Lineare und Kombinatorische Optimierung Zusammenfassung des Wichtigsten

Wintersemester 2015/2016

Prof. Dr. Alexander Martin

Autor:

- Christian Strate

Inhaltsverzeichnis

0 Organisatorisches

Note:

Bei dieser Zusammenfassung handelt es sich um eine inoffizielle Mitschrift von Studenten. Entsprechend sind weder Garantie auf Vollständigkeit noch auf Korrektheit gewährleistet.

Prüfungshinweise:

In der Prüfung wurden explizit die Algorithmen der Vorlesung in Pseudocode verlangt, einschließlich der Beweise weshalb diese korrekt sind. Darüber hinaus waren zu bestimmten Begrifflichkeiten die formalen Formeln gefragt, die ich in der Zusammenfassung häufig unterschlug. Solltet ihr diese also nicht sicher beherrschen, so lege ich euch nahe diese im offiziellen Skript nachzuschlagen.

1 Einleitung

Optimierungsproblem umdrehen:
$\max\{f(x) : x \in \chi\} = -\min\{-f(x) : x \in \chi\}$

Lineares Optimierungsproblem:
Zielfunktion und Nebenfunktionen sind linear vom $\mathbb{R}^n \mapsto \mathbb{R}$

Kontinuierliches Optimierungsproblem:
Problem ohne Integerbedingungen

Konvexe Optimierungsprobleme:
Jedes lokale Optimum entspricht einem globalen. Es Genügt Gradienten-Betrachtung

Globale Optimierungsprobleme:
Gradienten allein genügen nicht mehr.

2 Endliche Mengen

Symmetrische Differenz:

Die Menge $A \Delta B = (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$

Permutation mit Reihenfolge:

Permutation einer geordneten k-elementigen Teilmenge, einer Menge mit n Elementen

$$\frac{n!}{(n-k)!}$$

Permutation ohne Reihenfolge:

k-elementige Teilmenge einer n-elementigen Menge

$$\binom{n}{k} = \frac{n!}{(n-k)!k!} = \binom{n}{n-k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n \qquad \sum_{k=0}^{n} (-1)^k \binom{n}{k} = 0$$

- Pascalsche Dreieck
- Aussagenlogik

3 Kombinatorische Werkzeuge

Geometrische Summe:

$$\sum_{i=0}^{n} q^i = \frac{1-q^{n+1}}{1-q}, \forall q \neq 1$$

Bernoulli-Ungleichung:

$$h \in \mathbb{R}, h \geq 1, n \in \mathbb{N}.\ (1+h)^n \geq 1 + n \cdot h$$

- Inklusions-Exklusionsprinzip
- Taubenschlagprinzip

Binomialsatz:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} \cdot x^{n-k} \cdot y^k, \forall x, y \in \mathbb{R}, n \in \mathbb{N}$$

4 Graphen-Grundlagen

Inzidenzfunktion Ψ eines Graphen:

$\Psi : E \to V \times V$
Also wird jeder Kante e ein Knotenpaar u, v durch $\Psi(e) = uv$ zugewiesen.
inzident: keine Steigerung hervorrufend

Endlichkeit eines Graphen:

Endlich, falls $|V|, |E| < \infty$, sonst unendlich

induzierter Schnitt $\delta(W)$:

$\delta(U, W)$ Kantenmenge mit Endknoten in U und W.
$\delta(W, V \setminus W) = \delta(W)$, wobei $\delta(W)$ der von W induzierter Schnitt
gerichteter Fall:
$\delta^+(W) = \{(i, j) \in A : i \in W, j \in V \setminus W\}$ Ausgehende Bogenmenge
$\delta^-(W) = \{(i, j) \in A : i \in V \setminus W, j \in W\}$ Eingehende Bogenmenge

Vollständiger Graph:

Jeder Knoten ist mit jedem anderen verbunden. K_n ist ein n-kardinaler vollständiger Graph

bipartiter Graph:

Knotenmenge lässt sich in zwei disjunkte, nicht leere Teilmengen einteilen, so dass zwei Knoten der jeweiligen Teilmengen mit keinem anderen der selben Teilmenge benachbart sind.

vollständig bipartiter Graph:

$uv \in E \forall u \in V_1, v \in V_2$. Bezeichnet als $K_{m,n}, m = |V_1|, n = |V_2|$

Kette:

Endliche Folge K von Knoten und Kanten (beginnend und endend mit Knoten), deren Glieder aufeinanderfolgend miteinander verbunden sind, also die Kanten inzidieren mit Vorgänger- und Nachfolger-Knoten.

Weg:

Eine Kette in der sämtliche Knoten und Kanten voneinander verschieden sind.

Pfad:
Eine Kette, in der alle Kanten verschieden sind.

Eulerpfad:
Ein Pfad, der jede Kante genau einmal enthält.

Eulertour:
Ein Eulerpfad, bei dem der Pfad geschlossen ist. Der Graph ist eulersch. (Jeder Knoten hat geraden Grad). Problem ist leicht lösbar

Hamiltonweg:
Ein Pfad, der jeden Knoten genau einmal enthält.

Hamiltontour, Hamiltonkreis:
Ein Hamiltonweg mit geschlossenem Pfad. Der Graph ist hamiltonsch. Problem ist NP-Vollständig

Planarer Graph:
Lässt sich in der Ebene so zeichnen, dass sich keine zwei Kanten kreuzen.

Flächenkreis:
Ein Kreis ($C_F \subseteq E$) der eine Fläche umschließt, die nicht die äußere Fläche eines Graphen ist.

dualer Graph:
In jeder Fläche wird ein Knoten angelegt. Diejenigen Knoten benachbarter Flächen werden mit einer Kante verbunden. (deren Flächenkreise besitzen gemeinsame Kanten). Beispiel vgl fig. 1

Landau-Notation:

- $\mathcal{O}(g) = \{f \in M | \exists c, n_0 \in \mathbb{N} : f(n) \leq c \cdot g(n), \forall n \geq n_0\}$
- $\Omega(g) = \{f \in M | \exists c, n_0 \in \mathbb{N} : f(n) \geq c \cdot g(n), \forall n \geq n_0\}$
- $\Theta(g) = \mathcal{O}(g) \otimes \Omega(g)$
- Speicherung: Kanten-Bogenliste, Adjazenzmatrix und -listen werden auf 34f angesprochen.

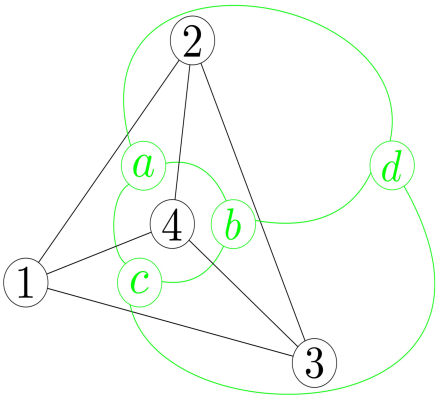

Abbildung 1: Dualer Graph

- Suchalgorithmen

5 Sortieren

Es gilt stets n - Knoten, m - Kanten

- Selection Sort - Sortieren durch Auswahl
  $\mathcal{O}(n^2)$, stabil, in-place
  Swappt die schwersten Elemente der Reihe nach vorne.

- Bubble Sort
  $\mathcal{O}(n^2)$, stabil, in-place
  Tauscht jeweils benachbarte Einträge, wenn Sortiertheit verletzt. In jeder Iteration wird das aktuell noch nicht sortierte schwerste Element an die richtige Stelle geblubbert.

- Quick Sort
  $\mathcal{O}(n \log n)$, worst case: $\mathcal{O}(n^2)$
  Pivotelement
  in-place möglich wenn je ein nach links und ein nach rechts zu verschiebendes Element geswappt werden

- Heap Sort
  Eigenschaft: Vaterelement $>$ beide Kinder (indices $2n+1$, $2n+2$)
  $\mathcal{O}(n \log n)$ (auch im Worst-Case), lässt sich mit Listen in place sortieren

- Bucket Sort
  $\mathcal{O}(n+m)$ (unter Einschränkungen), stabil
  Laufzeit: Wenn gemäß der Sortierwertigkeit für jedes Element ein Eimer existiert, bei einer großen Anzahl unterschiedlich möglicher Wertigkeiten muss die Eimer-Anzahl reduziert und die Werte innerhalb der Eimer erneut sortiert werden, was zu schlechteren oberen Schranken führt.

6 Bäume und Wälder

Komponenten eines Graphen:

Sind bezüglich Kanteninklusion maximal zusammenhängende Teilgraphen.

Komponentenanzahl und Artikulationsknoten v:

$\zeta(G)$ Komponentenanzahl
v heißt Artikulationsknoten von G, falls $\zeta(G \setminus \{v\}) > \zeta(G)$
Beispiel in fig. 2

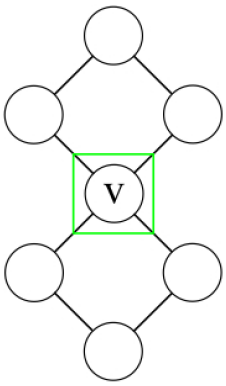

Abbildung 2: Artikulationsknoten Beispiel

Wald:

Kantenmenge eines Graphen, die keinen Kreis enthält

Baum:

Ein zusammenhängender Wald

Algorithmus von Kruskal - findet Baum minimalen Gewichts, maximalen Wald:

Kantengewichte sortieren: $c_{e_1} \leq \cdots \leq c_{e_m}$
Iteriere von i bis m und wenn der nächste Knoten keinen Kreis erzeugt, dann füge ihn hinzu (greedy). Das "keinen Kreis erzeugen" ist hier nicht ganz trivial und muss tatsächlich überprüft werden, bei Prim ist dies hingegen simpel mithilfe einer TODO-Liste realisierbar.
Laufzeit $\mathcal{O}(m \cdot n)$, m Kanten, n Knoten.
Das Ergebnis ist identisch mit einem minimal aufspannenden Baum, sofern der Graph zusammenhängend ist.

Algorithmus von Prim - findet minimal aufspannenden Baum - Laufzeit $\mathcal{O}(n^2)$:

Laufzeit ist best mögliche für vollständige Graphen
Eingabe ist gewichteter aufspannender Graph
Das "keinen Kreis erzeugen" ist trivial mit einer TODO-Liste realisierbar. Ist ein Knoten teil der TODO-Liste, kann die Kante zu diesem Knoten keinen Zyklus erzeugen.

Wähle beliebigen Startknoten
TODO-Liste: Alle verbleibenden Knoten
Ergebnis-Liste: initial leer
Betrachte alle ausgehenden Kanten der bisherigen Ergebnis-Knotenmenge, die einen noch zu erreichenden, neuen Knoten verbinden, und wähle die Kante mit geringstem Gewicht. Füge sie der Ergebnismenge hinzu und auch den neuen Knoten. Entferne den neuen Knoten entsprechend aus der TODO-Liste

7 Kürzeste Wege

Es gilt stets n - Knoten, m - Kanten

topologische Sortierung f von V - Detektieren von Kreisen:

Topologische Sortierung existiert genau dann wenn der Graph azyklisch ist.

$f : V \to \{1, \ldots, |V|\}$ ist topologische Sortierung von V, falls
$\forall (u, v) \in A.\ f(u) < f(v)$
Dies ist genau dann Möglich, wenn (V, A) azyklisch ist.
Die Sortierung erfolgt über die ein- bzw. ausgehenden Kanten. $deg(u) = |\delta^-(u)|$, gestartet wird bei den Knoten mit $deg(u) = 0$, also diejenigen die über keine eingehenden Kanten verfügen. Für jede ausgehende Kante (u, v) von u wird $deg(v)$ um eins verringert. Ist $deg(v) = 0$, so wird dieser Vorgang auch für v durchgeführt. Gibt es keine Knoten mehr mit $deg(x) = 0$ und ist dies noch nicht für n verschiedene Knoten durchgeführt worden, so enthält der Graph einen Kreis, andernfalls ist die Besuchsreihenfolge die topologische Sortierung.

Algorithmus - Topologische Sortierung - Laufzeit $\mathcal{O}(m)$:

Entfernt solange Knoten ohne Vorgänger, bis es keine Knoten mehr gibt oder kein Knoten mehr eine Quelle ist. Ist der Endgraph leer, so existiert eine Sortierung.

Bestimme eingehende Kantenanzahl aller Knoten und füge die Quellen zu L hinzu
Für alle Knoten in L, entferne einen Knoten und untersuche die ausgehenden Kanten und überprüfe ob die Zielknoten eine Quelle sind, wenn ihre Eingangs-Kantenanzahl um eins reduziert wird. Ist dies eine Quelle, so füge sie der Menge L hinzu

Algorithmus von Moore und Bellmann für azyklische Diagraphen $D = (V, A)$ - Laufzeit $\mathcal{O}(m)$:

Bestimmt einen kürzesten gerichteten Weg von s nach v, für alle $v \in V$.

Gemäß ihrer topologie sortierte Kantengewichte, die auch negativ sein dürfen.
Wähle Startknoten und setze Distanz zu sich selbst auf 0 und für alle anderen Knoten auf ∞
für alle Ausgehenden Kanten betrachte Distanz. Ist bisherige Distanz zu dem Knoten größer als die neue, mögliche, dann ersetze die alte-

Dieser Algorithmus funktioniert im Grunde wie Dijkstra, aktualisiert allerdings Knoten auch dann, wenn sie bereits als Source-Knoten verwendet wurden und aktualisiert entsprechende Nachfolger ebenso. Der Unterschied zu Dijkstra ist, dass die Knoten gemäß ihrer topologischen Sortiertheit besucht werden.

Dijkstra - Laufzeit $\mathcal{O}(n^2)$:

Funktioniert nicht für alle Graphen mit negativen Kantengewichten

Startknoten und Kosten wie oben: $d(v) = \begin{cases} 0 & , v = s \\ \infty & , v \neq s \end{cases}$

Solange nicht alle Knoten markiert:

markiere denjenigen unmarkierten Knoten mit der niedrigsten Distanz

Für alle unmarkierten Knoten ersetze die alte Distanz durch eine neue, mögliche, falls die neuere kürzer ist

Algorithmus - Grundversion von Moore-Bellmann - Laufzeit $\mathcal{O}(n^2 \cdot C), C = \max_{a \in A} |c_a|$:

Bestimmt den kürzesten Weg von s nach v, für alle $v \in V$. Dieser Algorithmus kann mit negativen Kantengewichten und nicht-negativen Kreisen umgehen. Es wird aber angenommen, dass man nicht mit negativen Kosten im Kreis laufen kann.

Initialisierung Startknoten und Kosten wie oben: $d(v) = \begin{cases} 0 & , v = s \\ \infty & , v \neq s \end{cases}$

Solange Bögen mit verringerbaren Kosten existieren, erniedrige sie

Algorithmus - Bellmann - Laufzeit $\mathcal{O}(nm)$:

Eingabe darf kein negativen Kreis enthalten, andere Zyklen sind aber erlaubt.

Startknoten und Kosten wie oben: $d(v) = \begin{cases} 0 & , v = s \\ \infty & , v \neq s \end{cases}$

Laufe für jeden Knoten einmal über ALLE Kanten des gesamten Graphen und wenn Kosten verbesserbar, do it - dabei merke dir jeweils den Vorgänger, sofern eine Verbesserung durch diesen verursacht wird.

Erkennen negativer Kreise - Laufzeit $\mathcal{O}(nC)$:

Prüfe für alle Distanzlabel, ob $d(j) < -nC$, falls für irgend einen Knoten ja, dann ist ein negativer vorhanden. Diese Überprüfung verursacht den Verlust der polynomialen Laufzeit. Besser ist das Folgende

Ergänzung der Algorithmen mit einer negativen Kreis-Erkennung:

Ist besser als das darüber, weil hierdurch die polynomiale Laufzeit nicht verloren geht. Nach jeder Iteration wird überprüft, ob ein negativer Kreis gefunden wurde (DFS) in $\mathcal{O}(n)$, das verschlechtert die Laufzeit der Algorithmen auf $\mathcal{O}(n^2 m)$, wird die Überprüfung hingegen nur alle αn Iterationen durchgeführt, so beschränkt sich die Laufzeit auf $\mathcal{O}(nm)$

Algorithmus Floyd - kürzeste Wege/Kreise aller Knotenpaare - Laufzeit $\mathcal{O}(n^3)$:

for i = 1 : n do
  for j = 1 : n do
    $w_{ij} = c_{ij}$, bzw. ∞ und $p_{ij} = i$, bzw. 0 - je nachdem ob $(i, j) \in A$ oder nicht

for l = 1 : n do
  for i = 1 : n do
    for j = 1 : n do
      falls neue Kosten geringer ($w_{ij} > w_{il} + w_{lj}$)
        ersetzte altes w_{ij}, p_{ij}
        falls $i = j$ und $w_{ii} < 0$ fertig

Ergebnis: w_{ij}: kürzester (i, j)-Weg
w_{ii}: kürzester (i, i)-Kreis
p_{ij}: Vorgänger eines kürzesten (i, j)-Weges
p_{ij}: Vorgänger eines kürzesten (i, i)-Kreises

Arboreszenz:

Enthält eine Wurzel, von der aus jeder Knoten auf genau einem gerichteten Weg erreicht werden kann.

8 Maximale Flüsse

Schnitt S:

Eine Echte Kanten-Teilmenge, die die Quelle (s), aber nicht die Senke (t) enthält. Gibt eine obere Schranke für den Wert der (s,t)-Flüsse an. Ein Schnitt, also eine Kantenmenge, unterteilt einen Graphen in zwei Knoten-Mengen bzw. Komponenten.

Kapazität eines Schnittes:

Die Summer der Kapazitäten aller von S nach $V \setminus S$ verlaufenden Kanten. Also derer Kanten, die die beiden Mengen vereinen.

Minimaler Schnitt:

Würde das entfernen der Kanten des Schnittes (Teilmenge des Graphen) dazu führen, dass der Graph in zwei Komponenten zerfiele, so handelt es sich bei dem Schnitt um einen minimalen. Weiterhin scheint die Kapazität relevant zu sein. Minimal ist dieser Fluss dessen Fluss-Kapazität geringer ist.

Fluss:

Eine Funktion f, die jeder Kante des Netzwerks einen nichtnegativen Flusswert zuordnet. Dabei darf der Flusswert einer Kante dessen Kapazität nicht übersteigen, d.h. $\forall e \in E : f(e) \leq u(e)$

zulässiger Fluss (s,t):

Funktion $x : A \to \mathbb{R}$ heißt zulässiger (s,t)-Fluss, wenn
$0 \leq x_a \leq c_a, \forall a \in A$, wobei c_a die Bogenkapazitäten sind. Für jeden Knoten $x \neq s,t$ muss $\delta^-(x) = \delta^+(x)$

Der Fluss wird stets mit f bezeichnet. Wert des zulässigen Flusses: $VAL(x) = \sum_{a \in \delta^+(s)} x_a - \sum_{a \in \delta^-(s)} x_a$

Der Wert eines $s-t$ Flusses gibt also den Überschuss im Knoten t an.

Sei die Bogenkapazität positiv für alle $a \in A$ und $s, t \in V$ verschieden und $\mathcal{P}$ die Menge aller gerichteten (s,t)-Wege des Digraphen $D = (V, A)$ und $\mathcal{C}$ dessen Menge der gerichteten Kreise
Dann ist x ein zulässiger (s,t)-Fluss, genau dann wenn, es Wege und Kreise mit positiven Gewichten gibt mit

$$x_{uv} = \sum_{\{i|uv \in P_i\}} \lambda_i + \sum_{\{j|uv \in C_j\}} \mu_j \leq c_{uv}.$$

Hierbei ist $\gamma := \min\{c_f(e)|e \in E_W\}$, des Weges W

Maximal-Fluss-Problem - max flow problem:
Problem einen zulässigen Fluss maximalen Wertes zu finden.

augmentierender $[s,v]$-Weg:
Vorwärtsbogen: von s nach v, Rückwärtsbogen: Gegenteil
augmentierender $[s,v]$-Weg bzgl. x: für jeden Vorwärtsbogen: $x_{ij} < c_{ij}$ und für jeden Rückwärtsbogen: $x_{ij} > 0$

maximaler (s,t)-Fluss:
Die Maximalität eines $s-t$ Flusses wird genau dann erreicht, wenn es keinen augmentierenden $[s,t]$-Weg bzgl. x gibt. Der maximale Wert ist gleich der minimalen Kapazität eines (s,t)-Schnittes.

Residualgraph:
Besteht aus den vom Fluss nicht ausgelasteten Kanten ergänzt um die Rückkanten.
Ein ziemlich gutes Beispiel findet sich in den Notizen "Vorlesung_Notizen_PreflowAlg" auf Seite 7.

Negative Kreise und Optimalität eines Flusses x für das Min-Cost-Flow-Problem:

x ist genau dann optimal, wenn $G(x)$ keinen negativen Kreis bezüglich folgender Kostenfunktion enthält

$$\overline{w}_{ij} = \begin{cases} w_{ij} & , ij \in A_1(x) \\ -w_{ji} & , ij \in A_2(x) \end{cases}$$

Wobei eine Kante Teil der Menge A_1 ist, wenn es sich um eine Hinkante mit Fluss kleiner der Kapazität und Teil der Menge A_2 ist, wenn es sich um eine Rückkante mit Fluss größer 0 handelt. Die Kosten einen bestimmten Fluss entlang einer Kante (u, v) zu senden ist $f_{u,v} \cdot w_{u,v}$, oder entsprechend verringerte Werte für f, falls nicht der maximal mögliche Fluss zu senden ist (in diesem Skript häufig mit $x_{u,v}$ bezeichnet. Ziel ist es nun beim Versenden der erforderlichen Menge d die Kosten des Flusses über alle Knoten zu minimieren: $\sum_{(u,v)\in E} w_{u,v} \cdot f_{u,v}$, wobei

- $f(u, v) \leq c(u, v)$
- $f(u, v) = -f(v, u)$
- $\sum_{y\in V} f(u, y) = 0, \forall u \neq s, t$
- $\sum_{y\in V} f(s, y) = d, \quad \sum_{y\in V} f(y, t) = d$

Die Lösung lässt sich mittels linear programming/linear optimisation bestimmen.

Residualkapazität:

Gibt für Hinkanten an, um wieviel der Fluss noch erhöht werden kann und für Rückkanten um wieviel der Fluss der zugehörigen Hinkante noch verringert werden kann.

$$\tilde{c}_{ij} = r_{ij} = \begin{cases} c_{ij} - x_{ij} & , ij \in A_1(x) \text{(Hinkante)} \\ x_{ji} & , ij \in A_2(x) \text{(Rückkante)} \end{cases}$$

wobei $G(x) = (V, A(x))$

$ij \in A(x) \Leftrightarrow \underbrace{ij \in A \text{ und } x_{ij} < c_{ij} \text{ oder}}_{=:A_1(x)} \underbrace{ji \in A \text{ und } x_{ji} > 0}_{=:A_2(x)}$

Augmentierender-Wege-Algorithmus nach Ford & Fulkerson - Laufzeit $\mathcal{O}(mv)$:

v ist Wert maximaler Fluss, falls c ganzzahlig.

positive Bogenkapazität. Ergebnis ist ein zulässiger (s, t)-Fluss x mit max. Wert $VAL(x)$

bestimme zulässigen Fluss x
solange ein augmentierender Weg von s nach t existiert:
bestimme für jede Kante in diesem Weg die Residualkapazität (s. nächstes Kapitel):

$$\varepsilon = \min_{i,j \in P} \begin{cases} c_{ij} - x_{ij}, & (i,j) \text{ Vorwärtsbogen} \\ x_{ij}, & (i,j) \text{ Rückwärtsbogen} \end{cases}$$

Für jede Kante des Weges modifiziere die Residualkapazität:
$x_{ij} += \varepsilon$ (Vorwärts)
$x_{ij} -= \varepsilon$ (Rückwärts)

Beispiel

Es wird also für jeden Weg bestimmt welche Kante hat den geringsten Fluss, dieser Fluss wird auf die restlichen Kanten angewendet. Anschließend wird Kapazität(Kante) - Fluss(Kante) berechnet und als neue Kapazitäten betrachtet. Für alle Kanten die einen positiven Fluss hatten wird eine Rückkante mit diesem Fluss eingeführt.

Anschließend werden die alten berechneten Fluss-Werte mit den neuen verrechnet (einige Kantenrichtungen haben sich verändert).

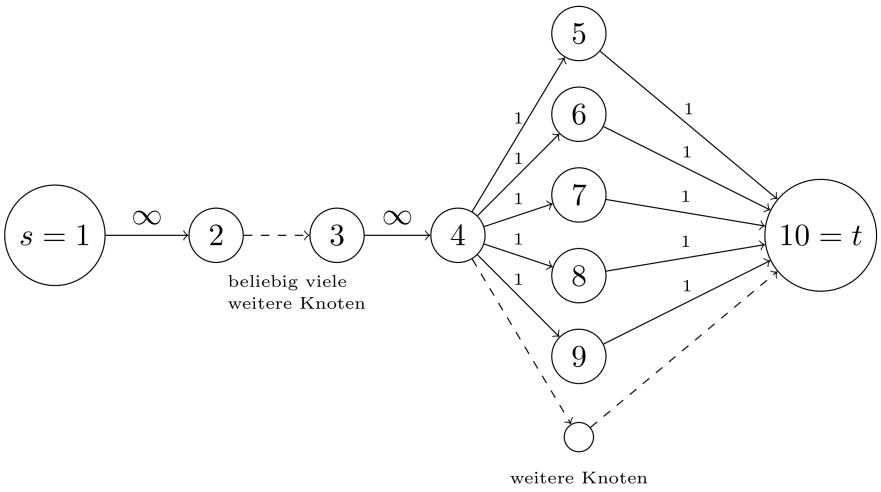

Abbildung 3: Beispielgraph für schlechtes Laufzeitverhalten von Augmentierenden-Wege-Algorithmen

Aug-W-A. werden immer entlang von $[s,t]$-Wegen geschickt. Die neue Idee ist, unter Verletzung der Flusserhaltungsbedingung, soviel wie möglich durch das Netz zu schieben (push). Dadurch entstehen Knoten mit Überschuss, der durch Schieben von Fluss, Richtung Senke, abgebaut werden muss.

Das maximal schiebbare von (i,j) ist die Residualkapazität $\tilde{c}_{ij} = c_{ij} - x_{ij} + x_{ji}$ (s. nächstes kapitel)

Aktiver Knoten:

Ein von s, t verschiedener Knoten v des Graphen, auf dem mehr Fluss ankommt als abfließt, also $\delta^- > \delta^+$

Pseudofluss x auf $D = (V, A)$:

Abbildung $x : A \to \mathbb{R}$ mit
$0 \leq x_a \leq c_a, \forall a \in A$ und
$e_x(v) = x(\delta^-(v)) - x(\delta^+(v)) \geq 0, \forall v \in V \setminus \{s, t\}$
Ein Pseudofluss ist genau dann ein Fluss, wenn er keine aktiven Knoten besitzt.

gültige Markierung d bzgl. des Pseudoflusses x:

Funktion $d : V \to (\mathbb{Z}_+ \cup \infty)$, falls
$d(s) = n, d(t) = 0$ und
$d(v) \leq d(w) + 1, \forall vw \in A(x)$
x ist maximal, sofern er über eine gültige Markierung verfügt.

Diff Augmentierende-Wege-Algorithmen und Pseudofluss:

Pseudofluss erhält gültige Markierung und damit saturierten Schnitt und terminiert wenn x zulässig ist.
Aug-W-A behalten zulässigen Fluss und terminieren wenn ein Schnitt saturiert wird.

Push-Relabel-Algorithmus, Goldberg &Tarjan - maximaler (s, t)-Fluss - Laufzeit $\mathcal{O}(n^2 m)$:

Der Fluss wird solange modifiziert bis er ein (s, t)-Fluss ist. Wird ein solcher erzeugt, so ist dieser auch sicher maximal.
Relabel: v aktiver Knoten, hat also Überschuss, es gibt aber keinen zulässigen Bogen, der v verlässt. Also setzen wir, ohne Gültigkeit der Markierung zu verletzen (relabel) $d(v) = 1 + \min\{d(w) : (v, w) \in A(x)\}$. Dieses Relabel erfolgt also immer dann, wenn ein Knoten v aktiv ist, aber kein Bogen existiert mit $d(v) = d(w) + 1$ (zulässiger Bogen)
maximal $2nm$ saturierte Schübe, maximal $\mathcal{O}(n^2)$ Relabels, maximal $\mathcal{O}(mn^2)$ nichtsaturierte Schübe

Initialisierung: Für alle Kanten $e \in \delta^+(s) : f(e) := u(e)$ für alle anderen $f(e) := 0$
Setze $d(s) := |V(G)|$ und für alle anderen $d(v) := 0$
solange aktive(r) Knoten v existiert(en), wähle ein solches v und:
gibt es einen zulässigen Bogen (v, w), dann
Schiebe $\delta := \min\{e(v), c_{vw} - x_{vw} + x_{wv}\}$ von v nach w (push) - augmentiere entlang einer ausgehenden Kante
sonst
$d(v) := 1 + \min\{d(w)|vw \in A(x)\}$ (relabel)
Durch die Distanzmarken wird garantiert, dass t niemals direkt von s erreichbar ist und der (s,t)-Fluss tatsächlich maximal ist. Beispiel in fig. 4 und etwas ausführlicher in den Notizen "Vorlesung_Notizen_PreflowAlg" auf Seite 7.

Es werden also die aktiven Knoten betrachtet und immer dann wenn im Residualgraphen eine Rückkante (durch push) erzeugt wird, so wird der Fluss (separater Graph) für die entsprechende Hinkante um eben den Wert der Rückkante erhöht. Wichtig dabei ist stets unterhalb der Kantenkapazitäten zu bleiben.
d ist stets das Potential (Markierung) und der Überschuss die Aktivität, die sich aus dem Flussgraphen bestimmen lässt.

Anderes Beispiel mit mehr Sonderfällen, in denen Fluss nicht vollständig durch den Residualgraphen gepusht werden kann, sondern wieder zurückgeschoben werden muss. Wichtig hierbei ist, dass der Startknoten mit $d =<$ Knotenanzahl $>$ initialisiert ist, weshalb beim zurückschieben zum Startknoten auch $d =<$ Knotenanzahl $>$ $+1$ für den folgenden Knoten entstehen wird. Der Maximale Flussgraph lässt sich aus dem Residualgraphen und dem initialen Kapazitätsgraphen ableiten. Der Fluss ist jeweils $\frac{\text{Rückkantenkapazität}}{\text{Kapazität}} = \frac{\text{Rückkantenkapazität}}{\text{Rückkantenkapazität+Hinkantenkapazität}}$

Schub heißt saturiert entlang (v, w), falls:
$\tilde{c}_{vw} \leq e_x(v)$, d.h. $\tilde{c}_{vw} = c_{vw} - x_{vw} + x_{wv}$ viele Einheiten werden geschoben und der Bogen (v, w) wird aus G entfernt. Andernfalls heißt der Schub nicht-saturiert. Also ist ein Schub saturiert, wenn über die entsprechende Kante (beide Richtungen aufsummiert) die maximale Kapazität fließt.

Tabelle 1: Laufzeiten alternative Implementierungen $C = \max_{a \in A} |c_a|$

Algorithmus	Aufwand
Augmentierende Wege (Grundversion)	$\mathcal{O}(mv) = \mathcal{O}(mnC)$ nicht polynomial
Kürzeste augmentierende Wege (Edmond & Karp)	$\mathcal{O}(nm^2)$
Kürzeste augmentierende Wege unter Ausnutzung, dass $d(\cdot)$ monoton steigend	$\mathcal{O}(n^2m)$
Kapazitätsskalierungs-Algorithmen	$\mathcal{O}(nm \log C)$
Preflow-Push-Algorithmen (Grundversion)	$\mathcal{O}(n^2m)$
FIFO-Preflow-Push-Algorithmen	$\mathcal{O}(n^3)$
Highest-Label-Preflow-Push-Algorithmus	$\mathcal{O}(n^2\sqrt{m})$
Excess-Scaling-Algorithmus	$\mathcal{O}(nm + n^2 \log C)$

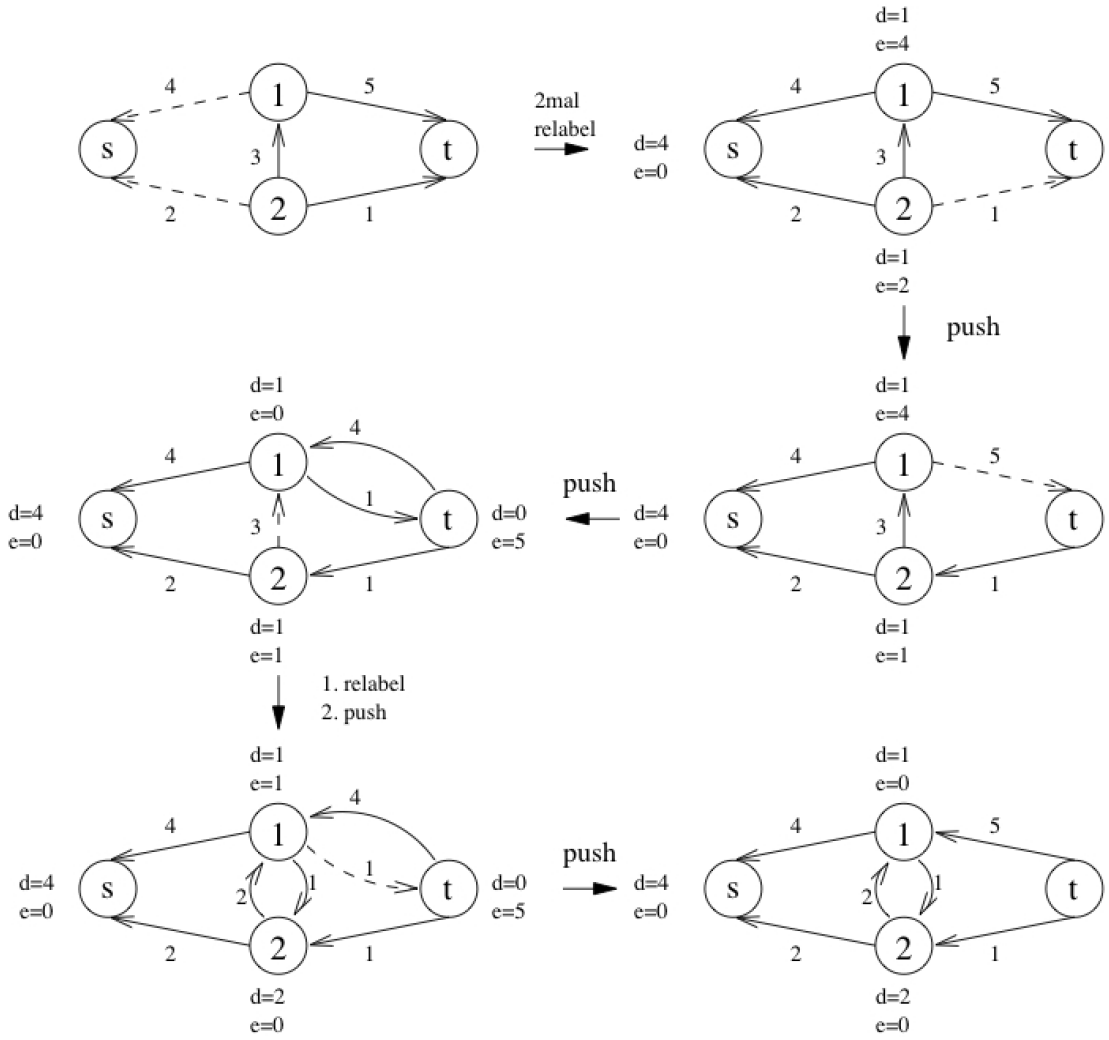

Abbildung 4: Beispiel Push-Relabel-Algorithmus

9 Minimalkosten-Fluss-Probleme

Bogen A:

Wann immer die Menge A verwendet wird, sind Bögen gemeint, also im Grunde Kanten. Hier wird wohl nicht wie gewohnt e verwendet, da dies bereits für die Unausgewogenheitsfunktion reserviert ist.

Minimalkosten-Fluss-Problem - Min-Cost-Flow-Problem eines Digraphen $D = (V, A)$:

Bogenkapazität c, Kosten w, Bedarfsfunktion $b : V \to \mathbb{R} : \sum_{v \in V} b(v) = 0$

$$\begin{aligned} \min \sum_{a \in A} w_a x_a \\ \forall v \in V.\ x(\delta^+(v)) - x(\delta^-(v)) = b(v) \\ \forall a \in A.\ 0 \leq x_a \leq c_a \end{aligned}$$

Vektor $x \in \mathbb{R}^A$ heißt zulässiger Fluss bzgl. b, c wenn die letzten beiden Gleichungen ($\uparrow$) erfüllt sind. Hierbei ist eine Beschränkung möglich auf

$$b \in \mathbb{R}^v, b(v) = 0,\ \forall c \in V \setminus \{s, t\}, b(s) > 0, b(t) = -b(s) < 0$$

dann spricht man auch von einem zulässigen (s, t)-Fluss mit Wert $b(s)$, was der zu transportierenden Gütermenge entspricht

Es lässt sich auch ein solches Problem mit Knotenkapazitäten lösen:

- Entferne einen Knoten v mit Knotenkapazität c
- Füge zwei Knoten v^-, v^+ ein und für alle Bögen $\delta^-(v)$ führe entsprechende Bögen zu v^- mit gleicher Kapazität und Kosten ein. Analog für v^+.
- Verbinde v^- mit v^+ durch einen Bogen von v^- nach v^+ mit Kosten 0 und Kapazität c

Zusammenhänge zu anderen Problemen:

Relevant zu dem Problem maximalen Fluss bei minimalen Kosten zu finden sind folgende Probleme:

- Kürzeste-Pfade-Problem: durch entfernen der Kapazitätsbeschränkungen
- Maximaler-Fluss-Problem: durch setzen aller Kosten auf 0

Knotenpotential Π und reduzierte Kosten:

$\Pi(i)$ Potential des Knoten i, welches mit 0 initiiert wird und später die Distanz der kürzesten Wege von s zu allen Knoten des Graphen angibt. Beispiel zur ersten Iteration des Graphen von G33 $d = (0, 3, 7, 2, 5, 3) = \Pi$

reduzierte Kosten von Bogen $ij \in A$:

$$w_{ij}^{\Pi} = w_{ij} + \Pi(i) - \Pi(j)$$

reduzierte Kosten bei Wegen:

$$c_{ij}^{d} = c_{ij} + d(i) - d(j)$$

Gerichteter Weg/Kreis:

gerichteter Weg P von k nach l: $\sum\limits_{ij \in P} w_{ij}^{\Pi} = \Pi(k) - \Pi(l) + \sum\limits_{ij \in P} w_{ij}$

gerichteter Kreis C: $\sum\limits_{ij \in C} w_{ij}^{\Pi} = \sum\limits_{ij \in C} w_{ij}$

Kreis-Löschungs-Algorithmus - Minimalkostenfluss x (falls existent) - Laufzeit $\mathcal{O}(m^2 nCW + mn^2)$:

$C = \max_{a \in A} c_A, W = \max_{a \in A} w_a$

Falls kein zulässiger Fluss x mit Hilfe des Max-Flow-Algorithmus gefunden wird, brich ab.

Solange $G(x)$ keinen negativen Kreis enthält:

Bestimme negativen Kreis C z.B. mit Floyd

bestimme $\varepsilon = \min\{r_{ij} | ij \in C\}$

Augmentiere ε Flusseinheiten entlang C und aktualisiere $G(x)$

Sukzessive-Kürzeste-Wege-Algorithmen:

Bei Kreis-Löschungs-Algorithmen ist x immer zulässig, aber nicht optimal. Bei den Sukzessive-Kürzeste-Wege-Algorithmen ist x immer optimal, aber nicht zulässig

Unausgewogenheitsfunktion $e : V \to \mathbb{R}$:

$e_x(v) = -b(v) + \sum_{a \in \delta^+(v)} x_a - \sum_{a \in \delta^-(v)} x_a$

$e_x(v) > 0$ Überschuss des Knoten $v \in V$, Menge dieser Knoten: P

$e_x(v) < 0$ Defizit des Knoten $v \in V$, Menge dieser Knoten: N

Da $\sum_{v \in V} e_x(v) = \sum_{v \in V} b(v) = 0$ gilt, also Bedarf $= 0$ impliziert dies $\sum_{v \in P} e_x(v) = -\sum_{v \in N} e_x(v)$

Sukzessive-Kürzeste-Wege-Algorithmus - Minimalkostenfluss x (falls existent):

Initialisiere $x = 0, \quad \Pi = 0, \quad \forall v \in V.\ e_x(v) = b(v)$

$P := \{v \in V | e_x(v) > 0\}, \quad N := \{v \in V | e_x(v) < 0\}$

Solange $P \neq \emptyset$:

Falls es keinen Weg von einem $k \in P$ zu einem $l \in N$ in $G(x)$ gibt
dann brich erfolglos ab

Bestimme kürzesten Weg und setze $\Pi = \Pi + d$ und $\varepsilon = \min\{e_x(k), -e_x(l), \min\{r_{ij} | ij \in W\}\}$

Augmentiere ε Flusseinheiten entlang W: $\quad x_{ij} = \begin{cases} x_{ij} + \varepsilon, & ij \in W \cap A_1(x) \\ x_{ij} - \varepsilon, & ij \in W \cap A_2(x) \\ x_{ij}, & \text{sonst} \end{cases}$

Skalierungsalgorithmus:

Δ-Skalierungsphase, in denen garantiert wird, dass mindestens Δ Einheiten an Fluss transportieren kann, initial ist $\Delta = 2^{\lfloor \log_2 B \rfloor}$, $B = \max_{v \in V} |b(v)|$. Am Phasenende wird Δ halbiert. Sobald $\Delta = 1$ wird Sukzessive-Kürzeste-Wege-Algorithmus durchgeführt. Beispiel G34

10 Heuristiken

Greedy - triff in jedem Schritt die Wahl, die momentan den meisten Profit verspricht

Unabhängigkeitssystem $\mathcal{I}$:

E endliche Grundmenge, $\mathcal{I} \subseteq \mathcal{P}(E)$ (also ein System von Teilmengen von E) heißt Unabhängigkeitssystem, falls $G \subseteq F \in \mathcal{I} \Rightarrow G \in \mathcal{I}$ und $\emptyset \in \mathcal{I}$. Alle Mengen $F \in \mathcal{I}$ heißen *unabhängig*, alle anderen *abhängig*. Es handelt sich also um eine Menge von Mengen, bei denen die Elemente der einzelnen Teilmengen Unabhängig sind.

Für einen ungerichteten Graphen ist die Menge aller Teilmengen, die Wälder (kreisfrei) sind ein Unabhängigkeitsproblem. Ist $F \subseteq E$, so ist eine unabhängige Teilmenge von F eine Basis von F, falls sie in keiner anderen unabhängigen Teilmenge von F enthalten ist.
Es gibt verschiedene Optimierungsprobleme:

$\max_{I \in \mathcal{I}} c(I)$ Maximierungsproblem über einem Unabhängigkeitssystem

$\min_{B Basis} c(B)$ Minimierungsproblem über einem Basissystem

Optimale Ergebnisse der Greedy-Algorithmen:

Greedy-Algorithmen Liefern genau dann eine optimale Lösung, wenn das Unabhängigkeitssystem ein Matroid ist, also über folgende Eigenschaft verfügt:
$\forall F \subseteq E, B, B'$, die Basen von F sind $\Rightarrow |B| = |B'|$, also auch $r_u(F) = r(F)$

Für $F \subseteq E$ bezeichne
$r(F) = \max\{|B| : B \text{ ist Basis für } F\}$ den Rang und
$r_u(F) = \min\{|B| : B \text{ ist Basis für } F\}$ den unteren Rang von F

$\min_{F \subseteq E} \frac{r_u(F)}{r(F)} \leq \frac{c(I_{\text{Greedy}})}{c(I_{\text{OPT}})} \leq 1$

Ist $\mathcal{I}$ ein Unabhängigkeitssystem auf E, dann sind folgende drei Aussagen äquivalent:
$\mathcal{I}$ ist Matroid
Greedy liefert optimale Lösung für alle $c \in \mathbb{R}^E$
Greedy liefert optimale Lösung für alle $c \in \{0,1\}^E$

Greedy-Max für Unabhängigkeitssysteme - zulässige Lösung $I_{\text{Greedy}} \in \mathcal{I}$:

Wähle Funktion w und sortiere E, so dass $w(e_1) \geq w(e_2) \geq \cdots \geq w(e_m), m = |E|$
$I_{\text{Greedy}} := \emptyset$
für alle $i := 1$ to m mit e_i, für die $w(e_i) > 0$ und $I_{\text{Greedy}} \cup \{e_i\} \in \mathcal{I}$ gilt setze:
$I_{\text{Greedy}} := I_{\text{Greedy}} \cup \{e_i\}$

Greedy-Min für Basissysteme - Basis B_{Greedy} von E - Laufzeit $\mathcal{O}(mn)$:

Liefert eine Optimallösung, sofern B_{Greedy} Kreisfrei. Also einen minimal aufspannenden Baum.

Wähle Funktion w und sortiere E, so dass $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$
$B_{\text{Greedy}} := \emptyset$
für alle $i := 1$ to m mit e_i, für die $B_{\text{Greedy}} \cup \{e_i\}$ eine Basis ist bzw. sich zu einer erweitern ließe setze:
$B_{\text{Greedy}} := B_{\text{Greedy}} \cup \{e_i\}$

Lokale Suche - Lösung $S \in \mathcal{F}$:

Eingaben:
$\mathcal{F}$ Menge zulässiger Lösungen
$N(S), S \in \mathcal{F}$ zulässige Nachbarschaft einer Lösung
$w : \mathcal{F} \to \mathbb{R}$ Gewichtsfunktion um Lösungen vergleichen zu können

Erfolgt hängt von Nachbarschaftsdefinitionen ab

Wähle Startlösung $S \in \mathcal{F}$
Wähle in Endlosschleife ein $S' \in N(S), w(S') < w(S)$, falls dies nicht existiert brich ab, sonst setze $S := S'$

Tabu Search - Lösung $S \in \mathcal{F}$:

Einige Nachbarschaften werden als Tabu markiert (Tabu-Liste beschränkter Größe, die die letzten Lösungen/Austäusche enthalten), um ein gewisses Maß an Kreiseln auszuschließen.

Eingabe wie bei der lokalen Suche
Stop-Bedingungen können eine Iterationsanzahl oder eine Iterationsanzahl ohne Verbesserung sein

Bestimme Startlösung $S \in \mathcal{F}$
Solange eine Stop-Bedingung nicht erfüllt do:
$N'(S) := N(S) \setminus T$ die tabulose Nachbarschaftsmenge
bestimme $S' \in \operatorname{argmin}\{w(R) : R \in N'(S)\}$ $S := S'$

Simulated Annealing - Lösung $S \in \mathcal{F}$ - hohe Laufzeit:

Bessere Lösungen werden stets erlaubt, schlechtere nur mit einer bestimmten Wahrscheinlichkeit, die umso geringer ist, je schlechter die Lösung ist. Diese Wahrscheinlichkeit wird im Laufe immer weiter verringert

Eingabe wie bei der lokalen Suche
Bestimmte Startlösung $S \in \mathcal{F}$
Bestimme Anfangstemperatoru $T \geq 0$
Solange Gefrierpunkt nicht erreicht do:
for $i : l$ do
$\Delta := w(S') - w(S)$
Wähle zufällig $x \in [0, 1]$
Falls $\Delta < 0 || x < e^{\frac{-\Delta}{T}}$, dann $S := S'$
Aktualisiere T, l

Genetischer Algorithmus - Lösung $S \in \mathcal{F}$:

Eingaben:
$\mathcal{F}$ Menge zulässiger Lösungen
w Fitnessfunktion

Ich bin etwas verwundert, dass hier nicht einmal eine Reparatur von verletzten Constraints erfolgt

Bestimme Population $S_1, \ldots, S_k$
Solange Abbruchkriterium nicht erfüllt do:
Elternwahl - Eine Menge von Lösungspaaren $S_1, \ldots, S_k$
Kreuzung - Bestimme Lösungen aus Elternpaaren
Mutation - Ändere zufällig einige der neuen Lösungen
Selektion - Bewerte Lösungen mittels Fittnessfunktion und wähle daraus neue Population

11 Abbildungsverzeichnis

Abbildungsverzeichnis

12 Tabellenverzeichnis

Tabellenverzeichnis