Online- und Approximationsalgorithmen

release-233-pweis-2004-10-17

Diese Mitschrift zur Vorlesung “Online- und Approximationsalgorithmen” von Prof. Dr. Susanne Albers an der Universität Freiburg im Sommersemester 2004 wurde von Philipp Weis (pweis@pweis.com) angefertigt. Fast alle Grafiken und zahlreiche Korrekturen stammen von Brian Schröder.

Inhaltsverzeichnis

1 Online-Algorithmen

Bei Onlinealgorithmen treffen die Eingabedaten nach und nach im Laufe der Zeit ein. Es müssten stehts Entscheidungen getroffen werden bzw. Ausgaben berechnet werden, ohne die zukünftigen Eingaben zu kennen.

1.1 Beispiele

- **Ski-Problem:** Ein Paar Ski kann für 500€ gekauft oder für 50€ geliehen werden. Wie lange soll geliehen werden, bevor man sich für den Kauf entscheidet, wenn nicht bekannt ist, wie lange man noch Ski fährt? Die optimale Strategie ist, so lange zu leihen, bis die Leihkosten gleich den Kaufkosten sind.
- **Geldwechselproblem:** Ein Geldbetrag (z.B. 10.000€) soll in eine andere Währung (z.B. ¥) gewechselt werden. Zu welchem Zeitpunkt soll getauscht werden, wenn nicht bekannt ist, wie sich der Wechselkurs entwickelt?
- **Paging/Caching:** Es soll ein zweistufiges Speichersystem verwaltet werden, das aus einem schnellen Speicher mit kleiner Kapazität und einem langsamen Speicher mit großer Kapazität besteht. Dabei müssen Anfragen auf Speicherseiten bedient werden. Ein Seitenfehler tritt auf, falls die angefragte Seite nicht im schnellen Speicher vorhanden ist. In diesem Fall muss die Seite aus dem langsamen Speicher in den schnellen Speicher geladen werden. Ist im schnellen Speicher nicht ausreichend Platz vorhanden, muss eine Seite entfernt werden. Welche Seite wählt man am besten, um eine minimale Anzahl an Seitenfehlern zu erreichen?
- **Verteilte Systeme:** Daten sollen in einem Netzwerk dynamisch so platziert werden, dass möglichst wenige Daten übertragen werden müssen. Dabei hat jeder Netzwerk-Knoten Speicherplatz zur Verfügung. Können Datenanfragen nicht lokal bedient werden, dann entstehen Kommunikationskosten. Die Daten sollen dynamisch so platziert werden, dass die Kommunikationskosten möglichst gering sind.
- **Scheduling:** Jobs mit im Voraus bekannter Bearbeitungsdauer treffen als Eingabesequenz ein und sollen von m Maschinen bearbeitet werden. Die Jobs müssen dabei unmittelbar einer bestimmten Maschine zugeordnet werden. Hier gibt es verschiedene Optimierungsziele, ein klassisches Ziel ist die Makespan-Minimierung, also die Minimierung der Gesamtbearbeitungszeit der eintreffenden Job-Sequenz.

1.2 Deterministische Online-Algorithmen

1.2.1 Formales Modell

Ein Online-Algorithmus A muss eine Anfragesequenz $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(t)$ bedienen. Jede Anfrage muss dabei ohne Kenntnis der zukünftigen Anfragen bedient werden. Ziel ist es, die bei der Bedienung der Anfrage entstehenden Kosten zu minimieren (bzw. zu maximieren).

Bei der *kompetitiven Analyse* wird ein Online-Algorithmus A mit einem optimalen Offline-Algorithmus OPT verglichen, der die gesamte Eingabesequenz σ im Voraus kennt. Die Kosten von A auf σ werden mit $A(\sigma)$ bezeichnet, die Kosten von OPT mit $\mathrm{OPT}(\sigma)$.

DEFINITION: A heißt *c-kompetitiv*, wenn es ein a gibt, so dass für alle Eingaben σ gilt:

$$A(\sigma) \leq c \cdot \mathrm{OPT}(\sigma) + a$$

1.2.2 Scheduling

Eine online eintreffende Folge von Jobs $\sigma = J_1, \ldots, J_n$ soll auf m identische Maschinen verteilt werden, wobei p_i die Bearbeitungszeit von J_i sei. Der Makespan (Planende) ist zu minimieren.

> ALGORITHMUS: Greedy-Scheduling
>
> Plane jeden neuen Job auf der Maschine ein, die zur Zeit am geringsten ausgelastet ist.

SATZ: Greedy ist $(2 - 1/m)$-kompetitiv.

Beweis: Betrachte beliebiges $\sigma = J_1, \ldots, J_n$. T_{Greedy} sei der von Greedy erreichte Makespan.

$$m \cdot T_{\mathrm{Greedy}} \leq \sum_{i=1}^{n} p_i + (m-1) \max_{1 \leq i \leq n} p_i \qquad (*)$$

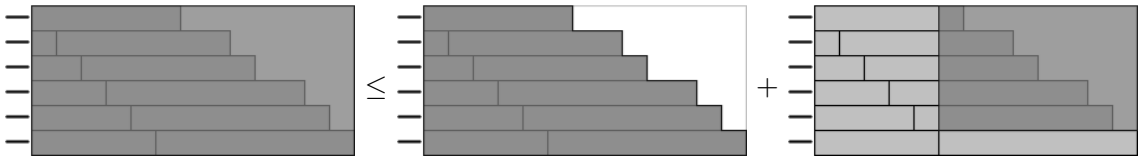


Abbildung 1: Beispiel-Schedule zu $(*)$

$$\begin{aligned} T_{\mathrm{Greedy}} &\leq \frac{1}{m} \sum_{i=1}^{n} p_i + \frac{m-1}{m} \max_{1 \leq i \leq n} p_i \\ &\leq T_{\mathrm{OPT}} + \frac{m-1}{m} T_{\mathrm{OPT}} \\ &\leq \left(2 - \frac{1}{m}\right) T_{\mathrm{OPT}} \end{aligned}$$

□

Der Faktor $(2-1/m)$ wird auch tatsächlich angenommen. Dazu kann eine günstige Eingabesequenz so gewählt werden, dass zuerst viele kleine Pakete eintreffen und am Schluss ein großes Paket eintrifft.

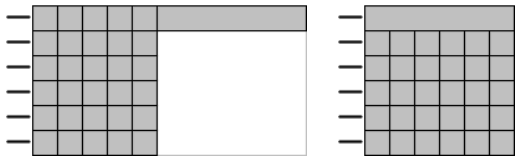


Abbildung 2: Worst Case Schedule für Greedy ($m = 6$)

Die entsprechende Sequenz ist $\sigma = \underbrace{1, \ldots, 1}_{m(m-1)\text{Mal}}, m.$

Greedy belegt hier mit den ersten $m(m-1)$ Jobs alle Maschinen mit Last $m-1$ und belegt dann eine der Maschinen zusätzlich mit Last m. OPT belegt eine Maschine mit dem Job der Länge m und die restlichen $m-1$ Maschinen mit je m Jobs der Länge 1. Der Faktor ist also:

$$c = \frac{GREEDY(\sigma)}{OPT(\sigma)} = \frac{2m-1}{m} = 2 - \frac{1}{m}$$

1.2.3 Paging

> ALGORITHMUS: MIN
>
> Entferne stets diejenige Seite aus dem Speicher, deren nächste Anfrage am weitesten in der Zukunft liegt.

SATZ: Der Algorithmus MIN ist ein optimaler Offline-Algorithmus, d.h. er erzeugt für jede Anfragesequenz σ stets die minimale Anzahl an Seitenfehlern.

Beweis: Sei σ eine beliebige Anfragesequenz und A ein Algorithmus, der auf σ eine minimale Anzahl von Seitenersetzungen erzeugt. Dazu bauen wir A so um, dass er schließlich wie MIN arbeitet. Bei diesem Umbau wird die Anzahl der Seitenersetzungen nicht erhöht.

Behauptung: Angenommen, A und MIN arbeiten auf den ersten $i-1$ Anfragen identisch, aber auf der i-ten Anfrage verschieden. Dann kann A in einen Algorithmus A' transformiert werden, so dass gilt:

- A' und MIN arbeiten auf den ersten i Anfragen identisch.
- $A'(\sigma) \leq A(\sigma)$.

Ist diese Behauptung gezeigt, dann können wir A schrittweise so transformieren, dass er identisch wie MIN arbeitet. Wegen des zweiten Teils der Behauptung erzeugt der so erhaltene Algorithmus höchstens gleiche Kosten wie A.

Wir zeigen nun die Behauptung. Die i-te Anfrage referenziere die Seite x. Um diese Anfrage zu bedienen, entferne A die Seite a aus dem schnellen Speicher und MIN die Seite v. Nun definieren wir A' so, dass er auf den ersten $i-1$ Anfragen wie A arbeitet und auf der i-ten Anfrage v entfernt. A' simuliert dann A, bis eines der folgenden Ereignisse eintritt:

a) Ersetzt A die Seite v, dann ersetzt A' zum gleichen Zeitpunkt u. Beide Algorithmen sind dann wieder in identischen Konfigurationen und A' arbeitet wie A weiter. Dabei hat sich die Anzahl der Seitenersetzungen nicht verändert.

b) u wird angefragt und A ersetzt z. In diesem Fall soll A' die Seite z durch v ersetzen. Beide Algorithmen sind dann wieder in der gleichen Konfiguration und haben die gleiche Anzahl von Seitenersetzungen erzeugt.

Der Fall, dass v angefragt wird, bevor der Fall b) eingetreten ist, kann nicht eintreten, da v erst nach der nächsten Anfrage von u erneut angefragt werden kann. Sonst hätte MIN v an dieser Stelle nicht aus dem schnellen Speicher entfernt. □

ALGORITHMUS: Least Recently Used (LRU)

Entfernt stets die Seite, die am längsten nicht mehr benutzt wurde.

SATZ: LRU ist k-kompetitiv, wobei k die Anzahl der Seiten ist, die gleichzeitig im schnellen Speicher gehalten werden können.

Beweis: Wir betrachten eine beliebige Anfragesequenz σ. LRU und OPT starten mit denselben Seiten im schnellen Speicher. Wir zerlegen σ in Phasen $P(1), P(2), \ldots$, so dass LRU in $P(i)$ mit $i \geq 2$ jeweils exakt k Seitenfehler hat und in der Phase $P(1)$ höchstens k Seitenfehler.

Nun zeigen wir, dass OPT in jeder Phase mindestens einen Seitenfehler erzeugt. In der Phase $P(1)$ ist der erste Seitenfehler bei LRU auch Fehler bei OPT, da beide mit dem gleichen Cache-Inhalt begonnen haben.

Seien $p_1, \ldots, p_k$ die Fehlerstellen von LRU in einer Phase $P(i)$ mit $i \geq 2$ und q die letzte vorangegangene Anfrage. Wir unterscheiden die folgenden Fälle.

1. $\forall i \neq j : p_i \neq p_j$ und $\forall i : p_i \neq q$: Beim Phasenwechsel hat OPT q im schnellen Speicher und kann höchstens $k-1$ Seiten aus $\{p_1, \ldots p_k\}$ im schnellen Speicher haben. Also tritt bei OPT mindestens ein Seitenfehler in dieser Phase auf.

2. $\exists i \neq j : p_i = p_j$: Zwischen den Anfragen p_i und p_j müssen mindestens k andere verschiedene Seiten angefragt worden sein, sonst hätte LRU diese Seite ja nicht aus dem schnellen Speicher entfernt. Also gibt es $k+1$ verschiedene Seiten in dieser Phase und OPT muss mindestens einen Seitenfehler haben.

3. $\forall i \neq j : p_i \neq p_j$, aber $\exists i : p_i = q$: Analog zur Argumentation bei 2. müssten zwischen der Anfrage q und der Anfrage p_i mindestens k andere verschiedene Seiten angefragt worden sein. Also hat auch hier OPT einen Seitenfehler. □

SATZ: Sei A ein deterministischer Online-Paging-Algorithmus. Ist A c-kompetitiv, dann ist $c \geq k$.

Beweis: Wir geben eine Sequenz an, die $k+1$ verschiedene Seiten $\{p_1, p_2, \ldots, p_{k+1}\}$ verwendet. Anfangs seien die Seiten $\{p_1, \ldots, p_k\}$ im schnellen Speicher. Der Gegner fragt nun stets diejenige Seite an, die bei A nicht im schnellen Speicher vorhanden ist. Bei A tritt also bei jeder Anfrage ein Seitenfehler auf, d.h. $A(\sigma) = |\sigma|$.

OPT muss höchstens alle k Anfragen eine Seite ersetzen, da nur $k+1$ Seiten vorhanden sind, und also nach einem Seitenfehler auf jeden Fall die k nächsten Seiten im Speicher sind.

Damit ist:

$$c = \frac{A(\sigma)}{\mathrm{OPT}(\sigma)} \geq \frac{|\sigma|}{\frac{|\sigma|}{k}} = k$$

□

BEMERKUNG: Wir haben angenommen, dass LRU und OPT mit der gleichen Speicherkonfiguration starten. Ist das nicht der Fall, dann kommen wir in der Kompetitivitäts-Analyse auf $\mathrm{LRU}(\sigma) \leq k \cdot \mathrm{OPT}(\sigma) + k$.

ALGORITHMUS: Most Recently Used (MRU)

Entferne diejenige Seite, die zuletzt gefragt wurde.

MRU ist ein Beispiel für eine schlechte Online-Strategie. Für die Anfragesequenz

$$\sigma = p_1, \ldots, p_k, p_{k+1}, p_k, p_{k+1}, \ldots$$

sind die Kosten nicht beschränkt.

1.2.4 Selbstorganisierende lineare Listen

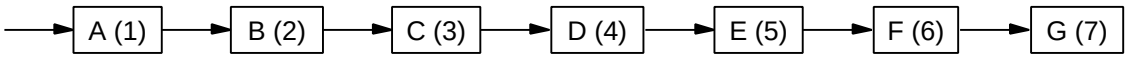


Abbildung 3: Beispiel einer linearen Liste. In Klammern stehen die Zugriffskosten.

Eine Anfragesequenz σ auf Listenelemente, die in einer unsortierten linearen Liste gehalten werden, soll bedient werden. Bei einem Zugriff auf ein Listenelement entstehen Kosten, die von der Position des Elements in der Liste abhängen. Das angefragte Element kann nach seiner Anfrage an eine beliebige Position weiter vorne in der Liste bewegt werden ("free exchange"). Außerdem ist es mit "paid exchanges" möglich, zwei benachbarte Elemente mit Kosten 1 zu vertauschen.

Die folgenden Online-Strategien bieten sich an:

- **Move-To-Front (MTF):** Das angefragte Element wird an den Kopf der Liste bewegt.
- **Transpose:** Das angefragte Element wird mit dem Vorgänger in der Liste vertauscht.
- **Frequency-Count (FC):** Verwalte für jedes Element der Liste einen Zähler der mit Null initialisiert wird. Bei jeder Anfrage wird der Zähler um eins erhöht. Die Liste wird nach jeder Anfrage gemäß nicht steigenden Zählerständen sortiert.

SATZ: MTF ist 2-kompetitiv.

Beweis: Wir zeigen mit amortisierter Analyse, dass $A(\sigma) \leq 2\,\mathrm{OPT}(\sigma)$ ist. Die Gleichung $A(\sigma(t)) \leq c \cdot OPT(\sigma(t))$ ist hier natürlich nicht für alle Zeitpunkte t erfüllbar.

Wir verwenden die Potentialfunktion Φ, die eine Konfiguration von A und OPT auf eine reelle Zahl abbildet, wobei $\forall t : \Phi(t) \geq 0$ und $\Phi(0) = 0$ gilt. Die tatsächlichen Onlinekosten bezeichnen wir mit $A(\sigma(t))$, die amortisierten Kosten sind definiert durch $A(\sigma(t)) + \Phi(t) - \Phi(t-1)$.

Wir wollen zeigen, dass für alle Zeitpunkte t gilt:

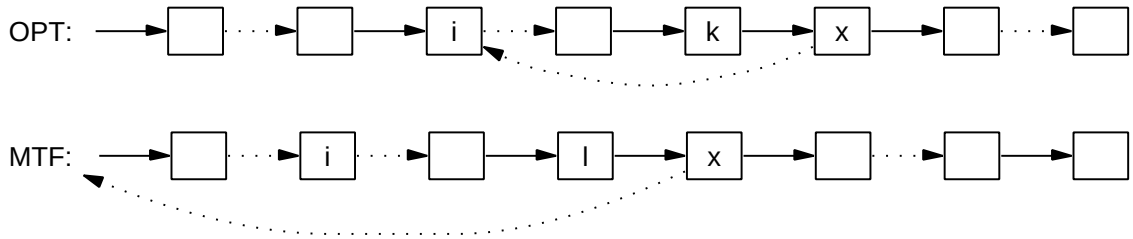
$$A(\sigma(t)) + \Phi(t) - \Phi(t-1) \leq c \cdot \mathrm{OPT}(\sigma(t))$$

Damit ist A c-kompetitiv, denn durch Summieren über alle t ergibt sich:

$$\begin{aligned} \sum_{t=1}^{m} (A(\sigma(t)) + \Phi(t) - \Phi(t+1)) &\leq c \sum_{t=1}^{m} \mathrm{OPT}(\sigma(t)) \\ A(\sigma) + \Phi(m) - \Phi(0) &\leq c\ \mathrm{OPT}(\sigma) \\ A(\sigma) &\leq c\ \mathrm{OPT}(\sigma) - \Phi(m) \end{aligned}$$

Wir wählen als Potentialfunktion Φ die Anzahl der Inversionen in der Liste, wobei eine Inversion ein geordnetes Paar (x, y) von Listenelementen ist, so dass x vor y in OPT's Liste ist und x hinter y in MTF's Liste ist.

Sei $\sigma(t) = x$. x befinde sich direkt nach k in OPTs Liste und direkt nach l in MTFs Liste. Außerdem verschiebe OPT das Listenelement x nach vorne, so dass es direkt nach dem Listenelement i eingefügt wird.



Durch OPTs Verschieben von x nach vorne können höchstens $k - i$ neue Inversionen entstehen. Durch MTFs Verschieben von x werden anschließend $l - i$ Inversionen aufgehoben und höchstens i neue Inversionen erzeugt.

$$\begin{aligned} \mathrm{MTF}(\sigma(t)) + \Delta\Phi(t) &\leq (l+1) + (k-i) + (i - (l-i)) \\ &= k + i + 1 \\ &\leq 2k + 1 \\ &< 2(k+1) \\ &= 2 \cdot \mathrm{OPT}(\sigma(t)) \end{aligned}$$

□

Satz: Sei A ein deterministischer Online-Algorithmus für selbstorganisierende Listen. Ist A c-kompetitiv, dann ist $c \geq 2$.

Beweis: Sei eine Liste mit n Elementen gegeben. Der Gegner erzeugt eine Anfragesequenz σ, in der stets das letzte Element in A's Liste angefragt wird. Für $|\sigma| = m$ ist $A(\sigma) = mn$.

Wir betrachten die Offline-Strategie SL, die σ mit einer statischen Liste beantwortet, in der die Elemente nach nicht-steigenden Häufigkeiten sortiert sind. Sei m ein Vielfaches von n. Der Worst-Case für SL ist, dass jedes Element gleich häufig angefragt wird. Dann entstehen für $i = 1 \dots m$ je $\frac{m}{n}$ mal Kosten i.

$$\mathrm{OPT}(\sigma) \leq \underbrace{\sum_{i=1}^{n} i \frac{m}{n}}_{\substack{\text{Bedienen der Anfragen} \\ \text{im Worst Case}}} + \underbrace{\binom{n}{2}}_{\substack{\text{paid exchanges} \\ \text{für das Umsortieren}}} = \frac{n+1}{2} \cdot m + \frac{n(n-1)}{2}$$

$$c = \frac{A(\sigma)}{\mathrm{OPT}(\sigma)} \geq \frac{mn}{\frac{n+1}{2}m + \frac{n(n-1)}{2}} \overset{m \to \infty}{\longrightarrow} \frac{2n}{n+1} = 2 - \frac{2}{n+1}$$

Für große n folgt damit das Ergebnis. Eine genauere Analyse ohne Grenzwertbildung, die allerdings erheblich aufwändiger ist, ist auch möglich. □

Satz: Transpose ist für keine Konstante c c-kompetitiv.

Beweis: Man betrachte die Liste $1, \dots, n$ und die Anfragesequenz $\sigma = n, n-1, n, n-1, \dots, n, n-1$ mit $|\sigma| = m$. Dann entstehen für Transpose Kosten von $\mathrm{Transpose}(\sigma) = m \cdot n$.

Mit MTF entstehen auf der selben Anfrage Kosten von

$$\mathrm{MTF}(\sigma) = n + n + 2(m-2) = 2n + 2m - 4 \quad .$$

Für $n > m$ ist

$$c = \frac{\mathrm{Transpose}(\sigma)}{\mathrm{OPT}(\sigma)} \geq \frac{\mathrm{Transpose}(\sigma)}{\mathrm{MTF}(\sigma)} = \frac{mn}{2n + 2m - 4} \geq \frac{mn}{4(n-1)} \geq \frac{m}{4} \quad ,$$

also abhängig von der Länge der Anfragesequenz. □

Satz: FC ist für keine Konstante c c-kompetitiv.

Beweis: Angenommen, FC wäre c-kompetitiv mit $c > 1$. Dann betrachten wir eine Liste mit den Elementen $p_1, \dots, p_{4l}$ mit $l = \lceil c \rceil$ und die Anfragesequenz

$$\sigma = {p_1}^{2l}, {p_2}^{2l}, \dots, {p_{4l}}^{2l} \quad .$$

Die Haufigkeitszähler von FC seien anfangs für alle Listenelemente mit Null initialisiert. Bei allen Anfragen an das Element p_i ist p_i nicht weiter vorne in der Liste als an Position i, da alle p_j mit $j < i$ bereits öfter angefragt wurden. Also gilt

$$\mathrm{FC}(\sigma) \geq \sum_{i=1}^{4l} 2l \cdot i = 2l \cdot \frac{4l(4l+1)}{2} > 16l^3 \quad .$$

MTF muss bei der ersten Aufrage auf p_i höchstens $4l$ bezahlen, bei den folgenden Anfragen auf p_i fallen nur noch Kosten von 1 an. Also gilt

$$\mathrm{MTF}(\sigma) \leq \sum_{i=1}^{4l} i + (2l-1)(4l) = \frac{4l(4l+1)}{2} + 8l^2 - 4l = \frac{16l^2 + 4l}{2} + 8l^2 - 4l = 16l^2 - 2l < 16l^2$$

Also ist

$$c = \frac{\mathrm{FC}(\sigma)}{\mathrm{OPT}(\sigma)} \geq \frac{\mathrm{FC}(\sigma)}{\mathrm{MTF}(\sigma)} > \frac{16l^3}{16l^2} = l = \lceil c \rceil$$

Somit erzeugt FC mehr als das c-fache der Kosten von MTF und ist damit nicht c-kompetitiv. □

1.2.5 Paging: Alternative Modelle

Die theoretischen Kompetitivitätsfaktoren sind wesentlich höher als die experimentell in der Praxis beobachteten Faktoren. Für LRU bzw. FIFO ergeben sich experimentelle Faktoren zwischen 1 und 4. Dabei ist LRU wesentlich besser als FIFO. Ursache dafür ist, dass die erzeugten Anfragesequenzen nicht beliebig sind, sondern *Anfragelokalität* (*Locality of Reference*) aufweisen, d.h. die Teilsequenzen einer Anfragesequenz referenzieren nur "relativ wenige" verschiedene Seiten.

Möglichkeiten für eine bessere Modellierung:

- **Zugriffsgraphen (1991):** Die Knoten repräsentieren Speicherseiten, wobei die Anfragesequenz konsistent mit dem Graphen sein muss, d.h. dass nach einer Anfrage auf eine Seite p nur Seiten angefragt werden, die adjazent zu p sind. Die Vorgehensweise bei der kompetitiven Analyse entspricht in diesem Modell einem vollständigen Graphen.

  Bisherige Ergebnisse.

  - Für alle Bäume T: $c_{\mathrm{LRU}}(T)$ ist optimal.
  - Für alle Graphen G: $c_{\mathrm{LRU}}(G) \leq c_{\mathrm{FIFO}}(G)$.

  Eine Quanitifizierung von $c_A(G)$ ist schwer.

  > ALGORITHMUS: Markierungs-Algorithmus FAR
  >
  > Bei einem Fehler wird eine unmarkierte Seite aus dem schnellen Speicher entfernt, die im Zugriffsgraphen am weitesten von der aktuellen Anfrage entfernt ist. $c_{\mathrm{FAR}}(G)$ ist optimal. Probleme dabei sind allerdings, dass G nicht bekannt ist und FAR sehr rechenaufwändig ist.

- **Markovketten:** n Speicherseiten

$$Q = \begin{pmatrix} q_{1,1} & \cdots & q_{1,n} \\ \vdots & & \vdots \\ q_{n,1} & \cdots & q_{n,n} \end{pmatrix}$$

  $q_{i,j}$ ist die Wahrscheinlichkeit, dass die Seite j als nächstes angefragt wird, wenn gerade die Seite i referenziert wurde. Eine Anfragesequenz σ wird dann entsprechend der Matrix Q erzeugt. Als Gütemaß setzt man die Seitenfehlerrate von A ein, also

$$\frac{\#\text{Seitenfehler von } A}{|\sigma|} \quad .$$

  Bestimme für gegebene A und Q die erwartete Seitenfehlerrate.

- **Working-Set-Modell (Denning 1968):** Idee: Anfragelokalität.

  Wir betrachten konkave, monoton steigende Funktionen, die einer Fensterlänge n die maximale Anzahl verschiedener Seiten in einer Teilanfrage der Länge n zuordnen.

  Beobachtung: Das Programm arbeitet zu jedem Zeitpunkt mit wenigen verschiedenen Seiten.

  Eine Anwendung wird charakterisiert durch eine konkave Funktion f. Wir betrachten Anfragesequenzen, die konsistent mit f sind, d.h. dass die Anzahl verschiedener Seiten in Fenstern der Länge n höchstens $f(n)$ ist.

  Wir betrachten die Worst-Case-Seitenfehlerrate. Für alle f konkav und monoton steigend ist die Seitenfehlerrate von LRU optimal. Außerdem gibt es konkave und monoton steigende f, für die die Fehlerrate von LRU deutlich besser ist als die von FIFO.

1.3 Randomisierte Online-Algorithmen

Ein randomisierter Online-Algorithmus darf Zufallsentscheidungen treffen.

Beispiel zum Paging: Entferne eine zufällig gemäß Gleichverteilung gewählte Seite.

Beispiel bei den selbstorganisierenden Listen: Bewege das angefragte Element an eine zufällig gewählte Position weiter vorne in der Liste.

1.3.1 Formales Modell

A sei ein randomisierter Online-Algorithmus. Dann ist $A(\sigma)$ eine Zufallsvariable. Wir wollen $E[A(\sigma)]$ bestimmen. Der kompetitive Faktor ist hier bezüglich eines Gegners definiert, der die Eingabesequenz σ konstruiert und sie auch bedienen muss. Der Gegner kennt bei der Konstruktion von σ die Beschreibung von A

Es werden zwei Klassen von Gegnern unterschieden:

Definition: Ein *blinder Gegner* sieht die Ergebnisse der Zufallsentscheidungen nicht, d.h. σ wird als Ganzes erzeugt. Ein Algorithmus A heißt c-kompetitiv gegen einen blinde Gegner, wenn eine Konstante a existiert, so dass für alle vom blinden Gegner erzeugten σ gilt:

$$E[A(\sigma)] \leq c \cdot OPT(\sigma) + a$$

Definition: Ein *adaptiver Gegner* sieht bei der Konstruktion von $\sigma(t)$ die Ergebnisse der ersten $t-1$ Zufallsentscheidungen.

Es wird zusätzlich unterschieden zwischen einem *adaptiven Online-Gegner*, der die erzeugte Anfragen online bedienen muss, und einem *adaptiven Offline-Gegner*, der die erzeugte Anfragesequenz offline bedienen darf.

A heißt c-kompetitiv gegen einen adaptiven Online-Gegner ADV, wenn es eine Konstante a gibt, so dass für alle vom Gegner erzeugten σ gilt:

$$E[A(\sigma)] \leq c \cdot E[\mathrm{ADV}(\sigma)] + a$$

A heißt c-kompetitiv gegen einen adaptiven Offline-Gegner ADV, wenn es eine Konstante A gibt, so dass für alle vom Gegner erzeugten σ gilt:

$$E[A(\sigma)] \leq c \cdot E[\mathrm{ADV}(\sigma)] + a. = c \cdot E[\mathrm{OPT}(\sigma)] + a$$

BEMERKUNG: Es gilt $c_{\text{blinder Gegner}} \leq c_{\text{adaptiver Online-Gegner}} \leq c_{\text{adaptiver Offline-Gegner}}$.

SATZ: $(*)$ Sei A ein randomisierter Online-Algorithmus, der c-kompetitiv gegen adaptive Offline-Gegner ist. Dann existiert auch ein c-kompetitiver deterministischer Algorithmus.

SATZ: $(**)$ Sei A ein randomisierter Online-Algorithmus, der c-kompetitiv gegen adaptive Online-Gegner ist. Gibt es auch noch einen d-kompetitiven Algorithmus gegen blinde Gegner, dann ist A (cd)-kompetitiv gegen adaptive Offline-Gegner.

KOROLLAR: Ist A c-kompetitiv gegen adaptive Online-Gegner, dann existiert ein c^2-kompetitiver deterministischer Algorithmus.

Beweis: Nach $(**)$ existiert ein gegen adaptive Offline Gegner c^2-kompetitiver randomisierter Algorithmus. Nach $(*)$ existiert dann auch ein deterministischer Algorithmus. □

1.3.2 Paging

Als Nächstes untersuchen wir die randomisierte Strategie RANDOM für das Paging-Problem.

> ALGORITHMUS: RANDOM
>
> Entferne bei jedem Seitenfehler eine zufällig gemäß Gleichverteilung gewählte Seite aus dem schnellen Speicher.

SATZ: RANDOM ist nicht besser als k-kompetitiv gegen blinde Gegner.

Beweis: (*Skizze*) Wähle für $N \gg k$ die folgende Eingabesequenz:

$$\sigma = \underbrace{b_0 a_1 \dots a_{k-1}}_{P_0} \underbrace{(b_1 a_1 \dots a_{k-1})^N}_{P_1} \underbrace{(b_2 a_1 \dots a_{k-1})^N}_{P_2} \dots$$

Da in jeder Phase P_i die Wahrscheinlichkeit für das Auslagern der Seite b_i bei jedem Zugriff k^{-1} ist, macht RANDOM in jeder Phase im Erwartungswert k Fehler, bis er sich von b_i trennt. □

ALGORITHMUS: MARKING

MARKING ist ein randomisierter Algorithmus, bei dem die Anfragesequenz σ in Phasen bedient wird. Zu Beginn einer Phase sind alle Seiten unmarkiert. Wird eine Seite angefragt, so wird sie markiert. Bei einem Seitenfehler wird eine zufällig gemäß Gleichverteilung gewählte unmarkierte Seite entfernt. Eine Phase endet unmittelbar vor einem Seitenfehler, wenn alle Seiten im schnellen Speicher markiert sind.

SATZ: MARKING ist $2H_k$-kompetitiv gegen blinde Gegner, wobei $H_k = \sum_{i=1}^{k} \frac{1}{i}$ die k-te harmonische Zahl ist.

BEMERKUNG: Es gilt $\ln(k+1) \leq H_k \leq \ln(k) + 1$.

Beweis des Satzes: Sei σ eine beliebige Anfragesequenz. Der Algorithmus MARKING zerlegt σ in Phasen $P(1), P(2), \ldots, P(m)$. Jede Phase $P(i)$ enthält genau k verschiedene Seiten. Die erste Anfrage von $P(i)$ ist verschieden von allen Seiten in $P(i-1)$. Für die Phase $P(i)$ sei n_i die Anzahl der neuen Seiten in $P(i)$, wobei eine Seite neu in $P(i)$ heißt, falls sie in $P(i)$, aber nicht in der vorangegangenen Phase $P(i-1)$ angefragt wurde.

Wir zeigen, dass OPT amortisierte Kosten von mindestens $n_i/2$ erzeugt, und MARKING mit erwarteten Kosten von $n_i \cdot H_k$ auskommt. Dann ergibt sich als Kompetitivitätsfaktor

$$c = \frac{\text{MARKING}(\sigma)}{\text{OPT}(\sigma)} \geq \frac{n_i H_k}{n_i/2} = 2H_k \quad .$$

Wir betrachten zwei aufeinanderfolgende Phasen. Die Phasen $P(i)$ und $P(i+1)$ enthalten zusammen genau $k + n_i$ paarweise verschiedene Seiten. OPT macht also in diesem Phasenpaar mindestens n_i Fehler. Es ergibt sich also

$$OPT \geq n_2 + n_4 + n_6 + \ldots = \sum_{i \text{ gerade}} n_i \quad .$$

Betrachtet man die um eine Phase verschobenen Phasenpaare, dann ergibt sich

$$OPT \geq n_1 + n_3 + n_5 + \ldots = \sum_{i \text{ ungerade}} n_i \quad .$$

Insgesamt erhalten wir damit als amortisierte Kosten

$$2OPT \geq \sum_{i \text{ gerade}} n_i + \sum_{i \text{ ungerade}} n_i$$

$$OPT \geq \frac{1}{2} \sum_{i} n_i \quad .$$

MARKING erzeugt in der Phase $P(i)$ genau n_i Seitenfehler für die Anfragen auf neue Seiten. Es kann aber auch Seitenfehler für Seiten geben, die nicht neu in der Phase $P(i)$ sind. Die Anzahl der in der Phase $P(i)$ angefragten alten Seiten bezeichnen wir mit o_i. Eine Seite soll dabei alt heißen, wenn sie in der Phase $P(i-1)$ angefragt wurde, aber in $P(i)$ noch nicht angefragt wurde oder in $P(i)$ überhaupt nicht angefragt wird. Es gibt also zu Beginn einer Phase genau k alte Seiten und es gilt $o_i + n_i = k$.

Wir betrachten die j-te Anfrage an eine alte Seite, wobei $1 \leq j \leq o_i$ gilt. Unmittelbar vor der Anfrage auf die j-te alte Seite enthält der Cache n_i^j neue Seiten, die bisher gefragt worden sind. Dabei gilt $n_i^j \leq n_i$. Es gibt $k-(j-1)$ alte Seiten, die vor der j-ten Anfrage noch nicht angefragt wurden. Von diesen alten Seiten sind n_i^j nicht mehr im schnellen Speicher. Die Wahrscheinlichkeit für einen Seitenfehler ist also

$$P(\text{“Seitenfehler zu Zeitpunkt j”}) = \frac{n_i^j}{k-(j-1)} \leq \frac{n_i}{k-(j-1)} \quad .$$

Da $n_i \geq 1$ gilt $o_i \leq k-1$ und damit $\frac{1}{k-o_o+1} \leq \frac{1}{2}$. Als erwartete Kosten für die alten Seiten ergibt sich also

$$\begin{aligned}\sum_{j=1}^{o_i} \frac{n_i}{k-(j-1)} &= n_i \left(\frac{1}{k} + \frac{1}{k-1} + \frac{1}{k-2} + \ldots + \frac{1}{k-o_i+1} \right) \\ &\leq n_i \left(\frac{1}{k} + \ldots + \frac{1}{2} \right) = n_i(H_k - 1) \quad .\end{aligned}$$

Die Gesamtkosten in einer Phase $P(i)$ können also durch

$$\text{cost}(P(i)) \leq n_i + n_i(H_k - 1) = n_i H_k$$

abgeschätzt werden. □

Satz: Sei A ein randomisierter Online-Paging-Algorithmus, der c-kompetitiv gegen blinde Gegner ist. Dann ist $c \geq H_k$.

Beweis: Sei A ein Online-Paging-Algorithmus. Die Menge der angefragten Seiten sei $\{p_1, \ldots, p_{k+1}\}$. Der blinde Gegner kennt die Beschreibung von A, nicht aber die Ergebnisse der Zufallsentscheidungen. Er verwaltet einen Vektor von Wahrscheinlichkeitswerten

$$Q = (q_1, \ldots, q_{k+1}) \quad .$$

q_i ist die Wahrscheinlichkeit, dass die Seite p_i nicht im schnellen Speicher von A ist. Es gilt $\sum_{i=1}^{k+1} q_i = 1$. Die Einträge im Vektor Q werden vom Gegner in jedem Schritt aktualisiert.

Wir unterteilen die Anfragesequenz σ in Phasen, die jeweils aus k Subphasen bestehen. Der Gegner markiert die angefragten Seiten wie bei MARKING und unterteilt σ in entsprechende Phasen. Mit jeder neu markierten Seite beginnt eine neue Subphase. In einer Subphase j wollen wir für A erwartete Kosten von $1/(k+1-j)$ erzeugen, so dass sich die Kosten pro Phase für A berechnen zu

$$\sum_{j=1}^{k} \frac{1}{k+1-j} = \frac{1}{k} + \frac{1}{k-1} + \ldots + 1 = H_k \quad .$$

Der Gegner soll pro Phase Kosten von 1 erzeugen. Dann gilt $c = {}^{H_k}/_1 = H_k$.

Zu Beginn einer Subphase j gibt es j markierte Seiten. Das sind gerade die Seiten, die in den letzten $j-1$ Subphasen angefragt worden sind.

Sei M die Menge der markierten Seiten. Wir setzen

$$\gamma = \sum_{p_i \in M} q_i$$

und unterscheiden zwei Fälle.

Ist $\gamma = 0$, dann gibt es $k + 1 - j$ unmarkierte Seiten, von denen wir eine unmarkierte Seite p_i mit $q_i \geq 1/(k + 1 - j)$ auswählen. Diese Seite wird vom Gegner angefragt und markiert. Damit endet die Subphase j.

Ist $\gamma > 0$, dann gibt es ein $p_i \in M$ mit $q_i = \varepsilon > 0$. Der Gegner fragt p_i an und erzeugt somit bei A erwartete Kosten von ε. Solange die erwarteten Kosten für die Subphase j kleiner als $1/(k+1-j)$ sind und $\gamma = \sum_{p_i \in M} q_i > \varepsilon$ ist, fragt der Gegner die markierte Seite $p_i \in M$ mit dem größten q_i an. Nach dem Ende dieser Schleife fragen wir die unmarkierte Seite p_i mit größtem q_i an und markieren sie, womit die Subphase j beendet wird.

Wird die Schleife verlassen, dann sind die erwarteten Kosten entweder mindestens $1/(k + 1 - j)$ oder es gilt $\gamma \leq \varepsilon$, womit für die erwarteten Kosten am Ende der Schleife gilt:

$$\varepsilon + \frac{1 - \sum_{p_i \in M} q_i}{k+1-j} = \varepsilon + \frac{1-\gamma}{k+1-j} \geq \varepsilon + \frac{1-\varepsilon}{k+1-j} \geq \frac{\varepsilon}{k+1-j} + \frac{1-\varepsilon}{k+1-j} = \frac{1}{k+1-j} \quad .$$

Am Phasenende bleibt die zuletzt markierte Seite für die neue Phase markiert.

Am Anfang einer Phase entfernt der Gegner gerade die Seite aus dem schnellen Speicher, die zu Beginn der folgenden Phase angefragt wird. So erreicht er Kosten von 1 pro Phase. □

Satz: RANDOM ist k-kompetitiv gegen adaptive Online-Gegner.

Beweis: Wir machen eine amortisierte Analyse und setzen die Potentialfunktion $\Phi := k \cdot |S_R \setminus S_{\text{ADV}}|$, wobei S_R bzw. S_{ADV} die Menge der Seiten ist, die RANDOM bzw. ADV im schnellen Speicher hat. Wir zeigen für alle Anfragen $\sigma(t)$, dass

$$E[R(\sigma(t)) + \Phi(t) - \Phi(t-1)] \leq k \cdot \text{ADV}(\sigma(t)) \quad .$$

Damit folgt

$$R(\sigma(t)) + E[\Phi(t)] - E[\Phi(t-1)] \leq k \cdot \text{ADV}(\sigma(t)) \quad .$$

Summieren wir über alle t, dann ergibt sich

$$R(\sigma) + E[\Phi(m)] - E[\Phi(0)] \leq k \cdot \text{ADV}(\sigma)$$

und damit

$$R(\sigma) \leq k \cdot \text{ADV}(\sigma(t)) - E[\Phi(m)] \leq k \cdot \text{ADV}(\sigma) \quad .$$

Sei $x = \sigma(t)$. Wir unterscheiden die folgenden Fälle.

1. $x \in S_R$ und $x \in S_{\text{ADV}}$: Keine Kosten und keine Potentialänderung.
2. $x \in S_R$ und $x \notin S_{\text{ADV}}$: R hat keine Kosten, der Gegner hat Kosten 1. x muss vom Gegner in den schnellen Speicher genommen werden. Eventuell wird dafür eine Seite, die auch in S_R ist, aus dem schnellen Speicher von ADV entfernt. Es gilt also $\Delta\Phi \leq 0$ und damit

$$R(x) + E[\Delta\Phi] \leq 0 \leq k \cdot \text{ADV}(x) = k \quad .$$

3. $x \notin S_R$ und $x \in S_{\text{ADV}}$: R hat Kosten 1, der Gegner dagegen keine Kosten. Die Menge $S_R \setminus S_{\text{ADV}}$ enthält mindestens eine Seite. Es gibt also zwei Fälle: (1) Wird diese Seite von S_R ausgelagert, dann sinkt das Potential um k. Die Wahrscheinlichkeit dafür ist mindestens $1/k$. (2) Wird eine andere Seite von R ausgelagert, dann kann das Potential nur gleichbleiben oder sinken. Somit erhalten wir $E[\Delta\Phi] \leq -k \cdot 1/k + 0 \cdot (1 - 1/k) = -1$. Es gilt also

$$R(x) + E[\Delta\Phi] = 1 - 1 = 0 = k \cdot \text{ADV}(x) \quad .$$

4. $x \notin S_R$ und $x \notin S_{\text{ADV}}$: Der Gegner hat Kosten 1. Er nimmt x in den schnellen Speicher und entfernt eine andere Seite y. Dadurch kann $|S_R \setminus S_{\text{ADV}}|$ um 1 steigen. Damit ist $\Delta\Phi \leq k$. Das Verhalten von R entspricht Fall 3. Die erwartete Potentialänderung ist hier -1. Also ergibt sich insgesamt $E[\Delta\Phi] \leq k - 1$ und damit

$$R(x) + E[\Delta\Phi] \leq 1 + k - 1 = k \cdot \text{ADV}(x) \quad .$$

□

1.3.3 Selbstorganisierende Listen

Satz: Sei A ein randomisierter Online-Algorithmus für selbstorganisierende Listen, der c-kompetitiv gegen adaptive Online-Gegner ist. Dann ist $c \geq 2$.

Beweis: Sei A ein randomisierter Online-Algorithmus, n die Länge der Liste und σ eine Anfragesequenz. Der Gegner fragt stets das letzte Listenelement an. Damit erzeugt A bei jeder Anfrage Kosten von n. Der Gegner arbeitet mit einer zufällig gemäß Gleichverteilung gewählten statischen Liste.

Die erwarteten Kosten des Gegners für eine Anfrage sind dann

$$\sum_{i=1}^{n} \frac{1}{n}\, i = \frac{n+1}{2} \quad .$$

Damit ist

$$c \geq \frac{n}{\frac{n+1}{2}} = \frac{2n}{n+1} = 2 - \frac{2}{n+1} \quad .$$

Wir können also zu jedem Online-Algorithmus A und zu jedem $c < 2$ eine Anfragesequenz angeben, auf die A mehr als das c-fache der optimalen Kosten benötigt. Also gilt $c \geq 2$. □

> Algorithmus: BIT
>
> Halte für jedes Listenelement x ein Bit $b(x)$. Anfangs werden diese Bits zufällig und unabhängig voneinander mit 0 oder 1 initialisiert. Bei jeder Anfrage x wird $b(x)$ komplementiert. Wechselt $b(x)$ dabei auf 1, so bewege x an den Kopf der Liste, andernfalls wird die Listenposition von x nicht geändert.

SATZ: BIT ist 1.75-kompetitiv gegen blinde Gegner.

Beweis: Sei σ beliebig. Zu jedem Zeitpunkt von σ und für jedes x ist $b(x)$ mit gleicher Wahrscheinlichkeit 0 bzw 1. Es gilt:

$$b(x) = (\text{Startwert} + \#\{\text{Anfragen auf } x\}) \mod 2$$

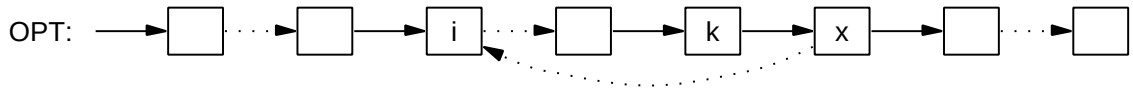
Wir führen eine amortisierte Analyse durch, wobei wir die Anzahl der Inversionen betrachten (siehe MTF-Analyse auf Seite 7). Bei einer Inversion (x, y) ist x vor y in OPT's Liste, aber y vor x in BIT's Liste. Wir unterscheiden Inversionen, bei denen $b(x) = 0$ ist (Typ 1) und bei denen $b(x) = 1$ ist (Typ 2). Als Potentialfunktion verwenden wir

$$\Phi = \#\{\text{Inversionen von Typ 1}\} + 2 \cdot \#\{\text{Inversionen von Typ 2}\} \quad .$$

Wir wollen zeigen:

$$\text{BIT}(\sigma(t)) + E[\Phi(t) - \Phi(t-1)] \leq 1.75 \text{ OPT}(\sigma(t))$$

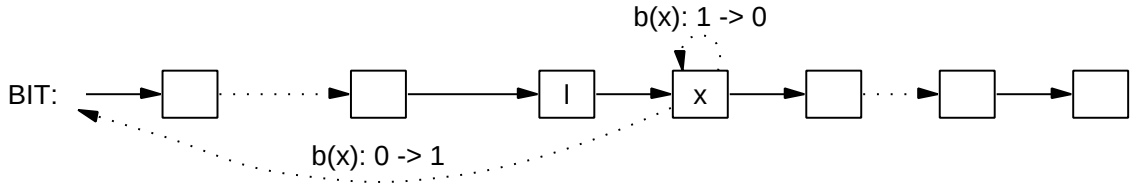
OPT bearbeite die Anfragen vor BIT. Wir betrachten nun die Fälle, in denen eine Inversion aufgehoben bzw. neu erzeugt werden kann. Sei k die Position in OPT's Liste direkt vor dem aktuell angefragten Listenelement x und i die Position direkt vor der neuen Position von x.



Für die Aktionen von OPT gilt:

$$OPT(\sigma(t)) = k + 1$$
$$E[\Delta_{\text{OPT}}\Phi] \leq (k - i)\left(\frac{1}{2}(1 + 2)\right) = 1.5(k - i)$$

In BIT's Liste sei x an Position $l + 1$.



$$\text{BIT}(\sigma(t)) = l + 1$$

Für die Potentialänderung durch BIT's Bedienen der Anfrage unterscheiden wir zwei Fälle.

Erster Fall: $l \leq i$.

(a) Ist $b(x) = 1$ vor dem Zugriff, dann wechselt $b(x)$ auf 0. Die Position von x wird nicht verändert, es können also keine neuen Inversionen entstehen. Damit ist $\Delta_{\text{BIT}_a}\Phi \leq 0$.

(b) Ist $b(x) = 0$, dann können höchstens l neue Inversionen entstehen, die jeweils entsprechend des Bitwerts des übersprungenen Listenelements bewertet werden. Damit ergibt sich

$$E[\Delta_{\text{BIT}_b}\Phi] \leq l\left(\frac{1}{2}(1 + 2)\right) = 1.5l \quad .$$

Da (a) und (b) je mit Wahrscheinlichkeit 0.5 auftreten, gilt:

$$\begin{aligned}
\mathrm{BIT}(\sigma(t)) + E[\Delta\Phi] &= \mathrm{BIT}(\sigma(t)) + E[\Delta_{\mathrm{OPT}}\Phi] + E[0.5\Delta_{\mathrm{BIT}_a}\Phi + 0.5\Delta_{\mathrm{BIT}_b}\Phi] \\
&\leq l + 1 + 1.5(k-i) + 0.5 \cdot 0 + 0.5 \cdot 1.5l \\
&= l + 1 + 0.75l + 1.5(k-i) \\
&= 1.75l + 1.5(k-i) + 1 \\
&\leq 1.75i + 1.5(k-i) + 1 \\
&\leq 1.75(k+1) \\
&= 1.75\ \mathrm{OPT}(\sigma(t))
\end{aligned}$$

Zweiter Fall: $i < l$.

(a) Ist $b(x) = 1$, dann ändert sich die Position von x nicht. Da x in OPT jetzt an Position $i+1$ ist, gibt es $l-i$ Inversionen, deren Typ von 2 auf 1 wechselt. Damit ergibt sich $\Delta_{\mathrm{BIT}_a}\Phi \leq -(l-i)$.

(b) Ist $b(x) = 0$, dann kommt x an den Kopf der Liste, womit $l-i$ Inversionen aufgehoben werden. Hier sinkt Φ mindestens um $l-i$. Außerdem werden höchstens i neue Inversionen (z,x) erzeugt, die im Erwartungswert mit 1.5 zu Φ beitragen. Also ist $E[\Delta_{\mathrm{BIT}_b}\Phi] \leq 1.5i - (l-i)$. Da (a) und (b) wieder mit gleicher Wahrscheinlichkeit auftreten, ergibt sich:

$$\begin{aligned}
\mathrm{BIT}(\sigma(t)) + E[\Delta\Phi] &= \mathrm{BIT}(\sigma(t)) + E[\Delta_{\mathrm{OPT}}\Phi] + E[0.5\Delta_{\mathrm{BIT}_a}\Phi + 0.5\Delta_{\mathrm{BIT}_b}]\Phi \\
&\leq l + 1 + 1.5(k-i) - 0.5(l-i) + 0.5(1.5i - l + i) \\
&= l + 1 + 1.5k - 1.5i - 0.5l + 0.5i + 0.75i - 0.5l + 0.5i \\
&= 1 + 1.5k + 0.25i \\
&\leq 1 + 1.75k \\
&\leq 1.75(k+1) \\
&= 1.75\ \mathrm{OPT}(\sigma(t))
\end{aligned}$$

□

1.4 Anwendung von linearen Listen in der Datenkompression

Eine interessante Anwendung finden selbstorganisierende Listen in der Datenkompression. Dabei soll ein Wort $S = x_{i_1}, x_{i_2}, \ldots$ über $\Sigma = \{x_1, \ldots, x_n\}$ kompakter dargestellt werden. Halte eine lineare Liste mit den Elementen aus Σ. Immer wenn ein Zeichen x_i geselen wird, schlägt der Kodierer die aktuelle Position von x_i in der Liste nach, gibt diese Position aus und aktualisiert die Liste z.B. gemäß MTF.

Da häufig vorkommende Zeichen so mit kleinen Zahlen kodiert werden, ist der Ausgabestring kompakter als das Ausgangswort.

Bei der Dekodierung wird der gleiche Algorithmus zur Aktualisierung der Liste angewandt. So kann der Ursprungsstring wieder rekonstruiert werden. Kodierer und Dekodierer müssen dabei natürlich mit der gleichen Anfangsliste starten.

Damit wir dadurch auch tatsächlich mit weniger Bits auskommen, müssen wir die Daten geschickt kodieren. Wir kodieren eine natürliche Zahl j durch die Bitfolge, indem wir zunächst die Länge der Zahl durch $\lfloor \log j \rfloor$ Nullen angeben und anschließend die Binärkodierung der Zahl j. Auf diese Weise kommen wir mit $2\lfloor \log j \rfloor + 1$ Stellen aus.

Zahlenfolge:	3	12	5	6	14
Kodierung:	011	0001100	00101	00110	0001110

Abbildung 4: Beispiel für eine Kodierung variabler Länge

Neben einer byte-basierten Kodierung ist auch eine Wort-Kodierung möglich. Dabei wird jedes natürlichsprachliche Wort als ein Element des Alphabets aufgefasst.

Burrows-Wheeler-Transformation:

- **Kodierung:** Berechne für S alle zyklischen Shifts und sortiere sie lexikographisch. Also z.B. für $S =$ `abraca`:

`abraca`		`aabrac`
`aabrac`		`abraca`
`caabra`	und sortiert dann:	`acaabr`
`acaabr`		`bracaa`
`racaab`		`caabra`
`bracaa`		`racaab`

  .

  Extrahiere aus der so erhaltenen Matrix die letzte Spalte und notiere den Zeilenindex des Originalwortes. In unserem Beispiel also (caraab, 2). Wende auf diesen transformierten String dann z.B. die MTF-Kodierung an. Dadurch erhält man eine Kompression auf etwa 20 bis 30%.

- **Dekodierung:** Jede Spalte der transformierten Matrix ist eine Permutation des Ausgangsstring. Die letzte Spalte der Matrix ist bekannt. Damit kann durch Sortieren dieser Spalte auch die erste Spalte rekonstruiert werden. Dadurch sind jetzt alle auftauchenden Buchstabenpaare bekannt. Durch lexikografisches Sortieren dieser Buchstabenpaare erhalten wir die zweite Spalte. Entsprechend können die verbleibenden Spalten rekonstruiert werden.

1.5 Scheduling und Lastbalancierung

Wir haben bisher den Spezialfall des Scheduling-Problems betrachtet, in dem der Makespan minimiert werden soll. Im Allgemeinen sind beim Scheduling m (möglicherweise verschiedene) Maschinen vorhanden, auf die eine Sequenz von Jobs verteilt werden soll. Die Bearbeitungsdauer eines Job kann davon abhängen, auf welcher Maschine er abläuft. Es ist eine gegebene Zielfunktion zu optimieren.

Die Eingabesequenz ist $\sigma = r_1, \ldots, r_n$ mit $r_j = (r_j(1), r_j(2), \ldots, r_j(m))$. Dabei ist $r_j(i)$ die Bearbeitungsdauer des j-ten Jobs auf der i-ten Maschine.

Mögliche Maschinenmodelle:

- identisch: $\exists v_j$ mit $r_j(i) = v_j$ für $i = 1, \ldots, m$.
- uniform: $\exists s_1, \ldots, s_m$ und $v_1, \ldots, v_n$, so dass $r_j(i) = v_j/s_i$ für $i = 1, \ldots, m$.
- nicht-uniform: $r_j(i)$ dürfen beliebige Werte annehmen.
- eingeschränktes Modell: $\forall r_j \exists v_j : r_j(i) \in \{v_j, \infty\}$. Dabei bedeutet $r_j(i) = \infty$, dass die Maschine i den Job j nicht bearbeiten kann.

Mögliche Zielfunktionen:

- Minimiere Makespan.
- Minimiere (gewichtete) Summe von Fertigstellungszeitpunkten der Jobs.
- Minimiere (gewichtete) Summe der verspäteten Jobs. Die Jobs haben dabei bestimmte Termine.

Die Jobbearbeitungszeiten müssen nicht unbedingt im Voraus bekannt sein.

Bei der *Lastbalancierung* sind ebenfalls m Maschinen vorhanden. Die Jobs sind nur eine begrenzte Zeit aktiv. Diese Zeitspanne ist im Allgemeinen nicht bekannt. Zu minimieren ist die aktuelle Last zu allen Zeitpunkten.

Im Folgenden wollen wir Algorithmen zur Lastbalancierung im eingeschränkten Modell untersuchen.

Sei m die Anzahl der Maschinen und $\sigma = r_1, \ldots, r_n$ die Anfragesequenz. r_j enthält folgende Informationen:

- Anfangszeit s_j, wobei $s_j \leq s_{j+1}$
- Menge der möglichen Maschinen $M_j \subseteq \{1, \ldots, m\}$
- Last l_j

Das Ziel ist eine Minimierung der größten auftretenden Last.

SATZ: Sei A ein determinister Onlinealgorithmus zur Lastbalancierung im eingeschränkten Modell, der c-kompetitiv ist. Dann gilt $c = \Omega(\sqrt{m})$, selbst dann, wenn alle Joblasten gleich sind.

Beweis: Sei A ein deterministischer Online-Lastbalancierungs-Algorithmus. $L_i(t)$ bezeichne die Last der i-ten Maschine unmittelbar bevor der neue Job r_t eintrifft. Mit der Last einer Maschine ist dabei die Summe der Lasten der einzelnen aktiven Jobs auf der Maschine gemeint.

In jedem Schritt nummerieren wir die Maschinen so um, dass $L_1(t) \geq L_2(t) \geq \ldots \geq L_{q(t)}(t)$ und $L_i(t) = 0$ für alle $i > q(t)$.

Wir bauen eine Sequenz σ, so dass stets exakt m Jobs mit Last 1 aktiv sind und OPT diese Jobs gleichmäßig auf alle m Maschinen verteilen kann. Mit l bezeichnen wir das Maximum über die Lasten auf allen Maschinen und zu allen Zeitpunkten. Für A wollen wir eine Lastverteilung mit $L_i(t) \geq L_{i+1}(t) + 1$ für alle $i \leq q(t)$ erreichen. Sobald dieser Zustand erreicht ist, gilt $q(t) \leq l$. Damit erhalten wir

$$m = \sum_{i=1}^{m} L_i(t) \leq l \cdot q(t) \leq l^2$$

und somit $l \geq \sqrt{m}$.

Für $L = (L_1(t), \ldots, L_{q(t)}(t))$ und $L' = (L_1(t'), \ldots, L_{q(t')}(t'))$ sei $L < L'$, falls es ein i gibt, so dass $L_i(t) < L_i(t')$ und $L_k(t) = L_k(t')$ für alle $k < i$ (lexikografische Ordnung).

Konstruktionsstufen:

- Zunächst werden m Jobs J_i mit $M_i = \{i\}$ für $i = 1 \ldots m$ angefragt.
- Sei $L^k = (L_1^k, L_2^k, \ldots, L_{q_k}^k)$ der Lastvektor nach der k-ten Konstruktionsstufe. Ist bereits $L_i^k \geq L_{i+1}^k + 1$ für alle $i \leq q(t)$, dann sind wir fertig. Ansonsten konstruieren wir einen neuen Lastvektor $L^{k+1} > L^k$ (siehe weiter unten).

Die Konstruktion bricht ab, da $(m, 0, \ldots, 0)$ der größtmögliche Lastvektor ist.

Wir wollen folgende Invariante erhalten: Hat OPT den Job j auf Maschine i und ist $L_i^k > 0$, so hat auch A Job j auf Maschine i.

Zur Konstruktion des neuen Lastvektors gehen wir folgendermaßen vor:

- Bestimme das erste Plateau der Lastverteilung von A, also das kleinste i mit $L_i^k = L_{i+1}^k$.
- Beende diejenigen Jobs, die bei OPT auf Maschinen i und $i+1$ laufen. Nach der Invarianten laufen diese Jobs auch bei A auf den Maschinen i und $i+1$.
- Wir erzeugen nun einen neuen Job r mit $M_r = \{i, i+1\}$. A plane Job r o.B.d.A. auf Maschine i ein. OPT plant Job r dann auf $i+1$ ein. Außerdem erzeugen wir noch einen weiteren Job r' mit $M_{r'} = \{i\}$.
- Beende alle Jobs, die A auf $i+1$ laufen hat, damit die Invariante wieder hergestellt wird.
- Für jede leere Maschine bei OPT führen wir einen neuen Job ein, der nur dort laufen darf. Dadurch stellen wir sicher, dass genau m Jobs vorhanden sind).

□

Wir wollen nun einen Algorithmus für das Lastbalancierungs-Problem konstruieren. Mit σ_j bezeichnen wir das Präfix $r_1, \ldots, r_j$ der Jobsequenz $\sigma = r_1, \ldots, r_n$. Unser Algorithmus hält eine untere Schranke $B(s_j)$ als Schätzung von $\mathrm{OPT}(\sigma_j)$, d.h. $B(s_j)$ ist eine untere Schranke für die maximale Last, die zwischen dem Startzeitpunkt und dem Zeitpunkt s_j aufgetreten ist. Dabei ist

$$B(s_0) = 0 \qquad \text{und} \qquad B(s_j) = \max\left\{B(s_{j-1}), l_j, \frac{1}{m}\left(l_j + \sum_{i=1}^{m} L_i(s_j)\right)\right\} \quad .$$

Eine Maschine i heißt zum Zeitpunkt j *reich*, wenn $L_i(s_j) > \sqrt{m}B(s_j)$, andernfalls heißt sie *arm*.

ALGORITHMUS: ROBIN-HOOD

Platziere r_1 auf einer beliebigen zulässigen Maschine. Platziere jeden weiteren Job r_j möglichst auf einer armen Maschine. Ist das nicht möglich, dann wird der Job auf der reichen Maschine platziert, die als letztes reich wurde.

SATZ: ROBIN-HOOD ist $(2\sqrt{m}+2)$-kompetitiv.

Beweis: Zu jedem Zeitpunkt gibt es höchstens $\sqrt{m}$ reiche Maschinen. Denn sonst wäre

$$\sum_{i=1}^{m} L_i(s_j) > \sqrt{m}\sqrt{m}B(s_j) \geq \sum_{i=1}^{m} L_i(s_j) \quad .$$

Zeige induktiv, dass die maximale Last, die online je aufgetreten ist, höchstens das $(2\sqrt{m}+2)$-fache der optimalen Last ist.

Trifft ein neuer Job r_j ein, dann platziert ROBIN-HOOD diesen Job auf Maschine i_0. Wir unterscheiden die beiden folgenden Fälle.

- i_0 arm:

$$L_{i_0}(s_j) + l_j \leq \sqrt{m}B(s_j) + B(s_j) = \left(\sqrt{m}+1\right) B(s_j) \leq \left(\sqrt{m}+1\right) \mathrm{OPT}(\sigma_j) \quad .$$

- i_0 reich: Sei $s_{t(j)}$ der Zeitpunkt, zu dem i_0 reich wurde (d.h. unmittelbar vor $s_{t(j)}$ war i_0 arm). Sei S die Menge der Jobs, die in $(s_{t(j)}, s_j)$ auf i_0 platziert wurden. Jeder Job $k \in S$ konnte nur auf reiche Maschinen verteilt worden sein, sonst wäre er ja nicht auf i_0 eingeplant worden. Für diese Jobs konnte also nur aus den Maschinen gewählt werden, die zum Zeitpunkt $s_{t(j)}$ reich waren. Denn hätte ein Job auf einer Maschine eingeplant werden können, die zum Zeitpunkt $s_{t(j)}$ noch arm war und erst später reich wurde, dann wäre er auch dort eingeplant worden und nicht auf i_0. Also gilt

$$\left| \bigcup_{k \in S} M_k \right| \leq \sqrt{m} \quad .$$

  OPT muss alle Jobs aus S also auf höchstens $\sqrt{m}$ Maschinen verteilen. Damit gilt

$$\frac{1}{\sqrt{m}} \sum_{k \in S} l_k \leq \mathrm{OPT}(\sigma_j) \quad .$$

  Wir erhalten für jeden Zeitpunkt j:

$$\begin{aligned} L_{i_0}(s_j) + l_j &= L_{i_0}(s_{t(j)}) + l_{s_{t(j)}} + \sum_{k \in S} l_k + l_j \\ &\leq \sqrt{m}B(s_{t(j)}) + \mathrm{OPT}(\sigma_j) + \sqrt{m}\mathrm{OPT}(\sigma_j) + B(s_j) \\ &\leq \sqrt{m}\,\mathrm{OPT}(\sigma_j) + \mathrm{OPT}(\sigma_j) + \sqrt{m}\mathrm{OPT}(\sigma_j) + \mathrm{OPT}(\sigma_j) \\ &\leq (2\sqrt{m}+2)\mathrm{OPT}(\sigma_j) \end{aligned}$$

□

1.6 Finanzielle Spiele

1. Suchprobleme: Wir suchen das Maximum (Minumum) in einer Sequenz von Preisen, die online präsentiert werden. Akzeptieren wir in der Periode i, dann ist unser Ertrag p_i. Andernfalls spielen wir weiter.

   Beispiele: Hausverkauf, Gehaltsverhandlung

2. Einwegehandel: Ein Anfangsguthaben D_0 soll in eine andere Währung getauscht werden. Zu welchem Zeitpunkt soll getauscht werden?

3. Verwaltung eines Portfolios: s Möglichkeiten, das Geld zu investieren. Für eine Periode i ist $p_i = (p_{i_1}, p_{i_2}, \ldots, p_{i_s})$ gegeben, wobei p_{i_j} die Anzahl der Einheiten der j-ten Anlageform ist, die für eine Geldeinheit gekauft werden können. Der Online-Spieler muss eine Verteilung $b_i = (b_{i_1}, b_{i_2}, \ldots, b_{i_s})$ des Guthabends D_i auf die Anlageformen vornehmen. Dabei ist $\sum_{j=1}^{s} b_{i_j} = 1$.

Wir konzentrieren uns auf die Suchprobleme. Um überhaupt Kompetivität erreichen zu können, gehen wir von einer beschränkten Preisspanne aus, also $p \in [m, M]$ mit $0 < m \leq M$, die dem Onlinespieler bekannt ist. Außerdem betrachten wir diskrete Zeitschritte und einen endlichen Zeithorizont.

> ALGORITHMUS: Reservation-Price-Policy
>
> Bei der *Reservation-Price-Policy* (RPP) setzen wir einen *Reservation Price* $p^* = \sqrt{mM}$ und akzeptieren den ersten Preis p mit $p \geq p^*$. Falls wir am Ende noch kein Angebot angenommen haben, geben wir uns mit dem letzten Angebot zufrieden.

SATZ: RPP ist $\sqrt{\varphi}$-kompetitiv, wobei $\varphi = M/m$ ist.

Beweis: Sei die Preissequenz beliebig und $p_{\max}$ der maximale Preis in der Sequenz.

1. Ist $p^* \geq p_{\max}$, dann
$$\frac{\mathrm{OPT}(\sigma)}{\mathrm{RPP}(\sigma)} \leq \frac{p_{\max}}{m} \leq \frac{p^*}{m} = \frac{\sqrt{Mm}}{m} = \sqrt{\varphi} \quad .$$

2. Ist $p^* \leq p_{\max}$, dann
$$\frac{\mathrm{OPT}(\sigma)}{\mathrm{RPP}(\sigma)} \leq \frac{p_{\max}}{p^*} \leq \frac{M}{\sqrt{Mm}} = \sqrt{\varphi} \quad .$$

□

SATZ: Kein deterministischer Online-Algorithmus zur Suche in Preissequenzen A ist besser als $\sqrt{\varphi}$-kompetitiv.

Beweis: Der Gegner bietet als ersten Preis $\sqrt{mM}$. Akzeptiert A das Angebot, dann bieten wir als nächstes M und erhalten als Verhältnis $\sqrt{\varphi}$. Akzeptiert A das Angebot nicht, dann bieten wir als nächstes m und erhalten ebenfalls als Verhältnis $\sqrt{\varphi}$. □

ALGORITHMUS: EXPO

EXPO ist ein randomisierter Algorithmus. Sei $M = m2^k$ mit $k \in \mathbb{N}$, also $\varphi = 2^k$. Mit RPP_i bezeichnen wir RPP mit $p^* = m2^i$. Wähle mit Wahrscheinlichkeit $1/k$ die Strategie RPP_i für $i = 1, \ldots, k$.

SATZ: EXPO ist $\log_2 \varphi \cdot c(\varphi)$-kompetitiv mit $c(\varphi) \to 1$ für $\varphi \to \infty$.

Beweis: Sei eine beliebige Preissequenz gegeben und $j \in \mathbb{N}$ so, dass $m2^j \leq p_{\max} < m2^{j+1}$.

Wir können die folgenden Annahmen machen, da der Quotient $\frac{\text{OPT}(\sigma)}{E[\text{EXPO}(\sigma)]}$ dadurch nur größer werden kann.

- Bevor $p_{\max}$ gezeigt wird, wird $m2^j$ gezeigt.
- Alle Preise, die kleiner als $p_{\max}$ sind, haben die Form $m2^i$ für ein $i \in \mathbb{N}$. Denn dadurch ändert sich an der Entscheidung für ein Angebot nichts.
- $p_{\max}$ ist $m(2^{j+1} - \varepsilon)$ für ein kleines ε.

Hat EXPO die Strategie RPP_i gewählt, dann ist für $i \leq j$ der Ertrag mindestens $m2^i$. Für $i > j$ und damit $p^* > p_{\max}$ ist der Ertrag von EXPO mindestens m. Als erwarteter Ertrag von EXPO ergibt sich daraus

$$E[\text{EXPO}] = \frac{1}{k}\left(\sum_{i=1}^{j} m2^i + (k-j)m\right) = \frac{m}{k}\left(2^{j+1} - 2 + k - j\right)$$

und damit

$$\frac{\text{OPT}(\sigma)}{E[\text{EXPO}(\sigma)]} = k\frac{2^{j+1} - \varepsilon}{2^{j+1} + k - j - 2} = \log \varphi \underbrace{\left(\frac{2^{j+1} - \varepsilon}{2^{j+1} + k - j - 2}\right)}_{=:\alpha} \quad .$$

α wird minimiert für $j^* = k - 2 + \frac{1}{\log 2}$. Dann ist

$$\alpha \leq \frac{1}{\frac{2^{j+1}+k-j-2}{2^{j+1}}} \leq \frac{1}{1 + \frac{1/\log 2}{2^{k-1+1/\log 2}}} \to 1 \qquad \text{für } k \to \infty \quad .$$

□

1.7 k-Server-Problem

Sei $\mathcal{M} = (M, \text{dist})$ ein metrischer Raum. Wir haben k Server zur Verfügung und betrachten eine Sequenz σ von Punkten in M. Die Anfrage wird bedient, indem Server zu den entsprechenden Punkten geschickt werden, sofern dort noch kein Server anwesend ist.

Sei $\text{dist}(i, j)$ die Entfernung bzw. die Kosten, um einem Server von i nach j zu schicken. Zu minimieren ist die Summe der von allen k Servern zurückgelegten Distanz.

Das k-Server-Problem ist eine Verallgemeinerung des Paging-Problems, denn wir können M als die Menge der Speicherseiten wählen, $\text{dist}(i, j) = 1$ für alle $i \neq j$ setzen und k als die Anzahl der Seiten im schnellen Speicher wählen.

Für dieses Problem exisitert ein $(2k - 1)$-kompetitiver deterministischer Online-Algorithmus. Die größte bekannte untere Schranke ist k.

Für randomisierte Online-Algorithmen ist eine untere Schranke von $\Omega(\sqrt{\log k / \log\log k})$ bekannt.

Für spezielle metrische Räume (Bäume, Linie) sind k-kompetitive Algorithmen bekannt.

1.8 Metrische Task-Systeme

Sei $\mathcal{M} = (M, \text{dist})$ wieder ein metrischer Raum. M ist die Menge der Zustände, in denen der Online-Spieler sein kann. Es sei $N = |M|$. Mit $\text{dist}(i, j)$ bezeichnen wir die Kosten für den Zustandswechsel von i nach j. Das Minimum über alle Distanzen sei 1.

$\mathcal{R}$ sei die Menge der möglichen Tasks (Aufgaben), wobei für jeden Task $r = (r(1), \ldots, r(N)) \in \mathcal{R}$ mit $r(i)$ die Kosten bezeichnet werden, um die entsprechende Aufgabe in Zustand i zu erledigen.

Der Online-Spieler startet in Zustand S_0. Sei $\sigma = r_1, \ldots, r_n$. Wenn r_j präsentiert wird, darf er den Zustand wechseln und muss dann die Aufgabe erledigen.

Ist $A[j]$ der Zustand, in dem r_j erledigt wird, dann gilt

$$A(\sigma) = \sum_{j=1}^{n} \text{dist}(A[j-1], A[j]) + \sum_{j=1}^{n} r(A[j]) \qquad .$$

Metrische Task-Systeme sind eine Verallgemeinerung von selbstorganisierenden Listen. Die Liste habe n Elemente. Als Zustände wählen wir die $n!$ Permutationen der Liste, als $\text{dist}(s_i, s_j)$ die Anzahl der paid exchanges, um die Liste s_i in die Liste s_j zu überführen. Die Kosten einer Anfrage an x im Zustand s_i ist die Position des angefragten Elementes in der Listenkonfiguration s_i.

Der optimale Faktor für deterministische Online-Algorithmen ist $2N - 1$. Die bekannten Schranken für randomisierte Algorithmen sind $O((\log N \log\log N)^2)$ und $\Omega(\sqrt{\log N / \log\log N})$.

2 Approximationsalgorithmen

Berechnung von Näherungslösungen für NP-harte Optimierungsprobleme.

Beispiel Jobscheduling: n Jobs mit Bearbeitungszeiten $r_1, \ldots, r_n$ sind auf m identischen Maschinen so einzuplanen, dass der Makespan minimiert wird.

> ALGORITHMUS: Greedy
>
> Plane jeden Job auf der Maschine mit der jeweils geringsten Last ein.

Wie wir bereits gezeigt haben, ist Greedy eine 2-Approximation.

DEFINITION: Ist Π ein Optimierungsproblem und $\mathfrak{I}$ die Menge der Probleminstanzen. Für eine Instanz $I \in \mathfrak{I}$ ist $F(I)$ die Menge der zulässigen Lösungen. Für $s \in F(I)$ ist $w(s) \in \mathbb{R}^+$ der Wert der Lösung. Ein *Approximationsalgorithmus* A für Π ist ein Algorithmus, der für eine Instanz $I \in \mathfrak{I}$ eine zulässige Lösung ausgibt und dabei eine Laufzeit hat, die polynomiell beschränkt in $|I|$ ist. Der *Approximationsfaktor* von A auf I ist

$$\delta_A(I) := \frac{w(A(I))}{\mathrm{OPT}(I)} \quad .$$

$\mathrm{OPT}(I)$ ist dabei der Wert einer optimalen Lösung. A hat *Approximationsfaktor* δ, gdw

$$\delta_A(I) \geq \delta \quad \forall I \in \mathfrak{I}$$

bei Maximierungsproblemen bzw. mit "$\leq$" bei Minimierungsproblemen.

2.1 Graphenalgorithmen

2.1.1 Max-Cut

DEFINITION: Beim *Max-Cut-Problem* ist ein ungerichteter Graph $G = (V, E)$ gegeben und eine Partition $(S, V \setminus S)$ gesucht, in der die Menge der Schnittkanten maximal ist. Eine *Schnittkante* ist eine Kante $\{a, b\} \in E$ mit $a \in S$ und $b \notin S$.

DEFINITION: *Symmetrische Differenz*

$$S \Delta \{v\} := \begin{cases} S \cup \{v\} & v \notin S \\ S \setminus \{v\} & v \in S \end{cases}$$

> ALGORITHMUS: Local Improvement (LI)
>
> 1 $S := \emptyset$
> 2 **while** $\exists\, v \in V :\ w(S \Delta \{v\}) > w(S)$ **do**
> 3 $\quad S := S \Delta \{v\}$
> **end**

SATZ: LI hat einen Approximationsfaktor von 1/2.

Beweis: Sei S der von LI berechnete Schnitt. Sei $\mathrm{LI}(I)$ die Menge der vom Algorithmus LI auf der Instanz I erzeugten Schnittkanten. Sei $v \in V$ beliebig. Das Gewicht der Schnittkanten unter den zu v adjazenten Kanten ist mindestens so groß wie das Gewicht der Nicht-Schnittkanten. Andernfalls wäre die Abbruchbedingung der Schleife in Zeile 2 nicht erfüllt.

Durch Summation über V erhalten wir also

$$w(\mathrm{LI}(I)) \geq w(E \setminus \mathrm{LI}(I)) = w(E) - w(\mathrm{LI}(I))$$

und damit

$$\begin{aligned} 2 \cdot w(\mathrm{LI}(I)) &\geq w(E) \\ w(\mathrm{LI}(I)) &\geq \frac{1}{2}\, w(E) \geq \frac{1}{2}\, \mathrm{OPT}(I) \quad . \end{aligned}$$

□

2.1.2 Traveling Salesperson

DEFINITION: Sei $G = (V, E)$ und $V = \{v_1, \ldots, v_n\}$. Eine *Rundreise* ist eine Permutation π von $\{1, \ldots n\}$ mit $\{v_{\pi(i)}, v_{\pi(i+1)}\} \in E$ für $i = 1, \ldots, n-1$ und $\{v_{\pi(n)}, v_{\pi(1)}\} \in E$.

DEFINITION: Beim *Problem des euklidischen Handelsreisenden* ETSP sind n Städte $s_1, \ldots, s_n \in \mathbb{R}^2$ gegeben. Finde eine Rundreise minimaler Länge, die jede Stadt genau einmal besucht.

DEFINITION: Beim allgemeinen *Problem des Handelsreisenden* TSP ist ein Graph $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$ und einer Funktion $w : E \to \mathbb{R}^+$ gegeben, die jeder Kante $e \in E$ ein Gewicht (bzw. Länge) $w(e)$ zuordnet. Finde eine Rundreise minimaler Länge, d.h. eine Permutation π, so dass

$$w(\pi) := \sum_{i=1}^{n-1} w(\{v_{\pi(i)}, v_{\pi(i+1)}\}) + w(\{v_{\pi(n)}, v_{\pi(1)}\})$$

minimal wird.

DEFINITION: Ein *Hamilton Kreis* zu einem Graphen $G = (V, E)$ ist eine Rundreise durch alle Knoten, bei der jeder Knoten $v \in V$ genau einmal besucht wird.

DEFINITION: Ein *Euler Kreis* zu einem Graphen $G = (V, E)$ ist eine Rundreise durch alle Knoten, bei der jede Kante $e \in E$ genau einmal besucht wird.

DEFINITION: Beim Problem *Hamilton-Kreis* (HK) ist für einen Graphen $G = (V, E)$ gefragt, ob G einen Hamiltonschen Kreis enthält.

SATZ: ETSP und TSP sind NP-hart.

Beweis: HK ist NP-hart und HK $\subseteq$ TSP, also ist TSP NP-hart. Für ETSP bleibt die Behauptung noch zu zeigen. □

DEFINITION: Für einen Graphen $G = (V, E)$ und eine Gewichtsfunktion $w : E \to \mathbb{R}^+$ ist ein *minimaler Spannbaum* T für G ein Baum, so dass $v \in T$ für alle $v \in V$ und das Gesamtgewicht der Kanten von T minimal ist.

ALGORITHMUS: MST (für ETSP)

1 Berechne einen minimalen aufspannenden Baum T für $G = (V, E)$ mit $V = \{s_1, \ldots, s_m\}$ und $E = V^2$ bezüglich der euklidischen Distanz.
2 Konstruiere einen Multigraphen H, in dem alle Kanten von T verdoppelt werden.
3 Finde einen Eulerkreis K in H (Kreis über alle Kanten).
4 Berechne Reihenfolge der ersten Auftritte $s_{\pi(1)}, \ldots s_{\pi(n)}$ der Knoten $s_1, \ldots, s_n$ in K und gib $s_{\pi(1)}, \ldots, s_{\pi(n)}$ aus.

SATZ: MST hat einen Approximationsfaktor von 2.

Beweis: Sei $I \in \mathfrak{I}$ und R eine Rundreise minimaler Länge bezüglich I mit Wert $\mathrm{OPT}(I)$. Sei T ein minimaler Spannbaum mit Gewicht $|T|$.

Es gilt $|T| \leq \mathrm{OPT}(I)$, denn sonst könnten wir aus R eine Kante entfernen und hätten damit einen Spannbaum mit kleinerem Gewicht.

K hat die Länge $2 \cdot w(T)$, da K durch Kantenverdopplung aus T entsteht. Die Rundreise $\mathrm{MST}(I)$ entsteht aus K, indem Abkürzungen genommen werden. Also gilt

$$\mathrm{MST}(I) \leq 2w(T) \leq 2\,\mathrm{OPT}(I)$$

und damit ist MST 2-approximativ. □

SATZ: Ein Graph $G = (V, E)$ hat eine gerade Anzahl von Knoten mit ungeradem Grad.

Beweis:

$$\underbrace{\sum_{\substack{v \in V \text{ mit} \\ \text{geradem Grad}}} \deg(v)}_{\text{gerade}} + \underbrace{\sum_{\substack{v \in V \text{ mit} \\ \text{ungeradem Grad}}} \deg(v)}_{\text{also auch gerade}} = \sum_{v \in V} \deg(v) = \underbrace{2 \cdot |E|}_{\text{gerade}}$$

Da die Summe über die $v \in V$ mit ungeradem Grad gerade ist, muss eine gerade Anzahl von ungeraden Zahlen aufsummiert worden sein. Also ist die Anzahl der ungeraden Knoten in jedem Graphen gerade. □

DEFINITION: Sei $G = (V, E)$ ein ungerichteter Graph mit Gewichtsfunktion $w : E \to \mathbb{R}^+$. Ein *perfektes Matching* ist eine Teilmenge $F \subseteq E$, so dass jeder Knoten $v \in V$ inzident zu genau einer Kante aus F ist. Ein perfektes Matching existiert nicht immer (z.B. bei isolierten Knoten oder ungerader Knotenanzahl). Ein *minimales perfektes Matching* ist ein perfektes Matching minimalen Gewichts.

BEMERKUNG: Für das Problem der Berechnung eines minimalen perfekten Matchings existiert ein polynomieller Algorithmus.

ALGORITHMUS: Christofides

1 berechne MST T
2 ermittele in T die Knotenmenge V' aller Knoten mit ungeradem Grad
3 berechne ein minimales perfektes Matching F für V' und füge diese Kantenmenge zu T hinzu
4 berechne einen Eulerkreis in $T \cup F$; die Rundreise R ist die Sequenz der ersten Vorkommen jeder Stadt

SATZ: Christofides erzielt einen Approximationsfaktor von 1.5.

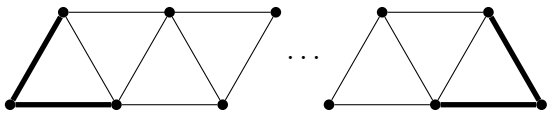
Beweis: Sei I eine Probleminstanz und OPT(I) die Länge der optimalen Rundreise. Das Gewicht eines minimalen Spannbaums ist durch OPT(I) beschränkt.

Betrachten wir nur V', dann ist das Gewicht einer optimalen Rundreise R' durch V' ebenfalls durch OPT(I) beschränkt. R' ist eine Kombination aus zwei perfekten Matchings. Also hat das minimale Matching höchstens ein Gewicht von $0.5 \cdot$ OPT(I). Das Gewicht der Kantenmenge $T \cup F$ ist damit höchstens $1.5 \cdot$ OPT(I). Durch Abkürzungen kann die Tour nicht länger werden, also erreicht Christofides einen Approximationsfaktor von 1.5. □

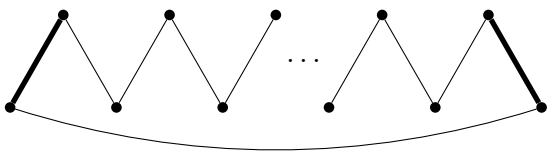
BEMERKUNG: Es ist kein Algorithmus zur Approximation von ETSP bekannt, der besser als 1.5-approximativ ist.

SATZ: Der Approximationsfaktor von Christofides ist nicht besser als 1.5.

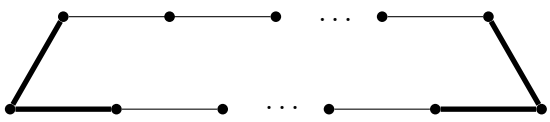
Beweis: Wir ordnen 2n + 1 Städte folgendermaßen an, wobei die normalen Kanten mit 1 und die dicken Kanten mit $1 + \varepsilon$ bewertet sind.



Von Christofides wird die folgende Rundreise berechnet, die Gewicht $2n + 2\varepsilon + n + 2\varepsilon = 3n + 4\varepsilon$ hat.



Die optimale Rundreise dagegen ist nur $2n + 1 + 4\varepsilon$ lang.



Also erhalten wir

$$c = \frac{3n + 4\varepsilon}{2n + 1 + 4\varepsilon} \leq \frac{3n + 4\varepsilon}{2n + 4\varepsilon} \stackrel{\varepsilon \to 0}{\longrightarrow} \frac{3}{2} \quad .$$

□

SATZ: Sei $c > 1$ eine Konstante. Ist P $\neq$ NP, dann existiert für TSP keine c-Approximation.

Beweis: Angenommen, es gibt eine c-Approximation für TSP. Wir zeigen, dass dann HK in polynomieller Zeit lösbar ist.

Sei A der c-Approximationsalgorithmus für TSP und $G = (V, E)$ die Eingabe für HK. Wir setzen

$$w(\{i, j\}) = \begin{cases} 1 & \{i, j\} \in E \\ c \cdot |V| & \text{sonst} \end{cases}$$

Berechne Rundreise R im Graphen mit Hilfe von A. Es gibt genau dann einen Hamilton-Kreis, wenn die Länge von R höchstens $c \cdot |V|$ ist.

Hat der Ausgangsgraph einen Hamilton-Kreis, dann hat der Graph für TSP eine Rundreise der Länge $|V|$. Die berechnete Rundreise ist also höchstens $c \cdot |V|$ lang. Enthält der Ausgangsgraph keinen Hamilton-Kreis, dann benötigt die Rundreise im TSP-Graphen mindestens eine lange Kante, also ist das Gewicht der gesamten Rundreise größer als $c \cdot |V|$. □

2.2 Jobscheduling

Sei m die Anzahl der Maschinen und n die Anzahl der Jobs mit Bearbeitungszeiten $p_1, \ldots, p_n$. Wir wollen wieder den Makespan optimieren.

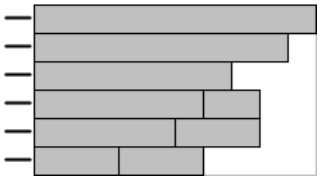
> ALGORITHMUS: Sorted List Scheduling (SLS)
>
> Sortiere die Jobs, so dass $p_1 \geq p_2 \geq \ldots \geq p_n$, und wende Greedy auf diese Sequenz an.

SATZ: SLS erzielt eine Approximationsgüte von 4/3.

LEMMA: Für $n = 2m - h$ gibt es einen optimalen Plan mit den folgenden Eigenschaften.

a) $J_1, \ldots, J_h$ laufen auf separaten Maschinen

b) Die restlichen Jobs sind wie folgt gepaart: $(J_{h+1}, J_n), (J_{h+2}, J_{n-1}), \ldots, (J_m, J_{m+1})$



Beweis: Betrachte optimalen Plan, wobei alle Maschinen mindestens einen Job haben.

a) Ist Eigenschaft a) verletzt, dann sei k der kleinste Index $1 \leq k \leq h$, so dass J_k Partner J_s hat. Dann existiert J_j mit $k < j \leq n$, der separat läuft. Wir ändern den Plan so um, so dass J_k alleine läuft und J_j mit J_s gepaart ist. Dann gilt: $p_j + p_s \leq p_k + p_s$, denn $p_k \geq p_j$. Der Makespan ändert sich dadurch also nicht.

b) Sei ein Plan gegeben, der a) erfüllt. Wir zeigen, dass $J_{h+1}, \ldots, J_m$ auf separaten Maschinen laufen. Wir nehmen an, dass J_1 und J_k mit $h+1 \leq j,k \leq m$ gemeinsam laufen. Dann gibt es $m+1 \leq s,t \leq n$, so dass die Jobs J_s und J_t zusammen laufen. Dann können die Jobs auch so eingeplant werden, dass wir die Paare (J_j, J_s) und (J_k, J_t) erhalten. Es gilt

$$p_1 + p_s \leq p_j + p_k \quad \text{denn} \quad p_s \leq p_k$$

$$p_k + p_t \leq p_k + p_j \quad \text{denn} \quad p_t \leq p_j$$

Es seien $(J_{h+1}, J_n), (J_{h+2}, J_{n-1}), \ldots, (J_{h+j}, J_{n+1-j})$ gepaart, aber J_{h+j+1} nicht mit J_{n-j} gepaart.

J_{h+j+1} mit J_s mit $m+1 \leq s < i-j$

J_t mit J_{n-j} mit $h+j+1 < t \leq n$.

$p_{n+j+1} + p_{n-j} \leq p_{h+j+1} + p_s$, denn $p_{n-j} \leq p_s$.

$p_s + p_t \leq p_s + p_{h+j+1}$ □

Beweis des Satzes: Wir nehmen an, dass SLS keinen Faktor von 4/3 erzielt. Sei $p_1 \geq \ldots \geq p_n$ eine kürzeste Sequenz, die den Faktor von 4/3 verletzt. Sei C der Makespan von SLS, opt der optimale Makespan und S_k der Startzeitpunkt von J_k in SLS' Plan. Außerdem sei J_l der Job, der als letztes fertig wird und den Makespan definiert.

1. Fall: $p_l \leq 1/3 \cdot \text{OPT}$. Zum Zeitpunkt S_l sind alle Maschinen beschäftigt, also ist $\text{OPT} \geq S_l$ und $\text{SLS} = S_l + p_l \leq S_l + \frac{1}{3}S_l$ und damit $c = S_l + p_l \leq 4/3 \cdot \text{OPT}$.

2. Fall: $p_l > 1/3 \cdot \text{OPT}$. Es gilt $l = n$, da die gegebene Anfragesequenz die kürzeste Sequenz ist, die den Approximationsfaktor verletzt. Im optimalen Plan können auf jeder Maschine höchstens zwei Jobs sein, denn sonst wären auf einer Maschine drei Jobs mit Lasten größer $1/3 \cdot \text{OPT}$ und damit wäre die Last auf dieser Maschine größer als OPT. Also ist $n \leq 2m$. Sei $h > 0$, so dass $n = 2m - h$ die tatsächliche Anzahl Jobs ist.

SLS platziert $J_1, \ldots, J_h$ auf separaten Maschinen. Wenn die übrigen Jobs nicht gemäß Lemma Teil b) platziert werden, so ist die Maschinenlast nur geringer und wir bleiben optimal. □

2.3 Approximationsschemata

Definition: Ein *Approximationsschema* für ein Optimierungsproblem ist eine Familie

$$\{A(\varepsilon) \mid \varepsilon > 0\} \quad ,$$

so dass $A(\varepsilon)$ ein Approximationsalgorithmus ist, der eine Approximationsgüte von $1+\varepsilon$ bei Minimierungsproblemen bzw. $1-\varepsilon$ bei Maximierungsproblemen erzielt.

2.3.1 Rucksackproblem

n Gegenstände mit Gewichten $g_1, \ldots, g_n \in \mathbb{N}$ und Nutzen $w_1, \ldots, w_n \in \mathbb{N}$. Rucksack mit Gewichtsschranke $b \in \mathbb{N}$. Gesucht ist $I \subseteq \{1, \ldots, n\}$, so dass

$$g(I) := \sum_{j \in I} g_j \leq b \qquad \text{und} \qquad w(I) \sum_{j \in I} w_j \quad \text{maximiert wird.}$$

Ohne Einschränkung können wir dabei annehmen, dass $g_i \leq b$ für alle i.

Wir stellen einen deterministischer Algorithmus für dieses Problem vor. Mit $\mathcal{F}_j(i)$ bezeichnen wir das minimales Gewicht einer Teilmenge von $\{1, \ldots, j\}$, die Nutzen von mindestens i hat. Wir setzen $\mathcal{F}_j(i) = \infty$, wenn keine solche Menge existiert.

$$\text{OPT} = \max\{i \mid \mathcal{F}_n(i) \leq b\}$$

Lemma:

a) $\mathcal{F}_j(i) = \min\{\mathcal{F}_{j-1}(i), g_j + \mathcal{F}_{j-1}(i - w_j)\}$

b) $\mathcal{F}_0(i) = \infty$ für $i > 0$

c) $F_j(i) = 0$ für $i \leq 0$

Algorithmus: ExactKnapsack

```
1 F_0(i) = ∞ für i > 0 und F_j(i) = 0 für i ≤ 0
2 i := 0
3 repeat
4     i := i + 1
5     for j := 1 to n do
6         F_j(i) = min{F_{j-1}(i), g_j + F_{j-1}(i - w_j)}
      end
7 until F_n(i) > b
8 gib i − 1 aus
```

Satz: ExactKnapsack hat eine Laufzeit von $O(n \cdot \text{OPT})$.

Bemerkung: Die Kodierung des Inputs für Knapsack hat eine Länge von

$$O\left(\sum_{j=1}^{n} (\log g_j + \log w_j) + \log b\right) \quad .$$

Unter Umständen ist $\text{OPT} = \sum_{j=1}^{n} w_j$, also ist die Laufzeit von ExactKnapsack im Allgemeinen nicht polynomiell.

ALGORITHMUS: ScaledKnapsack (ε)

1 $w_{\max} := \max_{1 \le i \le n} w_i$
2 $k := \max\{1, \lfloor \frac{\varepsilon \cdot w_{\max}}{n} \rfloor\}$
3 **for** $j = 1$ ***to*** n **do**
4 $\quad w_j(k) := \lfloor \frac{w_j}{k} \rfloor$
end
5 Berechne mit ExactKnapsack die optimale Lösung $S(k)$ mit Wert OPT(k) bezüglich der Eingabe $(g, w(k), b)$.
6 Gib $\text{opt}^* = \sum_{j \in S(k)} w_j$ aus.

SATZ: ScaledKnapsack hat eine Laufzeit von $O(1/\varepsilon \cdot n^3)$.

Beweis: Es gilt $\text{OPT}(k) \le n\frac{w_{\max}}{k}$, also erhalten wir eine Laufzeit von $O(n^2 \frac{w_{\max}}{k})$.

1. Fall: $k = 1$. Wegen Zeile 2 ist dann $\frac{\varepsilon w_{\max}}{n} < 2$. Also ist $\frac{w_{\max}}{k} = w_{\max} < \frac{2n}{\varepsilon}$.

2. Fall $k > 1$. Also $k = \lfloor \frac{\varepsilon \cdot w_{\max}}{n} \rfloor$ und damit:

$$\frac{\varepsilon \cdot w_{\max}}{n} < k + 1$$
$$\frac{w_{\max}}{k} < \frac{n}{\varepsilon}\left(1 + \frac{1}{k}\right) \le \frac{2n}{\varepsilon}$$

□

SATZ: Scaled Knapsack hat eine Approximationsgüte von $1 - \varepsilon$.

Beweis: Sei S eine optimale Lösung bezüglich w_j.

$$\begin{aligned}
\text{OPT}^* = \sum_{j \in S(k)} w_j = k \sum_{j \in S(k)} \frac{w_j}{k} &\ge k \sum_{j \in S(k)} \left\lfloor \frac{w_j}{k} \right\rfloor \\
&\ge k \sum_{j \in S} \left\lfloor \frac{w_j}{k} \right\rfloor \\
&\ge k \sum_{j \in S} \left(\frac{w_j}{k} - 1 \right) \\
&= \sum_{j \in S} w_j - k|S| \\
&= \text{OPT} - k|S| = \text{OPT}\left(1 - \frac{k|S|}{\text{OPT}}\right) \ge \text{OPT}\left(1 - \frac{kn}{w_{\max}}\right)
\end{aligned}$$

Fall 1: $k = 1$: $\text{OPT}^* = \text{OPT}$

Fall 2: $k > 1$, d.h. $k = \lfloor \frac{\varepsilon \cdot w_{\max}}{n} \rfloor$: Dann gilt $k \le \frac{\varepsilon \cdot w_{\max}}{n}$ und damit $\frac{kn}{w_{\max}} \le \varepsilon$. □

2.3.2 Scheduling

Nun stellen wir ein Approximationsschema für das Scheduling Problem mit konstantem m vor.

ALGORITHMUS: SLS(k)

1 Sortiere Jobs $J_1, \dots J_n$ nach ihrer Bearbeitungszeit, so dass $p_1 \geq p_2 \geq \dots \geq p_n$.
2 Berechne opitimales Scheduling für $J_1, \dots J_k$.
3 Vertausche Jobs $J_{k+1}, \dots, J_n$ mit Greedy.

SATZ: Für jedes ε und $k \geq \lceil \frac{n-1}{\varepsilon} \rceil$ hat SLS(k) Approximationsgüte $(1+\varepsilon)$.

Beweis: Sei J_l der Job, der den Makespan c definiert.

Fall 1: $l \leq k$. Also $c = \text{OPT}(J_1, \dots, J_k) \leq \text{OPT}$ und damit $c = \text{OPT}$.

Fall 2: $l > k$. Analog zu der Analyse von Greedy ergibt sich:

$$mc \leq \sum_{j=1}^{n} p_i + (m-1)p_l$$

$$c \leq \frac{1}{m} \sum_{j=1}^{n} p_i + (1 - 1/m)p_l \leq \text{OPT} + (1 - 1/m)p_l$$

p_l können wir folgendermaßen abschätzen:

$$\text{OPT} \geq \frac{1}{m} \sum_{j=1}^{n} p_j \geq \frac{1}{m} \sum_{j=1}^{k} p_j \geq \frac{1}{m} \sum_{i=1}^{k} p_l = \frac{kp_l}{m}$$

$$p_l \leq \frac{m}{k} \cdot \text{OPT}$$

Also gilt:

$$c \leq \text{OPT} + \frac{m-1}{k} \text{OPT} = \left(1 + \frac{m-1}{k}\right) \cdot \text{OPT}$$

Nach Voraussetzung ist $k \geq \lceil \frac{m-1}{\varepsilon} \rceil \geq \frac{m-1}{\varepsilon}$. Damit ist $\frac{m-1}{k} \leq \varepsilon$. Also ist $c \leq (1+\varepsilon) \cdot \text{OPT}$ für $k \geq \lceil \frac{m-1}{\varepsilon} \rceil$. □

SATZ: Die Laufzeit von SLS(k) ist $O(n \log n + (n-k)m + m^k) \subset O(n^2)$.

Beweis: Das Sortieren der Liste liegt in $O(n \log n)$ und das Finden des optimalen Schedule ist in $O(m^k)$ möglich. Für das Einfügen der restlichen $n-k$ Jobs mit Greedy müssen wir jeweils nach der minimal belasteten Maschine suchen, also ergibt sich dafür eine Laufzeit von $O((n-k)m)$. □

2.4 Max-SAT und Randomisierung

Wir betrachten das Problem Max–$\geq k$–SAT. Als Eingabe erhalten wir Klauseln $C_1, \ldots, C_m$ über Boolschen Variablen $x_1, \ldots x_n$, wobei

$$C_i = \ell_{i,1} \vee \ell_{i,2} \vee \ldots \vee \ell_{i,k_i} \text{ mit } k_i \geq k \quad \text{und} \quad \ell_{i,j} \in \{x_1, \overline{x}_1, \ldots, x_n, \overline{x}_n\} \text{ für } j = 1 \ldots, k_i \quad .$$

Finde eine Belegung der Variablen, die möglichst viele Klauseln erfüllt.

Beispiel: $C_1 = x_1 \vee x_2 \vee \overline{x}_3$, $C_2 = x_1 \vee x_3$, $c_3 = x_2 \vee \overline{x}_3$. Für $x_3 = 0$ und $x_1 = x_2 = 1$ ist die Klauselmenge erfüllt.

```
   Algorithmus: RandomSAT
1  for i := 1 to n do
2      Wähle Zufallsbit z ∈ {0,1} gemäß Gleichverteilung
3      if z = 1 then
4          x_i := 1
       else
5          x_i = 0
       end
   end
6  Gib Belegung aus.
```

Satz: Die erwartete Anzahl der erfüllten Klauseln von Random-SAT ist mindestens $(1 - 1/2^k) \cdot m$

Beweis: Für jede Klausel C_i definieren wir die Zufallsvariable

$$X_i = \begin{cases} 1 & \text{wenn Klausel } C_i \text{ erfüllt in RandomSAT's Belegung} \\ 0 & \text{sonst} \end{cases} \quad .$$

Dann ist $X = \sum_{i=1}^{m} X_i$ die Anzahl der erfüllten Klauseln.

$$E[X] = \sum_{i=1}^{m} E[X_i] = \sum_{i=1}^{m} P(X_i = 1) = \sum_{i=1}^{m} (1 - P(X_i = 0))$$

$X_i = 0$ gilt genau dann, wenn alle in C_i vorkommenden Literale nicht erfüllt werden. Also ist $P(X_i = 0) = (1/2)^{k_i}$ und damit $E[X] \geq m(1 - 1/2^k)$. □

Definition: Ein randomisierter Approximationsalgorithmus ist ein Approximationsalgorithmus, der Zufallsentscheidungen treffen darf. In polynomieller Zeit wird eine Zufallszahl aus $\{1, \ldots, n\}$ erzeugt, wobei die Kodierungslänge von n polynomiell in der Kodierungslänge der Eingabe I ist.

DEFINITION: Sei A ein randomisierter Approximationsalgorithmus für ein Optimierungsproblem Π mit den Eingabeinstanzen $\mathcal{I}$. Die Approximationsgüte von A auf $I \in \mathcal{I}$ ist

$$\delta_A(I) = \frac{E[w(A(I))]}{\mathrm{OPT}(I)} \quad .$$

A hat erwartete Approximationsgüte von δ, falls

$$\begin{aligned} \delta_A(I) &\leq \delta \quad \forall I \in \mathcal{I} \quad \text{für Minimierungsprobleme} \\ \delta_A(I) &\geq \delta \quad \forall I \in \mathcal{I} \quad \text{für Maximierungsprobleme} \end{aligned}$$

ALGORITHMUS: DetSAT

```
1 for i := 1 to n do
2     Berechne E_0 = E[X | x_j = b_j für j = 1,...i-1, x_i = 0]
3     Berechne E_1 = E[X | x_j = b_j für j = 1,...i-1, x_i = 1]
4     if E_0 ≥ E_1 then
5         x_i = b_i = 0
      else
6         x_i = b_i = 1
      end
  end
```

SATZ: DetSAT erfüllt mindestens $E[X] = (1 - 1/2^k) \cdot m$ viele Klauseln.

Beweis: Wir setzen $E^i = E[X \mid x_j = b_j, j = 1 \ldots, i]$. Damit ist $E^0 = E[X]$. E^n ist die Anzahl der erfüllten Klauseln von DetSAT. Ist $E^i \geq E^{i-1}$, dann folgt $E^n \geq E^0 = E[X]$.

$$\begin{aligned} E^{i-1} &= E[X \mid x_j = b_j \text{ für } j = 1, \ldots, i-1] \\ &= \frac{1}{2} E[X \mid x_j = b_j \text{ für } j = 1, \ldots, i-1, x_i = 0] + \frac{1}{2} E[X \mid x_j = b_j \text{ für } j = 1, \ldots, i-1, x_i = 1] \\ &= \frac{1}{2}(E_0 + E_1) \\ &\leq \max\{E_0, E_1\} \\ &= E[X \mid x_j = b_j, j = 1, \ldots, i] \\ &= E^i \end{aligned}$$

□

SATZ: DetSAT liefert das bestmögliche Ergebnis, da unter der Annahme P ≠ NP kein Approximationsfaktor von $(1 - 1/2^k) + \varepsilon$ mit $\varepsilon > 0$ erreicht werden kann.

2.4.1 Lineare Programmierung und Relaxierung

Nun betrachten wir einen anderen Ansatz mit *Linearer Programmierung* und *Relaxierung*.

DEFINITION: Ein *Lineares Programm* ist eine Menge von linearen Gleichungen, eine Zielfunktion und eine Menge von Randbedingungen. Die Ausführung des Linearen Programms besteht in dem finden einer Variablenbelegung, die die Randbedingungen erfüllt und die Zielfunktion maximiert.

DEFINITION: *Integer Linear Programming (ILP)* ist lineare Programmierung, mit der zusätzlichen Einschränkung, dass nur ganzzahlige Variablenbelegungen gültig sind.

Wir formulieren Max-SAT als lineares Programm. Die Eingabe für Max-SAT ist eine Formel φ bestehend aus $C_1, \ldots, C_m$ über $x_1, \ldots, x_n$.

Das lineare Programm sieht folgendermaßen aus. Für jede Boolesche Variable x_i führe LP-Variable y_i ein mit

$$y_i = \begin{cases} 1 & \text{wenn } x_i = 1 \\ 0 & \text{sonst} \end{cases} .$$

Für jede Klausel C_j führe Variable z_j ein, so dass

$$z_j = \begin{cases} 1 & \text{wenn } C_j \text{ erfüllt} \\ 0 & \text{sonst} \end{cases} .$$

Für jede Klausel C_j sei V_j^+ die Menge der unnegierten Variablen in C_j und V_j^- die Menge der negierten Variablen.

Damit können wir dieses Problem als lineares Programm formulieren. Zu maximieren ist $\sum_{j=1}^{m} z_j$ unter den folgenden Nebenbedingungen:

$$\sum_{i:x_i \in V_j^+} y_i + \sum_{i:x_i \in V_j^-} (1 - y_i) \geq z_j \text{ für } j = 1, \ldots, m.$$

$$y_i, z_j \in \{0, 1\} \text{ für } i = 1, \ldots, n \text{ und } j = 1, \ldots, m$$

SATZ: ILP (Integer Linear Programming) ist NP-hart.

SATZ:[Khachyion, 1980] LP ist in P.

DEFINITION: *Relaxierung* eines ILP-Programms ist die Aufhebung der Einschränkung auf ganzzahlige Lösungen.

Da ein relaxiertes Programm aus einem größeren Lösungsraum schöpfen kann, ist die Lösung des relaxierten Problems mindestens so gut wie die Lösung des Ausgangsproblems.

Das relaxierte Programm zu MaxSAT sieht wie folgt aus. Zu maximieren ist $\sum_{j=1}^{m} z_j$ unter den folgenden Nebenbedingungen:

$$\sum_{i:x_i \in V_j^+} y_i + \sum_{i:x_i \in V_j^-} (1 - y_i) \geq z_j \text{ für } j = 1, \ldots, m.$$

$$y_i, z_j \in [0, 1] \text{ für } i = 1, \ldots, n \text{ und } j = 1, \ldots, m$$

Um aus der Lösung des relaxierten MaxSAT Programms eine approximative Lösung für MaxSAT zu erhalten, fassen wir $y_i \in [0, 1]$ als Wahrscheinlichkeit dafür auf, dass x_i den Wert 1 bekommt.

ALGORITHMUS: RR-SAT (Randomized Rounding SAT)

```
1 Sei (ŷ_1, ... ŷ_n, ẑ_1, ... ẑ_m) die optimale Lösung des relaxierten ILP für MaxSAT.
2 for i = 1 to n do
3     Ziehe Zufallszahl z ∈ {0,1}, so dass z = 1 mit Wahrscheinlichkeit ŷ_i.
4     if z = 1 then
5         x_i := 1
      else
6         x_i = 0
      end
  end
```

SATZ: RRSAT erzielt eine Approximationsgüte von $(1 - 1/e) \approx 0.632$. Die erwartete Anzahl an erfüllten Klauseln ist $(1 - 1/e) \cdot m$.

LEMMA 1: Seien $u_1, \ldots, u_k$ positive reelle Zahlen mit $0 \leq u_i \leq 1$ und $\sum_{i=1}^{k} u_i \geq u$. Dann gilt:

$$\prod_{i=1}^{k} (1 - u_i) \leq (1 - u/k)^k$$

Beweis: Seien $a_1, \ldots, a_n$ nicht negative ganze Zahlen. Dann ist

$$\left(\prod_{i=1}^{k} a_i \right)^{1/k} \geq \frac{1}{k} \sum_{i=1}^{k} a_i$$

Sei $a_i = 1 - u_i$.

$$\prod_{i=1}^{k} (1 - u_i) \leq \left(\frac{1}{k} \sum_{i=1}^{k} (1 - u_i) \right)^k = \left(1 - \frac{\sum u_i}{k} \right)^k \leq (1 - u/k)^k$$

□

LEMMA 2: Sei $k \in \mathbb{N}, 0 \leq x \leq 1$. Dann gilt:

$$1 - (1 - x/k)^k \geq \left(1 - (1 - 1/k)^k\right) x$$

Beweis: Wir betrachten $g(x) = 1 - (1 - x/k)^k$. Es gilt $g'(x) = k(1 - x/k)^{k-1}$. Also ist g monoton steigend und konkav. Damit gilt: $g(x) \geq (1 - (1 - 1/k)^k)x$ □

Beweis des Satzes: Sei $\mathrm{OPT}(\varphi)$ die Anzahl der erfüllten Klauseln einer optimale Lösung für die gegebene Formel. Dann gilt $\sum_{j=1}^{m} \hat{z}_j \geq \mathrm{OPT}(\varphi)$. Für alle C_j betrachten wir die Zufallsvariablen

$$X_j = \begin{cases} 1 & \text{wenn } C_j \text{ erfüllt} \\ 0 & \text{sonst} \end{cases} .$$

$E[\#\text{erfüllte Klauseln}] = \sum_{j=1}^{m} E[X_j] = \sum_{j=1}^{m} P(X_j = 1)$.

$$P(X_j = 0) = \prod_{i:x_i \in V_j^+} (1 - \hat{y}_i) \prod_{i:x_i \in V_j^-} \hat{y}_i$$

Sei k_j die Anzahl der Literale in C_j. Mit den gerade gezeigten Lemmata gilt dann:

$$P(X_j = 0) = \prod_{i:x_i \in V_j^+} (1 - \hat{y}_i) \prod_{i:x_i \in V_j^-} (1 - (1 - \hat{y}_i)) \geq (1 - \hat{z}_j/k_j)^{k_j} \qquad \text{(nach Lemma 1)}$$

$$P(X_j = 1) \geq 1 - (1 - z_j/k_j)_j^k \geq \left(1 - (1 - 1/k_j)^{k_j}\right) \hat{z}_j \geq (1 - 1/k_j)\hat{z}_j \qquad \text{(nach Lemma 2)}$$

Damit gilt dann:

$$\sum_{k=1}^{m} P(X_j = 1) \geq (1 - 1/l \sum_{j=1}^{m} \hat{z}_j \geq (1 - 1/l\, \mathrm{OPT}(\varphi)$$

□

SATZ: Für gegebene Inputformel φ wende RandomSAT und dann RRMaxSAT an und nehme die bessere der beiden Lösungen. Dann erhalten wir einen Approximationsfaktor von $3/4$.

Beweis: Sei φ die Eingabe, n_1 bzw. n_2 die erwartete Anzahl der erfüllten Klauseln von RandomSAT bzw. RRMaxSAT. Es gilt:

$$n_1 = \sum_{j=1}^{m}(1 - 1/2^{k_j}) \geq \sum_{j=1}^{m}(1 - 1/2^{k_j})\hat{z}_j \quad \text{und} \quad n_2 = \sum_{j=1}^{m}\left(1 - (1 - 1/k_j)^{k_j}\right)\hat{z}_j$$

Zunächst gilt:

$$\left(1 - \frac{1}{k}\right)^k = \sum_{i=0}^{k}\binom{k}{i}\left(\frac{-1}{k}\right)^i \leq 1 - \binom{k}{i}\frac{1}{k} + \binom{k}{2}\frac{1}{k^2} + \underbrace{\ldots}_{\leq 0} = \frac{k-1}{2k} = \frac{1}{2} - \frac{1}{2k} \leq \frac{1}{2}\frac{1}{2^k}$$

Damit erhalten wir:

$$\begin{aligned}
\max\{n_1, n_2\} &\geq \frac{1}{2}\sum_{j=1}^{m}\left(\left(1 - \frac{1}{2^{k_j}}\right) + \left(1 - \left(1 - \frac{1}{k_j}\right)^{k_j}\right)\right)\hat{z}_j \\
&\geq \frac{1}{2}\sum_{j=1}^{m}\left(1 - \frac{1}{2^{k_j}} + 1 - \left(\frac{1}{2} - \frac{1}{2k_j}\right)\right)\hat{z}_j \\
&= \frac{3}{4}\sum_{j=1}^{m}\hat{z}_j \\
&\geq \frac{3}{4}\ \mathrm{OPT}(\varphi)
\end{aligned}$$

□

Problem HittingSet: Gegeben ist eine Menge $V = \{v_1, \ldots, v_n\}$ und $S_1, \ldots, S_m \subseteq V$. Gesucht wird eine möglichst kleine Menge $H \subseteq V$ mit $H \cap S_j \neq \emptyset$ für alle $j = 1, \ldots, m$. H wird auch HittingSet genannt.

Formulierung als lineares Programm:

Für v_i LP-Variable x_i mit $x_i = \begin{cases} 1 & \text{wenn } v_i \in H_{\mathrm{OPT}} \\ 0 & \text{wenn } v_i \notin H_{\mathrm{OPT}} \end{cases}$

Zu minimieren ist $\sum_{i=1}^{n} x_i$ unter den Nebenbedingungen

$$\sum_{i:v_i \in S_j} x_i \geq 1 \quad \text{für} j = 1, \ldots, m$$

$$x_i \in \{0, 1\} \quad \text{für} i = 1, \ldots, n$$

Relaxiertes LP: wie oben mit $x_i \in [0, 1]$.

ALGORITHMUS: RRHS (Randomized Rounding Hitting Set)

```
1 Sei (x̂_1, ... x̂_n) optimale Lösung des relaxierten LP.
2 H := ∅
3 for i = 1 to ⌈log(2m)⌉ do
4     Wähle z ∈ {0,1}, so dass z = 1 mit Wahrscheinlichkeit x̂_j.
5     if z = 1 then
6         H := H ∪ {v_i}
      end
  end
7 Gib H aus.
```

SATZ: Für jede Probleminstanz gilt:

a) RRHS liefert mit WSK $\geq 1/2$ eine korrekte Lösung.

b) $E[|H \text{ von RRHS berechnet}|] \leq \lceil \log(2m) \rceil \, \mathrm{OPT}(I)$

Beweis:

a) Sei $|S_j| = k_j$ und H_i die Menge der Elemente, die in der i-ten Iteration zu h hinzugefügt werden. Dabei zählen wir auch diejenigen Elemente mit, die schon in der Menge waren.

$$P(s_j \cap H_i = \emptyset) = \underbrace{\prod_{i:v_i \in S_j} (1 - \hat{x}_j)}_{k_j \text{ Terme}} \leq (1 - 1/k_j)^{k_j} \leq 1/e$$

Es folgt

$$P(S_j \cap H = \emptyset) = (1/e)^{\log 2m} \leq \frac{1}{2m}$$

b) Sei H die Menge der Elemente die hinzugefügt wurden. Dann gilt:

$$E[|H_i|] = \sum_{i=1}^{m} \hat{x}_i \leq \mathrm{OPT}(I)$$

Damit ist $H \leq \bigcup_{i=1}^{\lceil \log 2m \rceil} H_i$. Es folgt

$$E[|H|] \leq \lceil \log(2m) \rceil \, \mathrm{OPT}(i) \leq m \cdot \frac{1}{2m} = \frac{1}{2}$$

□

2.5 SetCover und probabilistische Approximationsalgorithmen

DEFINITION: Ein probabilistischer Approximationsalgorithmus ist ein randomisierter Approximationsalgorithmus, der mit einer Wahrscheinlichkeit von mindestens 1/2 eine zulässige Lösung liefert.

SATZ: Sei p ein Polynom und A ein probabilistischer Approximationsalgorithmus, der mit einer Wahrscheinlichkeit von $1/p(|I|)$ eine korrekte Lösung liefert. Dann gibt es für alle $\varepsilon > 0$ einen Algorithmus A_ε, der mit Wahrscheinlichkeit von mindestens $1 - \varepsilon$ eine korrekte Lösung berechnet. Der Approximationsfaktor bleibt dabei erhalten.

Beweis:

```
ALGORITHMUS: A_ε
for i := 1 to ⌈p(|I|) log 1/ε⌉ do
    Wende A auf I an und berechne eine Lösung S.
    if S zulässige Lösung then
        Gib S aus und breche ab.
    end
end
```

Die Wahrscheinlichkeit, dass keine zulässige Lösung berechnet wird, ist

$$\left(1 - \frac{1}{p(|I|)}\right)^{p(|I|)\log\frac{1}{\varepsilon}} \quad .$$

□

SetCover: Universum $U = \{u_1, \ldots, u_n\}$, Mengen $S_1, \ldots S_m \subseteq U$. Die Kosten einer Menge $c(S_j)$ sind nicht-negative reele Zahlen. Wir suchen $J \subseteq \{1, \ldots, m\}$, so dass

$$\bigcup_{j \in J} S_j = U \qquad \text{und} \qquad \sum_{j \in J} c(S_j) \text{ minimal} \quad .$$

Greedy-Ansatz: Wähle wiederholt die kosteneffektivste Menge. Sei C die Menge der bereits abgedeckten Elemente. Die Kosteneffizienz einer Menge S ist $c(S)/|S-C|$.

```
ALGORITHMUS: Greedy für SetCover
C := ∅
while C ≠ U do
    Ermittle die kosteneffizienteste Menge S
    Wähle S und setze price(e) := c(S)/|S−C| für e ∈ S − c
    C := C ∪ S
end
```

Es ist $\sum_{e \in U} \text{price}(e)$ die Kosten der von Greedy gewählten Mengen. Sei $e_1, \ldots e_n$ die Nummerierung der Elemente in der Reihenfolge, in der sie abgedeckt werden.

LEMMA: Es gilt $\text{price}(e_k) \leq \frac{\text{OPT}}{n-k+1}$ für $k = 1, \ldots, n$. OPT sind dabei die Kosten einer optimalen Lösung.

Beweis: Betrachte die Iteration, in der e_k zum ersten Mal abgedeckt wird. Wenn diese Iteration startet, dann seien $S_{i_1}, \ldots, S_{i_l}$ die von Greedy gewählten Mengen und $S_{i_{l+1}}, \ldots, S_{i_n}$ die noch nicht gewählten Mengen.

Sei $\overline{C}$ die Menge der noch nicht abgedeckten Elemente. Es gibt eine noch nicht gewählte Menge mit Kosteneffizienz höchstens $^{OPT}/_{|\overline{C}|}$.

Wähle aus den noch nicht gewählten Mengen eine optimale Lösung $S_{j_1}, \ldots, S_{j_k}$ aus. Dann gilt

$$c(S_{j_1}) + \ldots + c(S_{j_k}) \leq OPT \quad .$$

Sei n_{i_j} die Anzahl der neu abgedeckten Elemente bezüglich der jetzigen Lösung. Es gilt $n_{j_r} = |S_{j_r} \cap \overline{C}$. Es ist $\bigcup_{r=1}^{k} \supseteq \overline{C}$. Damit ist die Summe $n_{j_1} + n_{j_2} + \ldots + n_{j_k} \geq |\overline{C}|$. Hätten alle $S_{j_1}, \ldots, S_{j_k}$ eine Kosteneffektivität größer als $^{\mathrm{OPT}}/_{|\overline{C}|}$, dann wäre

$$\mathrm{OPT} \leq (n_{j_1} + \ldots + n_{j_k}) \frac{OPT}{\overline{C}} < c(S_{j_1}) + \ldots + c(S_{j_k}) \leq \mathrm{OPT} \quad .$$

Das ist ein Widerspruch. Also gibt es eine Menge mit Kosteneffektivität mit $^{\mathrm{OPT}}/_{\overline{C}}$. Dabei ist $|\overline{C}| \geq n - k + 1$. □

Satz: Der Approximationsfaktor von Greedy ist höchstens $\mathrm{OPT} \cdot H_n$.

Beweis:

$$\sum_{k=1}^{n} \mathrm{price}(e_k) = \sum_{k=1}^{n} \frac{\mathrm{OPT}}{n-k+1} = \mathrm{OPT}(\frac{1}{n} + \frac{1}{n-1} + \ldots + 1) = \mathrm{OPT} \cdot H_n$$

□

Beobachtung: Der Approximationsfaktor von Greedy ist nicht besser als H_n.

Zeichnung

Greedy wählt immer die kleinen Mengen, da deren Kosteneffizienz am besten ist. Damit erzeugt Greedy kosten von H_n. OPT dagegen kommt mit Kosten on $1 + \varepsilon$ aus.

Shortest Superstring: Sei Σ ein endliches Alphabet und $S = \{s_1, \ldots, s_n\}$ mit $s_i \in \Sigma^+$. Gesucht ist der kürzeste String s, der alle $s_i \in S$ als Teilstring enthält. O.B.d.A. ist kein String s_i kein Teilstring von s_j für $i \neq j$.

Reduktion auf SetCover: Universum $U = \{s_1, \ldots, s_n\}$. Wir betrachten s_i und s_j, so dass sich die letzten k Stellen von s_i mit den ersten k Stellen von s_j überlappen. Dann bezeichnen wir mit σ_{ijk} die Überlappung von s_i und s_j, die dann noch $i + j - k$ Zeichen hat. Nun setzen wir

$$M := \{\sigma_{ijk} \mid \text{für alle möglichen Kombination von } i, j, k\}$$

und

$$\text{Set}(\pi) = \{s_i \in S \mid s_i \text{ in } \pi \text{ enthalten}\} \quad .$$

Als Mengen wählen wir $\text{Set}(\pi)$ für alle $\pi \in M \cup U$ und als Kosten $c(\text{Set}(\pi)) = |\pi|$. Damit erhalten wir folgenden Algorithmus.

1. Wende Greedy-Algorithmus für das SetCover auf die obige Probleminstanz an.
2. Seien $\text{Set}(\pi_1), \ldots, \text{set}(\pi_k)$ gewählten Menge. Konkateniere $\pi_1 \circ \ldots \circ \pi_k$ in beliebiger Reihenfolge.

Satz: $\text{OPT} \leq \text{OPT}_{\text{SC}} \leq 2\,\text{OPT}$, wobei OPT_{SC} die Kosten der SetCover-Instanz und OPT die Länge des kürzesten Superstrings ist.

Korollar: Der obige Algorithmus ist $2H_n$-Approximation.

Beweis: Sei $\{\text{Set}(\pi_1), \ldots, \text{Set}(\pi_k\}$ optimale Lösung der SetCover-Instanz. Dann enthält $\pi_1 \circ \ldots \circ \pi_k$ alle $s_i \in U$, und $|\pi_1 \circ \ldots \circ \pi_k| = \sum_{i=1}^{n} |\pi_i| \geq \text{OPT}$. Sei s der kürzeste Superstring von $s_1, \ldots, s_n$. Dann ist $|s| = \text{OPT}$.

Wir zeigen: Es gibt eine SetCover-Lösung mit Kosten von $2 \cdot \text{OPT}$.

Da es keine zwei Strings gibt, so dass der eine den anderen enthält, sind alle Anfangs- und Endpositionen der Teilstrings in s paarweise verschieden.

Wir setzen S_{b_i} als den ersten noch nicht gruppierten String und S_{e_i} als den letzten String, der noch mit S_{b_i} überlappt. Als Gruppierungen wählen wir dann den Bereich π_i, der beim ersten Zeichen von S_{b_1} beginnt und bei dem letzten Zeichen von S_{e_i} endet. $\text{Set}(\pi_i)$ enthält genau die Teilstrings aus Gruppe i. Also ist $\text{Set}(\pi_1), \ldots, \text{Set}(\pi_k)$ eine zulässige Lösung der SetCover-Instanz

Es gilt $\sum_{i=1}^{k} |\pi_i| \leq 2|s| = 2\,\text{OPT}$, da jeder Buchstabe von s von höchstens zwei Mengen π_i abgedeckt, denn π_i und π_{i+1} sind disjunkt, da $s_{b_{i+2}}$ nicht mit $s_{b_{i+1}}$ überlappen kann und damit auch nicht mit s_{e_i} überlappt. □

Entwicklung einer 4-Approximation: Untere Schranke für $\text{OPT} = |s|$? Sei s der kürzeste Superstring. Markiere jeweils die ersten Vorkommen der Strings $s_1, \ldots, s_n$. Mit $\text{prefix}(s_i, s_j)$ bezeichnen wir das Präfix von s_i, das in s_j nicht enthalten ist. Am Ende ergänzen wir das durch $\text{prefix}(s_n, s_1)$ und erhalten noch einen Bereich $\text{overlap}(s_n, s_1)$, wobei $\text{overlap}(s_i, s_j)$ die längste Überlappung von s_i und s_j ist. Dann ist

$$|s| = \text{OPT} = |\text{prefix}(s_1, s_2)| + \ldots + |\text{prefix}(s_{n-1}, s_n)| + |\text{prefix}(s_n, s_1)| + |\text{overlap}(s_n, s_1)|$$

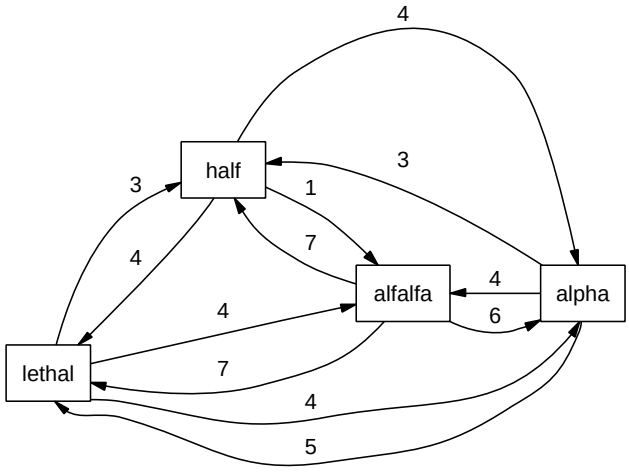


Abbildung 5: Beispiel für einen Präfixgraphen

DEFINITION: Präfixgraph. Ein Knoten für jedes s_i und eine Kante (s_i, s_j) mit Gewicht $|\text{prefix}(s_i, s_j)|$.

OP ist mindestens so groß wie das Gewicht einer TSP-Tour durch den Präfix-Graphen, der allerdings nicht in polynomieller Zeit berechnet werden kann. Daher suchen wir nur eine Zyklenabdeckung, also eine Menge von Zyklen, die jeden Knoten genau einmal enthält. Eine minimale Zyklenabdeckung ist eine zyklenabdeckung minimalen Gesamtgewichts.

Wir betrachten einen bipartiten Graphen, in den wir die Knoten $s_1, \ldots, s_n$ auf beiden Seite einmal einfügen und mit $|\text{prefix}(s_i, s_j)$ bewertete Kanten zwischen s_i auf der linken Seite und s_j auf der rechten Seite ein. Wir berechnen in diesem Graphen ein Matching, das uns Zyklen im ursprünglichen Gräfixgraphen liefert. Jede Rundreise ist mindestens so teuer wie eine Zyklenabdeckung.

Wir betrachten einen Zyklus $c : i_1 \to i_2 \to \ldots \to i_k \to i_1$. Dann ist

$$\alpha(c) = \text{prefix}(s_{i_1}, s_{i_2}) \circ \text{prefix}(s_{i_2}, s_{i_3}) \circ \ldots \circ \text{prefix}(s_{i_k}, s_{i_i}) \quad .$$

Dann $\sigma(c) = \alpha(c) \circ s_{i_k}$.

Algorithmus:

1. Berechne minimalen Zyklenabdeckung und erhalte Zyklen $c_1, \ldots, c_l$.
2. Gib $\sigma(c_1) \circ \ldots \circ \sigma(c_l)$

SATZ: Dieser Algorithmus liefert eine 4-Approximation.