einbahnstraße; verschiedene Leitungen treffen sich in Knoten. In einen der Knoten kann man Wasser einleiten („Quelle“, engl.: *source*), an einem anderen kann man es entnehmen („Senke“, engl.: *sink* (heißt auch Ausguss)). Die anderen Knoten sind dicht, d. h. es kann nichts hinzugefügt und nichts entnommen werden. Jede Leitung hat ein maximales Durchleitungsvermögen, Volumen pro Zeiteinheit, gemessen z. B. in m^3/s. Wir wollen einen kontinuierlichen Strom organisieren, in dem pro Zeiteinheit möglichst viel Wasser eingespeist und entnommen wird. Wieviel Wasser (pro Zeiteinheit) sollte durch jede der Leitungen fließen? (Illustration: Abb. 1, wobei man „Bahnhof“ als „Quelle“ und „Stadion“ als „Senke“ liest.)

Situation 2: („Fanströme“) In Dortkirchen gibt es ein Straßensystem und 60 000 Fußballfans, die am Bahnhof ankommen und zu Fuß zum Stadion wollen. Für jede Straße ist eine maximale Gesamtanzahl an Personen festgelegt, die sie passieren können. Wie soll die Polizei die Menschenmassen aufteilen und lenken, so dass alle ankommen? (Illustration: Abb. 1, Kapazitäten in Zehntausend.)

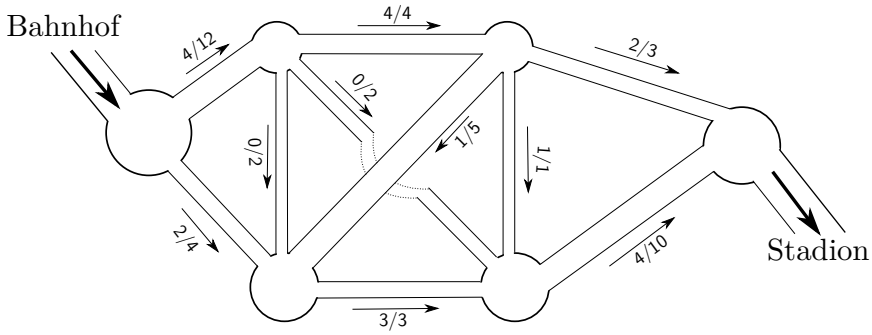


Abbildung 1: Beispiel Fanströme. Aufteilung der 60 000 Fussballfans gemäß der Personenbeschränkung der Straßen.

In diesem Abschnitt stellen wir diese Problemstellung abstrakt dar und betrachten effiziente Algorithmen, die sie lösen.

1.1 Netzwerke und Flüsse

Definition 1.1.1. *Ein* ***Flussnetzwerk*** **(*FNW*)** *ist ein gerichteter Graph (Digraph)* $G = (V, E, q, s, c)$ *mit Zusatzinformationen, wobei* (V, E) *Digraph ist,* $q, s \in V$ (***Quelle*** *und* ***Senke***) *ausgezeichnete Knoten sind und* $c\colon E \to \mathbb{R}_0^+$ *eine Funktion ist (die für jede Kante* e *eine* ***Kapazität*** $c(e) \geq 0$ *festlegt).*

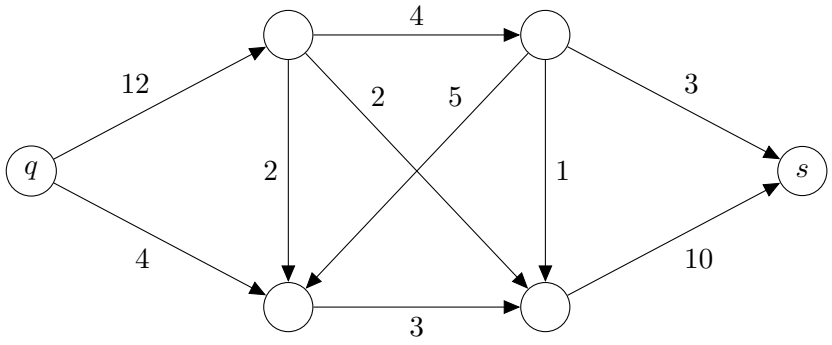


Abbildung 2: Das Flussnetzwerk zum Fanströmeproblem in Abbildung 1.

O. B. d. A. dürfen wir annehmen: Jeder Knoten $v \in V$ ist von q aus erreichbar, und von jedem Knoten $v \in V$ aus ist s erreichbar. (Eventuell vorhandene andere Knoten spielen in unseren Algorithmen keine Rolle. Daher kann man sie gleich weglassen. Die erreichbaren Knoten bestimmt man durch Tiefen- oder Breitensuche von q aus bzw. durch Suche von s aus im Umkehrgraphen G^{R}.) Oft nimmt man zudem an, dass q keine eingehenden Kanten und s keine ausgehenden Kanten hat. (Keiner der Algorithmen, die wir betrachten, benutzt solche Kanten. Daher kann man sie gleich weglassen.)

O. B. d. A. nehmen wir weiter an: (V, E) enthält keine Paare entgegengesetzter Kanten: $(u, v) \in E \Rightarrow (v, u) \notin E$.
(Wenn in einem FNW Kanten (u, v), (v, u) mit $c(u, v), c(v, u) > 0$ vorhanden sind, ersetzt man (v, u) durch zwei Kanten $(v, w), (w, u)$ mit $c(v, w) = c(w, u) = c(v, u)$, wobei w ein neuer Knoten ist. Wenn die Berechnung des maximalen Flusses (s. u.) beendet ist, kann man den neuen Knoten wieder eliminieren.)

Notation: $n = |V|$, $m = |E|$ stets. Mit den gemachten Annahmen gilt dann: $n - 1 \leq m \leq n(n-1)/2$.

Definition 1.1.2. *Ein **(zulässiger) Fluss** in einem Flussnetzwerk $G = (V, E, q, s, c)$ ist eine Funktion $f\colon E \to \mathbb{R}_0^+$, die jeder Kante e einen **Flusswert** $f(e)$ zuordnet, wobei folgende Bedingungen eingehalten werden:*

(i) **(Flusserhaltung in jedem Knoten, Kirchhoffsches Gesetz)** *Für jedes $v \in V$ betrachten wir, wieviel Fluss in v hineingeht und wieviel aus v heraus:*

$$f_{\mathrm{In}}(v) := \sum_{(u,v)\in E} f(u,v), \qquad f_{\mathrm{Out}}(v) := \sum_{(v,u)\in E} f(v,u).$$

Wir verlangen: $\forall v \in V - \{q, s\}\colon f_{\mathrm{In}}(v) = f_{\mathrm{Out}}(v)$.

(ii) **(Einhalten der Kapazitäten)** $\forall e \in E\colon 0 \le f(e) \le c(e)$.

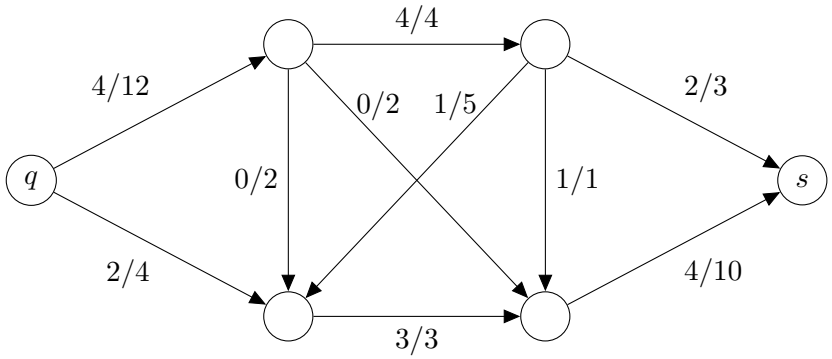
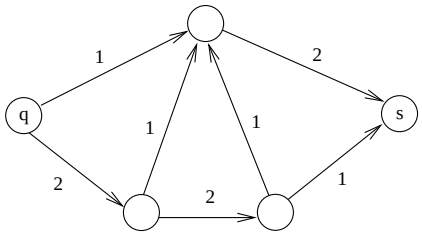


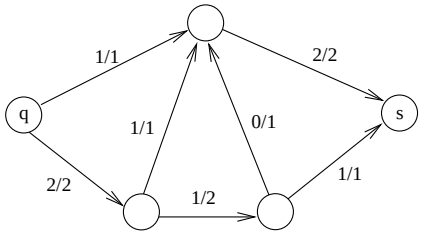
Abbildung 3: Ein zulässiger Fluss im Flussnetzwerk von Abbildung 2 mit Wert 6. Wir nutzen hierbei die Notation $f(e)/c(e)$ für die Darstellung des Flusses $f(e)$ über eine Kante e mit Kapazität $c(e)$.

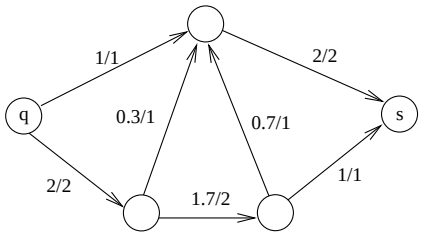
Definition 1.1.3. *Der **Wert** eines Flusses f ist $w(f) = f_{\mathrm{Out}}(q)$.*
(Sollte q eingehende Kanten haben, haben diese stets Flusswert 0.)

Problem: Zu gegebenem FNW G finde einen Fluss f mit möglichst großem $w(f)$.

Es wird sich herausstellen, dass es stets einen Fluss gibt, der den eindeutig bestimmten größtmöglichen Wert realisiert. Dies erscheint einfach, wenn die Kapazitäten ganzzahlig sind. Dann gibt es nur endlich viele legale Flüsse mit ganzzahligen $f(u, v)$, darunter also auch einen mit maximalem Wert. Das folgende Bild zeigt aber, dass es auch maximale Flüsse geben kann, die nicht-ganzzahlige Flusswerte benutzen.







Bei anderen Problemstellungen (Lineare Programmierung, Rucksack usw.) kann es durchaus sein, dass es eine solche „fraktionale“ Lösung gibt, die besser als alle ganzzahligen Lösungen ist. Man muss also noch überlegen, weshalb kein Fluss mit nichtganzzahligen Werten $f(u, v)$ besser sein kann als der beste ganzzahlige Fluss. Wenn die Kapazitäten rational sind, können wir sie mit dem Hauptnenner durchmultiplizieren, ohne das Problem wesentlich zu verändern, und erreichen so die Situation ganzzahliger Kapazitäten. Ganz unklar ist die Situation vorerst bei beliebigen reellen Kapazitäten. Die Existenz eines optimalen Flusses in diesem Fall beweisen wir später (in Abschnitt 1.3).

Bemerkung. (Für Studierende mit Kenntnissen in höherdimensionaler Analysis.) Die Menge der Tupel $f = (f(e))_{e \in E} \in \mathbb{R}^{|E|}$, die (ii) einhalten, ist abgeschlossen und beschränkt, also kompakt. Die Bedingungen (i), also die Flusserhaltungs-Gleichungen, definieren abgeschlossene Mengen, also ist die Menge aller Flüsse ebenfalls kompakt. Die Bewertungsfunktion $f \mapsto w(f)$ ist stetig. Also nimmt sie irgendwo im zulässigen Bereich ihr Maximum an.

1.2 Die Ford-Fulkerson-Methode und das MaxFlow-MinCut-Theorem

(L. R. Ford Jr. und D. R. Fulkerson, 1956.)

Grundidee: Iterative Verbesserung des Flusses, bis man den maximalen Fluss erreicht.

Grundvoraussetzung für das Gelingen: Die Kapazitäten sind ganzzahlig. (Oder rational: mit Hauptnennertrick auf ganzzahlige Kapazitäten umrechnen!)

Gegeben: Fluss f zu einem Flussnetzwerk G. Kann man diesen Fluss verbessern? Kann man noch mehr Fluss von q nach s schicken?

Definition 1.2.1. *Gegeben sei ein FNW G mit Fluss f.*

(a) *Wir definieren **Restkapazitäten** für Paare $(u, v) \in V \times V$, $u \neq v$.*

$$\mathrm{rest}_f(u,v) := \begin{cases} c(u,v) - f(u,v) & \text{, falls } (u,v) \in E \\ f(v,u) & \text{, falls } (v,u) \in E \\ 0 & \text{, sonst.} \end{cases}$$

(b) *Das **Restnetzwerk (RNW)** G_f hat Knotenmenge V, Kantenmenge $E_f = \{(u,v) \mid \mathrm{rest}_f(u,v) > 0\}$ und Kapazitäten $\mathrm{rest}_f(u,v)$ auf Kante $(u,v) \in E_f$.*

Wenn $(u,v) \in E_f$, gilt $(u,v) \in E$ oder $(v,u) \in E$, also ist $|E_f| \leq 2|E|$.

Man beachte, dass G_f im strengen Sinn kein Flussnetzwerk ist, da es entgegengesetzte Kanten haben kann und der Eingangsgrad von q und der Ausgangsgrad von s größer als 0 sein kann. Dennoch kann man definieren, was ein Fluss in G_f ist: Eine Funktion $f' \colon E_f \to \mathbb{R}_0^+$, die die Kapazitätsschranken $0 \leq f'(u,v) \leq \mathrm{rest}_f(u,v)$ und die Kirchhoffregel in $v \in V - \{q,s\}$ einhält *und* für die $f'(u,v) = 0$ oder $f'(v,u) = 0$ gilt, falls beide Kanten in E_f sind.

Kanten $(u,v) \in E \cap E_f$ heißen **Vorwärtskanten**. Die Interpretation der Restkapazität ist hier einfach: Man könnte noch **zusätzlich** Fluss im Wert von bis zu $c(u,v) - f(u,v)$ über diese Kante schicken.

Kanten $(u,v) \in E_f$ mit $(v,u) \in E$ heißen **Rückwärtskanten**. Hier kann man gewissermaßen den Fluss von u nach v erhöhen, indem man den Fluss von v nach u *verringert*, um einen Wert von bis zu $f(v,u)$.

Definition 1.2.2. *Gegeben sei ein FNW G mit Fluss f sowie das zugehörige Restnetzwerk G_f. Ein **flussvergrößernder Weg** (kurz: **fv Weg** oder **fvW**, engl.: **augmenting path**) ist ein einfacher Weg p von q nach s in G_f.*

*Die **Kapazität** eines solchen fvW p ist $c(p) := \min\{\mathrm{rest}_f(e) \mid e \text{ liegt auf } p\}$.*

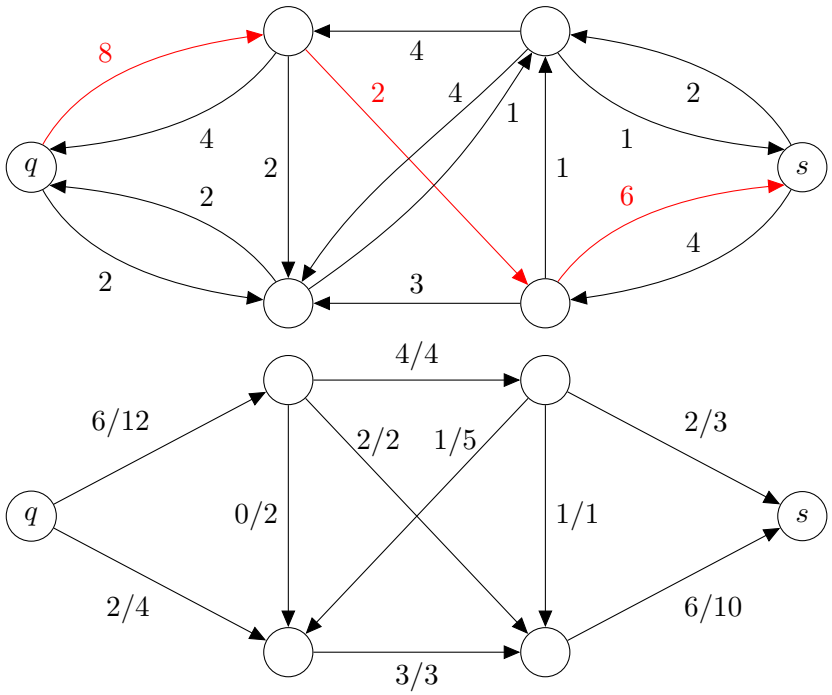


Abbildung 4: Oben: Das Restnetzwerk zum Fluss in Abbildung 3. Wir vergrößern den Fluss aus Abb. 3 entlang des rot markierten Weges p um $c(p) = 2$. Unten: Das Resultat der Flussvergrößerung. Der Fluss hat nun einen Wert von 8.

Bemerkung: Weil in G_f nur Kanten mit positiver Restkapazität vorkommen, gilt $c(p) > 0$.

Wenn p ein flussvergrößernder Weg ist, kann man auf G_f einen Fluss f_p mit Wert $w(f_p) = c(p)$ definieren: Für $(u, v) \in E_f$ setze:

$$f_p(u,v) := \begin{cases} c(p) & \text{, falls } (u,v) \text{ auf } p \text{ liegt} \\ 0 & \text{, sonst.} \end{cases}$$

Weil p ein einfacher Weg ist, können nie sowohl (u, v) als auch (v, u) auf p liegen.

Wir behaupten, dass die Kirchhoff-Bedingung in $v \in V - \{q, s\}$ erfüllt ist und dass $w(f_p) = c(p)$ gilt. Die erste Behauptung verifiziert man dadurch, dass man kontrolliert, dass in jeden Knoten im Inneren von p genau eine Kante von p hineinführt und genau eine aus ihm herausführt, beide haben Flusswert $c(p)$. Aus der Quelle führt eine Kante mit Flusswert $c(p)$ heraus, also ist $w(f_p) = c(p)$.

Immer wenn wir einen Fluss f in G und einen Fluss f' in G_f mit $w(f') > 0$ haben, der

keine Paare entgegengesetzter Kanten benutzt, können wir den Fluss f vergrößern: Für $(u,v) \in E$ setze

$$(f+f')(u,v) := \begin{cases} f(u,v) + f'(u,v) & \text{, falls } f'(u,v) > 0 \text{ ,} \\ f(u,v) - f'(v,u) & \text{, falls } f'(v,u) > 0 \text{ ,} \\ f(u,v) & \text{, sonst .} \end{cases} \tag{1}$$

Ist diese Definition sinnvoll („wohldefiniert“) und liefert sie einen Fluss für G? Wenn $f'(u,v) > 0$ gilt, muss $0 < f'(u,v) \leq \text{rest}_f(u,v) = c(u,v) - f(u,v)$ sein, also ist $0 < f(u,v) + f'(u,v) \leq c(u,v)$. Wenn $f'(v,u) > 0$ gilt, muss $0 < f'(v,u) \leq \text{rest}_f(v,u) = f(u,v)$ sein, also ist $0 \leq f(u,v) - f'(v,u) < f(u,v) \leq c(u,v)$. Die beiden Fälle können nicht gleichzeitig auftreten, weil f' keine Paare entgegengesetzter Kanten benutzt.

Die neuen Werte halten also die Kapazitätsgrenzen ein. Die Kirchhoffregeln sind erfüllt, weil sowohl f als auch f' sie erfüllen. Die Gleichheit $w(f+f') = w(f) + w(f')$ ist offensichtlich.

Die **Ford-Fulkerson-Methode** besteht nun einfach in Folgendem:

(1) Definiere $f := f_0$ für den „Null-Fluss“ f_0, mit $f_0(u,v) = 0$ für alle $(u,v) \in E$.

(2) **repeat**

  (i) Berechne RNW G_f.

  (ii) Wenn es in G_f keinen fvW mehr gibt, **return** f; sonst:

  (iii) Berechne fvW p und Fluss f_p, setze $f := f + f_p$.

Über diese Methode können wir folgendes feststellen:

Lemma 1.2.3. *Sei G FNW mit ganzzahligen Kapazitäten. Dann gilt:*

(a) *Alle Werte $f(u,v)$ und alle $c(p)$, die jemals auftreten, sind ganzzahlig.*

(b) *Sei $C_G := \sum_{(q,v)\in E} c(q,v)$ (die Kapazität aller aus q hinaus führenden Kanten). Dann führt die FF-Methode auf G maximal C_G Schleifendurchläufe aus und gibt einen Fluss f aus, für den es in G_f keinen fvW gibt. Es gilt $w(f) \leq C_G$.*

Beweis: (a) wird per Induktion über die Schleifendurchläufe bewiesen. Die wesentliche Feststellung ist, dass auf Zahlen nur die Operationen Addition, Subtraktion und Minimumsbildung angewendet wird, die die Ganzzahligkeit erhalten. (b) Wir beobachten den Wert $w(f)$ des Flusses im Verlauf des Algorithmus. Dieser ist anfangs 0 und erhöht sich in jedem Schleifendurchlauf um einen Wert $c(p) > 0$, der

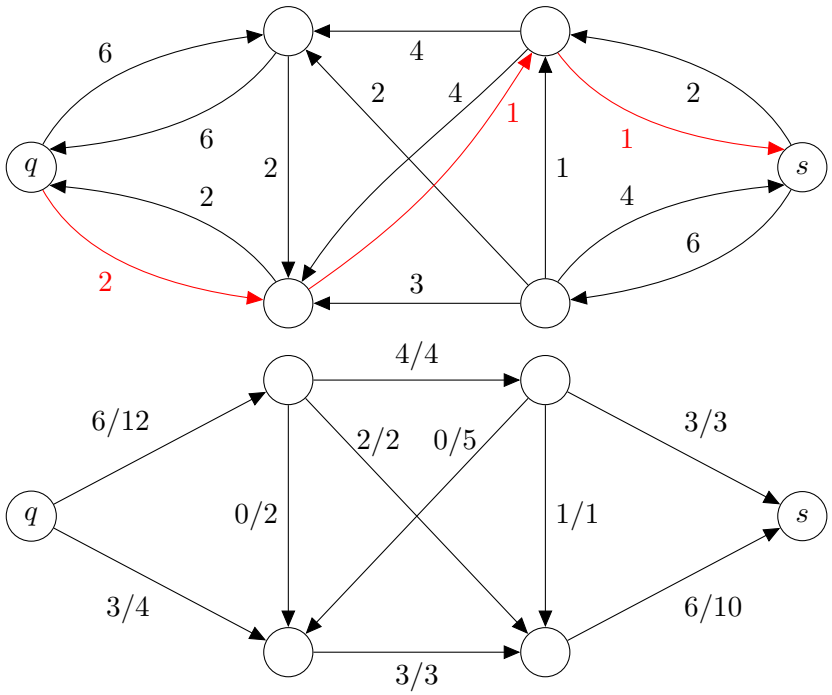


Abbildung 5: Das Restnetzwerk zum Fluss in Abbildung 4. Wir vergrößern den Fluss aus Abb. 4 entlang des rot markierten Pfades p um $c(p) = 1$. Das Resultat ist im unteren Bild zu sehen. Der Fluss hat nun einen Wert von 9.

das Minimum einer Menge von (positiven) Differenzen von ganzen Zahlen ist. Also gilt $c(p) \geq 1$. Größer als C_G kann $w(f)$ nicht werden, also endet die Schleife spätestens nach C_G Durchläufen. (Genauer: Es gibt höchstens $w(f)$ Durchläufe für den ganzzahligen Fluss f, mit dem die Schleife stoppt.) □

Die Schleife endet in einer Situation, wo kein Weg von q nach s in G_f gefunden wird. Das heißt, dass kein solcher Weg existiert. Aber was bedeutet es, wenn es in G_f keinen Weg von q nach s gibt? Wir vermuten, dass dann f maximal ist. Dies ist der wesentliche Inhalt des folgenden Satzes.

Achtung: Die folgenden Überlegungen benutzen die Ganzzahligkeit der Kapazitäten und Flüsse nicht. Die Ergebnisse gelten also allgemein.

Technisch benötigen wir noch einen zentralen Begriff. Ein ***Schnitt*** (engl. *cut*) zerlegt ein FNW in zwei Teilmengen, wobei q und s in verschiedenen Teilen liegen.

Definition 1.2.4. *Ein* ***Schnitt*** (Q, S) *in einem Flussnetzwerk* $G = (V, E, q, s, c)$ *ist eine Aufteilung der Knotenmenge* V *in zwei disjunkte Mengen* Q *und* S *(d. h.* Q *und*

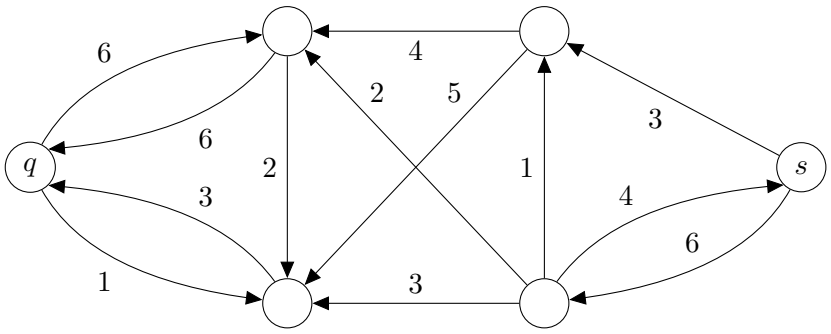


Abbildung 6: Das Restnetzwerk zum Fluss in Abbildung 5. Es existiert kein Weg von der Quelle zur Senke.

*S sind disjunkt und $Q \cup S = V$) mit $q \in Q$ und $s \in S$. Die **Kapazität** eines solchen Schnittes (Q, S) ist definiert als*

$$c(Q, S) := \sum_{\substack{(u,v)\in E,\\ u\in Q, v\in S}} c(u, v)$$

Zwischen Flusswerten und Schnittkapazitäten gibt es eine wesentliche Beziehung: Kein Fluss kann größer sein als die Kapazität eines Schnittes. Bei einem besonderen Schnitt ist dies klar: $Q = \{q\}$ und $S = V - \{q\}$. Die Kapazität ist $\sum_{(q,v)\in E} c(q, v)$, und $w(f) = \sum_{(q,v)\in E} f(q, v)$ kann wegen der Kapazitätsregel nicht größer als dieser Wert sein. Allgemein stellt man folgendes fest.

Lemma 1.2.5. *Sei G ein FNW, f ein beliebiger Fluss, (Q, S) ein beliebiger Schnitt. Wenn wir den **Fluss über den Schnitt** (Q, S) als*

$$f(Q, S) = \sum_{\substack{(u,v)\in E,\\ u\in Q, v\in S}} f(u, v) \quad - \sum_{\substack{(v,u)\in E,\\ v\in S, u\in Q}} f(v, u)$$

definieren[1]*, dann gilt*

$$w(f) = f(Q, S) \leq c(Q, S).$$

Beweis: Übung. (Die Gleichheit kann man gut durch Induktion über die Größe von Q beweisen. Die Ungleichung ist sehr leicht einzusehen.) □

Nun folgt der Hauptsatz über Flussalgorithmen.

[1]Positiv: was fließt von Q nach S? Negativ: was fließt von S nach Q?

Satz 1.2.6 (MaxFlow-MinCut-Theorem)**.** *Sei G ein FNW und f ein Fluss für G. Dann sind äquivalent:*

(i) *f ist maximal.*

(ii) *Das RNW G_f enthält keinen fvW.*

(iii) *Es gibt einen Schnitt (Q, S) mit $c(Q, S) = w(f)$.*

Beweis: Wir verwenden einen Ringschluss „(i) $\Rightarrow$ (ii) $\Rightarrow$ (iii) $\Rightarrow$ (i)“.

„(i) $\Rightarrow$ (ii)“: Wir beweisen die Kontrapositon. (Wenn (ii) falsch ist, ist auch (i) falsch.) Nehmen wir also an, dass das RNW G_f einen fvW p mit Kapazität $c(p) > 0$ hat. Dann ist, wie oben gesehen, $f + f_p$ ein Fluss mit Wert $w(f + f_p) > w(f)$, also ist f nicht maximal.

„(ii) $\Rightarrow$ (iii)“: (Dies ist das entscheidende Argument!) Nehmen wir an, dass es im RNW G_f keinen Weg von q nach s gibt. Wir **definieren** einen Schnitt:

$$\begin{aligned} Q &:= \{v \in V \mid v \text{ ist von } q \text{ aus in } G_f \text{ erreichbar}\}; \\ S &:= V - Q. \end{aligned}$$

Trivialerweise ist $q \in Q$; wegen der Annahme gilt $s \notin Q$, also $s \in S$.

Betrachte nun eine Kante $(u, v) \in E$ mit $u \in Q$ und $v \in S$. Nach der Definition von Q ist u in G_f von q aus erreichbar, aber v nicht. Daher ist $(u, v) \notin E_f$. Nach der Definition von E_f bedeutet dies, dass $f(u, v) = c(u, v)$ ist (die Kante (u, v) ist „gesättigt“). – Nun betrachte eine Kante $(v, u) \in E$ mit $v \in S$ und $u \in Q$. Wieder ist $(u, v) \notin E_f$, also muss $f(v, u) = 0$ sein, wieder nach der Definition von E_f.

Also gilt:

$$f(Q, S) = \sum_{\substack{(u,v)\in E,\\ u\in Q, v\in S}} f(u, v) \quad - \sum_{\substack{(v,u)\in E,\\ v\in S, u\in Q}} f(v, u) = \sum_{\substack{(u,v)\in E,\\ u\in Q, v\in S}} c(u, v) = c(Q, S).$$

Damit ist (iii) gezeigt.

„(iii) $\Rightarrow$ (i)“: Sei (Q, S) ein Schnitt mit $c(Q, S) = w(f)$, und sei f' ein beliebiger anderer Fluss. Dann gilt nach Lemma 1.2.5:

$$w(f') = f'(Q, S) \leq c(Q, S) = w(f).$$

Das heißt, dass kein Fluss einen größeren Wert als $w(f)$ haben kann, also dass f maximal ist. □

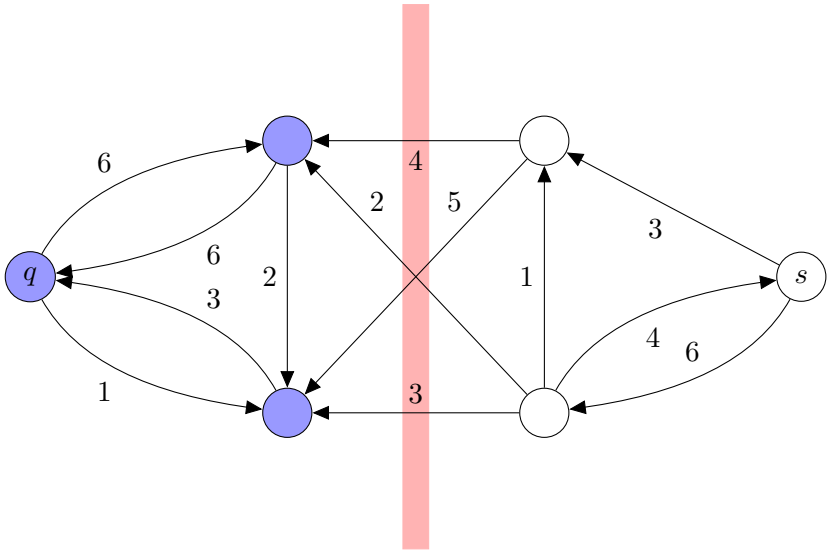


Abbildung 7: Das Restnetzwerk zum Fluss in Abbildung 5. Es existiert kein Weg von der Quelle zur Senke. Wie im Beweis von Satz 1.2.6 bestimmen wir den (Q, S)-Schnitt. Blau gefärbte Knoten sind in der Menge Q, alle anderen in der Menge S. Die Kapazität des Schnittes beträgt 9 (vgl. Abbildung 5).

Aus dem MaxFlow-MinCut-Theorem folgt sofort, dass in dem Moment, in dem die Ford-Fulkerson-Methode anhält, der Fluss f maximal ist, da Bedingung (ii) erfüllt ist. Diese Methode findet also einen maximalen Fluss, wenn die Kapazitäten ganzzahlig (oder rational) sind. Die Anzahl der Runden ist maximal C_G (oder genauer: $w(f)$ für den maximalen Fluss f). Jede Konstruktion eines Restnetzwerks benötigt Zeit $O(m)$, ebenso die Suche nach einem fvW (zum Beispiel mit Tiefensuche) und gegebenenfalls die Flussvergrößerung. Die gesamte Rechenzeit ist also $O(m \cdot C_G)$. Da die Größe der Kapazität (und nicht die Bitlänge der Zahldarstellung) in die Laufzeit eingeht, sprechen wir von einem *pseudopolynomiellen* Algorithmus.[2]

Beobachtung 1.2.7. *Wenn alle Kapazitäten ganzzahlig sind, dann gibt es einen maximalen Fluss f, in dem alle Flusswerte $f(u, v)$ ganzzahlig sind.* (*Grund: Die FF-Methode konstruiert einen solchen Fluss, vgl. Lemma 1.2.3.*)

Bemerkung zum Namen „MaxFlow-MinCut-Theorem“: Bisher haben wir nur nach einem „maximalen Fluss“ (also einem Fluss mit maximalem $w(f)$) gefragt. Die FF-

[2]Man kann eine Realisierung der FF-Methode finden, bei der die Rundenzahl durch $O(m \log C_G)$, also die Laufzeit durch $O(m^2 \log C_G)$ beschränkt ist. Wir werden allerdings sehr schnell noch bessere Algorithmen identifizieren.

Methode findet einen solchen Fluss (falls die Kapazitäten ganzzahlig sind). Genauso könnte man auch nach einem Schnitt (Q, S) fragen, der minimale Kapazität hat (also $c(Q, S) \leq c(Q', S')$ für alle Schnitte (Q', S') erfüllt). Überraschenderweise haben wir dieses Problem schon gelöst! Wenn die FF-Methode anhält, gibt es keinen fvW in G_f mehr. Wir betrachten den Schnitt (Q, S) im Beweisteil „(ii) $\Rightarrow$ (iii)“ und behaupten, dass er minimale Kapazität hat. Wieso? Betrachte einen beliebigen Schnitt (Q', S'). Nach Lemma 1.2.5 gilt $w(f) \leq c(Q', S')$, aber wir haben auch $c(Q, S) = w(f)$. Also gilt $c(Q, S) \leq c(Q', S')$. Daher hat (Q, S) minimale Kapazität. – Anders gesagt: Jeder Algorithmus, der einen maximalen Fluss berechnet, berechnet praktisch gleich einen minimalen Schnitt mit. Und: Der größtmögliche Wert eines Flusses ist gleich der kleinstmöglichen Kapazität eines Schnittes, oder: $\max\{w(f) \mid f \text{ Fluss}\} = \min\{c(Q, S) \mid (Q, S) \text{ Schnitt}\}$, oder: „MaxFlow = MinCut“. Diese Erscheinung ist charakteristisch für Probleme, die sich als „lineares Programm“ formulieren lassen. Das MaxFlow-Problem gehört in diese Klasse; das MinCut-Problem ist zum Flussproblem „dual“. (Details zur Dualität erfährt man in Veranstaltungen, die sich mit linearer Programmierung befassen.)

1.3 Der Algorithmus von Edmonds-Karp

Die Ford-Fulkerson-Methode kann auch für kleine Graphen sehr große Laufzeit haben, die mit dem Wert des maximalen Flusses wächst. Dies wird durch das folgende einfache *Beispiel* deutlich: $G = (V, E)$ hat 4 Knoten q, s, a und b, und Kanten $(q, a), (q, b), (a, s), (b, s)$ mit Kapazität M sowie eine Kante (a, b) mit Kapazität 1. Wenn man als fvW nie (q, a, s) oder (q, b, s) benutzt, sondern immer nur (q, a, b, s) bzw. (q, b, a, s), je nachdem, welcher gerade zur Verfügung steht, dann benötigt die Ermittlung des maximalen Flusses vom Wert $2M$ genau $2M$ Runden. Die ersten Schritte eines solchen Ablaufs sind in Abb. 8 dargestellt.

Es gibt mehrere Möglichkeiten, diese „Falle“ zu vermeiden, die durch das Zulassen völlig beliebiger flussvergrößernder Wege zustandekommt. Eine der einfachsten führt zum Algorithmus von Edmonds und Karp, dargestellt als Algorithmus 1.3.1. Dieser verlangt einfach, einen *möglichst kurzen* fvW zu wählen, also einen mit möglichst wenigen Kanten. Dies erreicht man am effizientesten dadurch, dass man den fvW im RNW mittels *Breitensuche* (Algorithmus BFS) vom Knoten q aus sucht und beim ersten Erreichen der Senke s aufhört.

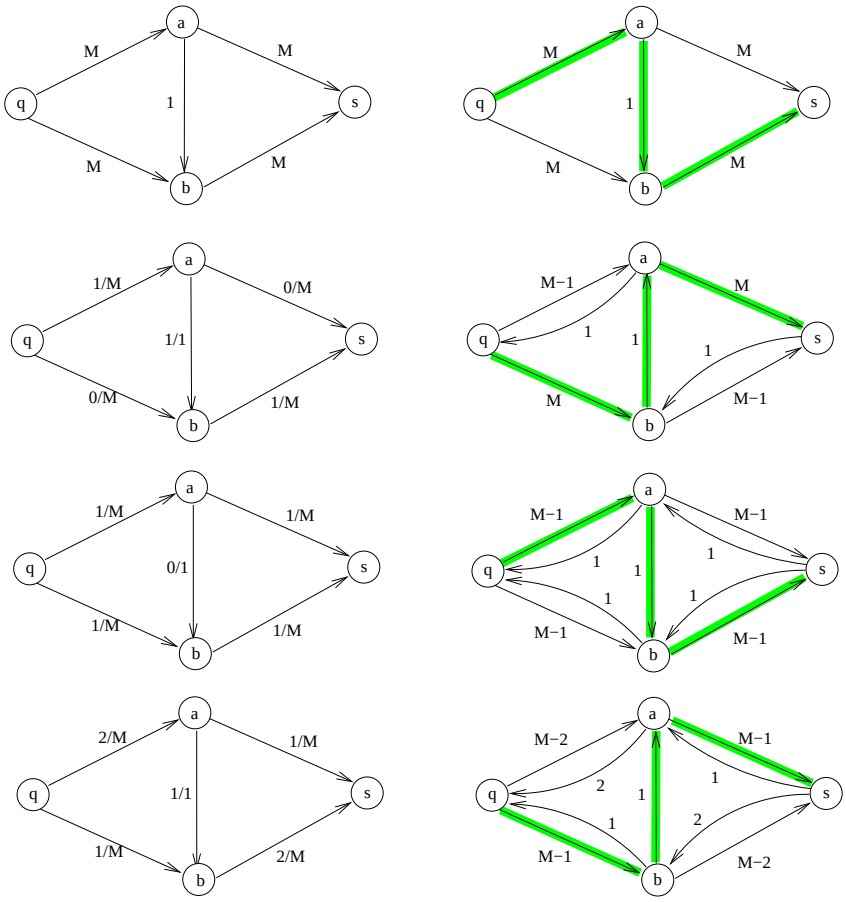


Abbildung 8: Langsame Flussvergrößerung mit der FF-Methode. Erste Zeile: Flussnetzwerk und Restnetzwerk mit FvW. Zweite und dritte Zeile: jeweils ein Fluss und das zugehörige RNW mit FvW. Letzte Zeile: Fluss mit Wert 3 nach drei Runden. Wenn man in derselben Weise fortfährt, erhält man den maximalen Fluss mit Wert $2M$ erst nach $2M$ Flussvergrößerungen. **(Bild korrigiert 19.10.2019)**

Algorithmus 1.3.1 (**Edmonds-Karp**)**.**

INPUT: Flussnetzwerk $G = (V, E, q, s, c)$

METHODE:

```
1   f := der Nullfluss;
2   repeat
3     Berechne das Restnetzwerk G_f;
4     Führe in G_f BFS mit Startknoten q aus; stoppe, sobald s erreicht;
5     Falls s nicht erreicht: return f;
6     Sonst: Der von der BFS gefundene Weg p von q nach s ist fvW;
7     f := f + f_p;   //    vergrößere Fluss entlang p
```

In Abb. 9 ist ein Beispieldurchlauf angegeben. Dabei sind nur die Flüsse im Netzwerk G gezeichnet – die Restnetzwerke muss man jeweils erschließen.

Wir analysieren nun die Laufzeit des Edmonds-Karp-Algorithmus.

Satz 1.3.2. *Sei G ein FNW. Dann hat der Algorithmus von Edmonds-Karp auf G Laufzeit $O(m^2n) = O(n^5)$. Dies gilt sogar, wenn die Kapazitäten beliebige Zahlen in $\mathbb{R}_0^+$ sind.*

Beweis: Jeder Aufbau des Restnetzwerks und jede Breitensuche kann in Zeit $O(m)$ bewerkstelligt werden. Wir müssen also nur zeigen, dass es maximal $O(nm)$ Iterationen der **repeat**-Schleife („*Runden*“) geben kann, bevor sie (über den **return**-Befehl) verlassen wird. Wir definieren dazu:

$$d_t(v) := \text{Abstand von } v \text{ von } q \text{ im RNW } G_f^{(t)} \text{ nach Runde } t, \text{ für } v \in V.$$

Auch hier bedeutet „Abstand“ natürlich die Länge eines Weges von q zu v mit möglichst wenigen Kanten. $d_t(v) = \infty$ heißt, dass es in $G_f^{(t)}$ keinen q-v-Weg gibt.

Der springende Punkt der Analyse ist, dass diese Abstände nie kleiner werden können.

Lemma 1.3.3. *Für jede Runde $t > 0$ und alle Knoten $v \in V$ gilt: $d_t(v) \geq d_{t-1}(v)$.*

Beweis: Wir betrachten $G_f^{(t-1)}$ und teilen V in die Breitensuche-Levels[3] ein:

$$V_\ell := \{v \in V \mid d_{t-1}(v) = \ell\}, \text{ für } \ell = 0, 1, 2, \ldots \text{ und } \ell = \infty.$$

Direkt aus der Definition folgt: Wenn $(u, v) \in E_f^{(t-1)}$, dann gilt $d_{t-1}(v) \leq d_{t-1}(u) + 1$. D. h.: Entlang von Kanten in $E_f^{(t-1)}$ steigt die Levelnummer um nicht mehr als 1 an. (Wenn $d_t(u) = \infty$, ist diese Ungleichung trivial, mit der Vereinbarung $\infty + 1 = \infty$.)

[3]Diese Levels muss der Algorithmus nicht berechnen; wir brauchen sie nur für die Analyse.

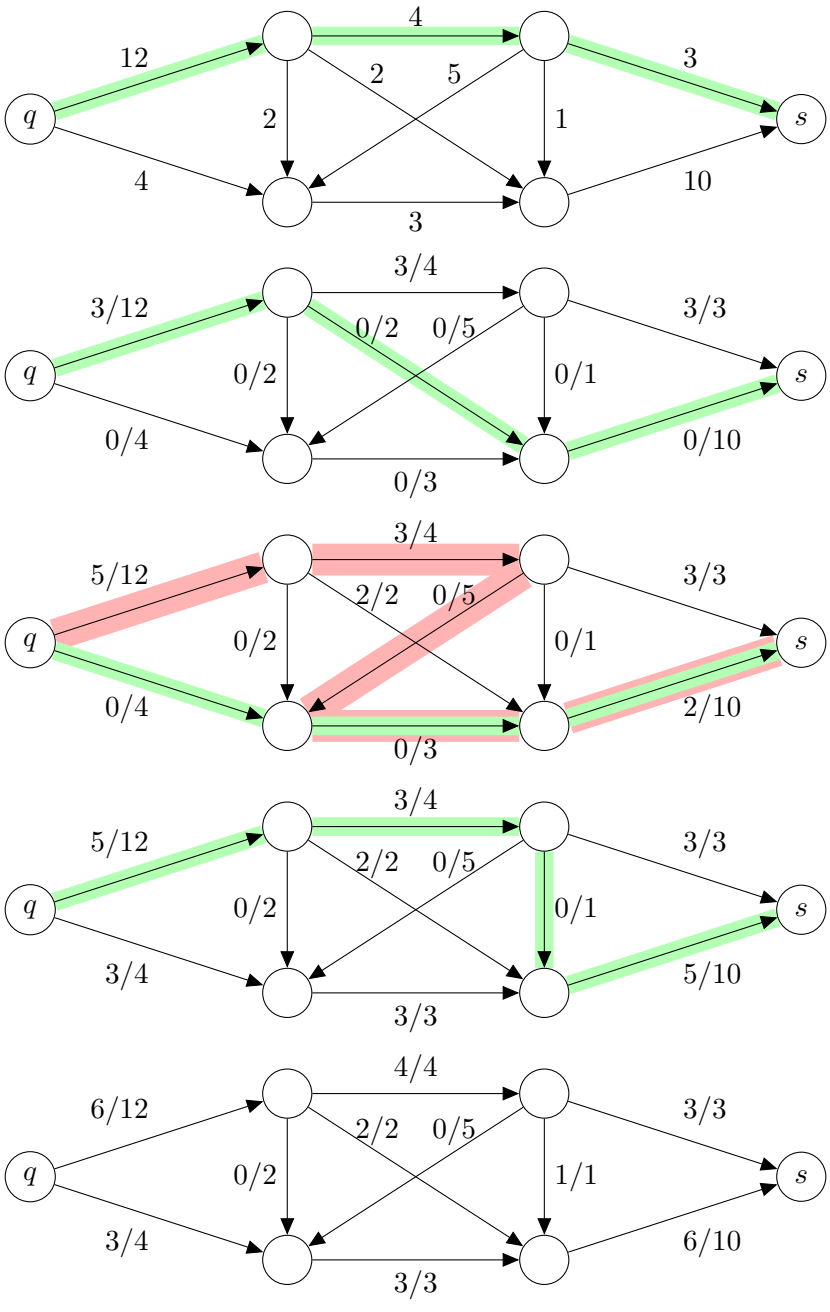


Abbildung 9: Ein Beispieldurchlauf des Edmonds-Karp-Algorithmus.

In Runde t wird der Fluss durch eine Flussvergrößerung $f := f + f_p$ entlang eines kürzesten fvW $p = (u_0, u_1, \ldots, u_L)$ mit $u_0 = q$ und $u_L = s$ vergrößert. Für jeden Knoten u_ℓ auf diesem Weg gilt $u_\ell \in V_\ell$. (Denn: Der Edmonds-Karp-Algorithmus schreibt vor, dass p ein *kürzester* fvW ist. Daher ist auch jedes Anfangsstück von p ein kürzester Weg zu seinem Endknoten u_ℓ.) Durch die Flussvergrößerung ändert sich das Restnetzwerk von $E_f^{(t-1)}$ zu $E_f^{(t)}$. Dabei können Kanten neu erscheinen. (Dass Kanten auch verschwinden, können wir hier ignorieren.) Wenn eine Kante (u, v) in $E_f^{(t)}$ neu erscheint, dann muss die Gegenkante (v, u) auf p liegen, also $v = u_{\ell-1}$ und $u = u_\ell$ gelten, für ein $\ell \in \{1, \ldots, L\}$. Das heißt $d_{t-1}(v) = \ell - 1 = d_{t-1}(u) - 1$: Entlang einer neuen Kante (u, v) sinkt die Levelnummer sogar strikt!

Wir erhalten: Entlang einer beliebigen Kante $(u, v) \in E_f^{(t)}$ (alt oder neu hinzugekommen) steigt die Levelnummer (bezüglich $V_0, V_1, \ldots$) nicht um mehr als 1 an.

Daraus folgt: (1) Für ein beliebiges $v \in V_\ell$, $\ell < \infty$, benötigt jeder Weg in $E_f^{(t)}$ von q ($\in V_0$) nach v mindestens ℓ Kanten, also gilt $d_t(v) \geq \ell = d_{t-1}(v)$. (2) Wenn $v \in V_\infty$, gibt es in $E_f^{(t)}$ überhaupt keinen Weg von q nach v, also gilt auch $d_t(v) = \infty$. □

Beweis von Satz 1.3.2 (Forts.): Wir betrachten ein festes Paar (u, v) mit $(u, v) \in E$ oder $(v, u) \in E$, das heißt, eine mögliche Kante in einem Restnetzwerk $G_f^{(t)}$. Die Kante kann im Verlauf des Algorithmus mehrfach im Restnetzwerk auftauchen und wieder verschwinden. Wir wollen zeigen, dass sie nicht allzu oft verschwinden kann. Dabei können wir $u \neq q$ voraussetzen, also $d_t(u) \geq 1$ stets, da Kanten (q, v) nur einmal verschwinden, aber nie wieder neu erscheinen können. (Kanten (v, q) können nicht auf dem fvW liegen.)

Wenn Kante (u, v) in Runde t aus dem RNW verschwindet, ist sie Flaschenhalskante, liegt also auf dem fvW p für Runde t. Sei $\ell = d_{t-1}(u)$ und $\ell + 1 = d_{t-1}(v)$. In einer späteren Iteration t' kann (u, v) wieder im Restnetzwerk auftauchen. Dazu muss (v, u) auf dem fvW p' für Runde t' liegen, also muss $d_{t'-1}(v) = \ell'$ und $d_{t'-1}(u) = \ell' + 1$ gelten, für ein ℓ'. Wenn nun in einer Runde $t'' > t'$ die Kante (u, v) erneut verschwindet, muss $d_{t''-1}(u) = \ell''$ und $d_{t''-1}(v) = \ell'' + 1$ gelten, für ein ℓ''. Da nach Lemma 1.3.3 der Abstand eines jeden Knotens von q von Runde zu Runde nicht sinken kann, gilt

$$\ell + 1 = d_{t-1}(v) \leq d_{t'-1}(v) = \ell' \text{ und } \ell' + 1 = d_{t'-1}(u) \leq d_{t''-1}(u) = \ell'',$$

also $d_{t''-1}(u) = \ell'' \geq \ell + 2 = d_{t-1}(u) + 2$. Bei jedem Verschwinden von (u, v) ist der Abstand von u von q also um mindestens 2 angestiegen.

Da wir mit $d_0(u) \geq 1$ beginnen, gilt $d_{t-1}(u) \geq 2k - 1$, wenn (u, v) in Runde t zum k-ten Mal verschwindet. Die maximale Distanz ist $n - 1$. Also gilt $d_{t-1}(u) \leq n - 1$, und es folgt $k \leq n/2$. Also kann Kante (u, v) nicht häufiger als $\lfloor \frac{1}{2} n \rfloor$-mal aus dem RNW verschwinden.

Nun beobachten wir, dass es in jeder Runde mindestens eine Flaschenhalskante gibt, dass also eine Kante aus dem RNW verschwindet, nämlich eine der Kanten e auf dem benutzten fvW p, die $c(p) = \text{rest}_f(e)$ erfüllt. Jede der $2m$ Kanten (u, v) mit $(u, v) \in E$ oder $(v, u) \in E$ kann aber nur $\lfloor \frac{1}{2}n \rfloor$-mal Flaschenhalskante sein, also gibt es maximal $2m\lfloor \frac{1}{2}n \rfloor \leq mn$ Runden. □

Bemerkung 1: Die Schleife im Algorithmus kann nur auf eine Weise verlassen werden, nämlich indem festgestellt wird, dass im RNW G_f die Senke nicht mehr erreichbar ist. Das MaxFlow-MinCut-Theorem (Satz 1.2.6) besagt dann, dass der nun vorliegende Fluss f maximal ist.

Bemerkung 2: Durch die kleine Änderung an der Ford-Fulkerson-Methode, die zum Algorithmus von Edmonds-Karp führte, haben wir einen großen Qualitätssprung gemacht, in zweierlei Hinsicht: (i) Die Rechenzeit ist unabhängig von der Größe der Kapazitäten (solange man annimmt, dass man in konstanter Zeit addieren, subtrahieren und das Maximum zweier Zahlen bilden kann). (ii) Der neue Algorithmus funktioniert auch, wenn die Kapazitäten beliebige reelle Zahlen sind. Die Ganzzahligkeitsannahme ist überflüssig geworden.

Bemerkung 3: Weil der Algorithmus von Edmonds-Karp nur eine spezielle Variante der FF-Methode ist, hat er alle Eigenschaften dieser Methode, insbesondere liefert er einen ganzzahligen optimalen Fluss, wenn die Kapazitäten ganzzahlig sind.

Im nächsten Abschnitt wollen wir noch Verfahren zur Laufzeitverbesserung kennenlernen.

1.4 Niveaunetzwerke und Sperrflussmethode

Es kostet $O(m)$ Zeit, ein Restnetzwerk G_f aufzubauen und darin einen kürzesten q-s-Weg zu suchen. Der Algorithmus von Edmonds und Karp nutzt dieses dann eigentlich nur sehr wenig: Er konstruiert einen einzigen flussvergrößernden Weg in G_f, dann wird G_f durch ein neues Restnetzwerk ersetzt und es wird erneut nach einem fvW gesucht. Die nächste Idee zur Laufzeitverbesserung besteht darin, auf der Basis *eines* Restnetzwerkes gleich einen größeren Flussvergrößerungs-Sprung zu machen, indem man nicht nur einen einzelnen Weg betrachtet, sondern in einer Runde den Fluss gewissermaßen entlang mehrerer Wege gleichzeitig erhöht. Allerdings bleibt man dabei, dass man sich nur für kürzeste q-s-Wege im (aktuellen) RNW interessiert. Dazu lässt man aus dem RNW alle Kanten weg, die nicht zu solchen kürzesten Wegen gehören.

1.4.1 Niveaunetzwerke und Sperrflüsse

Definition 1.4.1. *Seien f ein Fluss in einem FNW G und G_f das zugehörige RNW. Sei L der Abstand von q und s in G_f. Das* ***Niveaunetzwerk (NNW)*** *$G'_f = (V'_f, E'_f)$ besteht aus den Knoten und Kanten, die auf Wegen von q nach s der Länge L liegen. Jede Kante in E'_f wird mit ihrer Restkapazität* $\mathrm{rest}_f(u, v)$ *versehen.* Niveau V_ℓ *besteht aus den Knoten in V'_f, die Abstand ℓ von q haben. (Jede Kante (u, v) läuft von einem Niveau $V_{\ell-1}$ zum nächsten, V_ℓ.)*

Wir beschreiben, wie man G'_f aus G_f berechnet. Zunächst führt man in G_f eine Breitensuche von q aus durch. Diese BFS ordnet jedem Knoten v sein Level (sein Niveau) $d(v)$ zu. Sie kann abgebrochen werden, wenn das Level L von s festgestellt wurde und die Nachfolgerlisten aller Knoten auf Level $L - 1 = d(s) - 1$ bearbeitet worden sind. Schon in der Breitensuche kann man auch die Kanten in G_f, die nicht von einem Level $\ell - 1$ zum nächsten Level ℓ gehen, weglassen. Dies liefert ein Zwischenergebnis G''_f. Nun bildet man den Umkehrgraphen von G''_f (alle Kanten herumdrehen) und führt darin Breitensuche von s aus durch. Knoten und Kanten, die dabei nicht erreicht werden, werden ebenfalls weggelassen.

Was übrigbleibt, ist das Niveaunetzwerk G'_f. Zudem kann man aus der Berechnung für jeden Knoten v eine Liste $\mathrm{Out}(v)$ seiner Ausgangskanten (v, u) (zu Knoten u, die um ein Niveau höher als v liegen) und eine Liste $\mathrm{In}(v)$ seiner Eingangskanten (u, v) (von Knoten u, die um ein Niveau niedriger als v liegen) erhalten.

Beispiel: In Abbildung 10 ist dargestellt, wie aus einem Flussnetzwerk G mit Fluss f ein Niveaunetzwerk entsteht.

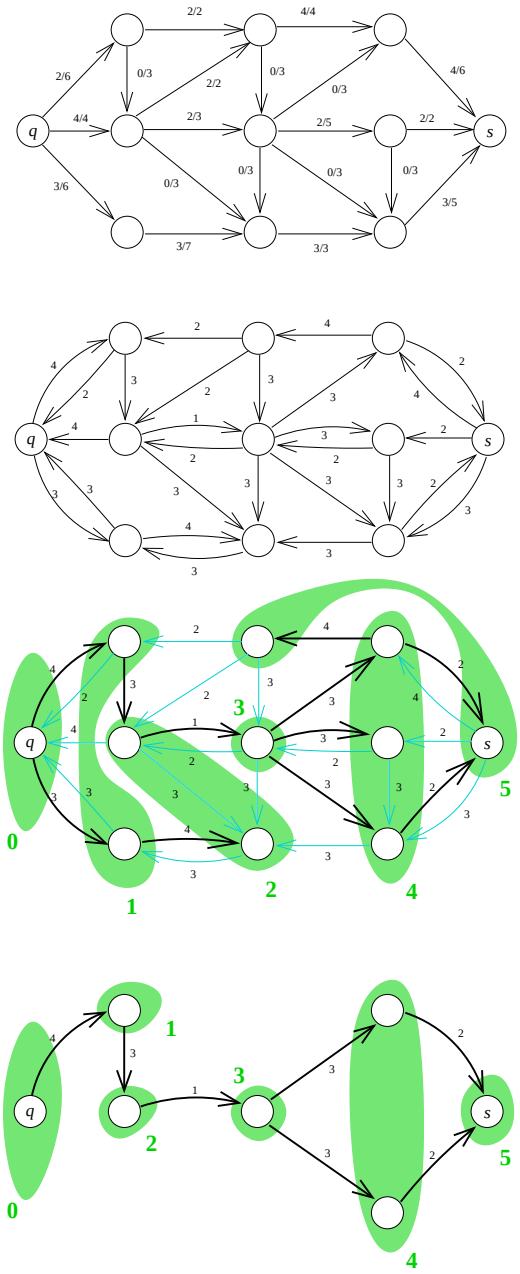


Abbildung 10: Zeile 1: Ein FNW mit Fluss. Zeile 2: Das zugehörige RNW. Zeile 3: das Ergebnis der Breitensuche von der Quelle aus im RNW, mit Levels (fett: Kanten von einem Level zum nächsten); Zeile 4: das resultierende Niveaunetzwerk.

Die Idee ist nun, einen Fluss in G'_f zu berechnen, der entlang von flussvergrößernden Wegen *der Länge L* nicht mehr vergrößert werden kann. Das bedeutet intuitiv, dass alle Wege dieser Länge „verstopft“ sind.

Definition 1.4.2. *Sei f ein Fluss in einem FNW G, und sei $G'_f = (V'_f, E'_f)$ das zugehörige Niveaunetzwerk. Ein* ***Sperrfluss*** *(*engl.: *blocking flow) für G'_f ist ein Fluss $\varphi\colon E'_f \to \mathbb{R}_0^+$ mit folgender Eigenschaft: Jeder q-s-Weg in G'_f hat mindestens eine* ***gesättigte*** *Kante, das ist eine Kante (u, v), die $\varphi(u, v) = \mathrm{rest}_f(u, v)$ erfüllt.*

Beispiel: Die Bilder in Abb. 11 zeigen ein Niveaunetzwerk mit Restkapazitäten und einen Sperrfluss in diesem Niveaunetzwerk.

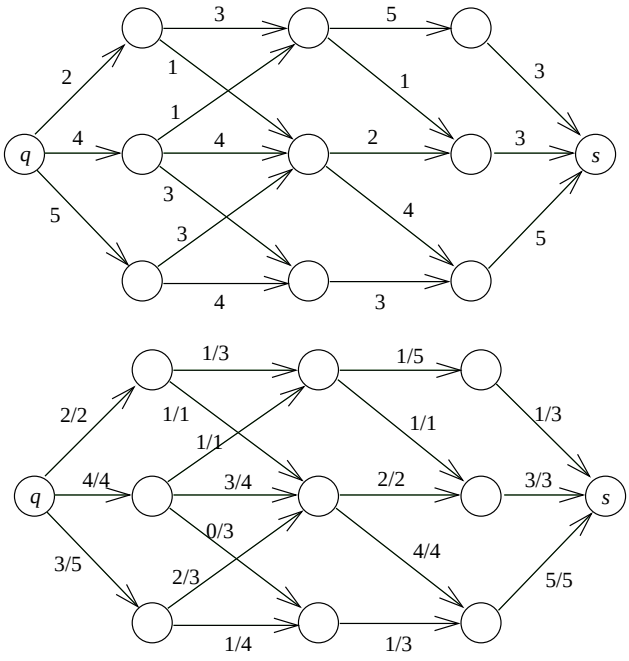


Abbildung 11: Oben: Ein Niveaunetzwerk. Unten: Ein Sperrfluss.

Bemerkung: Die Definition verlangt im Wesentlichen, dass man φ mit der FF-Methode nicht mehr vergrößern kann, wenn man ausschließlich Vorwärtskanten betrachtet. Diese Einschränkung führt zu der Frage, ob es Sperrflüsse gibt, die keine maximalen Flüsse für ihr NNW G'_f sind. Dies ist tatsächlich der Fall (Übung).

1.4.2 Die Sperrflussmethode mit Analyse

Algorithmus 1.4.3 (**Sperrflussmethode**).
INPUT: Flussnetzwerk $G = (V, E, q, s, c)$

METHODE:
1 $f :=$ der Nullfluss;
2 **repeat**
3 Berechne das Niveaunetzwerk G'_f;
4 Falls dabei in der ersten BFS s nicht erreicht: **return** f;
5 Konstruiere einen Sperrfluss φ in G'_f;
6 $f := f + \varphi$; // siehe Gleichung (1)

Algorithmus 1.4.3 gibt die Sperrflussmethode wieder. (Wir sprechen hier von einer „Methode“, weil eine wesentliche Komponente, nämlich die Bestimmung der Sperrflüsse, noch offen bleibt.) Wie vorher sehen wir folgendes: Wenn die Schleife terminiert, dann kann das nur daran liegen, dass im nun vorliegenden RNW s von q aus nicht mehr erreichbar ist, also nach dem MaxFlow-MinCut-Theorem, dass der nun vorliegende und ausgegebene Fluss maximal ist. Es bleibt zu zeigen, dass die Schleife (schnell) terminiert. Schließlich muss man überlegen, wie schnell man Sperrflüsse berechnen kann.

Lemma 1.4.4. *Die Sperrflussmethode terminiert nach maximal* $n-1$ *Sperrflussberechnungen.*

Beweis: Wir werden zeigen, dass die Anzahl der Levels im NNW in jedem Schleifendurchlauf strikt zunimmt. Da dieser Abstand anfangs mindestens 1 ist und nicht größer als $n-1$ sein kann, wenn s von q aus erreichbar ist, kann es nicht mehr als $n-1$ Sperrflussberechnungen und anschließende Flussvergrößerungen geben.

Sei dazu $L := L_{t-1} := d_{t-1}(s)$ der Abstand von q nach s im alten RNW $G_f^{(t-1)}$, für $t \geq 1$. Dann ist L auch die Anzahl der Levels im NNW G'_f in Iteration t. Zu G'_f wird der Sperrfluss φ berechnet, und der Fluss wird mittels $f := f + \varphi$ vergrößert. Wir betrachten nun einen beliebigen q-s-Weg $p = (v_0, v_1, v_2, \ldots, v_r)$ mit $v_0 = q$, $v_r = s$ im RNW $G_f^{(t)}$ *nach* Ausführung dieser Flussvergrößerung. (Falls es keinen solchen Weg gibt, stoppt der Algorithmus in der nächsten Runde ohne Sperrflussberechnung, es ist also nichts zu zeigen.) Wir behaupten, dass r größer als L sein muss.

Es gibt zwei Fälle:

1. Fall: Sämtliche Kanten von p sind im alten RNW $G_f^{(t-1)}$ enthalten. Weil in $G_f^{(t-1)}$ Knoten q und s Abstand L haben, muss $r \geq L$ gelten. *Annahme*: $r = L$. Dann ist p ein kürzester q-s-Weg in $G_f^{(t-1)}$, liegt also komplett in G'_f. Nach der Definition eines Sperrflusses ist in φ mindestens eine Kante e von p gesättigt, d. h. es gilt $\varphi(e) = \text{rest}_f^{(t-1)}(e)$. Nach der Definition von $f + \varphi$ ist dann e nicht im neuen Restnetzwerk $G_f^{(t)}$, ein Widerspruch. Daher gilt in diesem Fall $r > L$.

2. Fall: p enthält mindestens eine Kante (v_{i-1}, v_i), die *nicht* im alten RNW $G_f^{(t-1)}$ enthalten ist. Jede solche Kante kommt durch die Flussvergrößerung ins Restnetzwerk, und das heißt, dass v_i in Niveau $V_{\ell-1}$ und v_{i-1} in Niveau V_ℓ von G'_f liegt, für ein $\ell \in \{1, \ldots, L\}$, in anderen Worten $d_{t-1}(v_i) = d_{t-1}(v_{i-1}) - 1$. Solche Kanten gehen also aus Sicht des alten Niveaunetzwerks zurück in Richtung Quelle! Für Kanten (v_{i-1}, v_i) auf p, die im alten RNW $G_f^{(t-1)}$ enthalten sind, gilt $d_{t-1}(v_i) \leq d_{t-1}(v_{i-1}) + 1$. Weg p mit mindestens einer neuen Kante kann daher mit seinen r Schritten maximal den Abstand $d_{t-1}(v_r) \leq r - 2$ erreichen. Es ist aber $v_r = s$, und $d_{t-1}(s) = L$, also gilt $r \geq L + 2$. □

Der *Algorithmus von Dinitz* ist eine spezielle Version der Sperrflussmethode, in der Sperrflüsse iterativ gefunden werden, wie folgt: Man gibt den Kanten im NNW G'_f ihre Restkapazitäten $\text{rest}_f(u, v)$ und startet mit $\varphi = \varphi_0 \equiv 0$, dem Nullfluss. Nun verfährt man im Wesentlichen wie die Ford-Fulkerson-Methode auf G'_f, allerdings werden nie Rückwärtskanten betrachtet. Dies führt nach einer Reihe von Runden zu einem Sperrfluss. Es ist nicht besonders schwierig, dies so zu organisieren, dass eine Sperrflussberechnung Zeit $O(nm)$ benötigt (Übung). Da es maximal $n-1$ Sperrflussberechnungen gibt, ist die Gesamtlaufzeit des Algorithmus von Dinitz $O(n^2m) = O(n^4)$.

Es gibt noch weitere raffinierte Methoden für die Berechnung von Sperrflüssen. Eine heißt „Backward-Forward-Propagation“. Diese berechnet einen Sperrfluss in Zeit $O(n^2)$, und damit benötigt die gesamte Flussberechnung Zeit $O(n^3)$. Sleator und Tarjan haben 1980/83 eine Datenstruktur („dynamic trees“) angegeben, mit deren Hilfe sich die Konstruktion eines Sperrflusses sogar in Zeit $O(m \log n)$ bewerkstelligen lässt. Dies führt dann zu einem Flussalgorithmus mit Laufzeit $O(nm \log n)$.

Die Methode „Backward-Forward-Propagation“ wird im WS 2021/22 nicht behandelt, ist also nicht prüfungsrelevant.

1.5 Der Preflow-Push-Ansatz

Wir hatten folgende Veranschaulichung für ein Flussproblem betrachtet:

Situation: („Fanströme“) In Dortkirchen gibt es ein Straßensystem und 60 000 Fußballfans, die am Bahnhof ankommen und zu Fuß zum Stadion wollen. Für jede Straße ist eine maximale Gesamtanzahl an Personen festgelegt, die sie (in eine festgelegte Richtung) passieren können. Wie soll die Polizei die Menschenmassen aufteilen und lenken, so dass alle ankommen? Für die Abbildung siehe Abschnitt 1.1.

Wenn wir fragen, wie viele Fans maximal zum Ziel gelangen können, landen wir beim MaxFlow-Problem. Wir betrachten in diesem Abschnitt eine Strategie, die sich vom Ford-Fulkerson-Ansatz („flussvergrößernde Wege“) unterscheidet.

Im Fanströme-Beispiel sieht die Grundidee so aus. Es wird eine Simulation auf dem Papier durchgeführt, bei der man zunächst einmal so viele Fans in das Straßennetz hineinschickt wie auf die Straßen passen, die direkt vom Bahnhof wegführen. Man erlaubt vorläufig, dass Fans auf Plätzen und Straßenkreuzungen „herumstehen“, dass also mehr dort angekommen sind als wieder weggegangen sind („Überschuss“).

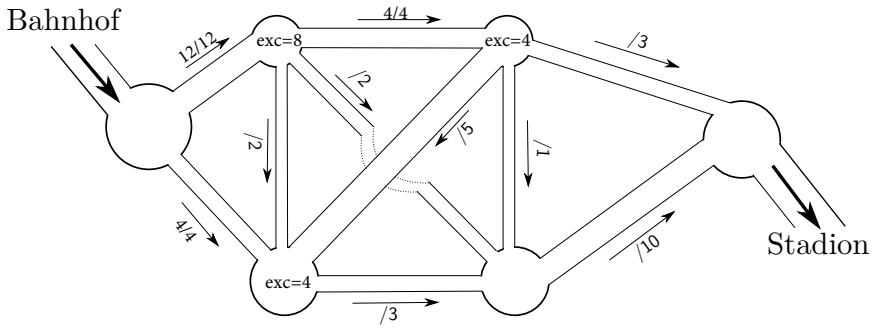


Abbildung 12: Beispiel Fanströme. Straßenkapazitäten und testweise Einleitung von Fanströmen. (Zahlen in Tausend Fans.) Erzeugt Überschuss („excess“) in einigen Knoten.

Nun verschiebt man geschickt Gruppen zwischen Plätzen hin und her und schreibt mit, wie viele Fans „netto“ durch eine Straße gegangen sind (wenn welche zurückgehen, subtrahiert man wieder). Dieser Nettowert muss immer innerhalb der Kapazität der jeweiligen Straße liegen, und man kann von einem Platz nicht mehr Fans abziehen als gerade da sind. Es kann auch passieren, dass Fans zum Bahnhof zurückgeschickt werden, also schließlich gar nicht ins Netzwerk eintreten dürfen. Wenn man es schließlich geschafft hat, alle Fans beim Stadion abzuliefern oder zum Bahnhof zurückzuschicken, ergeben die (Fluss-)Werte an den Straßen einen legalen Plan. Wenn man die Verschiebungen geschickt organisiert, ist das Ergebnis sogar ein maximaler

Fluss.

Wir modellieren das Vorgehen mathematisch. Dazu lockern wir die Kirchhoff-Bedingung und erhalten so den Begriff eines Präflusses („*preflow*“).

Definition 1.5.1. *Ein* ***Präfluss*** *in einem Flussnetzwerk $G = (V, E, q, s, c)$ ist eine Funktion $f\colon E \to \mathbb{R}_0^+$, die jeder Kante e einen Wert $f(e)$ zuordnet, wobei gilt:*

(i) *Mit*

$$f_{\mathrm{In}}(v) := \sum_{(u,v)\in E} f(u,v), \qquad f_{\mathrm{Out}}(v) := \sum_{(v,u)\in E} f(v,u).$$

gilt $f_{\mathrm{In}}(v) \geq f_{\mathrm{Out}}(v)$ für alle $v \in V - \{q\}$.

(ii) *$0 \leq f(e) \leq c(e)$, für alle $e \in E$.*

(Der einzige Unterschied zum Begriff des Flusses ist das „$\geq$“ in Bedingung (i).)

Es ergeben sich „Überschüsse“ (*excess*)

$$\mathrm{ex}(v) := f_{\mathrm{In}}(v) - f_{\mathrm{Out}}(v)$$

in den Knoten $v \in V$. Da q keine Eingangskanten hat, gilt stets $f_{\mathrm{In}}(q) = 0$, also $\mathrm{ex}(q) \leq 0$. Die Quelle ist nach Bedingung (i) der einzige Knoten, der negativen Überschuss haben kann. Auf der anderen Seite gilt stets $\mathrm{ex}(s) = f_{\mathrm{In}}(s) \geq 0$, weil s keine ausgehenden Kanten hat.

Ein Knoten $v \in V - \{s\}$ heißt *aktiv*, wenn $\mathrm{ex}(v) > 0$ ist, sonst *inaktiv*. Die Quelle ist nie aktiv, da stets $\mathrm{ex}(q) \leq 0$ gilt; die Senke ist nach Definition nie aktiv.

Die Grundidee von Preflow-Push-Algorithmen ist, einen Präfluss zu definieren und durch geeignete Operationen schrittweise so zu verändern, dass schließlich alle Überschüsse in Knoten $v \in V - \{s\}$ verschwunden sind. Dann ist ein Fluss entstanden, der sich als maximal erweist, wenn man richtig vorgeht.

Die *Restkapazitäten* $\mathrm{rest}_f(u, v)$ für $(u, v) \in E$ und $(v, u) \in E$ werden für einen Präfluss genauso definiert wie für einen Fluss, ebenso das Restnetzwerk (RNW) G_f, das alle Kanten mit positiver Restkapazität enthält.

Um die Operationen zu steuern, benötigt man ein neues zentrales Konzept, so genannte *Höhenfunktionen*. Dabei stellt man sich vor, dass jeder Knoten v auf einem bestimmten Level $h(v) \in \mathbb{N}$ platziert ist. Diese Niveaus (Höhen) ändern sich im Verlauf des Algorithmus.

Definition 1.5.2. ***(Korrigiert 5.11.2021.)*** *Gegeben sei ein* ***Präfluss*** *f in einem Flussnetzwerk $G = (V, E, q, s, c)$. Eine* Höhenfunktion für *f ist eine Funktion $h\colon V \to \mathbb{N}$ mit:*

(i) $h(q) = n = |V|$ *und* $h(s) = 0$, *und*

(ii) $h(u) \leq h(v) + 1$ *für alle* $(u, v) \in E_f$.

Wenn man sich die Knoten entsprechend ihrer Höhenwerte auf Niveaus angeordnet denkt, bedeutet dies, dass Restnetzwerkkanten (mit positiver Restkapazität) nicht mehr als ein Niveau nach unten gehen dürfen. Das heißt:

- Wenn $(u, v) \in E$ und $h(u) \geq h(v) + 2$, dann muss (u, v) *gesättigt* sein, d. h. es muss $f(u, v) = c(u, v)$ gelten.
- Wenn $(v, u) \in E$ und $h(u) \geq h(v) + 2$, dann muss (v, u) *leer* sein, d. h. es muss $f(v, u) = 0$ gelten.

Preflow-Push-Algorithmen manipulieren Paare (f, h), wobei f Präfluss und h eine Höhenfunktion für f ist. Das Ziel ist erreicht, wenn f ein Fluss geworden ist.

Die einfachste Weise, ein Paar (f, h) zu finden, mit dem man beginnen kann, ist folgende: Setze die Quelle auf Höhe n, alle anderen Knoten auf Höhe 0, und schicke über die Ausgangskanten von q den maximal erlaubten Fluss.

Algorithmus 1.5.3 (**Initialisierung**)**.**

Data: *Flussnetzwerk* $G = (V, E, q, s, c)$ *mit* $n = |V|$
Result: *Initialisierung für Preflow-Push*
1 $f(q, v) \leftarrow c(q, v)$ *für Kanten* (q, v) *aus der Quelle;*
2 $f(u, v) \leftarrow 0$ *für* $(u, v) \in E$ *mit* $u \neq q$*;*
3 $h(q) \leftarrow n$*;*
4 $h(v) \leftarrow 0$ *für* $v \in V - \{q\}$*;*

Wir beobachten: f ist ein Präfluss und h ist eine Höhenfunktion für f. (Die Höhen von Quelle und Senke sind n und 0 wie gefordert. Alle Flussnetzwerkkanten, die steil nach unten gehen, sind gesättigt.) Die Überschüsse in den Knoten berechnen sich wie folgt:

- $\mathrm{ex}(v) = c(q, v)$ für v mit $(q, v) \in E$.
- $\mathrm{ex}(v) = 0$ für $v \in V - \{q\}$ mit $(q, v) \notin E$.
- $\mathrm{ex}(q) = -C_G = -\sum_{(q,v)\in E} c(q, v)$.

Aktiv sind also die unmittelbaren Nachfolger von q, außer eventuell s. Die Summe aller Überschüsse (inklusive dem von q) ist 0. (Es wird sich herausstellen, dass dies im Verlauf des Algorithmus immer so bleibt.)

Bemerkung: Bei der Initialisierung verwendet man auch oft eine Variante, die zu einer leichten Beschleunigung führt. Anstatt Knoten $v \in V - \{q\}$ pauschal auf Höhe 0 zu setzen, kann man ihn auch auf Level $d_G(v, s)$ setzen (den Abstand von v zur Senke im Flussnetzwerk G, was nach der Initialisierung im Teil $V - \{q\}$ zum Restnetzwerk G_f identisch ist). Die Zahlen $d_G(v, s)$ können leicht durch Breitensuche im Umkehrgraphen (ohne q) von s aus ermittelt werden. Man sieht sofort, dass die Bedingung an die Höhenfunktion erfüllt ist: Wenn $(u, v) \in E = E_f$ gilt, dann ist $d_G(u, s) \leq d_G(v, s)+1$.

Es gibt zwei zentrale Operationen zum Umbau des Paares (f, h). Eine („push(u, v)“) ändert den Fluss über eine Kante (u, v) oder ihre Umkehrkante (v, u), die andere („relabel(u)“) ändert den Höhenwert eines Knotens u. (Im Gegensatz zur FF-Methode und ihren Abwandlungen wie die Sperrflussmethode sind diese Manipulationen sehr lokal.) Wegen der Namen der Operationen heißen Preflow-Push-Verfahren auch *Push-Relabel*-Verfahren.

Die erste Operation verändert den Flusswert auf einer Kante. Zweck ist, Überschuss von einem aktiven Knoten u zu einem Knoten v zu verschieben, entlang einer Kante $(u, v) \in E_f$, also einer Kante mit positiver Restkapazität. (Dann kann $(u, v) \in E$ oder $(v, u) \in E$ gelten, für das Flussnetzwerk $G = (V, E, q, s, c)$.) Diese Operation darf nur ausgeführt werden, wenn $h(u) = h(v) + 1$ gilt.

Intuition: Überschuss von u nach v verschieben kann man nur, wenn

- u höher sitzt als v,
- u Überschuss *hat*, d. h. $\mathrm{ex}(u) > 0$ gilt, und
- auf der Restnetzwerkkante (u, v) Platz ist, also $\mathrm{rest}_f(u, v) > 0$ gilt.

Die letzte Bedingung bedeutet für $(u, v) \in E$, dass (u, v) nicht gesättigt ist (*Vorwärtskante*, man kann noch mehr Fluss über die Kante schicken) und für $(v, u) \in E$, dass $f(v, u) > 0$ ist (*Rückwärtskante*, man kann den Fluss über (v, u) verringern). Aus der Definition des Konzepts „Höhenfunktion für f“ und der ersten und der letzten Bedingung ergibt sich, dass $h(u) = h(v) + 1$ gelten muss, damit man Überschuss von u nach v verschieben kann.

Der Fluss kann sich nicht um mehr als $\mathrm{ex}(u)$ und auch nicht um mehr als $\mathrm{rest}_f(u, v) > 0$ ändern.

Algorithmus 1.5.4 (push).

Data: f, h, Kante $(u,v) \in E_f$ mit $\mathrm{ex}(u) > 0$, $u \neq q$ und $h(u) = h(v) + 1$

1 $\delta \leftarrow \min\{\mathrm{ex}(u), \mathrm{rest}_f(u,v)\}$;

2 **if** $(u,v) \in E$ **then** $f(u,v) \leftarrow f(u,v) + \delta$;

3 **if** $(u,v) \notin E$ **then** $f(v,u) \leftarrow f(v,u) - \delta$;

// (implizit:) $\mathrm{ex}(u) \leftarrow \mathrm{ex}(u) - \delta$;

// (implizit:) $\mathrm{ex}(v) \leftarrow \mathrm{ex}(v) + \delta$;

// (implizit:) $\mathrm{rest}_f(u,v) \leftarrow \mathrm{rest}_f(u,v) - \delta$;

// (implizit:) $\mathrm{rest}_f(v,u) \leftarrow \mathrm{rest}_f(v,u) + \delta$;

Es gibt zwei Möglichkeiten:

- Wenn $\mathrm{ex}(u) \geq \mathrm{rest}_f(u,v)$ ist, verschieben wir $\mathrm{rest}_f(u,v)$ Überschuss. Dadurch wird $\mathrm{rest}_f(u,v) = 0$: die Kante (u,v) verschwindet aus dem Restnetzwerk. Wir nennen dies eine *saturierende* push-Operation.
  (Achtung: Die Kante (u,v) kann später auch wieder im RNW auftauchen, nämlich wenn push(v,u) ausgeführt wird.)

- Wenn $\mathrm{ex}(u) < \mathrm{rest}_f(u,v)$ ist, verschieben wir $\mathrm{ex}(u)$ viel Überschuss. Dadurch wird $\mathrm{ex}(u) = 0$, und u wird inaktiv. Wir nennen dies eine *nichtsaturierende* push-Operation. (Achtung: Später kann u auch wieder aktiv werden, nämlich wenn Überschuss von einem anderen Knoten nach u verschoben wird.)

Die Operation push(u,v) senkt $\mathrm{rest}_f(u,v)$ um δ und erhöht $\mathrm{rest}_f(v,u)$ um δ. Dies geschieht quasi automatisch, nicht algorithmisch, da diese Werte aus Flusswerten und Kapazitäten berechnet werden.

Lemma 1.5.5. (a) *Nach der Ausführung von push(u,v) ist das neue f immer noch ein Präfluss.*

(b) *Nach der Ausführung von push(u,v) ist h immer noch eine Höhenfunktion für das neue f.*

Beweis: (a) Dass Flusswerte auf (u,v) bzw. (v,u) nichtnegativ bleiben und die Kapazitätsbedingung eingehalten wird, liegt daran, dass wir $f(u,v)$ oder $f(v,u)$ nur um $\delta \leq \mathrm{rest}_f(u,v)$ ändern. Dass $f_{\mathrm{In}}(u) \geq f_{\mathrm{Out}}(u)$ bleibt, liegt daran, dass wir $f(u,v)$ nur um $\delta \leq \mathrm{ex}(u)$ erhöhen bzw. $f(v,u)$ nur um $\delta \leq \mathrm{ex}(u)$ vermindern. Es ist auch klar, dass $\mathrm{ex}(v)$ durch die push-Operation nur wachsen kann, also auf jeden Fall nachher positiv ist. (b) Bedingung (i) in Definition 1.5.2 bleibt erfüllt. ir Für (ii) müssen nur Kanten (u,v) und (v,u) betrachten, da sich auf den anderen Kanten nichts ändert. Nach wie vor gilt $h(u) \leq h(v) + 1$. Der rest_f-Wert auf der Gegenkante (v,u) steigt;

dadurch könnte (v, u) neu ins RNW gelangen. Dies ist aber für h auch kein Problem, da $h(v) = h(u) - 1 \leq h(u) + 1$ gilt. □

Es kann nun natürlich passieren, dass keine push-Operation ausführbar ist, obwohl es Knoten mit Überschuss gibt. Woran kann das liegen? Knoten sitzen auf bestimmten Höhen. Wenn u Überschuss hat (also $\text{ex}(u) > 0$ gilt), aber im Restnetzwerk bezüglich der Höhenfunktion in einem lokalen Minimum sitzt (also $h(u) \leq h(v)$ für alle $(u, v) \in E_f$ gilt), kann man diesen Überschuss nicht bewegen. In diesem Fall werden aktive Knoten u vorsichtig angehoben, so dass h Höhenfunktion bleibt, aber schließlich doch erreicht wird, dass ein aktiver Knoten u eine Ausgangskante (u, v) hat, auf der eine push-Operation zulässig ist. Dafür gibt es die Operation *relabel*(u), die nur die Höhenfunktion verändert: Wenn für einen aktiven Knoten u gilt, dass $h(u) \leq h(v)$ für alle v mit $(u, v) \in E_f$, dann können wir die Höhe von u um 1 vergrößern.

Im Bild des Röhrensystems, durch das Wasser geleitet werden soll, lässt sich der Zweck des Anhebens ebenfalls recht anschaulich beschreiben. Wir stellen uns vor, die Knoten hätten Reservoirs, in denen Wasser temporär aufbewahrt werden kann. Knoten können auf einem Niveau zwischen 0 und $2n - 1$ liegen. Anfangs wird die Quelle auf Niveau n gesetzt, alle anderen Knoten auf Niveau 0, und die maximale von den Kantenkapazitäten erlaubte Menge an Wasser wird von q zu seinen Nachfolgern geschickt. In diesen steht nun Überschuss-Wasser. Man kann Wasser von u nach v schicken, aber nur wenn auf der Restnetzwerkkante (u, v) Platz ist *und* wenn u auf einem höheren Niveau sitzt als v (dann fließt nämlich das Wasser von selber nach unten). Der Zweck der nun folgenden relabel-Operation ist es, Knoten vorsichtig anzuheben, so dass dieser Fluss nach unten stattfinden kann, sei es nun in Richtung Senke oder auch zurück zur Quelle. Der Algorithmus besteht dann darin, push- und relabel-Operationen anzuwenden, bis nur noch Knoten s Überschuss hat, also alle anderen temporären Reservoirs leer sind.

Algorithmus 1.5.6 (relabel).

Data: f, h, aktiver Knoten u mit $\forall v \in V : \text{rest}_f(u, v) > 0 \Rightarrow h(u) \leq h(v)$
Result: Hebe aktiven Knoten u ein Niveau höher
1 $h(u) \leftarrow h(u) + 1$;

Durch den Aufruf **relabel**(u) ändert sich nichts daran, dass h Höhenfunktion für f ist: Die Höhen von Knoten q und s werden nie verändert, da diese nie aktiv sind. Für Kanten $(u, v) \in E_f$ gilt nachher $h(u) \leq h(v) + 1$; für Kanten $(w, u) \in E_f$ galt vorher $h(w) \leq h(u) + 1$, also nachher $h(w) \leq h(u)$.

Wir können nun die Klasse der Preflow-Push-Algorithmen beschreiben.

Algorithmus 1.5.7 (**Preflow-Push-Methode**).

```
   Data: Flussnetzwerk G = (V, E, q, s, c)
   Result: Maximaler Fluss f für G
 1 Initialisierung(G);
 2 while es gibt aktiven Knoten u do
 3 |   wähle einen aktiven Knoten u;
 4 |   if h(u) ≤ h(v) für alle Nachfolger v von u im RNW G_f then
 5 |   |   relabel(u)
 6 |   else
 7 |   |   wähle v mit (u, v) ∈ E_f und h(u) = h(v) + 1; push(u, v);
 8 |   end
 9 end
10 return f;
```

In der Preflow-Push-Methode sind die Wahl des aktiven Knotens u und gegebenenfalls des Knotens v offengelassen. Es wird sich zeigen, dass diese Entscheidungen für die Terminierung und die Korrektheit des Ergebnisses irrelevant sind. Strategien für diese Entscheidungen bestimmen aber, wie schnell der Algorithmus abläuft.

Beispiel: Tafel, Internet.

`http://www.adrian-haarbach.de/idp-graph-algorithms/implementation/maxflow-push-relabel/index_en.html`

(Vorsicht: Quelle heißt s [„*source*“], Senke heißt t [„*target*“]. Etwas länger, aber instruktiv: Ablauf auf Graph „Jungnickel“. Manche Browser zeigen die Pfeilrichtungen nicht an. Im Internet Explorer sind sie aber zu sehen.)

In Beispielen sieht man, dass der Algorithmus in der Lage ist, einerseits Überschuss nach s zu verschieben und andererseits Überschuss, der nicht durch das Netzwerk hindurchpasst, zurück zu q zu schieben. Für den zweiten Teil ist es nötig, dass die Höhenwerte auch größer als n werden, damit es zu q mit $h(q) = n$ „abwärts“ geht wie in der push-Operation gefordert.

Wir betonen nochmals, dass die durch die Initialisierung festgelegten Werte $h(q) = n$ und $h(s) = 0$ nie geändert werden. Damit erhalten wir die erste entscheidende Eigenschaft von Systemen (f, h).

Lemma 1.5.8. *Wenn h eine Höhenfunktion für Präfluss f ist, dann gibt es im RNW G_f keinen Weg von q nach s.*

Beweis: Da es n Knoten gibt, aber die Menge $\{0, 1, \ldots, n\}$ Größe $n + 1$ hat, muss es ein $k \in \{1, \ldots, n-1\}$ geben, so dass kein Knoten v mit $h(v) = k$ existiert. Ein Weg

in G_f von q nach s müsste eine Kante $(u, v) \in E_f$ mit $h(u) > k > h(v)$ enthalten. Eine solche gibt es aber wegen der Definition einer Höhenfunktion nicht. □

Dieses Lemma macht einen zentralen Unterschied zwischen der Preflow-Push-Methode und der Ford-Fulkerson-Methode explizit: FF arbeitet mit Flüssen, und stoppt, wenn es keinen q-s-Weg im RNW mehr gibt. Bei Preflow-Push gibt es nie einen solchen Weg. Man arbeitet mit Präflüssen f und stoppt, wenn man einen Fluss hat.

Lemma 1.5.9. *Wenn h Höhenfunktion für f ist und kein Knoten aktiv ist, dann ist f ein maximaler Fluss für G.*

Beweis: Wenn kein Knoten aktiv ist, hat jeder Knoten $v \in V - \{q, s\}$ Überschuss 0, und das heißt, dass überall die Kirchhoff-Bedingung erfüllt ist. Daher ist f ein Fluss. Nach Lemma 1.5.8 gibt es im RNW G_f keinen Weg von q nach s. Das Max-Flow-Min-Cut-Theorem liefert dann, dass f maximal ist. □

Was wir noch zeigen müssen, ist, dass die Anwendung der Preflow-Push-Methode nach endlich vielen Schritten dazu führt, dass kein Knoten mehr aktiv ist. Die Strategie hierfür ist folgende: Wir zeigen, dass die Höhen von Knoten durch $2n - 1$ nach oben beschränkt sind; Höhen können also nicht „in den Himmel wachsen“ und die Anzahl der relabel-Operationen ist endlich. Weiter stellen wir fest, dass zwei saturierende push-Operationen auf ein und derselben Kante (u, v) nur vorkommen können, wenn zwischendurch die Höhe $h(u)$ strikt angewachsen ist. Dies führt dazu, dass es nur endlich viele saturierende push-Operationen geben kann. Schließlich zeigen wir (mit einem weiteren Trick), dass auch die Anzahl der nichtsaturierenden push-Operationen endlich ist.

Lemma 1.5.10. *Wenn f Präfluss ist und $\mathrm{ex}(v) > 0$ gilt, dann gibt es im RNW G_f einen Weg von v nach q.*

Beweis: Sei X die Menge aller Knoten v, von denen aus es einen G_f-Weg nach q gibt, und sei $Y = V - X$. Es ist klar, dass $q \in X$ gilt. Es gibt keine Kante $(u, v) \in E_f$ mit $u \in Y$ und $v \in X$. Daraus folgt:

$$f(v, u) = 0 \text{ für } (v, u) \in E, v \in X, u \in Y. \tag{2}$$

Wir zeigen nun, dass die Summe der Überschüsse der Knoten in Y nicht positiv sein kann. Da Überschüsse (außer in $q \in X$) immer nichtnegativ sind, müssen dann alle Knoten in Y Überschuss 0 haben. Daraus folgt die Behauptung: Jeder Knoten v mit positivem Überschuss muss in der Menge X liegen.

$$\begin{aligned}\sum_{u\in Y}\mathrm{ex}(u) &= \sum_{u\in Y}(f_{\mathrm{In}}(u)-f_{\mathrm{Out}}(u))\\ &= \sum_{u\in Y}f_{\mathrm{In}}(u)-\sum_{u\in Y}f_{\mathrm{Out}}(u)\\ &= \sum_{\substack{(v,u)\in E\\ v\in V,u\in Y}}f(v,u) \;-\; \sum_{\substack{(u,v)\in E\\ u\in Y,v\in V}}f(u,v)\\ &\overset{(*)}{=} \sum_{\substack{(v,u)\in E\\ v\in X,u\in Y}}f(v,u) \;-\; \sum_{\substack{(u,v)\in E\\ u\in Y,v\in X}}f(u,v).\end{aligned}$$

Die Gleichheit $(*)$ in der letzten Zeile ergibt sich dabei daraus, dass $f(u,v)$ für die Kanten $(u,v)\in E$ mit $u,v\in Y$ in beiden Summen vorkommt, mit entgegengesetzten Vorzeichen. Die erste Summe ist 0 wegen (2), die zweite Summe enthält nur nichtnegative Summanden. Zusammen ergibt sich ein nicht-positiver Wert. □

Intuitiv kann man aus Lemma 1.5.10 auch entnehmen, dass es immer im Prinzip möglich ist, einen Überschuss entlang von RNW-Kanten zurück zur Quelle zu schieben.

Lemma 1.5.11. (i) *Für alle $v\in V$ gilt stets $h(v)\le 2n-1$.*

(ii) *Die Anzahl der relabel-Operationen ist nicht größer als $2n(n-2)$.*

Beweis: (i) Wir haben stets $h(q)=n$ und $h(s)=0$. Betrachte nun $v\in V-\{q,s\}$ nach der Ausführung einer relabel(v)-Operation. Dann ist v aktiv. Aus Lemma 1.5.10 in Kombination mit Eigenschaft (ii) in der Definition von Höhenfunktionen folgt, dass $h(v)\le 2n-1$ gilt. Denn: Sei $v=v_0,v_1,\ldots,v_t=q$ ein einfacher Weg von v nach q im RNW G_f. Dann gilt $t\le n-1$, und wir haben $h(v_{i-1})-h(v_i)\le 1$, für $1\le i\le t$. Summation liefert $h(v)-h(q)=h(v_0)-h(v_t)\le t\le n-1$, also $h(v)\le h(q)+n-1=2n-1$.
(ii) Jeder Knoten $v\neq q,s$ startet mit $h(v)=0$. Mit jeder Operation relabel(v) steigt die Höhe um 1 an. Daher kann nach (i) auf v höchstens $(2n-1)$-mal eine relabel-Operation angewendet werden. Summiert über alle $v\neq q,s$ ergibt sich die Schranke $(2n-1)(n-2)<2n(n-2)$ für die Anzahl der relabel-Operationen. □

Lemma 1.5.12. *Die Anzahl der saturierenden push-Operationen ist nicht größer als $2nm$.*

Beweis: **(Ergänzt 6.11.2021.)** Wir betrachten zunächst eine beliebige Kante (u,v) mit $(u,v)\in E$ oder $(v,u)\in E$, das heißt, (u,v) ist potenzielle RNW-Kante, und zeigen,

dass über die Kante (u, v) nicht öfter als n-mal eine saturierende push-Operation läuft. Der Beweis ist sehr ähnlich zu dem in der Laufzeitanalyse für den Edmonds-Karp-Algorithmus. Betrachte einen Zeitpunkt t, an dem push(u, v) saturierend ausgeführt wird. Dann verschwindet (u, v) zu diesem Zeitpunkt aus dem Restnetzwerk. Weiter betrachte einen Zeitpunkt $t'' > t$, zu dem erneut push(u, v) ausgeführt wird. Dann ist (u, v) zwischenzeitlich wieder im RNW erschienen, das heißt, es gibt einen Zeitpunkt t' mit $t < t' < t''$, zu dem über die Gegenkante (v, u) eine push-Operation stattfindet. Die Höhen $h_t(u)$, $h_{t'}(u)$, $h_{t'}(v)$, $h_{t''}(u)$ und $h_{t''}(v)$ zu diesen drei Zeitpunkten erfüllen Folgendes, nach der Bedingung für push-Operationen und weil Höhen niemals sinken:

$$\begin{aligned} h_{t'}(u) &\geq h_t(u), \\ h_{t'}(v) &= h_{t'}(u) + 1, \\ h_{t''}(v) &\geq h_{t'}(v), \\ h_{t''}(u) &= h_{t''}(v) + 1. \end{aligned}$$

Wenn man diese Gleichungen und Ungleichungen addiert, ergibt sich $h_{t''}(u) \geq h_t(u) + 2$. Das heißt: Zwischen zwei saturierenden push(u, v)-Operationen erhöht sich $h(u)$ um mindestens 2. Da anfangs $h(u) = 0$ gilt, für die erste push(u, v)-Operation $h(u) \geq 1$ sein muss und stets $h(u) \leq 2n - 1$ gilt (Lemma 1.5.11(i)), gibt es nicht mehr als n viele saturierende push(u, v)-Operationen.

Die Anzahl der Kanten (u, v) mit $(u, v) \in E$ oder $(v, u) \in E$ ist $2m$. Daher gibt es insgesamt nicht mehr als $2m \cdot n = 2nm$ saturierende push-Operationen. □

Nun müssen wir noch die Anzahl der nichtsaturierenden push-Operationen abschätzen. Hierfür betrachten wir die Kombination (f, h) aus Präfluss und Höhenfunktion als einen „Zustand“ des Systems, und definieren eine Funktion Φ, die jedem solchen Zustand ein „*Potenzial*“ $\Phi(f, h) \in \mathbb{N}$ zuordnet. (Man muss dabei den Namen „Potenzial“ nicht verstehen – es handelt sich hier nur um eine Sprechweise.) Der Wert $\Phi(f, h)$ ist anfangs 0. Er steigt mit relabel-Operationen (höchstens $2n(n-2)$ viele) und mit saturierenden push-Operationen (höchstens $2nm$ viele) an und sinkt mit jeder nichtsaturierenden push-Operation um mindestens 1. Daraus folgt, dass es nur endlich viele nichtsaturierende push-Operationen geben kann. – Dies führen wir nun genauer aus. Wir definieren:

$$\Phi(f, h) := \sum_{\substack{v \in V \\ v \text{ ist aktiv}}} h(v).$$

Es ist klar, dass Φ nur Werte in $\mathbb{N}$ annimmt.

Lemma 1.5.13. (a) *Eine relabel-Operation erhöht $\Phi(f, h)$ um 1.*

(b) *Eine saturierende push-Operation erhöht* $\Phi(f,h)$ *um höchstens* $2n-2$.

(c) *Eine nichtsaturierende push-Operation verringert* $\Phi(f,h)$ *um mindestens* 1.

(d) *Es gibt höchstens* $4n^2m - 2nm$ *nicht-saturierende push-Operationen.*

Beweis: (a) relabel(u) vergrößert die Höhe des aktiven Knotens u um 1.

(b) Durch die saturierende push-Operation push(u,v) kann Knoten v vom Zustand „inaktiv“ in Zustand „aktiv“ wechseln. Seine Höhe ist maximal $2n-2$ (denn $h(u) \leq 2n-1$ und $h(u) = h(v)+1$), und um diesen Wert kann $\Phi(f,h)$ ansteigen.

(c) Wenn push(u,v) nichtsaturierend ist, ist nachher u inaktiv geworden und es könnte sein, dass v vorher inaktiv war und nun aktiv ist. $\Phi(f,h)$ sinkt also um mindestens $h(u) - h(v) = 1$.

(d) Der Anfangswert von $\Phi(f,h)$ nach der Initialisierung ist 0. Nach Lemma 1.5.11 kann $\Phi(f,h)$ durch alle relabel-Operationen zusammen um höchstens $2n(n-2)$ ansteigen. Nach Lemma 1.5.12 in Kombination mit (b) kann $\Phi(f,h)$ durch alle saturierenden push-Operationen zusammen um nicht mehr als $2nm(2n-2) = 4nm(n-1)$ ansteigen. Die Summe aller Anstiege ist also kleiner als $4nm(n-1) + 2n(n-2) \leq 4nm(n-1) + 2nm \leq 4n^2m - 2nm$. (Wir können natürlich annehmen, dass in G jeder Knoten außer s eine Ausgangskante hat, also $m \geq n-1$ gilt.) □

Aus Lemma 1.5.12 und Lemma 1.5.13(d) folgt, dass es insgesamt nicht mehr als $4n^2m$ push-Operationen geben kann. Daher hält die Preflow-Push-Methode nach endlich vielen Operationen an. Was bleibt noch zu tun? Zunächst einmal muss man überlegen, wieviel Rechenzeit für die $O(n^2m)$ push- und die $O(n^2)$ relabel-Operationen benötigt wird. Man kann die Dinge recht leicht so arrangieren, dass man in konstanter Zeit eine Kante (u,v) für ein push(u,v) findet (man speichert die Kanten $(u,v) \in E_f$ mit aktivem u und $h(u) = h(v)+1$ in einer Wartemenge) bzw. einen Knoten u für ein relabel(u) findet (ebenfalls eine Wartemenge), und dass push(u,v) $O(1)$ Zeit benötigt, relabel(u) Zeit $O(n)$ (durchlaufe die Ausgangskanten von u und prüfe, ob sie für push(u,v) in Frage kommen). Damit ist die Rechenzeit $O(n^2m) + O(n^3) = O(n^2m)$.

Satz 1.5.14. *Die Preflow-Push-Methode kann (auf einfache Weise) so implementiert werden, dass die Rechenzeit durch* $O(n^2m)$ *beschränkt ist.*

Durch geschickte Wahl der Reihenfolge der push- und relabel-Operationen kann die Rechenzeit noch sehr verbessert werden. Zunächst nimmt man lokale Strukturierungen vor, die es einem ersparen, in jeder Runde den ganzen Graphen anzusehen.

Eine ganz einfache Variante ist die, dass man einen aktiven Knoten u, für den keine push-Operation möglich ist, gleich auf ein Niveau hebt, bei dem dies möglich ist. Dies kürzt das Anheben in Einzelschritten ab.

Algorithmus 1.5.15 (**relabel-1**).

Data: f, h, aktiver Knoten u mit $\forall v \in V : \text{rest}_f(u,v) > 0 \Rightarrow h(u) \leq h(v)$
Result: Hebe aktiven Knoten u ein Niveau höher als niedrigsten Nachfolger in G_f

```
1 h(u) ← min{h(v) | (u,v) ∈ E_f} + 1;
```

Eine weitere Vorgehensweise zur Strukturierung der Operationen ist die, einen einmal ausgesuchten Knoten u mit positivem Überschuss *komplett zu leeren*. Das heißt: Man schiebt Überschuss mit saturierenden push-Operationen zu Nachbarn, so lange dies möglich ist. Wenn u dann immer noch Überschuss hat, wird relabel-1(u) ausgeführt; danach kann wieder mit push-Operationen begonnen werden. Die ganze Operation heißt discharge(u), sie hinterlässt u als inaktiven Knoten. Damit angestrebte Laufzeiteffekte realisiert werden können, muss man die Untersuchung der von u ausgehenden Kanten passend organisieren. Dazu benötigt man die *Adjazenzliste* von u, in der in der Form einer linearen Liste alle Kanten aufgeführt sind, die möglicherweise in E_f aus u herausführen, also alle Kanten $(u,v) \in E$ und alle Kanten (u,v) mit $(v,u) \in E$. Sobald einmal relabel-1(u) ausgeführt wurde, fängt man in der Adjazenzliste von u wieder von vorne an. Wenn Knoten u komplett geleert werden konnte, merkt man sich die aktuelle Stelle in der Adjazenzliste und fährt später (wenn u wieder aktiv geworden ist und danach erneut discharge(u) aufgerufen wird) an derselben Stelle in der Adjazenzliste fort.

Algorithmus 1.5.16 (**Discharge**).

Data: f, h, aktiver Knoten u, Zeiger $u.current$ in die Adjazenzliste von u
Result: bewirkt push- und relabel-Operationen an u; schließlich $\text{ex}(u) = 0$

```
1  while u ist aktiv do
2      v ← u.current;
3      if v = NULL (Ende der Adjazenzliste erreicht) then
4          relabel-1(u);
5          setze u.current auf den Anfang der Adjazenzliste von u;
6      else
7          if rest_f(u,v) > 0 und h(v) + 1 = h(u) then
8              push(u,v)
9          else
10             schiebe u.current in Adjazenzliste um 1 Position weiter
11         end
12     end
13 end
```

Man muss sich vergewissern, dass die Ausführung der relabel-1-Operation in Zeile

4 erlaubt ist, das heißt, dass $h(u) \leq h(v)$ für alle v mit $(u, v) \in E_f$ gilt. Wenn die Situation u.current = NULL auftritt, heißt das, dass vorher die Adjazenzliste von u (eventuell in mehreren Abschnitten) komplett durchlaufen worden ist, dass sich $h(u)$ geändert hat, und so dass alle betrachteten ausgehenden Kanten (u, w)

- $h(w) \geq h(u)$ erfüllten (daran ändert sich nichts, da kein relabel(u) stattfand), oder
- $h(w) = h(u) - 1$ erfüllten und entweder von vorneherein nicht in E_f waren, oder
- durch die aktuelle oder eine frühere Discharge-Operation gesättigt worden sind.

Daher kann es keine Kanten (u, w) geben, an denen push ausführbar wäre, und relabel ist erlaubt.

Eine geläufige Vorgehensweise für eine Konkretisierung der Preflow-Push-Methode ist die „Highest-Label-Heuristik“, bei der immer Knoten mit möglichst großer Höhe ausgewählt werden, auf die dann Discharge angewendet wird.

Algorithmus 1.5.17 (**Highest-Label**).

Data: Flussnetzwerk $G = (V, E, q, s, c)$
Result: Maximaler Fluss f für G

```
1 Initialisierung(G);
2 foreach u ∈ V − {q, s} do
3 |   setze u.current auf den Anfang der Adjazenzliste von u
4 end
5 while es gibt aktives u do
6 |   wähle aktives u mit h(u) maximal;
7 |   Discharge(u);
8 end
```

Eine erste, nicht sehr schwierige, Analyse ergibt eine Rechenzeitschranke von $O(n^3)$.

Wir zeigen hier nur kurz, dass die Anzahl der nichtsaturierenden push-Operationen durch $O(n^3)$ beschränkt werden kann. Dazu überlegen wir Folgendes: Eine nichtsaturierende push-Operation an Knoten u macht diesen auf jeden Fall inaktiv. Wenn (irgendwelche saturierenden push-Operationen und) $n - 2$ nichtsaturierende push-Operationen in Folge ausgeführt wurden, ohne jegliche relabel-Operation, dann sind alle Knoten in $V - \{q, s\}$ inaktiv geworden, und der Algorithmus hält an. Das heißt: Damit der Algorithmus weiterläuft, muss nach spätestens $n - 3$ nichtsaturierenden push-Operationen eine relabel-Operation folgen. Da die Anzahl der relabel-Operationen nur $O(n^2)$ ist, ist die Anzahl der nichtsaturierenden push-Operationen durch $n \cdot O(n^2) = O(n^3)$ beschränkt.

Eine gewisse technische Schwierigkeit liegt noch im Nachweis, dass es in derselben Zeitschranke möglich ist, immer einen aktiven Knoten mit größtem h-Wert zu finden. Hierfür wird auf die Literatur verwiesen.

Satz 1.5.18. *Die Preflow-Push-Methode mit der Highest-Label-Regel kann so implementiert werden, dass die Rechenzeit durch $O(n^3)$ beschränkt ist.*

Eine zweite, etwas aufwendigere Analyse (siehe die Arbeit von Cheryian und Mehlhorn) führt zu einer Rechenzeitschranke von $O(n^2\sqrt{m})$ für die Highest-Label-Heuristik. Dies ist für Kantenanzahlen $m = o(n^2)$ noch besser als $O(n^3)$.

Literatur:
Jon Kleinberg, Éva Tardos, Algorithm Design, Addison-Wesley 2006, Section 7.4.
J. Cheriyan, K. Mehlhorn, An analysis of the highest-level selection rule in the preflow-push max-flow algorithm, Information Processing Letters 69 (1999): 239–242.

Die folgende Tabelle dient nur zur Orientierung für Interessierte, soll also nicht auswendig gelernt werden ...

Max-Flow Algorithmen nach Veröffentlichungsjahr (Auswahl)
(Quelle: Wikipedia, `http://de.wikipedia.org/wiki/Flüsse_und_Schnitte_in_Netzwerken` und `https://en.wikipedia.org/wiki/Maximum_flow_problem`)

Jahr	Autor(en)	Name, Methode	Laufzeiten
1956	**Ford, Fulkerson**	Ford-Fulkerson, flussvergr. Wege	$\mathcal{O}(m \cdot n \cdot u_{\max})$ *
1969	**Edmonds, Karp**	Edmonds-Karp, kürzeste flussv. Wege	$\mathcal{O}\left(m^2 \cdot n\right)$
1970	**Dinitz**	Algorithmus von Dinitz, Sperrfluss	$\mathcal{O}\left(m \cdot n^2\right)$
1972	**Edmonds/Karp, Dinitz**		$\mathcal{O}\left(m^2 \cdot \log\left(u_{\max}\right)\right)$ *
1973	**Dinitz, Gabow**		$\mathcal{O}\left(m \cdot n \cdot \log\left(u_{\max}\right)\right)$ *
1974	Karzanov	Sperrfluss	$\mathcal{O}\left(n^3\right)$
1978	Malhotra, Pramodh-Kumar, Maheshwari	Sperrfluss, Backward-Forward-Propagation	$\mathcal{O}\left(n^3\right)$
1977	Cherkassky		$\mathcal{O}\left(n^2 \cdot \sqrt{m}\right)$
1980	Galil, Naamad		$\mathcal{O}\left(m \cdot n \cdot (\log n)^2\right)$
1983	Sleator, Tarjan	Dinitz-Algo mit *Dynamic Trees*	$\mathcal{O}\left(m \cdot n \cdot \log n\right)$
1986	**Goldberg, Tarjan**	Goldberg-Tarjan-Algo, **Preflow-Push** generisch	$\mathcal{O}(n^2 \cdot m)$
1986		Preflow-Push, FIFO-Regel oder Highest-Label	$\mathcal{O}(n^3)$
1986	**Goldberg, Tarjan**	Preflow-Push,	$\mathcal{O}\left(m \cdot n \cdot \log(n^2/m)\right)$
1987	Ahuja, Orlin		$\mathcal{O}\left(m \cdot n + n^2 \cdot \log\left(u_{\max}\right)\right)$ *
1990	Cheriyan, Hagerup, Mehlhorn	Preflow-Push (randomisiert)	$\mathcal{O}(n^3/\log n)$
1990	Alon	Preflow-Push	$\mathcal{O}\left(m \cdot n + n^{8/3} \cdot (\log n)\right)$
1992	**King, Rao, Tarjan**	Preflow-Push	$\mathcal{O}\left(m \cdot n + n^{2+\varepsilon}\right)$
1994	**King, Rao, Tarjan**	Preflow-Push	$\mathcal{O}\left(m \cdot n \cdot \log_{\frac{m}{n \cdot \log(n)}}(n)\right)$
1997	**Goldberg, Rao**	Sperrfluss	$\mathcal{O}\left(m^{3/2} \log(n^2/m) \log(u_{\max})\right)$
⋮	⋮	⋮	⋮
2012	**Orlin**	Neuer Algo für $m \leq n^{1+\varepsilon}$, KSR für $m \geq n^{1+\varepsilon}$	$\mathcal{O}\left(mn\right)$

$n = |V|$ = „Anzahl der Knoten“, $m = |E|$ = „Anzahl der Kanten“, $u_{\max}$ = „Maximum der Kapazitätsfunktion u“,
nur brauchbar, wenn alle Kapazitäten ganzzahlig sind. Beachte: $u_{\max} \leq C \leq n u_{\max}$.

(M. Dietzfelbinger, 12.11.2021)

2 Matchings in bipartiten Graphen

2.1 Grundbegriffe

Englischunterricht:

> "*to match*": passend (paarweise) zusammenfügen
> "*matchmaker*": Heiratsvermittler(in)

Beispiel 1: (s. Abb. 1 links und 2 rechts) Wir veranstalten einen Schulausflug mit $2k$ Kindern. Eine Menge von Paaren $E \subseteq \{\{i,j\} \mid i,j \in \{1,\ldots,2k\}, i \neq j\}$ legt fest, wer mit wem im gleichen (Zweier-)Zimmer schlafen kann. Ist es möglich, die Kinder auf genau k Zimmer aufzuteilen?

Beispiel 2 (das Standardbeispiel, s. Abb. 3): U und W sind n-elementige Mengen, die für n Männer und n Frauen stehen. Eine Menge $E \subseteq U \times W$ gibt an, wer mit wem „kann" (was auch immer: im Bus nebeneinander sitzen, Tango tanzen, heiraten). Gibt es eine Möglichkeit, die Personen einander paarweise zuzuordnen, so dass die durch E gegebene Einschränkung eingehalten wird? Technisch: Gibt es eine Teilmenge $M \subseteq E$ mit $|M| = n$ derart, dass jedes $i \in U$ und jedes $j \in W$ in M genau einmal vorkommt? Wenn nicht, finde eine möglichst große Menge von kompatiblen Paaren.

Beispiel 3: Eine Firma beschäftigt n Personen und hat n Tätigkeiten auszuführen. Die Personen sind bei den Jobs unterschiedlich schnell und fehleranfällig.

	Spülen	Putzen	Kochen	Bügeln
Anton	1	2	0	5
Berti	0	3	4	1
Conni	3	1	2	6
Det	2	0	3	4

Wenn Person i Job j ausführt, entsteht Wert w_{ij} (pro Stunde). Wie sollte man die Jobs auf die Personen aufteilen, damit der erzielte Wert in Summe möglichst groß wird? Hier ist nach einem „perfekten Matching" $M \subseteq U \times W$ mit $|M| = n$ gefragt, wobei $U = W = \{1,\ldots,n\}$ gilt, so dass $\sum_{(i,j)\in M} w_{ij}$ möglichst groß ist.

Wir definieren den Begriff „Matching" zunächst für allgemeine Graphen. Damit ist eine Menge von einander nicht berührenden Kanten gemeint. (Passend: Beispiel 1.)

Definition 2.1.1. *$G = (V,E)$ sei ein (ungerichteter) Graph. Ein **Matching** in G ist eine Menge $M \subseteq E$ mit folgender Eigenschaft: wenn (u,v) und (x,w) verschiedene Elemente von M sind, dann ist $\{u,v\} \cap \{x,w\} = \emptyset$.*

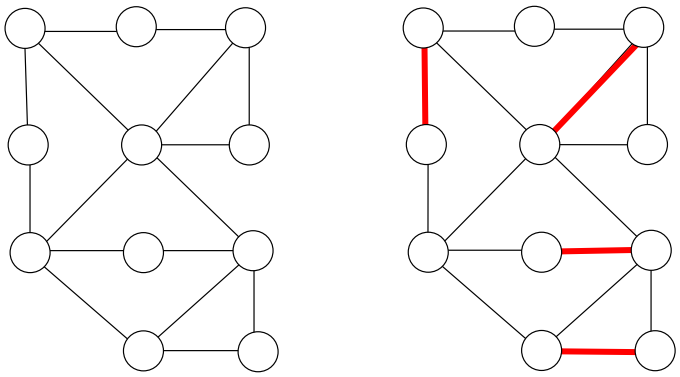


Abbildung 1: Ein Graph und ein (nicht erweiterbares) Matching

Definition 2.1.2. (a) *Ein Matching M in $G = (V, E)$ heißt **nicht erweiterbar** oder **inklusionsmaximal** (engl.: maxim**al** matching), wenn es keine Kante $e \notin M$ gibt, so dass $M' = M \cup \{e\}$ ebenfalls Matching ist.*

(b) *Ein Matching M in $G = (V, E)$ heißt **größtes Matching** oder **maximales Matching** oder **kardinalitätsmaximal** (engl.: maxim**um** matching), wenn es kein Matching M' mit $|M'| > |M|$ gibt.*

(c) *Ein Matching M in $G = (V, E)$ heißt **perfektes Matching**, wenn jeder Knoten $v \in V$ in einer Kante in M vorkommt.*

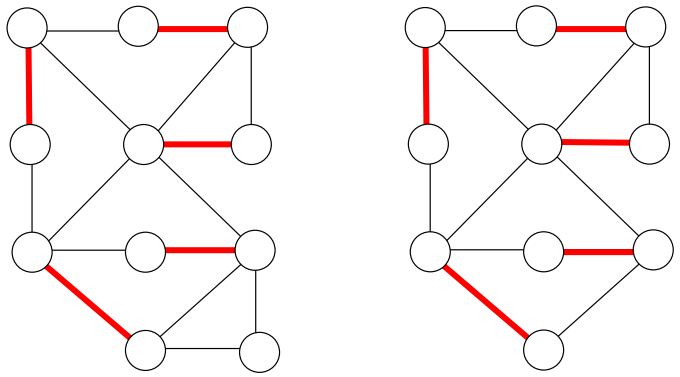


Abbildung 2: Ein (kardinalitäts)maximales Matching und ein perfektes Matching

Die Situation in Beispiel 1 kann als Suche nach einem perfekten Matching in einem allgemeinen Graphen modelliert werden, die in Beispiel 2 als Suche nach einem perfekten oder zumindest (kardinalitäts)maximalen Matching in einem „bipartiten“ Graphen, in Beispiel 3 geht es um ein perfektes Matching in einem vollständigen bipartiten Graphen, das eine gegebene Kostenfunktion maximiert.

Definition 2.1.3. *Ein (ungerichteter) Graph $G = (V, E)$ heißt **bipartiter** oder **paarer** Graph, wenn man V in disjunkte Mengen U und W zerlegen kann, so dass alle Kanten zwischen U und W verlaufen.*

Äquivalent hierzu ist die Bedingung, dass man die Knoten in V so mit zwei Farben („blau“ und „grün“) färben kann, dass jede Kante zwei verschiedenfarbige Endpunkte hat. Es gibt einfache Algorithmen, die einen Graphen G in Zeit $O(|V| + |E|) = O(n + m)$ darauf testen, ob er bipartit ist, und gegebenenfalls eine solche Aufteilung/Färbung berechnen (Übung).

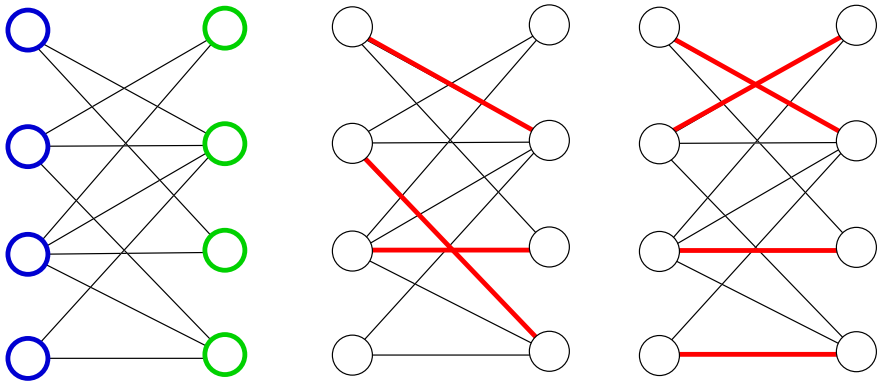


Abbildung 3: Ein bipartiter Graph, mit nicht erweiterbarem Matching, mit perfektem Matching

In diesem Kapitel betrachten wir Algorithmen, die in einem gegebenen Sinn bestmögliche Matchings für bipartite Graphen finden.

2.2 Kostenoptimale Matchings in bipartiten Graphen mit Kantengewichten: Auktionen

(Demange, Gale, Sotomayor, Multi-Item Auctions, Journal of Political Economy, Vol. 94, No. 4, Aug., 1986, pp. 863-872.)

Gegeben ist eine Menge $U = \{1, \ldots, n_\mathrm{b}\}$ (von „Bietern“) und eine Menge $W = \{1, \ldots, n_\mathrm{w}\}$ (von „Waren“), sowie eine „Wertmatrix“

$$(w_{ij})_{1 \le i \le n_\mathrm{b}, 1 \le j \le n_\mathrm{w}}$$

mit Werten $w_{ij} \in \mathbb{N}$. (Wert w_{ij} drückt aus, wie viel Ware j für Bieter i wert ist.) Das Ziel ist, einige der Waren Bietern zuzuordnen, und zwar jedem Bieter höchstens eine, so dass der gesamte realisierte Wert möglichst groß ist. Gesucht ist also ein Matching M im bipartiten Graphen $G = (U \dot\cup W, U \times W)$, so dass $w(M) = \sum_{(i,j) \in M} w_{ij}$ möglichst groß ist.

	Uhr	Fahrrad	Sessel	Reise
Alice	0	3	2	3
Bob	4	0	10	10
Carl	3	9	0	2
Daisy	5	7	0	12

Abbildung 4: Beispielinput für Auktionsmatching. Bieter: $i = 1, 2, 3, 4$ sind Alice, Bob, Carl und Daisy; Waren $j = 1, 2, 3, 4$ sind Uhr, Fahrrad, Sessel, Reise.
Eine mögliche Zuordnung ist $(\text{Alice}, \text{Reise})$, $(\text{Bob}, \text{Sessel})$, $(\text{Carl}, \text{Fahrrad})$, $(\text{Daisy}, \text{Uhr})$; diese hat Wert $3 + 10 + 9 + 5 = 27$. Sie ist nicht optimal.

Man kann offenbar auch das Problem, ein kardinalitätsmaximales Matching M in einem bipartiten Graphen $G = (U, W, E)$ zu finden, in dieser Terminologie formulieren: $w_{ij} \in \{0, 1\}$ für alle i, j, mit $w_{ij} = 1$ genau dann wenn $(i, j) \in E$. Dies stellt den einfachsten Fall dar.

Wir wollen das Problem lösen, indem wir eine Auktion durchführen, und zwar für alle Waren auf einmal. Wie bei einer normalen Auktion gibt es Gebote für die Waren; das aktuelle Höchstgebot für eine Ware j nennen wir den Preis p_j. Das (fiktive) Mindestgebot ist 0: der Preis jeder Ware ist anfangs 0.

Aus der Sicht einer einzelnen Ware j sieht die Auktion ganz normal aus. Die Bieter geben nacheinander Gebote ab. Diese Gebote steigen bei jedem neuen Gebot immer um einen festen Schrittwert $\delta > 0$, so dass das Höchstgebot bei einer Ware nacheinander δ, 2δ, 3δ, ... ist. Der Bieter, der aktuell das Höchstgebot für Ware j abgegeben hat, ist (temporärer) Besitzer, genannt $\text{owner}(j)$. Mit jedem höheren Gebot ändert sich dieser Besitzer. Natürlich dürfen Bieter wiederholt (immer höhere) Gebote für Ware j abgeben. Wenn die Auktion endet, erhält der aktuelle Besitzer $\text{owner}(j)$ die Ware j endgültig (und muss den aktuellen Preis p_j bezahlen).

Aus der Sicht der Bieter sieht es so aus: Jeder darf im Prinzip für jede Ware bieten; wer momentan Besitzer einer Ware ist, darf nicht bieten. Durch die Wahl $w_{ij} = 0$ wird bewirkt, dass sich Bieter i für Ware j nicht interessiert und nie für sie bietet.

Damit es nicht zu sehr durcheinander geht, läuft die Auktion in Runden. In einer Runde darf nur ein Bieter in Aktion treten, und zwar einer, der momentan keine Ware besitzt. Er sucht sich eine Ware aus, die er haben möchte, und gibt sein Gebot ab, sagen wir für Ware j, das um δ höher sein muss als das bisherige Höchstgebot p_j. Bleibt nur noch die Frage, wie ein Bieter entscheidet, für welche Ware er bieten will. Hier stellen wir uns vor, dass Bieter i mit Ware j „Gewinn“ $w_{ij} - (p_j + \delta)$ erzielt. (Er zahlt $p_j + \delta$ und zieht Wert w_{ij} aus dem Besitz.) Nun gibt es zwei Fälle. Wenn der Gewinn für alle Waren negativ ist, steigt Bieter i aus der Auktion aus – er kann

nichts mehr gewinnen. Andernfalls sucht er sich eine Ware j aus, die ihm den größten Gewinn bietet, also eine, für die $w_{ij} - p_j$ am größten ist, und bietet für diese.

Die Auktion endet, wenn kein Bieter mehr bieten möchte, das heißt, wenn jeder Bieter, der keine Ware besitzt, ausgestiegen ist.

Der folgende Algorithmus baut diese Sorte von Auktion nach. Die Bieter, die zur Abgabe eines Gebots bereit sind, halten sich in einer „Wartemenge“ Q auf. (Dabei spielt die Ordnung, in der Bieter aus Q entnommen werden, keine Rolle; manchmal nennt man eine solche Datenstruktur „Halde“.) Wir setzen $\delta = 1/(n_\mathrm{b}+1)$. Der Grund hierfür wird aus der Analyse klar werden.

Algorithm 1: Auktions-Matching

```
1  // Berechnet ein gewichtsmaximales Matching zu (w_ij)_{1≤i≤n_b, 1≤j≤n_w}
2  Initialisierung:
3     Für jede Ware j: setze p_j ← 0 und owner(j) ← NIL
4     Füge alle Bieter in die Wartemenge Q ein
5     Setze δ := 1/(n_b + 1).        // Dann gilt: δ n_b < 1.
6  Bietevorgang:
7     Solange Q nicht leer ist:
8           Entnehme einen Bieter i aus Q
9           Bestimme ein j, das w_ij − p_j maximiert        // bei Gleichheit: beliebig
10          Falls w_ij − p_j > 0:    // falls w_ij ≤ p_j: Bieter i wird inaktiv
11                Füge (bisherigen Besitzer) owner(j) in Q ein
12                owner(j) ← i
13                p_j ← p_j + δ
14    Ausgabe: Menge M aller Paare (owner(j), j) mit owner(j) ≠ NIL
```

Beispiel: Ablauf des Auktionsalgorithmus. Eine Zeile entspricht einer Runde. Jeweils links: Werte w_{ij} und p_j. Jeweils rechts: $w_{ij} - p_j$. Q: in Wartemenge. Hellgrau: $(\mathrm{owner}(j), j)$. Dunkelgrau: in dieser Runde neu zugewiesen.

p_j	0	0	0
Q	1	5	3
Q	1	4	3
Q	0	1	1

p_j	0	0	0
aktiv	1	5	3
Q	1	4	3
Q	0	1	1

p_j	0	0.25	0
	1	5	3
Q	1	4	3
Q	0	1	1

p_j	0	0.25	0
	1	4.75	3
Q	1	3.75	3
aktiv	0	0.75	1

p_j	0	0.25	0.25
	1	5	3
Q	1	4	3
	0	1	1

p_j	0	0.25	0.25
	1	4.75	2.75
aktiv	1	3.75	2.75
	0	0.75	0.75

p_j	0	0.5	0.25
Q	1	5	3
	1	4	3
	0	1	1

p_j	0	0.5	0.25
aktiv	1	4.5	2.75
	1	3.5	2.75
	0	0.5	0.75

p_j	0	0.75	0.25
	1	5	3
Q	1	4	3
	0	1	1

p_j	0	0.75	0.25
	1	4.25	2.75
aktiv	1	3.25	2.75
	0	0.25	0.75

p_j	0	1	0.25
Q	1	5	3
	1	4	3
	0	1	1

p_j	0	1	0.25
aktiv	1	4	2.75
	1	3	2.75
	0	0	0.75

p_j	0	1.25	0.25
	1	5	3
Q	1	4	3
	0	1	1

p_j	0	1.25	0.25
	1	3.75	2.75
aktiv	1	2.75	2.75
	0	-0.25	0.75

p_j	0	1.5	0.25
Q	1	5	3
	1	4	3
	0	1	1

p_j	0	1.5	0.25
aktiv	1	3.5	2.75
	1	2.5	2.75
	0	-0.5	0.75

p_j	0	1.75	0.25
	1	5	3
Q	1	4	3
	0	1	1

p_j	0	1.75	0.25
	1	3.25	2.75
aktiv	1	2.25	2.75
	0	-0.75	0.75

p_j	0	1.75	0.5
	1	5	3
	1	4	3
Q	0	1	1

p_j	0	1.75	0.5
	1	3.25	2.5
	1	2.25	2.5
aktiv	0	-0.75	0.5

p_j	0	1.75	0.75
	1	5	3
Q	1	4	3
	0	1	1

p_j	0	1.75	0.75
	1	3.25	2.25
aktiv	1	2.25	2.25
	0	-0.75	0.25

p_j	0	2	0.75
Q	1	5	3
	1	4	3
	0	1	1

p_j	0	2	0.75
aktiv	1	3	2.25
	1	2	2.25
	0	-1	0.25

p_j	0	2.25	0.75
	1	5	3
Q	1	4	3
	0	1	1

p_j	0	2.25	0.75
	1	2.75	2.25
aktiv	1	1.75	2.25
	0	-1.25	0.25

p_j	0	2.25	1
	1	5	3
	1	4	3
Q	0	1	1

p_j	0	2.25	1
	1	2.75	2
	1	1.75	2
aktiv gibt auf	0	-1.25	0

Q ist leer, Ende. Ausgegebenes Matching: $\{(1,2),(2,3)\}$ mit (optimalem) Wert 8.

Wir überlegen zunächst, dass der Algorithmus das korrekte Ergebnis liefert, *wenn* er eine Ausgabe erzeugt. (Dass nur endlich viele Runden durchgeführt werden, sehen wir weiter unten.)

Die Analyse beruht darauf, dass festgestellt wird, dass alle Bieter am Ende „zufrieden" sind, in folgendem Sinn:

Definition Ein Bieter i heißt „*δ-zufrieden*", wenn gilt:

(i) Wenn i Ware j hat, d. h. $\text{owner}(j) = i$, dann gilt $w_{ij} \geq p_j$ und für alle Waren j' gilt $\delta + (w_{ij} - p_j) \geq w_{ij'} - p_{j'}$;

(ii) Wenn i keine Ware hat, d. h. es gibt kein j mit $\text{owner}(j) = i$, dann gilt für alle Waren j': $w_{ij'} - p_{j'} \leq 0$.

(Intuition: Im ersten Fall darf die Ware j, die Bieter i momentan besitzt, keinen Verlust einbringen und maximiert bis auf eine Abweichung von δ seinen möglichen Gewinn. Im zweiten Fall sind die Preise aller Waren so hoch, dass Bieter i mit ihnen keinen positiven Gewinn machen kann.)

Schleifeninvariante SI:
Bieter, die nicht in der Wartemenge Q sind, sind δ-zufrieden.

Behauptung: SI gilt immer. Natürlich wird der Beweis durch Induktion über Runden des Algorithmus geführt. Am Anfang ist SI erfüllt, weil alle Bieter in Q sind. Nun betrachten wir eine Runde.

Wenn Bieter i aus Q entnommen wird, gibt es zwei Fälle:

1. Fall: Es gibt eine Ware j mit $w_{ij} - p_j > 0$. – Dann wählt der Algorithmus für i eine solche Ware j aus, die zudem $w_{ij} - p_j \geq w_{ij'} - p_{j'}$ für alle Waren j' erfüllt. Die Ungleichungen werden zu[1] $w_{ij} - p_j \geq 0$ und $\delta + w_{ij} - p_j \geq w_{ij'} - p_{j'}$ für alle j' abgeschwächt, weil der Preis p_j um δ steigt (Zeile 13 des Algorithmus). Damit ist (i) für i erfüllt. Solange i Besitzer von j ist, bleibt (i) für i gültig, weil die einzige mögliche Änderung ist, dass die $p_{j'}$ für $j' \neq j$ größer werden.

2. Fall: Für alle Waren j' gilt $w_{ij'} - p_{j'} \leq 0$. – Dann gibt Bieter i auf, und er wird nie mehr in die Wartemenge Q kommen, aber er ist δ-zufrieden, weil für ihn (ii) gilt. Da Preise $p_{j'}$ nur steigen können, bleibt (ii) gültig.

Wenn in der betrachteten Runde Bieter i seine Ware weggenommen wird, dann wandert i nach Q, und SI gilt auch für i.

Damit ist die Behauptung bewiesen.

Es ist offensichtlich, dass der Algorithmus irgendwann einmal anhält. (In jeder Runde steigt entweder ein Bieter (endgültig) aus oder ein Preis erhöht sich um δ. Preis p_j kann nicht größer als $\max_{1 \leq i \leq n_b} w_{ij}$ werden.) Wenn der Algorithmus anhält, muss Q leer sein, und daher sind dann alle Bieter δ-zufrieden. Das folgende Lemma betrachtet diese Situation und liefert die Korrektheit des Algorithmus.

[1] Ein Detail: Preise sind immer Vielfache von δ, und Werte sind ganzzahlig. Wenn also $w_{ij} - p_j > 0$ war und nun $p_j := p_j + \delta$ gesetzt wird, gilt nachher $w_{ij} - p_j \geq 0$.

Lemma 2.2.1. *Sei $M' \subseteq U \times W$ ein beliebiges Matching. Wenn alle Bieter δ-zufrieden sind, dann gilt für das (Algorithmen-)Matching $M = \{(i,j) \mid i = owner(j)\}$:*

$$\delta n_{\mathrm{b}} + \sum_{(i,j)\in M} w_{ij} \geq \sum_{(i,j)\in M'} w_{ij} \ .$$

Bevor wir das Lemma beweisen, bemerken wir, dass daraus die Korrektheit des Algorithmus folgt: Weil alle Gewichte ganzzahlig sind und $\delta n_{\mathrm{b}} < 1$ ist (nach der Wahl von δ), folgt $\sum_{(i,j)\in M} w_{ij} \geq \sum_{(i,j)\in M'} w_{ij}$. Da M' beliebig war, ist das vom Algorithmus gelieferte Matching M optimal.

Beweis von Lemma 2.2.1: Wir bemerken zunächst, dass $p_j > 0$ ist genau dann wenn irgendwann einmal irgendein Bieter Besitzer von j geworden ist. Da Waren nie mehr frei werden, wenn sie einmal gewählt worden sind, heißt dass, dass ein Paar $(\mathrm{owner}(j), j)$ in M vorkommt.

Für Bieter i bezeichne $j_i \in W \cup \{\mathrm{NIL}\}$ die Ware, die ihm in M zugeordnet wird, und $j'_i \in W \cup \{\mathrm{NIL}\}$ die Ware, die ihm in M' zugeordnet wird. Weiter setzen wir $p_{\mathrm{NIL}} = 0$ und $w_{i,\mathrm{NIL}} = 0$. Wir behaupten, dass Folgendes gilt:

$$\delta + (w_{ij_i} - p_{j_i}) \geq w_{ij'_i} - p_{j'_i}. \tag{1}$$

Dies folgt daraus, dass i δ-zufrieden ist, wie man durch Betrachten mehrerer Fälle sieht:

Fall A: $(i, j_i) \in M$ und $(i, j'_i) \in M'$. – Dann folgt (1) aus (i).

Fall B: $j_i = \mathrm{NIL}$ und $(i, j'_i) \in M'$. – Dann ist die linke Seite in (1) gleich δ, und die rechte Seite ist nicht positiv, weil auf i Bedingung (ii) zutrifft.

Fall C: $(i, j_i) \in M$ und $j'_i = \mathrm{NIL}$. – Wegen (i) ist die linke Seite in (1) mindestens δ, die rechte Seite ist 0.

Fall D: $j_i = \mathrm{NIL}$ und $j'_i = \mathrm{NIL}$. – Dann ist die linke Seite in (1) gleich δ, die rechte Seite ist 0.

Durch Summieren von (1) über alle Bieter i erhalten wir:

$$n_{\mathrm{b}}\delta + \sum_{1\leq i\leq n_{\mathrm{b}}} (w_{ij_i} - p_{j_i}) \geq \sum_{1\leq i\leq n_{\mathrm{b}}} (w_{ij'_i} - p_{j'_i}) \tag{2}$$

Wir addieren nun $\sum_{1\leq j\leq n_{\mathrm{w}}} p_j$ zu beiden Seiten von (2) und erhalten Folgendes:

$$n_{\mathrm{b}}\delta + \sum_{1\leq i\leq n_{\mathrm{b}}} w_{ij_i} - \sum_{1\leq i\leq n_{\mathrm{b}}} p_{j_i} + \sum_{1\leq j\leq n_{\mathrm{w}}} p_j \geq \sum_{1\leq i\leq n_{\mathrm{b}}} w_{ij'_i} - \sum_{1\leq i\leq n_{\mathrm{b}}} p_{j'_i} + \sum_{1\leq j\leq n_{\mathrm{w}}} p_j. \tag{3}$$

Auf der linken Seite heben sich die beiden p_j-Summen genau auf, weil die Bedingung $p_j > 0$ gleichbedeutend damit ist, dass es ein i mit $j = j_i$ gibt. Auf der rechten Seite gibt die Summe mit negativem Vorzeichen die Summe der Preise all der Waren an, die in M' vorkommen, die Summe mit positivem Vorzeichen die Summe der Preise aller Waren. Also ist der Gesamtbeitrag der beiden p_j-Summen ≥ 0.

Damit ist Lemma 2.2.1 bewiesen. □

Es fehlt nun noch die Laufzeitanalyse und der Nachweis, dass der Algorithmus anhält. Hierzu bemerken wir Folgendes: In jedem Schleifendurchlauf steigt einer der Preise p_j um δ an oder einer der Bieter „gibt auf“, das heißt, er scheidet dauerhaft aus der Wartemenge Q aus. Kein Wert p_j kann größer als $C := \max_{i,j'} w_{ij'}$ werden. Daher ist die gesamte Anzahl der Schleifendurchläufe nicht größer als

$$n_{\mathrm{b}} + \frac{Cn_{\mathrm{w}}}{\delta} \leq C(n_{\mathrm{b}} + n_{\mathrm{w}})/\delta \leq Cn^2,$$

für $n = n_{\mathrm{b}} + n_{\mathrm{w}}$. Jeder Schleifendurchlauf lässt sich trivial mit Laufzeit $O(n)$ implementieren, was eine Gesamtlaufzeit von $O(Cn^3)$ liefert.

Satz 2.2.2. *Sei n die Gesamtzahl der Knoten im Graphen G und sei C das maximale Kantengewicht. Algorithmus 1 liefert in Zeit $O(Cn^3)$ ein gewichtsmaximales bipartites Matching, für n_{b} Bieter und n_{w} Waren und $n = n_{\mathrm{b}} + n_{\mathrm{w}}$. Wenn die Werte nur 0 und 1 sein können, also ein maximales Matching gesucht wird, ist die Laufzeit $O(n^3)$.*

Bemerkung 2.2.3. (1) Man kann auch größere Werte für δ wählen. Lemma 2.2.1 gilt trotzdem. Das heißt, dass der Wert $w(M) = \sum_{(i,j)\in M} w_{ij}$ additiv nur um δn_{b} vom Optimum abweicht. Dies kann man benutzen, wenn man nicht das exakte Optimum benötigt.

(2) Eine besondere Anwendung gibt es beim Berechnen von maximalen Matchings in einem bipartiten Graphen mit n Knoten und m Kanten. Man setzt $\delta = 1/\lceil\sqrt{n}\rceil$. Wenn der Algorithmus stoppt, ist die Abweichung zwischen $|M|$ und dem Optimum nur $n/\lceil\sqrt{n}\rceil \leq \sqrt{n}$. Die Rechenzeit ist $O(m\sqrt{n})$. Im nächsten Abschnitt werden wir sehen, wie man nun mit $O(\sqrt{n})$ Berechnungen von flussvergrößernden Wegen in einem geeigneten Flussnetzwerk ein kardinalitätsmaximales Matching findet. Der Zeitaufwand hierfür ist nochmals $O(m\sqrt{n})$.

2.3 Kardinalitätsmaximale Matchings in bipartiten Graphen

Im Folgenden ist stets $G = (U \cup W, E)$ mit $E \subseteq U \times W$ ein bipartiter Graph mit Knotenmengen U (links) und W (rechts). Wir möchten in diesem Graphen ein (kardinalitäts)maximales Matching berechnen. Es stellt sich heraus, dass Flussalgorithmen dieses Problem direkt lösen.

Sei $G = (U \cup W, E)$ ein bipartiter Graph, mit disjunkten Mengen U und W.[2] Das zugehörige Flussnetzwerk $\hat{G} = (\hat{V}, \hat{E}, \hat{c})$ entsteht aus G wie folgt: $\hat{V}$ ergibt sich aus $U \cup W$ durch Hinzufügen von zwei neuen Knoten q und s. Die Kantenmenge $\hat{E}$ ist E (von U nach W gerichtet) zusammen mit neuen Kanten (q, u), für $u \in U$, und (w, s), für $w \in W$. Alle Kapazitäten sind 1. Wenn wir nun nur ganzzahlige Flüsse betrachten, dann können an einer Kante nur Flusswert 1 oder Flusswert 0 herrschen.

[2]Oft hat man $U = \{1, \ldots, n_1\}$ und $W = \{1, \ldots, n_2\}$. Man nimmt dann stillschweigend an, dass diese beiden Mengen künstlich disjunkt gemacht worden sind, etwa durch Hinzufügen einer zweiten Komponente zum Index.

Proposition 2.3.1. *Flüsse f in $\hat{G}$ mit ganzzahligen Werten $f(e)$ und Matchings M in G entsprechen einander eineindeutig. Dabei entspricht ein Fluss f mit Wert $w(f)$ einem Matching M mit $|M| = w(f)$. Insbesondere entsprechen maximale Flüsse in $\hat{G}$ genau kardinalitätsmaximalen Matchings in G.*

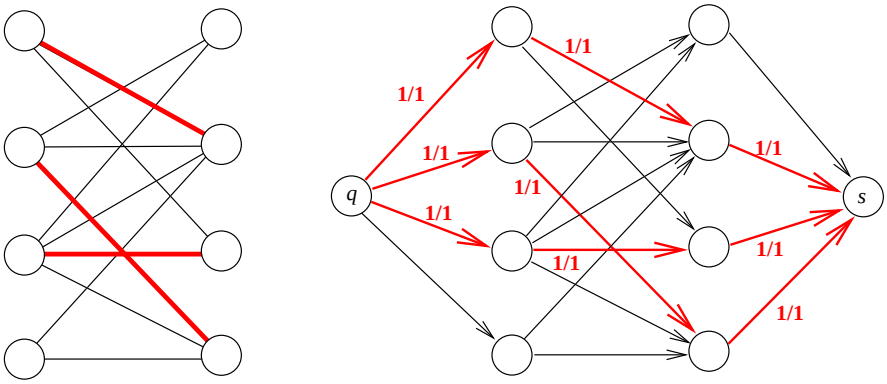


Abbildung 5: Links: Bipartiter Graph mit Matching. Rechts: Entsprechender Flussgraph mit Fluss. Kanten ohne Beschriftung haben Kapazität 1 und Flusswert 0.

Beweis: Wenn ein Matching M gegeben ist, definieren wir einen Fluss $f = f_M$ wie folgt: Für jede Kante $(u, w) \in M$ erhalten die Kanten (q, u), (u, w) und (w, s) Flusswert 1, alle anderen Kanten erhalten Flusswert 0. Es ist klar, dass die Kirchhoff-Gesetze gelten, und es ist klar, dass $w(f_M) = |M|$ ist.

Wenn umgekehrt ein ganzzahliger Fluss f gegeben ist, sieht man unmittelbar, dass dieser 0-1-wertig ist und dass er aus lauter Wegen der Länge 3 von q nach s besteht. Das entsprechende Matching M_f besteht aus den Kanten (u, w) mit $f(u, w) = 1$, für $u \in U$, $w \in W$. □

Damit ergibt sich unmittelbar aus jedem Flussalgorithmus mit der Ganzzahligkeitseigenschaft ein Algorithmus für die Berechnung eines kardinalitätsmaximalen Matchings. (In Frage kommen dafür alle Ford-Fulkerson-Varianten, Edmonds-Karp, Sperrflussvarianten, Dinitz, und alle Versionen der Preflow-Push-Methode.)

2.3.1 Matchingvergrößernde Wege

Definition 2.3.2. *Sei $G = (V, E)$ ein ungerichteter Graph und $M \subseteq E$ ein Matching. Knoten $v \in V$, die nicht auf einer Matchingkante liegen, heißen* ***frei****.*

Ein einfacher Weg $p = (v_0, v_1, \ldots, v_t)$ mit der Eigenschaft, dass sich auf diesem Weg Matching-Kanten mit Nicht-Matching-Kanten abwechseln (d. h. $(v_{i-1}, v_i) \in M \Leftrightarrow (v_i, v_{i+1}) \notin M$) heißt ein ***alternierender Weg (aW)****.*

Ein alternierender Weg $p = (v_0, v_1, \ldots, v_t)$ heißt ***matchingvergrößernder Weg (mvW)****, wenn t ungerade ist und v_0 und v_t frei sind.*

Lemma 2.3.3. *Sei G ein beliebiger Graph. Ein Matching M in G ist kardinalitätsmaximal genau dann wenn es zu G und M keinen matchingvergrößernden Weg gibt.*

Beweis:[3] „⇒“: Wenn M Matching ist und $p = (v_0, v_1, \ldots, v_t)$ ein mvW, dann erhalten wir aus M ein neues, um eine Kante größeres Matching M', indem wir (v_{2i-1}, v_{2i}), $1 \leq i \leq (t-1)/2$, entfernen und (v_{2i}, v_{2i+1}), $0 \leq i \leq (t-1)/2$, hinzufügen.

„⇐“: Sei M ein Matching, das nicht maximal ist. Wähle ein Matching M' mit $|M'| > |M|$. Wir betrachten die symmetrische Differenz $M \oplus M' = (M \cup M') - (M \cap M')$. Diese Menge bildet einen Teilgraphen von G, in dem jeder Knoten Grad 0, 1 oder 2 hat. Daraus folgt, dass es sich um knotendisjunkte Wege und Kreise handelt, auf denen sich M- und M'-Kanten abwechseln. (S. Abb. 6.) Da $M \oplus M'$ mehr M'-Kanten

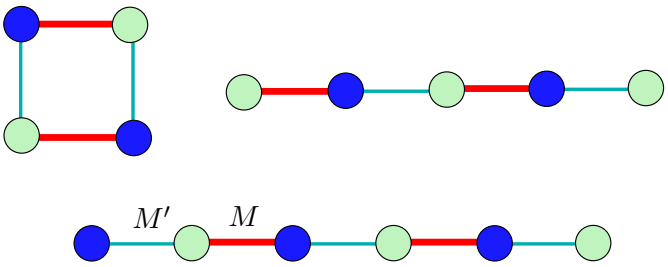


Abbildung 6: Schematische Darstellung von $M \oplus M'$. Kanten in M sind rot, Kanten in M' türkis und etwas dünner gezeichnet. Der Teilgraph $(U \cup W, M \oplus M')$ besteht aus Kreisen und einfachen Wegen, wobei mindestens einer dieser Wege mit einer M'-Kante beginnen und einer M'-Kante enden muss (unten). Dies ist ein matchingvergrößernder Weg bzgl. M.

als M-Kanten enthält, aber Kreise gleich viele M- und M'-Kanten enthalten, muss einer der Wege mehr M'-Kanten als M-Kanten enthalten. Das heißt, dass er in einem M-freien Knoten mit einer M'-Kante beginnt und in einem M-freien Knoten mit einer M'-Kante endet, also ein mvW ist. □

Ab jetzt sei $G = (U \cup W, E)$ bipartit. Der Anwendung der Ford-Fulkerson-Methode auf $\hat{G}$ entspricht folgendes Vorgehen: Gegeben M, suchen wir nach matchingvergrößernden Wegen. Wir beginnen an einem beliebigen freien Knoten $u_0 \in U$. Wenn u_0 nicht isoliert ist, können wir entlang einer Kante (u_0, w_1) zu einem Knoten $w_1 \in W$ gehen. Falls w_1 frei ist, haben wir einen mvW gefunden. Falls nicht, gibt es eine eindeutig bestimmte Kante $(w_1, u_1) \in M$. Nun gilt auch u_1 als gefunden. Wir gehen zu einem Nachbarn $w_2 \neq w_1$ von u_1, falls ein solcher existiert, und prüfen wieder, ob dieser frei ist. Falls ja, haben wir einen mvW gefunden. Falls nein, gibt es eine

[3] Im Fall von bipartiten Graphen könnte man auch die Entsprechung aus Prop. 2.3.1 benutzen und argumentieren, dass flussvergrößernde Wege im Restnetzwerk zu $\hat{G}$ und matchingvergrößernde Wege in G einander genau entsprechen. Dann folgt das Lemma aus dem MaxFlow-MinCut-Theorem. Wir geben hier aber einen einfachen direkten Beweis an, der auch für nicht bipartite Graphen gilt.

eindeutig bestimmte Matching-Kante (w_2, u_2), und wir betrachten u_2 als erreicht. Dieses Vorgehen wird iteriert, aber nicht nur entlang eines Weges, sondern von allen erreichten U-Knoten aus. Ein allgemeiner Iterationsschritt sieht also wie folgt aus: Wenn $U' \subseteq U$ die Menge der bisher gefundenen U-Knoten und $W' \subseteq W$ die Menge der bisher gefundenen W-Knoten ist, sucht man eine Kante (u, w), wobei $u \in U'$ und $w \in W - W'$ ein neuer, bisher unerreichter Knoten ist. Wenn w frei ist, bildet der alternierende Weg, entlang dessen man von u_0 zu w gekommen ist, einen matchingvergrößernden Weg; wenn w nicht frei ist, wird der eindeutig bestimmte Knoten $u' \in U$ mit $(w, u') \in M$ zu U' hinzugefügt. (Beachte: $u' \notin U'$, weil u' nicht frei ist, also $u' \neq u_0$, und weil alle anderen Knoten in U' immer gleichzeitig mit dem anderen Ende ihrer Matchingkante gefunden werden.) In dieser Weise fährt man fort, bis entweder ein freier Knoten auf der W-Seite gefunden wird oder bis alle Nicht-Matching-Kanten, die aus gefundenen Knoten $u \in U'$ herausführen, zu Knoten in W' führen. Im letzten

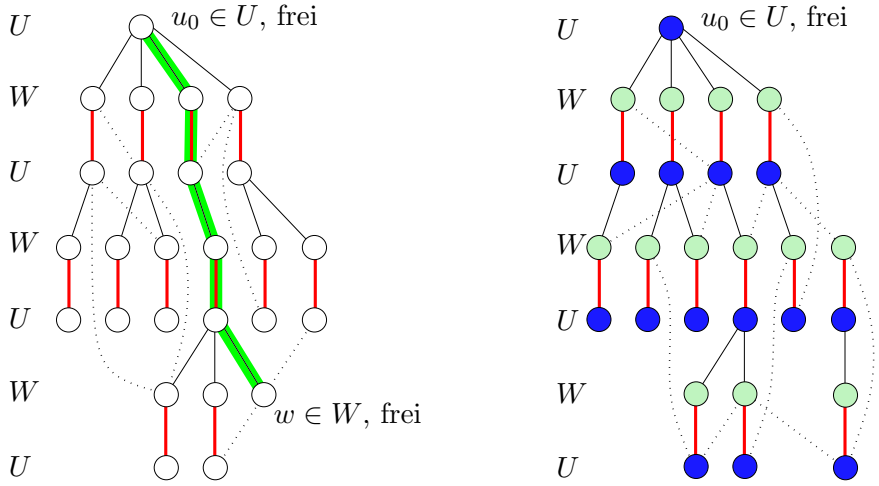


Abbildung 7: Links: Alternierender Baum mit matchingvergrößerndem Weg (grün). Rechts: Alternierender Baum ohne matchingvergrößernden Weg. Baumkanten sind schwarz/dünn (von U nach W) oder rot/dicker (Matchingkanten, von W nach U). Gestrichelte Kanten führen von U-Knoten zu schon entdeckten W-Knoten.

Fall muss man die gleiche Prozedur noch für die anderen freien Knoten in U durchführen. Es entsteht eine Menge von „alternierenden Bäumen“: Die Wurzeln sind frei, auf den Wegen wechseln sich Matchingkanten und Nichtmatchingkanten ab. Sobald ein freies $w \in W$ gefunden wird, kann man aufhören und das Matching vergrößern. Wenn das nie passiert, obwohl von allen freien Knoten in U gestartet wurde, dann gibt es keinen matchingvergrößernden Weg, also ist M maximal.

Bemerkung: Man kann diesen Ansatz („Konstruktion alternierender Bäume“) systematischer organisieren und straffen. Man setzt dazu

$F := \{u \in U \mid u \text{ ist frei}\}$ und

$N := \{u \in U \mid \text{ein Nachbar } w \in W \text{ von } u \text{ ist frei}\}$,

und definiert einen Hilfsgraphen (U, E_M) wie folgt: $(u, u') \in E_M$, wenn es einen Knoten $w \in W$ gibt, so dass $(u, w) \in E - M$ und $(w, u') = (u', w) \in M$. (Diese Definition zieht Vorwärtsgehen von u zu einem neuen Knoten $w \in W$ und das erzwungene Zurückgehen entlang einer eindeutig bestimmten Matchingkante (w, u') zu einem Schritt zusammen.) Wie man leicht sieht, ist dann die Suche nach einem matchingvergrößernden Weg in G äquivalent zur Suche nach einem Weg von F nach N in (U, E_M).

Wir können dann also folgendermaßen vorgehen: Ausgehend von M, berechne (U, E_M) und führe in diesem gerichteten Graphen eine Breitensuche durch, wobei nur Knoten in F als Wurzeln von BFS-Bäumen benutzt werden. (Siehe Vorlesung „Algorithmen und Datenstrukturen“, zu Breitensuche.) Notiere die Vorgänger der erreichten Knoten in ihren Breitensuchbäumen. Sobald ein Knoten $u' \in N$ gefunden wurde, kann der Weg von u' zu der zugehörigen Wurzel $u \in F$ rekonstruiert werden. Dieser Weg wird in einen matchingvergrößernden Weg umgerechnet, und das Matching M kann um eine Kante vergrößert werden.

Ausgehend vom leeren Matching $M_0 = \emptyset$ können höchstens $\frac{1}{2}n$ Matchingvergrößerungen durchgeführt werden. Wenn kein mvW mehr gefunden wird, ist das Matching M maximal. Jede Breitensuche kostet Zeit $O(m)$, für $m = |E|$. Die Gesamtlaufzeit des resultierenden Algorithmus ist $O(nm)$. (Wir kommen später, im Abschnitt über gewichtete Matchings, auf diesen Ansatz zurück.)

2.3.2 Matchings über Sperrflüsse

Die Sperrflussmethode mit der Sperrflussberechnung (z. B. nach Dinitz) führt auf den ersten Blick zu einem Matchingalgorithmus mit Laufzeit $O(n^2m)$ (bzw. $O(n^3)$) bei Knotenzahl n und Kantenzahl m. Im Folgenden wollen wir sehen, dass die Sperrflussmethode in Wirklichkeit zu deutlich besseren Laufzeitschranken führt – sogar besser als $O(nm)$.

Definition 2.3.4. *Ein Flussnetzwerk $G = (V, E, c)$ heißt **0-1-Netzwerk**, wenn alle Kantenkapazitäten den Wert 1 haben. (Nicht vorhandene Kanten haben Kapazität 0.) G heißt **einfaches Netzwerk**, wenn G 0-1-Netzwerk ist und jeder Knoten $v \in V - \{q, s\}$ Eingangsgrad 1 oder Ausgangsgrad 1 hat.*

Beispiel: Das oben zu einem bipartiten Graphen G definierte Flussnetzwerk $\hat{G}$ ist ein einfaches Netzwerk, weil jeder Knoten in U Eingangsgrad 1 hat und jeder Knoten in W Ausgangsgrad 1 hat.

Lemma 2.3.5. *Wenn G ein einfaches Netzwerk ist und f ein ganzzahliger Fluss in G ist, dann ist auch das Restnetzwerk G_f ein einfaches Netzwerk.*

Beweis: (Vgl. Abb. 8.) Wir betrachten einen Knoten v, in den genau eine Kante (u, v) hineinführt. (Knoten v mit Ausgangsgrad 1 behandelt man analog.)

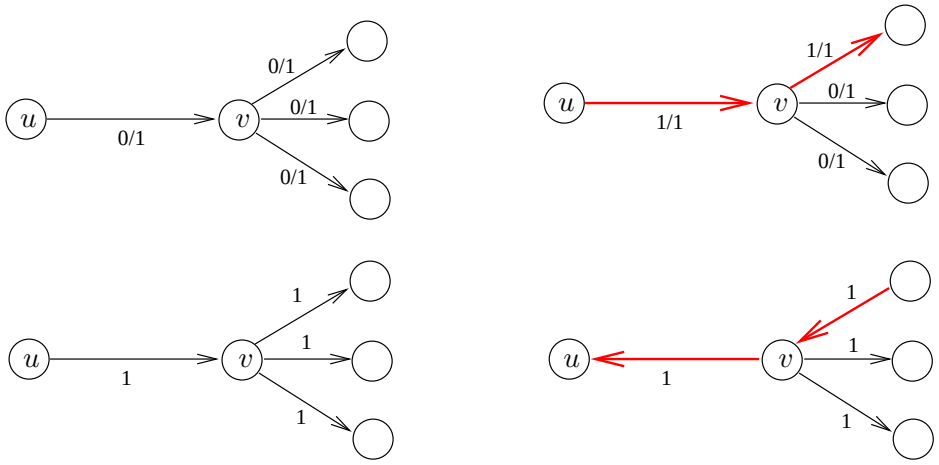


Abbildung 8: Links: Knoten v in einfachem Netzwerk mit $f_{\text{In}}(v) = f_{\text{Out}}(v) = 0$. Auch im Restnetzwerk bleibt der Eingangsgrad 1. Rechts: Knoten in einfachem Netzwerk mit $f_{\text{In}}(v) = f_{\text{Out}}(v) = 1$. Auch im Restnetzwerk ist der Eingangsgrad 1, obgleich zwei Rückwärtskanten auftreten.

1. Fall: $f(u, v) = 0$. In diesem Fall liegt auch auf den Kanten, die aus v hinausführen, Fluss 0. Die Restkapazität auf all diesen Kanten ist also 1; die Kanten in Rückwärtsrichtung haben Restkapazität 0, sind also nicht vorhanden.

2. Fall: $f(u, v) = 1$. In diesem Fall gilt $\text{rest}(u, v) = 0$ und $\text{rest}(v, u) = 1$, in G_f führt also (v, u) aus v hinaus. Es gibt genau eine Kante (v, w) mit $f(v, w) = 1$, für die also $\text{rest}(w, v) = 1$ und $\text{rest}(v, w) = 0$ gilt. Alle anderen Ausgangskanten haben Fluss 0, also auch Restkapazität 1. Also hat auch im Restnetzwerk G_f Knoten v Eingangsgrad 1. □

In einfachen Netzwerken ist es nicht möglich, dass ein ganzzahliger Fluss einen großen Wert hat und gleichzeitig der Abstand zwischen q und s groß ist. Dahinter steckt, dass sich in diesen Netzwerken ganzzahlige Flüsse in knotendisjunkte Flüsse zerlegen lassen. (Diese Tatsache werden wir später auf ein Restnetzwerk anwenden.)

Lemma 2.3.6. *Sei $G = (V, E)$ ein einfaches Netzwerk und sei f ein ganzzahliger Fluss in G mit Wert $w(f)$. Wenn L der Abstand von q nach s in G ist, gilt:*

$$(L - 1) \cdot w(f) \leq |V| - 2.$$

Beweis: Wir betrachten die Kantenmenge $E_{f=1}$, die aus den Kanten e mit $f(e) = 1$ besteht. Weil G einfaches Netzwerk ist, und wegen der Kirchhoff-Regel, gibt es für Knoten $v \in V - \{q, s\}$ nur zwei Fälle: v hat keine Kante in $E_{f=1}$ oder v hat in $E_{f=1}$ Eingangsgrad 1 und Ausgangsgrad 1. Aus solchen Knoten und inzidenten Kanten kann man nur einfache Wege bauen, die bei q beginnen und bei s enden, und Kreise, die weder q noch s berühren. Solche Kreise können wir einfach weglassen

(d. h., wir können den Flusswert auf den Kanten auf 0 setzen), ohne dass sich der Flusswert ändert. Es bleiben $w(f)$ knotendisjunkte Wege von q nach s übrig. Auf jedem dieser Wege liegen außer q und s noch mindestens $L-1$ andere Knoten. Wegen der Disjunktheit gilt: $|V - \{q, s\}| \geq (L-1) \cdot w(f)$, wie behauptet. □

Lemma 2.3.7. *Sei $G = (V, E)$ ein beliebiges Flussnetzwerk und sei f Fluss in G. Sei w^* der maximale Wert eines Flusses in G. Dann existiert im Restnetzwerk G_f ein Fluss f_0 mit Wert $w(f_0) = w^* - w(f)$, so dass $f + f_0$ maximaler Fluss ist.*

Beweis: (Siehe Abb. 9.) Sei f^* ein maximaler Fluss in G, mit $w(f^*) = w^*$. Wir definieren, für $(u, v) \in E$:

$$f_0(u,v) := \begin{cases} f^*(u,v) - f(u,v) & \text{, falls } f(u,v) < f^*(u,v) \\ 0 & \text{, sonst,} \end{cases}$$

und

$$f_0(v,u) := \begin{cases} f(u,v) - f^*(u,v) & \text{, falls } f(u,v) > f^*(u,v) \\ 0 & \text{, sonst,} \end{cases}$$

Man kontrolliert leicht nach, dass f_0 ein Fluss im Restnetzwerk ist, der $f + f_0 = f^*$ und $w(f) + w(f_0) = w(f^*) = w^*$ erfüllt. □

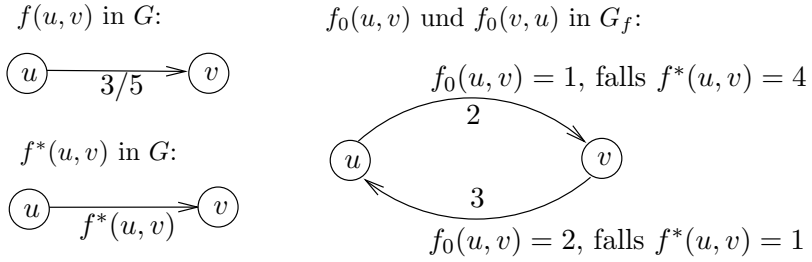


Abbildung 9: Links oben: Kante mit Fluss $f(u, v) = 3$ und Kapazität $c(u, v) = 5$. Links unten: Kante mit „Wunschfluss“ $f^*(u, v)$. Rechts: Kanten im RNW G_f mit Restkapazitäten. Je nachdem ob $f^*(u, v) > f(u, v)$ oder $f^*(u, v) < f(u, v)$ gilt, wird in G_f Fluss $f_0(u, v)$ oder $f_0(v, u)$ auf einen positiven Wert gesetzt.

Wir erinnern uns an die Sperrflussmethode (die hier auf ein einfaches Netzwerk angewendet werden soll und nur ganzzahlige Sperrflüsse benutzt).

Algorithmus 2.3.8 (**(Ganzzahlige) Sperrfluss-Methode**).
INPUT: Flussnetzwerk $G = (V, E, q, s, c)$, c ganzzahlig.

METHODE:
1 $f :=$ der Nullfluss;
2 **repeat**
3 Berechne das Restnetzwerk G_f und das Niveaunetzwerk G'_f;
4 Falls dabei in der ersten BFS s nicht erreicht: **return** f;
5 Konstruiere einen (ganzzahligen) Sperrfluss φ in G'_f;
6 $f := f + \varphi$;

Ohne irgendwelche Modifikationen läuft dieses Verfahren in einfachen Netzwerken besonders schnell.

Satz 2.3.9. *Sei G ein einfaches Netzwerk. Dann gilt, wenn Algorithmus 2.3.8 ausgeführt wird:*

(a) *Das Ergebnis ist ein maximaler Fluss.*

(b) *Der Algorithmus führt maximal $2(\lfloor\sqrt{n}\rfloor + 1)$ Runden aus.*

Beweis: (a) folgt direkt aus der Korrektheit der Sperrflussmethode. Wir beweisen (b). Sei w^* der Wert eines maximalen Flusses. Es ist klar, dass w^* und alle zwischendurch berechneten Flusswerte $w(f)$ ganze Zahlen sind.

Wir teilen die Runden in Algorithmus 2.3.8 in zwei Typen ein, je nach der Situation in Zeile 3. Man unterscheidet, ob für f, den bisher berechneten Fluss, der Wert $w(f)$ noch „weit von w^* entfernt“ ist oder nicht.

Typ 1: $w(f) < w^* - \sqrt{n}$. – Es sei L der Abstand von q nach s im Restnetzwerk G_f, also die Tiefe des Niveaunetzwerks G'_f. Nach Lemma 2.3.7 gibt es einen (ganzzahligen) Fluss f_0 im Restnetzwerk G_f mit Wert $w(f_0) = w^* - w(f)$, also $w(f_0) > \sqrt{n}$. Aus Lemma 2.3.5 wissen wir, dass auch G_f ein einfaches Netzwerk ist. Mit Lemma 2.3.6 folgt $(L-1)w(f_0) < |V| = n$. Mit $w(f_0) > \sqrt{n}$ folgt: $L < \sqrt{n} + 1$. (Intuitiv heißt das: Wenn der Abstand von $w(f)$ zum Zielwert noch „groß“ ist, dann hat das Niveaunetzwerk eher geringe Tiefe.) Im Beweis von Lemma 1.4.4 haben wir gesehen, dass bei der Sperrflussmethode die Tiefe des Niveaunetzwerks in jeder Runde strikt zunimmt. In der ersten Runde ist diese Tiefe mindestens 1. Daher kann es nicht mehr als $\lfloor\sqrt{n}\rfloor + 1$ Runden vom Typ 1 geben.

Typ 2: $w(f) \geq w^* - \sqrt{n}$. – Da mit jeder Runde der Flusswert um mindestens 1 wächst, gibt es maximal $\lfloor\sqrt{n}\rfloor + 1$ Runden vom Typ 2. □

Beobachtung 2.3.10. *In einem einfachen (Niveau-)Netzwerk $G'_f = (V', E')$ kann ein Sperrfluss in Zeit $O(|E'|)$ berechnet werden.*

(Idee: Führe eine Tiefensuche von q aus durch. Sobald s erreicht wird, kann man den gesamten aktiven Weg von q nach s aus dem Graphen streichen, inklusive der ausgehenden Kanten. Auf diese Weise wird jede Kante maximal einmal betrachtet.)

Satz 2.3.11 (Hopcroft-Karp)**.** (*Kardinalitäts-*)*Maximale Matchings in bipartiten Graphen bzw. maximale Flüsse in einfachen Netzwerken können in Zeit* $O(\sqrt{n} \cdot m)$ *berechnet werden, wobei* n *die Knotenzahl und* m *die Kantenzahl ist.*

(Bemerkung: In seiner Originalversion benutzt der Algorithmus von Hopcroft und Karp matchingvergrößernde Wege. In jeder Runde wird eine nicht erweiterbare Menge von kürzestmöglichen matchingvergrößernden Wegen berechnet und das Matching mit allen diesen Wegen erweitert. Genaues Hinsehen zeigt, dass dies genau der Verwendung eines Sperrflusses im Niveaunetzwerk entspricht. Details: Übung.)

2.4 Das Zuordnungsproblem und die Ungarische Methode

Wir verallgemeinern das Problem, ein kardinalitätsmaximales Matching zu finden, auf bipartite Graphen mit Kantengewichten. Zu einem bipartiten Graphen $G = (U \cup W, E)$ mit nichtnegativen Kantengewichten $c\colon E \to \mathbb{R}_0^+$ wollen wir ein Matching $M \subseteq E$ finden, das den Wert $c(M) = \sum_{e \in M} c(e)$ maximiert. Auch für dieses Problem gibt es effiziente Algorithmen, wie wir sehen werden. Der Auktionsalgorithmus aus Abschnitt 2.2 kann benutzt werden, wenn die Gewichte ganzzahlig sind. Allerdings kommt in der Laufzeitschranke das größte Gewicht vor. Wir suchen nach einem Algorithmus, dessen Laufzeit nur von $|U \cup W|$ abhängt.

Die Notation wird einfacher, wenn die disjunkten Mengen U und W die gleiche Größe haben und wenn $E = U \times W$ ist. Dazu fügen wir gegebenenfalls Knoten hinzu, so dass $|U| = |W| = n$ wird, und ergänzen fehlende Kanten mit Gewicht 0. Es entsteht ein vollständiger bipartiter Graph auf der Knotenmenge $V = U \cup W$ mit $2n$ Knoten. In diesem Graphen werden nur noch *perfekte* Matchings gesucht. (Weil die Kosten von „echten“ Kanten nichtnegativ sind, lassen sich maximale nicht-perfekte Matchings durch Hinzunehmen einiger neuer Kanten mit Kosten 0 zu einem maximalen perfekten Matching mit denselben Kosten ergänzen.)

Es ergibt sich das **„Zuordnungsproblem“** (engl.: *assignment problem*).

Max-Zuordnungsproblem:
Seien U und W Mengen mit $|U| = |W| = n$. Gegeben Zahlen („Gewichte“) $c_{uw} = c(u, w) \in \mathbb{R}_0^+$, $u \in U, w \in W$, finde ein perfektes Matching M in $E = U \times W$, so dass

$$c(M) = \sum_{(u,w) \in M} c(u, w)$$

möglichst groß ist.

Beispiel: n Personen sollen n Jobs zugeordnet werden. Wenn Person u Job w ausführt, entsteht (pro Stunde) Wert c_{uw}. Wie soll man Personen und Jobs einander zuordnen, um den gesamten erzeugten Wert zu maximieren?

	Spülen	Putzen	Kochen	Bügeln
Anton	$\underline{1}$	2	0	5
Berti	0	$\underline{3}$	4	1
Conni	3	1	2	$\underline{6}$
Det	2	0	$\underline{3}$	4

(Ist die durch Unterstreichung angedeutete Zuordnung optimal?)

Es ist leicht zu sehen, dass sich das Problem nicht ändert, wenn Kantengewichte auch negativ sein können (ersetze „$c_{uw} \in \mathbb{R}_0^+$“ durch „$c_{uw} \in \mathbb{R}$“). Man addiert einfach $C = -\min\{c_{uw} \mid u \in U, w \in W\}$ zu allen Kantengewichten und erhält eine äquivalente Instanz mit nichtnegativen Einträgen.

Natürlich gibt es auch ein entsprechendes Minimierungsproblem:

Beispiel: n Personen sollen n Jobs zugeordnet werden. Wenn Person u Job w ausführt, fallen (pro Stunde) Kosten c_{uw} an. Wie soll man Personen und Jobs einander zuordnen, um die Gesamtkosten zu *minimieren*?

Min-Zuordnungsproblem:
Seien U und W Mengen mit $|U| = |W| = n$. Gegeben Zahlen $c_{uw} = c(u, w) \in \mathbb{R}$ (ohne Vorzeichenbeschränkung), für $u \in U, w \in W$, finde ein perfektes Matching M in $E = U \times W$, so dass

$$c(M) = \sum_{(u,w)\in M} c_{uw}$$

möglichst klein ist.

Aus einem Minimierungsproblem wird ein äquivalentes Maximierungsproblem, wenn man alle Gewichte mit -1 multipliziert. Ab hier betrachten wir nur noch das Max-Zuordnungsproblem, und nennen es das ***Zuordnungsproblem***. Außerdem legen wir die Bezeichnung $V = U \cup W$ fest.

Der Ansatz für die Lösung ist, mit dem leeren Matching $M = \emptyset$ zu beginnen und in n Runden die Kantenzahl im Matching M zu erhöhen, bis ein perfektes Matching gefunden ist, das das Gesamtgewicht maximiert. Dazu findet man in jeder Runde einen matchingvergrößernden Weg in einem bipartiten **Hilfsgraphen G_ℓ** mit Knotenmenge $U \cup W$ (ohne Kantengewichte!) und vergrößert M entlang dieses Weges, ähnlich wie in Abschnitt 2.3.1. Wenn M nicht perfekt ist, aber G_ℓ keinen matchingvergrößernden Weg mehr hat, kann man den Hilfsgraphen so modifizieren, dass sich das Matching erweitern lässt. Wenn schließlich ein perfektes Matching erreicht ist, erweist sich dieses als optimal. Es sind natürlich noch viele Details zu klären.

Der zentrale Trick liegt darin, den Knoten ***Werte*** oder ***Markierungen*** zu geben:

Definition 2.4.1. *Eine Funktion $\ell\colon V \to \mathbb{R}$ heißt eine* ***zulässige Markierung*** *für den Gewichtssatz $(c_{uw})_{u\in U, w\in W}$, falls*

$$c_{uw} \leq \ell(u) + \ell(w)$$

gilt, für alle $u \in U$, $w \in W$.

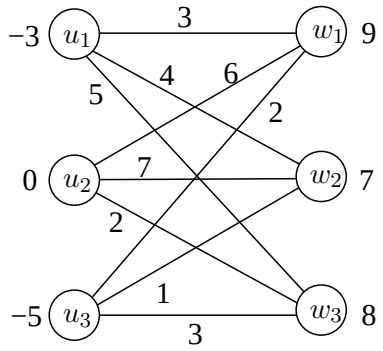


Abbildung 10: Ein vollständiger bipartiter Graph mit Kantengewichten und einer zulässigen Markierung.

Definition 2.4.2. *Sei $\ell\colon V \to \mathbb{R}$ zulässige Markierung. Dann ist der* ***Gleichheitsgraph*** *$G_\ell = (U \cup W, E_\ell)$ durch die Kantenmenge*

$$E_\ell := \{(u, w) \mid c_{uw} = \ell(u) + \ell(w)\}$$

definiert.

Zulässige Markierungen ℓ, der Hilfsgraph G_ℓ und Matchings M in G_ℓ werden verschränkt entwickelt.

Wir können eine kombinatorische Bedingung dafür angeben, dass ein Matching im Gleichheitsgraphen das Zuordnungsproblem löst: *jedes* perfekte Matching in G_ℓ ist geeignet!

Satz 2.4.3 (Kuhn/Munkres)**.** *Sei $\ell\colon V \to \mathbb{R}$ zulässige Markierung und sei $M \subseteq E_\ell$ ein* ***perfektes*** *Matching. Dann ist M optimal, d. h. der Wert $c(M)$ ist größtmöglich unter allen perfekten Matchings über $U \times W$.*

Beweis: Sei M' ein beliebiges perfektes Matching in $E = U \times W$. (Keine Beschränkung

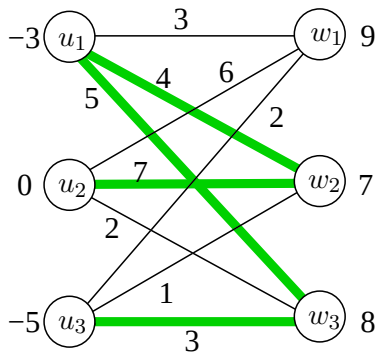


Abbildung 11: Ein vollständiger bipartiter Graph mit Kantengewichten und einer zulässigen Markierung ℓ. Die Kanten des Gleichheitsgraphen sind (grün) hervorgehoben.

auf E_ℓ!) Dann gilt:

$$\begin{aligned} c(M') &= \sum_{(u,w)\in M'} c(u,w) \\ &\overset{(1)}{\leq} \sum_{(u,w)\in M'} (\ell(u)+\ell(w)) \\ &\overset{(2)}{=} \sum_{v\in V} \ell(v) \\ &\overset{(3)}{=} \sum_{(u,w)\in M} (\ell(u)+\ell(w)) \\ &\overset{(4)}{=} \sum_{(u,w)\in M} c(u,w) \\ &= c(M). \end{aligned}$$

((1) gilt, weil ℓ zulässig ist; (2), weil M' perfekt ist; (3) gilt, weil M perfekt ist, und (4), weil $M \subseteq E_\ell$ ist.) □

Damit ist unser Ziel für den Rest dieses Abschnittes klar: wir entwerfen einen Algorithmus, der zugleich eine Markierung ℓ und ein perfektes Matching $M \subseteq E_\ell$ aufbaut.

Wir geben zunächst in Form eines Flussdiagramms den allgemeinen Plan des Algorithmus an (Abb. 12). Die Details werden im Weiteren genauer ausgeführt.

Der Beginn ist einfach: Wir setzen $M = \emptyset$ und $\ell(u) = 0$ für $u \in U$ und $\ell(w) = \max\{c_{uw} \mid u \in U\}$ für $w \in W$. Dann ist klar, dass ℓ zulässig ist und dass $M \subseteq E_\ell$ ist.

Nun nehmen wir an, dass ℓ und $M \subseteq E_\ell$ gegeben sind, wobei M (noch) nicht perfekt ist. Wir entwickeln die Situation weiter, indem wir in $G_\ell = (V, E_\ell)$ nach einem matchingvergrößernden Weg suchen. Dies geschieht systematisch, durch Aufbau eines

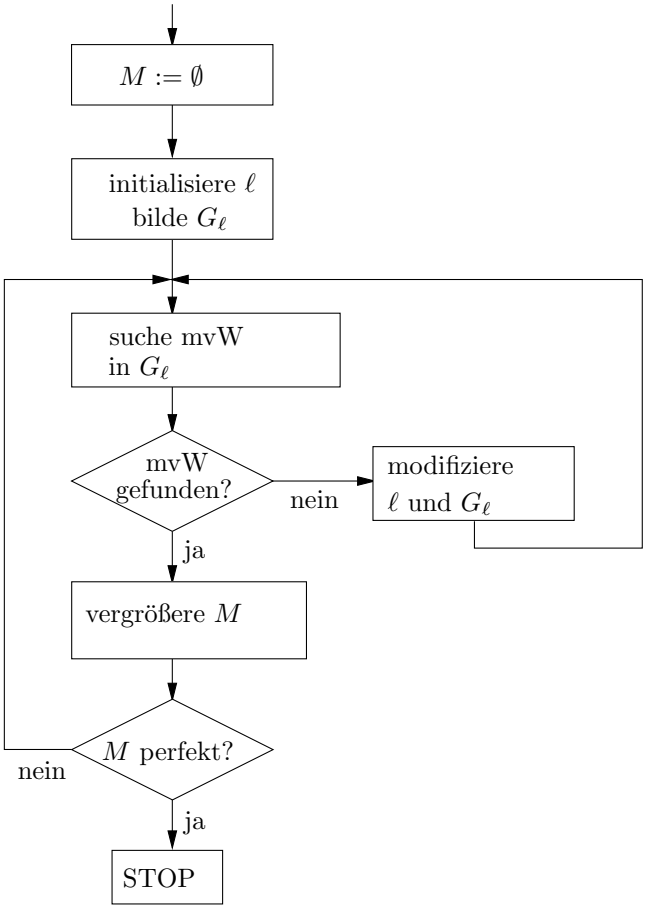


Abbildung 12: Ungarische Methode: Der Plan

alternierenden Baums B im bipartiten Graphen G_ℓ (Abbildungen 13 und 14).

Definition 2.4.4. *Sei* $G_\ell = (V, E_\ell)$ *sowie ein Matching* $M \subseteq E_\ell$ *gegeben. Ein* ***alternierender Baum*** B *besteht aus einem (von* r *weg gerichteten) Baum* B *mit Wurzel* $r \in U$*, wobei Folgendes gilt*:

(i) r *ist frei*;

(ii) *wenn* (u, w) *mit* $u \in U$*,* $w \in W$ *gerichtete Kante in* B *ist, dann ist entweder* w *frei oder es gibt ein* $u' \in U$*, so dass* $(u', w) \in M$ *ist und* (w, u') *Kante in* B *ist.*

Die Menge der Knoten in B*, die in* U *liegen, heißt* S*, die Menge der Knoten in* B*, die in* W *liegen, heißt* T*.*

Wenn G_ℓ und ein nicht perfektes Matching $M \subseteq E_\ell$ gegeben sind, lässt sich ein solcher alternierender Baum ganz leicht aufbauen. Man beginnt mit einem beliebigen

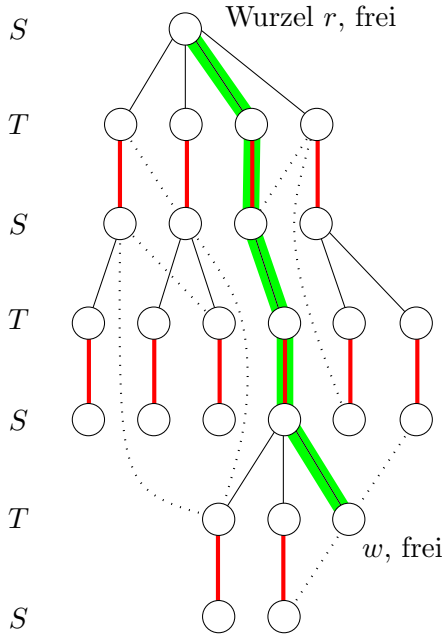


Abbildung 13: Ein (eventuell unvollständiger) alternierender Baum, der einen matchingvergrößernden Weg enthält (grün/schraffiert). Baumkanten sind schwarz/dünn (von S nach T) oder rot/dicker (Matchingkanten, von T nach S). Kanten in E_ℓ, die nicht zum Baum gehören, sind gepunktet dargestellt.

freien Knoten $r \in U$ als Wurzel, und setzt $S := \{r\}$, $T := \emptyset$. Dann iteriert man die folgende Aktion:

> Finde einen Baumknoten $u \in S$, so dass es eine Kante $(u, w) \in E_\ell$ mit $w \in W - T$ gibt. Füge (u, w) zu B hinzu; füge w zu T hinzu.
>
> **1. Fall:** Wenn w frei ist, breche die Konstruktion ab und melde den Weg in B von r zu w als matchingvergrößernden Weg.
>
> **2. Fall:** Wenn w nicht frei ist, dann gibt es einen (eindeutig bestimmten) Knoten $u' \in U$, so dass $(u', w) \in M$ ist. Füge die Kante (w, u') zu B hinzu; füge u' zu S hinzu.

Man beachte, dass in der Konstruktion immer beide Endpunkte einer Matchingkante zusammen entdeckt und in den Baum eingebaut werden. Insbesondere muss der Knoten u' im 2. Fall auch neu sein, weil w neu ist.

Es gibt zwei Möglichkeiten, wie dieser Prozess anhalten kann. Wenn ein freier Knoten $w \in T$ gefunden wird, können wir M anhand des matchingvergrößernden Weges vergrößern (1. Fall, Abb. 13). (Danach wird ein völlig neuer alternierender Baum aufgebaut.) – Der Prozess kann aber auch stoppen, wenn alle $w \in W$, für die es

$u \in S$ und eine Kante $(u, w) \in E_\ell$ gibt, schon in T sind (Abb. 14). In dieser Situation können wir ℓ so verändern, dass der alternierende Baum weitergebaut werden kann. Das wird getan, und nach endlich vielen solchen Modifikationen muss einmal der 1. Fall eintreten.

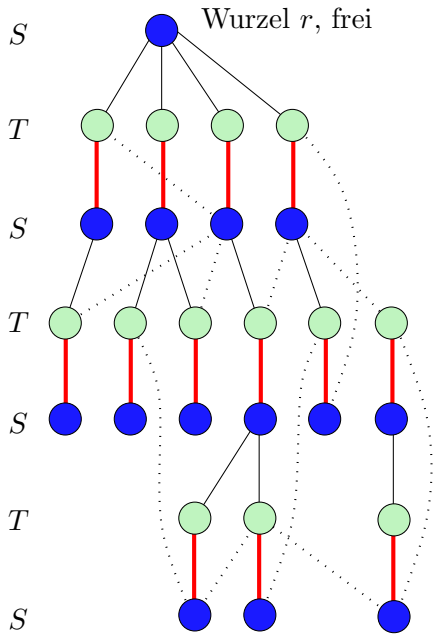


Abbildung 14: Ein alternierender Baum, dessen Konstruktion endet, weil alle Kanten aus S-Knoten zu schon bekannten (im Bild enthaltenen) T-Knoten führen. Die Knotenmenge S ist (dunkel)blau, die Menge T (hell)grün dargestellt. Weil die Wurzel frei ist und alle anderen Knoten paarweise auf Matchingkanten liegen, gilt $|S| = |T| + 1$.

Nehmen wir also an, der Aufbau von B endet, weil alle E_ℓ-Kanten aus S zu Knoten in T führen. Weil die Wurzel r frei ist und alle anderen Knoten in S und T durch die Matchingkanten in B einander paarweise zugeordnet sind, gilt $|S| = |T| + 1$. Insbesondere ist $T \subsetneq W$, und T ist exakt die Menge aller E_ℓ-Nachbarn von Knoten in S.

Bemerkung 2.4.5. In dieser Situation können wir noch Folgendes beobachten:

(a) Wenn $(u, w) \in E_\ell$ eine beliebige Kante ist, dann kann nicht zugleich $u \in S$ und $w \notin T$ sein. Jede Kante in E_ℓ enthält also einen Knoten aus $(U - S) \cup T$ (wir sagen: $(U - S) \cup T$ *überdeckt* E_ℓ).

(b) Wir können in dieser Situation sogar folgern, dass es in $G_\ell = (V, E_\ell)$ kein perfektes Matching geben kann.[4] Wieso? Es gilt $|(U - S) \cup T| = n - |S| + |T| = n - 1$. Daher kann ein Matching M in E_ℓ nicht mehr als $n - 1$ Kanten haben, da $(U - S) \cup T$ erst recht jede der disjunkten Kanten in M trifft.

[4]Es folgt nicht unbedingt, dass M nicht erweiterbar ist. Dies ist hier aber nicht von Bedeutung.

Wenn die Konstruktion des alternierenden Baums B von r aus zu Fall 2 geführt hat, verändern wir ℓ zu einer neuen Markierung (die dann wieder ℓ heißt), wie folgt:

Definiere

$$\alpha := \alpha_\ell(S) := \min\{\ell(u) + \ell(w) - c_{uw} \mid u \in S, w \in W - T\}.$$

Wegen Bemerkung 2.4.5(a) und der Definition des Gleichheitsgraphen sind alle Zahlen, über die hier das Minimum gebildet wird, positiv. Also ist $\alpha > 0$. Weiter gibt es ein Paar $(u, w) \in S \times (W - T)$ mit $\alpha = \ell(u) + \ell(w) - c_{uw}$. Wir definieren:

$$\begin{aligned}
\ell'(u) &:= \ell(u) - \alpha, \text{ für } u \in S;\\
\ell'(w) &:= \ell(w) + \alpha, \text{ für } w \in T;\\
\ell'(v) &:= \ell(v), \text{ für } v \notin S \cup T.
\end{aligned}$$

Das nächste Lemma besagt, dass mit dieser Konstruktion eine neue zulässige Markierung ℓ' entsteht, so dass M und der alternierende Baum B weiter zu ℓ' passen und B erweitert werden kann.

Lemma 2.4.6. *In der eben beschriebenen Situation ist ℓ' wieder zulässig, und es gilt*:

(i) $E_\ell \cap (S \times T) \subseteq E_{\ell'}$.
(*Damit liegen alle Kanten von B in $E_{\ell'}$.*)

(ii) $E_\ell \cap ((U - S) \times (W - T)) \subseteq E_{\ell'}$.
(*Damit liegen auch alle die Kanten von M in $E_{\ell'}$, die B nicht berühren.*)

(iii) *Es gibt mindestens eine Kante $(u, w) \in E_{\ell'}$ mit $u \in S$ und $w \notin T$ (die nicht in E_ℓ war).*

Beweis: Wir betrachten vier Arten von Paaren $(u, w) \in U \times W$.

(a) $u \in S, w \in T$: Es gilt:

$$\ell'(u) + \ell'(w) = (\ell(u) - \alpha) + (\ell(w) + \alpha) = \ell(u) + \ell(w) \geq c_{uw}.$$

Wenn $(u, w) \in E_\ell$, dann gilt auch $\ell'(u) + \ell'(w) = c_{uw}$, also $(u, w) \in E_{\ell'}$.

(b) $u \notin S, w \notin T$: Es gilt trivialerweise $\ell'(u) + \ell'(w) = \ell(u) + \ell(w) \geq c_{uw}$.
Wenn $(u, w) \in E_\ell$ gilt, dann auch $(u, w) \in E_{\ell'}$.

(c) $u \in S, w \notin T$: Nach Definition von α gilt

$$\ell'(u) + \ell'(w) = (\ell(u) - \alpha) + \ell(w) \geq c_{uw}.$$

Für jedes Paar (u, w) mit $\alpha = \ell(u) + \ell(w) - c_{uw}$ gilt $\ell'(u) + \ell'(w) = c_{uw}$. Hiervon gibt es mindestens eines, woraus (iii) folgt.

(d) $u \notin S, w \in T$: Es gilt:

$$\ell'(u) + \ell'(w) = \ell(u) + (\ell(w) + \alpha) > \ell(u) + \ell(w) \geq c_{uw}.$$

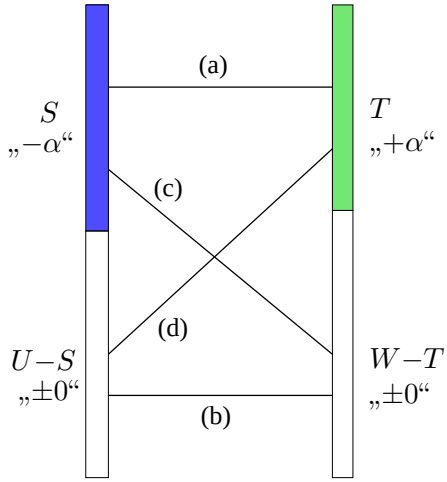


Abbildung 15: Vier Arten von Paaren (u, w) bei der Konstruktion von ℓ' und $E_{\ell'}$. Kanten vom Typ (c) und (d) können nicht in M liegen.

□

Beispiel: In Abb. 16 findet eine Aktualisierung von ℓ statt, die zu einer Änderung des Gleichheitsgraphen führt. Danach kann der alternierende Baum so erweitert werden, dass ein matchingvergrößernder Weg entsteht.

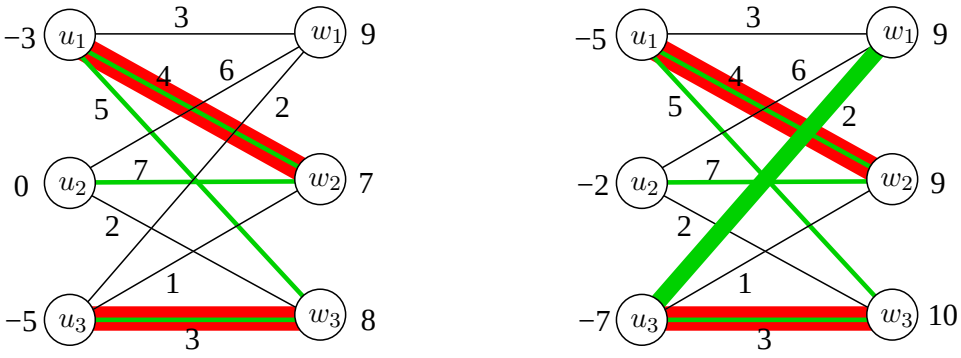


Abbildung 16: Links: Die Konstruktion des alternierenden Baums B von $r = u_2$ aus endet mit $S = \{u_1, u_2, u_3\}$ und $T = \{w_2, w_3\}$. Die Berechnung von α ergibt 2. – Rechts: Nach der Änderung der Markierungen ist (u_3, w_1) neue Kante im Gleichheitsgraphen. Nun findet sich auch ein matchingvergrößernder Weg $(u_2, w_2, u_1, w_3, u_3, w_1)$.

Nach der Änderung von ℓ auf ℓ' benennen wir ℓ' in ℓ um. Aus Lemma 2.4.6 folgt, dass alle Kanten, die für M und für B relevant sind, nach wie vor in E_ℓ sind, dass aber mindestens eine neue Kante $(u, w) \in S \times (W - T)$ dazugekommen ist, die es

erlaubt, B zu vergrößern. Wir bauen B weiter, bis ein matchingvergrößernder Weg gefunden wird oder bis neue, größere Mengen S und T gefunden wurden, über die B nicht hinausgeht. In diesem Fall kann ℓ erneut modifiziert werden, um B wieder zu vergrößern.

Insgesamt führt dies zu Algorithmus 2.4.10[5].

Satz 2.4.7. *Die Ungarische Methode (Algorithmus 2.4.10) löst das Zuordnungsproblem.*

Beweis: Der Algorithmus konstruiert eine zulässige Markierung $\ell\colon V \to \mathbb{R}$ und ein perfektes Matching M in G_ℓ. Nach Satz 2.4.3 ist $c(M)$ maximal. □

Satz 2.4.8. *Die Ungarische Methode (Algorithmus 2.4.10) kann so implementiert werden, dass sie Laufzeit $O(n^4)$ hat.*

Beweis: Es gibt genau n Matchingvergrößerungen, um von 0 auf n Kanten zu kommen. Für eine Matchingvergrößerung muss ein alternierender Baum aufgebaut, eventuell die Funktion ℓ mehrfach verändert und der Baum erweitert werden. Da $|S| = |T|+1$ dadurch wächst, gibt es höchstens n Änderungen von ℓ. Eine solche Änderung und die anschließende Erweiterung von B ist ohne Weiteres in Zeit $O(n^2)$ durchführbar. □

Bemerkung 2.4.9. Mit etwas mehr Sorgfalt kann die Laufzeit sogar auf $O(n^3)$ reduziert werden. Hierzu benötigt man etwas aufwendigere Datenstrukturen, die es erlauben, zu einem Knoten $u \in S$ genau die bisher noch nicht betrachteten Kanten (u, w) des Gleichheitsgraphen zu finden, so dass beim Aufbau und Weiterbau von B nicht wiederholt alle $w \in W$ betrachtet werden müssen. Wir lassen die Details weg.

[5] publiziert von H. Kuhn (1955). Er nannte das Verfahren „Ungarische Methode“, weil sich der Algorithmus auf die Arbeit zweier ungarischer Mathematiker stützte, nämlich Dénes Kőnig and Jenő Egerváry.

Algorithmus 2.4.10 (**Ungarische Methode**)**.**
// Löst das Zuordnungsproblem in vollständigen bipartiten Graphen
INPUT: Kostenmatrix $(c_{uw})_{u\in U, w\in W}$ mit $c_{uw} \in \mathbb{R}$, für $|U| = |W| = n$;

INITIALISIERUNG:
1 $\ell(u) \leftarrow 0$ für alle $u \in U$;
2 $\ell(w) \leftarrow \max\{c_{uw} \mid u \in U\}$ für alle $w \in W$;
3 $M \leftarrow \emptyset$;
ITERATION:
4 **while** $|M| < n$ **repeat**
5 // vergrößere M:
6 wähle freies $r \in U$;
7 $S \leftarrow \{r\}$; $T \leftarrow \emptyset$;
8 stecke r in neue, leere Warteschlange Q;
9 **repeat** // B aufbauen, ggfls. ℓ anpassen (wiederholt)
10 entnehme u aus Q;
11 prüfe Kanten $(u, w) \in E_\ell$ (d. h. mit $\ell(u) + \ell(w) = c_{uw}$) nacheinander:
12 **if** $w \notin T$ **then**
13 füge w zu T hinzu; füge (u, w) zu B hinzu;
14 **if** es gibt eine Kante $(u', w) \in M$
15 **then**
16 füge u' zu S hinzu; füge (w, u') zu B hinzu;
17 füge u' in Q ein;
18 **else** // $w \in W$ ist frei
19 bestimme den mv Weg p von r nach w in B; springe zu Zeile 28;
20 **if** Q ist leer // B kann mit aktuellem E_ℓ nicht erweitert werden
21 **then** // passe ℓ an
22 $\alpha := \min\{\ell(u) + \ell(w) - c_{uw} \mid u \in S, w \in W - T\}$;
23 füge alle $u \in S$ mit $(\exists w \in W - T\colon \ell(u) + \ell(w) - c_{uw} = \alpha)$ in Q ein;
24 **for** $u \in S$ **do** $\ell(u) \leftarrow \ell(u) - \alpha$;
25 **for** $w \in T$ **do** $\ell(w) \leftarrow \ell(w) + \alpha$;
26 // baue B weiter auf: weiter bei Zeile 9
27 // **Ende repeat aus Zeile 9**
28 // hier ist p ein mv Weg in G_ℓ von r zu einem $w \in W$
29 vergrößere M mit p zu M' mit $|M'| = |M| + 1$;
30 $M \leftarrow M'$;
31 // **Ende while aus Zeile 4**
32 Ausgabe: M.

2.5 Stabile Paarungen

Wir stellen uns folgende Situation bei einer Praktikumsbörse vor: n Studierende suchen einen Praktikumsplatz; Firmen bieten n solche Plätze an. Jeder Studierende hat ein eigenes Ranking für die Praktikumsplätze; die Firmen haben für jeden Platz ein Ranking der Studierenden nach deren Eignung für den Platz. Wie soll man eine Zuordnung treffen? Was soll eigentlich das Kriterium für eine „gute“ Zuordnung sein? – Probleme dieser Art stellen sich in vielen Zusammenhängen, zum Beispiel auch bei der Zuteilung von Studienplätzen an Bewerber(innen). Untersucht wurde es im Zusammenhang mit der Verteilung von Assistenzärzten auf Krankenhäuser in den USA. Das Thema ist so wichtig und vielseitig, dass es schon früh ein ganzes Buch darüber gab[6].

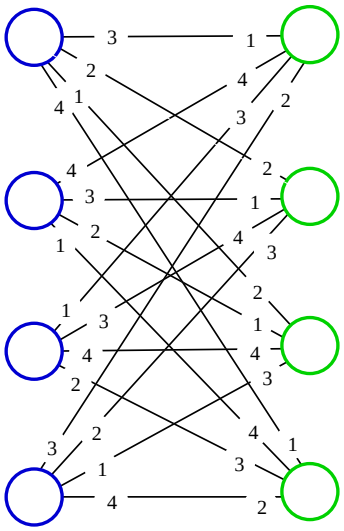


Abbildung 17: Ein vollständiger bipartiter Graph mit Präferenzen an den Knoten

Im etwas allgemeineren Fall hat man einen bipartiten Graphen mit Seiten U und W gegeben; jeder Knoten in U hat eine Rangfolge seiner Nachbarn in W und jeder Knoten in W hat eine Rangfolge seiner Nachbarn in U. Um die Sache nicht zu kompliziert zu machen, befassen wir uns hier nur mit dem vollständigen bipartiten Graphen $(U \cup W, E)$ mit $|U| = |W|$ und $E = U \times W$ und der Suche nach einem *perfekten* Matching $M \subseteq E$, das bestimmte Qualitätskriterien erfüllt. (Siehe Abb. 17. Varianten werden in der Übung betrachtet.) Die traditionelle Terminologie sieht so aus:

$U = \{u_1, \ldots, u_n\}$ ist eine Menge von n Männern;

$W = \{w_1, \ldots, w_n\}$ ist eine Menge von n Frauen;

[6] D. Gusfield und R. W. Irving, The Stable Marriage Problem: Structure and Algorithms. MIT Press, 1989

notfalls kommen alle Paare miteinander aus, aber jede Person hat ein (striktes) Ranking der Personen auf der anderen Seite.

Notation: $r_u \colon W \to \{1, \dots, n\}$ ist das Ranking von $u \in U$, eine injektive Abbildung. (Kleinere Werte bedeuten höhere Wertschätzung.) Analog ist $r_w \colon U \to \{1, \dots, n\}$ das Ranking von $w \in W$.

Wir suchen also ein „gutes“ perfektes Matching. Was soll das sein? Wir möchten erreichen, dass es kein Paar gibt, das in M nicht zusammen ist, aber miteinander zufriedener wäre als es in M ist. (Das ist natürlich nur *ein* mögliches Ziel bei der Herstellung von Zuordnungen.)

Definition 2.5.1. *Sei M ein Matching in $U \times W$. Wir nennen zwei Paare (u, w) und (u', w') in M eine* **Instabilität***, wenn zugleich gilt:*

$r_u(w') < r_u(w)$ *(d. h. u findet w' besser als w),*

$r_{w'}(u) < r_{w'}(u')$ *(d. h. w' findet u besser als u').*

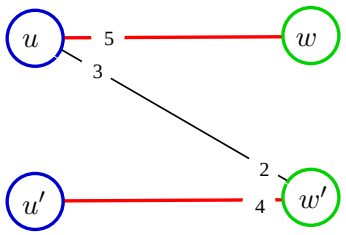


Abbildung 18: Eine Instabilität. Die waagerechten (roten) Kanten sind im Matching, aber die Endpunkte der diagonalen (schwarzen) Kante bevorzugen einander gegenüber den Matching-Partnern

In diesem Falle, so die Vorstellung, würden u und w' ihre Partner verlassen und sich zu einem neuen Paar zusammentun. Aus subjektiver Sicht der beiden ist dies eine Verbesserung, auch wenn das globale Ziel, dass jede(r) eine(n) Partner(in) hat, dadurch zerstört wird.

Definition 2.5.2. *Ein perfektes Matching M heißt eine* ***stabile Paarung****, wenn es keine Instabilität hat.*

Um eine stabile Paarung zu konstruieren, gibt es einen sehr eleganten Algorithmus (von Gale und Shapley[7], 1962), den wir hier betrachten wollen. Er formuliert den Ablauf als eine Folge von Heiratsanträgen, die sich gegenseitig Konkurrenz machen.

[7] D. Gale und L. S. Shapley: „College Admissions and the Stability of Marriage“, American Mathematical Monthly 69, 9–14, 1962. – Zusammen mit Alvin E. Roth erhielt Lloyd S. Shapley 2012 den Nobelpreis in Wirtschaftswissenschaften.

Er ist als Algorithmus 2 wiedergegeben. Er beginnt mit dem leeren Matching und besteht aus Runden. In jeder Runde macht irgendeine Frau w, die momentan nicht verlobt ist, einem Mann u einen Heiratsantrag. Wenn dieser frei ist, wird (u, w) zum Matching hinzugefügt. Wenn u mit w' verlobt ist, kommt es darauf an, ob er w oder w' höher schätzt. Wenn $r_u(w) < r_u(w')$, löst u die Verlobung und verlobt sich mit w (das Paar (u, w') im Matching wird durch (u, w) ersetzt), andernfalls ändert sich nichts. Die einzige Regel ist, dass jede Frau w für sich gesehen die Männer in der Reihenfolge ihres Rankings r_w abarbeitet. Der Algorithmus stoppt, wenn alle Frauen w verlobt sind (wenn das Matching perfekt geworden ist) oder alle Frauen ihre Listen abgearbeitet haben. Man sollte Algorithmus 2 am Graphen aus Abb. 17 durchspielen. Zum Vergleichen findet man das Resultat in Abb. 19.

Algorithm 2: Gale-Shapley: Heiratsantrags-Algorithmus

```
1  // Berechnet eine stabile Paarung M
2  Eingabe: Rankings r_u, u ∈ U der Männer und r_w, w ∈ W, der Frauen.
3  Es wird ein Matching M gebaut, das sich im Lauf der Zeit noch ändert.
4  (u, w) ∈ M heißt: u und w sind „verlobt“.
5  Initialisierung:
6    M := ∅;
7    while ∃w ∈ W(w ist momentan nicht verlobt und
8                 w hat noch nicht allen u einen Antrag gemacht) do
9      wähle ein solches w beliebig;
10     w macht dem ersten Mann u auf ihrer Liste,
11           den sie bisher noch nicht gefragt hat, einen Antrag.
12     1. Fall: u ist frei. – Dann: Verlobung, füge (u, w) zu M hinzu.
13     2. Fall: (u, w') ∈ M, u ist anderweitig verlobt.
14       Fall 2a: r_u(w') < r_u(w): u lehnt ab, keine Änderung.
15       Fall 2b: r_u(w) < r_u(w'): u nimmt an, M := (M − {(u, w')}) ∪ {(u, w)}.
16         (w' ist nicht mehr verlobt.)
17   Ausgabe: M.    „Massenhochzeit“
```

Es stellen sich Fragen nach Korrektheit und Zeitaufwand.

Beobachtungen: (i) Sobald ein Mann $u \in U$ einmal einen Antrag bekommen hat, bleibt er immer verlobt. Seine Partnerinnen können wechseln, aber er wechselt immer nur zu Partnerinnen, die ihm nach seiner Rangliste lieber sind: Einen Wechsel von (u, w') zu (u, w) gibt es nur, wenn $r_u(w') > r_u(w)$ gilt.
(ii) Eine Frau $w \in W$ kann auch wieder frei werden, wenn sie einmal mit einem $u \in U$ verlobt war. Wenn sie später wieder einen Partner u' findet, dann hat dieser aus ihrer Sicht einen schlechteren Rang als der vorherige: $r_w(u') > r_w(u)$. Dies liegt einfach daran, dass sie ihre Anträge in dieser Reihenfolge stellt.

Lemma 2.5.3. *Algorithmus 2 terminiert nach maximal n^2 Schleifendurchläufen.*

Beweis: Nach spätestens n^2 Durchläufen haben alle n Frauen ihre Ranglisten der

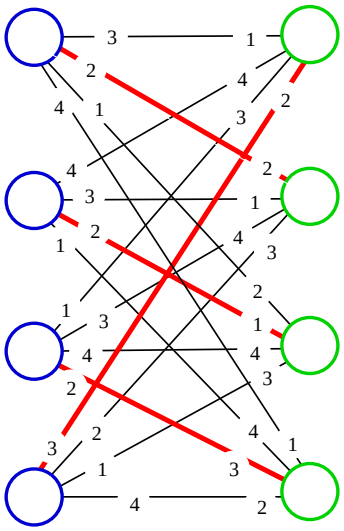


Abbildung 19: Resultat des Gale-Shapley-Algorithmus

Länge n abgearbeitet. □

Es ist auf den ersten Blick überhaupt nicht klar, dass am Ende jede Person eine(n) Partner(in) hat. Hierzu benutzt man ein Kardinalitätsargument.

Lemma 2.5.4. *Wenn Algorithmus 2 terminiert, dann ist M (die verlobten Paare) ein perfektes Matching.*

Beweis: Annahme: Der Algorithmus terminiert, aber M ist nicht perfekt. Weil $|U| = |W|$ gilt, muss es eine Frau w geben, die frei ist. Weil die Schleife beendet ist, hat w jedem Mann einen Antrag gemacht. Nach Beobachtung (i) ist daher jeder Mann verlobt, also ist $|M| = n$, Widerspruch. □

Satz 2.5.5. *Die Ausgabe M von Algorithmus 2 ist eine stabile Paarung.*

Beweis: Wir betrachten zwei beliebige Paare $(u, w), (u', w') \in M$ und zeigen, dass diese keine Instabilität bilden. Wenn $r_u(w') > r_u(w)$ gilt, ist nichts zu zeigen. Also können wir

$$r_u(w') < r_u(w)$$

annehmen. Wegen Beobachtung (i) kann w' niemals u einen Antrag gemacht haben. (Zum Zeitpunkt eines solchen Antrags wäre u entweder frei gewesen oder wäre mit einer Frau w'' verlobt gewesen, für die $r_u(w'') \geq r_u(w) > r_u(w')$ gilt. In beiden Fällen hätte u den Antrag angenommen. Dann müsste aber die endgültige Partnerin w von u bei u mindestens so beliebt sein wie w', d. h., es müsste $r_u(w) \leq r_u(w')$ gelten.) Aus der Tatsache, dass w' bei der Abarbeitung ihrer Liste nicht bis zu u gekommen ist, folgt nach Beobachtung (ii), dass $r_{w'}(u') < r_{w'}(u)$ gilt. Also bilden (u, w) und $(u', w') \in M$ *keine* Instabilität. □

Wir wollen unsere Überlegungen noch ein wenig erweitern. In Zeile 9 des Algorithmus wird die Frau w, die den nächsten Antrag macht, beliebig gewählt. Hat die Reihenfolge, in der die Anträge gemacht werden, Einfluss auf das Ergebnis? Kurioserweise ist das Ergebnis immer das gleiche – der „Nichtdeterminismus“ im Algorithmus bleibt ohne Auswirkungen. Algorithmen dieser Art sind natürlich besonders interessant.

Definition 2.5.6. *$u \in U$ heißt ein **möglicher Partner** für $w \in W$, falls es eine stabile Paarung M mit $(u, w) \in M$ gibt.* (Nach Satz 2.5.5 gibt es eine stabile Paarung, also hat jedes $w \in W$ mindestens einen möglichen Partner.)

$$\text{best}(w) := \text{der mögliche Partner } u \text{ von } w, \text{ der } r_w(u) \text{ minimiert.}$$

(*Der **beste** mögliche Partner von w.*)

$$M^* := \{(\text{best}(w), w) \mid w \in W\}.$$

Satz 2.5.7. *Die Ausgabe M von Algorithmus 2 ist M^*.*

Bevor wir den Satz beweisen, machen wir einige Bemerkungen.
(a) Bemerkenswert ist zunächst, dass der Nichtdeterminismus in Algorithmus 2 keine Auswirkungen auf das Ergebnis hat.
(b) Von der Definition her ist zunächst nicht einmal klar, dass M^* ein Matching ist.
(c) Alle Frauen bekommen gleichzeitig den besten Partner, den sie unter der Einschränkung „stabile Paarung“ (ein globales „soziales Gut“) überhaupt bekommen können.
(d) (Siehe Übung:) Alle Männer bekommen gleichzeitig die schlechteste mögliche Partnerin ...

Beweis von Satz 2.5.7: Indirekt. *Annahme*: Es gibt eine Ablaufreihenfolge $\mathcal{R}$ des Algorithmus von Gale-Shapley, die eine von M^* verschiedene stabile Paarung M erzeugt.

Nach Annahme und der Definition von M^* gibt es in M ein Paar (u, w) mit $u \neq \text{best}(w)$. Weil (u, w) in der stabilen Paarung M vorkommt, ist u möglicher Partner von w. Im Algorithmus ist festgelegt, dass w den Männern ihre Anträge in der Reihenfolge wachsender r_w-Werte (d. h. geringer werdender Wertschätzung) macht. Daraus folgt, dass w im Verlauf von $\mathcal{R}$ ihrem besten möglichen Partner best(w) einen Antrag gemacht hat, dieser sie aber abgewiesen hat oder sich später zugunsten einer anderen Frau von ihr getrennt hat. Damit wissen wir, dass es im Ablauf $\mathcal{R}$ einen Zeitpunkt t_0 gibt, zu dem *zum ersten Mal* einer Frau w_0 und einem möglichen Partner u_0 von w_0 Folgendes passiert:

w_0 macht u_0 einen Antrag, wird aber abgewiesen, da u_0 schon mit einer Frau w' verlobt ist, die er besser als w_0 findet
oder
w_0 ist bisher mit Mann u_0 verlobt, wird aber von diesem verlassen, da u_0 von einer Frau w' einen Antrag bekommt, die er besser als w_0 findet.
In beiden Fällen gilt also:

$$r_{u_0}(w') < r_{u_0}(w_0). \tag{4}$$

Wegen der Minimalität von t_0 und wegen der Anfragereihenfolge bei den Frauen ist u_0 nicht nur irgendein möglicher Partner von w_0, sondern sogar der beste mögliche Partner, d. h. $u_0 = \text{best}(w_0)$.

Weil u_0 möglicher Partner von w_0 ist, gibt es eine stabile Paarung M_0 mit $(u_0, w_0) \in M_0$. In M_0 kommt auch w' in einem Paar vor, etwa $(u', w') \in M_0$.

Wir betrachten nun die Geschichte von w' in $\mathcal{R}$: Unmittelbar nach Zeitpunkt t_0 ist w' mit u_0 verlobt. Nach Wahl von t_0 wurde w' selbst vor t_0 noch nie von einem möglichen Partner zurückgewiesen oder „entlobt“. Nun ist u' ein solcher möglicher Partner von w' (weil $(u', w') \in M_0$ für die stabile Paarung M_0 gilt). Es folgt, dass in $\mathcal{R}$ w' mit u' vor dem Zeitpunkt t_0 nichts zu tun hatte, das bedeutet, dass

$$r_{w'}(u_0) < r_{w'}(u') \tag{5}$$

gelten muss. Die Relationen (4) und (5) zusammen besagen, dass die Paare (u_0, w_0) und (u', w') eine Instabilität in der stabilen Paarung M_0 bilden (siehe Abb. 20), ein Widerspruch. □

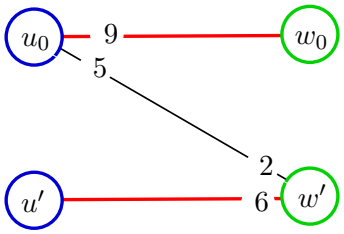


Abbildung 20: Die waagerechten (roten) Kanten gehören zu M_0. Mann u_0 schätzt w' mehr als seine M_0-Partnerin w_0 (nach (4)); Frau w' schätzt u_0 mehr als ihren M_0-Partner u' (nach (5)). Dies ist eine Instabilität. (Die konkret angegebenen Ränge sind nur Beispiele.)

(M. Dietzfelbinger, 26. Januar 2022)

3 Amortisierte Analyse

Wir betrachten hier ein Analyseproblem, das oft bei Datenstrukturen, mitunter auch in anderen algorithmischen Situationen auftritt. Angenommen, wir haben eine Datenstruktur, die einen Datentyp implementiert, z. B. einen Stack, eine Queue, ein Wörterbuch, eine Priority Queue. Bei solchen Datenstrukturen wird immer eine Folge $\mathrm{Op}_1, \ldots, \mathrm{Op}_n$ von Operationen ausgeführt. Diesen Operationen sind „Kosten“ $c_1, \ldots, c_n$ zugeordnet. (Meistens ist c_i eine vereinfachte Version der Rechenzeit für Operation Op_i, so geplant, dass die Rechenzeit $O(c_i)$ ist.) Die Gesamtkosten sind

$$C_n = c_1 + \cdots + c_n,$$

die Gesamtrechenzeit dann $O(C_n)$. Mit dem Ausdruck „Amortisierte Analyse“ ist die Bestimmung einer Abschätzung von C_n gemeint, oder Techniken hierfür.

Wir betrachten in diesem Kapitel ad-hoc-Methoden („*Aggregationsmethode*“) und zwei allgemeinere Techniken, die „*Bankkontomethode*“ und die „*Potenzialmethode*“. In späteren Abschnitten werden wir diese Methoden auf Datenstrukturen zur Implementierung von Priority Queues anwenden.

3.1 Die Aggregationsmethode

***Beispiel* 1:** Stack mit „*multipop*“. – Wir betrachten die bekannte Stack-Datenstruktur (Vorlesungen „Algorithmen und Programmierung“ und „Algorithmen und Datenstrukturen“). Diese besitzt die folgenden Operationen:

„*empty*“ zur Erzeugung eines leeren Stacks (die im Folgenden ignoriert wird),

„*push*“ zum Einfügen oben auf dem Stack,

„*pop*“ zum Entfernen und Ausgeben des obersten Elements (nur bei nichtleerem Stack),

„*isempty*“ zum Test, ob der Stack leer ist.

Alle diese Operationen haben Aufwand $O(1)$. Wir schreiben ihnen jeweils Kosten 1 zu. Nun kommt eine weitere Operation dazu:

„*multipop*(k)“ für $k \geq 1$ entfernt die obersten k Einträge und gibt sie aus. Falls die Stackhöhe p mindestens 1, aber kleiner als k ist, werden alle Einträge entfernt (kein Fehler!). Der Aufwand hierfür ist $O(\min\{k, p\})$; wir geben dieser Operation Kosten $\min\{k, p\} \geq 1$.

Behauptung: Die Kosten von n Operationen $\mathrm{Op}_1, \ldots, \mathrm{Op}_n$, startend mit einem leeren Stack, sind kleiner als $2n$.
Beweis: Wir können *isempty*-Operationen ignorieren, da sie Kosten 1 haben und den Stack nicht ändern. Wenn es n *push*-Operationen gibt, sind die Kosten n. Wenn es *pop*- und *multipop*-Operationen gibt, dann kann die Gesamtzahl der dadurch vom Stack entfernten Einträge nicht größer sein als die Anzahl der mit „*push*“ eingefügten Einträge: dies sind maximal $n-1$ viele. Nun gibt die Anzahl der entfernten Einträge ganz genau diese Kosten an. Einfügungen und Löschungen zusammen haben also Kosten nicht höher als $(n-1)+(n-1) < 2n$. □

Was haben wir gemacht? Wir haben die spezielle Struktur der Operationen benutzt, um mit einer geschickten Argumentation die Summe der Kosten zu beschränken. So etwas nennen wir „Ad-hoc-Methode“ oder „Aggregationsmethode“. (Natürlich ist dies eigentlich keine Methode, sondern man muss sich in jeder Situation wieder etwas Neues einfallen lassen.)

***Beispiel* 2:** Hochzählen eines Binärzählers.
Wir stellen uns vor, wir wollen auf einem Zähler, der Zahlen in Binärdarstellung darstellt, von 0 bis n zählen. Die Anzahl der Stellen des Zählers soll ausreichend sein. Am Anfang wird der Zähler (kostenlos) auf 0 gestellt. Dann wird n-mal eine Inkrementierungsoperation ausgeführt: $\mathrm{Op}_1, \ldots, \mathrm{Op}_n$, wobei Op_i die Erhöhung von $i-1$ auf i ist.

Kosten für eine Erhöhung: Die Anzahl der geänderten Bits.
(Von 110<u>0111</u> auf 110<u>1000</u> sind die Kosten 4; von 1101000 auf 1101001 sind sie 1.)

Wenn c_i die Kosten von Op_i sind, was ist dann $C_n = c_1 + \cdots + c_n$? Wir betrachten den Anfang der Entwicklung. In der folgenden Tabelle sind immer die Bits im Zähler unterstrichen, die sich im letzten Schritt geändert haben.

i	Zählerstand	Kosten
–	0	–
1	$\underline{1}$	1
2	$\underline{10}$	2
3	1$\underline{1}$	1
4	$\underline{100}$	3
5	10$\underline{1}$	1
6	1$\underline{10}$	2
7	11$\underline{1}$	1
8	$\underline{1000}$	4
9	100$\underline{1}$	1
10	10$\underline{10}$	2
11	101$\underline{1}$	1
12	1$\underline{100}$	3
13	110$\underline{1}$	1
14	11$\underline{10}$	2
15	111$\underline{1}$	1
16	$\underline{10000}$	5
17	1000$\underline{1}$	1

Die Kosten folgen einem regelmäßigen Muster, aber wie soll man sie summieren? Wir bemerken, dass jede unterstrichene Ziffer in der Tabelle genau 1 kostet. Nun schätzen wir die Anzahl der unterstrichenen Ziffern geschickt ab.

Für $\ell \geq 0$ sei u_ℓ die Anzahl der unterstrichenen Ziffern in Bitposition ℓ, die zur Zweierpotenz 2^ℓ gehört. Die letzte Ziffer (Position $\ell = 0$) ist in jeder Zeile unterstrichen, also ist $u_0 = n$. Die vorletzte Ziffer (Position $\ell = 1$) ist in jeder zweiten Zeile $(2, 4, 6, \dots)$ unterstrichen, beginnend mit der zweiten, also ist $u_1 \leq n/2$. Die drittletzte Ziffer ist in jeder vierten Zeile $(4, 8, 12, \dots)$ unterstrichen, beginnend mit der vierten, also ist $u_2 \leq n/4$. Allgemein: $u_\ell \leq n/2^\ell$, für $\ell = 0, 1, 2, \dots$. Damit:

$$\text{Gesamtkosten} \;<\; \sum_{\ell \geq 0} \frac{n}{2^\ell} = n \cdot 2 = 2n.$$

Beim Hochzählen eines Binärzählers von 0 auf n werden insgesamt weniger als $2n$ Bits gekippt.

Anmerkung: Später werden wir sehen, wie sich die Zahl der gekippten Bits ganz genau bestimmen lässt.

***Beispiel* 3:** Anzahl der Vergleiche beim Heapaufbau.
Wir betrachten binäre Heaps wie in der Vorlesung „Algorithmen und Datenstruktu-

ren“ (Bachelor)[1]. Die Frage ist, wie viele Vergleiche nötig sind, um aus einem beliebigen Array $A[1..n]$ einen binären Heap zu bauen. Die effizienteste Prozedur sieht wie folgt aus:

for `i` **from** $\lfloor n/2 \rfloor$ **downto** 1 **do** *bubble_down*(`i`, n).

Dabei führt *bubble_down*(i, n) auf jedem Niveau des Heaps unterhalb von Knoten i bis zu 2 Vergleiche aus.

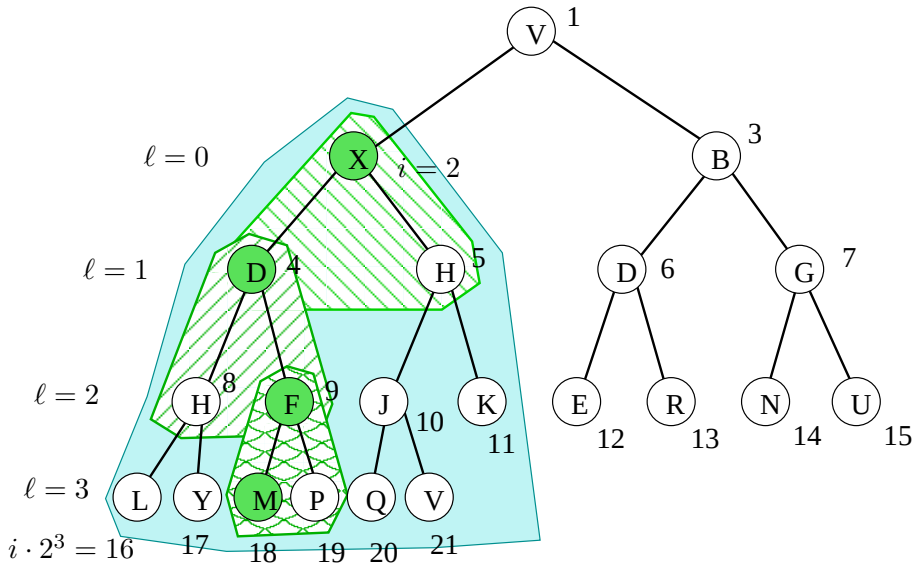


Abbildung 1: Aufruf *bubble_down*$(2, 21)$ führt zu dreimal 2 Vergleichen, auf Level $1, 2, 3$ unter Knoten $i = 2$. Ob Level ℓ noch nicht leer ist, wird durch die Ungleichung $i \cdot 2^\ell \leq n$ entschieden, weil der Knoten im Unterbaum mit Wurzel i, der am weitesten links sitzt, die Form $i \cdot 2^s$ hat (im Beispiel: $2 \cdot 2^3 = 16$).

Wir wollen die Gesamtzahl der Vergleiche nach oben durch ein C_n abschätzen. Wir betrachten den Aufruf *bubble_down*(i, n). Dabei werden entlang eines Wegs von Knoten i zu einem Blatt jeweils der Knoteninhalt mit dem kleineren Kind verglichen und eventuell vertauscht. Dies kostet zwei Vergleiche pro Level unterhalb von i. Der relevante Teil des Heaps für den Aufruf *bubble_down*(i, n) ist also der Teilbaum mit Wurzel i. (Beispiel: Abb. 1.) Level ℓ existiert in diesem Baum nur, wenn $i \cdot 2^\ell \leq n$

[1] `https://moodle2.tu-ilmenau.de/course/view.php?id=2465`, KW24, Folien Kapitel 6, Seite 67/68.

ist, weil Knoten $i \cdot 2^\ell$ der kleinste Knoten auf Level ℓ im Teilbaum unter i ist und n der größte Knoten im gesamten Heap ist. Damit:

$$C_n \leq \sum_{1 \leq i \leq \lfloor n/2 \rfloor} \; \sum_{\ell \geq 1 \colon i \cdot 2^\ell \leq n} 2.$$

Der Trick ist hier die Vertauschung der Summationsreihenfolge. Dabei können wir die ℓ-Summe auch unbegrenzt laufen lassen:

$$C_n \leq 2 \cdot \sum_{\ell \geq 1} \; \sum_{1 \leq i \leq n/2^\ell} 1 = 2 \cdot \sum_{\ell \geq 1} \lfloor n/2^\ell \rfloor < 2 \cdot \sum_{\ell \geq 1} n/2^\ell = 2n.$$

(Die letzte Gleichheit folgt daraus, dass $\sum_{\ell \geq 1} 2^{-\ell} = 1$ ist.)

Fazit: Um aus einem Array $A[1..n]$ einen Heap zu machen, sind weniger als $2n$ Schlüsselvergleiche nötig.

3.2 Die Bankkontomethode

Anschauliche Grundidee bei dieser Methode ist, dass der Benutzer für jede Operation der Datenstruktur eine pauschale Gebühr bezahlen muss. Die gezahlten Beiträge werden teilweise zum Begleichen der Kosten benutzt und teilweise zu einem Guthaben „angespart“, das dann später zum Bezahlen teurer Operationen benutzt werden kann. Die pauschale Gebühr für Op_i heißt „amortisierte Kosten“ a_i. Diese muss (geschickt) festgelegt werden. Bei der Ausführung von Operation Op_i gibt es dann zwei Fälle: Wenn $a_i \geq c_i$ für die echten Kosten c_i, wird der Überschuss $a_i - c_i \geq 0$ auf das Bankkonto eingezahlt. Wenn $a_i < c_i$ ist, bestreiten wir den fehlenden Betrag $c_i - a_i$ aus dem Guthaben. Dabei ist Schuldenmachen streng verboten: Es muss auf dem Konto immer ein Guthaben ≥ 0 vorhanden sein! Je nach Anwendung kann man zur Veranschaulichung das Guthaben auf Komponenten der Datenstruktur verteilen. Wir wenden die Methode auf Beispiel 1 an, den Stack mit „*multipop*“.

Operation	a_i	c_i	Bemerkung
isempty	1	1	keine Änderung des Guthabens
push	2	1	Einzahlung: 1
pop	0	1	Abhebung: 1
multipop(k)	0	$\min\{k, p\}$	Abhebung: $\min\{k, p\}$

Bei jeder Einfügung steigt das Guthaben um $a_i - c_i = 2 - 1 = 1$. Wir können uns daher vorstellen, dass jeder Eintrag ein Guthaben von 1 besitzt. Am Anfang ist das Guthaben 0, und der Stack ist leer. Der eingezahlte Wert 1 bei „*push*(x)“ gehört zu

diesem Eintrag x. Bei „*pop*“ wird ein Element y entfernt, die Kosten 1 werden durch das y zugeordnete Guthaben abgedeckt. Bei „*multipop*(k)“ werden $\min\{k, p\}$ Einträge entfernt, das Guthaben dieser Einträge deckt die Kosten genau ab.

Wir definieren das Guthaben nach Schritt i als B_i und die Stackhöhe nach Schritt i als p_i.

Lemma 3.2.1. *$B_i = p_i$, für $i \geq 0$. (Daraus folgt: $B_i \geq 0$ für alle $i \geq 0$.)*

Die Aussage folgt eigentlich direkt daraus, wie das Guthaben auf die aktuell vorhandenen Stackeinträge verteilt ist. Im Allgemeinen beweist man eine solche die Behauptung durch Induktion über i.

Aus $B_n = p_n \geq 0$ folgt dann:

$$0 \leq B_n = \sum_{1 \leq i \leq n} (a_i - c_i) \ , \ \text{also} \quad C_n = \sum_{1 \leq i \leq n} c_i \leq \sum_{1 \leq i \leq n} a_i.$$

Das bedeutet, dass die (echten!) Gesamtkosten C_n nicht größer als $2 \cdot$ (Anzahl der *push*-Operationen) sein können.

Wir formulieren das **Rezept „Bankkontomethode“** allgemein:

(i) Ordne jeder Operation Op_i (geschickt) amortisierte Kosten a_i zu.
(Meist hängen diese nicht von i ab, sondern nur von Op_i und eventuell von der Größe der Datenstruktur.)

(ii) Definiere $B_0 := 0$ und $B_i := B_{i-1} + (a_i - c_i)$, für $i \geq 1$.
(Die Veränderung $a_i - c_i$ des Kontostands kann positiv oder negativ oder gleich 0 sein.)

(iii) Formuliere eine Induktionsbehauptung (IB_i) über B_i, aus der folgt, dass stets $B_i \geq 0$ gilt.
(Die Wahl von (IB_i) ist der entscheidende und schwierige Schritt. Die Behauptung „$B_i \geq 0$“ genügt nicht!)

(iv) Beweise (IB_i).
(Meist durch Induktion, mit einer Fallunterscheidung darüber, was bei Op_i passiert.)

(v) Aus $B_n \geq 0$ und $B_0 = 0$ und $B_i = B_{i-1} + (a_i - c_i)$, für $1 \leq i \leq n$, folgt unmittelbar $\sum_{i=1}^{n}(a_i - c_i) \geq 0$, also

$$C_n = c_1 + \cdots + c_n \leq a_1 + \cdots + a_n.$$

Daraus erhält man die gewünschte Schranke für C_n.

***Beispiel* 2:** Hochzählen eines Binärzählers, Bankkontomethode.
(i) Wir definieren:

$$a_i := 2. \qquad \text{(„Clevere“ Idee!)}$$

(ii) $B_0 := 0$ und $B_i := B_{i-1} + (a_i - c_i)$, für $i \geq 1$.

(iii) (IB_i) $B_i = |\text{bin}(i)|_1 =$ Anzahl der 1-Ziffern in $\text{bin}(i)$. (Dies ist der „clevere“ Teil!)
(Beispiel: $\text{bin}(19) = 10011$, also muss $B_{19} = 3$ sein. Man kann sich das Ansparen so vorstellen: Beim Hochzählen wird genau eine neue 1 erzeugt, also eine 0 auf 1 gekippt. Die amortisierten Kosten bezahlen für dieses Kippen *und* für das spätere Zurückkippen dieser Stelle auf 0. Solange die Ziffer 1 Bestand hat, ist ihr das Guthaben von 1 Euro zugeordnet.)

(iv) Beweis von (IB_i) durch Induktion über i:
I. A.: $B_0 = 0$, und die Anzahl der 1-Ziffern in $\text{bin}(0)$ ist 0.
I. V.: (IB_{i-1}) stimmt.
I. S.: Der Zähler wird von $i-1$ auf i erhöht.

$$\text{bin}(i-1) = \underbrace{*\ldots*}_{z \text{ Einsen}}\, 0\, \underbrace{1\ldots1}_{\ell},$$
$$\text{bin}(i) = \underbrace{*\ldots*}_{z \text{ Einsen}}\, 1\, \underbrace{0\ldots0}_{\ell}.$$

Nach I. V. gilt $B_{i-1} = z + \ell$. Die echten Kosten c_i sind die Anzahl der gekippten Bits, also $\ell + 1$. Damit:

$$B_i = B_{i-1} + a_i - c_i = (z + \ell) + 2 - (\ell + 1) = z + 1,$$

und das ist gerade die Anzahl der Einsen in $\text{bin}(i)$, wie behauptet.

(v) Da die Anzahl der Einsen in $\text{bin}(i)$ stets nichtnegativ ist (genauer gesagt: sie ist positiv für $i > 0$), folgt

$$C_n = c_1 + \cdots + c_n \leq a_1 + \cdots + a_n = 2n.$$

Man kann die amortisierten Kosten sogar genau angeben:

$$C_n = c_1 + \cdots + c_n = a_1 + \cdots + a_n - |\text{bin}(n)|_1 = 2n - |\text{bin}(n)|_1.$$

(Beispiele: $C_{16} = 31$, $C_{17} = 32 = 2 \cdot 17 - 2$, $C_{18} = 34 = 2 \cdot 18 - 2$, usw.)

Um die Schlagkraft der Methode zu demonstrieren, betrachten wir noch ein etwas komplizierteres Beispiel:

***Beispiel* 3:** Stack mit Verdoppelungs- und Halbierungsstrategie.
Wir betrachten den Datentyp Stack mit den gewöhnlichen Operationen *empty*, *push*,

pop, *isempty*, *top*. Der Stack soll mit Hilfe eines Arrays $\texttt{A}[1..m]$ und eines Pegels $\texttt{p}$ dargestellt werden. Der Pegelstand p (in $\texttt{p}$) gibt die aktuelle Anzahl der Einträge an; der Stack besteht (von oben nach unten) aus den Einträgen $\texttt{A}[p]$, $\texttt{A}[p-1]$, ..., $\texttt{A}[1]$. Ein Problem entsteht dadurch, dass man beim Anlegen des Arrays nicht weiß, wie hoch der Stack wird. In der AuD-Vorlesung wurde schon die *Verdopplungsstrategie* behandelt und analysiert, die es gestattet, die Datenstruktur gegebenenfalls zu vergrößern, ohne dass die Kosten mehr als linear in der Anzahl der Operationen sind. Wir erweitern die Fragestellung noch. Wie kann man gegebenenfalls auch wieder Speicher freigeben, wenn der Stack wieder viel kleiner wird, so dass der zu einem beliebigen Zeitpunkt beanspruchte Platz nicht viel größer als die Anzahl der aktuellen Einträge ist? Wir geben eine Anfangs- und Mindestgröße m_0 für das Array vor.

Op_i = *empty*: Lege Array $\texttt{A}[1..m_0]$ an, setze $\texttt{p} \leftarrow 0$.

Op_i = *isempty*, *top*: Ausführung offensichtlich; echte Kosten: $c_i = 1$.

$\mathrm{Op}_i = \mathit{push}(x)$:

1. Fall: Wenn $\texttt{p} < m$, erhöhe $\texttt{p}$ um 1 und speichere x in $\texttt{A[p]}$.
Echte Kosten: $c_i = 1$.

2. Fall: Wenn $\texttt{p} = m$: **„Verdopple."** Das heißt: $\texttt{A}[1..m]$ wird durch ein doppelt so großes Array ersetzt, wie folgt. Ein Array $\texttt{AA}[1..2m]$ wird alloziert; die m Einträge werden von $\texttt{A}$ nach $\texttt{AA}$ kopiert; $\texttt{A}$ wird freigegeben; $\texttt{AA}$ wird in $\texttt{A}$ umbenannt. Nun ist Platz für das Einfügen von x (wie im 1. Fall).
Echte Kosten: $c_i = m + 1$ (für Umspeichern und Einfügen von x).

Op_i = *pop*:

1. Fall: Wenn $m = m_0$ oder $\frac{1}{4}m < \texttt{p}$, gib $\texttt{A[p]}$ aus und verringere $\texttt{p}$ um 1. Echte Kosten: $c_i = 1$.

2. Fall: Wenn $\texttt{p} = \frac{1}{4}m$ und $m > m_0$: **„Halbiere."** Das heißt: $\texttt{A}[1..m]$ wird durch ein halb so großes Array ersetzt, wie folgt. Ein Array $\texttt{AA}[1..\frac{1}{2}m]$ wird alloziert; die $m/4$ Einträge werden von $\texttt{A}$ nach $\texttt{AA}$ kopiert; $\texttt{A}$ wird freigegeben; $\texttt{AA}$ wird in $\texttt{A}$ umbenannt. Nun ist $\texttt{A}$ genau zur Hälfte gefüllt, und wir verfahren weiter wie im 1. Fall.
Echte Kosten: $c_i = \frac{1}{4}m + 1$ (für Umspeichern und Entfernen des obersten Eintrags).

Eine kurze Bemerkung zu der naheliegenden Frage, weshalb man nicht halbiert, wenn der Füllstand unter $\frac{1}{2}m$ fällt: In diesem Fall könnten abwechselnde *pop*- und *push*-Operationen zu aufeinanderfolgenden Verdopplungen und Halbierungen führen, die viel zu teuer wären.

Beispiel: Wir überlegen kurz, wie der Auf- und Abbau der Arrays abläuft, wenn man mit $m_0 = 100$ startet, zunächst 2500 Einträge einfügt und dann 2490 wieder löscht. Einfügen der ersten 100 Einträge kostet 100. Dann wird verdoppelt, mit Kosten 101. Einfügen der nächsten 99 Einträge kostet 99, die folgende Verdopplung 201, und so weiter. Durch weitere Verdopplungen wächst das Array auf Größen 400, 800, 1600, 3200 an. Die Kosten: 199 (einzeln), 401 (Verdopplung), 399 (einzeln), 801 (Verdopplung), 799 (einzeln), 1601 (Verdopplung), 899 (einzeln). Insgesamt: $2500 + 100 + 200 + 400 + 800 + 1600 = 2500 + 3100 = 5600$. Nun wird gelöscht. Nach 1700 Einzel-Löschungen ist die Anzahl der Einträge auf 800 gesunken. Das Array wird auf Größe 1600 halbiert, mit Kosten 801. Danach erfolgt die Halbierung auf 800 (Kosten 401), auf 400 (Kosten 201) auf 200 (Kosten 101) und auf $m_0 = 100$ (Kosten 51). Weitere Löschungen verkleinern das Array nicht mehr. Die Gesamtkosten für die Löschungen sind $2490 + 800 + 400 + 200 + 100 = 2490 + 1500 = 3990$.

Bei einem Ablauf mit einer so einfachen Struktur sieht man leicht, dass die Gesamtkosten linear in der Anzahl der Operationen sind. Aber wie sieht es bei Abläufen aus, bei denen Einfügungen und Löschungen wild durcheinander auftreten? Wir benutzen die Bankkontomethode.

(i) Wir definieren amortisierte Kosten wie folgt. Dabei beschränken wir uns auf die Operationen *push*(x) und *pop*. (Die anderen Operationen haben konstante echte Kosten und ändern die Datenstruktur nicht.)

Operation	a_i	c_i	Bemerkung
empty	1	1	wird nicht weiter betrachtet
top	1	1	wird nicht weiter betrachtet
push	3	1. Fall: 1; 2. Fall: $m + 1$	1. Fall: Einz.; 2. Fall: Abhebung
pop	2	1. Fall: 1; 2. Fall: $\frac{1}{4}m + 1$	1. Fall: Einz.; 2. Fall: Abhebung

(ii) $B_0 := 0$ und $B_i := B_{i-1} + (a_i - c_i)$, für $i \geq 1$.

(iii) Es sei m_i die Arraygröße nach Schritt i und p_i der Pegelstand nach Schritt i.

(IB$_i$) (a) Wenn $p_i \geq \frac{1}{2}m_i$, dann gilt $B_i \geq 2(p_i - \frac{1}{2}m_i)$ (≥ 0, s. Abb. 2);
(b) wenn $m_i > m_0 \wedge p_i < \frac{1}{2}m_i$, dann gilt $B_i \geq \frac{1}{2}m_i - p_i$ (≥ 0, s. Abb. 3);
(c) wenn $m = m_0$ und $p_i < \frac{1}{2}m$, gilt $B_i \geq 0$ (s. Abb. 4).

Idee: Wenn $p \geq \frac{1}{2}m$, wird Guthaben angespart, um für eine zukünftige teure Verdopplung zu bezahlen. (Man kann sich vorstellen, dass auf jedem Eintrag, der oberhalb von Level $\frac{1}{2}m$ liegt, ein Guthaben von 2 Euro liegt. Weiteres Guthaben ignorieren wir. Wenn der Pegelstand $\frac{1}{2}m_i$ ist, wird nur verlangt, dass das Guthaben nichtnegativ ist.) Wenn $p < \frac{1}{2}m$, wird ebenfalls Guthaben aufgebaut, um für eine zukünftige teure Halbierung zu bezahlen. (Jedem freien Platz unter Level $\frac{1}{2}m$ ist ein Guthaben von 1 Euro zugeordnet, weiteres Guthaben wird ignoriert.) Dies entfällt, wenn $m = m_0$ ist, also insbesondere in der Anfangsphase, wenn das Array noch fast ganz leer ist.

(iv) Beweis von (IB$_i$) durch Induktion über i:

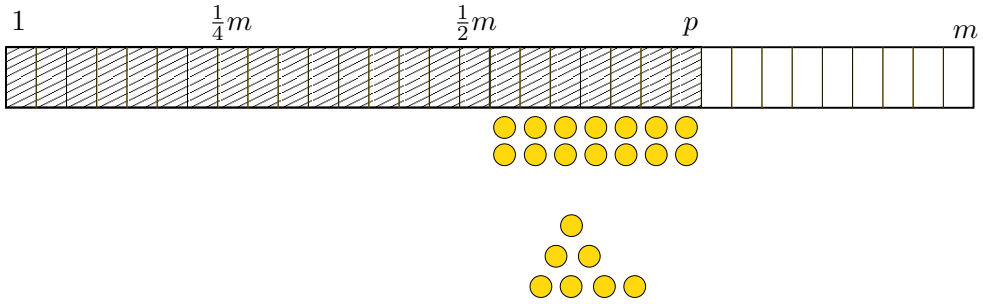


Abbildung 2: (a) Wenn der Pegelstand p größer oder gleich $\frac{1}{2}m$ ist, verlangt man: Kontostand $B \geq 2(p - \frac{1}{2}m)$: 2 Euro für jede besetzte Stelle oberhalb von $\frac{1}{2}m$.

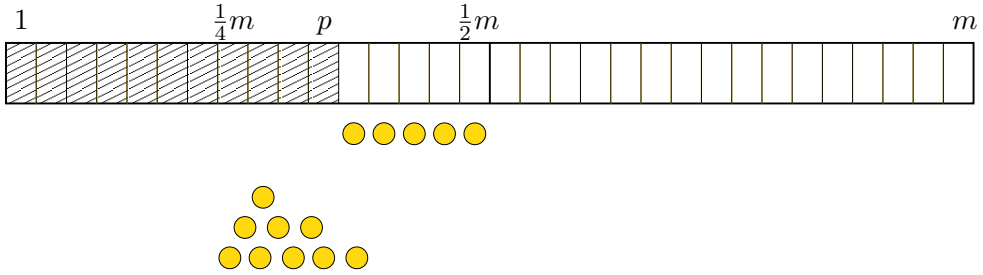


Abbildung 3: (b) Wenn der Pegelstand p zwischen $\frac{1}{4}m$ und $\frac{1}{2}m$ liegt und $m > m_0$ ist, verlangt man: Kontostand $B \geq \frac{1}{2}m - p$: ein Euro für jede Fehlstelle unterhalb von $\frac{1}{2}m$.

I. A.: $B_0 = 0$, und $p_0 = 0$ und m_0 ist der Startwert. (IB_0) ist trivialerweise erfüllt.
I. V.: (IB_{i-1}) stimmt.
I. S.: Op_i wird aufgeführt. Es gibt zwei Fälle:

Falls $\text{Op}_i = \mathit{push}(x)$:
Wenn $p_{i-1} < \frac{1}{2}m_{i-1}$, gilt $c_i = 1$ und $a_i = 3$, also $B_i > B_{i-1}$. Die Anzahl der freien Plätze unter Level $\frac{1}{2}m_{i-1}$ sinkt um 1, also wird die Anforderung an das Guthaben geringer. Auch wenn $p_i = \frac{1}{2}m_{i-1}$, gilt $B_i \geq 1 > 0$. Damit ist (IB_i) in allen möglichen Fällen gesichert.
Wenn $\frac{1}{2}m_{i-1} \leq p_{i-1} < m_{i-1}$, dann wächst das Guthaben um $a_i - c_i = 2$ und $2(p - \frac{1}{2}m)$ wächst ebenfalls um 2. Damit gilt auch Bedingung (b) weiter. (Wir legen die beiden eingezahlten Euros neben den neuen Eintrag x.)
Wenn schließlich $p_{i-1} = m_{i-1}$, dann erfolgt die Verdoppelung auf $m_i = 2m_{i-1}$. Das Guthaben B_{i-1} ist mindestens $2(p_{i-1} - \frac{1}{2}m_{i-1}) = m_{i-1}$. Dies reicht gerade, um die

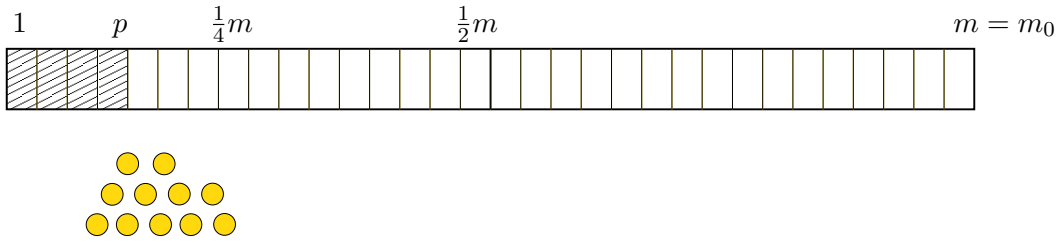


Abbildung 4: (c) Wenn $m = m_0$ und der Pegelstand p kleiner als $\frac{1}{2}m$ ist, verlangt man nur: Kontostand $B \geq 0$.

Kosten m_{i-1} des Umbaus zu decken. Die Einfügung hat noch Kosten 1, die amortisierten Kosten sind $a_i = 3$. Bleiben genau 2 Euros übrig, die wir neben den neuen Eintrag x legen. In Zahlen:

$$\begin{aligned} B_i &= B_{i-1} + (a_i - c_i) \overset{\text{I.V.(a)}}{\geq} 2(p_{i-1} - \frac{1}{2}m_{i-1}) + (3 - (m_{i-1} + 1)) \\ &= 2(m_{i-1} - \frac{1}{2}m_{i-1}) + (3 - (m_{i-1} + 1)) = 2 \\ &= 2(p_i - \frac{1}{2}m_i). \end{aligned}$$

Also gilt (IB_i).

Falls $\text{Op}_i = \mathit{pop}$:
Wenn $\frac{1}{2}m_{i-1} < p_{i-1} \leq m_{i-1}$, dann wächst das Guthaben um $a_i - c_i = 1$, aber die Anforderung in (a) wird schwächer, weil die Anzahl der Einträge oberhalb von Level $\frac{1}{2}m_{i-1}$ sinkt.
Wenn $\frac{1}{4}m_{i-1} < p_{i-1} \leq \frac{1}{2}m_{i-1}$ und $m_{i-1} > m_0$, dann wächst das Guthaben um $a_i - c_i = 1$, und $\frac{1}{2}m - p$ wächst ebenfalls um 1. Anschaulich: Wir legen den neu eingezahlten Euro neben die neu freigewordene Stelle.
Wenn $\frac{1}{4}m_{i-1} < p_{i-1} \leq \frac{1}{2}m_{i-1}$ und $m_{i-1} = m_0$, dann gibt es nur die Anforderung $B_i \geq 0$, die aber aus der I. V. folgt.
Wenn schließlich $m_{i-1} > m_0$ und $p_{i-1} = \frac{1}{4}m_{i-1}$, erfolgt die Halbierung, mit $c_i = \frac{1}{4}m_{i-1} + 1$ und $a_i = 2$. Diese hat Kosten $\frac{1}{4}m_{i-1}$. Diese Kosten werden durch die Eurostücke gedeckt, die neben den leeren Plätzen zwischen Level $\frac{1}{4}m_{i-1}$ und $\frac{1}{2}m_{i-1}$ liegen. Das Entfernen des nächsten Eintrags kostet noch 1, aber mit $a_i = 2$ bleibt

noch ein Euro für die neue leere Stelle unter $\frac{1}{2}m_i$ übrig. Rechnerisch:

$$\begin{aligned} B_i &= B_{i-1} + (a_i - c_i) \overset{\text{I.V.(b)}}{\geq} (\frac{1}{2}m_{i-1} - p_{i-1}) + (2 - (\frac{1}{4}m_{i-1} + 1)) \\ &= 1 = \frac{1}{2}m_i - p_i. \end{aligned}$$

Bei $m_{i-1} = m_0$ fallen keine Halbierungskosten an, das Guthaben bleibt positiv. Also gilt (IB_i).

(v) Da die in (IB_i) angegebenen unteren Schranken für B_i stets nichtnegativ sind, folgt

$$C_n = c_1 + \cdots + c_n \leq a_1 + \cdots + a_n \leq 3n_{push} + 2n_{pop},$$

wobei n_{push} und n_{pop} die Anzahl der *push*- bzw. *pop*-Operationen in der Folge ist.

Satz 3.2.2. *Die Arrayimplementierung von Stacks mit Verdoppelung und Halbierung hat folgende Eigenschaften:*

(a) *n Operationen haben insgesamt Zeitaufwand $O(n)$.*

(b) *Wenn p Einträge in der Datenstruktur gespeichert sind, ist das momentan benutzte Array nicht größer als* $\max\{m_0, 4p\}$.

3.3 Die Potenzialmethode

Auch die Potenzialmethode ist ein allgemeiner Ansatz, den man bei amortisierten Analysen in seinem Werkzeugkoffer haben sollte. Hier ist die Idee, jedem „inneren Zustand“ D der Datenstruktur eine (nichtnegative) reelle Zahl, genannt das Potenzial $\Phi(D)$ von D, zuzuordnen.[2] Wenn man möchte, kann man sich $\Phi(D)$ als eine Art in D gespeicherte Energie vorstellen. (Wie wir gleich sehen werden, handelt es sich hier um „monetäre Energie“: die Fähigkeit, Kosten zu übernehmen ...)

***Beispiel* 1:** Stack mit „*multipop*“. – (i) Wir ordnen einem Stack D mit momentan p Einträgen das Potenzial $\Phi(D) = p$ zu.

Nach Festlegung des Potenzials geht die Potenzialmethode recht schematisch vor.

(ii) Die amortisierten Kosten einer Operation Op, die die Datenstruktur von Zustand D in Zustand D' transformiert, mit echten Kosten c, hat *amortisierte Kosten*

$$a := c + (\Phi(D') - \Phi(D)) = c - (\Phi(D) - \Phi(D')). \tag{1}$$

[2] „Φ“ heißt „Phi“ und wird „Fi“ ausgesprochen.

Wir interpretieren: Wenn $\Phi(D') > \Phi(D)$, ist die Potenzialdifferenz positiv, und für das Erreichen des höheren (Energie-)Niveaus muss man zusätzlich zu c „Potenzialkosten“ $\Phi(D') - \Phi(D)$ aufwenden. Wenn dagegen $\Phi(D') < \Phi(D)$, geht das Potenzial nach unten; die Potenzialdifferenz $\Phi(D) - \Phi(D')$ kann benutzt werden, um die echten Kosten c zum Teil oder ganz zu bestreiten.

(iii) Nun werden für jede Operation Op die amortisierten Kosten a_{Op} durch eine Schranke K_{Op} nach oben abgeschätzt. Manchmal hängt K_{Op} von der Größe der Datenstruktur ab; manchmal ist es eine Konstante.

In unserem Beispiel sieht dies so aus:

1. Fall: Op = *isempty* oder Op = *top*. – Da sich die Datenstruktur nicht verändert, ist die Potenzialdifferenz 0. Also ist $a = c = 1$.

2. Fall: Op = *push*(x). – Die Stackhöhe wächst um 1, d. h. $\Phi(D') - \Phi(D) = 1$. Damit ist $a = c + 1 = 2$.

3. Fall: Op = *multipop*(k). – Die Stackhöhe sinkt um $\min\{k, p\}$, d. h. $\Phi(D') - \Phi(D) = \min\{k, p\}$. Die Kosten der Operation sind $c = \min\{k, p\}$. Daher ist $a = c - (\Phi(D) - \Phi(D')) = 0$.

4. Fall: Op = *pop*. – Wie *multipop*(1).

Wir erhalten also in diesem Fall, dass die amortisierten Kosten für ein *push* 2 betragen, für *pop* und *multipop*(k) dagegen 0.

Nun betrachten wir eine Operationenfolge $\mathrm{Op}_0, \mathrm{Op}_1, \ldots, \mathrm{Op}_n$, die eine Datenstruktur D_0 erzeugt und in n Runden zu $D_1, \ldots, D_n$ verändert. Die echten Kosten der Operation Op_i, die D_{i-1} in D_i transformiert, seien c_i. Die amortisierten Kosten dieser Operation sind dann

$$a_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

Die folgende Behauptung ist zentral für die Anwendung der Potenzialmethode.

Lemma 3.3.1. *Wenn Φ die Bedingung $\Phi(D_n) \geq \Phi(D_0)$ erfüllt, dann gilt $C_n = c_1 + \cdots + c_n \leq a_1 + \cdots + a_n$.*

(Insbesondere haben wir: Wenn $a_i \leq K$ für eine Konstante K ist, dann ist $C_n \leq Kn$. Die Bedingung $\Phi(D_n) \geq \Phi(D_0)$ wird oft gezeigt, indem man feststellt, dass $\Phi(D_0) = 0$ und $\Phi(D) \geq 0$ für beliebige Zustände D gilt.)

Beweis:

$$\begin{aligned} C_n &= c_1 + \cdots + c_n \\ &= \sum_{1 \le i \le n} (a_i + (\Phi(D_{i-1}) - \Phi(D_i))) \\ &= \sum_{1 \le i \le n} a_i + \sum_{1 \le i \le n} (\Phi(D_{i-1}) - \Phi(D_i)) \\ &\overset{(*)}{=} \sum_{1 \le i \le n} a_i + \Phi(D_0) - \Phi(D_n) \\ &\le \sum_{1 \le i \le n} a_i. \end{aligned}$$

Die Gleichheit $(*)$ gilt, weil sich positive und negative Terme ausgleichen („Ziehharmonikasumme“ oder „Teleskopsumme“).

(iv) Es gilt $\Phi(D_0) = 0$, weil D_0 der leere Stack ist. Weiter gilt $\Phi(D_n) \ge 0$, weil die Stackhöhe p nie negativ sein kann. Damit sind die Bedingungen von Lemma 3.3.1 erfüllt. Alle a_i sind ≤ 2, also folgt: $C_n \le 2n$.

Wir formulieren die ***Potenzialmethode*** allgemein:

(i) Ordne jedem Zustand D der Datenstruktur ein Potenzial $\Phi(D) \ge 0$ zu.

(ii) Für eine Operation Op, Zustand D in Zustand D' transformiert und dabei echte Kosten $c = c_{\mathrm{Op}}(D, D')$ hat, definiere „*amortisierte Kosten*“

$$a := a_{\mathrm{Op}}(D, D') := c + \Phi(D') - \Phi(D).$$

(iii) Schätze $a = a_{\mathrm{Op}}(D, D')$ nach oben ab, für jede Operation Op.

(iv) Gegeben sei eine Operationenfolge $\mathrm{Op}_1, \ldots, \mathrm{Op}_n$, mit echten Kosten $c_1, \ldots, c_n$ und amortisierten Kosten $a_1, \ldots, a_n$. Zeige, dass $\Phi(D_n) \ge \Phi(D_0)$.
(Sehr oft: $\Phi(D_0) = 0$ und $\Phi(D) \ge 0$ für alle D.)

(v) Folgere durch Anwendung von Lemma 3.3.1: $C_n = c_1 + \cdots + c_n \le a_1 + \cdots + a_n$.

***Beispiel* 2:** Hochzählen eines Binärzählers.

Wir wenden die ***Potenzialmethode*** an.

(i) Der Zustand der Datenstruktur ist hier die Binärdarstellung $\mathrm{bin}(i)$ des Zählerstandes i. Wir definieren:

$$\Phi(\mathrm{bin}(i)) := \text{ Anzahl der Einsen in } \mathrm{bin}(i).$$

(Beachte: Die Potenzialfunktion geeignet zu wählen ist der entscheidende Punkt.)

(ii) Die Operationenfolge $\mathrm{Op}_1, \ldots, \mathrm{Op}_n$ ist fest gegeben: Op_i zählt von $i-1$ auf i hoch. Kosten c_i: Anzahl der gekippten Bits. Definiere amortisierte Kosten

$$a_i := c_i + \Phi(\mathrm{bin}(i)) - \Phi(\mathrm{bin}(i-1)), \text{ für } 1 \leq i \leq n.$$

(iii) Wir berechnen a_i. Man vergleiche das Bild in Abschnitt 3.2, um folgende Rechnung zu rechtfertigen. Wenn $i-1$ auf ℓ Einsen endet und insgesamt $z+\ell$ Einsen enthält, dann ist

$$a_i = c_i + \Phi(\mathrm{bin}(i)) - \Phi(\mathrm{bin}(i-1)) = (\ell+1) + (z+1) - (z+\ell) = 2.$$

(iv) $\Phi(\mathrm{bin}(n)) \geq \Phi(\mathrm{bin}(0))$ gilt, weil $\Phi(\mathrm{bin}(0)) = 0$ ist und $\Phi(\mathrm{bin}(i)) \geq 0$ für alle i, einfach nach der Definition.

(v) Mit Lemma 3.3.1 folgt: $C_n = c_1 + \cdots + c_n \leq a_1 + \cdots + a_n = 2n$.

Bemerkung: Wenn man die Rechnungen in (i)–(iii) und im Beweis von Lemma 3.3.1 genau nachvollzieht, sieht man, dass an allen Stellen außer in (v) Gleichheit gilt. Wir können also genauer schreiben:

$$C_n = \sum_{1 \leq i \leq n} a_i + \Phi(\mathrm{bin}(0)) - \Phi(\mathrm{bin}(n)) = 2n - \#(\text{Einsen in } \mathrm{bin}(n)).$$

Diese Formel gibt also die exakte Anzahl von gekippten Bits beim Zählen bis n an. (*Beispiel*: Nach der Tabelle auf Seite 3 ist $C_5 = 1+2+1+3+1 = 8$; andererseits ist $2 \cdot 5 - \#(\text{Einsen in } \mathrm{bin}(5)) = 10 - 2 = 8$. Magie! Wenn man diesen Trick verstanden hat, lässt sich auch die Anzahl der Bits leicht angeben, die beim Hochzählen von k nach n gekippt werden: $2(n-k) + \#(\text{Einsen in } \mathrm{bin}(k)) - \#(\text{Einsen in } \mathrm{bin}(n))$. Man beachte, dass dies u. U. auch etwas größer als $2(n-k)$ sein kann.

***Beispiel* 3:** Stack mit Verdoppelungs- und Halbierungsstrategie.
Wir betrachten die gleiche Situation wie oben, aber analysieren sie mit der Potenzialmethode. Anfangs ist der Stack leer ($p_0 = 0$), das Array hat Größe m_0.

(i) Der Zustand der Datenstruktur D ist hier durch die Arraygröße m und den Pegelstand p gegeben (kurz: der Zustand ist $D = (m, p)$). Wir definieren das Potenzial:

$$\Phi(D) := \begin{cases} 0 & \text{, falls } m = m_0 \text{ und } p < \frac{1}{2}m; \\ \frac{1}{2}m - p & \text{, falls } m > m_0 \text{ und } \frac{1}{4}m \leq p < \frac{1}{2}m; \\ 2(p - \frac{1}{2}m) & \text{, falls } \frac{1}{2}m \leq p \leq m. \end{cases}$$

Wieder gilt: Dass man diese Werte geschickt wählt, ist für das Gelingen der Analyse entscheidend.

(ii) Für den Übergang von D zu D' mit echten Kosten c definieren wir amortisierte Kosten

$$a := a_{\mathrm{Op}}(D, D') = c + \Phi(D') - \Phi(D).$$

(iii) Wir berechnen die amortisierten Kosten a, wenn Op von Zustand (m, p) zu Zustand (m', p') führt und dabei echte Kosten c anfallen. Dafür gibt es eine Reihe von Fällen zu betrachten. Wir bemerken zunächst, dass für $m = m_0$ und $p, p' \leq \frac{1}{2}m$ das Potenzial 0 ist und weder Verdopplung noch Halbierung vorkommen, also $a = c = 1$ gilt. Ab hier nehmen wir an, dass dieser Sonderfall nicht vorliegt.

1. Fall: Op = *push*.

Fall 1a: $\frac{1}{4}m \leq p < \frac{1}{2}m$ (und $m > m_0$). –
Dann gilt $m' = m$ und $p' = p + 1$ und $c = 1$, also

$$a = c + \Phi((m', p')) - \Phi((m, p)) = 1 + (\tfrac{1}{2}m' - p') - (\tfrac{1}{2}m - p) = 0.$$

Fall 1b: $\frac{1}{2}m \leq p < m$. –
Dann gilt $m' = m$ und $p' = p + 1$ und $c = 1$, also

$$a = c + \Phi((m', p')) - \Phi((m, p)) = 1 + 2(p' - \tfrac{1}{2}m') - 2(p - \tfrac{1}{2}m) = 3.$$

Fall 1c: $p = m$. –
Dann gilt $m' = 2m$ und $p' = p + 1$ und $c = m + 1$, also

$$\begin{aligned} a &= c + \Phi((m', p')) - \Phi((m, p)) \\ &= (m + 1) + 2(p' - \tfrac{1}{2}m') - 2(p - \tfrac{1}{2}m) \\ &= (m + 1) + 2((m + 1) - m) - 2(m - \tfrac{1}{2}m) = 3. \end{aligned}$$

Im Fall 1c ist besonders schön zu sehen, wie der Potenzialunterschied benutzt wird, um für eine teure Operation zu bezahlen. Wenn Op_i eine *push*-Operation ist, sind die amortisierten Kosten durch 3 beschränkt.

2. Fall: Op = *pop*.

Fall 2a: $\frac{1}{2}m < p \leq m$. –
Dann gilt $m' = m$ und $p' = p - 1$ und $c = 1$, also

$$a = c + \Phi((m', p')) - \Phi((m, p)) = 1 + 2(p' - \tfrac{1}{2}m') - 2(p - \tfrac{1}{2}m) = -1.$$

Fall 2b: $\frac{1}{4}m < p \leq \frac{1}{2}m$ (und $m > m_0$). –
Dann gilt $m' = m$ und $p' = p - 1$ und $c = 1$, also

$$a = c + \Phi((m', p')) - \Phi((m, p)) = 1 + (\tfrac{1}{2}m' - p') - (\tfrac{1}{2}m - p) = 2.$$

Fall 2c: $p = \frac{1}{4}m$ (und $m > m_0$). –
Dann gilt $m' = \frac{1}{2}m$ und $p' = p - 1$ und $c = \frac{1}{4}m + 1$, also

$$\begin{aligned} a &= c + \Phi((m', p')) - \Phi((m, p)) = (\tfrac{1}{4}m + 1) + (\tfrac{1}{2}m' - p') - (\tfrac{1}{2}m - p) \\ &= (\tfrac{1}{4}m + 1) + 1 - (\tfrac{1}{2}m - \tfrac{1}{4}m) = 2. \end{aligned}$$

Im Fall 2c ist wieder zu sehen, wie der (vorher aufgebaute) Potenzialunterschied benutzt wird, um für eine teure Operation zu bezahlen. Wenn Op_i eine *pop*-Operation ist, sind die amortisierten Kosten durch 2 beschränkt. (Wenn sich wie in Fall 2a bei einer Situation negative amortisierte Kosten ergeben, ist das harmlos.)

(iv) $\Phi(D_n) \geq \Phi(D_0)$ gilt, weil $\Phi((m_0, 0)) = 0$ und $\Phi((m, p)) \geq 0$ für alle Zustände (m, p) ist, nach der Definition.

(v) Mit Lemma 3.3.1 folgt:

$$C_n = c_1 + \cdots + c_n \leq a_1 + \cdots + a_n \leq 3n_{push} + 2n_{pop} \leq 3n,$$

für n_{push} und n_{pop} die Anzahl der *push*- bzw. *pop*-Operationen.

Bemerkung: Der/Die aufmerksame Leser/in hat bemerkt, dass sich die Rechnungen für die Zähler- und die Stack-Beispiele bei der Bankkontomethode und der Potenzialmethode ähneln. Handelt es sich wirklich um verschiedene Methoden? Tatsächlich kann man beweisen, dass die Potenzialmethode in folgendem Sinn mindestens so stark wie die Bankkontomethode ist: Wenn man ein Analyseverfahren mit der Bankkontomethode konstruiert hat (durch Definition künstlicher amortisierter Kosten mit der Eigenschaft, dass der Kontostand nie negativ werden kann), dann kann man auch Potenziale für die möglichen Zustände D der Datenstruktur definieren, mit denen man bei Anwendung der Potenzialmethode zum gleichen Ergebnis wie bei der Bankkontomethode kommt. (Man definiert dazu $\Phi(D)$ als den *minimalen* Kontostand B, den man erreichen kann, wenn man mit dem Startzustand D_0 beginnt und eine Folge $\mathrm{Op}_1, \ldots, \mathrm{Op}_n$ ausführt, die zum Zustand D führt.) Also kann man im Prinzip mit der Potenzialmethode allein auskommen. In vielen Situationen ist aber die Bankkontomethode sehr bequem und bildet die Vorstellung vom „Ansparen“ für später auftauchende teure Operationen anschaulicher ab als die Potenzialmethode.

Ein weiteres Beispiel für die Anwendung der Potenzialmethode folgt im nächsten Kapitel.

(M. Dietzfelbinger, 17. Dezember 2021)

4 Implementierung von Priority Queues: Fibonacci-Heaps

Die vom Datentyp „Priority Queue (PQ)“ (deutsch auch: „Vorrangwarteschlange“) zur Verfügung gestellte Funktionalität ist folgende:

Es werden Objekte („Einträge“) e mit einem Attribut $key(e) \in U$ verwaltet („Schlüssel“ oder „Priorität“, mit $x, y, z, \ldots$ bezeichnet), wobei $(U, <)$ eine Totalordnung ist, z. B. $U = \mathbb{N}$. Dabei entsprechen *kleinere* Schlüssel einer *höheren* Priorität. Typische Anwendungen finden sich in Algorithmen wie dem Algorithmus von Dijkstra, bei Discrete-Event-Simulation, bei der Prozessverwaltung in Rechensystemen (s. AuD-Vorlesung).

Eine einfache Priority Queue bietet mindestens die folgenden Operationen an:

Name(Parameter)	Effekt
new()	erzeuge neue leere PQ
makeHeap($\{e_1, \ldots, e_n\}$)	erzeuge neue PQ mit Einträgen $e_1, \ldots, e_n$
insert(e)	füge e als neuen Eintrag hinzu
min()	gib einen Eintrag mit minimalem Schlüssel aus
deleteMin()	gib einen Eintrag mit minimalem Schlüssel aus **und** entferne ihn aus der Datenstruktur[1]

Einige Algorithmen und Anwendungen (insbesondere der Algorithmus von Dijkstra, und mit ihm alle Arten von Discrete-Event-Simulation) benötigen weitere Operationen, bei denen Schlüssel von Einträgen verändert werden, die nicht das Minimum sind. Hierzu benötigt man den Zugriff auf durch Zeiger identifizierte Einträge. Die Verwendung von Zeigern auf Objekte in der Datenstruktur durch den Benutzer ist extrem schlechter Programmierstil, da es sämtlichen Kapselungsprinzipien widerspricht. Dennoch verwenden wir in der Diskussion den Begriff des Zeigers (Java-Bezeichnung: Referenz). In realen Implementierungen sind auch diese Zeiger lokal für die Datenstruktur („`private`“); der Benutzer benutzt *Namen*, um die gewünschten Einträge zu bezeichnen. Details zu solchen Kapselungstechniken wurden schon in der Vorlesung AuD betrachtet.

[1] Anstelle von *deleteMin*() findet man auch die Bezeichnung *extractMin*().

Zusätzliche „fortgeschrittene“ Operationen („adressierbare PQs“):

Name(Parameter)	Effekt
insert(e)	füge e ein; Rückgabewert: ein Zeiger p auf den Eintrag
makeHeap$((e_1, \ldots, e_n))$	erzeuge neue PQ mit Einträgen $e_1, \ldots, e_n$; Rückgabewerte: n Zeiger $p_1, \ldots, p_n$ auf diese Einträge
decreaseKey(p, x)	verringere den Schlüssel im Eintrag, auf den p zeigt, auf x (wenn $x >$ aktueller Schlüssel: Fehler)
delete(p)	lösche den Eintrag, auf den p zeigt

Eine weitere interessante Operation ist

Name(Parameter)	Effekt
meld(H')	vereinige die PQ mit einer anderen, H', zu *einer* PQ

Adressierbare PQs, die auch diese Operation unterstützen, heißen „meldable PQs“ oder „PQs mit Verschmelzung“.

In der Vorlesung „Algorithmen und Datenstrukturen (AuD)“ (Bachelor) hat sich gezeigt, dass binäre Heaps alle Operationen von adressierbaren PSs (ohne *meld*) recht effizient implementieren. Die benötigten Zeiten bei Binärheaps im schlechtesten Fall sind, wenn die Anzahl der Einträge n ist:

Name(Parameter)	Zeitschranke
new()	$O(1)$
min()	$O(1)$
deleteMin()	$O(\log n)$
insert(e)	$O(\log n)$
makeHeap$((e_1, \ldots, e_n))$	$O(n)$
delete(p)	$O(\log n)$
decreaseKey(p, x)	$O(\log n)$
meld(H')	$O(n)$

Dabei wird „*delete*(p)“ dadurch realisiert, dass zuerst *decreaseKey*$(p, -\infty)$ für einen künstlichen Schlüssel $-\infty$ ausgeführt wird, der kleiner als alle realen Schlüssel ist, und dann *deleteMin*. Die *meld*(H')-Operation fällt aus dem Rahmen: sie erfordert bei Binärheaps einen kompletten Neuaufbau.

In diesem Kapitel besprechen wir eine Datenstruktur, die den abstrakten Datentyp „meldable PQ“ implementiert: Fibonacci-Heaps (Abschnitt 4.1), die amortisiert gesehen insbesondere *decreaseKey* besonders schnell erledigt und dadurch zu guten Laufzeiten des Algorithmus von Dijkstra beiträgt.

4.1 Fibonacci-Heaps

Fibonacci-Heaps implementieren *adressierbare Priority Queues*. Es werden alle dort angegebenen Operationen implementiert. Ein Fibonacci-Heap besteht aus *heapgeordneten Bäumen*. Die Datenstruktur ist so entworfen, dass eine amortisierte Analyse gut durchführbar ist.

Fibonacci-Heaps wurden 1984 von Michael L. Fredman und Robert Endre Tarjan vorgestellt.

4.2 Aufbau von Fibonacci-Heaps

Fibonacci-Heaps (oder F-Heaps) bestehen aus einer Menge von Bäumen. Elementarbausteine sind Baumknoten. Ein solcher Knoten v hat folgende Felder (siehe auch Abb. 1):

Zeiger zum Vorgänger: p

Zeiger auf linkes Geschwister: *prev*

Zeiger auf rechtes Geschwister: *next*

Kindzeiger: *child*

Schlüssel: *key* (aus totalgeordneter Menge U)

Informationsteil: *info* (anwendungsabhängig)

Rang: *rank*: integer (Anzahl der Kinder)

Markierung: *marked*: boolean (1: „markiert“, 0: “unmarkiert“.)

Schema:

Organisation in **einem Baum**: Für die Wurzel r ist der Vorgängerzeiger $p(r)$ gleich NIL, für Nichtwurzeln v zeigt $p(v)$ zum Vorgängerknoten. Der Zeiger $child(v)$ zeigt zu einem (beliebigen) Kindknoten von v (falls es einen gibt); die Kinder eines Knotens sind (mittels der *next*- und der *prev*-Zeiger) als zirkuläre doppelt verkettete Liste organisiert. Die Zahl $rank(v)$ gibt die Anzahl der Kinder von v an, die wir auch den *Rang* von v nennen. In Blättern ist der Kindzeiger NIL, der Rang 0.

In jedem Knoten v, der keine Wurzel ist, gilt die **Heapbedingung** $key(v) \geq key(p(v))$.

Ein ***Fibonacci-Heap*** ist eine Kollektion von solchen heapgeordneten Bäumen, wobei die Wurzeln mit Hilfe ihrer *next*- und *prev*-Zeiger als zirkuläre doppelt verkettete Liste

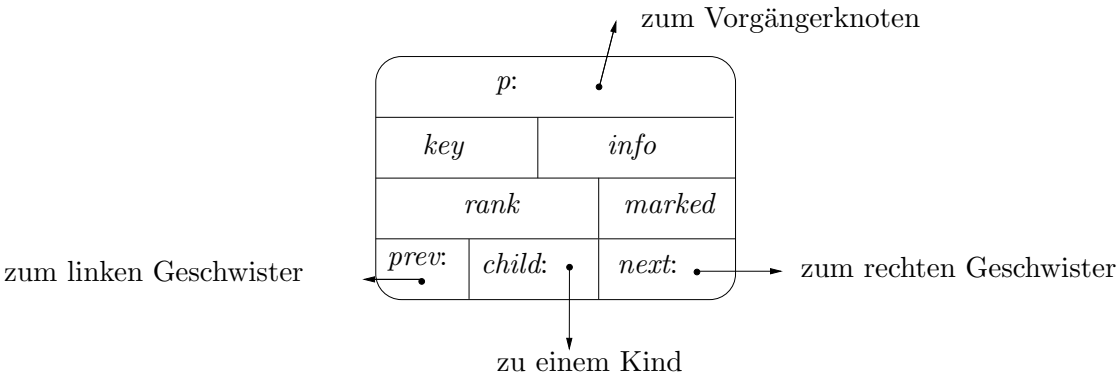


Abbildung 1: Fibonacci-Heaps: Das Format eines Knotens.

organisiert sind. Auf die gesamte Liste wird durch einen Ankerzeiger namens MIN zugegriffen, der auf eine Wurzel mit minimalem Wurzeleintrag zeigt. MIN hat Wert NIL genau dann wenn der Fibonacci-Heap leer ist.

Manche der Knoten sind „markiert“, manche „unmarkiert“. Dies wird explizit im Feld *marked*(v) angegeben. Später werden wir sehen, dass eine solche Markierung bedeutet, dass Knoten v irgendwann einen Kindknoten verloren hat. Markierte Knoten, bei denen ein weiterer Kindknoten entfernt werden soll, werden besonders behandelt.

Wir legen fest: *Wurzelknoten* sind immer *unmarkiert*.

4.3 Potenzialfunktion

Für die amortisierte Analyse der Operationen definieren wir das Potenzial $\Phi(H)$ eines Fibonacci-Heaps H, wie folgt:[1]

$$\Phi(H) := \#(\text{Wurzeln in } H) + 2 \cdot \#(\text{markierte Knoten in } H). \qquad (1)$$

Wenn eine Operation Op auszuführen ist, ist der F-Heap vorher H, der F-Heap nachher H'.

[1] „#“ bedeutet „Anzahl“.

4.4 Einfache Operationen

4.4.1 Erzeugen eines leeren Fibonacci-Heaps

new(): MIN $\leftarrow$ NIL.

Zeitaufwand: $O(1)$.
Echte Kosten: $c_{new} = 1$.
Amortisierte Kosten: $a_{new} = 1$.

Beachte: Das Potenzial des leeren Fibonacci-Heaps H_0 ist $\Phi(H_0) = 0$.

4.4.2 Minimum auslesen

min(): Gib den Eintrag zurück, auf den MIN zeigt, ohne H zu ändern.

Zeitaufwand: $O(1)$.
Echte Kosten: $c_{min} = 1$.
Amortisierte Kosten: $a_{min} = 1$.

4.4.3 Einfügen

insert(e):

Erzeuge neue Wurzel v mit Eintrag e;
falls MIN = NIL, wird v einziger Knoten in der Wurzelliste, sonst:
hänge v direkt neben Knoten MIN in Wurzelliste;
setze MIN auf v um, falls $key(v) < key(\text{MIN})$.

Zeitaufwand: $O(1)$.
Echte Kosten: $c_{insert} = 1$.
Amortisierte Kosten: $a_{insert}(H, H') = c_{insert}(H, H') + \Phi(H') - \Phi(H) = 1 + 1 = 2$
(Die neue Wurzel erhöht das Potenzial um 1.)

4.4.4 Heaperzeugung

makeHeap$((e_1, \ldots, e_n))$:
new();
for i **from** 1 **to** n **do** *insert*(e_i).

Zeitaufwand: $O(n)$.
Echte Kosten: $c_{makeHeap} = n + 1$.
Amortisierte Kosten: Sei H_0 der leere Heap, H der resultierende Heap. Dann:
$a_{makeHeap}(H_0, H) = c_{makeHeap}(H_0, H) + \Phi(H) - \Phi(H_0) = n + 1 + n = 2n + 1$.
($\Phi(H) = n$ wegen der n Wurzeln.)

4.4.5 Vereinigen zweier Heaps

meld(H'): Vereinigt zwei F-Heaps H und H'.

Die beiden Heaps sind durch Zeiger MIN und MIN$'$ gegeben.
Hänge die Wurzelliste von H' neben dem Objekt, auf das MIN zeigt, in die Wurzelliste von H ein;
wenn *key*(MIN$'$) $<$ *key*(MIN), setze MIN auf MIN$'$.

Zeitaufwand: $O(1)$.
Echte Kosten: $c_{meld} = 1$.
Amortisierte Kosten: $a_{meld} = c_{meld} = 1$, weil sich das Potenzial nicht ändert.

4.5 Hilfsoperationen *join* und *cleanup*

Wir beschreiben hier zwei für die Datenstruktur „private“ Operationen, die in der *deleteMin*-Operation benötigt werden. Sie können nicht vom Benutzer aufgerufen werden.

Die Hilfsoperation *join*(v, w) kann auf zwei Wurzeln v, w mit gleichem Rang *rank*$(v) =$ *rank*(w) angewendet werden. Sie vereinigt die Bäume mit diesen Wurzeln zu einem Baum mit Wurzelrang *rank*$(v)+1$. Die Operation ist sehr natürlich: Wenn der Schlüssel in w kleiner ist als der in v, wird v zusätzliches Kind von w, sonst wird w zusätzliches Kind von v.

Prozedur 4.5.1 (*join*(v, w))**.**
Input: Zwei Wurzelknoten v, w von gleichem Rang.

```
(1)  if key(v) ≤ key(w)
(2)  then
(3)       mache w zu neuem Kind von v
(4)          // w wird aus der Wurzelliste ausgeklinkt
             // und in der Kindliste von v direkt neben child(v) eingefügt
(5)       erhöhe rank(v) um 1
(6)  else
(7)       mache v zu neuem Kind von w
(8)       erhöhe rank(w) um 1.
```

Der Zeitaufwand für *join*(v, w) ist $O(1)$.

Mit Hilfe von *join* realisieren wir *cleanup*, eine weitere Hilfsoperation. Zweck dieser Prozedur ist es, die (eventuell sehr lange) Wurzelliste eines F-Heaps zu „kompaktieren“, und zwar so, dass alle verbleibenden Wurzeln verschiedene Ränge haben.

Die Idee ist sehr einfach: Solange man zwei Bäume findet, deren Wurzeln den gleichen Rang haben, wendet man auf diese die *join*-Operation an. Mit jedem *join* verringert sich die Anzahl der Wurzeln um 1, also muss der Prozess irgendwann anhalten. Dann haben alle Wurzeln verschiedene Ränge. Wir müssen dann nur noch die Wurzel mit dem kleinsten Eintrag suchen und den MIN-Zeiger auf diese Wurzel richten.

Die Zahl $D(n) \in \mathbb{N}$ bezeichnet eine (später zu berechnende) obere Schranke für den Rang von Knoten, die in einem F-Heap mit n Einträgen auftreten können. Zur vorläufigen Orientierung: Wir werden sehen, dass $D(n) = O(\log n)$ gewählt werden kann. Das hat den Effekt, dass das Suchen des Minimums am Schluss nur noch Zeit $O(D(n)) = O(\log n)$ dauert.

Das einzige kleine Problem ist, den Ablauf so zu organisieren, dass man nicht lange suchen muss, bis zwei Wurzeln mit dem gleichen Rang gefunden worden sind. Dazu benutzt man einen netten Trick: Man stellt ein Array $A[0..D(n)]$ von Zeigern bereit. Eintrag $A[i]$ kann NIL oder ein Zeiger auf eine Wurzel mit Rang i sein. Wenn man jetzt irgendeine Wurzel r mit Rang j hat, prüft man (in Zeit $O(1)$), ob $A[j] = \text{NIL}$ ist oder auf eine Wurzel zeigt. Im ersten Fall lässt man $A[j]$ auf r zeigen, und behandelt als nächstes irgendeine bislang noch nicht betrachtete Wurzel. Im zweiten Fall wendet man *join* auf r und $A[j]$ an (dadurch wird $A[j]$ wieder NIL), und man hat eine Wurzel mit Rang $j+1$, mit der man genauso weiter verfährt. Wenn man diese Idee konsequent durchführt, erhält man folgendes *cleanup*-Verfahren.

Prozedur 4.5.2 (*cleanup*(L))**.**
Input: Liste L von Wurzeln. // Ein Zeiger MIN wird nicht benötigt
Ausgabe: F-Heap H mit den gleichen Einträgen, Ränge aller Wurzeln verschieden.
Methode:
(1) $A[0..D(n)]$: Array von Zeigern, die anfangs alle NIL sind.
(2) **while** L ist nicht leer **do**
(3) entnehme nächste Wurzel r aus L; $i \leftarrow \mathit{rank}(r)$; $w \leftarrow r$
(4) **while** $A[i] \neq$ NIL **do**
(5) $w \leftarrow \mathit{join}(w, A[i])$; $A[i] \leftarrow$ NIL;
(6) $i \leftarrow i + 1$; (neuer Rang von w)
// der Baum unter w enthält alle Knoten der in (3)–(6) bearbeiteten Bäume
(7) $A[i] \leftarrow w$;
(8) Füge Wurzeln in $A[0..D(n)]$ in neue Wurzelliste ein;
(9) durchlaufe Wurzelliste, um Wurzel r mit minimalem Schlüssel zu finden;
(10) lasse MIN auf r zeigen;
(11) Ausgabe: MIN.

Amortisierte Kostenanalyse: Sei ℓ die anfängliche Länge der Wurzelliste L, $\ell' \leq D(n) + 1$ die Anzahl der Wurzeln in H am Ende. Wir definieren das Potenzial $\Phi(L)$ einer Wurzelliste L analog zu dem Potenzial eines Fibonacci-Heaps wie in (1).

Der Zeitaufwand ist $O(D(n) + \ell + 1)$, denn: Es wird Zeit $O(D(n) + 1)$ für die Initialisierung des Arrays $A[0..D(n)]$, das Durchmustern von $A[0..D(n)]$ in Zeile (8), und für das Finden des Minimums in Zeile (9) benötigt. Es werden ℓ Wurzeln betrachtet und $\ell - \ell'$ *join*-Operationen durchgeführt, die jeweils Zeit $O(1)$ kosten.

Als tatsächliche Kosten setzen wir daher $c_{\mathit{cleanup}}(L, H) = D(n) + \ell + 1$ an.

Amortisierte Kosten: Da $\ell - \ell'$ Wurzeln verschwinden und sich an den Markierungen nichts ändert, ist die Potenzialdifferenz $\Phi(H) - \Phi(L) = \ell' - \ell$. Dabei ist $\ell' \leq D(n) + 1$, und wir erhalten:

$$\begin{aligned} a_{\mathit{cleanup}}(L, H) &= c_{\mathit{cleanup}}(L, H) + \Phi(H) - \Phi(L) \\ &= (D(n) + \ell + 1) + \ell' - \ell \leq 2(D(n) + 1). \end{aligned}$$

(Intuition: Die negative Potenzialdifferenz, die sich durch die verschwindenden Wurzeln ergibt, genügt, um für die *join*-Operationen zu bezahlen.)

Lemma 4.5.3. *Die Operation cleanup auf einer Wurzelliste hat amortisierte Kosten höchstens* $2(D(n) + 1)$*, wobei* n *die Anzahl der Einträge (Knoten) bezeichnet.* □

4.6 Komplexere Operationen

Dieser Abschnitt befasst sich mit der Implementierung und der amortisierten Analyse der Operationen *deleteMin* und *decreaseKey*.

4.6.1 Minimum entnehmen

Prozedur 4.6.1 (*deleteMin*()).
Input: F-Heap H.
Ausgabe: F-Heap H', enthält alle Knoten außer dem Knoten v mit minimalem Eintrag; v.
Methode:
(1) Sei v die Wurzel, auf die MIN zeigt.
(2) Klinke v aus der Wurzelliste aus; Restliste: L; // kein Minimum bekannt!
(3) Durchlaufe Kindliste L' von v, dabei:
(4) für jeden Knoten w in L':
(5) $p(w) \leftarrow$ NIL; *marked*$(w) \leftarrow 0$; hänge w in L ein;
(6) *cleanup*(L); // liefert F-Heap H'
(7) **return**(H', v).

Der Zeitaufwand zur Erstellung von L einschließlich L' (also bis Zeile (5)), ist $O(rank(v)+1) = O(D(n)+1)$.

Für Zeilen (1)–(5) setzen wir echte Kosten $c_{1-5} = rank(v) + 1 \leq D(n) + 1$ an.

Amortisierte Kosten für Zeilen (1)–(5):
$a_{1-5} = c_{1-5} + \Phi(L) - \Phi(H) \leq 1 + 2 \cdot rank(v) \leq 1 + 2D(n)$.
(Die $rank(v)$ neuen Wurzeln erhöhen das Potenzial um je 1. Dies ist nur eine Abschätzung nach oben, da das Potenzial auch geringer sein kann, wenn Kinder von v markiert waren und nun die Markierung verlieren.)

Amortisierte Kosten inklusive *cleanup*(L):

$a_{deleteMin}(H, H') = a_{1-5} + a_{cleanup}(L, H') \leq 3 + 4D(n) = O(D(n))$.

4.6.2 Verringern eines Schlüssels

Der Zweck der Operation *decreaseKey*(v, x) ist, in einem Knoten v den dort vorhandenen Schlüssel y durch einen neuen Schlüssel $x \leq y$ zu ersetzen. Der Knoten v ist hierfür über einen Zeiger (bzw. über eine Referenz) gegeben.

Problem: Wenn $x < key(p(v))$ ist, kann man den Schlüssel nicht einfach verringern, weil dann die Heapbedingung verletzt ist.

Idee: Hänge dann v von seinem Vorgänger $p(v)$ ab und mache v zu einer neuen

Wurzel. Dann kann man $key(v)$ beliebig verringern, ohne dass die Heapbedingung verletzt wird. Man mache sich klar, dass das Abschneiden von v nur konstante Zeit erfordert (man benutzt den Vorgängerzeiger und die Tatsache, dass die Kinder von $p(v)$ als zirkuläre doppelt verkettete Liste organisiert sind), ebenso das Einfügen in die Wurzelliste. Als Ergebnis kennt man auch $p(v)$.

Durch dieses Vorgehen können Knoten Kinder verlieren.

Für die Analyse ist es nötig, dass die Ränge nicht zu groß werden; die Rangschranke $D(n)$ soll logarithmisch in n bleiben. Um zu vermeiden, dass ein Knoten einen großen Rang hat, obwohl es in seinem Unterbaum nicht viele Knoten gibt, werden die Markierungen verwendet. Die allgemeine Strategie ist folgende: Wenn ein Knoten w, der keine Wurzel ist, erstmals ein Kind verliert, wird er markiert. Wenn er nochmals ein Kind verliert, wird er selbst zu einer neuen Wurzel gemacht.

Aus der Sicht einer $decreaseKey(v, x)$-Operation sieht das dann so aus: Wenn $x < key(p(v))$ ist, wird v von seinem Vorgänger $v_1 = p(v)$ abgehängt. Wenn v_1 markiert war, wird v_1 von seinem Vorgänger $v_2 = p(v_1)$ abgehängt und zur Wurzel gemacht. Wenn v_2 markiert war ... . Abgebrochen wird, wenn ein unmarkierter Knoten erreicht wird, was spätestens dann passiert, wenn die Wurzel erreicht wird. Auf diese Weise kann eine ganze Folge $v, v_1, \ldots, v_k$ von Knoten zu neuen Wurzeln werden. Man nennt diesen Vorgang auf englisch „*cascading cut*“ (etwa: „wiederholtes Abschneiden“).

Prozedur 4.6.2 ($decreaseKey(v, x)$)**.**
Input: F-Heap H, (Zeiger auf) Knoten v, Schlüssel $x \leq key(v)$.
Ausgabe: F-Heap H' mit denselben Einträgen, Schlüssel in v auf x erniedrigt.
Methode:

(1) **if** v Wurzel **then**
    $key(v) \leftarrow x$; setze MIN-Zeiger auf v um, falls $x < key(\text{MIN})$; **return**.
(2) **if** $x \geq key(p(v))$ **then** $key(v) \leftarrow x$; **return**.
(3) // Verfolge die Folge $v_0 = v, v_1 = p(v_0), v_2 = p(v_1), \ldots$
(4) $i \leftarrow 1$; $v_1 \leftarrow p(v)$;
(5) Füge v als neue (unmarkierte) Wurzel neben dem MIN-Eintrag ein;
        // v_1 verliert Kind v
(6) $key(v) \leftarrow x$;
(7) setze MIN-Zeiger auf v um, falls $x < key(\text{MIN})$;
(8) **while** v_i ist markiert **do** // v_i ist keine Wurzel
(9) $v_{i+1} \leftarrow p(v_i)$;
(10) Füge v_i als neue Wurzel (unmarkiert) neben dem MIN-Eintrag ein
        // v_{i+1} verliert Kind v_i
(11) $i \leftarrow i + 1$;
(12) **if** v_i ist keine Wurzel **then** markiere v_i.

Die *decreaseKey*-Operation kann eventuell viele neue Wurzeln erzeugen und dabei hohen Zeitaufwand haben. Dabei verschwinden aber Markierungen, wodurch das Potenzial sinkt. Der (negative) Potenzialunterschied wird benutzt, um für diese Kosten zu bezahlen. Man beachte: Es wird jetzt *nicht* versucht, „aufzuräumen“. Vielmehr bleiben die neuen Wurzeln einfach in der Wurzelliste stehen. (*cleanup* würde zu amortisierten Kosten $O(\log n)$ führen!)

Es seien $v_0 = v, v_1, \ldots, v_k$ die Knoten, die zu neuen Wurzeln werden. Möglicherweise wird v_{k+1} markiert (wenn es keine Wurzel ist).

Wir analysieren den Zeitaufwand. Wenn v Wurzel ist oder $x \geq key(p(v))$ gilt, sich also die Struktur nicht ändert, sind die echten Kosten $O(1)$ und das Potenzial ändert sich nicht. Wir betrachten den Fall, wo Knoten umgehängt werden. Der Zeitaufwand ist dann $O(k+1)$ für das Umhängen der Knoten $v_0, v_1, \ldots, v_k$ und das Ändern der Vaterzeiger und Markierungen.

Als echte Kosten setzen wir an: $c_{decreaseKey}(H, H') = k + 1$.

Amortisierte Kosten: Das Potenzial verändert sich wie folgt: $v_1, \ldots, v_k$ sind nicht mehr markiert (Änderung -2), aber werden zu Wurzeln (Änderung $+1$), das Potenzial sinkt also um $k \cdot (2-1) = k$; Knoten v wird zu neuer Wurzel, dadurch steigt das Potenzial sicher um 1. Falls v_{k+1} neu markiert wird, steigt das Potenzial zusätzlich um 2.
Für die Potenzialänderung gilt also: $\Phi(H') - \Phi(H) \leq -k + 3$.

Die amortisierten Kosten erfüllen demnach

$$a_{decreaseKey}(H, H') = c_{decreaseKey}(H, H') + \Phi(H') - \Phi(H) \leq k + 1 + (-k + 3) = 4,$$

sie sind also durch eine Konstante beschränkt.

4.6.3 Löschen

Die Operation *delete*(v), die einen Knoten entfernt, der durch einen Zeiger gegeben ist, kann wie folgt realisiert werden: Man verringert mittels *decreaseKey* den Schlüssel in v auf einen Wert $< key(\text{MIN})$ (liefert H') und führt dann auf H' die Operation *deleteMin*() aus. Die amortisierten Kosten hierfür sind $a_{decreaseKey}(H, H') + a_{deleteMin}(H', H'') \leq 6 + 4D(n) = O(D(n))$.

Damit sind alle Operationen implementiert und die amortisierten Kosten sind ermittelt – bis auf den Schönheitsfehler, dass in den Formeln die obere Schranke $D(n)$ für den maximalen Rang bei n Einträgen vorkommt. Im nächsten und letzten Abschnitt bestimmen wir eine passende Zahl $D(n)$. Hier erklärt sich dann auch endlich der Name „Fibonacci-Heap“.

4.7 Analyse des maximalen Rangs in Fibonacci-Heaps

Wir zeigen, dass es eine Schranke $D(n)$ für den maximalen Rang (Grad, Anzahl der Kinder) eines Knotens in einem F-Heap mit n Einträgen gibt, die $D(n) = O(\log n)$ erfüllt.

Definition 4.7.1. *Die Fibonaccizahlen sind wie folgt definiert:*
$F_0 = 0$, $F_1 = 1$, $F_i = F_{i-2} + F_{i-1}$ *für* $i \geq 2$.
Bekanntlich sind folgendes die ersten Fibonaccizahlen:
$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$.

Betrachte die bekannte Zahl $\Phi = \frac{1}{2}(1 + \sqrt{5}) = 1.618\ldots$.
(Das Verhältnis $1 : \Phi$ heißt „goldener Schnitt“.)
Die Fibonaccizahlen wachsen exponentiell, im wesentlichen mit der Basis Φ:

Fakt 4.7.2. (a) $1 + \Phi = \Phi^2$. (b) *Für* $i \geq 0$ *gilt:* $F_{i+2} \geq \Phi^i$.

Beweis: (a) Nachrechnen.

(b) Vollständige Induktion.

$i = 0 : F_2 = 1 = \Phi^0$.

$i = 1$: $F_3 = 2 > \Phi^1$.

Nun sei $i \geq 2$, die Behauptung gelte für $i - 2$ und $i - 1$. Wir rechnen mit Hilfe der Definition und der Induktionsvoraussetzung:

$$F_{i+2} = F_i + F_{i+1} \overset{\text{I.V.}}{\geq} \Phi^{i-2} + \Phi^{i-1} = \Phi^{i-2}(1 + \Phi) \overset{\text{(a)}}{=} \Phi^i. \qquad \square$$

Fakt 4.7.3. *Für* $i \geq 1$ *gilt:* $1 + F_1 + \ldots + F_i = F_{i+2}$.

(Beispiel: $1+1+1+2+3+5+8 = 21$.) Allgemein ist die Gleichung ganz leicht durch Induktion zu beweisen. Für $i = 1$ ist sie richtig: $1 + F_1 = 1 + 1 = 2 = F_3$. Wenn sie für $i \geq 1$ stimmt (I.V.), folgt:

$$1 + F_1 + \ldots + F_i + F_{i+1} \overset{\text{I.V.}}{=} F_{i+2} + F_{i+1} \overset{\text{Def.}}{=} F_{i+3},$$

das ist die Induktionsbehauptung.

Wir benutzen die Fibonaccizahlen, um auszudrücken, dass die Anzahl der Nachfahren eines Knotens in einem Fibonacci-Heap exponentiell mit seinem Rang wächst.

Lemma 4.7.4. *Betrachte einen Fibonacci-Heap H. Dann gilt*:

(a) *Wenn v ein Knoten mit rank$(v) = i$ Kindern ist, dann hat der Unterbaum unter v mindestens F_{i+2} Knoten.*

(b) *Wenn H genau n Einträge hat, dann können die Rangwerte in H nicht größer als $D(n) = \lfloor 1.4405 \log_2 n \rfloor$ sein.*

Beweis: (a) Wir benutzen Induktion über die *Tiefe* des Baums T_v unter dem Knoten v. (Achtung: Man führt nicht Induktion über den Rang!)

Ind.-Anfang: T_v hat Tiefe 0, d. h. er besteht nur aus dem Knoten v. Weil $F_2 = 1$, stimmt die Behauptung.

Ind.-Schritt: Sei v Knoten mit Rang $i \geq 1$.

Es seien $w_1, \ldots, w_i$ die aktuell vorhandenen Kinder von v in der Reihenfolge, in der sie zu Kindern von v gemacht wurden (in *join*-Operationen). Der Unterbaum mit Wurzel w_j sei T_{w_j}. Betrachte ein w_j mit $j \in \{2, \ldots, i\}$. Als w_j Kind von v wurde, waren $w_1, \ldots, w_{j-1}$ schon da, also hatte v zu diesem Zeitpunkt Rang mindestens $j-1$. Nach den Regeln der *join*-Operation (die beiden Knoten haben gleichen Rang) hatte auch w_j zu diesem Zeitpunkt Rang mindestens $j - 1$. Nachher kann sich der Rang von w_j höchstens um 1 verringert haben (wenn ein zweites Kind von w_j abgetrennt worden wäre, wäre w_j nach den Regeln für die Behandlung markierter Knoten zur Wurzel gemacht worden). Also gilt aktuell:

$$\textit{rank}(w_j) \geq j - 2, \text{ für } 2 \leq j \leq i.$$

Nach Induktionsvoraussetzung (T_{w_j} hat geringere Tiefe als T_v) hat T_{w_j} mindestens $F_{(j-2)+2} = F_j$ Knoten, für $2 \leq j \leq i$.

Wir schließen: T_v hat als Knoten mindestens v und w_1 und die Knoten in $T_{w_2}, \ldots, T_{w_i}$, zusammen mindestens

$$1 + F_1 + F_2 + \ldots + F_i = F_{i+2}$$

viele (mit Fakt 4.7.3).

(b) Sei v Knoten mit Rang i in einem Fibonacci-Heap mit n Einträgen. Nach (a) hat T_v mindestens F_{i+2} Knoten, also ist $F_{i+2} \leq n$. Mit Fakt 4.7.2: $n \geq F_{i+2} \geq \Phi^i$. Durch Logarithmieren: $i \leq \log_\Phi n$, oder $i \leq (\log_\Phi 2) \cdot \log_2 n$. Dabei ist $\log_\Phi 2 = (\ln 2)/(\ln \Phi) = 1.4404\ldots < 1.4405$. □

4.8 Zusammenfassung

Fibonacci-Heaps sind eine Implementierung des Datentyps „Adressierbare Priority Queue“. Die Operationen und ihre amortisierten Kosten sind wie folgt, wobei n die

Anzahl der Einträge ist:[2]

Operation	amortisierte Kosten
$new()$	$O(1)$
$insert(e)$	$O(1)$
$makeHeap((e_1, \ldots, e_n))$	$O(n)$
$meld(H')$	$O(1)$
$min()$	$O(1)$
$deleteMin()$	$O(\log n)$
$delete(v)$	$O(\log n)$
$decreaseKey(v, x)$	$O(1)$

Anwendung: In der Vorlesung „Algorithmen und Datenstrukturen“ im Bachelorstudium wurden die Algorithmen von **Jarník/Prim** (für Minimale Spannbäume) und der Algorithmus von **Dijkstra** (für kürzeste Wege von einem Startknoten aus) vorgestellt. Beide benutzen eine adressierbare Priority Queue. Wenn man diese mit einem Fibonacci-Heap implementiert, und der eingegebene Graph $G = (V, E)$ $n = |V|$ Knoten und $m = |E|$ Kanten hat, erhält man Laufzeiten von

$$O(n \log n + m).$$

Dies liegt daran, dass n *insert*-Operationen und $\leq n$ *deleteMin*-Operationen sowie höchstens m *decreaseKey*-Operationen ausgeführt werden müssen, und dass maximal n Einträge in der Priority Queue liegen. Die gesamten *amortisierten* Kosten sind daher $O(n \log n + m)$, und nach dem allgemeinen Resultat zur Potenzialmethode aus Kapitel 3 gilt dies dann auch für die gesamten *tatsächlichen* Kosten.

Für alle Graphen mit mindestens $n \log n$ Kanten haben diese beiden Algorithmen also lineare Laufzeit!

[2] In der Praxis sind gewöhnliche *quaternäre* Heaps, also solche, die wie Binärheaps über Arrays implementiert sind, bei denen aber die Knoten der gedachten Bäume vier Kinder haben, sehr schnell, *meld* ausgenommen. Fibonacci-Heaps sind theoretisch besser. Andere modernere Heapimplementierungen, die mit Fibonacci-Heaps vergleichbar sind, heißen „Pairing heaps“, „Relaxed heaps“, „Hollow heaps“. Beschreibungen findet man in in der Originalliteratur.

Kapitel 5

Textalgorithmen

5.1 Grundbegriffe

5.1.1 Das Textsuchproblem

In diesem Kapitel geht es um das Problem der *Textsuche* (engl.: *pattern matching*). Im Hintergrund steht immer ein *Alphabet* Σ, eine endliche Menge, mit $|\Sigma| \geq 2$. *Beispiele*:

$\{0, 1\}$, das binäre Alphabet;

$\{\mathrm{A}, \mathrm{G}, \mathrm{C}, \mathrm{T}\}$[1]

ASCII, ein klassischer Code mit 128 Zeichen, erstmals standardisiert 1963;

ISO 8859-1 (Latin-1, auch DIN 66303:2000-06) oder Windows CP 1252 beschreiben Zeichensätze („ANSI-Familie“) mit 256 Buchstaben und zugehörige Binärkodierungen mit 8 Bits;

$\{0, 1\}^8$, die Binärkodierungen zu den ANSI-Alphabeten, und $\{0, 1, \ldots, 255\}$, die entsprechenden numerischen Werte;

Unicode mit momentan etwa 128 000 Zeichen.

Alphabete gleicher Größe (z. B. die Alphabete der Größe 256) unterscheiden sich nur durch die Bezeichnungen der Buchstaben. Wie die Buchstaben heißen, ist für die Zwecke der Textsuche irrelevant, und man kann auch annehmen, dass das Alphabet einfach $\{0, 1, \ldots, |\Sigma| - 1\}$ ist. Bei einigen wichtigen Algorithmen ist es sogar ganz

[1] Die Buchstaben stehen für die Namen der Nukleinbasen Adenin (A), Guanin (G), Cytosin (C) und Thymin (T), die bei der Funktionalität der DNA als genetischer Code eine zentrale Rolle spielen. Tatsächlich ist die *Computational Biology* ein zentrales Anwendungsgebiet von Textalgorithmen.

gleichgültig, was das Alphabet ist, weil auf Buchstaben a und b nur die Operation „gilt $a = b$?“ ausgeführt wird.

Notation. (a) $\Sigma^* = \{a_1 \dots a_t \mid t \geq 0, a_1, \dots, a_t \in \Sigma\}$ bezeichnet die Menge aller *Wörter* (*Strings*, *Zeichenreihen*, *Zeichenketten*) (d. h. aller endlichen Folgen) über Σ.[2]
Dabei steht $a_1 \dots, a_t$ als bequeme Abkürzung für $(a_1, \dots, a_t)$. *Beispiel*: $\{\mathtt{A}, \mathtt{B}, \mathtt{C}\}^* = \{\varepsilon, \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{AA}, \mathtt{AB}, \mathtt{AC}, \mathtt{BA}, \dots, \mathtt{ACBB}, \dots\}$.
Beachte: Für $t = 0$ erhält man immer das Wort mit 0 Buchstaben, das *leere Wort* ε.
(b) Arrayschreibweise für Wörter: $W = a_1 \dots a_t$ wird als $W[1..t]$ geschrieben; $W[i]$ bedeutet a_i, und $W[i..j]$ bedeutet das *Teilwort* $a_i \dots a_j$.
Beispiel: Wenn $W = \mathtt{ACBB}$, ist $W[2] = \mathtt{C}$, $W[1..3] = \mathtt{ACB}$ und $W[1..0] = W[4..2] = \varepsilon$.
Wir verwenden diese Notation nur für $1 \leq i \leq t+1$ und $0 \leq j \leq t$. Für $i > j$ bedeutet $W[i..j]$ stets das leere Wort ε.
(c) Ein Teilwort $W[1..i] = a_1 \dots a_i$ mit $0 \leq i \leq t$ heißt ein *Präfix* von $W = a_1 \dots a_t$.
Beispiel: Die Präfixe von $\mathtt{ACBB}$ sind ε, $\mathtt{A}$, $\mathtt{AC}$, $\mathtt{ACB}$ und $\mathtt{ACBB}$.
Ein Teilwort $W[i..t] = a_i \dots a_t$ mit $1 \leq i \leq t+1$ heißt ein *Suffix* von $W = a_1 \dots a_t$.
Beispiel: Die Suffixe von $\mathtt{ACBB}$ sind $\mathtt{ACBB}$, $\mathtt{CBB}$, $\mathtt{BB}$, $\mathtt{B}$ und ε.

Das einfache Textsuchproblem besteht in Folgendem.

Eingabe:
„Muster“ (engl.: *pattern*) $P = P[1..m] \in \Sigma^*$, $m \geq 1$,
„Text“ (engl.: *text*) $S = S[1..n] \in \Sigma^*$, $n \geq 1$.

Aufgabe: Finde erstes (oder einige oder alle) Vorkommen von P in S, d. h. finde das kleinste oder einige oder alle $\ell \in \{1, \dots, n - m + 1\}$ mit $P = S[\ell..\ell + m - 1]$.

Beispiel:
(Blanks $\sqcup$ zählen als Buchstabe; wir wollen alle Vorkommen des Musters finden.)

$S = \mathtt{IM}\sqcup\mathtt{HEUHAUFEN}\sqcup\mathtt{DIE}\sqcup\mathtt{NADEL}\sqcup\mathtt{FINDEN}$, $P = \mathtt{NADEL}$: Ausgabe: $\{18\}$

$S = \mathtt{IM}\sqcup\mathtt{NADELHAUFEN}\sqcup\mathtt{DIE}\sqcup\mathtt{NADEL}\sqcup\mathtt{FINDEN}$, $P = \mathtt{NADEL}$: Ausgabe: $\{4, 20\}$

$S = \mathtt{IM}\sqcup\mathtt{WALD}\sqcup\mathtt{DEN}\sqcup\mathtt{BAUM}\sqcup\mathtt{FINDEN}$, $P = \mathtt{NADEL}$: Ausgabe: $\emptyset$

$S = \mathtt{IM}\sqcup\mathtt{WALD}\sqcup\mathtt{DEN}\sqcup\mathtt{BAUM}\sqcup\mathtt{FINDEN}$, $P = \mathtt{WAL}$: Ausgabe: $\{4\}$.

5.1.2 Naive Algorithmen

Natürlich lässt sich das Textsuchproblem im Prinzip ganz einfach lösen: Man probiert einfach alle Möglichkeiten durch.

[2] Dabei steht $a_1 \dots, a_t$ nur als bequeme Abkürzung für $(a_1, \dots, a_t)$. Es ist also nie ein Problem, die Buchstaben voneinander abzugrenzen.

Naive Textsuche: Für jedes $\ell \in \{1, \ldots, n-m+1\}$ tue Folgendes:
Vergleiche $P[1..m]$ Buchstabe für Buchstabe mit $S[\ell..\ell+m-1]$, bis Übereinstimmung aller m Buchstaben festgestellt wird (dann wird ℓ in die Ausgabe geschrieben) oder bis eine Fehlerstelle (engl.: *mismatch*) gefunden wurde (d. h. ein Index $q \in \{0, \ldots, m-1\}$ mit $P[q+1] \neq S[\ell+q]$).

Es gibt einige Varianten dieses Prinzips, je nachdem, in welcher Reihenfolge die Plätze ℓ bearbeitet werden und die Buchstaben der Wörter P und $S[\ell..\ell+m-1]$ verglichen werden. Naheliegend ist es, jeweils von links nach rechts vorzugehen.[3]

Algorithmus 5.1.1 (**Naive Textsuche, Vergleiche von links nach rechts**).

Naive-PM-left-right(`P[1..`m`]`,`S[1..`n`]`)

Eingabe: `P[1..`m`]`, `S[1..`n`]` // `P`: Muster, `S`: Text
Ausgabe: `A` $\subseteq \{1, \ldots, n-m+1\}$;
(1) `A` $\leftarrow \emptyset$; // Initialisierung
(2) **for** `l` **from** 1 **to** $n-m+1$ **do**
(3) `q` $\leftarrow 0$;
(4) **while** `q` $< m \wedge$ `P[q+1]` $=$ `S[l+q]` **do** `q` $\leftarrow$ `q+1`;
(5) **if** `q` $= m$ **then** `A` $\leftarrow$ `A` $\cup \{$`l`$\}$;
(6) **return** `A`.

Anschaulich stellt man sich vor, dass man für jedes $\ell = 1, \ldots, n-m+1$ nacheinander das Muster P unter den Abschnitt $S[\ell..\ell+m-1]$ des Textes legt und von links nach rechts gehend Paare von Buchstaben vergleicht. Von Runde zu Runde wird das Muster also um eine Position nach rechts verschoben.

Beispiel: Muster $P =$ `abra` ($m = 4$), Text $S =$ `abracababrabrac` ($n = 15$). Die folgende Tabelle zeigt die 12 Positionen für das Muster und die für die jeweilige Position verglichenen Buchstaben:

[3] Bei den später behandelten Algorithmen der Boyer-Moore-Familie wird es sich als nützlich herausstellen, Positionen ℓ in aufsteigender Reihenfolge zu betrachten, für eine Position ℓ die Buchstaben in $P[1..m]$ jedoch von rechts nach links.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\ell \setminus$ Text	a	b	r	a	c	a	b	a	b	r	a	b	r	a	c
1	a	b	r	a											
2		a	b	r	a										
3			a	b	r	a									
4				a	b	r	a								
5					a	b	r	a							
6						a	b	r	a						
7							a	b	r	a					
8								a	b	r	a				
9									a	b	r	a			
10										a	b	r	a		
11											a	b	r	a	
12												a	b	r	a

Alle 12 Positionierungen des Musters sind angegeben, verglichene Buchstaben sind grau (Übereinstimmung: hellgrau, Fehlerstelle: dunkelgrau). Das Muster kommt dreimal vor. Insgesamt gibt es 24 Vergleiche. Man bemerkt, dass $S[4]$ und $S[11]$ zweimal erfolgreich mit Musterbuchstaben verglichen werden, und dass $S[9]$ und $S[10]$ erfolglos mit $P[1]$ verglichen werden, obwohl man vorher schon gesehen hat, dass an diesen Stellen `b` und `r`, also nicht $P[1] = \texttt{a}$ steht.

Die Laufzeit des naiven Algorithmus beträgt $O((n-m+1)m)$ im schlechtesten Fall; die Anzahl der Buchstabenvergleiche ist maximal $(n-m+1)m$. Folgendes *Beispiel* zeigt, dass der schlechteste Fall auch eintreten kann: Muster $P = \texttt{a}^{m-1}\texttt{b}$ und Text $S = \texttt{a}^{n-1}\texttt{b}$ führen zu genau $(n-m+1)m$ Buchstabenvergleichen, weil in jeder Position ℓ die Fehlerstelle erst nach m Vergleichen gefunden wird. Oft ist die Vergleichszahl geringer, weil in der inneren Schleife schon früh eine Fehlerstelle gefunden wird. Besonders bei natürlichsprachigen Texten wird dies so sein. Man kann dies mit den Beispielen in Abschnitt 5.1.1 ausprobieren. Allgemein gilt: Wenn Muster P oder Text S aus rein zufällig aus Σ^m bzw. Σ^n sind, dann ist die erwartete Anzahl von Buchstabenvergleichen bei der naiven Textsuche kleiner als $(n-m+1) \cdot \frac{|\Sigma|}{|\Sigma|-1} < 2(n-m+1)$.

In diesem Kapitel betrachten wir Algorithmen, die eine lineare Laufzeit von $O(m+n)$ *garantieren*. Dies sind der Algorithmus von Knuth, Morris und Pratt und der Algorithmus von Boyer und Moore. Daneben betrachten wir einen Algorithmus, der nach Vorkommen von Wörtern aus einer ganzen Liste von Mustern sucht (Algorithmus von Aho und Corasick).

5.1.3 Ränder

Grundlegend für viele Textalgorithmen ist der Begriff des **Randes** (engl.: *border*). Anschaulich ist ein Rand eines Wortes W ein kürzeres Teilwort, das man am linken

Ende und ebenso am rechten Ende von W findet.

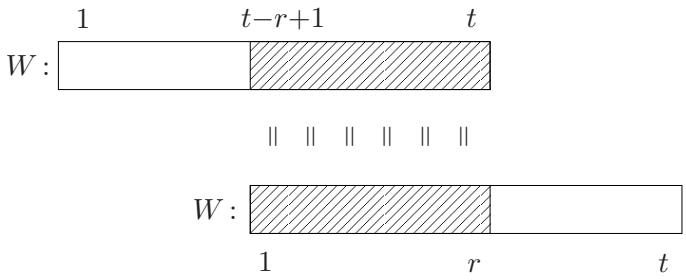


Abbildung 5.1: Ein Teilwort $\neq W$ von W, das sowohl Präfix als auch Suffix von W ist, heißt ein *Rand* von W.

Definition 5.1.2. *Sei $W = a_1 \dots a_t$ ein Wort, $t \geq 0$. $W[1..r]$ heißt ein **Rand** von W, wenn $0 \leq r < t$ und $W[1..r]$ (nicht nur Präfix, sondern auch) Suffix von W ist, d. h. wenn $W[1..r] = W[t-r+1..t]$ gilt* (s. Abb. 5.1).

Man beachte folgende Sonderfälle:
(a) Jedes nichtleere Wort $W = W[1..t]$ hat $\varepsilon = W[1..0] = W[t+1..t]$ als Rand.
(b) W selbst zählt *nicht* als Rand von W, obwohl es Präfix und Suffix von sich selbst ist. (Mitunter wird W als „uneigentlicher/unechter Rand“ von W bezeichnet.)
(c) Das leere Wort ε hat überhaupt keinen Rand, da es kein r mit $0 \leq r < 0$ gibt.

Beispiel: Die Ränder von `abababcabab` sind ε, `ab` und `abab`. Die Ränder von `ababa` sind ε, `a` und `aba`.

Uns interessiert oft der *längste* Rand von W. Der längste Rand von `ababab` ist `abab`, der längste Rand von `ababcabc` ist ε.

Die längsten Ränder der *Präfixe* des Musters $P[1..m]$ sind von besonderem Interesse. Man könnte die Funktion

$$\text{bord}\colon \{P[1..q] \mid 0 \leq q \leq m\} \to \{P[1..q] \mid 0 \leq q < m\} \cup \{-\},$$

$$P[1..q] \mapsto \begin{cases} \text{längster Rand } P[1..q'] \text{ von } P[1..q] & , \text{ falls } 1 \leq q \leq m; \\ - & , \text{ falls } q = 0. \end{cases}$$

betrachten, die jedem Präfix $P[1..q]$ seinen längsten Rand $P[1..q']$ zuordnet, bzw. den Wert „$-$“ (für „undefiniert“) für $P[1..0] = \varepsilon$. Es ist aber technisch bequem, eine Zahlfunktion auf den entsprechenden Längen zu benutzen.

Definition 5.1.3 (Randfunktion). *Gegeben sei das Muster $P[1..m]$. Wir definieren die Funktion $f_{\text{bord}} = f^P_{\text{bord}}$ mit $f_{\text{bord}}\colon \{0, 1, \dots, m\} \to \{-1, 0, \dots, m-1\}$ durch:*

$$f_{\text{bord}}(q) := \begin{cases} \text{Länge des längsten Randes von } P[1..q] & , \text{ falls } 1 \leq q \leq m; \\ -1 & , \text{ falls } q = 0. \end{cases}$$

Offensichtlich ist immer $f_{\text{bord}}(q) < q$. Die Festlegung $f_{\text{bord}}(0) = -1$ wird sich auch (programmier-)technisch als günstig erweisen.

q	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`P[`q`]`		a	b	r	a	c	a	b	a	b	r	a	b	r	a	c
$f_{\text{bord}}(q)$	−1	0	0	0	1	0	1	2	1	2	3	4	2	3	4	5

Abbildung 5.2: Randfunktion f_{bord} bei $P[1..15] =$ `abracababrabrac`.

In Abb. 5.2 ist die Randfunktion für das Muster `abracababrabrac` angegeben. Es gilt zum Beispiel $f_{\text{bord}}(10) = 3$, weil $\text{bord}(P[1..10]) = \text{bord}(\texttt{abracababr}) = \texttt{abr} = P[1..3]$ gilt. – Es gibt effiziente Verfahren für die Berechnung einer Tabelle für die Randfunktion. Um diese Algorithmen kümmern wir uns später. (WS 2020/21: Übung.)

Übungsaufgabe. Wie kann man bei gegebener Randfunktion f_{bord} zu gegebenem q *alle* Ränder von $P[1..q]$, d. h. alle r, für die $P[1..r]$ Rand von $P[1..q]$ ist, berechnen? Für jeden auszugebenden Index r soll nur Zeit $O(1)$ aufgewendet werden.

5.2 Der Algorithmus von Knuth, Morris und Pratt

5.2.1 Vorüberlegungen

Der Algorithmus von Knuth, Morris und Pratt (KMP-Algorithmus)[4] geht vom naiven Textsuchalgorithmus 5.1.1 aus, also der Version, in der für jede Stelle $\ell = 1, \ldots, n - m + 1$ im Text $S[1..n]$ die Musterbuchstaben $P[q + 1]$, $q = 0, \ldots, m - 1$, von links nach rechts mit den Textbuchstaben $S[\ell + q]$, $q = 0, \ldots, m - 1$, verglichen werden. Auch der KMP-Algorithmus stellt diese Vergleiche von Wörtern für eine monotone Folge von Werten ℓ an, versucht aber nach Möglichkeit,

> ℓ-Werte zu überspringen (also das Muster in einem Schritt um ein größeres Stück nach rechts zu schieben: „*shift*“ um mehr als eine Position) und
>
> überflüssige Buchstabenvergleiche auszulassen.

Wir erklären die Ideen genauer.[5] Angenommen, der Algorithmus ist an Position $\ell \leq n - m + 1$ angekommen, und wir haben das größte $q \in \{0, 1, \ldots, m\}$ mit $P[1..q] = S[\ell..\ell + q - 1]$ ermittelt. Es gibt zwei Fälle:

[4] Donald E. Knuth, James H. Morris, Vaughan R. Pratt (1977): Fast Pattern Matching in Strings. SIAM Journal on Computing. **6**(2): 323–350. doi:10.1137/0206024

[5] Die Entwicklung des KMP-Algorithmus aus dem naiven Algorithmus wird recht detailliert beschrieben, als ein Lehrbeispiel dafür, wie man durch genaue Betrachtung und Verbesserung eines einfachen Algorithmus zu einer sehr effizienten Version kommen kann.

1. Fall: $q < m$ und $P[q+1] \neq S[\ell+q]$. – Dann kann an Stelle ℓ das Muster nicht vorkommen. Wir verschieben $P[1..m]$ an eine neue Position $\ell' > \ell$ und testen erneut auf Übereinstimmung (bzw. hören auf, wenn $\ell' > n-m+1$). Der naive Algorithmus versucht die Stelle $\ell' = \ell+1$ und vergleicht $P[1..m]$ mit $S[\ell'..\ell'+m-1]$ buchstabenweise von links nach rechts. Der KMP-Algorithmus nutzt das Wissen

$$P[1..q] = S[\ell..\ell+q-1] \quad \text{und} \quad P[q+1] \neq S[\ell+q] \tag{5.1}$$

geschickt aus, und zwar doppelt! Erstens ist es überflüssig, Positionen $\ell' \in \{\ell+1, \ldots, \ell+q\}$ zu testen, für die aus (5.1) schon folgt, dass sie nicht als Position des Musters in Frage kommen. Zweitens kann man für eine mögliche neue Position $\ell' \in \{\ell+1, \ldots, \ell+q-1\}$ ausnutzen, dass schon erfolgreiche Vergleiche mit den Buchstaben $S[\ell'..\ell+q-1]$ im Text durchgeführt worden sind.

In Abb. 5.3 und Abb. 5.4 sind die möglichen Situationen dargestellt. Wir betrachten zuerst einen Verschiebung um einen „Shiftwert“ $s \in \{1, \ldots, q\}$, von Position ℓ zu Position $\ell' = \ell + s$. Wir definieren $q' = q - s \geq 0$. Dann gilt $\ell + q = \ell' + q'$.

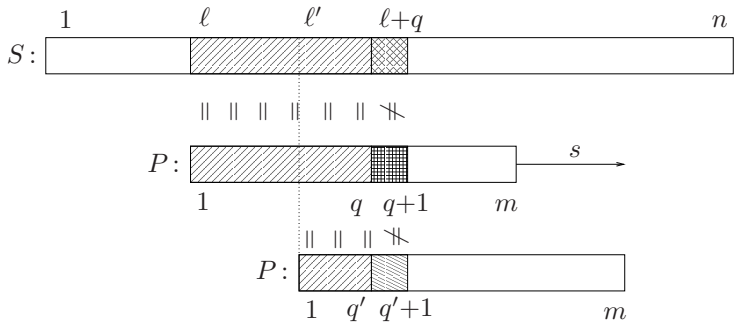


Abbildung 5.3: Grundidee des KMP-Algorithmus, 1. Fall, Standardsituation: Eine Verschiebung um $s = \ell' - \ell = q - q' \leq q$ kann nur sinnvoll sein, wenn $P[1..q'] = P[q-q'+1..q]$ und $P[q'+1] \neq P[q+1]$ gelten. Diese Überlegung gilt auch im Fall $q = 1$ und $q' = 0$. Im Fall einer solchen Verschiebung ist der nächste Vergleich zwischen $S[\ell+q]$ und $P[q'+1]$ auszuführen.

Wenn das Muster an Position ℓ' vorkommt, muss

$$P[1..q'] = S[\ell'..\ell+q-1] \quad \text{und} \quad P[q'+1] = S[\ell+q]$$

gelten. In Kombination mit (5.1) liefert dies die notwendige Bedingung

$$P[1..q'] = P[q-q'+1..q] \quad \text{und} \quad P[q'+1] \neq P[q+1]. \tag{5.2}$$

Shiftwerte $s = q - q'$, die (5.2) erfüllen, heißen *zulässig*. Die erste Gleichheit in (5.2) bedeutet, dass $P[1..q']$ Rand von $P[1..q]$ sein muss, die zweite ergänzt, dass in P auf $P[1..q']$ und $P[1..q]$ verschiedene Buchstaben folgen müssen. Wenn es ein zulässiges

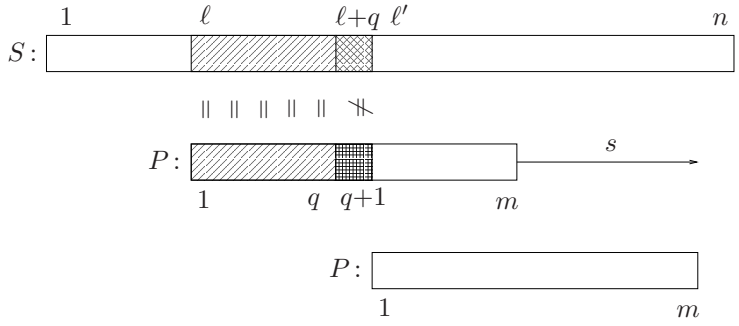


Abbildung 5.4: Grundidee des KMP-Algorithmus, 1. Fall, Sonderfall: Eine Verschiebung zur Position $\ell' = \ell+q+1$, also mit $s = q+1$ und $q' = q-s = -1$, kann nie ausgeschlossen werden, da kein Buchstabe in $S[\ell'..\ell' + m - 1]$ bisher gesehen wurde. Im Fall einer solchen Verschiebung ist der nächste Vergleich zwischen $S[\ell+q+1]$ und $P[1] = P[q'+2]$ auszuführen.

$s \leq q$ gibt, dann verschieben wir das Muster um den *kleinsten* zulässigen Wert s, setzen also $\ell' := \ell + s$. Damit ist sichergestellt, dass keine mögliche Position des Musters übersehen wird. Mit den buchstabenweisen Vergleichen von links nach rechts geht es nun weiter wie im naiven Algorithmus, allerdings brauchen wir in P erst bei $P[q' + 1]$ und in S bei $S[\ell' + q'] = S[\ell + q]$ anzufangen, da die Gleichheit $P[1..q'] = S[\ell'..\ell' + q' - 1]$ wegen (5.1) schon gesichert ist.

Es gibt noch einen Sonderfall. Wenn es überhaupt kein zulässiges $s \in \{1, \ldots, q\}$ gibt, dann kommt keine der Positionen $\ell + 1, \ldots, \ell + q$ als neues ℓ' in Frage. Wir setzen dann $\ell' = \ell + q + 1$, also $s = q + 1$, was zu $q' = q - s = -1$ führt. Diese Position ℓ' kann (nach der in (5.1) angegebenen Information) nie ausgeschlossen werden, da kein Buchstabe aus $S[\ell'..n]$ dort erwähnt wird. Der Shiftwert $s = q+1$ gilt also *immer* als zulässig. Wenn dies der kleinste zulässige Wert ist, wird um $q+1$ verschoben. Der erste neue Vergleich ist in diesem Fall natürlich an der Stelle $P[1]$ und $S[\ell'] = S[\ell + q + 1]$ auszuführen. Dieser Sonderfall tritt im 1. Fall für $q = 0$ (also bei $P[1] \neq S[\ell]$) immer ein.

2. Fall: $q = m$. – Dann steht das Muster P im Bereich $S[\ell..\ell + q - 1]$, und wir geben ℓ aus. Wenn wir alle Vorkommen des Musters in S finden sollen, müssen wir noch weiterarbeiten. Wir suchen also eine neue Position $\ell' > \ell$, indem wir das Wissen

$$P[1..m] = S[\ell..\ell + m - 1] \tag{5.3}$$

geschickt ausnutzen. In Abb. 5.5 ist diese Situation dargestellt.

Weil für eine erfolgreiche Position $\ell' = \ell + s = \ell + q - q'$ die Gleichheit $P[1..q'] = S[\ell'..\ell' + q' - 1] = S[\ell'..\ell + m - 1]$ gelten muss, folgt die notwendige Bedingung

$$P[1..q'] = P[m - q' + 1..m]. \tag{5.4}$$

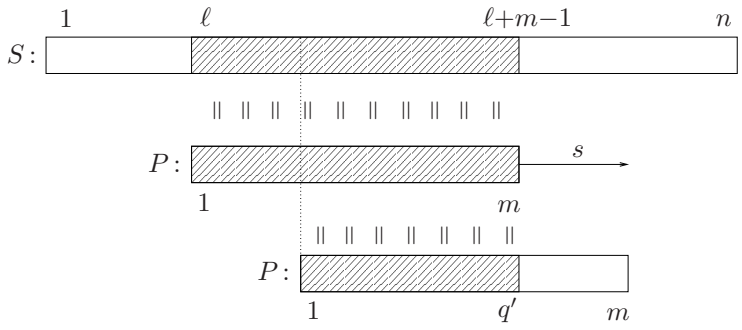


Abbildung 5.5: Grundidee des KMP-Algorithmus, 2. Fall: Wir wissen, dass $P[1..m] = S[\ell..\ell+m-1]$ gilt. Eine Verschiebung des Musters an eine Stelle $\ell' = \ell + s$ kann nur sinnvoll sein, wenn für $q' = q - s$ das Präfix $P[1..q']$ ein Rand von P ist. (Es gibt immer mindestens einen Rand, nämlich ε.)

In diesem Fall gelten also alle Shiftwerte s, für die $P[1..m-s]$ ein Rand des Musters $P[1..m]$ ist, als zulässig. ($s = m$ ist hier immer zulässig.) Von allen zulässigen Werten s wählen wir den kleinsten, damit keine mögliche Position des Musters übersehen wird, und setzen $\ell' := \ell + s$ und $q' := m - s$. In der Bearbeitung der Position ℓ' können wir mit dem Vergleich zwischen $P[q'+1]$ und $S[\ell'+q'] = S[\ell+m]$ fortfahren, weil schon bekannt ist, dass $P[1..q']$ und $S[\ell'..\ell'+q'-1]$ übereinstimmen.

5.2.2 Die Fehlerfunktion

In den Vorüberlegungen zum KMP-Algorithmus muss man nach dem Finden einer Fehlerstelle oder dem Auftreten des Musters die richtige (kleinste) zulässige Shiftweite $s = \ell' - \ell = q - q'$ finden. Aus den Bedingungen (5.2) und (5.4) ist klar, dass diese Shiftweite nur von P und q abhängt, nicht von S. Wir können also diese Shiftweite für $q = 0, \ldots, m$ aus $P[1..m]$ vorab berechnen. Es ist technisch günstig, nicht die kleinste Shiftweite s, sondern den neuen Wert $q' = q - s$ zu betrachten. Die Funktion $q \mapsto q'$, die *KMP-Fehlerfunktion*, ist eine verschärfte Variante der Randfunktion aus Abschnitt 5.1.3. Für ein Präfix $P[1..q]$ mit $q < m$ sind nur solche Ränder $P[1..q']$ interessant, die $P[q'+1] \neq P[q+1]$ erfüllen.

Definition 5.2.1 (**KMP-Fehlerfunktion**). *Gegeben sei das Muster $P[1..m]$. Die KMP-Fehlerfunktion $f_{\mathrm{KMP}}\colon \{0,1,\ldots,m\} \to \{-1,0,\ldots,m-1\}$ ist wie folgt definiert*:

$$f_{\mathrm{KMP}}(q) := \begin{cases} \text{Länge } q' \text{ des längsten Randes } P[1..q'] \text{ von } P[1..m], \text{ falls } q = m; \\ \text{Länge } q' \text{ des längsten Randes } P[1..q'] \text{ von } P[1..q] \\ \qquad\qquad \text{mit } P[q'+1] \neq P[q+1], \text{ falls ein solcher existiert,} \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{für } 0 < q < m; \\ -1, \text{ falls kein solcher Rand existiert, für } 0 \leq q < m. \end{cases}$$

Bemerkung: Man kann $f_{\mathrm{KMP}}(q)$ auch kompakt wie folgt beschreiben: $f_{\mathrm{KMP}}(q)$ ist die *größte* Zahl $q' \in \{-1, 0, 1, \ldots, q-1\}$ mit:

$$P[1..q'] = P[q-q'+1..q] \quad \text{und} \quad (q' \geq 0 \wedge q < m) \Rightarrow P[q'+1] \neq P[q+1]. \tag{5.5}$$

Die erste Aussage in (5.5) gilt trivialerweise für $q' \in \{-1, 0\}$, die zweite gilt trivialerweise für $q' = -1$ und ebenso für $q = m$. Also ist (5.5) durch $q' = -1$ immer erfüllt, und im Fall $q = m$ durch $q' = 0$ erfüllt.

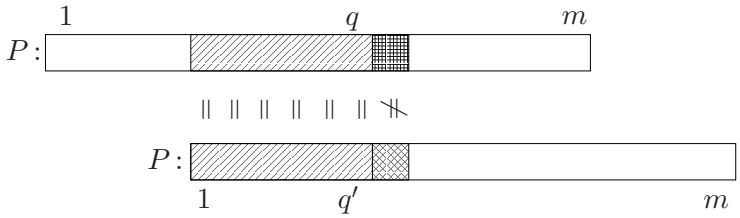


Abbildung 5.6: Für $0 \leq q < m$ ist $f_{\mathrm{KMP}}(q)$ die Länge des längsten Randes $P[1..q']$ von $P[1..q]$, so dass die folgenden Buchstaben $P[q'+1]$ und $P[q+1]$ verschieden sind – falls ein solcher Rand existiert. Sonst ist $f_{\mathrm{KMP}}(q) = -1$. Anmerkung: $f_{\mathrm{KMP}}(0) = -1$, weil $P[1..0] = \varepsilon$ überhaupt keinen Rand hat. $f_{\mathrm{KMP}}(m)$ ist einfach $f_{\mathrm{bord}}(m)$.

In Abb. 5.7 sind die Randfunktion und die KMP-Fehlerfunktion für $P = \texttt{abracababrabrac}$ angegeben, zusammen mit Beispielen für die verschiedenen möglichen Situationen. Es gilt zum Beispiel:

$f_{\mathrm{KMP}}(1) = 0$, weil für den Rand $P[1..0] = \varepsilon$ von $P[1..1]$ die zweite Bedingung erfüllt ist ($P[1] = \texttt{a} \neq \texttt{b} = P[2]$);

$f_{\mathrm{KMP}}(4) = 1$, weil für den längsten Rand $P[1..1] = \texttt{a}$ von $P[1..4] = \texttt{abra}$ die zweite Bedingung erfüllt ist ($P[2] = \texttt{b} \neq \texttt{c} = P[5]$);

$f_{\mathrm{KMP}}(5) = -1$, weil für den einzigen Rand $P[1..0] = \varepsilon$ von $P[1..5] = \texttt{abrac}$ die zweite Bedingung nicht erfüllt ist ($P[1] = \texttt{a} = P[6]$);

$f_{\mathrm{KMP}}(11) = 4$, weil für den längsten Rand $P[1..4] = \texttt{abra}$ von $P[1..11]$ die zweite Bedingung erfüllt ist ($P[5] = \texttt{c} \neq \texttt{b} = P[12]$);

$f_{\mathrm{KMP}}(14) = 1$, weil für den längsten Rand $P[1..4] = \texttt{abra}$ von $P[1..14]$ die zweite Bedingung nicht erfüllt ist ($P[5] = \texttt{c} = P[15]$), aber für den zweitlängsten Rand $P[1..1] = \texttt{a}$ die zweite Bedingung erfüllt ist ($P[2] = \texttt{b} \neq \texttt{c} = P[15]$);

$f_{\mathrm{KMP}}(15) = 5$, weil $m = 15$ und $P[1..5] = \texttt{abrac}$ der längste echte Rand des Musters ist.

Ein Verfahren zur effizienten Berechnung einer Tabelle für die KMP-Fehlerfunktion wird später vorgestellt. Wir bemerken aber schon hier, dass die KMP-Fehlerfunktion leicht iterativ aus der gewöhnlichen Randfunktion berechnet werden kann.

q	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
$P[q]$		a	b	r	a	c	a	b	a	b	r	a	b	r	a	c	
$f_{\text{bord}}(q)$	−1	0	0	0	1	0	1	2	1	2	3	4	2	3	4	5	
$f_{\text{KMP}}(q)$	−1	0	0	−1	1	−1	0	2	0	0	−1	4	0	−1	1	5	
$P[1..0], P[1]$	ε	a															
$P[1..15]$			a	b	r	a	c	a	b	...							
$P[1..1], P[2]$		a	b														
$P[1..15]$			a	b	r	a	c	a	b	...							
$P[1..4], P[5]$		a	b	r	a	c											
$P[1..15]$					a	b	r	a	c	a	b	...					
$P[1..5], P[6]$		a	b	r	a	c	a										
$P[1..15]$								a	b	r	a	c	a	b	...		
$P[1..11], P[12]$		a	b	r	a	c	a	b	a	b	r	a	b				
$P[1..15]$									a	b	r	a	c	a	b	...	
$P[1..14], P[15]$		a	b	r	a	c	a	b	a	b	r	a	b	r	a	c	
$P[1..15]$															a	b	...
$P[1..15]$		a	b	r	a	c	a	b	a	b	r	a	b	r	a	c	
$P[1..15]$												a	b	r	a	c	...

Abbildung 5.7: Randfunktion und KMP-Fehlerfunktion f_{KMP} für $P[1..15]$ = `abracababrabrac`, mit Beispielen für die Positionierung der entsprechenden Ränder.

q	0	1	2	3	4	5	6
`P[`q`]`		a	b	r	a	c	a
$f_{\mathrm{KMP}}(q)$	-1	0	0	-1	1	-1	1

Abbildung 5.8: KMP-Fehlerfunktion des Musters $P =$ `abraca` ($m = 6$).

Übungsaufgabe. Angenommen, f_{bord} ist (z. B. als Tabelle) gegeben. Man beweise, dass die folgende iterative Berechnungsvorschrift für f_{KMP} korrekt ist:

(1) $f_{\mathrm{KMP}}(0) \leftarrow -1$;

(2) **for** q **from** 1 **to** $m-1$ **do**
  $q' \leftarrow f_{\mathrm{bord}}(q)$; // ein Wert in $\{0, \ldots, q-1\}$
  if $P[q'+1] \neq P[q+1]$ **then** $f_{\mathrm{KMP}}(q) \leftarrow q'$ **else** $f_{\mathrm{KMP}}(q) \leftarrow f_{\mathrm{KMP}}(q')$;

(3) $f_{\mathrm{KMP}}(m) \leftarrow f_{\mathrm{bord}}(m)$.

Wir können den Algorithmus von Knuth, Morris und Pratt („KMP-Algorithmus“) jetzt einfach aufschreiben. Für die veränderlichen Werte q und q' genügt eine Variable `q`, für ℓ und ℓ' eine Variable `l`. Für den Shiftwert s benutzen wir eine Variable `s`.

Algorithmus 5.2.2 (**Knuth-Morris-Pratt, anschaulich**)**.**

KMP-Textsuche(`P[1..`m`]`,`S[1..`n`]`)

Eingabe: `P[1..`m`]`, `S[1..`n`]`; // Muster, Text
Ausgabe: `A` $\subseteq \{1, \ldots, n-m+1\}$;
Vorberechnet: `f_KMP[0..`m`]`: KMP-Fehlerfunktion von `P[1..`m`]` als Tabelle;

(1) `A` $\leftarrow \emptyset$;
(2) `q` $\leftarrow 0$; `l` $\leftarrow 1$;
(3) **while** `l` $\leq n-m+1$ **do**
(4) **while** `q` $< m \wedge$ `P[q+1]` $=$ `S[l+q]` **do** `q` $\leftarrow$ `q+1`;
(5) **if** `q` $= m$ **then** `A` $\leftarrow$ `A` $\cup \{$`l`$\}$; // Muster gefunden
(6) `s` $\leftarrow$ `q - f_KMP[q]`;
(7) `l` $\leftarrow$ `l + s`;
(8) `q` $\leftarrow$ `q - s`; // d. h. `q` $\leftarrow$ `f_KMP[q]`
(9) **if** `q` $= -1$ **then** `q` $\leftarrow 0$;
(10) **return** `A`.

Beispiel: Muster $P =$ `abraca` ($m = 6$), Text $S =$ `babracababradabrab` ($n = 18$). Die KMP-Fehlerfunktion für P ist in Abb. 5.8 angegeben.

ℓ	S: b	a	b	r	a	c	a	b	a	b	r	a	d	a	b	r	a	b	q	q'	s
1	a	b	r	a	c	a													0	−1	1
2		a	b	r	a	c	a												6	1	5
7							a	b	r	a	c	a							2	0	2
9									a	b	r	a	c	a					4	1	3
12												a	b	r	a	c	a		1	0	1
13													a	b	r	a	c	a	0	−1	1
14														a	...					STOP	

Hellgrau sind die Musterbuchstaben, die erfolgreich verglichen werden, dunkelgrau die Fehlerstellen. Dass in jeder Spalte nur ein hellgrauer Buchstabe steht, bedeutet, dass kein Textbuchstabe mehr als einmal erfolgreich verglichen wird. Im Vergleich zum naiven Algorithmus fehlen viele Zeilen. Für jede der getesteten Positionen $\ell \in \{1, \ldots, n-m+1\}$ gibt es maximal einen Fehlerbuchstaben (das ist klar: mit der Entdeckung der Fehlerstelle ist diese Position erledigt). Im Beispiel werden 16 Buchstabenvergleiche durchgeführt; das naive Verfahren würde 26 Vergleiche benötigen.

Satz 5.2.3. *Algorithmus 5.2.2 ist korrekt und hat Rechenzeit $O(n)$. Die Anzahl der durchgeführten Buchstabenvergleiche ist maximal $2n - m + 1$.*

Beweis: Die Korrektheit folgt aus den Vorüberlegungen. Der Ablauf ist ganz genau wie im naiven Algorithmus, mit dem Unterschied, dass einige ℓ-Werte ausgelassen werden und einige Vergleiche eingespart werden, weil die entsprechenden Gleichheiten schon aus früheren Runden bekannt sind.

Für die Rechenzeit überlegt man sich folgendes. Die große **while**-Schleife (Zeilen (3)–(9)) wird höchstens $(n-m+1)$-mal durchlaufen, weil sich der Inhalt ℓ von `l` in jedem Durchlauf erhöht und bei $\ell > n-m+1$ abgebrochen wird. In jedem Durchlauf dieser großen Schleife gibt es maximal einen Test der Schleife in Zeile (4), der zum Abbruch führt (Fehlerstelle). Also gibt es höchstens $n-m+1$ erfolglose Vergleiche. Es bleiben die Durchläufe durch die Schleife in Zeile (4), in denen die Buchstabenvergleiche erfolgreich sind. Hierfür betrachten wir den Wert $\ell + q$, mit ℓ in `l` und q in `q`. Dieser Wert startet bei 1, wird nie erniedrigt und wird bei jedem „erfolgreichen“ Vergleich im Schleifentest in Zeile (4) echt erhöht; der Algorithmus endet auf jeden Fall, wenn $\ell + q$ größer als n wird. Also kann es nicht mehr als n erfolgreiche Vergleiche und nicht mehr als n solche Schleifendurchläufe geben. □

In Abschnitt 5.2.3 werden wir sehen, wie sich die Randfunktion (und damit auch die KMP-Fehlerfunktion) in Zeit $O(m \cdot |\Sigma|)$ berechnen lässt; in Abschnitt 5.2.4 wird dann ein Algorithmus angegeben, der in Zeit $O(m)$ eine Tabelle für die KMP-Fehlerfunktion f_{KMP} berechnet.

5.2.3 Der KMP-Algorithmus in der Präfixautomaten-Auffassung

Wir haben gesehen, dass der KMP-Algorithmus als eine optimierte Version des naiven Textsuchalgorithmus verstanden werden kann, die das Muster in Sprüngen weiterschiebt und Vergleiche einspart, deren Ergebnis schon bekannt ist. Es ist lehrreich und nützlich, insbesondere für Verallgemeinerungen wie in Abschnitt 5.3, den Algorithmus noch auf eine andere, abstraktere Weise zu betrachten, die einen Bezug zur Verarbeitung von Wörtern durch endliche Automaten[6] herstellt. Als Vorbereitung betrachten wir einen recht einfachen endlichen Automaten. Der eigentliche KMP-Automat ist noch etwas subtiler aufgebaut.

Der volle Präfixautomat

Gegeben sei ein Muster $P = P[1..m]$. Wir konstruieren einen deterministischen endlichen Automaten (DFA) $M^P_{\text{voll}} = (Q, \Sigma, q_0, F, \delta)$. Dieser Automat soll den Text S Buchstabe für Buchstabe von links nach rechts lesen und dabei über den Akzeptierungsmechanismus melden, wenn das Muster gefunden worden ist. Die Zustandsmenge Q repräsentiert die Präfixe $P[1..q]$, $0 \leq q \leq m$, des Musters. Ein solches Präfix wird einfach durch seine Länge dargestellt, also wählen wir

$$Q = Q_P = \{0, 1, \ldots, m\}.$$

Das Alphabet $\Sigma = \Sigma_P$ ist fest; es enthält mindestens die Buchstaben, die in P und S vorkommen.[7] Der Startzustand q_0, die Übergangsfunktion $\delta = \delta_P$ mit $\delta\colon Q \times \Sigma \to Q$ und die Menge $F = F_P$ der akzeptierenden Zustände werden im Folgenden festgelegt.

Die Idee ist folgende: Wir lesen den Text $S[1..n]$ Buchstabe für Buchstabe. Nach Leseschritt i, also nachdem $S[1..i]$ gelesen wurde, soll der Zustand $q^{(i)}$ das *längste Präfix* $P[1..q]$ von P benennen, das Suffix von $S[1..i]$ ist. Dies formulieren wir als Invariante:

($\mathbf{I_i}$): Nach Lesen von $S[1..i]$ ist M^P_{voll} in Zustand

$$q^{(i)} = \max\{q \in Q \mid P[1..q] = S[i-q+1..i]\}.$$

Wie müssen Startzustand und Übergangsfunktion aussehen, damit stets (I_i) gilt? Am Anfang soll $P[1..q^{(0)}]$ das längste Präfix von P sein, das Suffix von $[1..0] = \varepsilon$ ist; das ist natürlich das leere Wort, und wir legen fest:

$$q_0 := 0.$$

[6] s. Vorlesungen „Rechnerorganisation“ und „Automaten, Sprachen und Komplexität“ im Bachelorstudium.

[7] Alle Buchstaben, die in S, aber nicht in P vorkommen, werden gleich behandelt. Daher kann man sie im Automaten wie einen einzigen „Fremdbuchstaben“ behandeln, und ihre Anzahl ist unerheblich.

Nun betrachten wir einen beliebigen Schritt $i > 0$ und nehmen an, dass die Konstruktion bislang erfolgreich war, dass also $q^{(i-1)}$ Invariante (I_{i-1}) erfüllt. Buchstabe $S[i]$ wird gelesen. Sei $P[1..q']$ das längste Präfix von P, das Suffix von $S[1..i]$ ist. Wir wollen erreichen, dass der neue Zustand $q^{(i)}$ gerade q' ist. Zunächst einmal ist sicher $P[1..q'-1]$ Suffix von $S[1..i-1]$, also folgt aus (I_{i-1}) die Beziehung $q'-1 \leq q^{(i-1)}$ oder $q' \leq q^{(i-1)}+1$. Die Entscheidung darüber, welche Präfixe von P Suffix von $S[1..i]$ sind, hängt also nur von $S[i-q^{(i-1)}..i]$ ab, und das ist das gleiche wie $P[1..q^{(i-1)}] \circ S[i]$, wobei mit $\circ$ die Konkatenation von Wörtern gemeint ist. Daher sollte $P[1..q']$ das längste Präfix von P sein, das Suffix von $P[1..q^{(i-1)}] \circ S[i]$ ist.

Wir setzen daher für $q \in Q$ und $a \in \Sigma$:

$$\delta(q,a) := \text{Länge des längsten Präfixes von } P, \text{ das Suffix von } P[1..q] \circ a \text{ ist.}$$

Dann gilt $q^{(i)} = \delta(q^{(i-1)}, S[i]) = q'$, wie gewünscht. Wir beobachten gleich noch Folgendes: Diese Definition von δ führt dazu, dass alle Buchstaben $a \in \Sigma$, die nicht in P vorkommen, $\delta(q,a) = 0$ erfüllen. Man braucht diese im Automaten also gar nicht zu unterscheiden, sondern kann sie zu einer Kategorie „Fremdbuchstabe“ zusammenfassen. Damit hängt δ nur von P ab, und δ kann in einer Vorverarbeitungsphase berechnet werden, ohne Kenntnis von S. (Wir werden gleich sehen, wie diese Vorverarbeitung abläuft.)

Der Automat soll nach dem Lesen von $S[1..i]$ genau dann in einem akzeptierenden Zustand sein, wenn soeben ein Vorkommen des Musters fertig gelesen worden ist, das heißt, wenn $S[i-m+1..i] = P[1..m]$ gilt. Nach der Invarianten gilt dies genau dann wenn $q^{(i)} = m$ ist. Daher legen wir fest:

$$F := \{m\}.$$

Mit diesem Automaten kann die Verarbeitung des Textes „in Echtzeit“ erfolgen, also so, dass in jedem Schritt ein Buchstabe des Textes gelesen wird. Offenbar kann man δ als $(m+1) \times |\Sigma|$-Tabelle darstellen, mit einer Zeile für jeden Zustand und einer Spalte für jeden Buchstaben in Σ. Da alle „Fremdbuchstaben“ in den Zustand $q_0 = 0$ führen, braucht man sie in der Tabelle gar nicht darzustellen. Wir zeigen nun, wie man die Tabelleneinträge in Zeit $O(m \cdot |\Sigma|)$ berechnen kann. Dieser Zeit- und Platzaufwand ist dann tolerierbar, wenn $O(m \times |\Sigma|)$ gegenüber der Länge n des Textes nicht groß ist.

Man erinnere sich, wie in Definition 5.1.3 die *Randfunktion* für ein Muster $P[1..m]$ definiert wurde:

$$f_{\mathrm{bord}}(0) = -1, f_{\mathrm{bord}}(q) = \text{Länge des längsten Randes von } P[1..q], \text{ für } 1 \leq q \leq m.$$

Berechnung der Übergangsfunktion und der Randfunktion zu $P[1..m]$:

Wir berechnen f_{bord} und δ verschränkt, durch Induktion über q, in Zeit $O(m \cdot |\Sigma|)$.

$f_{\text{bord}}(0) \leftarrow -1$.

$\delta(0, P[1]) \leftarrow 1$; $\delta(0, a) \leftarrow 0$ für $a \neq P[1]$.

$f_{\text{bord}}(1) \leftarrow 0$. // ε ist Rand von $P[1]$.

Für $q = 1, 2, \ldots, m-1$ nacheinander:
$\delta(q, P[q+1]) \leftarrow q+1$; // passender Buchstabe
$\delta(q, a) \leftarrow \delta(f_{\text{bord}}(q), a)$ für $a \neq P[q+1]$; // unpassender Buchstabe
$f_{\text{bord}}(q+1) \leftarrow \delta(f_{\text{bord}}(q), P[q+1])$.

$\delta(m, a) := \delta(f_{\text{bord}}(m), a)$ für $a \in \Sigma$.

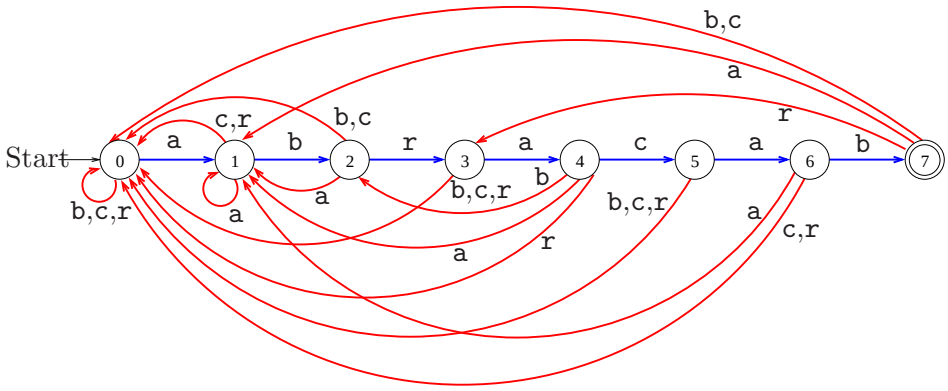


Abbildung 5.9: Der volle Präfixautomat für das Muster `abracab`, mit $m = 7$. (Die Kanten für Fremdbuchstaben führen alle zum Zustand 0; sie sind nicht dargestellt.)

Übungsaufgabe: (a) Führen Sie den Algorithmus am Beispiel $P[1..7] =$ `abracab` durch und vergleichen Sie Ihr Ergebnis mit Abb. 5.9.
(b) Beweisen Sie, dass das Verfahren δ und f_{bord} korrekt berechnet.

Bemerkung: In der Bachelorvorlesung „Automaten, Sprachen und Komplexität“ werden auch nichtdeterministische endliche Automaten (NFAs) behandelt. Man kann ganz leicht einen NFA M angeben, der genau die Wörter akzeptiert, die mit $P = P[1..m]$ enden. Die Zustandsmenge ist Q, der Startzustand ist $q_0 = 0$, die Menge der akzeptierenden Zustände ist $\{m\}$. Die Übergangsfunktion erlaubt es, mit jedem Buchstaben von Zustand 0 zu Zustand 0 zu gehen. Zudem kann man mit Buchstaben $P[q+1]$ von Zustand q in Zustand $q+1$ wechseln, für $0 \leq q < m$. (Die einzige Stelle, wo „Nichtdeterminismus“ vorkommt, ist im Zustand 0, wo man sich entscheiden kann, ob man beim Lesen von $a = P[1]$ in Zustand 0 bleibt oder nach 1 wechselt.) Wenn man auf diesen NFA die Potenzmengenkonstruktion anwendet, entsteht genau M^P_{voll}. (Dabei gibt es keine Größenexplosion wie sonst bei dieser Konstruktion üblich.)

Der KMP-Präfixautomat

Der große Nachteil des vollen Präfixautomaten M^P_{voll} ist, dass der Platzbedarf und die Zeit für den Aufbau proportional mit $m \cdot \#(\text{Buchstaben in } P)$ wächst. Dies ist allerdings unvermeidlich, wenn in einem Leseschritt für jeden Zustand und jeden Buchstaben eine eigene Aktion vorgeschrieben sein soll. Der KMP-Algorithmus benutzt einen anderen Automaten M^P_{KMP}. Die Zustandsmenge besteht aus $0, 1, \ldots, m$ sowie einem speziellen Zustand -1, dessen Rolle später genauer erklärt wird. Wir setzen also

$$Q := \{-1, 0, 1, \ldots, m-1, m\}.$$

Auch der neue Automat M^P_{KMP} verarbeitet den Text S von links nach rechts. Die „Bedeutung“ (oder der „Informationsgehalt“) der Zustände ist aber gegenüber dem vollen Präfixautomaten abgeändert. Es wird auch zugelassen, dass ein Buchstabe zwar „*inspiziert*“, aber nicht „verbraucht“ wird und damit im nächsten Schritt immer noch verfügbar ist. Hierfür verwenden wir „ε-Züge“.[8]

Im folgenden soll „Buchstabe $S[i]$ wird *gelesen*“ heißen, dass dieser Buchstabe auch „verbraucht“ wird (kein ε-Zug).

Nach wie vor soll Folgendes gelten: Wenn $S[1..i-1]$ gelesen (und verbraucht) worden ist und Buchstabe $S[i]$ als nächstes ansteht, und wenn der Zustand $q \in \{0, \ldots, m-1\}$ ist, dann ist $P[1..q]$ Suffix von $S[1..i-1]$. Daraus ergibt sich, dass es einen „richtigen“ Buchstaben gibt, der in einem solchen Zustand q erwartet wird, nämlich $P[q+1]$. Wenn $S[i]$ dieser richtige Buchstabe ist, werden wir $S[i]$ lesen und in Zustand $q+1$ wechseln. Wenn $S[i] \neq P[q+1]$ ist, gibt es einen ε-Schritt mit einer geeigneten Zustandsänderung. Dieser Ansatz ermöglicht es also, für jeden Zustand q nur noch zwei Alternativen vorzusehen und nicht mehr $|\Sigma|$ viele wie bei M^P_{voll}. Das spart Speicherplatz und Vorberechnungszeit. (Zustände -1 und m sind Sonderfälle. In Zustand m wird *immer* ein „ε-Zug“ ausgeführt, wobei der nächste Buchstabe inspiziert und ignoriert wird; in Zustand -1 wird *immer* der nächste Buchstabe gelesen und verbraucht, wobei es ebenfalls unerheblich ist, was dieser Buchstabe ist.)

Wir wollen Folgendes erreichen: In dem Moment, in dem $S[i]$ gelesen (und verbraucht) wird, geht Automat M^P_{KMP} in einen Zustand $q^{(i)} \in \{0, 1, \ldots, m\}$ über. Für diesen soll dieselbe Invariante wie im vorigen Abschnitt gelten:

($\mathbf{I_i}$): $P[1..q^{(i)}]$ ist das längste Präfix von P, das Suffix von $S[1..i]$ ist.

In diesen speziellen Schritten verhält sich der neue Automat also exakt wie M^P_{voll}. Nur hat M^P_{KMP} eventuell „Zwischenschritte“, und die Zustände, die in diesen Schritten angenommen werden, haben eine andere, speziellere Bedeutung. Wir formulieren diese „Bedeutung“, d. h., welche Information über $S[1..i]$ im aktuellen Zustand q dargestellt sein soll, wenn der nächste zu verarbeitende Buchstabe $S[i]$ ist, als Invariante:

[8] ε-Züge kamen in der Vorlesung „Automaten, Sprachen und Komplexität“ bei den Kellerautomaten vor, insbesondere bei den deterministischen Kellerautomaten.

(Inv$_{i,q}$):
Wenn sich M^P_{KMP} in Zustand q befindet und der nächste zu lesende Buchstabe $S[i]$ ist, dann gilt (s. Abb. 5.10):

> $P[1..q]$ ist Suffix von $S[1..i-1]$ **und**
> für kein $\hat{q} \in \{q+1, \ldots, m-1\}$ ist $P[1..\hat{q}+1]$ Suffix von $S[1..i]$.

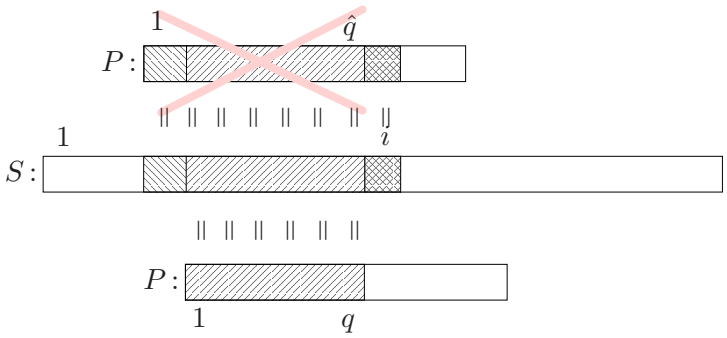


Abbildung 5.10: Illustration von (Inv$_{i,q}$): $P[1..q]$ ist Suffix von $S[1..i-1]$, und es gibt kein $\hat{q} > q$ mit der Eigenschaft, dass $P[1..\hat{q}+1]$ Suffix von $S[1..i]$ ist.

In Zustand q ist also eventuell Information über $S[i]$ gespeichert! Wie kann das sein, wenn doch $S[i]$ noch gar nicht „gelesen“ worden ist? Nun, in vorangegangenen ε-Zügen kann $S[i]$ schon „inspiziert“ und für Zustandsübergänge verwendet worden sein (sogar mehrfach)!

Die Fälle $q = m$ und $q = -1$ verdienen spezielle Aufmerksamkeit.

> Zustand $q = m$ wird nur erreicht, wenn im selben Moment ein Buchstabe $S[i]$ „gelesen“ wird. Dieser Zustand signalisiert (wegen (I$_i$)), dass $S[i-m+1..i] = P[1..m]$ gilt, dass also das Muster an Position $i-m+1$ gefunden wurde. In Zustand m gibt es nur eine einzige nächste Aktion: Ohne den nächsten Buchstaben in S anzusehen, erfolgt *immer* ein ε-Übergang zum Zustand $q' = f_{\text{bord}(m)}$, und eine Erhöhung von i um 1. Man sieht leicht, dass dann (Inv$_{i,q'}$) wieder gilt.
>
> Wenn Zustand $q = -1$ erreicht wurde, ist die erste Bedingung in (Inv$_{i,q}$) immer erfüllt (weil $P[1..-1] = \varepsilon$ Suffix von $S[1..i-1]$ ist), und es kommt nur auf die zweite Bedingung an: „Für kein $\hat{q} \in \{0, \ldots, m-1\}$ ist $P[1..\hat{q}+1]$ Suffix von $S[1..i]$.“ Das heißt, dass kein Präfix $\neq \varepsilon$ von P Suffix von $S[1..i]$ ist. In diesem Fall liest (und verbraucht) man $S[i]$, ohne es weiter zu beachten, erhöht i um 1, und setzt den Zustand q auf 0. Es ist klar, dass (Inv$_{i,0}$) erfüllt ist.

Wir können nun den Automaten M^P_{KMP} angeben. Die Zustandsmenge Q ist schon festgelegt. Explizit benannt werden nur die Buchstaben von P. In jedem Zustand

$q \in \{0, \ldots, m-1\}$ wird der Buchstabe $P[q+1]$ konkret angesprochen; die einzige Alternative ist „$\neq P[q+1]$“. Dadurch können vom Automaten auch Buchstaben in S verarbeitet werden, die in P nicht vorkommen. In Zustand -1 wird ein Buchstabe gelesen, aber ignoriert; in Zustand m erfolgt ein ε-Zug ohne Verbrauch eines Zeichens.

Als Startzustand wählen wir $q_0 = 0$. Dann ist die Invariante (Inv_{1,q_0}) erfüllt: $P[1..0] = \varepsilon$ ist Suffix von $S[1..0] = \varepsilon$ und für kein $\hat{q} \in \{1, \ldots, m-1\}$ ist $P[1..\hat{q}+1]$ Suffix von $S[1]$ (weil $P[1..\hat{q}+1]$ länger als $S[1]$ ist).

Als Menge akzeptierender Zustände wählen wir $F = \{m\}$.

Nun fehlt nur noch die Übergangsfunktion δ. Wir schreiben $\delta(q, a) = q'$, wenn es sich um einen Leseschritt handelt, der Buchstabe a also „verbraucht“ wird, und $\delta(q, \langle a \rangle) = q'$, wenn es sich um einen ε-Schritt handelt.

Nehmen wir an, der Automat ist in Zustand $q = q^{(i-1)}$, das nächste zu verarbeitende Zeichen in der Eingabe ist $a = S[i]$, und die Invariante ($\mathrm{Inv}_{i,q}$) gilt. Wir wollen den nun auszuführenden Schritt so festlegen, dass die Invariante weiter gilt. Es gibt einige Fälle.

Fall 1: $0 \leq q < m$ und $P[q+1] = a$. – Nun sollte der Automat den nächsten Buchstaben $S[i]$ *lesen* und im Zustand vermerken, dass ein längeres Präfix von P beobachtet worden ist. Wir setzen also:

$$\delta(q, P[q+1]) = q+1,$$

und damit $q^{(i)} = q+1$. Wegen des ersten Teils von ($\mathrm{Inv}_{i,q}$) gilt $P[1..q+1] = S[i-q..i]$. Wegen des zweiten Teils von ($\mathrm{Inv}_{i,q}$) gilt auch, dass für kein $\hat{q} > q$ die Gleichheit $P[1..\hat{q}+1] = S[i-\hat{q}..i]$ gilt. Das bedeutet, dass $P[1..q+1]$ das *längste* Präfix von P ist, das Suffix von $S[1..i]$ ist, also (I_i). Nun gibt es zwei Unterfälle:
(1a) Wenn $q+1 < m$ ist, dann gilt wegen (I_i) auch:

$$\text{für kein } \hat{q} \in \{q+2, \ldots, m-1\} \text{ ist } P[1..\hat{q}+1] \text{ Suffix von } S[1..i+1];$$

es gilt also die Invariante ($\mathrm{Inv}_{i+1,q+1}$), und wir können im neuen Zustand $q+1$ den nächsten Buchstaben $S[i+1]$ bearbeiten.
(1b) Wenn $q+1 = m$ ist, dann ist $P[1..m]$ Suffix von $S[1..i]$. Dies wird im Zustand registriert (mit der Festlegung $F := \{m\}$), aber es erfolgt auch sofort ein Wechsel in einen neuen Zustand, um auf die Bearbeitung von $S[i+1]$ vorzubereiten. Wir suchen das längste echte Präfix $P[1..q']$ von P, das Suffix von $S[1..i]$ ist. Wir können unsere Suche auf Präfixe von P beschränken, die kürzer als m sind und Suffixe von $S[1..i]$ sind. Das sind aber genau die Ränder von P. Hiervon betrachten wir den längsten, und setzen

$$\delta(m, \langle a \rangle) = q', \text{ mit } q' = f_{\mathrm{bord}}(m).$$

(Der nächste Buchstabe $S[i+1]$, falls er existiert, wird ignoriert.) Dann gilt die Invariante ($\mathrm{Inv}_{i+1,q'}$): $P[1..q']$ ist Suffix von $P[1..m]$, also von $S[1..i]$. Für $\hat{q} > q'$ ist

$P[1..\hat{q}]$ kein Suffix von $P[1..m]$, also auch keines von $S[1..i]$, also ist $P[1..\hat{q}+1]$ kein Suffix von $S[1..i+1]$. Daher gilt auch jetzt die Invariante ($\text{Inv}_{i+1,q'}$), und wir können im neuen Zustand q' den nächsten Buchstaben $S[i+1]$ bearbeiten.

Fall 2: $0 \leq q < m$ und $P[q+1] \neq a$. – Wegen Invariante ($\text{Inv}_{i,q}$), zweiter Teil, sind die einzigen Präfixe $P[1..q'+1]$ des Musters, die Suffix von $S[1..i]$ sein können, kürzer als $q+1$. Welche q' kommen hierfür in Frage? Es muss $P[1..q'] = S[i-q'..i-1]$, also $P[1..q'] = P[q-q'+1..q]$ gelten (das heißt: $P[1..q']$ ist Rand von $P[1..q]$) und es muss $P[q'+1] = S[i] = a$ gelten, also $P[q'+1] \neq P[q+1]$. Von allen q', die diese Bedingung erfüllen, sollten wir das größte wählen, dann gilt die Invariante ($\text{Inv}_{i,q'}$). Dieses q' hatten wir früher $f_{\text{KMP}}(q)$ genannt.
Achtung: Es könnte $f_{\text{KMP}}(q) = -1$ gelten, was bedeutet, dass es überhaupt keinen Rand $P[1..q']$ von $P[1..q]$ mit $P[q'+1] \neq P[q+1]$ gibt. Daraus folgt aber sofort, dass es kein Präfix $P[1..q'+1]$ von P gibt, das Suffix von $S[1..i]$ ist. Damit ist ($\text{Inv}_{i,-1}$) erfüllt (vgl. auch die obige Diskussion zu Zustand -1). Wir können also in jedem Fall

$$\delta(q, \langle a \rangle) := f_{\text{KMP}}(q)$$

setzen, und die Invariante ($\text{Inv}_{i,q'}$) gilt.

Fall 3: $q = -1$. – Wir haben oben schon gesehen, dass die Invariante ($\text{Inv}_{i,-1}$) impliziert, dass kein Präfix von P Suffix von $S[1..i]$ sein kann. Wir lassen den Automaten den Buchstaben $S[i]$ lesen und verbrauchen, und in Zustand 0 gehen:

$$\delta(-1, a) = 0.$$

Ist die Invariante ($\text{Inv}_{i+1,0}$) erfüllt? Tatsächlich ist $P[1..0] = \varepsilon$ Suffix von $S[1..i]$ und für kein $\hat{q} \in \{1, \ldots, m-1\}$ ist $P[1..\hat{q}+1]$ Suffix von $S[1..i+1]$ (weil $P[1..\hat{q}]$ kein Suffix von $S[1..i]$ ist). Außerdem gilt (I_i) mit $q^{(i)} = 0$, wie gewünscht, weil $P[1..0] = \varepsilon$ das längste Präfix von P ist, das Suffix von $S[1..i]$ ist.

In Abb. 5.11 ist der entsprechende Automat für das Muster $P[1..7] = \texttt{abracab}$ dargestellt. Man beachte, dass zur Herstellung des Automaten im Wesentlichen die Fehlerfunktion f_{KMP} des Musters benötigt wird.

Wir fassen zusammen: $M^P_{\text{KMP}} = (Q, \Sigma, q_0, F, \delta)$, ein deterministischer ε-Automat für das Muster $P[1..m]$, ist durch die folgenden Komponenten gegeben. Dabei kann der Automat auch Buchstaben verarbeiten, die nicht in Σ liegen (so genannte „Fremdbuchstaben“).

$Q = \{-1, 0, 1, \ldots, m\}$.

$\Sigma = \{a \mid \text{Buchstabe } a \text{ kommt in } P[1..m] \text{ vor}\}$.

$q_0 = 0$.

$F = \{m\}$.

$\delta(-1, a) = 0$, für beliebige Buchstaben a.

Für $0 \leq q < m$: $\delta(q, a) = \begin{cases} q+1 & \text{für } a = P[q+1], \\ f_{\mathrm{KMP}}(q) & \text{für beliebiges } a \neq P[q+1]. \end{cases}$

$\delta(m, \langle a \rangle) = f_{\mathrm{bord}}(m)$, für beliebige Buchstaben a.

Lemma 5.2.4. *Wann immer M^P_{KMP} in einem Zustand q ist und der nächste zu lesende Buchstabe $S[i]$ ist, gilt die Invariante* ($\mathrm{Inv}_{i,q}$).

Beweis: Wir haben M^P_{KMP} so konstruiert, dass die Invariante ($\mathrm{Inv}_{i,q}$) ständig gilt. □

Wir müssen noch zeigen, dass die Invariante stark genug ist, um zu erzwingen, dass der Automat alle Vorkommen von P erkennt.

Lemma 5.2.5. *P ist Suffix von $S[1..i]$ genau dann wenn der Automat M^P_{KMP} unmittelbar nach dem Lesen von $S[i]$ in Zustand m ist.*

Beweis: „⇒“: Sei P Suffix von $S[1..i]$. Betrachte die Situation unmittelbar bevor $S[i]$ gelesen wird. Der Automat ist in einem Zustand $q < m$. Wegen der Invarianten ($\mathrm{Inv}_{i,q}$) gilt: $P[1..q] = S[i-q..i-1]$ und für kein $\hat{q} \in \{q+1, \ldots, m-1\}$ ist $P[1..\hat{q}+1]$ Suffix von $S[1..i]$. Nun gilt aber, dass $P[1..m]$ Suffix von $S[1..i]$ ist. Dies entspricht dem Wert $\hat{q} = m-1$, und die einzige Möglichkeit dafür, dass die Invariante gilt, ist, dass $m-1 \notin \{q+1, \ldots, m-1\}$ ist. Das bedeutet, dass $q+1 > m-1$ ist, also $q = m-1$ ist. Da $S[i] = P[m]$, trifft Fall 1 zu, und der Automat wechselt in den Zustand m.

„⇐“: Nehmen wir an, der Automat geht beim Lesen von $S[i]$ in Zustand m. Das kann nur passieren, wenn Fall 1 zutrifft, der vorige Zustand $m-1$ war und der gelesene Buchstabe $S[i]$ gleich $P[m]$ ist. Nach der Invarianten ($\mathrm{Inv}_{i,m-1}$) gilt auch $S[i-m+1..i-1] = P[1..m-1]$. Damit ist P Suffix von $S[1..i]$. □

In Abb. 5.12 ist die Arbeitsweise des Automaten an einem Beispiel angegeben. Wir verfolgen die Zustände des Automaten anhand der Präfixe selbst. Von Zeile zu Zeile wächst das Suffix $P[1..q]$ entweder um einen Buchstaben an oder es wird verkürzt, manchmal auf Länge 0, manchmal auf eine Länge > 0. Im Beispiel tritt die Bedingung von Lemma 5.2.5 für $i = 12$ ein.

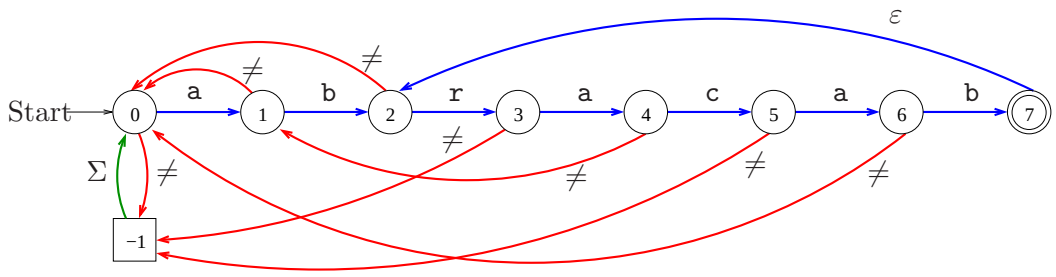


Abbildung 5.11: Der KMP-Präfixautomat für das Muster `abracab`, mit $m = 7$. Die Zustandsmenge ist $\{-1, 0, \ldots, m\}$. Dabei entsprechen die Zustände $0, \ldots, m$ den $m+1 = 8$ Präfixen $P[1..q]$ des Musters, Zustand -1 spielt eine Sonderrolle. Man beginnt in Zustand 0. Der Text $S[1..n]$ wird Buchstabe für Buchstabe gelesen. Die blauen Zustandsübergänge bestimmen, was bei erfolgreichen Vergleichen passiert. Von Zustand $q \in \{0, \ldots, m-1\}$ gelangt man unter Lesen des Buchstabens $P[q+1]$ in den Zustand $q+1$, von Zustand $q = m-1$ gelangt man unter Lesen des Buchstabens $P[m]$ zunächst in den akzeptierenden Zustand m (Ausgabe „Muster gefunden“) und dann, ohne einen Buchstaben anzusehen, in den Zustand $f_{\mathrm{KMP}}(m)$ (hier: 2). Die roten Zustandsübergänge („*mismatch*“) bestimmen das Verhalten, wenn in Zustand $q \in \{0, \ldots, m-1\}$ ein Buchstabe $\neq P[q+1]$ gesehen wird. Ohne diesen Buchstaben zu verbrauchen, also mit einem ε-Übergang, wird in den Zustand $f_{\mathrm{KMP}}(q)$ gegangen. Im Zustand -1 wird der nächste Buchstabe im Text gelesen und es wird in den Zustand 0 gewechselt (grüner Übergang).

Programmtechnische Umsetzung: Der Algorithmus verläuft in Runden $i = 1, 2, \ldots, n$. In Runde i wird Buchstabe $S[i]$ des Textes eventuell mehrmals inspiziert (Fall 2) und schließlich gelesen (Fall 1). Wenn bei Betrachten von $S[i]$ das Muster gefunden wird und man Zustand m betritt, wird die Aktion aus Fall 3, also der ε-Schritt zu Zustand $f_{\mathrm{KMP}}(m)$, noch im gleichen Schleifendurchlauf ausgeführt. Die Runden werden über eine (äußere) **for**-Schleife realisiert. Der Inhalt der Schleifenvariablen `i` gibt dabei immer die aktuelle Runde an. Der aktuelle Zustand q wird in einer Variablen `q` gehalten. Die ε-Übergänge im *mismatch*-Fall (Fall 2) finden in einer inneren **while**-Schleife statt. Wenn diese Schleife in den Zustand $q = -1$ führt, wird die in Fall 3 vorgesehene Erhöhung von `q` auf 0 ausgeführt.

Man beachte (und staune über) die extrem kompakte Umsetzung als Programm.

Algorithmus 5.2.6 (Knuth-Morris-Pratt)**.**

KMP-Textsuche(`P[1..`m`]`,`S[1..`n`]`)

Eingabe: `P[1..`m`]`, `S[1..`n`]` // Muster, Text

Ausgabe: `A` $\subseteq \{1, \ldots, n-m+1\}$;

Vorberechnen: `f_KMP[0..`m`]`: KMP-Fehlerfunktion von `P[1..`m`]` als Tabelle;

(1) `A` $\leftarrow \emptyset$;

(2) `q` $\leftarrow 0$;

Schritt \ Text		a	b	c	a	b	a	b	r	a	c	a	b	r	a	b	a	b	r
i		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	ε																		
0→1		a																	
1→2		a	b																
$2\xrightarrow{\varepsilon}0$			ε																
$0\to-1\xrightarrow{\varepsilon}0$				ε															
0→1					a														
1→2					a	b													
$2\xrightarrow{\varepsilon}0$						ε													
0→1							a												
1→2							a	b											
2→3							a	b	r										
3→4							a	b	r	a									
4→5							a	b	r	a	c								
5→6							a	b	r	a	c	a							
6→7							a	b	r	a	c	a	b						
$7\xrightarrow{\varepsilon}2$												a	b						
2→3												a	b	r					
3→4												a	b	r	a				
$4\xrightarrow{\varepsilon}1$															a				
1→2															a	b			
$2\xrightarrow{\varepsilon}0$																ε			
0→1																	a		
1→2																	a	b	
2→3																	a	b	r

Abbildung 5.12: *Beispiel* für die Arbeitsweise des KMP-Präfixautomaten.
Muster: $P[1..7]$ = `abracab`; Text: $S[1..18]$ = `abcababracabrababr`.

```
(3)   for i from 1 to n do
(4)     while q ≥ 0 ∧ P[q+1] ≠ S[i] do q ← f_KMP[q];
(5)     q ← q+1;
(6)     if q = m
(7)       then A ← A ∪ {i-m+1}; q ← f_KMP[m];
(8)   return A.
```

Wo findet man die Runden und die Fälle im Programm? Die Initialisierung, Runde 0, besteht aus der Zuweisung $\mathtt{q} \leftarrow 0$, und $q_0 = 0$ (Zeile (2)). Der Zustand q steht stets in der Variablen `q`. Runde i ist der Durchlauf der **for**-Schleife, in dem `i` die Zahl i enthält. Hier wird Buchstabe $S[i]$ behandelt. Man geht davon aus, dass $q < m$ gilt.

Text		c	a	b	a	b	r	a	d	a	c	a	r
i	0	1	2	3	4	5	6	7	8	9	10	11	12
$q_7 = 4$					a	b	r	a	c	...			
$q'_1 = 1$								a	b	...			
$q'_2 = 0$									a	...			
$q'_3 = -1$										a	...		

Abbildung 5.13: Beispiel für wiederholte ε-Übergänge im *mismatch*-Fall: Wir suchen im Text `cababradacar` nach dem Muster `abraca` (für die KMP-Fehlerfunktion dieses Musters s. Abb. 5.8). Es ist $q^{(7)} = 4$. Nach einigen ε-Zügen durch Zustände q'_1, q'_2, q'_3 findet man $q^{(8)} = q'_3 + 1 = 0$.

Die Bedingung der **while**-Schleife ist der Test darauf, ob Fall 2 eintritt. Solange dies so ist, wird q durch $f_{\mathrm{KMP}}[q]$ ersetzt. Wenn die Schleife endet, ist einer von zwei Fällen eingetreten.

(i) $q = -1$: Dies ist Fall 3. Die Zuweisung in Zeile (4) stellt den Zustand $q = 0$ ein. Der Test in Zeile (6) ergibt *false*. Man geht zur nächsten Runde über.

(ii) $q \geq 0$ und $P[q+1] = S[i]$: Dies ist Fall 1. Zeile (4) bewirkt die nötige Erhöhung von `q` um 1. Die Aktionen für Fall 1b werden unmittelbar angeschlossen (Zeilen (6)-(7)). Wenn $q = m$ gilt, wurde das Muster gefunden, und man kann den Startindex $i - m + 1$ in der Menge A vermerken. Der ε-Übergang von Fall 1b wird ebenfalls in Zeile (7) durchgeführt.

Wir fassen die Eigenschaften des Algorithmus von Knuth, Morris und Pratt zusammen.

Satz 5.2.7. (a) *Algorithmus 5.2.6 liefert in* `A` *alle Positionen* ℓ *in* $S[1..n]$ *zurück, an denen das Muster* $P[1..m]$ *vorkommt.*

(b) *Die Rechenzeit von Algorithmus 5.2.6 ist* $O(n)$*. Es werden nicht mehr als* $2n$ *Buchstabenvergleiche durchgeführt.*

Beweis: Die Korrektheit folgt aus der Vorüberlegung, insbesondere Lemma 5.2.5. Für die Anzahl der Vergleiche betrachten wir die Zahlen i und q in `i` und `q`. Für jeden erfolgreichen Vergleich mit Ergebnis $P[q+1] = S[i]$ steigt i um 1 (man geht zur nächsten Runde); dies kann nicht öfter als n-mal geschehen. Für jeden erfolglosen Vergleich (Ergebnis $P[q+1] \neq S[i]$) steigt die Zahl $i - q$ strikt an (man ersetzt q durch $f_{\mathrm{KMP}}(q) < q$); diese Zahl startet mit dem Wert 1, sie sinkt nie und sie kann nicht größer als $n + 1$ werden. Also gibt es nicht mehr als n erfolglose Vergleiche. □

Wenn man die beiden Versionen 5.2.2 und 5.2.6 des KMP-Algorithmus vergleicht, erkennt man, dass in beiden genau die gleichen Buchstabenvergleiche stattfinden, es sich also tatsächlich im Wesentlichen um denselben Algorithmus handelt. Der erste

orientiert sich, wie der naive Algorithmus, an einer Position ℓ des Musters im Text. Dieses ℓ wird in 5.2.2 in der äußeren Schleife mitgeführt und in Sprüngen erhöht. Version 5.2.6 löst sich von dieser Vorstellung. In der äußeren Schleife geht es um die Verarbeitung von Buchstaben von S, mit dem kontinuierlich laufenden Index i, wie bei Automaten üblich. Nur in der inneren Schleife ist mit der sprungweisen Verringerung von q noch eine Ahnung vom Verschieben des Musters nach rechts übrig.

5.2.4 Die Berechnung der Randfunktion und der KMP-Fehlerfunktion

Es bleibt noch zu zeigen, wie man zu einem gegebenen Muster $P[1..m]$ in Zeit $O(m)$ eine Tabelle für die KMP-Fehlerfunktion f_{KMP} (Definition 5.2.1) berechnen kann. Es ist nützlich, hierfür zunächst die Berechnung der Randfunktion f_{bord} (Definition 5.1.3) zu betrachten.

Nach Definition ist $f_{\text{bord}}(0) = -1$ und $f_{\text{bord}}(i) =$ die Länge des längsten Randes von $P[1..i]$, für $0 \leq i \leq m$. Die Grundidee für die Berechnung von f_{bord} ist, den KMP-Automaten mit $P[1..m]$ als Text ablaufen zu lassen. Das ist natürlich ziemlich langweilig, weil man nur feststellt, dass das Muster genau einmal vorkommt. Dieses „triviale“ Vorkommen können wir abschalten, indem wir $P[1]$ im Text durch einen „Fremdbuchstaben“ $* \notin \Sigma$ ersetzen. Wir lassen also den KMP-Automaten auf Eingabe $P^*[1..m] = * \circ P[2..m]$ ablaufen.

Wir betrachten die angepasste Version der Invarianten (I_i) aus dem vorigen Abschnitt. Für $1 \leq i \leq m$ gilt:

(I_i^*) Nach dem Lesen von $P^*[i]$ ist der KMP-Automat in Zustand $q^{(i)}$, wobei $P[1..q^{(i)}]$ das längste Präfix von P ist, das Suffix von $P^*[1..i]$ ist.

Das längste Präfix von P, das Suffix von $P^*[1..i] = * \circ P[2..i]$ ist, ist aber genau der längste Rand von $P[1..i]$! Unsere Invariante verkürzt sich zu:

(I_i^*) Nach dem Lesen von $P^*[i]$ ist der KMP-Automat in Zustand $f_{\text{bord}}(i)$.

Aha! Wenn wir f_{KMP} haben, können wir den KMP-Automaten ablaufen lassen und die Werte der Randfunktion ablesen.

Wie schon in Abschnitt 5.2.2 (Seite 12, Übungsaufgabe) bemerkt, können wir dann leicht die Fehlerfunktion f_{KMP} berechnen, wie folgt:

(i) $f_{\text{KMP}}(0) \leftarrow -1$.

(ii) Für $i = 1, \ldots, m-1$ tue nacheinander:

$q' \leftarrow f_{\text{bord}}(i)$ // ein Wert in $\{0, \ldots, i-1\}$.

$$f_{\text{KMP}}(i) \leftarrow \begin{cases} q' & \text{, wenn } P[i+1] \neq P[q'+1], \\ f_{\text{KMP}}(q') & \text{, wenn } P[i+1] = P[q'+1]. \end{cases}$$

(iii) $f_{\mathrm{KMP}}(m) \leftarrow f_{\mathrm{bord}}(m)$.

Es ist leicht zu sehen, dass man mit dieser Methode in Kombination mit einem Verfahren, das nacheinander $f_{\mathrm{bord}}(0), f_{\mathrm{bord}}(1), \ldots$ liefert, unmittelbar nach der Berechnung von $f_{\mathrm{bord}}(i)$ auch $f_{\mathrm{KMP}}(i)$ ausrechnen kann.

Leider ist das ganze Verfahren zirkulär, weil man zur Berechnung von f_{KMP} eine Tabelle von f_{KMP} benötigt. Oder nicht? Man beobachtet, dass bei der Bearbeitung des Buchstabens $P^*[i]$ in der Eingabe (ε-Schritte oder Leseschritt) der aktuelle Zustand q kleiner als i sein muss. (Nach dem Lesen von $P^*[i-1]$ ist der Zustand $q = f_{\mathrm{bord}}(i-1) < i-1$. Während ε-Schritten auf $P^*[i]$ wird q höchstens kleiner; beim Lesen von $P^*[i]$ kann q höchstens um 1 wachsen, ist also immer noch $< i$.) Daher können wir den KMP-Automaten *gleichzeitig aufbauen und benutzen*, da in jedem Schritt der benötigte Teil von f_{KMP} schon bekannt ist.

Der Ablauf aus Automatensicht, der Arrays `f_bord`$[0..m]$ und `f_KMP`$[0..m]$ beschriftet und liest, ist wie folgt.

`f_bord`$[0] \leftarrow -1$; `f_KMP`$[0] \leftarrow -1$;

Startzustand: $q \leftarrow 0$.

ε-Schritt mit $P^*[1] = *$ führt in Zustand $q = -1$.

Lese $P^*[1]$, gehe in Zustand $q = q^{(1)} = 0$.

`f_bord`$[1] \leftarrow 0$; `f_KMP`$[1] \leftarrow \begin{cases} 0 & \text{, wenn } P[2] \neq P[1], \\ -1 & \text{, wenn } P[2] = P[1]. \end{cases}$.

Setze $i \leftarrow 2$.

Wiederhole, bis `f_bord`$[m]$ und `f_KMP`$[m]$ berechnet sind:

Fall 1: $q \geq 0$ und $P[i] = P[q+1]$:

erhöhe q um 1; `f_bord`$[i] \leftarrow q$;

`f_KMP`$[i] \leftarrow \left\{ \begin{array}{ll} q & \text{, wenn } i = m \text{ oder } P[i+1] \neq P[q+1], \\ \texttt{f\_KMP[}q\texttt{]} & \text{, sonst.} \end{array} \right\}$;

erhöhe i um 1;

Fall 2: $q \geq 0$ und $P[i] \neq P[q+1]$: Setze q auf `f_KMP`$[q]$.

Fall 3: $q = -1$:

setze q auf 0; `f_bord`$[i] \leftarrow q$;

`f_KMP`$[i] \leftarrow \left\{ \begin{array}{ll} 0 & \text{, wenn } i = m \text{ oder } P[i+1] \neq P[1], \\ -1 & \text{, sonst.} \end{array} \right\}$;

erhöhe i um 1;

Fall (1b) aus dem KMP-Algorithmus („Muster P in Text P^* gefunden“) kann nicht vorkommen.

Aus der Vorüberlegung folgt, dass dieser Automat die Tabellen `f_bord[`$0..m$`]` und `f_KMP[`$0..m$`]` korrekt berechnet.

Wir können den Ablauf genau wie beim KMP-Algorithmus ganz leicht in ein Programm umsetzen. Der Buchstabe $P^*[1] = *$ muss natürlich nicht benannt werden; es genügt, den Automaten anfangs in den Zustand -1 zu setzen, damit das Zeichen $P[1]$ als „Fremdzeichen“ gelesen wird. Eine Sonderbehandlung für $i = 1$ ist dann nicht nötig. Die Aktionen für Fall (1b) fallen weg; nach einem erfolgreichen Vergleich „$P[i] = P[q+1]$“ wird in beiden Arrays ein neuer Eintrag berechnet. Fälle 1 und 3 können identisch behandelt werden, genau wie im KMP-Algorithmus.

Algorithmus 5.2.8 (Berechnung der Randfunktion und der KMP-Fehlerfunktion)**.**
KMP-Rand-Preprocessing(`P[`$1..m$`]`)
Eingabe: `P[`$1..m$`]`: Muster;
Ausgabe: `f_bord[`$0..m$`]`: Randfunktion als Tabelle;
`f_KMP[`$0..m$`]`: KMP-Fehlerfunktion als Tabelle;

```
(1)   f_bord[0] ← −1; f_KMP[0] ← −1;
(2)   q ← −1;
(3)   for i from 1 to m do
(4)     while q ≥ 0 ∧ P[q+1] ≠ P[i] do q ← f_KMP[q];
(5)     q ← q+1;
(6)     f_bord[i] ← q;
(7)     if i = m or P[q+1] ≠ P[i+1]
(8)       then f_KMP[i] ← q
(9)       else f_KMP[i] ← f_KMP[q];
(10)  return f_bord[0..m], f_KMP[0..m].
```

Ein Beispielergebnis ist in Abb. 5.14 dargestellt. Man prüfe den Ablauf des Algorithmus anhand dieses Beispiels und vergewissere sich, dass nach dem Lesen von `P[`i`]` und Erhöhen von `q` stets die Invariante (I_i^*) gilt.

Satz 5.2.9. *Algorithmus 5.2.8 berechnet f_{bord} und f_{KMP} korrekt und in Zeit $O(m)$.*

Beweis: Wie beim KMP-Algorithmus selbst überprüft man leicht, dass der Algorithmus nichts anderes tut als den KMP-Automaten auf $P^*[1..m]$ ablaufen zu lassen. Dabei werden die Werte $q^{(i)}$ ausgelesen und als $f_{\mathrm{bord}}(i)$ eingetragen (Zeile (6)); weiter wird $f_{\mathrm{KMP}}(i)$ ausgerechnet und eingetragen (Zeilen (7)–(9)). Die Rechenzeitanalyse ist identisch zu der des KMP-Algorithmus selbst. □

Bemerkung: Man kann sich überlegen, dass der KMP-Algorithmus auch dann korrekt abläuft, wenn man anstelle der KMP-Fehlerfunktion die Randfunktion benutzt. Das führt zu einer Abwandlung von Algorithmus 5.2.8, in der nur die Randfunktion berechnet wird. Alternativ kann man in Algorithmus 5.2.8 auf die explizite Darstellung

Abbildung 5.14: Randfunktion f_{bord} und KMP-Fehlerfunktion f_{KMP} bei $P[1..12] =$ `abababcababa`:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
`P[`i`]`		a	b	a	b	a	b	c	a	b	a	b	a
$f_{\text{bord}}(i)$	−1	0	0	1	2	3	4	0	1	2	3	4	5
$f_{\text{KMP}}(i)$	−1	0	−1	0	−1	0	4	−1	0	−1	0	−1	5

und Ausgabe der Randfunktion verzichten und nur die KMP-Fehlerfunktion berechnen. Dies führt zu folgendem wiederum sehr kompakten Algorithmus, der in linearer Zeit die KMP-Fehlerfunktion berechnet:

Algorithmus 5.2.10 (Berechnung der KMP-Fehlerfunktion)**.**
KMP-Preprocessing(`P[`$1..m$`]`)
Eingabe: `P[`$1..m$`]`: Muster;
Ausgabe: `f[`$0..m$`]`: KMP-Fehlerfunktion als Tabelle;

```
(1)  f[0] ← −1;
(2)  q ← −1;
(3)  for i from 1 to m do
(4)    while q ≥ 0 ∧ P[q+1] ≠ P[i] do q ← f[q];
(5)    q ← q+1;
(6)    if i = m or P[q+1] ≠ P[i+1]
(7)      then f[i] ← q
(8)      else f[i] ← f[q];
(9)  return f[0..m].
```

Kapitel 5

Textalgorithmen

5.3 Der Algorithmus von Aho und Corasick

Der Zweck des hier vorgestellten Algorithmus ist, in einem Text $S = S[1..n]$ über einem Alphabet Σ nach Mustern aus einer endlichen Menge $\Pi = \{P_1, \ldots, P_r\} \subseteq \Sigma^+$ zu suchen, und alle Vorkommen zu melden. Der Algorithmus wurde 1975 von Alfred V. Aho und Margaret J. Corasick vorgestellt[1].

1. Schritt: Preprocessing/Vorverarbeitung von Π.
Die Rechenzeit ist O(Gesamtlänge aller Suchmuster).

2. Schritt: Durchlauf durch S. Der Algorithmus gibt für jede Stelle i und jedes Muster P_t das Paar (t, i) aus, falls $S[i..i + |P_t| - 1] = P_t$.
Die Rechenzeit ist $O(n +$ Länge der Ausgabe$)$, unabhängig von Π.

(Die Rechenzeit könnte länger als $O(n)$ sein, wenn die Ausgabe umfangreicher als n ist. Dies kann durchaus passieren, wenn viele Muster an der gleichen Stelle vorkommen, zum Beispiel wenn $\Pi = \{a, a^2, a^3, a^4, \ldots, a^r\}$ und $S = a \ldots a$ ist. Wir nehmen an, dass $\varepsilon \notin \Pi$ ist, da das Suchwort ε trivialerweise an jeder Stelle des Textes S vorkommt.)

Wir diskutieren den Algorithmus anhand eines konzeptuellen Baums $T_{\mathrm{Pref}(\Pi)}$, der als

[1] Alfred V. Aho, Margaret J. Corasick, Efficient String Matching: An Aid to Bibliographic Search. Commun. ACM 18(6): 333-340 (1975), `https://doi.org/10.1145/360825.360855`.

Knoten die Menge aller Präfixe

$$\text{Pref}(\Pi) := \{w \mid \exists t \in \{1, \ldots, r\} \colon w \text{ ist Präfix von } P_t\}$$

von Mustern in Π hat. Knoten $w \in \text{Pref}(\Pi)$ hat die Knoten wa als Kinder, für die $wa \in \text{Pref}(\Pi)$ ist. Die Knoten, die in Π sind (Blätter oder nicht) werden markiert. Es ist klar, dass $\text{Pref}(\Pi)$ maximal $1 + |P_1| + \cdots + |P_r|$ Elemente hat. Mit $\text{length}(\Pi)$ bezeichnen wir $|P_1| + \cdots + |P_r|$, die Gesamtlänge aller Muster in Π.

Beispiel: $\Pi = \{P_1, \ldots, P_7\} = \{\text{bei}, \text{beide}, \text{beine}, \text{eis}, \text{eid}, \text{ein}, \text{nein}\}$. Der entsprechende Baum ist in Abb. 5.1 dargestellt.

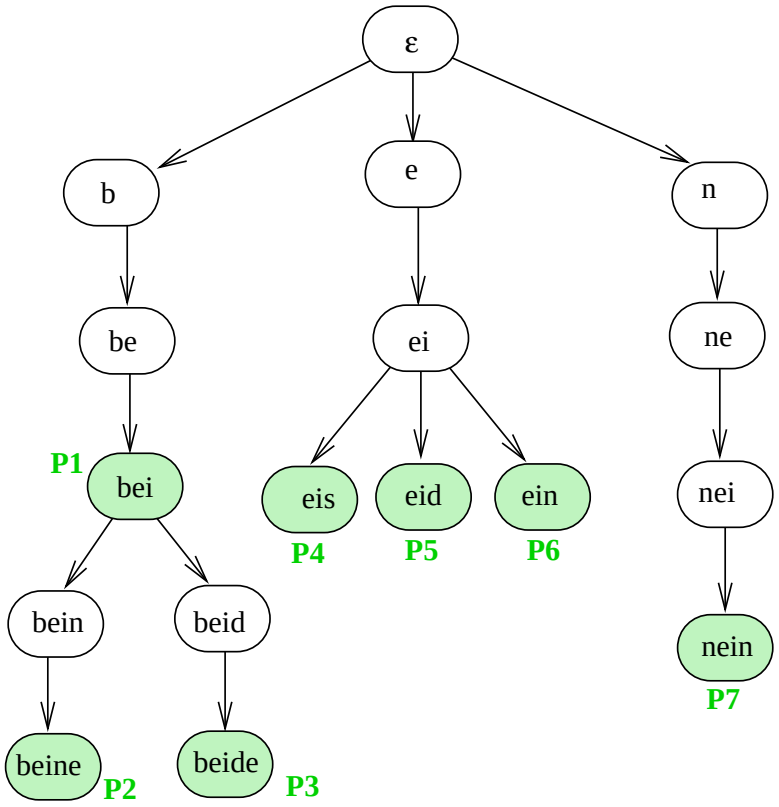


Abbildung 5.1: Baum der Präfixe für eine Menge Π

Wir verändern diesen Baum etwas, ohne dass Information verlorengeht: Wir schreiben den Buchstaben a an die Kante von w nach wa. Die ursprüngliche Beschriftung eines

Knotens ist dann das Wort, das an den Kanten von der Wurzel zu diesem Knoten abzulesen ist. Wir nummerieren die Knoten (im Wesentlichen beliebig, zum Beispiel in Präorder-Reihenfolge) durch, die Wurzel erhält dabei die Nummer 0. Die Markierung der Knoten, die Wörtern aus Π entsprechen, wird beibehalten. Zudem gibt es einen künstlichen Knoten mit Nummer -1, der außerhalb des Baums sitzt. Die Menge $\{0, 1, \ldots, |\mathrm{Pref}(\Pi)|-1\}$ der Baumknoten heißt $V_\Pi = V$, der Knoten -1 kommt hinzu. Das Wort $w \in \mathrm{Pref}(\Pi)$, das dem Knoten v entspricht, wollen wir $w(v)$ nennen. Ein Beispiel ist in Abb. 5.2 angegeben.

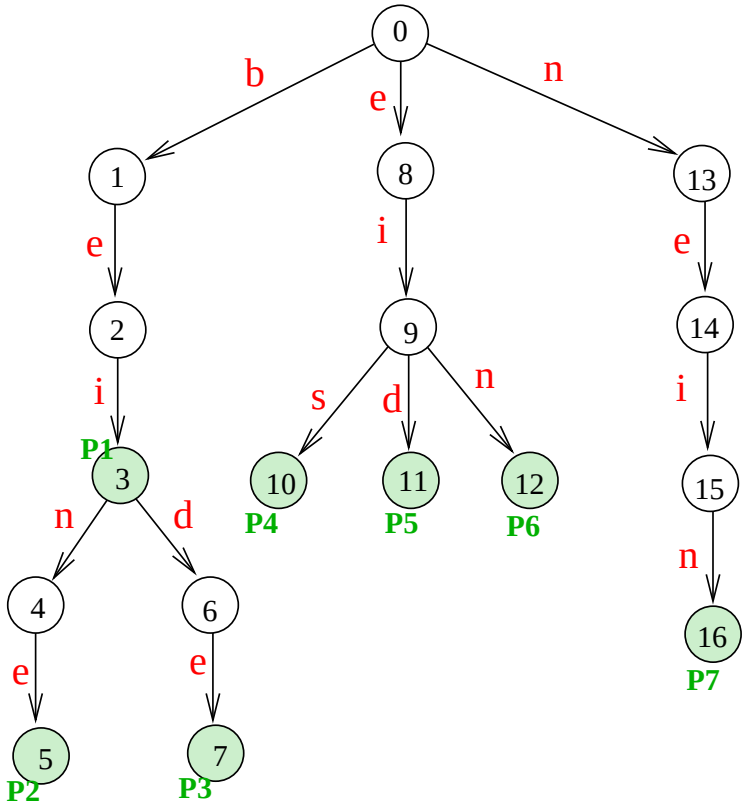


Abbildung 5.2: Baum (Trie) T_Π. Die Muster stehen an den Wegen. Es gilt beispielsweise $w(0) = \varepsilon$, $w(4) =$ bein, $w(9) =$ ei, usw.

Wir nehmen an, dass dieser Baum T_Π vorliegt, und dass man in ihm in konstanter Zeit pro Schritt wie folgt navigieren kann: (i) zu v seinen Vorgänger $p(v)$ und den Buchstaben a an der Kante $(p(v), v)$ finden; (ii) zu Knoten v und Buchstabe a den

a-Nachfolger von v finden, falls er existiert. (Man benötigt hier eigentlich eine Datenstruktur „Trie“, deren Konstruktion eigenen Aufwand erfordert. Beschrieben wird sie in vielen Büchern zu Datenstrukturen, beispielsweise im Buch von Ottmann und Widmayer.)

Der Platzbedarf für T_Π ist $O(|V|) = O(|\text{Pref}(\Pi)|) = O(\text{length}(|\Pi|)$. Wir können uns vorstellen, dass wir zwischen Knoten in T_Π Zeiger setzen können. Dies wird einfach durch die Angabe von Knotennummern realisiert.

Diesen Baum wollen wir so präparieren, dass ein Textsuchalgorithmus im Stil von KMP (mit f_{bord} anstelle von f_{KMP}) möglich ist. Dabei soll der Platzbedarf $O(\text{length}(\Pi))$ bleiben, ganz gleich wie groß Σ ist.

Anmerkung: Alternativ kann man aus T_Π einen endlichen Automaten mit Zustandsmenge V bauen, der für jedes $v \in V$ und jeden Buchstaben $a \in \Sigma$ den Nachfolgezustand $\delta(v, a)$ angibt. Die Tabelle für δ erfordert Platz $O(\text{length}(\Pi) \cdot |\Sigma|)$, was für kleine Alphabete eventuell akzeptabel, für große Alphabete normalerweise nicht erwünscht ist.

Wir definieren eine Fehlerfunktion $f\colon V \to V \cup \{-1\}$. In der abstrakten Beschreibung des Baumes, in der die Knoten Präfixe sind, soll die Fehlerfunktion zu w das längste *echte* Suffix von w finden, das in Pref(Π) ist. *Beispiele*: f(nei) „=“ ei, f(eid) „=“ ε, f(beine) „=“ ne, usw. (Natürlich ist $f(\varepsilon)$ „=“ -1.) Wir wollen aber f in der T_Π-Version des Baumes definieren, bei der die Knoten durch Nummern gegeben sind.

Was wir noch brauchen, ist für jeden Knoten $v \in V$ die Information, ob $w(v)$ ein echtes Suffix hat, das in Π ist. (Ob $w(v)$ selbst in Π ist, steht schon an dem Knoten.) Diese Information wird gegebenenfalls mit einem Wert $g(v) = g_\Pi(v) \in \{1, \ldots, |V|-1\}$ angegeben, der auf das längste echte Suffix von $w(v)$ zeigt, *das in* Π *ist*, falls so etwas existiert. Falls $w(v)$ kein echtes Suffix in Π hat, setzen wir $g_\Pi(v) = 0$. Beispielsweise ist $w(4) =$ bein, und dieses Wort hat $w(12) =$ ein als Suffix. Es wird also $g_\Pi(4) = 12$ sein.

Fehlerfunktion f und „gefunden“-Funktion g werden in einer Vorverarbeitungsprozedur berechnet, die der Berechnung der Randfunktion f_{bord} ähnelt.

Algorithmus 5.3.1 (**AC: Berechnung der Fehlerfunktion**).
AC-Preprocessing(Π)
Eingabe: Menge Π von Mustern, gegeben als T_Π;
Wenn $w(v) = P_t$ für ein t, dann ist v mit „P_t“ markiert;
Ausgabe: `f[0..`$|V|-1$`]` und `g[0..`$|V|-1$`]` als Tabellen;
(1) `g[0]` $\leftarrow 0$;
(2) `f[0]` $\leftarrow -1$;
(3) **for** jedes Kind v der Wurzel **do** `f[`v`]` $\leftarrow 0$; `g[`v`]` $\leftarrow 0$;
(4) **for** die restlichen Knoten v in Levelorder (ebenenweise!) **do**
(5) $u \leftarrow p(v)$; // Vorgänger; wir wissen: $u \neq 0$
(6) $a \leftarrow$ Buchstabe an Kante (u, v);
(7) $z \leftarrow$ `f[`u`]`;
(8) **while** $z \neq -1 \wedge z$ hat kein a-Kind **do** $z \leftarrow$ `f[`z`]`;
(9) **if** $z = -1$ **then** `f[`v`]` $\leftarrow 0$ **else** `f[`v`]` $\leftarrow$ das a-Kind von z;
(10) **if** `f[`v`]` ist mit „P_t“ markiert **then** `g[`v`]` $\leftarrow$ `f[`v`]` **else** `g[`v`]` $\leftarrow$ `g[f[`v`]]`.

Was passiert in diesem Algorithmus? Wir ignorieren zunächst die Funktion g. Zeilen (1) und (2) setzen `f[`v`]` auf den richtigen Wert für die Knoten, die zu ε und zu den einbuchstabigen Präfixen gehören. Die levelweise Verarbeitung der anderen Knoten stellt sicher, dass bei Bearbeitung des Knotens v die Knoten für alle Wörter, die kürzer als $w(v)$ sind, schon bearbeitet sind. Wir bearbeiten v. Zeilen (5)–(7) identifizieren den Vorgänger u (der nicht die Wurzel ist) und den Buchstaben a an der Kante von u nach v. Die **while**-Schleife sucht das längste echte Suffix von $w(u)$, so dass der zugehörige Knoten einen a-Nachfolger hat. (Man kann sich leicht überlegen, dass der Knoten für das längste echte Suffix von $w(v) = w(u)a$ genau der a-Nachfolger dieses Knotens sein muss.) Um dies durchzuführen, benutzt die Schleife die Funktion f auf Knoten weiter oben im Baum, wo sie (nach Induktionsvoraussetzung) schon korrekt berechnet ist. Wenn die **while**-Schleife mit $z = -1$ endet, dann gibt es kein echtes Suffix von $w(u)$ in Pref(Π), das einen a-Nachfolger hat, also hat $w(v)$ kein echtes Suffix in Pref(Π), und es ist $f(v) = 0$. Wenn ein passendes z gefunden wird, dann ist $f(v)$ der a-Nachfolger dieses Knotens.

Noch ein Wort zur Berechnung der g-Funktion (folgend einem Hinweis eines Studierenden): Wir wollen alle Suffixe von $w(v)$ finden, die in Π liegen. (Man beachte, dass dies eine andere Anforderung ist als das längste Suffix in Pref(Π).) Wenn v mit P_t markiert ist, ist dies für v der Fall. Es kann aber auch sein, dass ein *echtes* Suffix w' von $w(v)$ zu Π gehört. Weil $w(f(v))$ das längste Suffix von $w(v)$ in Π *überhaupt* ist, muss dann w' (echtes oder unechtes) Suffix von $w(f(v))$ sein. Wir können annehmen

(nach Induktionsvoraussetzung), dass Knoten $z = f(v)$ schon die richtige Information hat: Dieser Knoten könnte selbst mit einem P_t markiert sein (dann lassen wir $g(v)$ darauf verweisen) oder mit $g(z) \neq 0$ zu dem längsten Suffix von $w(f(v))$ führen, *das in* Π *ist*. – Man kann sich nun überlegen, dass für jeden Knoten v das iterierte Verfolgen der g-Verweise (v, $g(v)$, $g(g(v))$, ..., bis 0 erreicht ist) zu den Knoten für alle Suffixe von $w(v)$ führt, die in Π liegen. Bei jedem Iterationsschritt wird ein neues solches Suffix gefunden. Der Zeitaufwand für das Durchlaufen dieser Kette ist linear in der Anzahl der betreffenden Suffixe.

Satz 5.3.2. *Algorithmus 5.3.1 hat Zeitbedarf* $O(\mathit{length}(\Pi))$. *Er berechnet korrekt die Fehlerfunktion* f *auf den Knoten des Baum* T_Π *und die „gefunden"-Funktion* g.

Beweis: Der Beweis der *Korrektheit* ist praktisch der gleiche wie bei der Berechnung der Randfunktion (s. Übung). Man benutzt induktiv, dass die $f(v')$-Werte für v' in Levels oberhalb von v schon korrekt berechnet sind, und führt eine Unterinduktion durch, die die folgende Invariante benutzt:

$w(z)$ ist echtes Suffix von $w(u)$; es gibt kein längeres echtes Suffix w' von $w(u)$ derart dass $w'a \in \mathrm{Pref}(\Pi)$ ist.

Laufzeitanalyse: Sei P_t eines der Muster, die zu einem Blatt gehören. Man betrachtet den Zeitaufwand für die Knoten v auf dem Weg im Baum, der zu diesem Muster gehört. Hierzu wendet man die Technik aus der Zeitanalyse des KMP-Algorithmus an, auch wenn es keine explizite Variable `q` gibt. Wir benutzen einen gedachten Zähler q, der anfangs auf 0 steht. Wenn der nächste Knoten v auf dem Weg bearbeitet wird, erhöhen wir q um 1. Wenn in der **while**-Schleife in Zeile (8) die Operation $z \leftarrow \mathtt{f[}z\mathtt{]}$ ausgeführt wird, verringern wir q um die Differenz der beiden Werte $|w(z)|$ und $|w(\mathtt{f[}z\mathtt{]})|$, also mindestens um 1. Man sieht dann leicht, dass stets $q \geq |w(f(v))|$ ist, für den Knoten, dessen Bearbeitung eben abgeschlossen wurde. Das heißt: Am Ende ist $q \geq 0$. Genau wie in der KMP-Analyse folgt daraus, dass die Anzahl der Durchläufe durch die **while**-Schleife bei der Bearbeitung der Knoten auf dem P_t-Weg nicht größer als $|P_t|$ ist. Die gesamte Anzahl der Durchläufe ist also nicht größer als $|P_1| + \cdots + |P_t| = \mathrm{length}(\Pi)$. □

Wir führen den Algorithmus am in Abb. 5.2 gegebenen Baum durch.

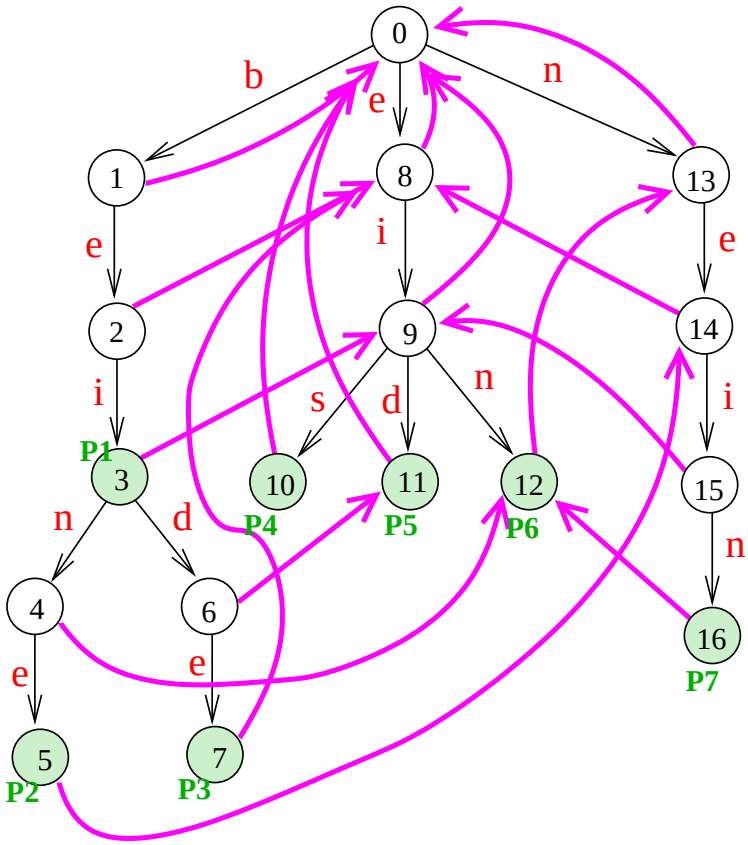
$f(0) \leftarrow -1$ Wurzel
$f(1) \leftarrow 0$ Kind der Wurzel
$f(8) \leftarrow 0$ Kind der Wurzel
$f(13) \leftarrow 0$ Kind der Wurzel
Knoten 2: $u = 1$, $a =$ e, $z = f(1) = 0$,Knoten $z = 0$ hat e-Nachfolger $8 =: f(2)$
Knoten 9: $u = 8$, $a =$ i, $z = f(8) = 0$, Knoten $z = 0$ hat keinen i-Nachfolger:
$z \leftarrow f(z) = -1$, Schleifenabbruch, $f(9) \leftarrow 0$
Knoten 14: $u = 13$, $a =$ e, $z = f(13) = 0$, Knoten $z = 0$ hat e-Nachfolger $8 =: f(14)$
Knoten 3: $u = 2$, $a =$ i, $z = f(2) = 8$, Knoten $z = 8$ hat i-Nachfolger $9 =: f(3)$
Knoten 10: $u = 9$, $a =$ s, $z = f(9) = 0$, Knoten $z = 0$ hat keinen s-Nachfolger:
$z \leftarrow f(z) = -1$, Schleifenabbruch, $f(10) \leftarrow 0$
Knoten 11: $u = 9$, $a =$ d, $z = f(9) = 0$, Knoten $z = 0$ hat keinen d-Nachfolger:
$z \leftarrow f(z) = -1$, Schleifenabbruch, $f(11) \leftarrow 0$
Knoten 12: $u = 9$, $a =$ n, $z = f(9) = 0$, Knoten $z = 0$ hat n-Nachfolger $13 =: f(12)$
Knoten 15: $u = 14$, $a =$ i, $z = f(14) = 8$, Knoten $z = 8$ hat i-Nachfolger $9 =: f(15)$
Knoten 4: $u = 3$, $a =$ n, $z = f(3) = 9$, Knoten $z = 9$ hat n-Nachfolger $12 =: f(4)$
12 ist mit „P_6“ markiert, daher: $g(4) \leftarrow 12$
Knoten 6: $u = 3$, $a =$ d, $z = f(3) = 9$, Knoten $z = 9$ hat d-Nachfolger $11 =: f(6)$
11 ist mit „P_5“ markiert, daher: $g(6) \leftarrow 11$
Knoten 16: $u = 15$, $a =$ n, $z = f(15) = 9$, Knoten $z = 9$ hat n-Nachfolger $12 =: f(16)$
12 ist mit „P_6“ markiert, daher: $g(16) \leftarrow 12$
Knoten 5: $u = 4$, $a =$ e, $z = f(4) = 12$, Knoten $z = 12$ hat keinen e-Nachfolger;
$z \leftarrow f(12) = 13$, Knoten $z = 13$ hat e-Nachfolger $14 =: f(5)$
Knoten 7: $u = 6$, $a =$ e, $z = f(6) = 11$, Knoten $z = 11$ hat keinen e-Nachfolger;
$z \leftarrow f(11) = 0$, Knoten $z = 0$ hat e-Nachfolger $8 =: f(7)$.

Die resultierende Funktion f ist in Abb. 5.3 graphisch dargestellt. Dabei wurden der Übersichtlichkeit halber die „g-Pfeile“ für $g(4) = g(16) = 12$ und $g(6) = 11$ nicht dargestellt. Alle anderen $g(v)$-Werte sind 0.

Mit Hilfe der Fehlerfunktion f und der „gefunden“-Funktion g können wir nun in einem gegebenen Text nach Vorkommen von Mustern aus Π suchen.

Der Algorithmus benutzt den Baum T_Π mit seiner Fehlerfunktion ähnlich wie einen endlichen Automaten. Wir bezeichnen die Knoten v in diesem Zusammenhang daher auch als „Zustände“.

Abbildung 5.3: Die Fehlerfunktion f



Startzustand/-knoten: $v_0 = 0$.

Die Buchstaben $S[i]$ werden in Runden $i = 1, \ldots, n$ bearbeitet, nach Runde i ist ein Knoten v_i erreicht.

Invariante: Am Ende von Runde i gilt Folgendes:
Die Inschrift auf dem Weg zu v_i ist das längste Präfix in Pref(Π), das Suffix von $S[1..i]$ ist.

Nehmen wir an, Runde $i-1$ ist zuende, und Knoten $v = v_{i-1}$ ist erreicht, mit $p_{i-1} = w(v_{i-1})$.

Sei $a = S[i]$ der nächste Buchstabe.

1. Fall: $p_{i-1}a \in \text{Pref}(\Pi)$.
(Das erkennt man daran, dass v einen a-Nachfolger v' hat.)
$v_i \leftarrow v'$.
Wenn v' mit einem P_t markiert ist: Paar $(t, i - |P_t| + 1)$ in die Ausgabe;
wenn $g(v') > 0$: Verfolge die Kette $g(v'), g(g(v')), \ldots$; jeder so erreichte Knoten u ist mit einem Muster P_t markiert, gib $(t, i - |P_t| + 1)$ aus.

2. Fall: $p_{i-1}a \notin \text{Pref}(\Pi)$.
(Das erkennt man daran, dass v keinen a-Nachfolger v' hat.)
Setze $v' \leftarrow v$, und führe folgende Schleife aus:
while $v' \neq -1 \wedge v'$ hat kein a-Kind **do** $v' \leftarrow f(v')$.
(Man sucht auf diese Weise das längste Suffix w' von $w(v)$ mit $w'a \in \text{Pref}(\Pi)$, falls so etwas existiert.)
Fall 2a: v' ist ein Knoten mit a-Kind. Setze $v_i \leftarrow v'$.
Fall 2b: $v' = -1$. Setze $v_i \leftarrow 0$.

Algorithmus 5.3.3 (**Aho-Corasick, mehrere Muster**).
AC-Textsuche(`S[1..n]`, Π)
Eingabe: Π: Menge von Mustern;
`S[1..n]`: Text;
Vorberechnet: Baum T_Π, Knoten haben Nummern $0, \ldots, m$; 0 ist Wurzel;
Fehlerfunktion f und „gefunden“-Funktion g;
Ausgabe: Menge `A` von Paaren; initialisiert: `A` $\leftarrow \emptyset$;

```
(1)  int v ← 0;
(2)  for i from 1 to n do
(3)    while v ≥ 0 ∧ v hat kein S[i]-Kind do v ← f[v];
(4)    if v ≥ 0 then v ← S[i]-Kind von v else v ← 0;
(5)    u ← v;
(6)    repeat
(7)      if u ist mit P_t markiert then A ← A ∪ {(t, i − |P_t| + 1)};
(8)      u ← g[u]
(9)    until u = 0.
```

Wir führen den Algorithmus an der Beispieleingabe

$$S[1..21] = \text{esbeidebeineineisbiss}$$

durch. Wenn für einen Knoten v kein g-Wert angegeben ist, ist dieser 0. Wenn für einen Knoten v nicht vermerkt ist, dass er mit P_t markiert ist, ist er unmarkiert.

Startzustand $v = 0$;

$S[1] = \text{e}$	von $v = 0$ zu e-Kind $v = 8$
$S[2] = \text{s}$	$v = 8$ hat kein s-Kind; $v \leftarrow f(8) = 0$;
	$v = 0$ hat kein s-Kind; $v \leftarrow f(0) = -1$;
	Schleifenabbruch: setze $v \leftarrow 0$;
$S[3] = \text{b}$	von $v = 0$ zu b-Kind $v = 1$;
$S[4] = \text{e}$	von $v = 1$ zu e-Kind $v = 2$;
$S[5] = \text{i}$	von $v = 2$ zu i-Kind $v = 3$; mit $P_1 =$ bei markiert, Ausgabe $(1, 3)$;
$S[6] = \text{d}$	von $v = 3$ zu d-Kind $v = 6$; $g(6) = 11$, mit $P_5 =$ eid markiert, Ausgabe $(5, 4)$;
$S[7] = \text{e}$	von $v = 6$ zu e-Kind $v = 7$; mit $P_3 =$ beide markiert, Ausgabe $(3, 3)$;
$S[8] = \text{b}$	$v = 7$ hat kein b-Kind; $v \leftarrow f(7) = 8$;
	$v = 8$ hat kein b-Kind; $v \leftarrow f(8) = 0$;
	von 0 zu b-Kind $v = 1$;
$S[9] = \text{e}$	von $v = 1$ zu e-Kind $v = 2$;
$S[10] = \text{i}$	von $v = 2$ zu i-Kind $v = 3$; mit $P_1 =$ bei markiert, Ausgabe $(1, 8)$;
$S[11] = \text{n}$	von $v = 3$ zu n-Kind $v = 4$; $g(4) = 12$, mit $P_6 =$ ein markiert, Ausgabe $(6, 9)$;
$S[12] = \text{e}$	von $v = 4$ zu e-Kind $v = 5$; mit $P_2 =$ beine markiert, Ausgabe $(2, 8)$;
$S[13] = \text{i}$	$v = 5$ hat kein i-Kind; $v \leftarrow f(5) = 14$;
	von $v = 14$ zu i-Kind $v = 15$;
$S[14] = \text{n}$	von $v = 15$ zu n-Kind $v = 16$; mit $P_7 =$ nein markiert, Ausgabe $(7, 11)$;
	$g(16) = 12$, mit $P_6 =$ ein markiert, Ausgabe $(6, 12)$; $g(12) = 0$;
$S[15] = \text{e}$	$v = 16$ hat kein e-Kind; $v \leftarrow f(16) = 12$;
	$v = 12$ hat kein e-Kind; $v \leftarrow f(12) = 13$;
	von $v = 13$ zu e-Kind $v = 14$;
$S[16] = \text{i}$	von $v = 14$ zu i-Kind $v = 15$;
$S[17] = \text{s}$	$v = 15$ hat kein s-Kind; $v \leftarrow f(15) = 9$;
	von $v = 9$ zu s-Kind $v = 10$; mit $P_4 =$ eis markiert, Ausgabe $(4, 15)$;
$S[18] = \text{b}$	$v = 10$ hat kein b-Kind; $v \leftarrow f(10) = 0$;
	von $v = 0$ zu b-Kind $v = 1$;
$S[19] = \text{i}$	$v = 1$ hat kein i-Kind; $v \leftarrow f(1) = 0$;
	$v = 0$ hat kein i-Kind; $v \leftarrow f(0) = -1$;
	Schleifenabbruch: setze $v \leftarrow 0$;
$S[20] = \text{s}$	$v = 0$ hat kein s-Kind; $v \leftarrow f(0) = -1$;
	Schleifenabbruch: setze $v \leftarrow 0$;
$S[21] = \text{s}$	$v = 0$ hat kein s-Kind; $v \leftarrow f(0) = -1$;
	Schleifenabbruch: setze $v \leftarrow 0$;

Satz 5.3.4. *Algorithmus 5.3.3 hat Zeitbedarf $O(n+$ Länge der Ausgabe$)$. Er berechnet korrekt sämtliche Stellen in S, an denen ein Muster aus Π vorkommt.*

Beweis: Die **Zeitanalyse** ist praktisch identisch zu der des KMP-Algorithmus. Wenn v der aktuelle Zustand ist, ist $q = |w(v)|$ (also das Level von Knoten v) die geeignete Maßzahl (die man als Potential auffassen kann). Anfangs ist $q = 0$, am Ende ist $q \geq 0$. Mit jedem gelesenen Buchstaben wird q um 1 erhöht, mit jedem Durchlauf durch den Rumpf der **while**-Schleife wird q strikt erniedrigt. Daher kann es maximal n solche Durchläufe geben.

Für die Korrektheitsanalyse zeigt man durch Induktion über die Schleifendurchläufe die folgenden Invarianten:

Äußere Schleife: Nach Zeile (5) gilt: $w(v_i)$ ist das längste Suffix von $S[1..i]$, das in Pref(Π) vorkommt.

Für die **while**-Schleife, in Runde i: $w(v)$ ist Suffix von $S[1..i-1]$, und es gibt in Pref(Π) kein Suffix w' von $S[1..i]$ mit Länge $> |w(v)| + 1$.

(Die Argumentation ist dabei sehr ähnlich zu der im Beweis der Korrektheit des KMP-Algorithmus.) □

Übungsaufgabe: Erstelle Baum T_Π, Fehlerfunktion f und „gefunden"-Funktion g für $\Pi = \{\text{dein}, \text{ein}, \text{herein}, \text{rein}, \text{sein}, \text{dasein}, \text{in}\}$.

Wende dann den Suchalgorithmus auf das Eingabewort $S =$ „deinhereinseindasein" an. Beachte besonders die Wirkungsweise der g-Funktion an der Stelle, wo das Teilwort „deinherein" gelesen worden ist.

Kapitel 5

Textalgorithmen

5.4 Der Algorithmus von Boyer und Moore

Der Algorithmus von Boyer und Moore findet ein oder alle Vorkommen eines Musters $P = P[1..m]$ in einem Text $S = S[1..n]$. Er löst also dieselbe Aufgabe wie der KMP-Algorithmus. Auch hier gibt es eine Vorverarbeitung des Musters und dann die Suche im Text.

Eigentlich handelt es sich dabei nicht um einen einzelnen Algorithmus, sondern um mehrere Varianten. Man kann von Algorithmen „vom Boyer-Moore-Typ“ sprechen. Allen Varianten ist gemeinsam, dass das Muster vorverarbeitet wird, um sogenannte *Shiftwerte* zu berechnen. Die Methoden, um zu diesen Shiftwerten zu kommen, sind aber unterschiedlich.

Algorithmen vom Boyer-Moore-Typ werden in der Praxis dem KMP-Algorithmus vorgezogen, da sie in der Suchphase oft schneller arbeiten als der KMP-Algorithmus. Das liegt daran, dass oft nicht alle Buchstaben von S angesehen werden müssen, dass also viel weniger als n Buchstabenvergleiche ausgeführt werden. Ein kleiner Nachteil, der aber nur in Sonderfällen relevant ist, ist folgender: Wenn das Muster im Text sehr oft vorkommt ($\Omega(n)$-mal, also mit vielen Überlappungen), dann kann die Laufzeit $\Omega(nm)$ betragen, im Gegensatz zum KMP-Algorithmus ($O(n+m)$ garantiert). Wenn das Muster nicht oder nur einmal vorkommt oder man nur nach dem ersten Vorkommen sucht, ist die Laufzeit $O(n + m)$. Die Laufzeitanalyse des BM-Algorithmus ist allerdings komplex; hierfür verweisen wir auf die Literatur [Dan Gusfield: Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology. Cambridge University Press 1997] oder [Volker Heun, Grundlegende Algorithmen, 2. Aufl., Vieweg, 2003].

5.4.1 Die Grundideen

Algorithmen vom Boyer-Moore-Typ suchen nach Vorkommen des Musters im Text, indem sie für eine strikt wachsende Folge von Werten $i \in \{m, m+1, \ldots, n\}$ testen, ob das Muster P „an der Stelle i in S vorkommt“, das heißt, ob $P = S[i-m+1..i]$ gilt. Hierzu wird das Muster $P[1..m]$ Buchstabe für Buchstabe mit $S[i-m+1..i]$ verglichen. (**Achtung:** Die Stelle i entspricht hier dem *letzten* Zeichen $P[m]$ des Musters.) Im Gegensatz zum KMP-Algorithmus läuft man dabei im Muster „von rechts nach links“.

function compare(i) // Vorbedingung: $m \le i \le n$
(1) `j` $\leftarrow m$;
(2) **while** `j` $\ge 1 \wedge$ `P[j]` $=$ `S[`$i-m+$`j]` **do** `j--`;
(3) **return** `j`.

Abbildung 5.1 Buchstabenvergleich „von rechts nach links“ an Stelle i.

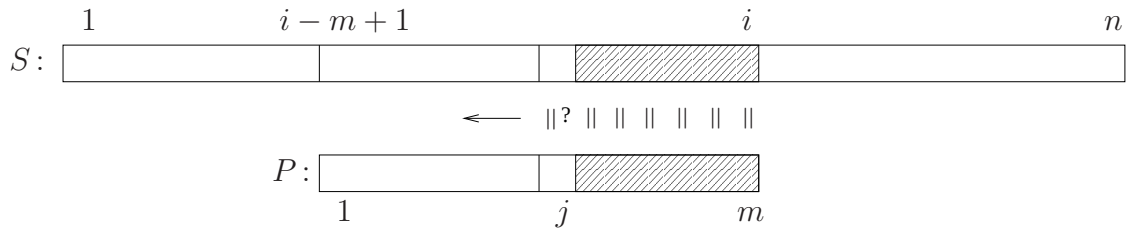


Abbildung 5.2
Buchstabenvergleich an Stelle i von rechts nach links.

Der Rückgabewert ist eine Zahl j, wobei entweder j die erste (von rechts gesehen) Position in P ist, die nicht zur entsprechenden Stelle in S passt, d. h., für die

$$j \ge 1 \text{ und } P[j+1..m] = S[i-m+j+1..i] \wedge P[j] \ne S[i-m+j]$$

gilt, oder $j = 0$ ist. Die Situation $j = 0$ bedeutet, dass das Muster gefunden worden ist.

In einem naiven Verfahren würde man diesen Vergleich für jedes $i = m, m+1, \ldots, n$ durchführen, d. h., man würde das Muster immer wieder um eine Position nach rechts verschieben. Die Kernidee der Algorithmen vom Boyer-Moore-Typ ist, aus der Information, die man bei der Suche an der Stelle i gewonnen hat, abzuleiten, dass man vielleicht einige Positionen, nämlich $i+1, i+2, \ldots, i+s-1$ für ein $s \ge 1$, überspringen kann, weil das Muster an diesen Stellen auf keinen Fall passt. Die nächste Vergleichsposition ist dann $i+s$. Dieser *Shiftwert* s sollte natürlich möglichst groß

sein, aber nicht so groß, dass man ein Vorkommen des Musters übersieht. Der Shiftwert hängt von j und dem Muster und eventuell auch von einem Buchstaben in $S[i-m+j..i]$ ab. Methoden, um zu solchen Shiftwerten zu kommen, werden durch zwei Ideen geliefert. Die erste Idee ist, einen Buchstaben im Bereich $S[i-m+j..i]$ zu benutzen, um die nächste sinnvolle Position zu bestimmen („bad character rule“ und „Horspool-Regel“). Die zweite Idee, die den eigentlichen Boyer-Moore-Algorithmus ausmacht, ist es, nur aufgrund der Position j des ersten Mismatch-Buchstabens zu entscheiden, wohin das Muster verschoben werden kann. Dies wird durch die „good suffix rule“ ausgedrückt. Diese führt zu einer (am Ende tabellierten, nur von P abhängigen) Funktion $\{0,1,\ldots,m\} \ni j \mapsto \mathrm{GS}(j)$, die zu jedem j den benötigten Shiftwert s liefert. Wie beim KMP-Algorithmus ist zur Berechnung dieser Funktion aus P eine etwas knifflige Vorverarbeitung des Musters nötig.

Wir bemerken noch, dass, im Gegensatz zum KMP-Algorithmus, Algorithmen vom Boyer-Moore-Typ nach dem Übergang zu einer neuen Position $i+s$ alle Information aus früheren Runden und Vergleichen vergessen und wieder mit dem naiven Vergleich „von rechts nach links“ beginnen.

5.4.2 Die BC-Regel

Wir beschreiben zunächst die „bad character rule“ („Schlechte-Buchstaben-Regel“, BC-Regel), die zur Ermittlung von Shiftwerten dient. Wenn ein Vergleich wie in Abb. 5.1 mit einem $j \in \{1,\ldots,m\}$ endet, dann hat man soeben den Buchstaben $x := S[i-m+j]$ gelesen, mit $x \neq P[j]$ („bad character“). An Stelle i kann das Muster also nicht vorkommen. Die BC-Regel benutzt nun die Einsicht, dass das Muster in S nur an Positionen $i+s$ sitzen kann, wo der Buchstabenposition $S[i-m+j]$ eine Stelle in P gegenübersteht, die gleich x ist. Man betrachtet also zunächst einmal alle Positionen in $P[1..m-1]$, die gleich x sind[1], und wählt davon die maximale, entsprechend einem möglichst kurzen Shift, damit keine mögliche Position von P ausgelassen wird. Rechnerisch ausgedrückt: Sei

$$\mathrm{BC}(x) := \max(\{0\} \cup \{k < m \mid P[k] = x\}),$$

das ist der größte Index $k < m$ mit $P[k] = x$, falls x in $P[1..m-1]$ vorkommt, und 0, falls x nicht vorkommt. Wenn $j \leq \mathrm{BC}(x)$, wählen wir Shiftwert 1. Wenn $j > \mathrm{BC}(x)$, dann kann für $i < i' < i + (j - \mathrm{BC}(x))$ keinesfalls $P = S[i'-m+1..i']$ gelten: Weil $\mathrm{BC}(x) < j+i-i' < m$, haben wir $P[j+i-i'] \neq x$; andererseits gilt $S[i'-m+(j+i-i')] = S[i-m+j] = x$. Wenn wir also als nächste zu testende Position $i + \max\{1, j - \mathrm{BC}(x)\}$ wählen, wird garantiert keine mögliche Position des Musters ausgelassen. Die Situation ist in Abb. 5.3 veranschaulicht.

[1] Ein Vorkommen von x an der Stelle $P[m]$ kann man ignorieren. Da P mindestens um 1 nach rechts geschoben wird, kann der letzte Buchstabe des Musters nach dem Schieben nie unter einem Buchstaben aus $S[i-m+j..i]$ stehen. Dies gilt für alle ähnlichen Verfahren, die einen Buchstaben aus $S[i-j+1..i]$ betrachten, wie z. B. die Horspool-Regel.

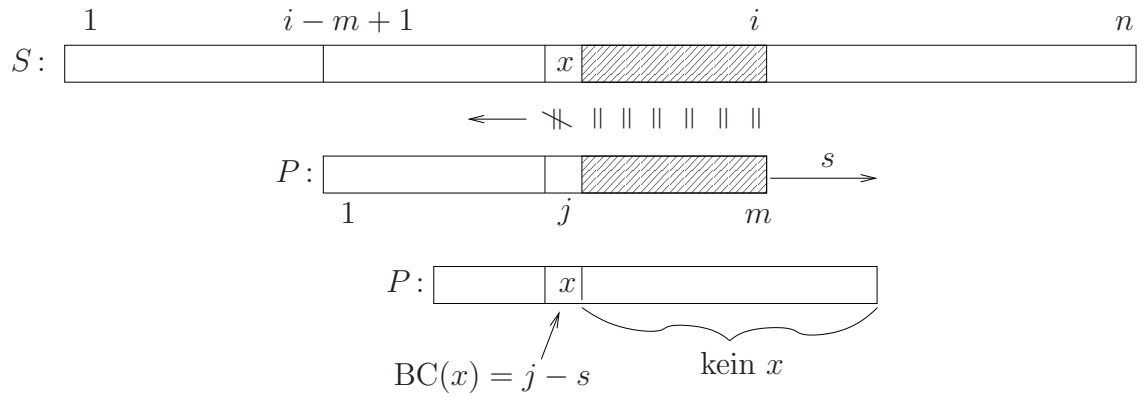


Abbildung 5.3 BC-Regel: Wir wählen als Shiftweite $s = j - \mathrm{BC}(x)$, falls dies positiv ist. Wenn $j \leq \mathrm{BC}(x)$, hilft die Regel nichts, und wir wählen $s = 1$.

Die Werte $\mathrm{BC}(x)$, $x \in \Sigma$, lassen sich ganz einfach in Zeit $O(m)$ berechnen. Wir benutzen dazu ein (assoziatives) Array BC, das für jeden Buchstaben $x \in \Sigma$ eine Position hat. Es genügt ein Durchlauf durch das Muster von links nach rechts. Man beobachte, wie durch die Benutzung von $\mathrm{BC}[P[k]]$ als Speicherziel mit $k = 1, \ldots, m-1$ die maximale Position $k < m$ mit $T[k] = x$ in $\mathrm{BC}[x]$ gespeichert wird, falls x in $P[1..m-1]$ vorkommt.[2]

function buildBCTable($P[1..m]$)
(1) **for** $x \in \Sigma$ **do** $\mathrm{BC}[x] \leftarrow 0$;
(2) **for** $\mathtt{k}$ **from** 1 **to** m **do** $\mathrm{BC}[P[\mathtt{k}]] \leftarrow \mathtt{k}$;
(3) **return** $\mathrm{BC}[x\colon \Sigma]$.

Wenn die Buchstaben in P nicht das ganze Alphabet Σ ausschöpfen, wird man die „Fehlbuchstaben“, die in P nicht vorkommen, in *einem* Eintrag der Tabelle BC zusammenfassen: $\mathrm{BC}[\text{„Fehlbuchstabe“}] = 0$.

Beispiel: $\Sigma = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$ und $P[1..11] = \mathtt{abracadabra}$. Dann ist BC für P durch die folgende Tabelle gegeben. Man beachte, dass der Eintrag $P[11] = \mathtt{a}$ ignoriert wird und daher $\mathrm{BC}(\mathtt{a}) = 8$ ist.

x	a	b	c	d	r	sonstige
$\mathrm{BC}(x)$	8	9	5	7	10	0

Die ***einfache* BC-Regel** sagt nun: Wenn an der Stelle j ein Mismatch auftritt, dann verschiebe das Muster um $s := \max\{1, j - \mathrm{BC}(x)\}$ nach rechts. Dann verfahre wieder

[2] Dieser Trick zur Bildung eines Maximums wird uns im Weiteren noch mehrfach begegnen.

wie in Abb. 5.1. Wenn das Muster an Stelle i gefunden worden ist, verschiebe um $s = 1$. – Es ergibt sich der folgende Algorithmus.

Algorithmus 5.4.1 (Boyer-Moore mit einfacher BC-Regel)**.**
Eingabe: `P[1..`m`]`, `S[1..`n`]` // `P`: Muster, `S`: Text
Vorberechnet: BC-Shiftwerte als Tabelle `BC`$[x\colon \Sigma]$;
Ausgabe: Menge `A` $\subseteq \{1, \ldots, n-m+1\}$; initialisiert: `A` $\leftarrow \emptyset$;

(1) **int** `i` $\leftarrow m$;
(2) **while** `i` $\leq n$ **do**
(3) `j` $\leftarrow m$;
(4) **while** `j` $\geq 1 \wedge$ `P[j]` $=$ `S[i-`m`+j]` **do** `j--`;
(5) **if** `j` $= 0$
(6) **then** `A` $\leftarrow$ `A` $\cup \{$`i-`m`+1`$\}$; `i++`;
(7) **else** `i` $\leftarrow$ `i +` $\max(1,$ `j -` BC`[S[i-`m`+j]]`$)$;
(8) **return** `A`.

Beispiel: (Blanks zählen als Buchstaben.)[2]

$S[1..42] =$ `IM␣HEU-␣ODER␣NUDELHAUFEN␣FINDE␣ALLE␣NADELN`, $P[1..5] =$ `NADEL`.

Vorverarbeitung:

x	`A`	`D`	`E`	`N`	sonstige
BC(x)	2	3	4	1	0

$i = 5$	Mismatch bei $j = 5$ mit $x =$ `E`	$s = 5 - \mathrm{BC}($`E`$) = 1$
$i = 6$	Mismatch bei $j = 5$ mit $x =$ `U`	$s = 5 - \mathrm{BC}($`U`$) = 5$
$i = 11$	Mismatch bei $j = 5$ mit $x =$ `E`	$s = 5 - \mathrm{BC}($`E`$) = 1$
$i = 12$	Mismatch bei $j = 5$ mit $x =$ `R`	$s = 5 - \mathrm{BC}($`R`$) = 5$
$i = 17$	Mismatch bei $j = 5$ mit $x =$ `E`	$s = 5 - \mathrm{BC}($`E`$) = 1$
$i = 18$	Mismatch bei $j = 2$ mit $x =$ `U`	$s = 2 - \mathrm{BC}($`U`$) = 2$
$i = 20$	Mismatch bei $j = 5$ mit $x =$ `A`	$s = 5 - \mathrm{BC}($`A`$) = 3$
$i = 23$	Mismatch bei $j = 5$ mit $x =$ `E`	$s = 5 - \mathrm{BC}($`E`$) = 1$
$i = 24$	Mismatch bei $j = 5$ mit $x =$ `N`	$s = 5 - \mathrm{BC}($`N`$) = 4$
$i = 28$	Mismatch bei $j = 5$ mit $x =$ `N`	$s = 5 - \mathrm{BC}($`N`$) = 4$
$i = 32$	Mismatch bei $j = 5$ mit $x =$ `A`	$s = 5 - \mathrm{BC}($`A`$) = 3$
$i = 35$	Mismatch bei $j = 5$ mit $x =$ `E`	$s = 5 - \mathrm{BC}($`E`$) = 1$
$i = 36$	Mismatch bei $j = 5$ mit $x =$ `␣`	$s = 5 - \mathrm{BC}($`␣`$) = 5$
$i = 41$	Muster gefunden, $j = 0$	$s = 1$
$i = 42$	Mismatch bei $j = 5$ mit $x =$ `N`	$s = 5 - \mathrm{BC}($`N`$) = 4$
$i = 46$	Stop.	

[2] Zum Ausprobieren: `https://people.ok.ubc.ca/ylucet/DS/BoyerMoore.html` .
Aber Achtung: Die App behandelt mit BC(`L`) = 5 den letzten Buchstaben „`L`“ des Musters anders (ungeschickter) als unser Algorithmus.

Es werden 22 Buchstabenvergleiche ausgeführt. Das ist wenig im Vergleich zur Länge des betrachteten Textes (42), besonders wenn man bedenkt, dass allein für das Finden des Musters schon fünf (erfolgreiche) Vergleiche nötig sind.

Nicht in allen Situationen sind die möglichen weiten Sprungweiten so hilfreich: Betrachte $S[1..13] = \texttt{abababcababac}$, $P[1..4] = \texttt{caba}$.

Vorverarbeitung:

x	a	b	c	sonstige
BC(x)	2	3	1	0

$i = 4$	Mismatch bei $j = 4$ mit $x = \texttt{b}$	$s = 4 - \mathrm{BC}(\texttt{b}) = 1.$
$i = 5$	Mismatch bei $j = 1$ mit $x = \texttt{b}$	$s = \max\{1, \underbrace{1 - \mathrm{BC}(\texttt{b})}_{=-2}\} = 1.$
$i = 6$	Mismatch bei $j = 4$ mit $x = \texttt{b}$	$s = 4 - \mathrm{BC}(\texttt{b}) = 1.$
$i = 7$	Mismatch bei $j = 4$ mit $x = \texttt{c}$	$s = 4 - \mathrm{BC}(\texttt{c}) = 3.$
$i = 10$	Muster gefunden, $j = 0$	$s = 1.$
$i = 11$	Mismatch bei $j = 4$ mit $x = \texttt{b}$	$s = 4 - \mathrm{BC}(\texttt{b}) = 1.$
$i = 12$	Mismatch bei $j = 1$ mit $x = \texttt{b}$	$s = \max\{1, 1 - \mathrm{BC}(\texttt{b})\} = 1.$
$i = 13$	Mismatch bei $j = 4$ mit $x = \texttt{c}$	$s = 4 - \mathrm{BC}(\texttt{c}) = 3.$
$i = 16$	Stop.	

Ablauf von Algorithmus 5.4.1 als Schema: „ “ bezeichnet „erfolgreicher Vergleich“, „ “ bedeutet „Mismatch“. Die unterstrichenen Buchstaben sind diejenigen, die von der BC-Regel unter die passenden Textbuchstaben gesetzt werden. Positionen ohne unterstrichene Buchstaben kommen durch einen Shiftwert von 1 zustande, wenn die BC-Regel nicht anwendbar ist.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	a	b	a	b	a	b	c	a	b	a	b	a	c	–	–	–
4	c	a	b	a												
5		c	a	b	a											
6			c	a	b	a										
7				c	a	b	a									
10							c	a	b	a						
11								c	a	b	a					
12									c	a	b	a				
13										c	a	b	a			
16	Stop.												c	a	b	a

Bei Textlänge 13 gibt es 17 Vergleiche, davon 7 erfolglose („Mismatches“) und 10 erfolgreiche. Ein Buchstabe des Textes ($S[1]$) wird nicht gelesen, andererseits wird $S[10]$ zweimal erfolgreich verglichen. Bei größeren Mustern und Texten kann dies natürlich auch noch häufiger auftreten.

Die einfache BC-Regel wirkt nicht, wenn $j < \mathrm{BC}(x)$ ist, wenn also x rechts von der Mismatch-Stelle j in P vorkommt, oder nach dem Finden des Musters. Man

schiebt dann nur um einen Buchstaben. (Im letzten Beispiel ist bei Positionen $i = 5$ und $i = 12$ der Wert j zu klein; bei $i = 10$ wurde das Muster gefunden.) Dies kann bei kleinen Alphabeten eher vorkommen als bei großen. Besonders ungünstig sind hier zum Beispiel das binäre Alphabet $\{0, 1\}$ und vierelementige Alphabete wie $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Wir betrachten nun Methoden, die diesem Problem ausweichen.

Variante 1: Anstelle des „Mismatch-Buchstabens“ $x = S[i - m + j]$ betrachtet man $y = S[i]$ und schiebt das Muster um $s = m - \mathrm{BC}(y)$ Positionen nach rechts. (Der Buchstabe $S[i]$ ist natürlich nicht unbedingt ein „bad character“.) Dadurch erhält man immer eine positive Verschiebung aus der BC-Regel, und das am weitesten rechts stehende y in $P[1..m-1]$ kommt an die Position von $y = S[i]$. Es ergibt sich folgender Algorithmus, der 1980 von Horspool[4] vorgeschlagen wurde:

Algorithmus 5.4.2 (Boyer-Moore: Horspool-Variante)**.**
Eingabe: `P[1..`m`]`, `S[1..`n`]` // `P`: Muster, `S`: Text
Vorberechnet: BC-Shiftwerte als Tabelle `BC`$[x\colon \Sigma]$;
Ausgabe: Menge `A` $\subseteq \{1, \ldots, n - m + 1\}$; initialisiert: `A` $\leftarrow \emptyset$;

(1) **int** `i` $\leftarrow m$;
(2) **while** `i` $\leq n$ **do**
(3) `j` $\leftarrow m$;
(4) **while** `j` $\geq 1 \wedge$ `P[j]` $=$ `S[i-`m`+j]` **do** `j--`;
(5) **if** `j` $= 0$ **then** `A` $\leftarrow$ `A` $\cup \{$`i-`m`+1`$\}$;
(6) `i` $\leftarrow$ `i + `m` - `BC`[S[i]]`; // *nur hier* Unterschied zu Algorithmus 5.4.1
(7) **return** `A`.

Ablauf von Algorithmus 5.4.2 als Schema. „ “ bezeichnet „erfolgreicher Vergleich“, „ “ bedeutet „Mismatch“. Die unterstrichenen Buchstaben sind diejenigen, die von der Horspool-Regel unter die passenden Textbuchstaben gesetzt werden.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	a	b	a	b	a	b	c	a	b	a	b	a	c	–
4	c	a	b	a										
5		c	a	b	a									
7				c	a	b	a							
10							c	a	b	a				
12									c	a	b	a		
14	Stop.										c	a	b	a

Hier gibt es nur 5 Mismatches und 10 erfolgreiche Vergleiche. Textpositionen $S[1]$, $S[6]$, $S[13]$ werden nicht gelesen; Textposition $S[10]$ wird zweimal erfolgreich verglichen.

Man vergleiche die Abläufe der beiden Algorithmen und beachte, dass sich BM mit BC-Regel und BM-Horspool an der Stelle i gleich verhalten, wenn ein Mismatch bei $j = m$ auftritt. Wenn der Mismatch bei $j < m$ passiert oder wenn das Muster gefunden worden ist, schiebt BM-Horspool immer um denselben Wert $s = m - \mathrm{BC}(P[m])$

[4] R. Nigel Horspool, britisch-kanadischer Informatiker.

weiter. (Im Beispiel ist dies 2.)

Die einfache BC-Regel, insbesondere die Horspool-Variante, hat den Vorteil, sehr einfach anwendbar und sehr effizient zu sein. Es wird ihr nachgesagt, dass sie alleine (ohne die noch folgenden komplexeren Teile des BM-Algorithmus!) zu sehr schnellen Suchzeiten in natürlichsprachigen Texten führt. Allgemein wirkt die Regel umso besser, je größer das Alphabet ist. Wenn dann in **compare** (Abb. 5.1) schon früh, also für große j, ein Mismatch mit einem solchen Buchstaben auftritt, kann das Muster ohne große Kosten auf einen Schlag sehr weit geschoben werden.

Bemerkung. Wenn die Buchstabenverteilung zufällig ist, dann tritt typischerweise früh ein Mismatch ein: An einer Position i erwarten wir nur konstant viele Buchstabenvergleiche. Wenn zudem das Alphabet groß ist, dann ist die erwartete Shiftweite groß, nämlich $\Theta(\min\{m, |\Sigma|\})$. Daraus ergibt sich eine erwartete Rechenzeit für die Textsuche von $O(n/\min\{m, |\Sigma|\})$. Wir skizzieren eine heuristische Rechnung, die dies belegt. Nehmen wir an, das Alphabet Σ habe Größe σ und Text $S[1..n]$ und Muster $P[1..m]$ sind rein zufällig gewählt, d. h., an jeder Stelle steht jeder Buchstabe mit derselben Wahrscheinlichkeit $1/\sigma$, und die Positionen sind unabhängig.[3]
Nehmen wir zur Vereinfachung weiter an, dass die Buchstaben im Text nach jeder Verschiebung des Musters aufs Neue zufällig bestimmt werden, so dass wir uns um Abhängigkeiten bei überlappenden Musterpositionen nicht kümmen müssen. **Übung:** (i) Dann ist die erwartete Anzahl von Buchstabenvergleichen an Stelle i genau $\frac{\sigma}{\sigma-1}(1-(1/\sigma)^m)$, ein Wert in $(1, 2)$, also $\Theta(1)$. (ii) Die erwartete Shiftdistanz ist $\sigma(1-(1-1/\sigma)^m)$. Für $m > \frac{\sigma}{2}$ führt dies zur Abschätzung $\sigma(1-(1-1/\sigma)^{\sigma/2}) = O(\sigma)$. Die Shiftdistanz ist nie größer als m. (iii) Insgesamt ist damit die erwartete Anzahl von Buchstabenvergleichen in $O(n/\min\{m, \sigma\})$.

Variante 2: Eine gewisse Verbesserung kann man auch durch Verwendung der ***starken Version*** der BC-Regel erreichen. Wenn $j < \mathrm{BC}(x)$ ist, wäre es wünschenswert, das erste Vorkommen von x in P *strikt links von* Position j zu finden. Dazu sei $\mathrm{BC}'(x, j)$ der größte Index $k < j$ mit $P[k] = x$ (dabei sei $\mathrm{BC}'(x, j) = 0$, falls x in $P[1..j-1]$ nicht vorkommt). Dann können wir Shiftwert $s = j - \mathrm{BC}'(x, j) \geq 1$ wählen.

Man kann die BC′-Funktion auf naive Weise bereitstellen: mit $\Theta(m|\Sigma|)$ Speicherplatz und Vorbereitungszeit. Dann kostet jeder Zugriff $O(1)$ Zeit. Allerdings möchte man besonders für größere Alphabete keinen solchen Aufwand treiben. Ein guter Kompromiss ist folgender: Man legt für jeden Buchstaben $x \in \Sigma$ eine lineare Liste L_x an, in der die Elemente der Menge $\{k \mid 1 \leq k < m, P[k] = x\} \cup \{0\}$ in fallender Reihenfolge aufgeführt sind. (Diese Vorverarbeitung kann in Zeit $O(m)$ durchgeführt werden. Auch der Platzbedarf ist $O(m)$; wenn m viel kleiner als $|\Sigma|$ ist, kann man eine Hashtabelle o. ä. benutzen, um viele leere Listen zu vermeiden.)

Beispiel: Für $P[1..14] = \texttt{araratararatar}$ ist $L_{\texttt{a}} = (13, 11, 9, 7, 5, 3, 1, 0)$, $L_{\texttt{r}} = (10, 8, 4, 2, 0)$, $L_{\texttt{t}} = (12, 6, 0)$ und $L_{\text{sonst}} = (0)$ bzw. die Konvention, dass Buchstaben, die nicht in P vorkommen, stets den Wert 0 liefern.

[3] Vorsicht: Diese Annahme trifft in in wirklichen Anwendungen nicht zu. Aber sie zeigt das Einsparpotenzial des BM-Horspool-Algorithmus auf.

Mit Hilfe dieser Listen lässt sich für jedes x und j der Wert $\mathrm{BC}'(x, j)$ in Zeit $O(m-j)$ berechnen: Man liest Liste L_x bis zum ersten Eintrag, der kleiner als j ist. Da Zeit $\Omega(m-j)$ auch für den Durchlauf der Schleife in der Funktion **compare** (Abb. 5.1) benötigt wird, bis Position j erreicht ist, erhöht sich der Zeitbedarf höchstens um einen konstanten Faktor gegenüber dem, was bei der Benutzung einer kompletten Tabelle für BC' nötig ist.

Trotz aller Erfolge und Vorteile der verschiedenen BC-Regeln ist doch festzuhalten, dass sie im schlechtesten Fall alle zu einer sehr schlechten Rechenzeit führen können, wie die folgende Übungsaufgabe zeigt.

Übung: Zeigen Sie, dass die Boyer-Moore-Varianten mit der BC-Regel, der Horspool-Regel und auch der BC′-Regel auf der Eingabe (P, T) mit $P = P[1..m] = \mathtt{ba}^{m-1}$ und $S = S[1..n] = \mathtt{a}^{n-m}\mathtt{ba}^{m-1}$ mindestens $(n-2m+1)m$ Buchstabenvergleiche durchführen. (Der Grund für die hohe Anzahl an Vergleichen ist bei allen drei Varianten, dass für jede Position $i \in \{m, \ldots, n-m\}$ alle Buchstaben des Musters angesehen werden und dass der Shift jeweils nur 1 beträgt. Das Muster wird ganz am Ende einmal gefunden. Sobald $S[n-m+1] = \mathtt{b}$ gelesen wird, erfolgt ein Shift zur richtigen Position.)

5.4.3 Die GS-Regel

Die eigentliche Idee des von Boyer und Moore vorgeschlagenen Algorithmus ist die „good suffix rule“ („Gute-Suffix-Regel“, GS-Regel). Dabei geht es darum, gewisse Shiftweiten für das Muster dadurch auszuschließen, dass die bereits erfolgreich verglichenen Buchstaben $P[j+1..m]$ betrachtet werden sowie für $j \geq 1$ die Tatsache eines Mismatches an dieser Stelle. Von den verbleibenden „zulässigen“ Shiftweiten wird dann die kleinste gewählt, damit keine mögliche Position ausgelassen wird. Erst die GS-Regel führt zu garantierter Suchzeit $O(n)$. Die Überlegungen zu dieser Regel liefern eine Funktion $\mathrm{GS}\colon \{0, \ldots, m\} \ni j \mapsto \mathrm{GS}(j) \in \{1, \ldots, m\}$, die für die Anwendung im Algorithmus tabelliert wird. Dann wird im Algorithmus nach dem Testen an Stelle i mit erster Mismatch-Stelle j zur neuen Position $i + \mathrm{GS}(j)$ gesprungen. Die Definition der Funktion GS erfordert einige Überlegung, die Berechnung der Tabelle in Linearzeit weitere Mühe.

Sei $0 \leq j \leq m$, und sei j die Zahl, die vom Aufruf **compare**(i) (s. Abb. 5.1) geliefert wird. Wenn $1 \leq j \leq m$, dann handelt es sich um eine Mismatch-Stelle, bei $j = 0$ wurde soeben das Muster gefunden. Sei $1 \leq s \leq m$. Wir überlegen, ob es sinnvoll sein kann, das Muster um s Stellen nach rechts zu schieben. Es gibt zwei Fälle.

1. Fall: $s < j$ („kleiner Shift“).

Wir wissen nach dem Ergebnis von **compare**(i): $P[j+1..m] = S[i-m+j+1..i]$ und $P[j] \neq S[i-m+j]$. Wenn nach Verschieben um s Stellen nach rechts (an die Position $i+s$) das Muster „passt“, also $P[1..m] = S[i+s-m+1..i+s]$ gilt, dann muss $P[j-s+1..m-s] = S[i-m+j+1..i]$ und $P[j-s] = S[i-m+j]$ gelten (s.

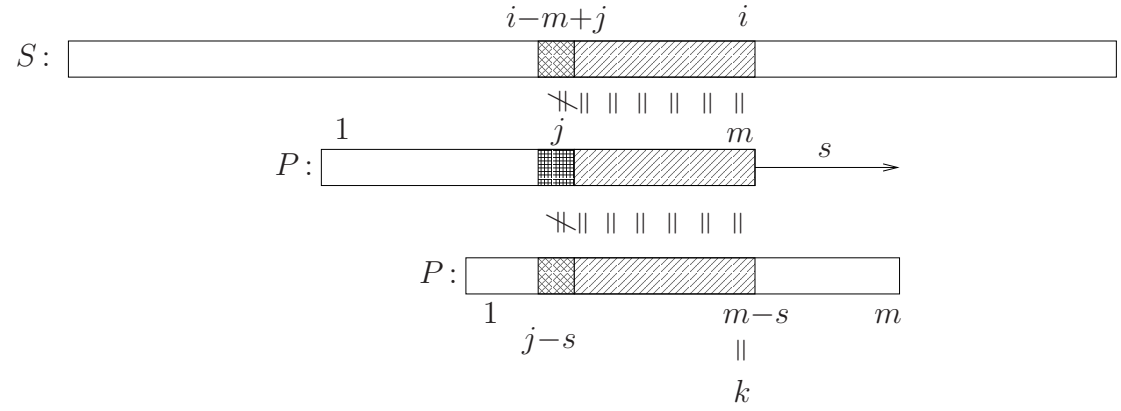


Abbildung 5.4 GS-Regel, 1. Fall: „kleiner Shift“. Gleichheiten und Ungleichheiten zwischen Positionen des Musters und des Textes, falls ein Shift von $s < j$ zu einer Position $i + s$ im Text führt, an der das Muster vorkommt.

Abb. 5.4). Es folgt:

$$P[j+1..m] = P[j-s+1..m-s] \text{ und } P[j] \neq P[j-s]. \tag{5.1}$$

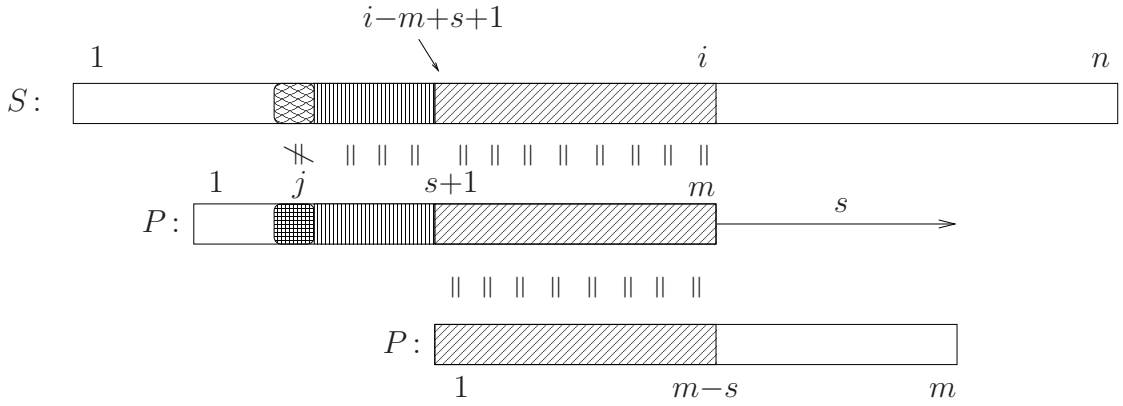


Abbildung 5.5 GS-Regel, 2. Fall: „großer Shift“. Gleichheiten und Ungleichheiten zwischen Positionen des Musters und des Textes, falls ein Shift von $s \geq j$ zu einer Position $i + s$ im Text führt, an der das Muster vorkommt.

2. Fall: $j \leq s \leq m$ („großer Shift“).

(Hier ist der Fall $j = 0$ eingeschlossen, was bedeutet, dass das Muster an Stelle i gefunden worden ist.) Nach dem Ergebnis von **compare**(i) wissen wir: $P[j+1..m] = S[i-m+j+1..i]$, also insbesondere $P[s+1..m] = S[i-m+s+1..i]$. Bei einer Verschiebung um s steht das Muster P komplett rechts von der Stelle j, der nicht passende Buchstabe spielt also keine Rolle. Aus $S[i+s-m+1..i+s] = P[1..m]$ folgt

$S[i+s-m+1..i] = P[1..m-s]$, also (s. Abb. 5.5):

$$P[s+1..m] = P[1..m-s] \qquad \text{(d. h.: } P[1..m-s] \text{ ist Rand von } P\text{).} \tag{5.2}$$

Bemerkungen: (a) Bedingungen (5.1) und (5.2) betreffen das Suffix $P[j..m]$ bzw. das Suffix $P[s+1..m]$ von P, was zu dem Namen "good suffix rule" führt.
(b) Manchmal unterscheidet man „schwache GS-Regel" (*ohne* die Bedingung „$P[j] \neq P[j-s]$" in (5.1)) und „starke GS-Regel" (*mit* dieser Bedingung). Es stellt sich heraus, dass die Mismatch-Bedingung für die lineare Laufzeit entscheidend ist, daher betrachten wir *nur* die starke Version.

Definition 5.4.3. *Sei $P[1..m]$ das Muster und $0 \leq j \leq m$. Eine Zahl s, $1 \leq s \leq m$, heißt **zulässige Shiftweite für j**, wenn $1 \leq s < j$ und (5.1) oder $j \leq s \leq m$ und (5.2) gilt.*

Wegen $P[m+1..m] = \varepsilon = P[1..0]$ ist die Shiftweite $s = m$ immer zulässig.

Definition 5.4.4. $\mathrm{GS}(j) := \min\{s \mid s \text{ ist zulässige Shiftweite für } j\}$ *heißt der **GS-Shiftwert** für j.*

Beispiel: $P[1..14] =$ `araratararatar`.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\mathrm{GS}(j)$	6	6	6	6	6	6	6	12	12	12	12	12	4	14	1

Die Werte 6 für $j = 0, \ldots, 6$ in dieser Tabelle rühren daher, dass der längste Rand von $P[1..14]$ das Teilwort `araratar` (mit Länge $8 = 14 - 6$) ist. Es ist eine gute Übung, auch die anderen Werte anhand der Definition zu überprüfen. Insbesondere der Wert $\mathrm{GS}(13) = 14$ ist interessant. Er bedeutet, dass im Fall $S[i] =$ `r` und $S[i-1] \neq$ `a` das Muster um 14 Positionen weitergeschoben werden kann. Tatsächlich: Links von jedem Vorkommen von `r` im Muster steht ein `a`, so dass es keine Möglichkeit gibt, dass Position $S[i]$ in einem Vorkommen des Musters enthalten ist.

Aus den bisherigen Überlegungen folgt: Wenn wir nach einer Runde für i, die entweder mit einem Mismatch $P[j] \neq S[i-m+j]$ für $j \geq 1$ oder mit einem Auffinden des Musters ($j = 0$) geendet hat, den nächsten Versuch an Position $i := i + \mathrm{GS}(j)$ starten, so wird hierdurch kein Vorkommen des Musters P in S übersehen. Nun können wir den Boyer-Moore-Algorithmus formulieren. Wir tun dies zunächst in der ursprünglichen Version.

Algorithmus 5.4.5 (Boyer-Moore)**.**
BM-Textsuche(S[1..n],P[1..m])
Eingabe: P[1..m], S[1..n] // P: Muster, S: Text
Vorberechnet: GS[0..m]: GS-Regel-Shiftwerte als Tabelle;
Ausgabe: Menge A $\subseteq \{1,\ldots,n-m+1\}$; initialisiert: A $\leftarrow \emptyset$;

```
(1)  int i ← m;
(2)  while i ≤ n do
(3)    j ← m;
(4)    while j ≥ 1 ∧ P[j] = S[i-m+j] do j--;
(5)    if j = 0 then A ← A ∪ {i-m+1};
(6)    i ← i + GS[j];
(7)  return A.
```

Man bewundere die Einfachheit dieses Algorithmus. (Das Geheimnis liegt natürlich in der GS-Funktion.) Wir können den Algorithmus noch erweitern, indem wir auch die BC-Regel in der einen oder anderen Form berücksichtigen.

Algorithmus 5.4.6 (Boyer-Moore mit BC($'$)-Regel)**.**
BM+BC-Textsuche(S[1..n],P[1..m])
Eingabe: P[1..m], S[1..n] // P: Muster, S: Text
Vorberechnet: GS[0..m]: GS-Regel-Werte als Tabelle;
Datenstruktur für (BC(x) oder) BC$'(x,j)$, $x \in \Sigma$, $1 \le j \le m$;
Ausgabe: Menge A $\subseteq \{1,\ldots,n-m+1\}$; initialisiert: A $\leftarrow \emptyset$;

```
(1)  int i ← m;
(2)  while i ≤ n do
(3)    j ← m;
(4)    while j ≥ 1 ∧ P[j] = S[i-m+j] do j--;
(5)    if j = 0 then A ← A ∪ {i-m+1};
(6)    s ← GS[j];
(7)    if j ≥ 2 then s ← max(s,j-BC'[S[i-m+j],j]);
(8)    i ← i+s;
(9)  return A.
```

Der folgende Satz zum Berechnungsaufwand wird hier nicht bewiesen. Für Interessierte findet sich der Beweis im Buch von V. Heun oder dem von D. Gusfield.

Satz 5.4.7. *Algorithmus 5.4.5 führt maximal* $4n$ *Buchstabenvergleiche durch und hat Laufzeit* $O(n)$*, wenn das Muster nicht im Text vorkommt.*

Es ist eine typische Subtilität im Verhalten dieser Algorithmen vom Boyer-Moore-Typus und eine große Schwierigkeit in der Zeitanalyse, dass nicht ohne Weiteres gesagt ist, dass die Anzahl der Vergleiche in Algorithmus 5.4.6 geringer ist als die in Algorithmus 5.4.5. (Wieso? Der Grund ist, dass durch die BC-Shifts ganz andere i-Werte erreicht werden können als im gewöhnlichen Ablauf ohne BC-Shifts. Was dies dann auf die Laufzeit für Auswirkungen hat, ist schwierig zu sehen.) Man kann

aber auch im Fall des BM-Algorithmus mit BC-Shifts lineare Laufzeit beweisen, *falls das Muster nicht vorkommt*. Daraus folgt auch lineare Laufzeit, wenn man nur das erste Vorkommen des Musters finden will. Wenn man *alle* Vorkommen finden will und dennoch lineare Laufzeit garantiert sein soll, muss man den Algorithmus etwas modifizieren („Galil-Modifikation“, siehe das schon erwähnte Buch von Gusfield).

Was ist noch zu tun? Wir müssen einen Linearzeit-Algorithmus für die Berechnung der GS-Regel-Shiftwerte angeben. Dies erfordert zusätzlichen (mentalen und Rechen-) Aufwand, und einen eigenen Abschnitt.

Die folgenden Seiten stellen für interessierte Leser/innen die systematische Entwicklung einer Berechnung der GS-Tabelle dar. Dies wird in der Vorlesung aber nicht besprochen, gehört also auch nicht zum Prüfungsstoff.

WS 2021/22: Ende des prüfungsrelevanten Teils.

5.4.4 Vorbearbeitung des Musters für den BM-Algorithmus

Die Ermittlung der Shiftwerte GS(j) für die GS-Regel erfordert mehrere Schritte. Der erste und zentrale Schritt ist die Berechnung der sogenannten *Präfixwerte* oder *Z-Werte* eines Wortes $T = T[1..n]$. Diese Werte liefern auch für andere Anwendungen nützliche Strukturinformation über T. Dazu gespiegelt definieren wir *Suffixwerte*. Wenn die Suffixwerte des Musters vorliegen, lassen sich die Shiftwerte recht einfach ermitteln.

5.4.4.1 Die Präfixwerte eines Wortes

Definition 5.4.8. *Sei $T[1..n]$ ein Wort. Für $2 \leq i \leq n$ definieren wir den* Z-Block an Stelle i *als das längste Präfix $T[i..i+z-1]$ von $T[i..n]$, das auch Präfix von T (also gleich $T[1..z]$) ist. Die Länge z dieses Z-Blocks nennen wir Z_i.*

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$T[i]$	a	r	a	b	a	r	a	b	a	r	a	r	t	a	r	a
2		a														
3			a	r												
4				a												
5					a	r	a	b	a	r	a	b				
6						a										
7							a	r								
8								a								
9									a	r	a	b				
10										a						
11											a	r	a			
12												a				
13													a			
14														a	r	a
15															a	
16																a
Z_i	–	0	1	0	7	0	1	0	3	0	2	0	0	3	0	1

Abbildung 5.6 Z-Blöcke und Präfixwerte Z_i für $T[1..16]$ = `arabarabarartara` (... bedeutet Übereinstimmung mit einem Präfix von T, bedeutet Mismatch).

Beachte: Wenn $i+Z_i-1 < n$, dann folgt auf den Z-Block $T[i..i+Z_i-1]$ ein Buchstabe $T[i+Z_i] \neq T[1+Z_i]$. Wenn $i+Z_i-1 = n$, dann reicht der Block an Stelle i bis ans Ende des Wortes, das heißt, dass dieser Block ein Rand von T ist. Der Z-Block an Stelle i hat Länge 0 genau dann, wenn $T[i] \neq T[1]$ gilt.

Beispiel: Abbildung 5.6 stellt die Z-Blöcke für $T[1..16] =$ `arabarabarartara` dar und gibt alle Präfixwerte an. Man erkennt, dass auf Z-Blöcke im Inneren des Wortes ein Mismatch-Buchstabe folgt, auf Z-Blöcke am Ende des Wortes nicht – sie sind Ränder von T. Man sieht auch, dass es für Z-Blöcke keine Schachtelungsregeln gibt. Jedoch kann man beobachten, dass die Blöcke für $i = 2, \ldots, 7$ im Abschnitt $T[1..7]$ dasselbe Muster bilden wie die Blöcke für $i = 6, \ldots, 11$ innerhalb des Z-Blocks $T[5..11]$.

Wir wollen die Präfixwerte $Z_2, \ldots, Z_n$ berechnen. Natürlich geht das ganz einfach in Zeit $O(n + Z_2 + \cdots + Z_n)$, indem man $T[i..n]$ mit $T[1..n-i+1]$ vergleicht, buchstabenweise von links nach rechts, für jedes $i = 2, \ldots, n$. Wenn wir mit Rechenzeit $O(n)$ auskommen wollen, müssen wir cleverer vorgehen. Aber auch dann ist das „lineare Durchmustern“ (*linear scan*) nötig und hilfreich. Wir formulieren eine entsprechende Prozedur **scan**. Gegeben sind u und v mit $1 \leq u < v \leq n+1$. Die Werte $T[u+z]$ und $T[v+z]$ werden verglichen, für $z = 0, 1, 2, \ldots$, bis entweder ein „Mismatch“ auftritt ($T[u+z] \neq T[v+z]$) oder das Wortende erreicht ist (mit $v + z = n + 1$). In beiden Fällen registrieren wir z. Offenbar ist dann z die größte Zahl in $\{0, 1 \ldots, n - v + 1\}$ mit $T[u..u+z-1] = T[v..v+z-1]$. Man beachte den Spezialfall $z = 0$, der auftritt, wenn $v \leq n$ und $T[u] \neq T[v]$ oder wenn $v = n + 1$. Der Mechanismus ist in der in Abb. 5.7 angegebenen Prozedur **scan** zusammengefasst.[6]

function scan(u, v) // Vorbedingung: $1 \leq u < v \leq n + 1$
(1) `z` $\leftarrow 0$;
(2) **while** v`+z` $\leq n \wedge$ `T[`u`+z] = T[`v`+z]` **do** `z++`;
(3) **return** `z`.

Abbildung 5.7 Direkter Buchstabenvergleich „von links nach rechts“ ab Stellen u und v mit $1 \leq u < v \leq n + 1$.

Unser Algorithmus berechnet die Werte Z_i nacheinander, in Runden $i = 2, \ldots, n$. Zur Verbesserung der Effizienz wollen wir erreichen, dass jeder Buchstabe von $T[2..n]$ höchstens einmal „erfolgreich verglichen“ wird, das heißt, als zugehörig zu einem Z-Block erkannt wird. In Abb. 5.8 wird der Ablauf veranschaulicht.

Wir betrachten Runde i, und nehmen an, dass $Z_2, \ldots, Z_{i-1}$ schon vorliegen. Weiter benötigen wir

$$r := r_{i-1} := \max(\{l + Z_l - 1 \mid 2 \leq l < i, Z_l > 0\} \cup \{0\}), \text{ für } i = 1, \ldots, n,$$

den maximalen Endpunkt eines nichtleeren Z-Blocks, der strikt links von Position i beginnt. Wenn es keinen solchen Z-Block gibt, setzen wir (künstlich) $r = r_{i-1} = 0$. (Insbesondere ist $r_1 = 0$.) Wenn $r \geq 2$ gilt, dann soll $l = l_{i-1} < i$ Startpunkt eines

[6] Wenn man dies tatsächlich programmiert, wird man zur Beschleunigung folgenden Standardtrick verwenden: Der Text T steht in einem Array der Länge mindestens $n + 1$, an Position $n + 1$ steht als „Stopper“ („*sentinel*“) ein Buchstabe, der in $T[1..n]$ nicht vorkommt. Damit entfällt der Test auf das Erreichen des Arrayendes, und Zeile (2) reduziert sich zu **while** `T[`u`+z] = T[`v`+z]` **do** `z++`;

Blocks mit Endpunkt r sein. (Der Algorithmus arbeitet immer mit dem größten solchen l – das ist aber nicht wesentlich.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	Fall
$T[i]$	a	r	a	b	a	r	a	b	a	r	a	r	t	a	r	a	
2		a															1
3			a	r													1
4				a													1
5					a	r	a	b	a	r	a	b					1
6						a											2a
7							a	r									2a
8								a									2a
9									a	r	a	b					2b
10										a							2a
11											a	r	a				2b
12												a					2a
13													a				1
14														a	r	a	1
15															a		2a
16																a	2b
Z_i	–	0	1	0	7	0	1	0	3	0	2	0	0	3	0	1	
r_i	0	0	3	3	11	11	11	11	11	11	12	12	12	16	16	16	
l_i	–	–	3	3	5	5	5	5	9	9	11	11	11	14	14	16	

Abbildung 5.8 Berechnung der Z-Blöcke und Präfixwerte Z_i für $T[1..16]$ = `arabarabarartara` durch Algorithmus 5.4.9. Dargestellt sind: Präfixwerte Z_i, Hilfszahlen r_i und l_i, Nummern der Fälle im Algorithmus. (... bedeutet erfolgreiche Buchstabenvergleiche in **scan**, bedeutet Mismatch in **scan**, ... bedeutet im Fall 2 abgeleitete Übereinstimmung, bedeutet im Fall 2 abgeleitete Mismatchposition.)

Fall 1: $r < i$.

Das bedeutet: Keine Position in $T[i..n]$ ist in einem bisher bekannten Z-Block enthalten, d. h., diese Z-Blöcke enthalten keine Information über den Bereich $T[i..n]$. Eventuell ist Stelle $T[i]$ schon einmal oder mehrmals getestet worden, aber es handelte sich dann nur um Mismatch-Ergebnisse. Durch die Festlegung $r_1 = 0$ wird erreicht, dass für die erste Runde, $i = 2$, dieser Fall eintritt.

In diesem Fall finden wir den Z-Block an Stelle i durch eine direkte Durchmusterung, nämlich durch einen Aufruf $z :=$ **scan**$(1, i)$. Dann ist $T[i..i + z - 1]$ der Z-Block an Stelle i, und es gilt $Z_i = z$.

Die Aktualisierung von r und l in Vorbereitung für die nächste Runde $i + 1$ ist auch einfach: Wenn $Z_i = 0$, ist der neu gefundene Z-Block leer, und wir behalten die alten

Werte r und l bei. (In Abb. 5.8 ist dies für $i = 2, 4, 13$ der Fall.) Wenn $Z_i > 0$, ist $T[i..i + Z_i - 1]$ der Z-Block mit Startpunkt $< i + 1$, der am weitesten nach rechts reicht, und wir setzen $r := r_i := i + Z_i - 1$ und $l := l_i := i$. (In Abb. 5.6 geschieht dies für $i = 3, 5, 14$.)

Fall 2: $r \geq i$.

Das bedeutet: Position i ist im schon bekannten nichtleeren Z-Block $T[l..r]$ enthalten. Wir benutzen dies, um bei der Ermittlung des Z-Blocks an Stelle i Buchstabenvergleiche einzusparen.

Wir haben $l < i \leq r$. Da $T[l..r]$ ein nichtleerer Z-Block ist, gilt $T[l..r] = T[1..r-l+1]$. Uns interessieren in $T[l..r]$ nur Positionen von i ab nach rechts, daher definieren wir $j := i - l + 1$ (dann ist $2 \leq j < i$) und halten fest:

$$T[i..r] = T[j..r - l + 1]. \tag{5.3}$$

Wir verfolgen nun (in Gedanken, nicht algorithmisch) $T[i + z]$, $T[j + z]$, $T[1 + z]$, für $z = 0, 1, 2, \ldots$, bis entweder ein $z \geq Z_j$ erreicht wird (dann kann man Z_i direkt ablesen, ohne einen einzigen Buchstabenvergleich) oder $i + z$ den Wert r erreicht und (5.3) nicht mehr weiterhilft (dann muss man weitere Buchstaben vergleichen). Es ergeben sich zwei Fälle.

Fall 2a: $Z_j \leq r - i$.

Dann gilt $j + Z_j \leq (i - l + 1) + (r - i) = r - l + 1$. Damit liegt der Z-Block an Stelle j inklusive seiner Mismatch-Stelle $j + Z_j$ vollständig im Teilwort $T[j..r - l + 1]$. Weil dieses nach (5.3) mit $T[i..r]$ identisch ist, gilt $T[j..j + Z_j] = T[i..i + Z_j]$. Also haben wir $T[i..i+Z_j-1] = T[j..j+Z_j-1] = T[1..Z_j]$ und $T[i+Z_j] = T[j+Z_j] \neq T[1+Z_j]$. Daraus folgt $Z_i = Z_j$, und wir haben Z_i ohne weiteren Buchstabenvergleich ermittelt. Die Werte r und l werden beibehalten, weil (offensichtlich) das rechte Ende des Blocks an Stelle i bei $i + Z_i - 1 = i + Z_j - 1 \leq i + (r - i) - 1 < r$ liegt.

Im Beispiel in Abb. 5.8 tritt Fall 2a mit $Z_i = 0$ für $i = 6, 7, 8, 10, 12, 15$ ein. Für $i = 7$ haben wir $r = r_6 = 11$ und $l = l_6 = 5$, also $T[l..r] = T[5..11] =$ `arabara`, der Wert j ist $7 - l + 1 = 3$ mit $Z_3 = 1 < 5 = r - 7 + 1$ und wir haben $T[7..8] = T[3..4]$ (= `ab`). Ohne $T[1..2]$ anzusehen, können wir schließen, dass $Z_7 = Z_3$ (= 1) gilt.

Fall 2b: $Z_j > r - i$.

Dann ist das (nichtleere) Teilwort $T[j..r - l + 1]$ vollständig im Z-Block an Stelle j enthalten, und wir erhalten:

$$T[1..r - i + 1] = T[j..r - l + 1] \overset{(5.3)}{=} T[i..r]. \tag{5.4}$$

Mit dem Aufruf $z := \textbf{scan}(r - i + 2, r + 1)$ führen wir weitere direkte Buchstabenvergleiche durch, ab Positionen $r - i + 2$ und $r + 1$ in T nach rechts gehend. Das Ergebnis z gibt die Anzahl der übereinstimmenden Buchstaben an. Also gilt nun $T[r - i + 2..r - i + z + 1] = T[r + 1..r + z]$ und ($r + z = n$ oder $T[r - i + z + 2] \neq$

$T[r+z+1]$). Zusammen mit (5.4) ergibt sich $T[1..r-i+z+1] = T[i..r+z]$. Daher gilt $Z_i = r-i+z+1$. Weil $i+Z_i-1 = r+z \geq r$ und $Z_i \geq 1$ gilt, haben wir $r_i = r+z$, und wir können $l_i = i$ setzen.

Im Beispiel in Abb. 5.8 tritt Fall 2b für $i = 9$, 11, 16 ein. Für $i = 9$ stellt sich in **scan** sofort ein Mismatch ein, für $i = 11$ verlängert sich die Folge der bekannten Buchstaben, im Fall $i = 16$ wird die Schleife in **scan** gar nicht ausgeführt, weil $r + 1 = r_{16} + 1 > 16$ ist.

Wir setzen nun diese Überlegungen in Pseudocode um. Die jeweils relevanten Werte r und l werden in Variablen `r` bzw. `l` aufbewahrt, die nur falls nötig mit neuen Werten überschrieben werden.

Algorithmus 5.4.9 (Präfixwerte)**.**
Z-Algorithmus(`T[1..n]`)
Eingabe: `T[1..n]`: Text;
Ausgabe: `Z[2..n]`: Vektor der Präfixwerte.

```
(1)   r ← 0;
(2)   for i from 2 to n do
(3)     if i > r
(4)       then      // 1. Fall
(5)         z ← scan(1, i);
(6)         Z[i] ← z;
(7)         if z > 0 then r ← i+z-1; l ← i;
(8)       else      // 2. Fall
(9)         if Z[i-l+1] < r-i+1
(10)          then      // Fall 2a
(11)            Z[i] ← Z[i-l+1];
(12)          else      // Fall 2b
(13)            z ← scan(r-i+2, r+1);
(14)            Z[i] ← r-i+z+1;
(15)            r ← r+z; l ← i;
(16)  return Z[2..n].
```

Zum besseren Verständnis führe man Algorithmus 5.4.9 mit Hilfe von Abb. 5.8 auf der Beispieleingabe `arabarabarartara` aus.

Für die Aufwandsanalyse von Algorithmus 5.4.9 überlegen wir Folgendes: Für jedes $i \in \{2, \ldots, n\}$ werden außerhalb der **scan**-Aufrufe in Zeilen (5) und (13) nur einige Zahlen berechnet und Fallunterscheidungen vorgenommen. Dies zusammen kostet $O(1)$ Zeit für jedes i. Wenn ein **scan**-Aufruf erfolgt, dann werden sequentiell Positionen $T[u+z]$ und $T[v+z]$ des Textes mit anderen verglichen, und zwar beginnt $v+z$ bei $\max\{r_{i-1}+1, i\} > r_{i-1}$. Jeder solche Aufruf liefert eventuell Übereinstimmungen („erfolgreiche Vergleiche“) und am Ende ein Mismatch. Wenn ein Aufruf sofort zu einem Mismatch führt, ist er in konstanter Zeit beendet, mit einem Vergleich. Wenn erfolgreiche Vergleiche vorkommen, liegen die übereinstimmenden Positionen $v+z$

im Bereich $\max\{r_{i-1}+1, i\}, \ldots, r_i$. Da die Schleife mit $i = 2$ beginnt und $r_n \leq n$ gilt, gibt es insgesamt maximal $n-1$ solche erfolgreichen Vergleiche, und die **scan**-Aufrufe zusammen haben Rechenzeit $O(n)$. – Wir haben damit die folgende Behauptung bewiesen.

Proposition 5.4.10. *Algorithmus 5.4.9 berechnet die Präfixwerte für $T[1..n]$ in Zeit $O(n)$ und führt dabei maximal $2n-2$ Vergleiche zwischen Buchstaben durch.* □

5.4.4.2 Anwendung der Präfixwerte

Wir zeigen hier, dass die Berechnung der Präfixwerte direkt zu einem Linearzeitalgorithmus für Textsuche führt und dass man die Randfunktion und die KMP-Fehlerfunktion eines Musters direkt aus den Präfixwerten berechnen kann.

Bemerkung 5.4.11. Der Z-Algorithmus kann unmittelbar zu einem Textsuchalgorithmus mit linearer Laufzeit ausgebaut werden. Wenn Muster $P[1..m]$ und Text $S[1..n]$ gegeben sind, setzt man $T := T[1..n+m] := P \circ S$ (Konkatenation von Muster und Text) und wendet auf T Algorithmus 5.4.9 an. Es ist leicht zu sehen, dass $S[i..i+m-1] = P[1..m]$ genau dann gilt, wenn der Z-Block in $T[1..n+m]$ an Stelle $m+i$ Länge mindestens m hat, d. h., wenn $Z_{m+i} \geq m$ ist. Man berechnet also die Präfixwerte und sucht dann die heraus, die $\geq m$ sind. Dieser Algorithmus würde (wie Algorithmus 5.4.9) alle $n+m$ Präfixwerte speichern. Eine leichte Modifikation dieser Idee erlaubt es sogar, den Platzbedarf auf $O(m+|A|)$ zu senken, wo $A = \{i \mid S[i..i+m-1] = P[1..m]\}$ die Ausgabemenge ist. Man arbeitet mit $T := T[1..n+m+1] := P \circ \# \circ S$, wobei $\#$ ein Buchstabe ist, der in $P \circ S$ nicht vorkommt. Dann gilt $P[1..m] = S[i..i+m-1]$ genau dann wenn $Z_{m+1+i} \geq m$ ist. Wegen des Fremdbuchstabens an Stelle $m+1$ kann kein Z-Block von T mehr als m Buchstaben haben. Das bedeutet, dass für den Wert j im 2. Fall in Abschnitt 5.4.4.1 die Ungleichung $j = i-l+1 \leq r-l+1 \leq m$ gilt. Daher werden in Algorithmus 5.4.9 in Zeilen (9) und (11) nur Werte $Z_2, \ldots, Z_m$ abgefragt. Die weiteren Z-Werte $Z_{m+1}, Z_{m+2}, \ldots, Z_{m+n+1}$ werden nur berechnet und mit m verglichen, aber nicht gespeichert.

Bemerkung 5.4.12. Wir betrachten ein Muster $P[1..m]$ und überlegen, wie man aus der Folge der Präfixwerte $(Z_2, \ldots, Z_m)$ für P die Randfunktion (als Tabelle `f[0..m]`) berechnen kann, ohne P nochmals anzusehen. Man erinnere sich: Für $0 \leq q' < q \leq m$ heißt $P[1..q']$ ein *Rand* von $P[1..q]$, wenn $P[1..q'] = P[q-q'+1..q]$ gilt. Die *Randfunktion* für P ist durch

$$f_{\text{bord}}(q) := f^P_{\text{bord}}(q) = \begin{cases} -1 & \text{für } q = 0 \\ \max\{q' \mid P[1..q'] \text{ ist Rand von } P[1..q]\} & \text{für } 1 \leq q \leq m \end{cases}$$

definiert. Wir wollen diese Funktion als Tabelle `f[0..m]` berechnen. Offensichtlich muss man `f[0]` $\leftarrow -1$ setzen. Ab hier betrachten wir nur noch $1 \leq q \leq m$. Da ε Rand jedes nichtleeren Wortes $P[1..q]$ ist, gilt $f_{\text{bord}}(q) \geq 0$ für $1 \leq q \leq m$.

Wir unterscheiden zwei Typen von Rändern $P[1..q']$ für ein $P[1..q]$:

Typ 1: (Nicht verlängerbar) $q = m$ oder ($q < m$ und $P[q'+1] \neq P[q+1]$).

Typ 2: (Verlängerbar) $q < m$ und $P[q'+1] = P[q+1]$.

Wir definieren:

$$f_1(q) := \max(\{0\} \cup \{q' \mid P[1..q'] \text{ ist Typ-1-Rand von } P[1..q]\}) \text{ für } 1 \leq q \leq m. \quad (5.5)$$

Wir berechnen zunächst $f_1(q)$. Betrachte dazu einen beliebigen Typ-1-Rand $P[1..q']$ von $P[1..q]$, also $P[1..q'] = P[q-q'+1..q]$. Wir möchten, dass $q - q' + 1 \leq m$ gilt. Dies ist im Fall $q < m$ sicher erfüllt. Bei $q = m$ gibt es nur eine Ausnahme, nämlich $q' = 0$. Wenn $P[1..m]$ außer ε keine weiteren Ränder hat, gilt $f_1(m) = f_{\text{bord}}(m) = 0$. Im Folgenden betrachten wir nur noch nichtleere Ränder $P[1..q']$ von $P[1..m]$. Für $i := q - q' + 1$ haben wir dann $2 \leq i \leq m$. Dass der Rand $P[1..q']$ nicht verlängert werden kann, bedeutet, dass $P[i..q]$ der Z-Block an Stelle i ist und dass $q' = Z_i$ gilt, also $i + Z_i - 1 = q$. Wir erhalten damit:

$$f_1(q) = \max(\{0\} \cup \{Z_i \mid 2 \leq i \leq m \wedge i + Z_i - 1 = q\}).$$

Um das größte Z_i zu erhalten, muss i so klein wie möglich gewählt werden.

Um die Werte $f_1(q)$ zu berechnen, könnten wir für jedes q einzeln die Folge $i = m, \ldots, 2$ durchlaufen, und im Fall $i + Z_i - 1 = q$ die Zuweisung `f[q]` $\leftarrow Z_i$ ausführen. Da das kleinste passende i, entsprechend dem größten Z_i, zuletzt getestet wird, steht am Ende das größte passende Z_i in `f[q]` (bzw. immer noch 0, wenn es kein passendes i gibt). Man kann diese Berechnung mit einem netten Trick beschleunigen: Initialisiere zunächst `f[1..m]` mit Nullen, für $1 \leq q \leq m$. Da $q = i + Z_i - 1$ aus i und Z_i berechnet werden kann, genügt die einfache Schleife

for `i` **from** m **downto** 2 **do** `f[`$i + Z_i - 1$`]` $\leftarrow Z_i$

mit Rechenzeit $O(m)$, um *alle* `f[q]` auf den Wert $f_1(q)$ zu bringen. Wenn $q < m$ gilt und $P[1..q]$ gar keinen Typ-1-Rand hat, oder wenn $q = m$ gilt und $P[1..m]$ keinen nichtleeren Rand hat, steht in `f[q]` immer noch der Initialisierungswert 0.

Nun wollen wir in einem weiteren linearen Durchlauf mit $q = m-1, \ldots, 1$ in `f[1..m]` die korrekten Werte $f_{\text{bord}}(q)$ erzeugen. Dazu überlegen wir Folgendes: Der Wert `f[m]` $= f_1(m)$ ist schon korrekt, da $P[1..m]$ keinen Typ-2-Rand hat. Für $q < m$ gibt es zwei Fälle: (1) Der längste Rand von $P[1..q]$ ist vom Typ 1. Dann gilt $f_{\text{bord}}(q) = f_1(q)$. Weiter gilt $f_{\text{bord}}(q+1) - 1 \leq f_{\text{bord}}(q) = f_1(q)$. (Jeder Rand von $P[1..q+1]$ liefert einen für $P[1..q]$ mit um 1 geringerer Länge.) (2) Der längste Rand von $P[1..q]$ hat Typ 2. Dann gilt $f_1(q) \leq f_{\text{bord}}(q) = f_{\text{bord}}(q+1) - 1$. – In beiden Fällen erhalten wir $f_{\text{bord}}(q) = \max\{f_1(q), f_{\text{bord}}(q+1) - 1\}$. Daher vervollständigt die folgende Schleife die Berechnung der Randfunktion:

for q **from** $m-1$ **downto** 1 **do** f[q] $\leftarrow \max\{$f[q], f[$q+1$] $-1\}$.

Das gesamte Verfahren zur Berechnung der Randfunktion ist in Algorithmus 5.4.13 dargestellt.

Algorithmus 5.4.13 (Randfunktion aus Präfixwerten)**.**
bord-from-prefix-numbers$(Z_2, \ldots, Z_m)$
Eingabe: $(Z_2, \ldots, Z_m)$: Präfixwerte von $P[1..m]$;
Ausgabe: f[$0..m$]: Randfunktion von $P[1..m]$ als Tabelle.
(1) f[0] $\leftarrow -1$; **for** q **from** 1 **to** m **do** f[q] $\leftarrow 0$;
(2) **for** i **from** m **downto** 2 **do** f[$i + Z_i - 1$] $\leftarrow Z_i$;
(3) **for** q **from** $m-1$ **downto** 1 **do** f[q] $\leftarrow \max\{$f[q], f[$q+1$] $-1\}$;
(4) **return** f[$0..m$].

Beispiel: Im Beispiel in Abb. 5.6 wird f[11] mit 0 initialisiert und in der Schleife in Zeile (2) erst mit $i = 9$ auf 3, dann mit $i = 5$ auf $f_1(11) = 7$ verändert. Die Schleife in Zeile (3) vergleicht f[11] mit f[12] $- 1 = 1$, das Maximum ist 7. Bei f[7] ist der Ablauf der folgende: Initialisierung mit 0, Änderung in Zeile (2) auf 1, mit $i = 7$, Änderung in Zeile (3) auf 3, weil f[8] $= 4$ ist.

Bemerkung 5.4.14. Noch viel einfacher als die Berechnung der Randfunktion gestaltet sich die Berechnung der KMP-Fehlerfunktion aus den Präfixwerten $Z_2, \ldots, Z_m$ eines Musters. Erinnerung: $f_{\mathrm{KMP}}(q)$ ist die Länge q' eines längsten Randes $P[1..q']$ von $P[1..q']$ vom Typ 1 (siehe Bem. 5.4.12). Wenn $P[1..q']$ keinen Typ-1-Rand hat, ist $f_{\mathrm{KMP}}(q) = -1$. Um diese Werte zu berechnen, gehen wir wie folgt vor: Wir initialisieren f_KMP[q] mit -1, für $0 \leq q < m$, und f_KMP[m] mit 0 (weil $P[1..m]$ immer den Typ-1-Rand ε hat). Dann korrigieren wir die Einträge, so dass f_KMP[q] die Länge q' des längsten Typ-1-Randes von $P[1..q]$ ist, wenn ein solcher überhaupt existiert. Dafür können wir, wie oben diskutiert, Zeile (2) in Algorithmus 5.4.13 benutzen. Es ergibt sich Algorithmus 5.4.15.

Algorithmus 5.4.15 (KMP-Fehlerfunktion aus Präfixwerten)**.**
KMP-from-prefix-numbers$(Z_2, \ldots, Z_m)$
Eingabe: $(Z_2, \ldots, Z_m)$: Präfixwerte von $P[1..m]$;
Ausgabe: f_KMP[$0..m$]: KMP-Fehlerfunktion von $P[1..m]$ als Tabelle.
(1) **for** q **from** 0 **to** $m-1$ **do** f_KMP[q] $\leftarrow -1$; f_KMP[m] $\leftarrow 0$;
(2) **for** i **from** m **downto** 2 **do** f_KMP[$i + Z_i - 1$] $\leftarrow Z_i$;
(3) **return** f_KMP[$0..m$].

5.4.4.3 Suffixwerte eines Textes

Gegeben sei ein Text $T[1..n]$. Spiegelverkehrt zu Präfixwerten sind „Suffixwerte“ N_k, für $1 \leq k < n$.

Definition 5.4.16. *Sei $T[1..n]$ ein Text. Für $1 \leq k < n$ definieren wir N_k als die Länge z des längsten Suffixes $T[k-z+1..k]$ von $T[1..k]$, das auch Suffix von T (also gleich $T[n-z+1..n]$) ist.*

Die Suffixwerte N_k zu berechnen ist offensichtlich dasselbe Problem wie das in Abschnitt 5.4.4.1 ausführlich betrachtete *Präfix*werteproblem, nur spiegelverkehrt. Man kann daher das Spiegelwort $T^{\mathsf{R}}[1..n]$ von T bilden, mit $T^{\mathsf{R}}[i] = T[n-i+1]$, für $1 \leq i \leq n$, und dann mit Algorithmus 5.4.9 die Z-Werte $Z_2, \ldots, Z_n$ von $T^{\mathsf{R}}[1..n]$ berechnen. Dann gilt:

$$N_k = Z_{n-k+1}, \text{ für } 1 \leq k < n.$$

Alternativ kann man eine gespiegelte Variante von Algorithmus 5.4.9 benutzen, der Indizes $i = 2, \ldots, n$ durch $k = n - i + 1$ ersetzt. Diese ist in Anhang A.1 dargestellt.

Wir geben in Abb. 5.9 zur Veranschaulichung späterer Verfahren die Suffixwerte und die zugehörigen Blöcke von $T^{\mathsf{R}}[1..16] =$ `aratrarabarabara` an, dem Spiegelbild des Wortes in Abb. 5.6.

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$T[k]$	a	r	a	t	r	a	r	a	b	a	r	a	b	a	r	a
N_k	1	0	3	0	0	2	0	3	0	1	0	7	0	1	0	–

1	a															
2		a														
3	a	r	a													
4				a												
5					a											
6				a	r	a										
7							a									
8					b	a	r	a								
9									a							
10									r	a						
11											a					
12					b	a	r	a	b	a	r	a				
13													a			
14													r	a		
15															a	

Abbildung 5.9 Suffixwerte N_k und zugehörige Blöcke für $T[1..16] =$ `aratrarabarabara` (a ... a bedeutet Übereinstimmung mit einem Suffix von T, a bedeutet Mismatch).

5.4.4.4 Die Berechnung der GS-Werte für ein Muster P

Gegeben sei das Muster $P[1..m]$. Wir nehmen an, dass die Suffixwerte N_k, $1 \leq k < m$, für P schon vorliegen. Diese werden nun geschickt benutzt, um $\mathrm{GS}(j)$ zu berechnen, für $0 \leq j \leq m$.

Man erinnere sich an Definitionen 5.4.3 und 5.4.4. Danach ist $\mathrm{GS}(j)$ die *kleinste* Zahl $s \geq 1$, die (5.1) oder (5.2) erfüllt. Zur Notationsvereinfachung schreiben wir

$$k = m - s, \text{ also } s = m - k,$$

und formen (5.1) und (5.2) um. Für jedes $j \in \{0, \ldots, m\}$ suchen wir die *größte* Zahl $k < m$, die

$$m - j < k < m \text{ und } P[j+1..m] = P[j-m+k+1..k] \text{ und } P[j] \neq P[j-m+k] \tag{5.6}$$

oder

$$0 \leq k \leq m - j \text{ und } P[m-k+1..m] = P[1..k] \tag{5.7}$$

erfüllt. Zur Illustration siehe die Abbildungen 5.11 und 5.12.

Bei einem „großen“ Shift mit $s \geq j$ gilt $0 \leq k \leq m - j$, und $P[1..k]$ ist ein Rand des Musters. Unter diesen k suchen wir das größte, das Länge $\leq m - j$ hat. Wir definieren:

$$\ell(j) := \max\{k \leq m - j \mid P[1..k] \text{ ist ein Rand von } P\}, \text{ für } 0 \leq j \leq m.$$

Der Wert $j = 0$ ist ein Sonderfall. Da ein längster Rand höchstens $m - 1$ Buchstaben haben kann, kommen auch für $j = 0$ nur Werte $k \leq m - 1$ in Frage, und es gilt $\ell(0) = \ell(1)$. Es bleibt die Aufgabe, $\ell(j)$ für $1 \leq j \leq m$ zu berechnen. Wenn $P[1..k]$ ein nichtleerer Rand von P ist, gilt $N_k = k$. (Illustration: $N_1 = 1$ und $N_3 = 3$ in Abb. 5.9.) Wir haben also:

$$\ell(j) = \max\{k \leq m - j \mid k = 0 \text{ oder } N_k = k\}, \text{ für } 1 \leq j \leq m.$$

Wir berechnen $\ell(j)$ für $j = m, m-1, \ldots, 1$ nacheinander. Bei $j = m$ ist die Sache klar: Es kommt nur der leere Rand, also $k = 0$, in Frage, also ist $\ell(m) = 0$. Für $1 \leq j < m$ gibt es zwei Möglichkeiten: Setze $k(j) = m - j$. Wenn $N_{k(j)} = k(j)$ gilt, dann ist $P[1..k(j)]$ ein Rand der Länge $k(j) = m - j$, also gilt $\ell(j) = k(j) = m - j$. Wenn aber $N_{k(j)} < k(j)$ gilt, dann ist das maximale $k \leq m - j$, für das $k = 0$ oder $N_k = k$ gilt, in Wirklichkeit sogar $\leq m - (j+1)$, d. h., wir haben $\ell(j) = \ell(j+1)$.

In Abb. 5.10 ist Pseudocode für die Berechnung der ℓ-Werte angegeben.

Nun wenden wir uns „kleinen Shifts“ mit $1 \leq s < j$ zu. Damit eine solche Shiftweite in Frage kommt, muss für $k = m - s > m - j$ die Bedingung (5.6) gelten. Wir setzen:

$$L(j) := \max(\{k \mid m - j < k < m \mid k \text{ erfüllt } (5.6)\} \cup \{0\}), \text{ für } 0 \leq j \leq m.$$

```
(1)  l[m] ← 0;
(2)  for j from m − 1 downto 1 do
(3)    if N[m-j] = m-j then l[j] ← m-j else l[j] ← l[j+1].
(4)  l[0] ← l[1].
```

Abbildung 5.10 Berechnung der ℓ-Werte

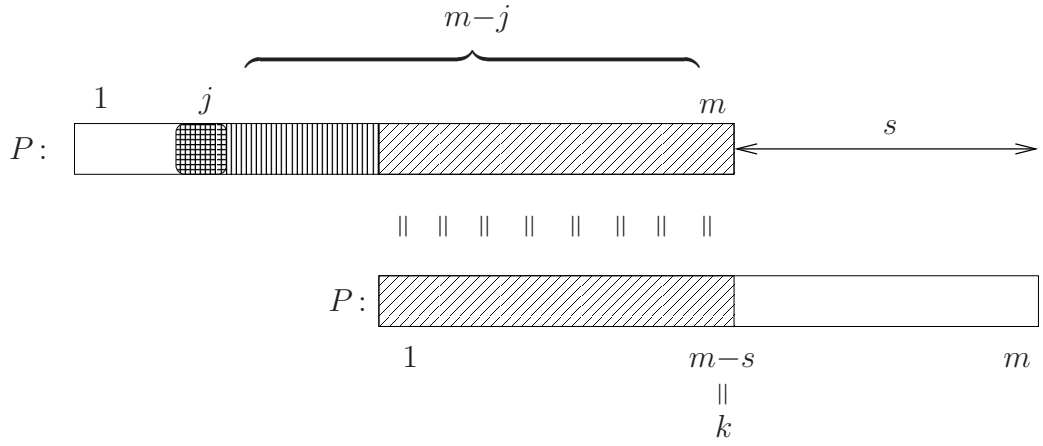


Abbildung 5.11 Ein „großer Shift“ $s = m - k \geq j$ ist zulässig nach Mismatch an Stelle j genau dann wenn $P[1..k]$ ein Rand von $P[1..m]$ der Länge $\leq m - j$ ist.

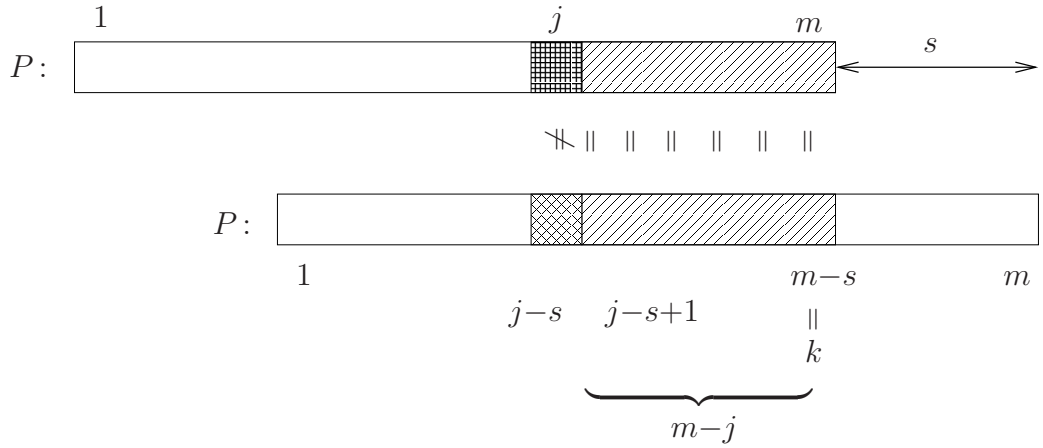


Abbildung 5.12 Ein „kleiner Shift“ $s = m - k < j$ ist zulässig nach Mismatch bei j genau dann wenn $N_k = m - j$ gilt. Eine kleinste Shiftweite s mit dieser Eigenschaft entspricht einem möglichst großen solchen $k > m - j$. Es muss kein solches k geben.

Berechnung der $L(j)$: Es kommt hier darauf an, alle $L(j)$-Werte, von denen jeder eine Maximumsbildung enthält, in Linearzeit zu berechnen. Der zentrale Trick ist folgende Beobachtung, die einen Zusammenhang zwischen den Suffixzahlen N_k und

Bedingung (5.6) herstellt, siehe auch Abb. 5.12:

Lemma 5.4.17. *Für $m - j < k < m$ gilt* (5.6) *genau dann wenn $N_k = m - j$ ist.*

Beweis: Bedingung (5.6) lautet, etwas umgeschrieben:

$$P[j+1..m] = P[k-(m-j)+1..k] \wedge P[j] \neq P[k-(m-j)].$$

Das heißt: Wenn man an der Stelle $P[k]$ startend nach links geht, findet man $m-j$ Buchstaben, die mit $P[m-j+1..m]$ übereinstimmen, der nächste Buchstabe passt nicht mehr. Nach Definition der Suffixzahlen heißt das, dass $N_k = m - j$ ist. □

Die in Abb. 5.13 angegebene Schleife berechnet die Zahlen $L(j)$, gespeichert in einem Array `L[0..m]`, aus den Suffixwerten. Der Werte $L(0) = L(1) = 0$ werden einfach gesetzt.

```
(1)   for j from 0 to m do L[j] ← 0;
(2)   for k from 1 to m − 1 do
(3)     j ← m − N[k]; L[j] ← k;
```

Abbildung 5.13 Berechnung der L-Werte

Um zu verstehen, was im Programmstück in Abb. 5.13 passiert, betrachtet man ein festes j und beobachtet den Inhalt von `L[j]`. Nach der Initialisierung ist dies 0. Für jedes k mit $N_k = m - j$, in aufsteigender Reihenfolge, wird k nach `L[j]` geschrieben. Am Ende enthält `L[j]` natürlich das maximale solche k (oder immer noch 0, wenn es kein solches k gibt), und dies ist nach der Definition von $L(j)$ und Lemma 5.4.17 genau $L(j)$.

Wir haben nun $\ell(j)$-Werte, $0 \leq j \leq m$, und $L(j)$-Werte, $0 \leq j \leq m$. Wie oben schon gesagt, ist dann

$$\mathrm{GS}(j) = m - \max\{\ell(j), L(j)\}, \quad \texttt{für } 0 \leq j \leq m.$$

Wir fassen das Ganze in einem Programm zur Berechnung der GS-Shiftwerte zusammen. Angenommen ist, dass die Suffixzahlen N_k, $1 \leq k < m$, des Musters vorliegen.

Algorithmus 5.4.18 (Berechnung der GS-Shiftwerte)**.**
BM-GS-Preprocessing(`N[1..m − 1]`)
Eingabe: `N[1..m − 1]`, die Suffixzahlen des Musters `P[1..m]`;
Ausgabe: `GS[0..m]`, die GS-Shiftwerte.

```
  // Berechnung der ℓ(j), vgl. Abb. 5.10 :
(1)   l[m] ← 0;
(2)   for k from 1 to m − 1 do
(3)     if N[k]=k then l[m-k] ← k else l[m-k] ← l[m-k+1].
```

(4) `l[0]` $\leftarrow$ `l[1]`;
// Berechnung der $L(j)$, vgl. Abb. 5.13
(5) **for** `j` **from** 0 **to** m **do** `L[j]` $\leftarrow$ 0;
(6) **for** `k` **from** 1 **to** $m-1$ **do**
(7) `j` $\leftarrow m -$ `N[k]`; `L[j]` $\leftarrow$ `k`;
// Berechnung der GS(j):
(8) **for** `j` **from** 0 **to** m **do** `GS[j]` $\leftarrow m$ - `max{l(j),L(j)}`;
(9) **return** `GS[`$0..m$`]`.

Wir betrachten den Ablauf des Algorithmus 5.4.18 auf dem Beispielwort `araratara-ratar`. Wir stellen uns vor, die N_k-Werte liegen schon vor:

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`P[`k`]`	a	r	a	r	a	t	a	r	a	r	a	t	a	r
N_k	0	2	0	2	0	0	0	8	0	2	0	0	0	–

Die $\ell(j)$-Werte, eingetragen in der Reihenfolge $j = m$, $j = m-k = m-1, m-2, \ldots, m-(m-1) = 1$, dann $\ell(0) = \ell(1)$, sind:

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\ell(j)$	8	8	8	8	8	8	8	2	2	2	2	2	2	0	0

Wir überprüfen, dass der Eintrag $\ell(9) = 2$ korrekt ist. Der längste Rand von P mit Länge höchstens $14-9 = 5$ ist `ar`, mit Länge 2. Dagegen gilt $\ell(3) = 8$, denn der längste Rand von P mit Länge höchstens $14-3 = 11$ ist `araratar`, mit Länge 8.
Nun betrachten wir die Berechnung der $L(j)$-Werte. Um zu verdeutlichen, wie diese sich verändern, werden sie untereinander geschrieben, wo eigentlich Überschreiben stattfindet. Man beachte, dass die Zahlen $k = 1, \ldots, m-1$ in dieser Reihenfolge eingetragen werden. Die Initialisierung erfolgt mit 0.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$L(j)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
							8						2		1
													4		3
													10		5
															6
															7
															9
															11
															12
															13

Eintragungen:
$k = 1$ an Position $14 - N(1) = 14 - 0 = 14$,
$k = 2$ an Position $14 - N(2) = 14 - 2 = 12$,
$k = 3$ an Position $14 - N(3) = 14 - 0 = 14$,
$k = 4$ an Position $14 - N(4) = 14 - 2 = 12$,
$k = 5$ an Position $14 - N(5) = 14 - 0 = 14$,
$\vdots$
$k = 8$ an Position $14 - N(8) = 14 - 8 = 6$,
$\vdots$

Die unterste Zahl in Spalte j ist $L(j)$. Man beachte, wie durch die Eintragungsreihenfolge automatisch das Maximum gefunden wird. Eine Ausnahme bilden die $j = m - N_k$, wo $k = m - j$ ist. Diese erfüllen $N_k = k$. Für solche j setzt das Programm `L[j]` auf k. Man beachte, dass dies auch der Wert $\ell(k)$ ist, so dass bei der Maximumsbildung nichts passiert. Im Beispiel sind $j = 6$, $k = 8$ mit $N(8) = 8$ eine solche Konfiguration.
Ergebnis durch Maximumsbildung und Komplementierung:

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\max\{\ell(j), L(j)\}$	8	8	8	8	8	8	8	2	2	2	2	2	10	0	13
$\mathrm{GS}(j)$	6	6	6	6	6	6	6	12	12	12	12	12	4	14	1

Wir können einige dieser Shiftwerte interpretieren: $\mathrm{GS}(14) = 1$ heißt: Wenn beim Vergleich im BM-Algorithmus das rechte Ende von $P[1..14]$ unter einem Buchstaben $\neq$ `r` steht, kann man nur um 1 verschieben.
$\mathrm{GS}(13) = 14$ heißt: Wenn $P[14]$ unter einem `r` steht, aber $P[13]$ nicht unter einem `a`, dann passt keine Stelle des Musters hierher (im Muster steht vor jedem `r` ein `a`), also ergibt sich ein großer Sprung von 14.
$\mathrm{GS}(9) = 12$ heißt: Wenn $P[10..14] =$ `ratar` passen, aber $P[9] =$ `a` nicht, dürfen wir um 12 Positionen schieben – die Überlappung zwischen alter und neuer Position des Musters sind nur die beiden letzten Buchstaben `ar`.
$\mathrm{GS}(5) = 6$ heißt: bei einer Übereinstimmung mit dem Suffix $P[6..14] =$ `atararatar`, und einem Fehler bei $P[5] = a$ können wir um 6 Positionen schieben, so dass das Präfix $P[1..8] =$ `araratar` an die Stelle zu stehen kommt, wo vorher das Suffix $P[7..14] =$ `araratar` stand.

Die in Algorithmus 5.4.18 gewählte Notation für die Vorverarbeitung hat den Vorteil, dass man die dahinterliegenden Ideen noch einigermaßen erkennen kann. Es ist aber unnötig, drei Arrays bereitzustellen: eines genügt vollkommen. In ihm werden zuerst die Zahlen $m - \ell(j)$, $0 \leq j \leq m$, berechnet. Hierbei spart man sich durch die Verwendung einer Variablen `ell`, die den laufenden $\ell(m-k)$-Wert enthält, Zugriffe auf das Array und Kopiervorgänge. Danach erfolgt die Korrektur für die $L(j)$-Zahlen wie in Abb. 5.13, wobei auch hier der Eintrag gleich in der komplementären Form $m - k$ erfolgt. Die Minimumsbildung erfolgt dabei automatisch, da die Werte k in aufsteigender Reihenfolge bearbeitet werden. Einträge $L(j) = 0$ können ignoriert werden. Algorithmus 5.4.19 gibt das resultierende Programm an (das natürlich nicht mehr zu verstehen ist).

Algorithmus 5.4.19 (Berechnung der GS-Shiftwerte direkt in *einem* Array)**.**
BM-GS-Preprocessing(`N[1..`$m-1$`]`)
Eingabe: `N[1..`$m-1$`]`: die Suffixzahlen des Musters `P[1..`m`]`;
Ausgabe: `GS[0..`m`]`: die GS-Shiftwerte.

```
  // Berechnung der Werte ℓ(m−k) in ell und der Werte m−ℓ(m−k) in GS[..]:
(1)   ell ← 0;
(2)   GS[m] ← m;
(3)   for k from 1 to m−1 do
(4)     if N[k]=k then ell ← k;
(5)     GS[m-k] ← m - ell;
(6)   GS[0] ← m - ell;
  // Korrektur durch die Werte m−L(j):
(7)   for k from 1 to m−1 do GS[m-N[k]] ← m - k;
(8)   return GS[0..m].
```

Anhang A

A

A.1 Direkte Berechnung von Suffixwerten

Wir geben an, wie man in Analogie zu Algorithmus 5.4.9 die Suffixwerte eines Wortes direkt berechnet, indem man es einmal von rechts nach links liest. Zunächst formulieren wir die gespiegelte **scan**-Funktion:

function invscan(u, v) // Vorbedingung: $0 \leq u < v \leq n$
(1) `z` $\leftarrow 0$;
(2) **while** u`-z` $\geq 1 \wedge$ `T[`u`-z]` $=$ `T[`v`-z]` **do** `z++`;
(3) **return** `z`.

Abbildung A.1 Direkter Buchstabenvergleich $T[u-z] \stackrel{?}{=} T[v-z]$ „von rechts nach links“ ab Stellen u und v mit $0 \leq u < v \leq n$. Die Ausgabe ist die größte Zahl $z \in \{0, 1 \ldots, u\}$ mit $T[u-z+1..u] = T[v-z+1..v]$. (Die Ausgabe $z = 0$ erscheint, wenn $u \geq 1$ und $T[u] \neq T[v]$ gilt oder wenn $u = 0$ gilt.)

Algorithmus A.1.1 (Suffixwerte eines Textes)**.**
Suffixwert-Algorithmus(`T[1..n]`)
Eingabe: `T[1..n]`: Text;
Ausgabe: `N[1..n-1]`: Vektor der Suffixwerte.

```
(1)   l ← n+1;
(2)   for k from n-1 downto 1 do
(3)     if k < l
(4)       then      // 1. Fall, gespiegelt
(5)         z ← invscan(k, n);
(6)         N[k] ← z;
(7)         if z > 0 then l ← k-z+1; r ← k;     // (Fall 1b)
(8)       else      // 2. Fall, gespiegelt
(9)         if N[n-r+k] < k-l+1
(10)          then      // (Fall 2a)
(11)            N[k] ← N[n-r+k];
(12)          else      // (Fall 2b)
(13)            z ← invscan(l-1, n-r+l-1);
(14)            N[k] ← k-l+z+1;
(15)            l ← l-z; r ← k;
(16)  return N[1..n-1].
```

Die Tabelle in Abb. A.2 gibt die Suffixwerte für den Text `araratararatar` der Länge 14 an.

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`T[`k`]`	a	r	a	r	a	t	a	r	a	r	a	t	a	r
N_k	0	2	0	2	0	0	0	8	0	2	0	0	0	–

Abbildung A.2 Die Suffixwerte für $T[1..14] =$ `araratararatar`.