panikzettel.htwr-aachen.de

Effiziente Algorithmen Panikzettel™

Luca Oeljeklaus, Christoph von Oy, Tobias Polock,
Philipp Schröer, Caspar Zecha

Version 1 — 04.08.2024

Inhaltsverzeichnis

1 Einleitung

Dies ist ein Panikzettel. Doch, doch, du hast richtig gelesen. Wir sind leider auf der Maus ausgerutscht und dabei ist ein kleines Buch entstanden.

Wir haben wieder unnötigen Formalismus weggelassen und Erklärungen so intuitiv gemacht, dass jedes Problem mindestens trivial wirkt. Zum ersten Mal gibt es auch Beweise, die den Vorhang der Hexerei hinter einigen Algorithmen aufdecken und mit groben Mengen Formalismus untermauern.

Der Panikzettel basiert auf der Vorlesung "Effiziente Algorithmen" bei PD Dr. Walter Unger im Wintersemester 2017/18.

Aufgrund der unerwarteten Tiefen des Vorlesungsstoffes haben wir in den 41 Seiten noch nicht jede Kleinigkeit erklärt. Daher haben wir im letzten Kapitel einige Stellen aufgelistet, an denen du (ja du!) auch selber Hand anlegen kannst.

2 Flussprobleme

2.1 Maximum Flow

Bei dem *Flussproblem* versuchen wir den größten Fluss durch Kanten mit Kapazitäten c von einer Quelle s zu seiner Senke t zu senden.

Mehrere Quellen S / Senken T sind möglich, indem man von den Knoten aus S bzw. T einfach Kanten mit unendlicher (ausreichend großer) Kapazität mit einer *Superquelle* bzw. *Supersenke* verbindet.

Definition: Flussproblem

Eingabe: $G = (V, E, s, t, c)$ mit:

- (V, E) gerichteter Graph,
- $s, t \in V$ und $s \neq t$,
- $c : E \to \mathbb{N}^+$

Ausgabe: $f : E \in \mathbb{R}_0^+$ mit:

- $\forall\, e : f(e) \in [0, c(e)]$,
- $\forall\, v \in V \setminus \{ s, t \}$:
  $\sum_{(a,v)\in E} f((a,v)) = \sum_{(v,a)\in E} f((v,a))$

Ziel: Maximiere $w(f) = \sum_{(s,v)\in E} f((s,v))$.

Das *Restnetzwerk* zu G und f enthält Kanten mit Gewichten entsprechend möglicher Flussveränderung. Explizit sind auch Rückwärtskanten enthalten, um einen Fluss rückgängig zu machen.

Ein s-t-Pfad im Restnetzwerk ist immer ein *vergrößernder Pfad*, d.h. über ihn kann man zusätzlichen Fluss schicken.

Definition: Restnetzwerk

Zu einem Netzwerk $G = (V, E, s, t, c)$:

Ein *Restnetzwerk* G_f zu einem Fluss f hat Kanten $(v, w) \in V_G^2$ mit Gewicht $\text{rest}_f(v, w) > 0$.

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

2.1.1 Min-Cut Max-Flow

Ein *Schnitt* (S, T) teilt die Knoten in zwei disjunkte Mengen auf. Die *Kapazität des Schnittes* ist $\sum c(v, w)$ mit $v \in S, w \in T$ (also nur mit Kanten in einer Richtung!).

Für jeden Fluss f und jeden Schnitt (S, T) gilt: $w(f) = f(S, T) \leq c(S, T)$.

Satz: Min-Cut Max-Flow

Sei f ein Fluss auf Netzwerk $G = (V, E, s, t, c)$.

Dann sind äquivalent:

1. f ist maximaler Fluss.
2. Das Restnetzwerk G_f enthält keinen vergrößernden Pfad.
3. Es gibt einen Schnitt (S, T) mit $w(f) = c(S, T)$.

Grobe Beweisideen:

- $3 \Rightarrow 1$: Wegen Max-Flow $\leq$ Min-Cut $\leq c(S,T)$ ist f maximal.
- $1 \Rightarrow 2$: Wenn es einen vergrößernden Weg geben würde, könnte man f mit diesem vergrößern.
- $2 \Rightarrow 3$: Wir teilen den Graphen in von s erreichbare und nicht erreichbare Knoten auf. Per Definition muss dies ein passender Schnitt sein, da es ansonsten noch einen vergrößernden Pfad von s nach t gäbe.

2.1.2 Methode von Ford-Fulkerson, Algorithmus von Edmonds-Karp

Die *Methode von Ford-Fulkerson* sucht einen maximalen Fluss. Der tatsächliche Algorithmus, einen flussverbessernden Weg in G_f zu finden, bleibt abstrakt. Die Laufzeit ist in $\mathcal{O}(C \cdot m_{\text{Weg finden}})$, wobei C der Wert des maximalen Flusses ist. Den größten Fluss kann man auch einfach abschätzen mit $C = \sum_{e \in E} c(e)$.

Algorithmus: Methode von Ford-Fulkerson

Eingabe: Flussnetzwerk $G = (V, E, s, t, c)$.
Ausgabe: Maximaler Fluss f.

1. Setze $f(e) = 0 \quad \forall e \in E$.
2. Wiederhole:
   a) Bestimme Restnetzwerk G_f.
   b) Finde einen flussvergrößernden Weg von s zu t in G_f (BFS/DFS/...).
   c) Gibt es keinen Weg, fertig.
   d) Vergrößere f um den Weg, begrenzt durch den Wert der kleinsten Kante des Weges.

Der *Algorithmus von Edmonds-Karp* ist einfach Ford-Fulkerson mit BFS. Laufzeit: $\mathcal{O}(m^2 \cdot n) = \mathcal{O}(n^5)$.

- Zeige, dass $\mathcal{O}(m \cdot n)$ Iterationen reichen.
  - Zeige $\forall v \in V : \text{dist}_{G_f^i}(s,v) \leq \text{dist}_{G_f^{i+1}}(s,v))$ für jede Iteration i.
    - Gilt für in G_f^i **gelöschte Kanten**.
    - **Neue Kanten** von G_f^i zu G_f^{i+1} per Induktion: Sei (v,w) eine neue Kante. Sie wurde eingefügt, weil sie vorher als Kante (w,v) auf einem flussvergrößernden Weg lag. Also kann $\text{dist}_{G_f^{i+1}}(s,v)$ nicht kleiner sein als in G_f^i. Also gilt $\text{dist}_{G_f^i}(s,v) + 1 = \text{dist}_{G_f^{i+1}}(s,w)$. Weil der Abstand von w zu s sich nicht verbessern kann, gilt die Aussage für jedes $w \in V$.
  - Wir betrachten Kante (v,w) in Iteration i:
    - Falls (v,w) **gelöscht** wird, so gilt: $\text{dist}_{G_f^i}(s,v) < \text{dist}_{G_f^{i+1}}(s,w)$ ("Flaschenhals").
    - Wenn (v,w) **wieder eingefügt** wird, gilt $\text{dist}_{G_f^i}(s,w) + 2 \leq \text{dist}_{G_f^{i+1}}(s,w)$, denn wegen Breitensuche war (v,w) auf kürzestem Weg in G_f^i.
  - Wegen $\text{dist}_{G_f^i}(s,t) \leq n-1$ kann jede Kante höchstens $\lfloor (n-1)/2 \rfloor$ mal gelöscht werden.
  - In jeder Iteration wird mindestens eine Kante gelöscht.
  - In jedem G_f^i sind max. $2 \cdot m$ Kanten.
  - Also maximal $\frac{n}{2} \cdot 2 \cdot m = n \cdot m$.
- BFS ist in $\mathcal{O}(n+m)$.

2.1.3 Algorithmus von Dinitz

Definition: Sperrfluss

In einem Niveaunetzwerk $G'_f = (V, E'_f, s, t, c')$ mit Fluss f' auf G':

- Eine Kante $e \in E'_f$ ist *saturiert*, falls $f'(e) = rest_f(e)$.
- f' heißt *Sperrfluss*, falls jeder Weg von s nach t in G'_f eine saturierte Kante hat.
- Ein Knoten v heißt *saturiert*, falls $\text{pot}(v) = 0$ mit:
  $\text{pot}(v) := \min \left\{ \sum_{e \in N_{\text{in}}(v)} rest_f(e), \sum_{e \in N_{\text{out}}(v)} rest_f(e) \right\}$

Die Idee: Suche gleich mehrere verbessernde Wege in G_f. Dies machen wir in einem *Niveaunetzwerk*, einem Restnetzwerk in dem es nur Kanten zwischen benachbarten Schichten gibt, d.h. nur Kanten zwischen Knoten mit unterschiedlicher Distanz zu s. Darin bestimmen wir einen Sperrfluss.

Die Anzahl der Iterationen ist maximal $n - 1$: In jedem Schritt i werden alle Wege der Länge $l_i = \text{dist}_{G'_f}(s, t)$ getrennt. Neu entstehende Kanten führten vorher von Schicht i zu $i - 1$, damit müssen neue Wege mindestens Länge $l_i + 2$ haben.

Algorithmus: Dinitz

Eingabe: Flussnetzwerk $G = (V, E, s, t, c)$.
Ausgabe: Maximaler Fluss f.

1. Setze $f(e) = 0 \quad \forall e \in E$.
2. Wiederhole:
   a) Bestimme Restnetzwerk G_f.
   b) Bestimme Niveaunetzwerk G'_f.
   c) Bestimme Sperrfluss f' in G'_f.
   d) Falls $w(f') = 0$, fertig.
   e) Sonst erweitere f um f'.

Ein Sperrfluss kann in $\mathcal{O}(n^2)$ mit wiederholter *Forward-Backward-Propagation* bestimmt werden.

Forward Propagation: Knotenpotentiale $\text{pot}(v)$ vorwärts durch das Niveaunetzwerk schieben, beginnend vom Knoten mit kleinstem Potential. *Backward Propagation* ist analog: Statt $V_{\text{out}}(v)$ $V_{\text{in}}(v)$ verwenden. Eine *Propagationsphase* führt Forward und Backward Propagation aus.

Jede Propagationsphase läuft in $\mathcal{O}(n + l_i)$ mit l_i Anzahl neu saturierter Kanten. Nach maximal $n - 1$ Propagationsphasen ist ein Sperrfluss gefunden. Also Laufzeit für die iterierte Propagation: $\mathcal{O}(\sum_{i=1}^{n-1}(n + l_i)) = \mathcal{O}(n^2 + m)$.

Algorithmus: Iterierte Propagation

Eingabe: $G'_f = (V, E'_f, s, t, c')$.
Ausgabe: Sperrfluss f'.

Solange kein Sperrfluss f' berechnet ist, wiederhole:

1. Führe eine *Propagationsphase* aus: Forward, dann Backward Propagation.
2. Entferne alle saturierten Kanten und Knoten.

Algorithmus: Forward Propagation bzw. Backward Propagation

Eingabe: Niveaunetzwerk $G'_f = (V, E'_f, s, t, c')$.
Ausgabe: Neuer Fluss f''.
(Verwende V_{in} für Backward Propagation)

1. Setze $f(e) = 0 \quad \forall e \in E$.
2. Wiederhole:
   a) Bestimme $v = \min_{v \in V, \text{pot}(v) > 0} \text{pot}(v)$.
   b) Setze $U(v) = \text{pot}(v)$ und sonst $U(w) = 0$.
   c) v in neue Schlange Q.
   d) Solange (neues) v aus Q entnehmbar ist, für jedes $e = (v, w) \in V_{\text{out}}(v)$:
      i. $f''(e) = \min\{\text{pot}(e), U(v)\}$.
      ii. $U(v) = U(v) - f''(e)$.
      iii. $U(w) = U(w) + f''(e)$.
      iv. Falls $w \neq t$ und $U(w) = f''(e)$, füge w in Q ein.

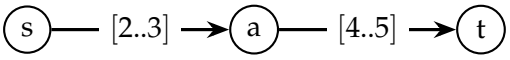
Bonus: BFS, DFS, ...

Die Propagation verwendet eine Schlange Q, also BFS. DFS wäre aber ebenso möglich, hat u.U. schlechtere Laufzeit.

2.2 Mindestfluss

Bei dem *Mindestflussproblem* versuchen wir den größten Fluss durch Kanten mit Kapazitäten c von einer Quelle s zu seiner Senke t zu senden. Dazu besitzt jede Kante eine untere Schranke c' die den kleinsten Fluss darstellt, der mindestens durch diese Kante fließen muss.

Es ist außerdem am folgenden Beispiel festzustellen, dass es nicht immer eine Lösung gibt:



Definition: Mindestflussproblem

Eingabe: Ein Graph analog zum Flussproblem, zusätzlich:

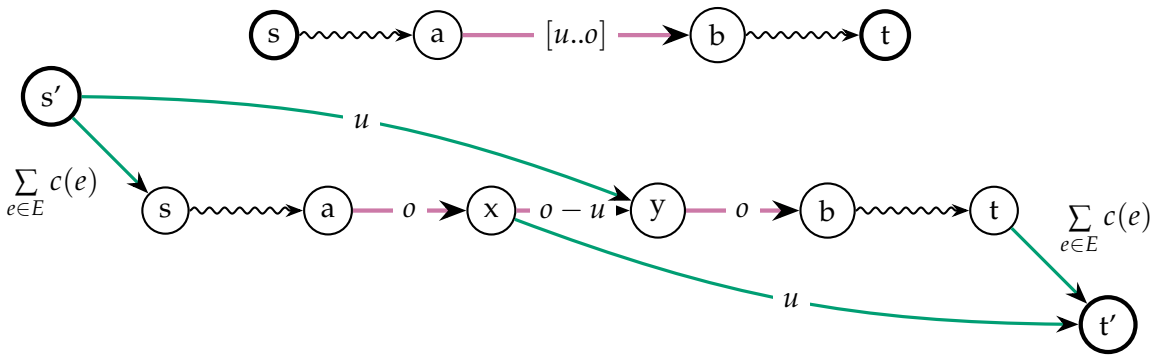
- $c' : E \to \mathbb{N}^+$

Ausgabe: $f : E \in \mathbb{R}_0^+$ analog zum Flussnetzwerk, zusätzlich:

- $\forall e : c'(e) \leq f(e) \leq c(e)$

Ziel: Bestimme, ob eine solcher Fluss existiert. Falls ja, maximiere:

$$w(f) = \sum_{(s,v) \in E} f((s, v))$$



Wir können das Problem mit einer effizienten Reduktion auf das normale Flussproblem lösen.

Dafür wandeln wir mit dem nebenstehenden Algorithmus ein Netzwerk G mit Mindesfluss in ein Netzwerk G' um.

Wenn dann in G gilt $u \leq f(a,b) \leq o$, dann gibt es einen Fluss f' in G', sodass gilt:

$$\begin{aligned} f'(s',y) = f'(x,t') &= u \\ f'(a,x) = f'(y,b) &= f(a,b) \\ f'(x,y) &= f(a,b) - u \end{aligned}$$

Andersherum können wir von einem korrekten Fluss in G' wieder zu einem Fluss in G kommen. Wenn alle Kanten (s',y) und (x,t') saturiert sind, gilt $f'(a,x) = f'(y,b)$ und damit auch:

$$f(a,b) = f'(a,x)$$

Oder anders gefragt: Was wäre, wenn $f(a,b) < u$ in G? Dann ist in G' mindestens $s \rightsquigarrow a$ oder $b \rightsquigarrow t$ begrenzend. Also kann (s',y) oder (x,t') nicht saturiert werden.

Algorithmus: Reduktion des Mindestflussproblems auf das Flussproblem

Eingabe: $G = (V,E,s,t,c,c')$
Ausgabe: $G' = (V',E',s',t',c'')$.

1. Füge neue Quelle und Senke s' und t' ein.
2. Ersetze jede Kante (a,b) durch drei Kanten (a,x), (x,y) und (y,b).
3. Erzeuge Kanten (s',y) und (x,t').
4. Setze Kantengewichte:

$$\begin{aligned} c''(a,x) = c''(y,b) &:= c(a,b) \\ c''(x,y) &:= c(a,b) - c'(a,b) \\ c''(s',y) = c''(x,t') &:= c'(a,b) \\ c''(t,t') = c''(s',s) &:= \sum_{e \in E} c(e) \end{aligned}$$

2.3 Flüsse mit Alternativen *(nicht relevant)*

Man könnte das Mindestflussproblem so erweitern, dass ein Fluss über eine Kante e nicht nur in $[c(e), c'(e)]$ liegen muss, sondern alternativ auch gleich null sein kann.

Bäm, schon ist das Problem **NP**-schwer. Dies kann man mit einer Reduktion von Exakt-3-SAT zeigen.

2.4 Min-Cost-Flow

Beim *Min-Cost-Flow-Problem* wird nach einem Fluss mit exaktem Betrag W gesucht, der Kosten minimiert. Die Kosten einer Kante sind dabei proportional zum Fluss, der durch sie fließt.

Wenn $W = 1$, dann haben wir das Problem des kürzesten Weges.

Definition: Min-Cost-Flow-Problem

Genau wie das normale Flussproblem, aber zusätzlich:

Eingabe: $l : E \to \mathbb{N}$ und $W \in \mathbb{N}$

Ziel: Bestimme Fluss f mit $w(f) = W$ und minimalem $l(f) = \sum_{e \in E} f(e) \cdot l(e)$.

Für den Algorithmus brauchen wir den Begriff des *Kreisflusses*:

Ein Fluss f' ist ein *Kreisfluss* gdw.
$\forall v \in V : f'_{in}(v) = f'_{out}(v)$.

Zum Algorithmus.

In jedem Kreisfluss f' gilt $w(f') = 0$. Wir können also durch Kreise Kosten verbessern, ohne den Flusswert zu ändern. Dies geht immer, bis f kostenminimal ist.

Wenn $l(e) = 0 \ \forall e \in E$, dann muss nur ein Fluss mit $w(f) = W$ gefunden werden.

Algorithmus: Min-Cost-Flow

Eingabe: $G = (V, E, s, t, c, l)$, $W \in \mathbb{N}$.
Ausgabe: Fluss f mit $w(f) = W$.

1. Bestimme beliebigen Fluss f mit $w(f) = W$.
2. Verbessere die Kosten:
   - Solange es in G_f einen Kreis C mit negativen Kosten gibt:
     $f = f + f'$

Beweis. Es gelte $l(f^*) < l(f)$ für einen besseren Fluss f^*. Sei dann $f'(e) := f^*(e) - f(e) \ \forall e \in E$.

f' ist dann zyklisch, weil f^* und f zur Quelle und Senke gleichen Fluss haben müssen.

f' besteht aus $m' < m$ Kreisflüssen f'_i. Also: $l(f') = l(f^*) - l(f) = \sum_{i=1}^{m'} l(f'_i)$. Wegen $l(f^*) < l(f)$ muss es ein j mit $l(f'_j) < 0$ geben. □

Satz: Kostenminimalität

Falls f nicht kostenminimal ist, dann gibt es einen Kreis C in G_f, auf dem f negative Kosten hat.

Wir können den Algorithmus schneller machen, indem wir "bessere" Kreise bevorzugt wählen. Mit dem Min-Mean-Cycle-Algorithmus kommen wir so insgesamt auf eine bessere Laufzeit. Genauer wollen wir Kreise so wählen, dass die durchschnittlichen Kantenkosten $\bar{l}(C)$ eines Kreises C minimiert werden:

$$\bar{l}(C) = \frac{\sum_{e \in C} l(e)}{|C|}$$

Der Algorithmus läuft in $\mathcal{O}(m^3 \cdot n^2 \cdot \log n) \cap \mathcal{O}(m^2 \cdot n^2 \cdot \log(nL))$, dabei wird $\mathcal{O}(m^2 \cdot n \cdot \log n) \cap \mathcal{O}(m \cdot n \cdot \log(nL))$ oft ein Min-Mean-Cycle gesucht. (Hier dein MR)

Algorithmus: Mean-Algorithmus

Eingabe: $G = (V, E, s, t, c, l)$, $W \in \mathbb{N}$.
Ausgabe: Fluss f mit $w(f) = W$.

1. Bestimme beliebigen Fluss f mit $w(f) = W$.
2. Verbessere die Kosten:
   - Solange es in G_f einen Kreis C mit negativen Kosten gibt:
     a) Setze $C = min - mean - cycle(V)$
     b) Bestimme maximalen zyklischen Fluss f' auf C.
     c) $f = f + f'$.

Für den Laufzeitbeweis wird die Ausführung in Phasen aufgeteilt. Jede Phase endet, wenn die Flusskosten um den Faktor $1 - 1/n$ gesunken sind. Die Anzahl der Phasen ist durch $n \ln(nL) + 1$, mit $L = \sum_{e \in E} |l(e)|$, beschränkt. Sei dazu $\mu(f) = -\bar{l}(C)$ und T die Anzahl der Phasen

1. Sei f_i der Fluss am Anfang von Phase i.
2. Phase i endet, wenn Fluss f_{i+1} mit $\mu(f_{i+1}) \leq (1 - 1/n) \cdot \mu(f_i)$ (nächste Phase) oder $\mu(f_{i+1}) \leq 0$ (Terminierung) gefunden ist.
3. Es gilt $\mu(f_i) \leq (1 - 1/n) \cdot \mu(f_{i-1}) \leq \frac{\mu(f_{i-1})}{e^{1/n}}$ (wegen $1 - x \leq e^{-x}$ für $x = 1/n$).
4. Wegen Ganzzahligkeit der Kosten gilt $\mu(f_i) \geq 1/n$ für die vorletzte Phase $T - 1$.

5. Wir erhalten $T - 1 \leq n \ln(nL)$.

In jeder Phase gibt es höchstens m Iterationen.

- Betrachte eine Iteration wo der Kreis C nur Kanten mit negativen Kosten hat.
  Dann wird mindestens eine Kante saturiert und entfernt. Alle neuen Kanten haben positive Kosten. Nach spätestens m Iterationen dieser Art terminiert der Algorithmus.
- Hat also eine Phase mehr als m Iterationen, muss spätestens Iteration $m - 1$ eine Kante mit positiven Kosten haben. Dies ist aber nicht möglich (Details dem Leser überlassen).

Damit erhalten wir insgesamt die oben genannte Laufzeit, wenn jede Iteration, d.h. Suche nach Min-Mean-Cycles, in $\mathcal{O}(mn)$ läuft.

2.5 Min-Mean-Cycle

Definition: Min-Mean-Cycle-Problem

Eingabe: $G = (V, E)$ gerichteter Graph, $l : E \to \mathbb{R}$ Kostenfunktion.

Ausgabe: Kreis $C \subseteq E$.

Ziel: Minimiere $\bar{l}(C)$.

Wir werden Werte $d_k(v)$ für die Kosten des günstigsten Kantenzuges eines beliebigen Knotens nach v mit Länge k verwenden, wobei $v \in V, k \in \mathbb{N}$.

Dann ist $d_0(v) = 0$ für alle $v \in V$ und $d_{n+1}(v) = \min_{(w,v)\in E} d_n(w) + l((w, v))$. Also können diese Werte in $\mathcal{O}(nm)$ mittels dynamischer Programmierung bestimmt werden.

Wir betrachten zunächst den Spezialfall, in dem der Min-Mean Kreis Wert 0 hat. Dann:

- Enthält G keine negativen Kreise, da diese auch negativen Durchschnittswert hätten.
- Ist für jeden Knoten v der Wert des günstigsten Kantenzugs, der bei v endet, $\min_{j=0}^{n-1} d_j(v) =: d(v)$.
- Ist $\max_{j=0}^{n-1} \left(\frac{d_n(v) - d_j(v)}{n-j} \right) \geq 0$ für alle $v \in V$. Dies gilt, da der Kantenzug mit n Kanten und Wert $d_n(v)$, der bei v endet, einen nicht-negativen Kreis enthalten muss. Also entsteht durch Streichen des Kreises ein nicht-teurerer Kantenzug mit weniger Kanten.
- Gibt es auf jedem Min-Mean Kreis einen Knoten v mit $\max_{j=0}^{n-1} \left(\frac{d_n(v) - d_j(v)}{n-j} \right) = 0$.
  Beweis. Sei dazu w ein beliebiger Knoten auf dem Min-Mean Kreis und P der günstigste Kantenzug, der bei w endet. Da es keine negativen Kreise gibt, ist dieser wohldefiniert und hat o.B.d.A. weniger als n Kanten.
  Diesen erweitern wir nun entlang des Kreises, bis wir n Kanten haben. Sei v der Knoten, an dem dieser Kantenzug endet.
  Dann muss dieser Kantenzug schon minimale Kosten von Kantenzügen zu v haben:
  Ein kürzerer Kantenzug könnte entlang des Kreises zu einem Kantenzug zu w erweitert werden. Da der Kreis Gewicht 0 hat, muss dieser Kantenzug auch günstiger als P sein, was ein Widerspruch ist. □

Wenn der Min-Mean Kreis Wert 0 hat, gilt also auch

$$\min_{v\in V} \max_{j=0}^{n-1} \left(\frac{d_n(v) - d_j(v)}{n-j} \right) = 0.$$

Wenn wir l durch $l' : e \mapsto l(e) + t$ ersetzen, ändern sich sowohl obiger Term, als auch der Wert jedes Kreises um t. Dann ist der Durchschnittswert des Min-Mean-Kreises stets:

$$\min_{v \in V} \max_{j=0}^{n-1} \left(\frac{d_n(v) - d_j(v)}{n - j} \right).$$

Wir können also das allgemeine Problem auf den Spezialfall, dass der Min-Mean Kreis Wert 0 hat, reduzieren. Jetzt nutzen wir eine Potentialfunktion, mit der alle Kanten des Min-Mean Kreises Wert 0 haben:

Wir ersetzen die Kostenfunktion l durch $l' : (v, w) \mapsto l((v, w)) + d(v) - d(w)$. $d(v)$ ist der Wert des günstigsten Kantenzuges zu v. Dabei ändert sich kein der Wert eines Kreises, da das Potential jedes Knotens je einmal aufaddiert und abgezogen wird. Da $d(w) \leq d(v) + l(e)$ ist, ist l' überall nicht-negativ. Da der Min-Mean Kreis Wert 0 hat, müssen alle Kanten auf dem Kreis bereits Wert 0 haben. Also können wir im Teilgraphen (V, E') mit $E' = \{ e \in E \mid l'(e) = 0 \}$ einen beliebigen Kreis suchen, um einen Min-Mean Kreis von (V, E) bzgl. l zu finden.

Algorithmus: Min-Mean-Cycle

Eingabe: $G = (V, E)$ gerichteter Graph, $l : E \to \mathbb{R}$ Kostenfunktion.
Ausgabe: Kreis C mit $\bar{l}(C)$ minimal.

1. Bestimme $d_j(v)$ für $v \in V, 0 \leq j \leq n$:
   - Setze $d_0(v) = 0$ für alle $v \in V$.
   - Für alle j von 1 bis n:
     - Setze $d_j(v) = \infty$.
     - Für alle $(w, v) \in E$:
       Falls es kleiner ist, ersetze $d_j(v)$ durch $d_{j-1}(w) + l((w, v))$.
2. Bestimme $\alpha := \min_{v \in V} \max_{j=0}^{n-1} \left(\frac{d_n(v) - d_j(v)}{n-j} \right)$.
3. Bestimme $p(v) = \min_{j=0}^{n-1} d_j(v) - j\alpha$ für alle $v \in V$.
4. Bestimme $E' = \{ (v, w) \in E \mid l((v, w)) - \alpha + p(v) - p(w) = 0 \}$.
5. Bestimme Kreis C in (V, E') mit Tiefensuche.
6. Gib C zurück.

3 Matchings

Ein *Matching* ist eine Kantenmenge, in der jeder Knoten höchstens eine Kante besitzt.

Oder anders gesagt: eine Auswahl von Knotenpaaren, sodass kein Knoten mehr als einmal vergeben wird.

Ein *maximales Matching* ist ein Matching, zu dem keine weitere Kante hinzugefügt werden kann ohne die Matching-Eigenschaft zu verletzen.

Ein *Maximum Matching* hat die größtmögliche Mächtigkeit, die ein Matching haben kann.

Der folgende Algorithmus zeigt, wie man durch beliebiges Auswählen von Kanten ein maximales Matching greedy berechnen kann.

Definition: Matching

In einem ungerichteten Graphen $G = (V, E)$ heißt $M \subseteq E$ Matching falls:
$\forall v \in V : \delta_{G'}(v) \leq 1, G' = (V, M)$

Definition: Maximales Matching

Analog zum Matching, nur dass:
$\forall M' : M \subsetneq M' \subseteq E : M'$ kein Matching

Definition: Maximum Matching

Analog zum Matching, nur dass:
$\forall M' \subseteq E : |M| < |M'| : M'$ kein Matching

Algorithmus: Greedy-Algorithmus für maximales Matching

Eingabe: $G = (V, E)$.
Ausgabe: Maximales Matching M.

1. Setze $M = \emptyset$.
2. Wiederhole, bis E leer ist:
   a) Füge beliebige Kante $e \in E$ zu M hinzu.
   b) Entferne alle benachbarten Kanten aus E.

(Hier dein MR)

3.1 Bipartites Matching

Bei dem *Bipartiten Matchingproblem* wird in einem bipartiten Graph ein (kostenminimales) Maximum Matching gesucht.

Das bipartite Matchingproblem kann auf das Flussproblem reduziert werden.

Dazu betrachtet man die Kanten zwischen V und W als Flusskanten. Außerdem wird eine neue Quelle mit allen Knoten aus V verbunden, analog eine Senke mit W. Allen Kanten wird Kapazität 1 zugeordnet.

Gibt es Kosten, so werden diese übernommen. Für neue Kanten setze Kosten auf 0.

Löst man das Flussproblem mit Dinitz, so erhalten wir die Laufzeit $\mathcal{O}(n^3)$.

Definition: Bipartites (kostenminimales) Matchingproblem

Eingabe: Bipartiter Graph $G = (V, W, E)$ mit $E \subseteq V \times W$ und Kostenfunktion $g : E \to \mathbb{N}$.

Ausgabe: Maximum Matching M.

Ziel: M soll minimale Kosten haben.

Algorithmus: Bipartites Matching über Dinitz

1. Transformation links auf Eingabe anwenden.
2. Algorithmus von Dinitz.

3.1.1 Alternierende Pfade

Wenn ein bipartites Matching vergrößert wird, werden Kanten entlang einem Pfad bzgl. dem Matching getauscht. Matching-Kanten werden Nichtmatching-Kanten und umgekehrt. Weil Matching-Kanten keine gemeinsamen Knoten haben, ist der Pfad *alternierend*: Jede zweite Kante auf dem Pfad wird Matching-Kante und jede Zweite wird Nichtmatching-Kante.

Ein solcher Pfad ist *erweiternd*, falls die beiden Endknoten frei sind, d.h. an keiner Matching-Kante liegen.

Der Algorithmus Alternierende Pfade 1 hat eine Laufzeit von $\mathcal{O}(n \cdot m)$.

Definition: Alternierender Pfad

Ein Pfad $\{ v_0, v_1 \}, \ldots, \{ v_{l-1}, v_l \}$ heißt *alternierend*, falls für alle $i \in (0, l)$ gilt:

$$\{ v_{i-1}, v_i \} \in M \iff \{ v_i, v_{i+1} \} \notin M$$

Algorithmus: Alternierende Pfade 1

Eingabe: $G = (V, W, E)$.
Ausgabe: Maximum Matching M.

1. Setze $M = \emptyset$.
2. Solange es erweiternden Pfad P gibt:
   - $M = M \oplus E(P)$

Zum Laufzeitbeweis:

- Ist P ein verbessernder Pfad, dann gilt $|M \oplus E(P)| = |M| + 1$.
- Sind M, N Matchings mit $|M| < |N|$, dann hat $H = (V, M \oplus N)$ mindestens $|N| - |M|$ knotendisjunkte verbessernde Pfade bzgl. M.
- M ist genau dann ein Maximum Matching, wenn es keinen verbessernden Pfad gibt.
- Wegen $|M| \leq \lfloor n/2 \rfloor$ gibt es höchstens $\lfloor n/2 \rfloor$ Schleifendurchläufe, jede Schleife läuft in $\mathcal{O}(m)$.

Eine Verbesserung der Laufzeit ist möglich, wenn direkt alle kürzesten verbessernden Pfade gesucht werden.

Mit nicht-Maximum Matching M und Maximum Matching M' gibt es in M einen verbessernden Pfad der Länge maximal $2 \cdot \left\lfloor \frac{|M|}{|M'|-|M|} \right\rfloor + 1$.

Teilt man die Iterationen in zwei Phasen auf, die Erste mit $|M| \leq \lfloor |M'| - \sqrt{|M'|} \rfloor$, ergeben sich maximal $\sqrt{|M'|}$ Iterationen pro Phase.

Insgesamt erhält man die Laufzeit $\mathcal{O}(\sqrt{n} \cdot m)$.

Algorithmus: Alternierende Pfade 2

Eingabe: $G = (V, W, E)$.
Ausgabe: Maximum Matching M.

1. Setze $M = \emptyset$.
2. Solange es verbessernde Pfade gibt:
   a) Finde inklusions-maximale Menge von knotendisjunkten verbessernden Pfaden $P_1, \ldots, P_i$ der kürzesten Länge l.
   b) $M = M \oplus E(P_1) \oplus \ldots \oplus E(P_i)$.

Die Bestimmung der inklusions-maximalen Menge von kürzesten verbessernden Pfaden ist in $\mathcal{O}(m)$ möglich.

Dazu wird einfach die Suche nach verbessernden Pfaden als Suche zwischen freien Knoten von V zu freien Knoten in W umformuliert. Dann lassen sich mit Breitensuche und Tiefensuche alle kürzesten verbessernden Pfade bestimmen.

Algorithmus: Inkl.-max. verbessernde Pfade

Eingabe: $G = (V, W, E)$.
Ausgabe: Pfade $P_1, \ldots, P_l$.

1. $G' = (V \cup W, E' \cup E'')$ mit
   - $E' = \{ (v,w) \in V \times W \mid \{ v,w \} \in E \setminus M \}$
   - $E'' = \{ (w,v) \in W \times V \mid \{ v,w \} \in M \}$
2. Sei in G'' zusätzlich Quelle s und Senke t, verbunden mit freien Knoten aus V respektive W.
3. Suche alle Kanten K, die auf einem kürzesten Pfad liegen (Breitensuche).
4. Finde alle P_i durch Tiefensuche auf Kanten K.

3.1.2 Mit Kosten

Wir betrachten nun das Maximum Matching-Problem mit Kosten.

Dazu passen wir die Idee aus dem vorigen Abschnitt an und suchen gewichtsmaximale verbessernde Pfade.

Die Laufzeit des ersten Algorithmus' ist $\mathcal{O}(n^2 \cdot m)$. Dazu ist wichtig, dass im Schritt i das bestimmte Matching das gewichtsmaximale Matching mit Kardinalität i ist und der Algorithmus maximal $\lfloor n/2 \rfloor$-mal einen verbessernden kostenmaximalen Weg sucht. Weil es keine negativen Kreise gibt, kann der Bellman-Ford-Algorithmus genutzt werden, welcher Laufzeit $\mathcal{O}(n \cdot m)$ hat.

Algorithmus: Gewichtsmaximales Matching 1

Eingabe: $G = (V, W, E)$, $g : E \to \mathbb{N}$.
Ausgabe: Gewichtsmaximales Matching M_{opt}.

1. Setze $M = M_{opt} = \emptyset$.
2. Solange es verbessernden Pfad bzgl. M gibt:
   a) $G' = (V \cup W, E' \cup E'')$ mit
      - $E' = \{ (v,w) \in V \times W \mid \{ v,w \} \in E \setminus M \}$
      - $E'' = \{ (w,v) \in W \times V \mid \{ v,w \} \in M \}$
   b) Bestimme verbessernden Pfad P mit maximalem Gewicht bzgl. M und g_M.
   $$g_M(e) := \begin{cases} g(e), & \text{falls } e \in E', \\ -g(e), & \text{falls } e \in E'' \end{cases}$$
   c) Setze $M = M \oplus E(P)$.
   d) Falls $g(M) > g(M_{opt})$, setze $M_{opt} = M$.

Bei der *Ungarischen Methode* wird der Algorithmus von Dijkstra ($\mathcal{O}(m + n \log n)$) benutzt, um eine noch bessere Laufzeit von $\mathcal{O}(n \cdot (m + n \log n))$ zu erreichen. Die Idee soll hier nicht weiter erläutert werden, aber wichtig ist, dass wegen negativer Kantengewichte der Graph zunächst so umgewandelt werden muss, dass er nur positive Kantengewichte hat. Dies ist wieder einmal mit einer Potentialfunktion zu tun.

3.2 Allgemeine Graphen

Die vorherigen Algorithmen sind nur auf bipartiten Graphen korrekt, aber lediglich ungerade Kreise stören am Ende.

Dazu ist die Idee, *Blüten* zu suchen und diese durch einen einzigen Knoten zu ersetzen. Dann kann auf dem neuen Graphen die Suche nach verbessernden Pfaden fortgesetzt werden. Eine Blüte ist ein ungerader Kreis mit jeder zweiten Kante im Matching.

Dies wird beim *Algorithmus von Edmonds* verwendet. Insgesamt ist das Matchingproblem auf allgemeinen Graphen in Zeit $\mathcal{O}(m \cdot \sqrt{n})$ lösbar. Das gewichtete Matchingproblem ist lösbar in $\mathcal{O}(n^3)$.

3.3 Minimum-Weight Perfect Matching

(Hier dein MR)

4 Approximation

Wir betrachten NP-schwere Probleme und stellen Algorithmen vor, die bis auf einen bestimmten Fehler das Problem in Polynomialzeit lösen.

Sei A ein Algorithmus. Die Bedingungen müssen für alle Eingabeinstanzen I gelten.

	Additiver Approximationsfehler	*Multiplikativer Approximationsfehler*
Maximierungsproblem	$\text{opt}(I) \leq A(I) + k$	$\frac{A(I)}{\text{opt}(I)} \geq \alpha$ und $\alpha \leq 1$
Minimierungsproblem	$\text{opt}(I) \geq A(I) + k$	$\frac{A(I)}{\text{opt}(I)} \leq \alpha$ und $\alpha \geq 1$
Zusammengefasst		$\max\left\{ \frac{A(I)}{\text{opt}(I)}, \frac{\text{opt}(I)}{A(I)} \right\} \leq \alpha$

4.1 Clique

Falls P $\neq$ NP gilt, gibt es für beliebiges $k \in \mathbb{N}$ keinen Polynomialzeitalgorithmus mit Approximationsfehler k für das Cliquenproblem.

Das selbe folgt für Independent Set (äquivalent zu Clique im Komplementgraphen).

Der Beweis geht über das Graphenprodukt $G^k = G \times C_k$ von zyklischen Graphen, wobei C_k der Cliquengraph in der Iteration k ist. Wenn es einen Algorithmus mit additivem Fehler k gäbe, dann erhält man über einige Schritte mit Eingabe G^{k+1}, dass Clique in P liegen muss. (Hier dein MR)

Definition: Cliquenproblem

Eingabe: $G = (V, E)$.

Ausgabe: $V' \subset V$, wobei $\{ v, w \} \in E$ für jedes Knotenpaar $v, w \in V'$.

Ziel: Maximiere $|V'|$.

4.2 Vertex Cover

Beim Vertex Cover soll jede Kante durch einen Knoten abgedeckt sein.

Ein einfacher Greedy-Algorithmus, der Knoten mit höchstem Grad wählt, kann keinen konstanten Approximationsfaktor haben.

Dies lässt sich mit einem tripartiten Graphen $G = (V, W, X, E)$ zeigen, in dem es zwischen V und W für zwei Knoten genau eine Kante gibt und jedes $w_i \in W$ mit allen $x_j \in X$ mit $j \geq i$ verbunden ist.

Eine 2-Approximation ist durch Bestimmung des inklusions-maximalen Matchings möglich.

Definition: Vertex Cover-Problem

Eingabe: $G = (V, E)$.

Ausgabe: $C \subset V$ mit $\forall e \in E : C \cap e \neq \emptyset$.

Ziel: Minimiere $|C|$.

Algorithmus: Vertex Cover 2-Approximation

Eingabe: $G = (V, E)$.
Ausgabe: Vertex Cover C.

1. Bestimme inklusions-maximales Matching $M \subset E$ auf G.
2. Wähle Vertex Cover $\bigcup_{\{u,v\} \in M} \{u, v\}$.

4.3 TSP und Delta-TSP

Beim *Travelling Salesman Problem* soll in einem vollständigen Graph ein Hamiltonkreis gefunden werden, also einer, der über alle Knoten führt. TSP ist nicht approximierbar.

Definition: Travelling Salesman Problem

Eingabe:

- $G = (V, E)$ ungerichtet, vollständig
- Kostenfunktion: $c : E \to \mathbb{Q}$

Ausgabe: Hamiltonkreis $C \subset E$

Ziel: Minimiere $c(C)$.

Definition: Δ-TSP

Wie TSP, nur dass die Kostenfunktion zusätzlich eine Metrik ist:

- $c(e) \geq 0 \ \forall e \in E$,
- $c(\{v, z\}) \leq c(\{v, w\}) + c(\{w, z\})$ $\forall v, w, z \in V$.

Δ-TSP ist mit einem Faktor von 2 in $\mathcal{O}(n^2 \cdot \log n)$ approximierbar.

Dazu wird ein minimaler Spannbaum bestimmt. Die Kanten werden verdoppelt, um gerade Knotengrade zu haben.

Dann kann einfach ein Euler-Kreis gefunden werden und doppelt auftauchende Knoten ausgelassen werden (möglich wegen vollständigem Graph).

Dieses Überspringen verkürzt die Pfadlänge, weil die Distanzen der Dreiecksungleichung unterliegen.

Algorithmus: Δ-TSP 2-Approximation

Eingabe: $G = (V, E)$, $c : E \to \mathbb{Q}$.
Ausgabe: TSP-Pfad $C \subseteq E$.

1. Finde minimalen Spannbaum T von G.
2. Verdopple Kanten von T in T'.
3. Bestimme Euler-Kreis C' in T'.
4. Verkürze C' durch Überspringen doppelter Knoten.

Die Verschlechterung gegenüber der optimalen Lösung wird dabei in Schritt 2 erzeugt: Da die optimale TSP-Tour nach Weglassen einer Kante einen Spannbaum bildet, sind die Kosten des

minimalen Spannbaums eine untere Schranke der Kosten der optimalen TSP-Tour. Der verdoppelte Baum, und damit der Euler-Kreis, ist also nicht mehr als doppelt so teuer wie die optimale Lösung. Schritt 4 vergrößert die Kosten nicht.

Da eben die Kantenverdopplung die Kosten erhöht, kann auch hier angesetzt werden, um den Approximationsfaktor zu verbessern. Das Verdoppeln sorgt dafür, dass der Grad aller Knoten gerade ist. Es reicht aber aus, den Grad aller Knoten mit ungeradem Grad um 1 zu erhöhen. Dies lösen wir über ein Minimum-Weight-Perfect-Matching auf den Knoten, die im minimalen Spannbaum ungeraden Grad haben.

Algorithmus: Algorithmus von Christofides

Eingabe: Vollständiger Graph $G = (V, E)$, $c : E \to \mathbb{Q}$.
Ausgabe: TSP-Tour $C \subseteq E$.

1. Bestimme minimalen Spannbaum T von G.
2. Bestimme von Knoten mit ungeradem Grad in (V, T) induzierten Teilgraph von G.
3. Bestimme Minimum-Weight-Perfect-Matching M in diesem Teilgraph.
4. Bestimme Euler-Kreis C' von $(V, T \cup M)$.
5. Verkürze C' durch Entfernen doppelter Knoten.

Um die 1.5-Approximation zu erreichen, darf das Matching in Schritt 3 nicht größer als die Hälfte der optimalen Lösung sein. Betrachten wir dazu das Minimum-Weight-Perfect-Matching M auf der Knotenmenge V', sowie die optimale TSP-Tour C von G. Sei C' die Tour auf V' eingeschränkt. Dann ist das Gesamtgewicht von C' nicht höher als das von C.

Da sich aber C' in zwei perfekte Matchings auf V' aufteilen lässt, indem man jede zweite Kante in ein Matching aufnimmt, und beide dieser Matchings nicht kleiner als das minimale Matching sein können, ist die optimale Lösung mindestens doppelt so teuer wie M.

Insgesamt liefert der Algorithmus eine 1.5-Approximation.

4.4 Steiner-Bäume

Bei dem *Steinerbaum Problem* wird ein Baum gesucht, der alle Terminale $T \subseteq V$ abdeckt. Dabei sollen die Kosten der gewählten Kanten im Baum minimiert werden.

Die zusätzlich zu den Terminalen gewählten Knoten P heißen *Steinerpunkte*.

Falls $T = V$, wird nur ein minimaler Spannbaum gesucht. Falls $|T| = 2$, wird nur ein minimaler Weg gesucht.

Definition: Steinerbaum Problem

Eingabe: $G = (V, E)$, $T \subseteq V$, $c : E \to \mathbb{Q}$.

Ausgabe: Spannbaum $S = (T \cup P, F)$, der Knoten T abdeckt.

Ziel: Minimiere $\sum_{e \in F} c(e)$.

Algorithmus: Kou, Markowsky, Berman (2-Approximation)

Eingabe: $G = (V, E)$, $T \subset V$, $c : E \to \mathbb{Q}$.
Ausgabe: Steinerbaum S_{KMB}.

1. Setze $G_D = (T, F)$ als den Distanzgraphen für Terminale T mit
   - $c : F \to \mathbb{Q}$, $\{ a, b \} \mapsto \text{dist}_G(a, b)$
   - $F = V \times V$
2. Bestimme minimalen Spannbaum $S_D = (T, F')$ auf G_D.
3. Rekonstruiere S_D in G als H,
   d.h. ersetze jede Kante $\{ v, w \} \in F'$ durch einen Weg der Länge $c(\{ v, w \})$ von v zu w.
4. Bestimme minimalen Spannbaum S_H auf H.
5. S_{KMB} ist S_H, aus dem sukzessive Blätter gelöscht wurden, die nicht in T sind.

Das *Verfahren von Kou, Markowsky und Berman* bestimmt eine 2-Approximation für das Steinerbaum Problem. Die Idee: Bestimme zunächst minimalen Spannbaum als wären alle Terminale direkt verbunden (über Distanzgraphen). Dann wird dieser Spannbaum wieder in dem ursprünglichen Graphen rekonstruiert, so dass die Distanzen zwischen Terminalen Wegen dieser Länge entsprechen. Darauf wird wieder ein minimaler Spannbaum bestimmt, da unter Umständen Kreise existieren können. Zuletzt werden daraus Pfade entfernt, die nicht zu Terminalen führen.

Für die Laufzeit sollte man sich klarmachen, dass der minimale Spannbaum mit Breiten- oder Tiefensuche in $\mathcal{O}(n + m)$ gefunden werden kann. Weiterhin ist die Rekonstruktion in Schritt 3 auch effizient, wenn man einfach zwischen allen Knotenpaaren die kürzesten Wege bestimmt. Dies geht, mit etwa Floyd-Warshall, in $\mathcal{O}(n^3)$.

Für den Approximationsfaktor 2:

- Wir lassen den optimalen Steinerbaum S^* vom Himmel fallen.
- Wir bestimmen eine Tour über alle Knoten, die jede Kante zweimal benutzt (vgl. TSP 2-Approximation).
  Das Gesamtgewicht dieser Tour ist also doppelt so groß wie das von S^*.
- Mit dieser Tour definieren wir eine Tour in dem Distanzgraphen, die die Terminale in der selben Reihenfolge besucht.
  Diese ist nicht länger, da wir im Prinzip jeden Pfad zwischen zwei Terminalen, der in der Tour im Steinerbaum liegt, durch den kürzesten Pfad ersetzen.
- Durch Weglassen doppelt besuchter Knoten erhalten wir einen Spannbaum des Distanzgraphens.
  Dieser ist nicht teurer als die Tour, da der Distanzgraph metrisch ist.
- Also ist dieser Spannbaum nicht mehr als doppelt so teuer wie S^*.
  Gleichzeitig ist der im Algorithmus in Schritt 2 berechnete Spannbaum des Distanzgraphen minimal, und der berechnete Steinerbaum nicht teurer als der Spannbaum.

4.5 Zentrumsproblem

Bei dem *Zentrumsproblem* wird eine Knotenmenge Z (Zentren) der Größe k gesucht, sodass der längste Weg aller Knoten zu einem Zentrum minimal ist.

Definition: Zentrumsproblem mit Knotenkosten

Eingabe: $G = (V, E)$, $k \in \mathbb{N}$, $c : E \to \mathbb{Q}^+$, $w : V \to \mathbb{Q}^+$.

Ausgabe: $Z \subset V$ mit $|Z| = k$.

Ziel: Minimiere $\text{rad}(Z)$.

$$\text{rad}(Z) := \max_{v \in V} w(v) \cdot \text{dist}(v, Z)$$

Das ursprüngliche Zentrumsproblem *ohne Knotenkosten* kann erreicht werden, indem man die Knotenkosten auf 1 setzt.

Falls P $\neq$ NP gilt, gibt es keinen polynomiellen $2 - \varepsilon$ Approximationsalgorithmus für das Zentrumsproblem. Eine Reduktion auf das Dominating Set Problem zeigt dies.

Algorithmus: GreedyZentrum

Eingabe: $G = (V, E)$, $c : E \to \mathbb{Q}^+$, $w : V \to \mathbb{Q}^+$, $R \in \mathbb{N}$.

Ausgabe: $Z \subset V$.

1. Setze $Z = \emptyset$ und $U = V$.
2. Solange $U \neq \emptyset$:
   a) $z = \text{argmax}_{u \in U} w(u)$.
   b) $Z = Z \cup \{z\}$.
   c) $U = U \setminus \{v \in U | w(v) \cdot dist(v, z) \leq 2 \cdot R\}$.

Algorithmus: 2-Approximation

Eingabe: $G = (V, E)$, $k \in \mathbb{N}$, $c : E \to \mathbb{Q}^+$, $w : V \to \mathbb{Q}^+$.

Ausgabe: $Z \subset V$.

1. Sortiere Radien
   $R_i \in \{ w(u) \cdot \text{dist}(u, v) \mid u, v \in V \wedge u \neq v \}$.
2. Durchlaufe R_i aufsteigend:
   a) $Z = \textit{GreedyZentrum}(V, c, w, R_i)$
   b) Wenn $|Z| \leq k$, gib Z aus.

Der Approximationsalgorithmus benutzt einen Hilfsalgorithmus *GreedyZentrum*, der den optimalen Radius R^* benötigt. GreedyZentrum wählt greedy den konstengünstigsten Knoten aus, bis alle Knoten überdeckt sind, und benutzt als Radius $2R$.

GreedyZentrum gibt für ein $R \geq R^*$ ein $|Z| \leq |Z^*|$ zurück. Dies kann gezeigt werden, indem man überlegt, dass der Überdeckungsbereich, mit Radius $2R$, eines approximierten Zentrums den Überdeckungsbereich eines optimalen Zentrums, mit Radius R, vollständig beinhaltet. Somit wählt GreedyZentrum nicht mehr Knoten als der optimale Algorithmus aus.

GreedyZentrum hat eine Laufzeit von $\mathcal{O}(n^2)$.

Der Approximationsalgorithmus ruft nun GreedyZentrum aufsteigend für alle möglichen Werte für R auf, bis GreedyZentrum ein Z mit $|Z| \leq k$ ausgibt. Dies passiert $\mathcal{O}(n^2)$-mal.

Insgesamt hat die 2-Approximation eine Laufzeit von $\mathcal{O}(n^4)$.

4.6 Set Cover

Das *Set Cover-Problem* kann auf das Vertex-Cover-Problem reduziert werden. Deshalb ist Set Cover in NPC. Set Cover kann man sogar als Verallgemeinerung von Vertex-Cover auf Hypergraphen verstehen; das sind Graphen, in denen eine Kante zu beliebig vielen Knoten inzident sein kann. Zur Approximation gibt es einen einfachen Greedy-Algorithmus: Dieser wählt greedy die Menge, die das beste (minimale) Verhältnis von Kosten zu neu überdeckten Elementen hat.

Dadurch kann ein Approximationsfaktor von H_n (Harmonische Zahl) erreicht werden.

$$H_n = \sum_{i=1}^{n} \frac{1}{i}$$

Der Beweis des Approximationsfaktors ist lang.

(Hier dein MR)

Definition: Set Cover-Problem

Eingabe:

- Grundmenge X mit $n = |X|$.
- m Teilmengen $S_1, \ldots, S_m$ mit $\bigcup_{i \in \{1,\ldots,m\}} S_i = X$.
- Kosten $c_i \in \mathbb{Q}$ für jede Menge.

Ausgabe:

- $A \subset \{1, \ldots, m\}$.
- $\bigcup_{i \in A} S_i = X$.

Ziel: Minimiere $\text{cost}(A) = \sum_{i \in A} c_i$.

Algorithmus: H_n-Approximation

Eingabe: $X, S_1, \ldots, S_m, c_1, \ldots, c_m \in \mathbb{Q}$.

Ausgabe: Set Cover A.

1. Setze $A = \emptyset$.
2. Solange $\bigcup_{j \in A} S_j \neq X$, wiederhole:
   a) Bestimme $i = \operatorname{argmin}_{i \in \{1,\ldots,m\} \setminus A} \frac{c_i}{|S_i \setminus \bigcup_{j \in A} S_j|}$.
   b) Setze $A = A \cup \{i\}$.

5 Scheduling

Beim *Scheduling* geht es darum, Jobs einer bestimmten Laufzeit möglichst gut auf Maschinen zu verteilen. Dabei soll der *Makespan*, die längste Arbeitszeit einer Maschine, minimiert werden.

5.1 Identische Maschinen

Zunächst gehen wir von identischen Maschinen aus, d.h. alle Maschinen sind gleich schnell. Die Joblaufzeiten p_i sind ganzzahlig und können nicht aufgeteilt werden.

Die Verteilung ist durch den *Schedule* gegeben, der einen Job i der Maschine $f(i)$ zuordnet. Man beachte, dass die Reihenfolge, in der die Maschinen einen ihr zugeordneten Job ausführen, keine Rolle spielt.

Subset-Sum lässt sich auf dieses Problem reduzieren.

Wir haben zwei sehr einfache Approximationsalgorithmen.

Die *LL-Heuristik* verteilt die Jobs auf die zum aktuellen Zeitpunkt am wenigsten ausgelastete Maschine. Sie erreicht so einen Approximationsfaktor von $2 - \frac{1}{m}$.

Für den Faktor betrachtet man die Maschine j, auf die der Job $p_{i'}$ der als letztes fertig wird, gelegt wurde. Diese kann höchstens eine Last von $\frac{1}{m}\sum_{i<i'} p_i$ (weil minimal ausgelastet) vor der Zuteilung gehabt haben.

$$\begin{aligned}(\frac{1}{m}\sum_{i<i'} p_i) + p_{i'} &= (\frac{1}{m}\sum_{i\leq i'} p_i) + (1-\frac{1}{m})\cdot p_{i'} \\ &\leq (2-\frac{1}{m})\cdot \text{opt}\end{aligned}$$

Definition: Makespan-Scheduling Problem auf identischen Maschinen

Eingabe:

- $p_1, \ldots, p_n \in \mathbb{N}$ Jobs.
- $m \in \mathbb{N}$ Maschinen.

Ausgabe:

- Schedule $f : \{1, \ldots, n\} \to \{1, \ldots, m\}$

Ziel: Minimiere $\max_{j\in\underline{m}} \sum_{\substack{i\in\underline{n} \\ f(i)=j}} p_i$.

Algorithmus: Least Loaded (LL)

Eingabe: $p_1, \ldots, p_n \in \mathbb{N}$, $m \in \mathbb{N}$.

Ausgabe: $f : \{1, \ldots, n\} \to \{1, \ldots, m\}$.

Für jeden Job k:

1. Bestimme $j = \min \sum_{\substack{i\in\underline{k-1} \\ f(i)=j}} p_i$.
2. Setze $f(k) = j$.

Algorithmus: Longest Processing Time (LPT)

Wie LL, nur sortiere Jobs absteigend:

$$p_1 \geq p_2 \geq \ldots \geq p_n$$

Die *LPT-Heuristik* erweitert die LL-Heuristik und sortiert die Jobs zusätzlich absteigend nach Dauer. Sie hat einen Approximationsfaktor von $\frac{4}{3}$.

Um den Approximationsfaktor zu zeigen, nimmt man an, dass LPT auf einer Eingabe mit minimaler Länge (!) $p_1, \ldots, p_n$ einen Schedule mit $\tau > \frac{4}{3}$ opt erstellt. Dann ist der Job, der zuletzt fertig wird, auch der kleinste Job. Die zugehörige Maschine hat zum Zeitpunkt der Zuweisung höchstens Last $\frac{1}{m}\sum_{i=1}^{n-1} p_i \leq$ opt. Also muss gelten, dass $p_n > \frac{1}{3}$opt und damit auch $p_i > \frac{1}{3}$ opt für jeden Job i (wegen der Sortierung).

Daraus folgt, dass im optimalen Schedule jede Maschine maximal zwei Jobs haben kann. Dann ist der optimale Schedule aber genau der, der von LPT berechnet wird. Also war die Annahme falsch; LPT hat einen Approximationsfaktor $\leq \frac{4}{3}$.

5.2 Approximationsschema

Für das Zentrumsproblem (4.5) haben wir gesehen, dass, falls P $\neq$ NP gilt, es keinen effizienten Algorithmus mit Approximationsfaktor kleiner 2 geben kann. Im Folgenden werden wir zeigen, dass es für Makespan-Scheduling keine solche untere Schranke gibt. Dazu geben wir für jedes $\varepsilon > 0$ einen Polynomialzeitalgorithmus an, der eine $1+\varepsilon$-Approximation liefert.

Definition: Polynomialzeitapproximationsschema (PTAS)

Ein Algorithmus, der, für ein konstantes aber beliebiges $\varepsilon > 0$, in polynomieller Zeit eine Approximation mit Approximationsfaktor $1+\varepsilon$ (bzw. $1-\varepsilon$ bei Maximierungsproblemen) liefert, heißt *PTAS*.

Ein PTAS heißt *Voll-Polynomialzeitapproximationsschema* (FPTAS), wenn die Laufzeit auch polynomiell in $\frac{1}{\varepsilon}$ ist.

Für das PTAS werden wir zunächst einen Algorithmus ansehen, der zusätzlich zur Eingabe einen vermuteten optimalen Makespan Z^* erhält. Der Algorithmus wird dann entweder eine $1+\varepsilon$-Approximation an Z^*-berechnen oder feststellen, dass Z^* zu klein ist. Danach werden wir mit binärer Suche die Vermutung verbessern, bis wir eine $1+\varepsilon$-Approximation haben.

Algorithmus: Hilfsalgorithmus

Eingabe: n Jobs p_i, m Anzahl Maschinen, Makespan Z^*, ε erlaubter Fehler.
Ausgabe: Zuweisung Jobs auf Maschinen oder `Makespan zu klein`.

1. Bestimme große Jobs $J' = \{\, i \leq n \mid p_i > \varepsilon Z^* \,\}$.
2. Bestimme normierte gerundete Laufzeiten $p_i^* = \left\lceil \frac{p_i}{\varepsilon^2 Z^*} \right\rceil$.
3. Bestimme mit Bin Packing Algorithmus Schedule für J' mit $Z' = \left\lfloor (1+\varepsilon)\frac{1}{\varepsilon^2} \right\rfloor$, oder gebe `Makespan zu klein` zurück, wenn dies nicht möglich ist.
4. Verteile mit LL die kleinen Jobs.

Algorithmus: PTAS für Makespan-Scheduling

Eingabe: n Jobs p_i, m Anzahl Maschinen, ε erlaubter Fehler.
Ausgabe: Zuweisung Jobs auf Maschinen.

1. Bestimme $S = \sum_{i=1}^{n} p_i$
2. Führe binäre Suche nach $Z^* \in \{\, 1 \ldots S \,\}$ durch:
   - Führe Hilfsalgorithmus aus.
   - Falls `Makespan zu klein` bzw. `groß`, erhöhe untere bzw. verringere obere Schranke.
   - Wiederholen, wenn Intervall nicht degeneriert ist.
3. Gebe zur letzten oberen Schranke berechnete Zuweisung zurück.

Hilfsalgorithmus Um die Approximation an einen gegebenen Makespan zu finden, unterteilen wir zunächst die Jobs in *große* und *kleine* Jobs: Ein Job ist *groß*, wenn er mindestens Laufzeit εZ^* hat. Wir vereinfachen nun die Verteilung der großen Jobs, indem wir statt der exakten Laufzeit Slots der Größe $\varepsilon^2 Z^*$ Verteilen. Dann gibt es nur $\frac{1}{\varepsilon^2}$ verschiedene Jobgrößen zu betrachten.

Wie wir diese Jobs verteilen, werden wir in Teil 5.3 betrachten. Aus dem dort beschriebenen Algorithmus erhalten wir entweder eine Verteilung auf die m Maschinen mit Makespan $(1+\varepsilon)Z^*$ oder erkennen, dass es keine solche Zuweisung gibt.

Anschließend müssen wir noch die kleinen Jobs verteilen. Dafür verwenden wir einfach die LL-Heuristik (Algorithmus 5.1). Dabei können wir nicht mehr als εZ^* additiven Fehler über den optimalen Makespan machen, da der Fehler bei LL nur aus einem Job kommen kann. Also erhalten wir den gewünschten Approximationsfaktor, solange Z^* nicht größer als der optimale Makespan ist.

Die Laufzeit wird hier durch die Verteilung der großen Jobs dominiert. In Abschnitt 5.3 werden wir sehen, dass dies Laufzeit $\mathcal{O}(n^{\lceil \frac{1}{\varepsilon^2} \rceil})$ benötigt.

Binärsuche Nun nutzen wir diesen Algorithmus und binäre Suche, um eine $1+\varepsilon$-Approximation an den tatsächlichen optimalen Makespan zu finden. Zu Beginn ist die Summe der Laufzeiten aller Jobs S eine obere Schranke. Die möglichen Makespans sind also $\{\, 1 \dots S \,\}$. Damit findet Binärsuche in $\log(S)$ Schritten den optimalen Makespan (oder einen Wert der kleiner ist, aber nicht so viel kleiner, dass die Zuweisung der großen Jobs fehlschlägt).

Da der Wert S exponentiell in der Eingabelänge N ist, haben wir also $\mathcal{O}(N)$ Aufrufe des Algorithmus' und damit einer Gesamtlaufzeit $\mathcal{O}(Nn^{\lceil \frac{1}{\varepsilon^2} \rceil})$.

5.3 Bin Packing

Wir müssen für den Hilfsalgorithmus des PTAS noch die *großen* Jobs verteilen. Dies modellieren wir als Bin Packing Problem mit eingeschränkten Gewichten.

Definition: Bin Packing mit eingeschränkten Gewichten

Eingabe:

- n Objekte.
- Gewichte $w_i \in \{\, 1, 2, \dots, k \,\}$, für alle $i \in \{\, 1, \dots, n \,\}$.
- $m, b \in \mathbb{N}$ mit $m \geq 1$ Bins und Kapazität $b \geq k$.

Ausgabe:

- $z : \{\, 1, 2, \dots, n \,\} \to \{\, 1, 2, \dots, m \,\}$, ein Funktion die Objekte auf Bins zuordnet.

Ziel:

- $\forall i \in \{\, 1, 2, \dots, m \,\} : \sum_{\substack{j=1 \\ z(j)=i}}^{n} w_j \leq b$

  Jeder Bin enthält Objekte mit Gesamtgewicht höchstens b.

Dieses Problem lösen wir mit Dynamischer Programmierung: Wir bestimmen die k-dimensionale Tabelle, die uns zu jeder möglichen Teilmenge der Jobs angibt, wie viele Bins gebraucht werden.

Dazu bestimmen wir zunächst alle möglichen Multikombinationen von Gewichten, die auf eine

Maschine passen. Sei dazu n_j die Anzahl der Objekte mit Gewicht j.

$$Q = \left\{ (a_1, \ldots, a_k) \in \mathbb{N}^k \;\middle|\; \sum_{i=1}^{k} i \cdot a_i \leq b \right\} \setminus \{ (0, \ldots, 0) \}$$

Dann haben wir die Tabelle

$$\begin{aligned} f : \underline{n_1} \times \ldots \times \underline{n_k} &\to \mathbb{N} \\ (0, \ldots, 0) &\mapsto 0 \\ (a_1, \ldots, a_k) &\mapsto 1 + \min_{\overline{q} \in Q} f(a_1 - q_1, \ldots, a_k - q_k). \end{aligned}$$

Diese können wir nun schrittweise berechnen. Dabei müssen höchstens $(n+1)^k$ Einträge berechnet werden, die Laufzeit der Berechnung jedes Eintrags ist in $\mathcal{O}(|Q|)$. Da jeder Eintrag $q \in Q$ an Stelle i höchstens Wert $\lfloor \frac{b}{i} \rfloor$ annehmen kann, ist $|Q| \leq \frac{(b+1)^k}{k!}$. Also ist die gesamte Laufzeit in $\mathcal{O}((n+1)^k \cdot (b+1)^k \cdot \frac{1}{k!})$.

Für das Schedulingproblem hängen k und b nur von ε ab, sind also konstant. Mit $k = \lceil \frac{1}{\varepsilon^2} \rceil$ ergibt sich also die Laufzeit $\mathcal{O}(n^{\lceil \frac{1}{\varepsilon^2} \rceil})$.

5.4 Allgemeine Maschinen

(Hier dein MR)

6 Lineare Programmierung

In einem *Linearen Programm (LP)* soll eine lineare *Zielfunktion* unter Einhaltung von linearen *Nebenbedingungen* maximiert werden.

In der Vorlesung werden LP's nur in *kanonischer Form* in *Matrixschreibweise* betrachtet.

Umformungen in kanonische Form:

- Für ein Minimierungsproblem setzt man einfach c auf $-c$.
- Eine Nebenbedingung $a_{j,-} \cdot x = b_j$ wird durch zwei Nebenbedingungen mit $\leq$ und $\geq$ ersetzt.
- Eine Nebenbedingung $a_{j,-} \cdot x \geq b_j$ wird zu $-a_{j,-} \cdot x \leq -b_j$.

Definition: Lineares Programm (kanonische Form, Matrixschreibweise)

Eingabe:

- Zielvektor $c \in \mathbb{R}^d$,
- Bedingungen $A \in \mathbb{R}^{m \times d}$, $b \in \mathbb{R}^m$.

Ausgabe: $x \in \mathbb{R}^d$,
sodass $A \cdot x \leq b$ und $x \geq 0$.

Ziel: Maximiere $c^T \cdot x$.

Die geometrische Interpretation ist einfach: Die Variablenbelegung x entspricht einem Punkt im d-dimensionalen Raum. Eine Nebenbedingung ist eine Hyperebene, die einen Halbraum definiert. Der Schnitt aller solchen Halbräume ist der Lösungsraum, ein Polyhedron.

Ein lokales Optimum in einem Linearen Programm ist auch ein globales Optimum. Dies kann gezeigt werden, da das Polyhedron als Schnitt von Halbräumen konvex ist.

Ein LP heißt *zulässig*, falls es Lösungen gibt. Es heißt *unbeschränkt*, wenn die Lösungswerte nicht beschränkt sind. Das LP ist *degeneriert*, wenn mehr als d Nebenbedingungen sich in einem Punkt treffen. Bei einem degenerierten LP kann es zu Problemen bei Simplex kommen.

6.1 Geometrische Lösung

(Hier dein MR)

6.2 Simplex

Der Ansatz des *Simplex-Verfahrens* ist intuitiv: Man läuft entlang der Kanten des Polyhedrons in Richtung Optimum. Simplex geht zwar nicht immer den schnellsten Weg, aber kommt immer am Optimum an. Die Anzahl der Schritte ist nicht bekannt, denn die Laufzeit von Simplex ist nicht bewiesen.

Zunächst einige Definitionen. Simplex arbeitet auf der Algebraischen Gleichungsform und bestimmt Basislösungen auf geordneten Spaltenauswahlen.

Die *Algebraische Gleichungsform* drückt die Nebenbedingungen durch Einführung von *Schlupfvariablen* in linearer Gleichungsform aus. Damit können wir die Matrixkalkül-Theorie nutzen.

Eine *geordnete Auswahl* δ ist eine Abbildung
$\delta : \underline{m} \mapsto \underline{n}$ mit $\forall i \in \underline{m-1} : \delta(i) < \delta(i+1)$.

$\overline{\delta}$ ist die geordnete Auswahl der restlichen d Elemente. Damit können Spalten aus A ausgewählt und zu zwei neuen Matrizen A_δ und $A_{\overline{\delta}}$ gruppiert werden, analog für Elemente aus x und c.

Damit aufgespaltenene Nebenbedingungen:
$$A \cdot x = A_\delta \cdot x_\delta + A_{\overline{\delta}} \cdot x_{\overline{\delta}} = b.$$

Mit $x_{\overline{\delta}} = 0$ gilt $x_\delta = A_\delta^{-1} b$.

Das so aus den *Basisvariablen* x_δ und den *Nichtbasisvariablen* $x_{\overline{\delta}}$ zusammengesetzte x heißt *Basislösung zur Basis* δ.

Algorithmus: Umwandlung in die Algebraische Gleichungsform

Eingabe: LP in kanonischer Form:
Maximiere ${c'}^T \cdot x'$ unter $A' \cdot x' \leq b, x' \geq 0$.

Ausgabe: Algebraische Gleichungsform:
Maximiere $c^T \cdot x$ unter $A \cdot x = b, x \geq 0$.

Ersetze jede Ungleichung
$a_i \cdot x_i \leq b_i$
durch
$a_i \cdot x_i + s_i = b_i$
mit neuer Schlupfvariable $s_i \geq 0$.
Dadurch entsteht ein neues LP mit:

$$c = \begin{pmatrix} c' \\ 0 \end{pmatrix} \in \mathbb{R}^{d+m}$$

$$A = \begin{pmatrix} A' & I_m \end{pmatrix} \in \mathbb{R}^{m \times (d+m)}$$

$$x = \begin{pmatrix} x' \\ s \end{pmatrix} \in \mathbb{R}^{d+m}$$

Eine Basislösung heißt *zulässig*, wenn $x_\delta \geq 0$ gilt. In einer zulässigen Basislösung sind d Nichtbasisvariablen gleich 0. Wenn also eine Nichtbasisvariable x_i gleich 0 ist, dann ist sie entweder eine Schlupfvariable s_l oder eine Variable x'_k aus dem originalen LP.

Im ersten Fall liegt die Lösung auf der l-ten Hyperebene im originalen LP, im zweiten Fall liegt die Lösung auf der Hyperebene, die durch $x'_k = 0$ bestimmt ist.

Somit liegt eine zulässige Basislösung auf d Hyperebenen im originalen LP und entspricht einem Knoten vom Lösungspolyhedron des originalen LPs.

Algorithmus: Simplex

Eingabe: LP in kanonischer Form:
Maximiere ${c'}^T \cdot x$ unter $A' \cdot x' \leq b$.

Ausgabe: Lösung x_δ des LPs.

1. Wandle Eingabe in Algebraische Gleichungsform um.
2. Bestimme initiale Basislösung p:
   a) Erstelle neue Variablen h_i für jede Gleichung.
   b) Ersetze $\sum_{j=1}^{n} a_{i,j} \cdot x_j = b_i$ durch $\sum_{j=1}^{n} a_{i,j} \cdot x_j + h_i = b_i$.
   c) Neue Zielfunktion: Minimiere $f(h) = \sum_{k=1}^{m} h_k$.
   d) Starte mit der direkten Basislösung $x = 0$ und löse mithilfe von Schritten 3 und 4.
   e) Falls $f(h) > 0$, dann gibt es gar keine Basislösung. Abbruch.
   f) Falls $f(h) = 0$, dann benutze Lösung und konstruiere das initiale δ.
3. Berechne $r = c_{\overline{\delta}}^T - c_\delta^T \cdot \hat{A}$.
4. Solange r einen positiven Eintrag $r_{j'}$ mit $j = \overline{\delta}(j')$ hat:
   a) Falls $\forall\, i \in \{1, \ldots, m\} : \hat{a}_{i,j} \leq 0$, ist das LP unbeschränkt. Abbruch.
   b) Wähle $i = \underset{1 \leq k \leq m}{\operatorname{argmin}} \left\{ \frac{\hat{b}_k}{\hat{a}_{k,j}} \;\middle|\; \hat{a}_{k,j} > 0 \right\}$.
   c) Ersetze Spalte $A_{\delta(i)}$ durch Spalte A_j, d.h. definiere δ so um, dass $\delta(i) = j$ und $\delta(j') = \delta(i)$ ist.
   d) Berechne $r = c_{\overline{\delta}}^T - c_\delta^T \cdot \hat{A}$.

Simplex benötigt eine initiale zulässige Basislösung. Dafür wird in Schritt 2 des Algorithmus aus dem LP P ein neues LP P' konstruiert, welches per Konstruktion schon eine initiale Basislösung hat: $x = 0$. Damit wird P' mit dem Simplex-Verfahren rekursiv gelöst, wobei die Hilfsvariablen h_k minimiert werden.

Wenn hier für das Ergebnis x^* bzw. f^* gilt, dass $f(h^*) = 0$, dann sind alle Hilfsvariablen auch gleich 0. Außerdem sind dann alle $x_j^* \neq 0$ und und x^* erfüllt alle Nebenbedingungen von P. Also ist x^* eine zulässige Basislösung von P.

Für Schritte 3 und 4 sind $\hat{b} = A_\delta^{-1} \cdot b$ und $\hat{A} = A_\delta^{-1} \cdot A_{\overline{\delta}}$, definiert. Die in Schritt 4 ersetzte Spalte $A_{\delta(i)}$ wird *Eingangspivotspalte* genannt. Ihre Ersetzung A_j mit $j = \overline{\delta}(j')$ heißt *Ausgangspivotspalte*.

Durch Umformungen können die Basisvariablen in Abhängigkeit der Nichtbasisvariablen ausgedrückt werden: $x_\delta = \hat{b} - \hat{A} \cdot x_{\overline{\delta}}$. Selbiges für die Zielfunktion: $c^T \cdot x = c_\delta^T \cdot \hat{b} - r \cdot x_{\overline{\delta}}$.

Es kann nun Folgendes gezeigt werden:

- Eine Basislösung mit nichtpositivem *Vektor der reduzierten Kosten* r ist eine optimale Lösung des LPs.
- Eine Erhöhung einer Nichtbasisvariable x_j, mit entsprechendem positiven Eintrag in r, erhöht die Zielfunktion.
- Wenn x_j erhöht wird und die restlichen Nichtbasisvariablen auf 0 fixiert bleiben, verändert sich die Basisvariable $x_{\delta(i)}$ in Abhängigkeit zu $\hat{a}_{i,j}$:
  - $\hat{a}_{i,j} \geq 0$: $x_{\delta(i)}$ vergrößert sich.
    Falls dies darüber hinaus für alle Basisvariablen gilt, dann ist das LP unbeschränkt.
  - $\hat{a}_{i,j} < 0$: $x_{\delta(i)}$ verkleinert sich.

Diese Eigenschaften ermöglichen die Erkennung des Optimums in Schritt 3. Weiterhin kann sonst in Schritt 4 immer ein Pivotschritt in Richtung Optimum ausgeführt werden.

Ein Pivotschritt läuft in polynomieller Zeit, weil $\mathcal{O}(n \cdot m)$ viele algebraische Rechenoperationen ausgeführt werden. Die dabei verwendeten Zahlen lassen sich über gekürzte Brüche von Binärzahlen abschätzen. Mithilfe der Cramerschen Regel kann man dann sehen, dass die Länge der Zahlendarstellungen polynomiell in der Länge der Eingabezahlen ist.

Satz: Laufzeit von Simplex

1. Die Laufzeit eines Pivotschrittes ist polynomiell beschränkt in der Eingabelänge.
2. Es kann ein LP mit n Variablen konstruiert werden, welches beim Lösen $2^n - 1$ Pivotschritte brauchen kann.

Wir können außerdem eine Eingabe konstruieren, auf der Simplex exponentiell viele Pivotschritte nehmen kann. In dieser Eingabe gibt es aber auch Pfade polynomieller Länge. Es ist unbekannt, ob es auch eine Eingabe mit nur exponentiell langen Pfaden gibt, sodass es keine Ausführung von Simplex in Polynomialzeit gibt. (Hier dein MR)

6.2.1 Perturbierung

Wenn ein LP degeneriert ist, kann es zu Problemen bei Simplex kommen. In einem Pivotschritt kann die Lösung auf dem selben Punkt bleiben, da in einem degenerierten LP mehrere Knoten in einem Punkt landen. Schlimmstenfalls kann es zu Zykeln in den Pivotschritten kommen.

Um dieses Problem zu umgehen, *peturbiert* man LPs. Dabei wird auf b ein Vektor mit kleinen Zahlen ε^i addiert.

Insbesondere kann man garantieren, dass alle Nichtbasisvariablen größer null sind, wenn man ε kleiner γ und γ als kleinste aller Nullstellen aller als Polynome $p_{\delta,i}(\varepsilon)$ aufgefassten veränderten Basisvariablen $x_{\delta(i)}$ wählt (Korrektheit 1).

$$p_{\delta,i}(s) = \hat{b}_i + \sum_{j=1}^{m} a'_{i,j} \cdot s^i$$

Darüber erhalten wir, dass $p_{\delta,i}(0)$ als Basisvariable im nicht perturbierten LP strikt positiv ist und ferner Korrektheit 2.

Algorithmus: Perturbierung

Eingabe: Degeneriertes LP in algebraischer Gleichungsform.
Maximiere $c^T \cdot x$ unter $A \cdot x = b, x \geq 0$.

Ausgabe: Nicht degeneriertes LP $LP(\varepsilon)$.

1. Wähle ε klein (siehe Text).
2. Addiere $\vec{\varepsilon}$ auf b mit $\vec{\varepsilon} = (\varepsilon, \varepsilon^2, \ldots, \varepsilon^n)^T$.

Satz: Korrektheit der Perturbierung

Für ein $\gamma > 0$ gilt für alle $\varepsilon \in (0, \gamma)$:

1. Das LP $LP(\varepsilon)$ ist nicht degeneriert.
2. Eine zulässige Basislösung von $LP(\varepsilon)$ ist eine zulässige Lösung von LP.
3. Eine optimale Basislösung von $LP(\varepsilon)$ ist eine optimale Lösung von LP.

Da die Perturbierung nur b verändert, bleibt der Vektor der reduzierenden Kosten gleich. Damit ist weiterhin eine optimale Basislösung in $LP(\varepsilon)$ auch optimal für LP (Korrektheit 3).

Noch ein paar Worte zur Wahl von ε. Anstatt viele Nullstellen zu bestimmen, kann man $\varepsilon = \frac{1}{2\beta^2}$ wählen, wobei β größer als Zähler und Nenner aller β_i in $p(s) = \sum_{i=0}^{m} \beta_i \cdot s^i$ ist. Denn dann kann man zeigen, dass $|p(q)| > 0$ für $q \in (0, \frac{1}{2\beta^2}) = (0, \varepsilon)$.

Die Laufzeit der Perturbierung ist abhängig von der Länge der Zahlendarstellungen.

Wird ε wie oben gewählt, kann analog wie im Beweis zur Laufzeit eines Pivotschrittes bewiesen werden, dass die Länge der Zahlendarstellung von ε polynomiell in der Eingabelänge ist.

Satz: Laufzeit mit Perturbierung

Die Laufzeit eines Pivotschrittes mit Perturbierung ist polynomiell in der Eingabelänge.

Daraus folgt eine polynomielle Längenabschätzung von ε, und so auch $b_i + \varepsilon^i$ in Abhängigkeit von der Eingabelänge.

Die *symbolische Perturbierung* ermöglicht eine Verbesserung der Laufzeit der Perturbierung. Dazu wird bei Simplex in Schritt 4b $\hat{b}_k$ durch $p_{\delta,k}(\varepsilon)$ ersetzt und aus der resultierenden Abbildung $i(\varepsilon)$ den Grenzwert für $\varepsilon \to 0$ als Koeffizient für die Ausgangspivotspalte benutzt.

6.3 Ellipsoid-Methode

Die *Ellipsoid-Methode* ist ein Algorithmus, der in polynomieller Laufzeit bestimmen kann, ob ein lineares Ungleichungssystem eine Lösung hat, d.h. der Lösungsraum nicht leer ist.

Die Begriffe *Kugel* und *Ellipsoid* sind wichtig: $K_r(y)$ ist eine Kugel um den Punkt y mit Radius r. Ein Ellipsoid ist eine affin transformierte Kugel.

Algorithmus: Ellipsoid-Methode

Eingabe: Lineare Ungleichungen $A \cdot x \leq b$.
Mit α der größte Absolutwert aller Eingabezahlen.
Mit n Variablen, m Ungleichungen und Eingabelänge L.

Ausgabe: Ist der Lösungsraum leer?

1. Perturbiere das System zu $A \cdot x \leq b + (\varepsilon, \varepsilon^2, \ldots, \varepsilon^m)^T$.
2. Füge $\forall i \in \underline{n} : x_i \leq (\alpha \cdot m)^m$ und $x_i \geq -(\alpha \cdot m)^m$ hinzu.
3. Setze $u = \sqrt{n} \cdot (\alpha \cdot m)^m$ und $l = \frac{\varepsilon^i}{u}$.
4. Initialisiere $E = K_u(0)$.
5. Solange $|E| \geq |K_l(0)|$:
   a) Setze z als Mittelpunkt von E.
   b) Falls z das System erfüllt, gib **Ja** aus.
   c) Wähle eine Ungleichung aus, die z nicht erfüllt.
   d) Setze H als Hyperebene mit $z \in H$, welche parallel zur Hyperebene zur ausgewählten Ungleichung ist.
   e) Setze $\hat{H}$ als Halbraum von H mit $z \in \hat{H}$.
   f) Bestimme Ellipsoid E' mit kleinstem Volumen und $E \cap \hat{H} \subset E'$.
   g) Setze $E = E'$.
6. Gib **Nein** aus.

Mit den Veränderungen des initialen Ungleichungssystems in Schritt 1 und 2 kann man 4 Aussagen über das neue Gleichungsystem machen.

Zu Teil 4: Durch Konstruktion ist der maximale Absolutwert von G' $\alpha' = (\alpha \cdot m)^m$, welches in $\mathcal{O}(L^2)$ ist.

Zu Teil 1: $\Leftarrow$: Ist offensichtlich, da Ungleichungen weggenommen werden. $\Rightarrow$: Es wurde für die Laufzeit von Simplex gezeigt, dass für eine Lösung x eines lineares Gleichungssystems, hier G, mit maximalen Absolutwert α und m Gleichungen gilt: $|x_i| \leq (\alpha \cdot m)^m$. So ist auch x eine Lösung von G'.

Zu Teil 2: Durch Konstruktion von G' mit den neuen Ungleichungen gilt für eine Lösung x: $|x_i| \leq (\alpha \cdot m)^m$. So ist der Lösungraum ein in Ursprung zentrierter Hyperwürfel, der in eine Kugel mit Radius $r = \sqrt{n} \cdot (\alpha \cdot m)^m \in 2^{\mathcal{O}(L^2)}$ passt.

Satz: Eigenschaften des Ellipsoid-Methode

Ein lineares Ungleichungssystem G mit Eingabenlänge L kann in polynomieller Laufzeit in ein lineares Ungleichungssystem G' mit folgenden Eingenschaften umgewandelt werden:

1. G hat eine Lösung $\Leftrightarrow$ G' hat eine Lösung.
2. Der Lösungsraum von G' ist in einer Kugel $K_r(0)$ mit $r \in 2^{\mathcal{O}(L^2)}$.
3. Falls der Lösungraum von G' nicht leer ist, enthält er eine Kugel $K_r(y)$ mit $r \in 2^{-\mathcal{O}(L^4)}$.
4. Die Eingabenlänge von G' ist in $\mathcal{O}(L^2)$.

Zu Teil 3: Es kann gezeigt werden, dass der Abstand zwischen der $i$ten Hyperebene in G und der perturbierten Ebene in G' mindestens $\frac{\varepsilon^i}{\sqrt{n} \cdot \alpha'} \geq 2^{-\mathcal{O}(L^4)}$ ist. Dies hat zur Folge, dass eine zulässige Lösung x in G', welche auch eine zulässige Lösung von G ist, $2^{-\mathcal{O}(L^4)}$ von allen Hyperebenen in G' entfernt ist. Und so der Lösungraum mindestens eine Kugel $K_l(x)$ enthält.

Die erste Behauptung wird nur für 2 dimensionale Ungleichungssysteme gezeigt. Erst wird die Behauptung für einen Spezialfall gezeigt. Danach kann gezeigt werden, dass der allgemein Fall durch flächenverhältniserhaltende Transformationen auf den Spezialfall zurück gebracht werden.

Der Spezialfall hat folgende Eigenschaften:

- E ist ein Kreis mit Radius 1 um den Ursprung.
- Die neue Hyperebene H verläuft entlang der y-Achse, sodass der Lösungsraum links der y-Achse liegt.

Satz: Laufzeit der Ellipsoid-Methode

1. Pro Schleifendurchlauf verringert sich das Volumen des Ellipsoiden mindestens um einen Faktor von $2^{\frac{-1}{2(n+1)}}$.
2. Die Ellipsoid-Methode terminiert nach $\mathcal{O}(n^2 \cdot L^4)$ Schleifendurchläufen.
3. Die Ellipsoid-Methode kann in polynomieller Zeit bestimmen, ob ein lineares Ungleichungssystem eine Lösung hat.

Damit ergeben sich 3 Extrempunkte, $(0,1)$, $(0,-1)$ und $(-1,0)$, die auf dem Rand von E' liegen müssen. Damit lässt sich E' eindeutig bestimmen. Es lässt sich dann zeigen, dass $E \cap \hat{H} \subset E'$ gilt. Das Flächenverhältnis von E und E' kann nun errechnet werden.

Der zweite Teil kann gezeigt werden, indem man das durch $(\frac{u}{l})^n$ bestimmte Volumenverhältnis und die Volumenreduktion pro Schleifendurchlauf benutzt, um die Anzahl der Iteration abzuschätzen.

Der dritte Teil folgt aus dem polynomiell langen, polynomiell oft zu durchlaufenen Schleifenkörper und der polynomiellen Länge der Transformation von G zu G'.

6.3.1 Polynomieller Algorithmus für LP

Algorithmus: Polynomielle Lösung eine LP's

Eingabe: LP in algebraischer Form:
Maximiere ${c'}^T \cdot x$ unter $A' \cdot x' \leq b$. Mit α der größte Absolutwert aller Eingabezahlen.
Mit n Variablen, m Ungleichungen und Eingabelänge L.

Ausgabe: Lösung x_δ des LPs.

1. Falls $\mathcal{A}(A \cdot x \geq b, A \cdot x \leq b, x \geq 0) = \emptyset$, gib **Keine Lösung** aus.
2. Falls $\mathcal{A}(A \cdot x \geq b, A \cdot x \leq b, x \geq 0, c^T \cdot x \geq (\alpha \cdot m)^{m+1} + 1) \neq \emptyset$, gib **unbeschränkt** aus.
3. Setze $\tau = (\alpha \cdot m)^{-2(m+1)}$.
4. Bestimme mit Binärsuche $K \in \mathbb{N}$ mit $\tau \cdot K \leq z^* < \tau \cdot (K+1)$.
   - Rufe dazu $\mathcal{A}(A \cdot x \geq b, A \cdot x \leq b, x \geq 0, c^T \cdot x \geq \tau \cdot K)$ auf.
5. Initialisiere $S = \emptyset$ und $m' = 1$.
6. Wiederhole $\forall i \in \underline{n}$:
   - Falls $\mathcal{A}(A \cdot x \geq b, A \cdot x \leq b, x \geq 0, c^T \cdot x \geq \tau \cdot K, x_{j \in S \cup \{i\}} \leq 0) \neq \emptyset$.
   - so setze $S = S \cup \{i\}$, $m' = m' + 1$, und $\overline{\delta}(m') = i$.
7. Bestimme δ aus $\overline{\delta}$ dadurch $x_\delta = A_\delta^{-1} \cdot b$.

Die Ellipsoid-Methode kann verwendet werden um ein LP zu lösen. Die Idee dahinter ist, dass für ein LP, $(c^T \cdot x \geq z, A \cdot x \leq b)$ eine gültige Eingabe für die Ellipsoid-Methode ist. Somit kann z mit einer Binärsuche maximiert werden.

Dies geschieht in polynomiell vielen Schritten, wenn wir z, bis auf einen additiven Fehler $\delta \in \mathcal{O}(2^{-poly(L)})$, bestimmen.

Zum Glück lässt sich eine untere Schranke $\tau = (\alpha \cdot m)^{-2/(m+1)}$ für die Differenz der Zielfunktionswerte von zwei verschiedenen Lösungen bestimmen.

Wenn wir τ als additiven Fehler wählen, können wir z genau maximieren, indem wir per Binärsuche ein $K \in \mathbb{N}$ finden mit $\tau \cdot K \leq z^* < \tau \cdot (K+1)$.

Nun müssen wir nur noch herausfinden, wie x^* aussieht. Wir wissen es ist eine Basislösung. Also wird jedes Element von x auf 0 fixiert und getestet ob sich das LP dann noch lösen lässt. Daraus könne wir dann die Basis bestimmen.

Da alle Eingaben von $\mathcal{A}$ polynomiell in L sind und $\mathcal{A}$ polynomiell oft aufgerufen wird, hat der Algorithmus eine polynomielle Laufzeit.

6.4 Algorithmus von Seidel

Die Idee hinter Seidel ist, dass Ungleichungssysteme mit wenig Dimensionen oder mit wenig Nebenbedingungen einfach zu lösen sind. Der Idee ist nun nach und nach Nebenbedingungen hinzu zu nehmen, und zu überprüfen ob die Nebenbedingung verletzt wird. Wenn nicht, kann die nächste Bedingung gewählt werden, wenn ja, muss die Lösung angepasst werden.

Algorithmus: Seidel

Eingabe: LP in kanonischer Form:
Maximiere $c^T \cdot x$ unter $A \cdot x \leq b$.
Das LP ist beschränkt und hat eine eindeutige Lösung.
H ist die Menge der Nebenbedingungen.

Ausgabe: Lösung $\text{opt}(H)$ des LP's.

1. Falls $d = 1$ oder $m = 1$ gebe $\text{opt}(H)$ aus.
2. Wähle uniform ein $h \in H$ und berechne rekursiv $\text{opt}(H \setminus \{h\})$.
3. Falls $\text{opt}(H \setminus \{h\})$ h nicht verletzt, gebe $\text{opt}(H) := \text{opt}(H \setminus \{h\})$.
4. Löse rekursiv ein LP, dessen Lösungspolyhedron dem Schnitt des Lösungpolyhedrons von H und der Hyperebene von h ist.

Man kann ein zulässiges LP so verändern, dass es die Vorraussetzungen für den Algorithmus von Seidel erfüllt.

Eindeutigkeit: Der Zielfunktionsvektor eines Ungleichungssystems kann so virtuell perturbiert werden, dass erstens zwei unterschiedliche Lösungen mit gleichen altem Zielfunktionswert unterschiedliche neue Zielfunktionswerte besitzen, und zweitens eine Lösung im neuen System maximal ist genau dann wenn sie ein Maximum im alten System ist. So kann die Eindeutigkeit gewährleistet sein.

Beschränktheit: Um die Beschränktheit zu gewährleisten, fügen wir die Box-Bedingungen

$$-t \leq x_i \leq t \quad \forall i \in \underline{n}$$

hinzu. Man kann in Polynomialzeit dazu einen Wert t berechnen, der die zulässigen Belegungen nicht einschränkt, außer das LP ist unbeschränkt. Alternativ kann man auch mit einer symbolischen Schranke rechnen.
(Hier dein MR)

Korrektheit: Sei $k = 2 \cdot d + m$. In jedem Rekursionsschritt nimmt k um eins ab und bei $k = 3$ mit $d = 1$ oder $m = 1$ terminiert die Rekursion. So läuft der Algorithmus nicht unendlich.

Laufzeit: Sei $T(d, m)$ die Laufzeit von Seidel.

Zu Schritt 1: Falls $m = 1$ gilt hat man ein relaxiertes Rucksack-Problem welches in $\mathcal{O}(d \log d) \subseteq \mathcal{O}(d^2)$ gelöst werden kann. Falls $d = 1$ kann man in $\mathcal{O}(m)$ die Lösung für die eine Variable x besimmt werden. Also insgesamt $\mathcal{O}(1)$.

Zu Schritt 2: Hier wird Seidel rekursiv aufgerufen mit der Laufzeit $T(d, m-1)$.

Zu Schritt 3: Der Test, ob h verletzt wird, kann in $\mathcal{O}(d)$ ausgeführt werden.

Zu Schritt 4: Da die optimale Lösung des LP's von d Hyperebenen bestimmt wird, kann die rekursiv berechnete Lösung nur h verletzen, falls h Teil der d Hyperebenen ist. Dies passiert mit einer Wahrscheinlichkeit von höchstens $\frac{d}{m}$. Somit wird Schritt 4 höchstens mit deser Wahrscheinlichkeit ausgeführt. Zur Vorbereitung auf diesen Schritt wird h als Gleichung aufgefasst und nach einer beliebigen Variable aufgelöst. Jedes Vorkommnis dieser Variable, in allen Gleichungen und der Zielfunktion, wird durch diese umgeformte Gleichung von h ersetzt. Die Box-Bedingungen dieser Variable werden genau so behandelt und mit in H aufgenommen. Dies ergibt $m+2-1 = m+1$ Gleichungen und $d-1$ Dimensionen und so eine Laufzeit von $\mathcal{O}(d-1, m+1)$.

Die gesamte Laufzeit, nun ohne konstante Faktoren, ist $T(d,m) \leq d^2 + T(d,m-1) + \frac{d}{m} \cdot T(d-1, m+1)$ für $d > 1$ und $m > 1$ und $T(d,m) \leq d^2 + m$ für $d = 1$ oder $m = 1$.

Mit etwas Rechnerei kann man eine geschlossene Formel für $T(d,m)$ finden: $T(d,m) \leq (m-1) \cdot f(d) + 2 \cdot d^2$ mit $f(d) \in \mathcal{O}(d!)$.

Satz: Laufzeit des Seidel Algorithmus

Der Seidel-Algorithmus löst ein d-dimensionales LP mit m Nebenbedingungen in $\mathcal{O}(d! \cdot m)$, also linear zu m.

7 Randomisierte Algorithmen

7.1 Minimaler Schnitt

Wir betrachten im Folgenden randomisierte Algorithmen zur Bestimmung minimaler Schnitte C^*, das heißt $\forall\, U \subset V : c(U) \geq c(C^*)$.

Da wir einen minimalen Schnitt suchen, gehen wir davon aus, dass dieser klein sein wird. Wir wählen also zufällig (entsprechend ihrer Gewichte) Kanten aus und *kontrahieren* sie, d.h. verschmelzen die anliegenden Knoten. Aus den Eigenschaften der Kantenkontraktion folgt, dass inzidente Knoten einer zufällig gewählten Kante mit hoher Wahrscheinlichkeit beide im gleichen Teil des Schnittes liegen.

Algorithmus: Kantenkontraktion

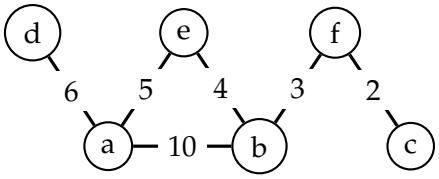
Eingabe: Graph $G = (V, E, c)$ und Kante $\{ v, w \}$.

Ausgabe: Graph G' mit kontrahierter Kante.

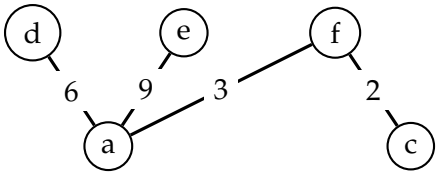
1. Knoten entfernen: $V' = V \setminus \{ w \}$.
2. Kanten verschmelzen: $E' = E \cup \{ \{ x, v \} \mid \{ x, w \} \in E \wedge x \neq v \} \setminus \{ \{ x, w \} \in E \}$.
3. Gewichte setzen:

$$c'(e) = \begin{cases} c(e) & \text{falls } v, w \notin e, \\ c(e) & \text{falls } e = \{ x, v \} \wedge \{ x, w \} \notin E, \\ c(\{ x, w \}) & \text{falls } e \neq \{ x, v \} \wedge \{ x, w \} \in E, \\ c(e) + c(\{ x, w \}) & \text{falls } e = \{ x, v \} \wedge \{ x, w \} \in E \end{cases}$$

4. $G' = (V', E', c')$.



Bsp. Kantenkontraktion



Nach Kontraktion von $\{ a, b \}$

Der *Algorithmus von Karger* setzt die obige Idee einfach um. Er erstellt für jeden Knoten eine Menge die ihn selbst enthält. Zwei Knoten werden durch Kantenkontraktion fusioniert und zugehörige Mengen zusammengeführt, bis nur noch zwei Knotenmengen vorhanden sind: Ein Schnitt.

Dann wird die Knotenmenge des einen Knoten ausgegeben.

Da eine Kontraktion eine Laufzeit von $\mathcal{O}(n)$ hat, und in jeder Kontraktion ein Knoten abgearbeitet wird, ergibt sich für den Algorithmus von Karger insgesamt eine Laufzeit von $\mathcal{O}(n^2)$.

Algorithmus: Algorithmus von Karger

Eingabe: Ein Graph $G = (V, E, c)$.

Ausgabe: Ein Schnitt $S(v)$.

1. Setze $\forall v : \ S(v) := \{ v \}$ und $H := G$.
2. Solange $|V(H)| > 2$:
   a) Wähle $\{ v, w \} \in V(H)$ zufällig, mit Wahrscheinlichkeit $c(\{ v, w \})$.
   b) Setze $S(v) = S(v) \cup S(w)$ und kontrahiere $\{ v, w \}$ in H.
   c) Gebe $S(v)$ für ein $v \in V(H)$ aus.

Satz: Erfolgswahrscheinlichkeit des Schnitts

$P(C \text{ ist ein Min-Cut von } G) \geq \binom{n}{2}^{-1}$.

(Hier dein MR)

Lassen wir diesen Algorithmus nun $t \cdot \binom{n}{2}$ Mal laufen (wobei wir $t \in \mathbb{N}$ frei wählen), so ist die Wahrscheinlichkeit, den Min-Cut nicht zu finden:

$$\left(1 - \binom{n}{2}^{-1}\right)^{t \cdot \binom{n}{2}} \leq \left(\frac{1}{e}\right)^{t}.$$

Heißt, wir können die Fehlerwahrscheinlichkeit beliebig klein machen und so unsere Laufzeit töten, die mittlerweile wegen dem $\binom{n}{2}$ schon $\mathcal{O}(n^4)$ beträgt; und mal ganz ehrlich, das ist keine Verbesserung gegenüber herkömmlichen Algorithmen.

Aber die Wahrscheinlichkeit, eine Kante des Min-Cuts wegzukontrahieren, ist am Anfang des Algorithmus relativ gering und steigt, sobald die Anzahl der Kanten sich zu sehr verringert. Die Idee wäre also den Algorithmus so zu verändern, dass wir am Anfang schnell Kanten kontrahieren und die letzten Schritte sicher machen.

Das ist der Gedanke des *Algorithmus von Karger und Stein*, auch *FastCut* genannt. Dieser wird mit einem beliebigen Graph aufgerufen. Auf zwei Instanzen dieses Graphen werden rekursiv $t = \lceil 1 + \frac{n}{\sqrt{2}} \rceil$ Kanten kontrahiert, bis nur noch höchstens c (oft mit $c = 6$) Kanten übrig sind.

Von da an wird mit Brute-Force der Min-Cut bestimmt. Danach wird der kleinere Schnitt ausgegeben.

Das verbessert maßgeblich unsere Laufzeit: FastCut läuft in $\mathcal{O}(n^2 \cdot \log n)$.

Algorithmus: Algorithmus von Karger und Stein (FastCut)

Eingabe: Ein Graph $G = (V, E, c)$.

Ausgabe: Ein Schnitt C_x.

1. Gilt $|V(G)| \leq 6$, bestimme optimalen Schnitt C_U mit Brute-Force.
2. Setze $H := H' := G$ und $t = \lceil 1 + \frac{n}{\sqrt{2}} \rceil$.
3. Kontrahiere zufällig Kanten, bis in H und H' jeweils nur noch t Kanten sind.
4. Rufe FastCut rekursiv auf und setze $C = \text{FastCut}(H)$, $C' = \text{FastCut}(H')$.
5. Gebe $\min(C, C')$ aus.

Beweisideen zur Laufzeit:

- Es gilt $T(n) = 2 \cdot T(\lceil 1 + \frac{n}{\sqrt{2}} \rceil) + \mathcal{O}(n^2)$ für $n > 6$ und $T(n) = \mathcal{O}(1)$ für $n \leq 6$.
- Wegen des konstanten Faktors $\frac{1}{\sqrt{2}}$ haben wir eine Rekursionstiefe von $D = \mathcal{O}(\log n)$.
- Wir definieren uns n_I die Menge der Knoten auf Rekursionstiefe I.
- Auf Rekursionstiefe I lösen wir 2^I Teilproblem der Größe $n_I \equiv \frac{n}{\sqrt{2^I}}$.
- Dieses Teilproblem hat Laufzeit $\mathcal{O}(n_I^2) = \mathcal{O}(\frac{n^2}{2^I})$, auf einem Level also $2^I \cdot \mathcal{O}(\frac{n^2}{2^I}) = \mathcal{O}(n^2)$.
- Das multipliziert man mit der Rekursionstiefe für eine Laufzeit von $\mathcal{O}(n^2 \cdot \log n)$.

Satz: Zuverlässigkeit von FastCut

FastCut berechnet den Min-Cut mit einer Wahrscheinlichkeit von $\Omega(\frac{1}{\log n})$.

FastCut liefert nicht sicher einen Min-Cut, aber wenn wir den Algorithmus $(\log n)$-mal ausführen, erhalten wir eine Erfolgswahrscheinlichkeit von $\Omega(1)$, d.h. eine konstante Wahrscheinlichkeit.

(Hier dein MR)

7.2 3-SAT

Definition: 3-SAT

Eingabe:

- Eine Boolsche Formel $\mathcal{F}$ in 3-KNF $\mathcal{F}(x_1, \ldots, x_n) = \bigwedge_{i=1}^{m} c_i$,
- bestehend aus Klauseln $\forall\, i \in \underline{m} : c_i = (l_i^1 \vee l_i^2 \vee l_i^3)$,
- wiederum bestehend aus Literalen:

$$\forall\, i \in \underline{m} \quad \forall\, j \in \underline{3} : \quad l_i^j = \begin{cases} \neg x_p & \text{oder} \\ x_p & \text{für ein } p \in \underline{n} \end{cases}$$

Ausgabe: Eine Belegung $W : \{ x_1, \ldots, x_n \} \to \{ 0, 1 \}$.

Ziel: Finde eine erfüllende Belegung für $\mathcal{F}$.

Satz: 3-SAT

Für eine 3-KNF Formel zu bestimmen ob eine passende Belegung existiert ist NP-vollständig.

Das Problem wollen wir aber sinnvoll angehen und lösen. Die Idee dabei ist, eine Belegung zu raten, ähnliche Lösungen im Lösungsraum zu betrachten und ansonsten eine neue Belegung testen.

Algorithmus: Algorithmus von Schöning

Eingabe: Eine Boolsche Formel $\mathcal{F}(x_1, \ldots, x_n)$ in 3-KNF.
Ausgabe: Eine erfüllende Belegung $W : \{ x_1, \ldots, x_n \} \to \{ 0, 1 \}$ oder eine Fehlermeldung.

1. Wiederhole höchstens $\lceil 20 \cdot \sqrt{3 \cdot \pi \cdot n} \cdot \left(\frac{4}{3}\right)^n \rceil$ mal:
   a) Rate eine zufällige Belegung $W = (w_1, \ldots, w_n) \in \{ 0, 1 \}^n$, falls sie passt, gebe sie aus.
   b) Ansonsten führe maximal $3n$ mal aus:
      i. Wähle beliebige Klausel und daraus beliebiges Literal.
      ii. Kehre dieses Literal w_j um. Falls $\mathcal{F}$ erfüllt ist, gebe $W = (w_1, \ldots, \neg w_j, \ldots, w_n)$ aus.
2. Sonst terminiere und gebe aus, dass $\mathcal{F}$ nicht erfüllbar ist.

Satz: Laufzeit von Schöning

Der Algorithmus von Schöning hat eine Laufzeit von $\mathcal{O}(m \cdot n^{3/2} \cdot \left(\frac{4}{3}\right)^n)$.

Satz: Fehlerwahrscheinlichkeit von Schöning

Es können nur false negatives auftreten.

Falls $\mathcal{F}$ erfüllbar ist, ist die Fehlerwahrscheinlichkeit $5 \cdot 10^{-5}$.

Eine Belegung zu testen benötigt Laufzeit $\mathcal{O}(3m)$. Angesichts der Schleifen ist die Laufzeit des Algorithmus von Schöning relativ schnell ersichtlich.

Beweisideen der Fehlerwahrscheinlichkeit:

- Für eine passende Belegung W^* definieren wir Belegungsklassen, die der Anzahl der differierenden Bits j entsprechen.
- In der Klasse j zu landen hat eine Wahrscheinlichkeit von $p_j = \binom{n}{j} \cdot 2^{-n}$.

- Mit einer Mindestwahrscheinlichkeit von $\frac{1}{3}$ verbessern wir uns um eine Klasse. Umgekehrt ist die Verschlechterungswahrscheinlichkeit höchstens $\frac{2}{3}$.
- Die Wahrscheinlichkeit, um in $j + 2i \leq 3n$ Schritten nach W^* zu kommen ist

$$q_{j,i} = \binom{j+2i}{i} \cdot \frac{j}{j+2i} \cdot \left(\frac{1}{3}\right)^{j+i} \cdot \left(\frac{2}{3}\right)^{i}.$$

- Es ergibt sich für das Erreichen von W^* aus j, die Summe über alle i:

$$q_j = \frac{1}{2 \cdot \sqrt{3\pi j}} \cdot \left(\frac{1}{2}\right)^{j}.$$

- Summieren wir über alle j und runden das Ganze schlau, erhalten wir für das Erreichen von W^*:

$$\tilde{p} = \frac{1}{2 \cdot \sqrt{3\pi n}} \cdot \left(\frac{3}{4}\right)^{n}.$$

- Die Fehlerwahrscheinlichkeit ist dann nach $t = \lceil 20 \cdot \sqrt{3\pi n}\left(\frac{4}{3}\right)^{n} \rceil$ lokalen Suchen:

$$(1 - \tilde{p})^t \leq e^{\tilde{p}t} = e^{-10} \leq 5 \cdot 10^{-5}.$$

7.3 Vergleich

Randomisierte Algorithmen lassen sich in zwei Kategorien einteilen, nämlich einerseits *Las Vegas Algorithmen* und andererseits *Monte-Carlo Algorithmen*.

Definition: Las Vegas Algorithmus $\mathcal{A}$

- $\mathcal{A}$ liefert immer eine optimale Lösung.
- Die Laufzeit hängt von Zufallvariablen ab.

Definition: Monte-Carlo Algorithmus $\mathcal{B}$

- $\mathcal{B}$ liefert eine Lösung, die nur eventuell optimal ist.
- Die Lösung hängt nicht unbedingt von Zufallsvariablen ab.

Die Algorithmen, die wir im Laufe des Kapitels behandelt haben, sind allesamt Monte-Carlo Algorithmen. Es ist aber möglich, Las Vegas und Monte-Carlo Algorithmen aufeinander zu reduzieren.

Satz: Las Vegas $\leftrightarrow$ Monte-Carlo

Las Vegas und Monte-Carlo Algorithmen können aufeinander reduziert werden.

Ideen (Las Vegas $\mathcal{A} \rightarrow$ Monte-Carlo $\mathcal{B}$):

- Hat $\mathcal{A}$ erwartete Laufzeit $f(n)$, brechen wir nach $\alpha \cdot f(n), 0 < \alpha < 1$ ab.
- Ergebnis ist ein Monte-Carlo Algorithmus $\mathcal{B}$ mit Fehlerwahrscheinlichkeit $\frac{1}{\alpha}$.

Ideen (Monte-Carlo $\mathcal{B} \rightarrow$ Las Vegas $\mathcal{A}$):

- Habe $\mathcal{B}$ eine Laufzeit von $f(n)$ mit Wahrscheinlichkeit $p(n)$ und sei $\mathcal{C}$ eine Routine, die in $g(n)$ die Korrektheit von $\mathcal{B}$ prüft.
- Wir lassen $\mathcal{A}$ so lange $\mathcal{B}$ und $\mathcal{C}$ aufrufen, bis $\mathcal{C}$ die Korrektheit bestätigt.
- Ergebnis ist ein Las Vegas Algorithmus $\mathcal{A}$ mit erwarteter Laufzeit $\frac{f(n)+g(n)}{p(n)}$.

8 Online-Algorithmen

Offline-Algorithmen bekommen die gesamte Eingabe und können darauf das Problem lösen. *Online-Algorithmen* bekommen die Eingabe in Stücken, wobei für jedes Stück sofort eine unwiderrufliche Entscheidung getroffen werden muss.

Definition: Online-Algorithmus

Sei $\rho = \rho_1, \rho_2, \ldots, \rho_m$ die Eingabe für ein Problem Π. Ein Algorithmus A ist ein Online-Algorithmus, wenn:

- Im i-ten Schritt erfolgt die Eingabe ρ_i.
- A gibt auf $\rho_{1\ldots i} = \rho_1, \rho_2, \ldots, \rho_i$ eine Lösung für Π an.
- A darf die Entscheidung auf $\rho_{1\ldots i-1}$ nicht verändern.

Definition: c-competitive

Sei $C_A(\rho)$ die Kosten der von A berechneten Strategie auf der Eingabe ρ und $C^*(\rho)$ die Kosten für die optimale Strategie. Dann ist A *c-competitive* gdw. ein a existiert, so dass für jede Eingabe ρ gilt:

$$C_A(\rho) \leq c \cdot C^*(\rho) + a.$$

8.1 File-Allocation-Problem

Bei dem *File-Allocation-Problem* (FAP) gibt es mehrere Kopien einer Datei, die von mehreren Nutzern gelesen und beschrieben werden können. Die *Migration* zwischen verschiedenen Speicherorten erzeugt Kosten. Für dieses Problem möchten wir einen guten Online-Algorithmus entwickeln.

Dazu betrachten wir ein System mit zwei Rechnern a, b mit dem folgende Konfigurationen möglich ist: Die Datei ist nur auf a: [a], nur auf b: [b] oder auf a und b: [a,b].

Die Eingabesequenz ρ besteht aus Lese- und Schreibanfragen auf den Knoten.

Je nach Aktualität der Daten muss eventuell nachgeladen werden.

Definition: File-Allocation-Problem

Eingabe: Eingabesequenz $\rho = \rho_1, \ldots, \rho_m$ mit Anfragen

- $\rho_i = r(v)$: Knoten $v \in \{a, b\}$ will Datei lesen.
- $\rho_i = w(v)$: Knoten $v \in \{a, b\}$ will Datei schreiben.

Ausgabe: Sequenz von Übergängen der Form

- $[v] \to [a, b]$, $[a, b] \to [v]$
- für $v, u \in \{ a, b \}$

Ziel: Minimiere entstehende Kosten.

8.1.1 Kosten

Es entstehen je nach Änderung der Konfiguration *Service- und Migrationskosten*.
Sei $v \in \{a, b\}$, dann entstehen Servicekosten von einer Einheit...

- beim Lesen: Anfrage $\rho_i = r(v)$ in Konfiguration $[u]$, mit $u \in \{a, b\} \setminus \{v\}$
- beim Schreiben: Anfrage $\rho_i = w(v)$ in Konfiguration $[a, b]$ oder $[u]$, mit $u \in \{a, b\} \setminus \{v\}$

Migrationskosten von 1 enstehen, wenn die Konfiguration geändert wird, etwa wenn die Datei von einem auf beide Leser kopiert wird, jedoch nicht beim Löschen.

Wenn wir einen konstanten competitiven Faktor wollen, müssen wir intelligent migrieren:

- Von $[a, b]$ nach $[x]$, falls eine Anfrage $w(x)$ mit $x \in \{a, b\}$.
- Von $[a]$ nach $[b]$ oder $[a, b]$, falls eine Anfrage $r(b)$ erfolgt.
- Von $[b]$ nach $[a]$ oder $[a, b]$, falls eine Anfrage $r(a)$ erfolgt.

Die Konfiguration $[a, b]$ ist offenbar zu bevorzugen.

Satz: Obere und Untere Schranke

1. Für das uniforme FAP auf zwei Knoten gibt es einen 3-kompetitiven Online-Algorithmus.
2. Für das uniforme FAP auf zwei Knoten gibt es keinen c-kompetitiven Online-Algorithmus mit $c < 3$.

8.1.2 Obere Schranke

Im Folgenden zeigen wir einen Online-Algorithmus, der 3-competitive ist.

Die initialen Kosten sind einmalig, und erhöhen den competitiven Faktor nicht.

Fall 1: Tritt auf, wenn eine Lese- oder Schreibanfrage an einen Knoten ohne Datei kommt. Es entstehen je eine Kosteneinheit für die Anfrage und die Migration, also Kosten von 2.

Fall 2: Tritt auf, wenn eine Schreibanfrage an einen der Knoten kommt und beide die Datei aktuell hatten. Es entstehen Kosten von 1 für die Anfrage, da Löschen nichts kostet.

Algorithmus: Online-Algorithmus zu FAP

Eingabe: Lese- und Schreibanfragen.

Ausgabe: Änderung der optimalen Konfiguration nach jedem Schritt.

Initial gehe in die Kofiguration $[a, b]$.

Danach unterscheide zwei Fälle:

- Falls in der Konfiguration $[x]$ eine Anfrage $\rho_i \in \{r(v), w(v)\}$ mit $x \neq v$ kommt, gehe in die Konfiguration $[a, b]$.
- Falls in der Konfiguration $[a, b]$ eine Anfrage $\rho_i = w(v)$ kommt, gehe in die Konfiguration $[v]$.

Wir zeigen, dass das obige Verfahren 3-competitive ist. Im Folgenden sei dazu $x \in \{a, b\}$.

Wir starten in der Konfiguration $[a, b]$. Diese Konfiguration ändert sich erst nach dem ersten Schreibzugriff in $[a]$ oder $[b]$. O.B.d.A. gehen wir in die Konfiguration $[a]$. Von da aus gibt es nur einen möglichen Konfigurationswechsel auf $[a, b]$, ausgelöst durch Lesen oder Schreiben auf b. Es gibt also keine direkten Übergänge von $[a]$ auf $[b]$ oder umgekehrt, sondern nur von $[x]$ auf $[a, b]$ und von $[a, b]$ auf $[x]$.

Wir teilen die Anfragesequenz in disjunkte *Phasen* maximaler Länge auf. Die Zugriffe einer Phase enden dabei immer in der gleichen Kofiguration. Die ersten Kosten enstehen erst bei dem ersten Schreibzugriff der auf $[x]$ endet. O.B.d.A. nehmen wir deshalb an, dass die Sequenz mit einer $[x]$-Phase beginnt. Danach folgt eine $[a, b]$-Phase und diese wechseln sich dann ab.

Wir führen eine *Doppelphase* ein, die aus einer $[x]$-Phase und einer $[a, b]$-Phase besteht. Die Online-Sequenz besteht somit aus disjunkten aneinander gereihten Doppelphasen.

Eine Doppelphase hat Kosten 3. Am Anfang einer Doppelphase entstehen Servicekosten von 1 durch den Schreibzugriff auf x. Alle folgenden Zugriffe in der $[x]$-Phase erzeugen keine Kosten.

Erst der erste Zugriff in der $[a, b]$-Phase hat Servicekosten und Migrationskosten, also insgesamt Kosten von 2. Alle weiteren Anfragen in dieser Phase verursachen keine Kosten. Insegsamt sind das Kosten von 3 pro Doppelphase.

Jetzt zeigen wir, dass jeder andere Algorithmus Kosten von mindestens 1 in einer Doppelphase hat. Angenommen A sei ein Algorithmus mit Kosten 0 in einer Doppelphase. Dann muss die Konfiguration zu Beginn $[x]$ sein, da die Phasen mit einem Schreibzugriff beginnen und sonst schon Kosten entstehen würden. Weiterhin kann A diese Konfiguration $[x]$ nicht verlassen, da sonst Migrationskosten entstehen. Da aber der Beginn einer $[a, b]$-Phase von dem anderen Knoten initiiert wird, müssen doch Kosten entstehen. Also hat A mindestens Kosten 1 pro Doppelphase.

Damit ist gezeigt, dass unser Online-Algorithmus 3-competitive ist.

8.1.3 Untere Schranke

Wir zeigen nun, dass kein Algorithmus einen competitive-Faktor kleiner 3 hat.

Dazu zeigen wir für einen beliebigen Algorithmus A, dass es eine Eingabe und 3 andere Algorithmen gibt, die auf der Eingabe zusammen die selben Kosten wie A verursachen.

Diese Eingabe wird nur aus Schreibzugriffen bestehen. Auf solchen kann der Algorithmus die Kosten nicht verringern, indem er die Konfiguration $[a, b]$ nutzt. Also nehmen wir o.B.d.A. an, dass diese nicht verwendet wird.

Dann sei die Eingabe so, dass wenn der Algorithmus nach den ersten i Eingaben in Konfiguration $[x]$ ist, als nächstes Zugriff $w(y)$ mit $x \neq y$ kommt.

Auf der selben Eingabe betrachten wir diese anderen Algorithmen:

- B, der stets das Gegenteil von A macht. Das heißt, wenn A in Konfiguration $[x]$ übergeht, geht B in Konfiguration $[y]$ mit $y \neq x$ über.
- C_A, der stets in Konfiguration $[a]$ verbleibt.
- C_B, der stets in Konfiguration $[b]$ verbleibt.

Dann hat B genau dann Migrationskosten, wenn A Migrationskosten hat, aber auf dieser Eingabe nie Servicekosten. C_A und C_B haben nie Migrationskosten, aber in jedem Schritt hat genau einer von beiden Servicekosten.

Also sind die Gesamtkosten von A die Summe der Kosten dieser Algorithmen. Da der optimale Algorithmus nicht schlechter sein kann, kann A keinen competitive-Faktor kleiner 3 haben.

8.2 Paging-Problem

Wir betrachten ein zweischichtiges Speichersystem: Ein großer, langsamer *Hauptspeicher* und ein schneller *Cache* mit Platz für k Seiten. Dazu gibt es *Anfragen* t an den Speicher, die genau einer Seite entsprechen. Falls t nicht im Cache ist (*Seitenfehler*) so muss diese Seite *nachgeladen werden*. Falls der Cache voll ist, muss eine Seite *verdrängt* werden.

8.2.1 Deterministisch

Dazu gibt es verschieden *deterministische* Strategien:

- *LRU* (Least Recently Used): Verdrängte die Seite, deren letzter Zugriff am längsten zurückliegt.
- *LFU* (Least Frequently Used): Verdrängt die am seltensten nachgefragte Seite.
- *FIFO* (First In First Out): Verdrängt die Seite, die sich am längsten im Cache befindet.
- *LIFO* (Last In First Out): Verdrängt die Seite, die als letztes in den Cache geladen wurde.
- *RANDOM*: Verdrängt eine zufällig ausgewählte Seite.
- *FWF* (Flush When Full): Leert den Cache vollständig, sobald eine Seite verdrängt werden muss.
- *LFD* (Longest Forward Distance): Verdrängt die Seite, deren Zugriff am weitesten in der Zukunft liegt.

LFD ist das *optimale* Verfahren, aber kein Online-Algorithmus.

Um den competitive-Faktor eines Paging Verfahrens zu bestimmen, müssen wir Seitenauslagerungen erkennen. Dazu führen wir ein Marking ein.

Marking: Da jedes Verfahren bei $k+1$ verschiedenen Seitenzugriffen mindestens einen Fehler macht, teilen wir die Eingabesequenz in Phasen mit Zugriffen auf k verschiedene Seiten auf, d.h. die nächste Phase startet beim Zugriff auf die $k+1$-te neue Seite.

In einer Phase wird jede Seite beim erstmaligen Zugriff *markiert* und bleibt bis zum Ende der Phase markiert.

Damit können wir die Verfahren in zwei Klassen aufteilen: Verfahren, die nie eine markierte Seite verdrängen und solche, die auch Markierte verdrängen.

Definition: Marking-Algorithmus

Ein *Marking-Algorithmus* verdrängt niemals eine markierte Seite.

Eine *markierte Seite* ist eine Seite, auf die in der aktuellen Phase schon zugegriffen wurde.

In einer *Phase* wird auf k verschiedenen Seiten zugegriffen.

LRU und *FWF* sind Marking-Algorithmen.

Beweis (LRU): Betrachte eine beliebige Phase P_i, in der höchstens k Seiten markiert sind. Auf jeder der markierten Seiten ist in P_i zugegriffen worden, sodass *LRU* diese Seiten in dieser Phase nicht verdrängt.

Beweis (FWF): *FWF* entleert den Cache erst, wenn eine Seite verdrängt werden muss, also nach mehr als k verschiedenen Seitenzugriffen und genau am Ende einer Phase. Damit ist *FWF* Marking-Algorithmus.

Wir beweisen, dass jeder Marking-Algorithmus k-competitive ist: Zu jeder Zeit sind alle markierten Seiten im Cache.

- Jede neue Markierung verursacht höchstens einen Seitenfehler.
- Damit gibt es pro Phase höchstens k Seitenfehler.
- Außerdem muss jeder Paging-Algorithmus mindestens eine Seite pro Phase verdrängen.

Also sind Marking-Algorithmen k-competitive.
(Hier dein MR)

Damit sind auch *FWF* und *LRU* k-competitive.

Satz: FIFO

FIFO ist kein Marking-Algorithmus, aber k-competitive.

Satz: LFU

LFU ist nicht c-competitive für beliebiges $c \in \mathbb{R}$.

Satz: LIFO

LIFO ist nicht c-competitive für beliebiges $c \in \mathbb{R}$.

Beweis. Mit der Eingabesequenz $\rho = \rho_1^m, \rho_2^m, \ldots, \rho_{k-1}^m, (\rho_k, \rho_{k+1})^{m-1}$ werden nach $m \cdot (k-1)$ Zugriffen $2(m-1)$ viele Fehler erzeugt. Mittels m kann man den competitive-Faktor beliebig verschlechtern. □

8.2.2 Untere Schranke

Wir beweisen nun, dass deterministische Verfahren nur begrenzt gut sein können.

Satz: Untere Schranke

Es gibt keinen deterministischen Online-Algorithmus für das Paging-Problem mit competitive-Faktor besser als k.

Um dies zu beweisen, vergleichen wir einen Online-Algorithmus A mit einem optimalen Offline-Algorithmus B. Sei der Hauptspeicher beider Algorithmen $k+1$ Seiten und der Cache jeweils k Seiten groß.

Fülle nun beide Caches mit den gleichen k Seiten. Jetzt wird die Eingabesequenz so gewählt, dass A immer die Seite nachlädt, die nicht im Cache ist, sondern als einzige nur im Hauptspeicher liegt. B dagegen lagert diejenige Seite aus, die mindestens in den nächsten $k-1$ Schritten nicht angefragt wird.

Bevor beide den Seitenfehler machen, müssen sie die selben Seiten im Cache haben. Also kann man dies unbeschränkt fortführen. Dann macht A bei einer Sequenz mit Länge $k+m$ auch $k+m$ Seitenfehler, B macht aber bei $k+m$ Anfragen höchstens $k + \lceil \frac{m}{k} \rceil$ Seitenfehler.

8.2.3 Random

Da wir gesehen haben, dass deterministische Paging-Algorithmen höchstens k-competitive sind und der "Gegner" eine unvorteilhafte Sequenz wählen kann, schauen wir uns jetzt *randomisierte* Online-Algorithmen an.

Dabei wird der Zufallsanteil erst nach dem Wählen der Sequenz genutzt. Wir nennen den Algorithmus, oder Gegner, der die Eingabe erstellt, *oblivious* (blind, vergesslich).

Definition: c-competitive für Random

Ein randomisierter Online-Algorithmus A ist c-competitive gdw. ein a existiert, so dass für jede durch einen oblivous Gegner bestimmte Anfragesequenz ρ gilt:

$$\mathbb{E}[C_A(\rho)] \leq c \cdot C^*(\rho) + a.$$

Dazu gibt es einen einfachen randomisierten Online-Algorithmus namens *MARK*.

Algorithmus: MARK

Eingabe: Cache mit k vollen Seiten.

Ausgabe: Cache mit $k-1$ vollen Seiten.

Für jeden Seitenfehler pro Phase:

1. Bestimme unmarkierte Seiten
2. Verdränge eine uniform gewählte unmarkierte Seite

Satz: Schranke H_k-competitive

Es gibt einen anderen Paging-Algorithmus, der H_k-competitive ist, aber keinen der besser als H_k-competitive ist.

In der Vorlesung wurde außerdem bewiesen, dass $MARK$ $(2 \cdot H_k)$-competitive ist. Der Beweis wird hier aufgrund seiner Länge nicht erklärt.

8.2.4 Asymmetrischer Online-Algorithmus

Da der Offline-Algorithmus die gesamte Eingabe kennt, hat er einen unfairen Vorteil. Wir geben dem Online-Algorithmus daher mehr Cache.

Der Online-Algorithmus bekommt einen Cache der Größe $m \cdot k$, wenn der Offline-Algorithmus einen Cache der Größe k hat. Ein solcher Unterschied in den Einschränkungen an die Algorithmen nennt man *Asymmetrie*.

Definition: (m, c)-competitive

Ein Online-Algorithmus A für das Paging-Problem ist (m, c)-competitive gdw.

- m-mal so großer Cache wie Offline,
- es existiert ein a, sodass für jede Eingabesequenz ρ gilt $C_A(\rho) \leq c \cdot C^*(\rho) + a$.

Wir beweisen nun, dass $Random$ $(2, 2)$-competitive ist.

Dazu werden wir eine Potentialfunktion ϕ nutzen und zeigen, dass stets

$$\mathbb{E}[C] + 2\mathbb{E}[\phi] \leq 2C^*$$

ist, wobei $\mathbb{E}[C]$ die erwarteten Kosten von $Random$ und C^* die Kosten der optimalen Offline-Strategie OPT sind.

Wir definieren ϕ als die Anzahl der Seiten, die im Cache von OPT sind, aber nicht im Cache von $Random$. Dann kann ϕ nicht negativ sein, also reicht die obige Invariante, um den competitive-Faktor zu zeigen.

Die Invariante zeigen wir nun schrittweise (deswegen auch die Potentialfunktion). Wir betrachten den Seitenzugriff auf eine Seite ρ_t und unterscheiden vier Fälle:

1. Beide Strategien haben ρ_t bereits im Cache. Dann bleiben C, ϕ und C^* gleich.
2. Keine Strategie hat ρ_t schon im Cache. Dann erhöhen sich C und C^* beide um 1, $\mathbb{E}[\phi]$ erhöht sich höchstens um $\frac{1}{2}$, da die Wahrscheinlichkeit, dass $Random$ eine Seite verdrängt, die OPT im Cache hält, höchstens $\frac{1}{2}$ ist.
3. OPT hat die Seite im Cache, aber $Random$ nicht. Dann bleibt C^* gleich, C erhöht sich um 1 und $\mathbb{E}[\phi]$ verringert sich mindestens um $\frac{1}{2}$, da $Random$ sich um ρ_t an OPT angleicht und eine andere Seite, die OPT im Cache hat, mit Wahrscheinlichkeit höchstens $\frac{1}{2}$ verdrängt.
4. $Random$ hat die Seite im Cache, aber OPT nicht. Dann erhöht sich C^* um 1, C bleibt gleich und $\mathbb{E}[\phi]$ kann sich höchstens verringern.

Also wird die Invariante in allen Fällen beibehalten. Da sie zu Beginn gilt, muss sie also immer gelten.

9 Verständnisfragen

Flussprobleme

- Was ist die Laufzeit der Ford-Fulkerson Methode?
- Wann hat die Ford-Fulkerson Methode die maximale Laufzeit?
- Warum liefert die Ford-Fulkerson Methode den maximalen Fluss?
- Was ist die Laufzeit der Ford-Fulkerson Methode mit Breitensuche?
- Wie ist die Idee des Min-Cut-Max-Flow Theorems?
- Wie wird der Schnitt zu dem maximalen Fluss gefunden?
- Was ist die Idee des Algorithmus von Dinitz?
- Wie funktioniert die Forward-Propagation?
- Wie ist die Laufzeit vom Algorithmus von Dinitz?
- Wie ist die Begründung zur Laufzeit vom Algorithmus von Dinitz?
- Wie bestimmt man Flüsse mit einem Mindestfluss auf den Kanten?
- Wie bestimmt man Flüsse mit einem Mindestfluss auf den Kanten, auf denen aber auch ein leerer Fluss erlaubt ist?
- Wie ist die Idee der Algorithmen zu kostenminimalen Flüssen?
- Was ist die Laufzeit der Algorithmen zu kostenminimalen Flüssen?
- Was ist die Idee zum Beweis der Laufzeit der Algorithmen zu kostenminimalen Flüssen?

Matchings

- Wie kann man das bipartite Matching mit Flussalgorithmen lösen?
- Welche Laufzeiten haben die verschiedenen Matchingprobleme?
- Welcher Zusammenhang besteht zwischen verbessernden Pfaden und einem Maximum Matching?
- Wie ist die Vorgehensweise, um eine Laufzeit von $\mathcal{O}(m\sqrt{n})$ für das Matchingproblem zu erhalten?
- Wie ist die Vorgehensweise, um das Matching auf allgemeinen Graphen zu bestimmen?

Approximation

Wie können die folgenden Probleme approximiert werden? Welche untere Schranke ist bekannt?

- Cliquenproblem
- Färbungsproblem
- Vertex Cover
- Independent Set
- Steinerbaum Problem
- TSP
- Δ-TSP
- Zentrumsproblem
- Set Cover

Scheduling

- Wie arbeitet die LL Heuristik?
- Wie arbeitet die LPT Heuristik?
- Wie ist die Güte der LL Heuristik? Wie ist der Beweis?
- Wie ist die Güte der LPT Heuristik? Wie ist der Beweis?
- Wie arbeitet das Approximationsschema für Makespan Scheduling? Wie ist die Beweisidee?
- Wie kann des Makespan Problem auf allgemeinen Maschinen approximiert werden? Wie ist die Beweisidee?

Randomisierte Algorithmen

- Was ist ein Monte-Carlo-Algorithmus?
- Was ist ein Las-Vegas-Algorithmus?
- Wie können sich Monte-Carlo- und Las-Vegas-Algorithmen gegenseitig simulieren?
- Wie arbeitet der randomisierte Algorithmus für einen minimalen Schnitt?
- Wie ist die Laufzeit des randomisierten Algorithmus für einen minimalen Schnitt?
- Wie funktioniert der Algorithmus von Schöning?
- Wie ist die Fehlerwahrscheinlichkeit beim Algorithmus von Schöning?

Lineare Programmierung

- Wie ist die Idee der Simplexmethode?
- Welche Laufzeit erhalten wir bei der Simplexmethode?
- Wie ist die Idee der Ellipsoidmethode?
- Welche Laufzeit erhalten wir bei der Ellipsoidmethode?
- Wie arbeitet die Perturbierung?
- Wie arbeitet der Algorithmus von Seidel?
- Wie ist die Laufzeit für den Algorithmus von Seidel?

Online-Algorithmen

- Wie arbeitet der Online-Algorithmus für das File Allocation-Problem?
- Wie ist die untere Schranke für jeden deterministischen Online-Algorithmus für das FAP?
- Welche Paging-Algorithmen gibt es?
- Was ist ein Marking-Algorithmus?
- Was ist der competitive Faktor von Marking Algorithmen?
- Was ist der competitive Faktor vom *LFU* Algorithmus?
- Wie gut können deterministische Paging Algorithmen sein?
- Wie gut sind randomisierte Paging Algorithmen?
- Wie gut können asymmetrische Paging Algorithmen sein?

10 Wir wollen eure Merge Requests!

Trotz der 41 Seiten, die wir schon haben, gibt es noch viel zu tun. Falls ihr euch damit quälen und gleichzeitig Anderen helfen wollt, sehen wir gerne Merge Requests zu beliebigen Themen, insbesondere aber zu den folgenden Stellen:

- Mean Algorithmus
  Von den beiden Laufzeitschranken wurde bisher nur eine erklärt.
- Maximum Matching
  Der Beweis, dass der Greedy-Algorithmus eine 2-Approximation für Maximum Matching liefert, war mal in einer Übung. Den könnte man hier nochmal umreißen.
- Set Cover
  Der Beweis des Approximationsfaktors fehlt.
- Scheduling: Allgemeine Maschinen
  Leicht unvollständig.
- Clique
  Nichtapproximierbarkeit könnte man noch besser erklären.
- LP: Geometrische Lösung
  Könnte man hier eigentlich auch noch erklären.
- LP: Simplex
  Die Laufzeitbeweise.
- LP: Die Box-Bedingungen bei Seidel
  Das könnte man noch besser erklären.
- Fastcut
  Der Zuverlässigkeitsbeweis fehlt noch.
- Minimum Weight Perfect Matching
  Wurde in der Übung Äquivalenz zu Max-Weight Matching gezeigt. Da das Problem beim Algorithmus von Christofides wieder auftaucht, könnte man das hier zumindest definieren.
- Marking Algorithms
  Man könnte den competitive-Faktor noch besser begründen.
- Fastcut: Erfolgswahrscheinlichkeit
  Die Ideen für die Erfolgswahrscheinlichkeit könnte man noch erklären.