

Effiziente Algorithmen 1 - Zusammenfassung

Patrick Dammann

21.05.2017

1 Probleme und Algorithmen

Lineares kombinatorisches Optimierungsproblem

Gegeben sind eine endliche Menge E , ein System von Teilmengen $\mathcal{I} \subseteq 2^E$ (zulässige Lösungen) und eine Funktion $c : E \rightarrow \mathbb{R}$. Es ist eine Menge $I^* \in \mathcal{I}$ zu bestimmen, so dass $c(I^*) = \sum_{e \in I^*} c(e)$ minimal bzw. maximal ist.

Euklidisches Traveling-Salesman-Problem

Gegeben sind n Punkte in der Euklidischen Ebene. Zu bestimmen ist eine geschlossene Tour, die jeden Punkt genau einmal besucht und möglichst kurz ist.

E = Menge der Kanten

\mathcal{I} = Alle Mengen von Kanten, die eine Tour bilden

Euklidisches Matching-Problem

Gegeben sind n Punkte in der Euklidischen Ebene (n gerade). Zu bestimmen sind $\frac{n}{2}$ Linien, so dass jeder Punkt Endpunkt genau einer Linie ist und die Summe der Linienlängen so klein wie möglich ist.

E = Menge der Kanten

\mathcal{I} = Alle Mengen von Kanten mit der Eigenschaft, dass jeder Knoten zu genau einer der Kanten gehört.

Einheitskosten-Modell Es werden nur die Schritte des Algorithmus gezählt, die Zahlengrößen bleiben unberücksichtigt.

Bit-Modell Die Laufzeit für eine arithmetische Operation ist M , wobei M die größte Kodierungslänge einer an dieser Operation beteiligten Zahl ist.

Definition 1.1. Die Laufzeitfunktion $f_A : \mathbb{N} \rightarrow \mathbb{N}$ ist in $\mathcal{O}(g)$ für eine Funktion $g : \mathbb{N} \rightarrow \mathbb{N}$ falls es eine Konstante $c > 0$ und $n_o \in \mathbb{N}$ gibt, so dass $f_A \leq c \cdot g(n)$ für alle $n \geq n_o$.

Definition 1.2. Ein Algorithmus heißt **effizient** bzw. **polynomialer Algorithmus**, wenn seine Laufzeit in $\mathcal{O}(n^k)$ liegt.

Ein Problem, das mit einem polynomialen Algorithmus gelöst werden kann, heißt **polynomiales Problem**.

Definition. Ein **Graph** G ist ein Tupel $G = (V, E)$ ¹ bestehend aus einer nicht-leeren Knotenmenge V und einer Kantenmenge E .

- Ein Graph heißt **endlich**, wenn V und E endlich sind.
- Wenn $e = \{u, v\} \in E$ und $u, v \in V$, dann sind u und v **Nachbarn** bzw. **adjazent**, sind **Endknoten** von e und werden von e **verbunden**.
- Eine Kante $e = \{u, u\} \in E$ heißt **Schleife**.
- Kanten mit $E \ni e = \{u, v\} = f \in E$ heißen **parallel** oder **mehrfach**.
- Ein Graph ohne Mehrfachkanten heißt **einfach**.
- Für $W \subseteq V$ bekommt die Menge aller Knoten in $V \setminus W$ mit Nachbarn in W die Bezeichnung $\Gamma(W)$.
- Kurzform von $\Gamma(\{v\})$ ist $\Gamma(v)$.
- Die Menge $\delta(W)$ aller Kanten mit je einem Endknoten in W und $V \setminus W$ heißt **Schnitt**.
- Kurzform von $\delta(\{v\})$ ist $\delta(v)$.
- Der **Grad** eines Knoten v ist die Anzahl seiner Nachbarn, bzw. $|\delta(v)|$.
- Ein **(s,t)-Schnitt** ist ein Schnitt $\delta(V)$ mit $s \in W$ und $t \in V \setminus W$ und gleichzeitig ein (t,s)-Schnitt.
- Mit $W \subseteq V$ ist $E(W)$ die Menge aller Kanten mit beiden Endknoten in W .
- Mit $F \subseteq E$ ist $V(F)$ die Menge aller Knoten, die Endknoten von mind. einer Kante in F sind.
- Sind $G = (V, E)$ und $H = (W, F)$ Graphen und $W \subseteq V$ und $F \subseteq E$, so heißt H **Untergraph** von G .
- Mit $W \subseteq V$ ist $G - W$ der Graph G ohne die Knoten in W und ohne alle Kanten an W .
- $G[W] = G - (V \setminus W)$ ist der **von W induzierte Untergraph**.
- Mit $F \subseteq E$ ist $G - F = (V, E \setminus F)$.
- Kurzform von $G - \{x\}$ ist $G - x$ für $x \in E$ oder $x \in V$.
- Ein einfacher Graph heißt **vollständig**, wenn jede mögliche Kante zwischen seinen Knoten existiert.
- Der vollständige Graph mit n Knoten wird mit $K_n = (V_n, E_n)$ bezeichnet.
- Das **Komplement** des Graphen $G = (V, E)$ ist $\bar{G} = (V, E_n \setminus E)$.
- Ein Graph heißt **bipartit**, wenn er sich in zwei disjunkte Teilmengen V_1, V_2 mit $V_1 \cup V_2 = V$ teilen lässt, ohne dass es Kanten $\{u, v\}$ mit $u, v \in V_1 \vee u, v \in V_2$ gibt.

Definition. Ein **Digraph** G ist ein Tupel $D = (V, A)$ bestehend aus einer nicht-leeren Knotenmenge V und einer Kantenmenge A .

- Wenn $a = (u, v) \in A$ und $u, v \in V$, dann ist u **Anfangsknoten** und v **Endknoten** von a . Hier heißt u **Vorgänger** von v und v **Nachfolger** von u .
- Die Kanten (u, v) und (v, u) heißen **antiparallel**.
- Mit $W \subseteq V$ ist $A(W)$ die Menge aller Kanten mit Anfangs- und Endknoten in W .
- Mit $B \subseteq A$ ist $V(B)$ die Menge aller Knoten, die Anfangs- oder Endknoten von mind. einer Kante in B sind.

¹In der Vorlesung werden primär endliche, einfache, schleifenfreie Graphen behandelt, die der Einfachheit halber eine Notation ohne Inzidenzfunktion nutzen können. In diesem Skript wird (sofern nicht anders angegeben) von solchen Graphen ausgegangen.

- $G = (V, E)$ ist der unterliegende Graph von $D = (V, A)$, wenn E genau die Kanten $\{u, v\}$ enthält, für die (u, v) oder (v, u) in A liegen.
- Ein einfacher Digraph heißt **vollständig**, wenn jede mögliche Kante (in beide Richtungen) zwischen seinen Knoten existiert.
- Der vollständige Digraph mit n Knoten wird mit $D_n = (V_n, A_n)$ bezeichnet.
- Für $W \subseteq V, W \neq V \neq \emptyset$ enthält $\delta^+(W) = \{(i, j) \in A \mid i \in W, j \notin W\}$ alle Kanten, die W verlassen, $\delta^-(W) = \delta^+(V \setminus W)$ alle Kanten, die in W hinein führen und $\delta(W) = \delta^+(W) \cup \delta^-(W)$ beide. Diese Mengen heißen **Schnitt**. Es gelten die Kurzformen für einzelne Knoten.
- Für $s \in W$ und $t \notin W$ heißt $\delta^+(W)$ auch (s, t) -**Schnitt**.
- Die Kardinalitäten der Schnitte heißen **Außengrad** $(|\delta^+(v)|)$, **Innengrad** $(|\delta^-(v)|)$ und **Grad** $(|\delta(v)|)$.
- $\delta^+(W)$ heißt **gerichteter Schnitt**, wenn $\delta^-(W) = \emptyset$.

Definition. Eine endliche Folge $W = (v_0, e_1, v_1, e_2, v_2, \dots, e_k, v_k)$ heißt **Kette** oder $[v_0, v_k]$ -Kette der Länge k , wenn jede Kante e_i die Knoten v_{i-1} und v_i in einem (Di-)Graphen indiziert und **gerichtete Kette** oder (v_0, v_k) -Kette, wenn alle Kanten in der Form $e_i = (v_{i-1}, v_i)$ sind. v_0 und v_k heißen **Anfangs- und Endknoten**.

- Gibt es in einem (Di-)Graphen keine parallelen Kanten, ist eine (gerichtete) Kette durch ihre Knoten eindeutig identifiziert.
- Ein **(gerichteter) Weg** oder **(gerichteter) Pfad** ist eine (gerichtete) Kette, in der alle Knoten verschieden sind.
- Die Notation $u \xrightarrow{D} v$ bedeutet, dass es einen (u, v) -Weg in D gibt.
- Man spricht auch von (u, v) -Wegen bzw. (u, v) -Pfaden.
- Die Knoten s und t eines Graphen heißen **zusammenhängend**, wenn ein (s, t) -Weg existiert.
- Ein **zusammenhängender Graph** enthält nur Knoten, die paarweise zusammenhängend sind.
- Ein Digraph heißt **stark zusammenhängend**, wenn es zu jedem Knotenpaar s, t einen (s, t) -Weg und einen (t, s) -Weg gibt.
- **Komponenten** eines Graphen sind die (bezüglich Kanteninklusion) maximalen zusammenhängenden Untergraphen.
- **Starke Komponenten** eines Digraphen sind die (bezüglich Kanteninklusion) maximalen stark zusammenhängenden Unterdigraphen.
- Ein Graph heißt **k -fach zusammenhängend**, wenn jedes Paar Knoten s, t durch mindestens k (s, t) -Wege verbunden ist, die keine inneren Knoten gemeinsam haben.
- Ein Digraph heißt **k -fach stark zusammenhängend**, wenn jedes Paar Knoten s, t durch mindestens k (s, t) -Wege und (t, s) -Wege verbunden ist, die keine inneren Knoten gemeinsam haben.
- Eine **geschlossene Kette** hat mehr als 0 Kanten und den gleichen Anfangs- und Endknoten.
- Ein **Kreis** ist eine geschlossene Kette mit paarweise verschiedenen inneren Knoten. Seine **Länge** ist die Anzahl seiner Kanten.
- Ein **Eulerpfad** ist eine Kette, die jede Kante eines (Di-)Graphen einmal enthält.
- Eine **Eulertour** ist ein geschlossener Eulerpfad.
- Ein **Eulergraph** ist ein Graph, der eine Eulertour enthält.

- Ein **Hamiltonkreis** (oder **Hamiltontour**) ist ein Kreis der Länge $|V|$.
- Ein **hamiltonischer** Graph enthält einen Hamiltonkreis.
- Ein **Hamiltonweg** ist ein (gerichteter) Weg der Länge $|V| - 1$.
- Ein **Wald** ist eine Kantenmenge in einem Graphen, die keinen Kreis enthält.
- Ein **Baum** ist ein zusammenhängender Wald.
- Ein **aufspannender** Baum enthält alle Knoten des Graphen.
- Ein **azyklischer** Digraph enthält keine gerichteten Kreise.
- Ein **Branching** in einem Digraphen ist eine azyklische Kantenmenge, sodass jeder Knoten maximal eine Eingangs-kante besitzt.
- Eine **Arboreszenz** ist ein zusammenhängendes Branching.
- Eine **aufspannende Arboreszenz** enthält alle Knoten ihres Digraphen.
- **Gewichte** (auch "Kosten", "Distanzen", "Kapazitäten", usw.) werden durch Funktionen der Form $c : E \rightarrow \mathbb{R}$ bzw. $c : A \rightarrow \mathbb{R}$ mit Kanten assoziiert.

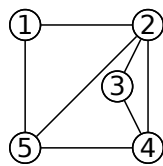
2 Grundlegende Graphenalgorithmen

2.1 Repräsentationen von Graphen

Adjazenzliste Jeder Knoten hat eine Liste seiner Nachbarn gespeichert. Speichersparend für dünne Graphen. Hinzufügen und Entfernen von Knoten und Kanten sehr einfach. Existenz von Kanten prüfen teuer (einen Knoten durchlaufen). Speicheraufwand: $\mathcal{O}(V + E)$

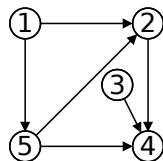
Adjazenzmatrix $|V| \times |V|$ -Matrix, die an der Stelle (u, v) eine 1 hat, wenn die Kante (u, v) bzw. $\{u, v\}$ existiert. Existenz von Kanten prüfen in $\mathcal{O}(1)$. Nachbarn durchlaufen unabhängig von ihrer Anzahl in $\mathcal{O}(V)$. Speicheraufwand: $\mathcal{O}(V^2)$

Inzidenzmatrix $|V| \times |E|$ -Matrix, die an der Stelle (v, e) eine 1 hat, wenn der Knoten v Endknoten der Kante e ist. Erlaubt Untersuchung des Graphen mit diversen, algebraischen Methoden. *[Kein Beispiel]*



$$\begin{bmatrix} 1 & \rightarrow 2 & \rightarrow 5 \\ 2 & \rightarrow 1 & \rightarrow 3 & \rightarrow 4 & \rightarrow 5 \\ 3 & \rightarrow 2 & \rightarrow 4 \\ 4 & \rightarrow 2 & \rightarrow 3 & \rightarrow 5 \\ 5 & \rightarrow 1 & \rightarrow 2 & \rightarrow 4 \end{bmatrix}$$

$$\begin{bmatrix} & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$



$$\begin{bmatrix} 1 & \rightarrow 2 & \rightarrow 5 \\ 2 & \rightarrow 4 \\ 3 & \rightarrow 4 \\ 4 & \\ 5 & \rightarrow 2 & \rightarrow 4 \end{bmatrix}$$

$$\begin{bmatrix} \nearrow & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 1 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

2.2 Durchsuchen von Graphen

Das "färben" von Knoten ist eine Kurzschreibweise für folgende Sachverhalte:

- weiß Der Knoten ist noch nicht erreicht. (Im Normalfall Grundzustand)
- grau Der Knoten wurde erreicht, seine Nachbarn jedoch noch nicht abgearbeitet.
- schwarz Der Knoten wurde komplett bearbeitet.

2.2.1 Breitensuche (BFS)

Start bei Knoten s . $\pi[u] = v$ heißt v ist direkter Vorgänger von u . Q ist eine Queue.

BFS(G, s):

- (1) Färbe s grau, setze $d[s] = 0$, $\pi[s] = \perp$ und initialisiere Q mit s .
- (2) Färbe alle Knoten $v \in V \setminus \{s\}$ weiß und setze $d[v] = +\infty$ und $\pi[v] = \perp$.
- (3) Falls Q leer, *Stop*(“BFS beendet”), sonst sei u erster Knoten in Q .
- (4) Für alle v aus Adjazenzliste von u :
 - (4.1) Falls v weiß ist, färbe v grau, setze $d[v] = d[u] + 1$, $\pi[v] = u$ und füge v ans Ende von Q ein.
- (5) Entferne u aus Q , färbe u schwarz und gehe zu (3).

Laufzeit: linear in Bezug auf Adjazenzstruktur. Das heißt $\mathcal{O}(V + E)$ bei Adjazenzlisten und $\mathcal{O}(V^2)$ bei Adjazenzmatrizen.

Definition. Für $v \in V$ sei $\delta(s, u)$ die Zahl der Kanten des kürzesten (s, u) -Weges, bzw. ∞ wenn kein solcher existiert.

Lemma 2.1. Sei $G = (V, E)$ ein Graph und $s \in V$.

- Für jede Kante $uv \in E$ gilt: $\delta(s, v) \leq \delta(s, u) + 1$.
- Nach Terminierung von $\text{BFS}(G, s)$ gilt: $\forall v \in V : d[v] \geq \delta(s, v)$
- Enthält Q während $\text{BFS}(G, s)$ v_1, v_2, \dots, v_r gilt: $d[v_r] \leq d[v_1] + 1$ und $d[v_i] \leq d[v_{i+1}], 1 \leq i < r$

Satz 2.2. Sei $G = (V, E)$, $s \in V$ und $\text{BFS}(G, s)$ ausgeführt. Dann ist jeder Knoten, der von s aus erreichbar ist, schwarz gefärbt und es gilt $d[v] = \delta(s, v)$.

2.2.2 Tiefensuche (DFS)

Kein spezieller Startknoten. Die Zeit der Grau-Färbung wird in $d[v]$ gespeichert, die Zeit der Schwarz-Färbung (Terminierungszeit) in $t[v]$.

DFS(D):

- (1) Färbe alle Knoten $u \in V$ weiß und setze $\pi[u] = \perp$.
- (2) Setze globale Zeit $t = 0$.
- (3) Für jeden Knoten $u \in V$:
 - (3.1) Falls u weiß gefärbt ist, dann $\text{DFSVisit}(u)$.

DFSVisit(u):

- (1) Färbe u grau, setze $t = t + 1$ und $d[u] = t$.
- (2) Für alle v aus Adjazenzliste von u :
 - (2.1) Falls v weiß ist, dann setze $\pi[v] = u$ und vollziehe $\text{DFSVisit}(v)$.
- (3) Färbe u schwarz, setze $t = t + 1$ und $f[u] = t$.

Laufzeit: linear in Bezug auf Adjazenzstruktur. Das heißt $\mathcal{O}(V + A)$ bei Adjazenzlisten und $\mathcal{O}(V^2)$ bei Adjazenzmatrizen.

Definition. Der **DFS-Wald** des (Di-)Graphen $G = (V, E)$ ist ein Digraph der Form $(V, \{\{\pi[v], v\} \mid v \in V, \pi[v] \neq \perp\})$. Er symbolisiert also den Weg durch den Graphen während einer DFS. Alle Kanten $\{u, v\}$ aus E , die nicht zum Wald gehören sind:

Vorwärtskanten Es gibt einen (u, v) -Weg im DFS-Wald

Rückwärtskanten Es gibt einen (v, u) -Weg im DFS-Wald

Kreuzungskanten Es gibt keinen der Wege im DFS-Wald

Für ungerichtete Graphen gibt es nur Rückwärtskanten.

Satz 2.3. Ein Knoten v ist Nachfolger eines Knotens u im DFS-Wald genau dann, wenn gilt: Zu dem Zeitpunkt, zu dem u grau gefärbt wird, ist v von u aus auf einem Weg erreichbar, der nur aus weißen Knoten besteht.

2.3 Topologisches Sortieren

Die topologische Sortierung eines Digraphen ist eine Knotenreihenfolge, in der alle Kanten nur zu später auftretenden Knoten führen.

Topsort(D):

- (1) Initialisiere leere Liste $L = \emptyset$.
- (2) Führe DFS(D) aus, mit Zusatz: Wenn ein Knoten v schwarz gefärbt wird, füge ihn am Anfang der Liste L ein.
- (3) L liefert topologische Sortierung.

Laufzeit: siehe DFS(D)

Lemma 2.4. Ein Digraph D ist genau dann kreisfrei, wenn DFS(D) keine Rückwärtskanten liefert.

Satz 2.5. Ein azyklischer Digraph D wird durch Topsort(D) topologisch sortiert.

2.4 Starke Zusammenhangskomponenten

StrongComponents(D):

- (1) Führe DFS(D) aus, merke Terminierungszeiten $f[u]$ für alle $u \in V$.
- (2) Generiere D^\top .
- (3) Führe DFS(D^\top) aus, wobei im Schritt (3) die Knoten nach absteigenden Werten von $f[u]$ sortiert werden.
- (4) Die Knoten jedes Baumes im DFS-Wald von D^\top bestimmen eine starke Zusammenhangskomponente.

Lemma 2.6. Jeder Weg, der zwei Knoten aus derselben Komponente verbindet, enthält nur Knoten aus dieser Komponente.

Lemma 2.7. Alle Knoten einer Komponente sind im gleichen DFS-Baum enthalten.

Definition. Für einen Knoten u bezeichne $\phi(u)$ den Knoten v mit der größten Terminierungszeit bei DFS(D), der von u aus in D erreichbar ist, das heißt:

$$\phi(u) = \operatorname{argmax}_v \left\{ f[v] \mid u \xrightarrow{D} v \right\}$$

Satz 2.8. Für $D = (V, A)$ sei DFS(D) ausgeführt. Dann ist für jeden Knoten $\phi(u)$ Vorgänger von u im DFS-Wald.

Satz 2.9. Nach DFS(D) liegen zwei Knoten u und v genau dann in der gleichen Komponente, wenn $\phi(u) = \phi(v)$.

Satz 2.10. Der Algorithmus StrongComponents identifiziert die starken Zusammenhangskomponenten eines Digraphen.

3 Optimale Bäume und Branchings

3.1 Minimale aufspannende Bäume

Die Probleme, einen minimalen, aufspannenden Baum oder einen maximalen, aufspannenden Wald zu finden lassen sich einfach ineinander transformieren. Darum beschränken wir uns auf ersteres.

Minimaler-aufspannender-Baum-Problem

Gegeben ist ein Graph $G = (V, E)$ mit Kantengewichten c_e für $e \in E$. Zu bestimmen ist ein aufspannender Baum für G , dessen Gesamtgewicht möglichst klein ist.

(Wir nehmen an, dass G zusammenhängend ist, da das Problem sonst einzeln pro Komponente lösbar wäre.)

3.1.1 Ein allgemeiner MST-Algorithmus

Dieser Algorithmus färbt Kanten nach festen Regeln blau und rot. Es ist stets die Invariante „Es gibt einen minimalen, aufspannenden Baum, der alle blauen und keine rote Kante enthält.“ erfüllt.

Regel B Wähle einen Schnitt, der keine blaue Kante enthält. Färbe eine seiner kürzesten Kanten blau.

Regel R Wähle einen Kreis, der keine rote Kante enthält. Färbe eine seiner längsten Kanten rot.

MinimumSpanningTree(G, c):

- (1) Zu Beginn seien alle Kanten ungefärbt.
- (2) Wende Regeln B und R an, bis alle Kanten gefärbt sind.
- (3) Die blauen Kanten bilden einen minimalen aufspannenden Baum.

Satz 3.1. Der Algorithmus färbt alle Kanten eines zusammenhängenden Graphen und erhält dabei die Invarianz-Bedingung, d.h. er berechnet einen minimalen aufspannenden Baum.

3.1.2 Der Algorithmus von Boruvka

Boruvka(G, c):

- (1) Initialisiere n blaue Bäume bestehend aus je einem Knoten.
- (2) Solange mehr als ein blauer Baum vorhanden ist, wähle gleichzeitig zu jedem blauen Baum die minimale inzidente Kante und färbe alle ausgewählten Kanten blau.
- (3) Die blauen Kanten bilden einen minimalen Baum.

Der Algorithmus kann blaue Kreise erzeugen, wenn es im Graphen Kanten gleichen Gewichts gibt.

3.1.3 Der Algorithmus von Kruskal

Kruskal(G, c):

- (1) Initialisiere n blaue Bäume bestehend aus je einem Knoten.
- (2) Sortiere die Kanten von G nach nichtabsteigenden Gewichten, so dass $c_{e_1} \leq c_{e_2} \leq \dots \leq c_{e_m}$.
- (3) Für $i = 1, 2, \dots, m$:
 - (3.1) Sind die Endknoten von e_i im gleichen blauen Baum, färbe e_i rot, andernfalls blau.
- (4) Die blauen Kanten bilden einen minimalen Baum.

Laufzeit: $\mathcal{O}(E \log V)$, wenn die Bäume per fast-union-find verwaltet werden. Ein Heap ermöglicht Optimierung bezüglich der Sortierung, da abgebrochen werden kann, wenn es $n - 1$ blaue Kanten gibt.

3.1.4 Der Algorithmus von Prim

Der Algorithmus verwendet nur Regel R und benötigt einen Startknoten s .

Prim(G, c, s):

- (1) Initialisiere n blaue Bäume bestehend aus je einem Knoten.
- (2) Solange noch blaue Bäume aus einem Knoten existieren, färbe minimale Kante im Schnitt induziert durch den Baum, der s enthält, blau.
- (3) Die blauen Kanten bilden einen minimalen Baum.

Es folgt eine einfache Implementierung. T ist der angehende Baum:

Prim1(G, c, s):

- (1) Setze $V_T = \{s\}$, $T = \emptyset$ und $i = 0$.
- (2) Falls $i = n - 1$, *Stop*(„ T ist MST“).
- (3) Bestimme $u \in V_T$ und $v \in V \setminus T_T$ mit $c_{uv} = \min \{c_{xy} | x \in V_T, y \in V \setminus V_T\}$.
- (4) Setze $V_T = V_T \cup \{v\}$ und $T = T \cup \{uv\}$ und $i = i + 1$. Gehe zu (2).

Laufzeit: $\mathcal{O}(VE)$

Es gibt eine bessere Implementierung. $t[v]$ speichert den nächsten Baumknoten, $d[v]$ die zugehörige Distanz.

Prim2(G, c, s):

- (1) Setze $V_T = \{s\}$, $T = \emptyset$ und $i = 0$. Setze $d[v] = c_{sv}$ und $t[v] = s$, falls $sv \in E$, setze $d[v] = \infty$ und $t[v] = \perp$, falls $sv \notin E$.
- (2) Falls $i = n - 1$, *Stop*(„ T ist MST“).
- (3) Bestimme $v \in V \setminus T_T$ mit $d[v] = \min \{d[u] | u \in V \setminus V_T\}$.
- (4) Setze $V_T = V_T \cup \{v\}$ und $T = T \cup \{t[v]v\}$.
- (5) Für alle w adjazent zu v :
 - (5.1) Falls $w \in V \setminus V_T$ und $c_{vw} < d[w]$, dann setze $d[w] = c_{vw}$ und $t[w] = v$.
- (6) Setze $i = i + 1$ und gehe zu (2).

Laufzeit (naiv): $\mathcal{O}(V^2)$

Laufzeit (Heap): $\mathcal{O}(E \log V)$ (besser für dünnbesetzte Graphen)

Laufzeit (Fib-Heap): $\mathcal{O}(E + V \log V)$

3.1.5 Der Round-Robin-Algorithmus

RoundRobin(G, c):

- (1) Initialisiere n blaue Bäume bestehend aus je einem Knoten.
- (2) Solange weniger als $n - 1$ blaue Kanten vorhanden sind, wähle einen blauen Baum, bestimme die kürzeste Kante im durch diesen Baum induzierten Schnitt und färbe sie blau.
- (3) Die blauen Kanten bilden einen minimalen Baum.

Eine $\mathcal{O}(E \log \log V)$ -Implementation ist möglich, wenn immer der kleinste Baum gewählt wird.

3.1.6 Eine Anwendung aufspannender Bäume

Sogenannte 1-Bäume sind eine untere Schranke für eine optimale TSM-Tour.

OneTree(G, c):

- (1) Bestimme für die Knoten $\{2, 3, \dots, n\}$ einen MST T . Sei c_T die Länge des MST.
- (2) Seien e_1 und e_2 die zwei kürzesten Kanten an Knoten 1.
- (3) $T \cup \{e_1, e_2\}$ ist optimaler 1-Baum mit Wert $c_T + c_{e_1} + c_{e_2}$.

3.2 Maximale Branchings

Neue Notationen: $s : A \rightarrow V$, $t : A \rightarrow V$ und $c : A \rightarrow \mathbb{R}$ sind Start- und Endknoten sowie Gewichte von Kanten.

Maximales-Branching-Problem

Gegeben ist ein Digraph $D = (V, A)$ mit Kantengewichten c_e , für $e \in A$. Zu bestimmen ist ein Branching für D , dessen Gesamtgewicht möglichst groß ist.

3.2.1 Der Branching-Algorithmus von Edmonds

Definition 3.2. Sei $D = (V, A)$ ein Digraph mit Kantengewichten c_e , für $e \in A$.

- a) Eine Kante $e \in A$ heißt **kritisch**, falls $c(e) > 0$ und falls $c(e') \leq c(e)$, für alle $e' \in A$ mit $t(e') = t(e)$.
- b) Ein Subgraph $H \subseteq A$ heißt **kritisch**, falls er nur aus kritischen Kanten besteht, jeder Knoten Endknoten von höchstens einer dieser Kanten ist, und falls er inklusionsmaximal bezüglich dieser Eigenschaft ist.

Lemma 3.3. Ein azyklischer kritischer Graph H ist ein maximaler Branching.

Lemma 3.4. Sei H ein kritischer Graph. Dann ist jeder Knoten von H in maximal einem Kreis enthalten.

Lemma 3.5. Seien B ein Branching und u, v, w drei Knoten. Falls $u \xrightarrow{B} v$ und $w \xrightarrow{B} v$, dann gilt entweder $u \xrightarrow{B} w$ oder $w \xrightarrow{B} u$.

Definition. Sei B ein Branching. Eine Kante $e \notin B$ heißt **zulässig** (relativ zu B), falls die Menge $B' = B \cup \{e\} \setminus \{f \mid f \in B \wedge t(f) = t(e)\}$ ebenfalls ein Branching ist.

Lemma 3.6. Sei B ein Branching und $e \in A \setminus B$. Dann ist e genau dann zulässig relativ zu B , wenn kein Weg von $t(e)$ nach $s(e)$ in B existiert.

Lemma 3.7. Sei B ein Branching und C ein Kreis mit der Eigenschaft, dass keine Kante aus $C \setminus B$ zulässig relativ zu B ist. Dann gilt $|C \setminus B| = 1$.

Satz 3.8. Sei H ein kritischer Graph. Dann existiert ein Branching B mit maximalem Gewicht, so dass für jeden Kreis $C \subseteq H$ gilt $|C \setminus B| = 1$.

Im Folgenden bezeichne $V_i = V(C_i)$ und a_i die kürzeste Kante in C_i .

Korollar 3.9. Sei $D = (V, A)$ ein Digraph mit Kantengewichten und H ein kritischer Graph mit Kreisen $C_i, i = 1, 2, \dots, k$. Es existiert ein gewichtsmaximales Branching B mit den Eigenschaften

- a) $|C_i \setminus B| = 1$, für $i = 1, 2, \dots, k$
- b) Falls für jede Kante $e \in B \setminus C_i$ gilt, dass $t(e) \notin V_i$, dann folgt $C_i \setminus B = \{a_i\}$.

Schrumpfen von kritischen Graphen:

Sei $\tilde{V} = V \setminus \bigcup_{i=1}^k V_i$. Für $v \in V_i$ sei $\tilde{e}(v)$ die Kante aus C_i mit $t(\tilde{e}(v)) = v$, d.h. die Kreiskante mit Endknoten v .

$$\begin{aligned}\bar{A} &= \{e \in A \mid \text{für kein } i \text{ gilt } s(e) \in V_i \text{ und } t(e) \in V_i\} \\ &= A \setminus \bigcup_{i=1}^k A(V(C_i)), \\ \bar{V} &= \tilde{V} \cup \{w_1, w_2, \dots, w_k\}, \text{ wobei die } w_i \text{ neue Symbole sind,} \\ \bar{s}(e) &= \begin{cases} s(e), & \text{falls } s(e) \in \tilde{V}, \\ w_i & \text{falls } s(e) \in V_i, \end{cases} \\ \bar{t}(e) &= \begin{cases} t(e), & \text{falls } t(e) \in \tilde{V}, \\ w_i & \text{falls } t(e) \in V_i, \end{cases} \\ \bar{c}(e) &= \begin{cases} c(e), & \text{falls } t(e) \in \tilde{V}, \\ c(e) - c(\tilde{e}(t(e))) + c(a_i) & \text{falls } t(e) \in V_i. \end{cases}\end{aligned}$$

Es entsteht ein neues Problem \bar{P} im Digraphen $\bar{D} = (\bar{V}, \bar{A})$ mit den neuen Inzidenzfunktionen \bar{s} , \bar{t} und Gewichtsfunktion \bar{c} .

Satz 3.10. *Es gibt eine bijektive Abbildung zwischen \mathcal{B} (der Menge der Branchings im Originalproblem) und der Menge der Branchings im Problem \bar{P} . Es korrespondiert das Branching $B \in \mathcal{B}$ mit dem Branching $\bar{B} = B \cap \bar{A}$ in \bar{P} und es gilt:*

$$c(B) - \bar{c}(\bar{B}) = \sum_{i=1}^k c(C_i) - \sum_{i=1}^k c(a_i).$$

Branching(D, c):

- (1) Bestimme einen kritischen Graphen H für D .
- (2) Ist H azyklisch, dann $Stop(„H$ ist optimales Branching für $D^{\circ}“)$.
- (3) Schrumpfe die Kreise von H , um \bar{D} und \bar{c} zu erhalten.
- (4) Berechne durch den rekursiven Aufruf $Branching(\bar{D}, \bar{c})$ ein optimales Branching \bar{B} für \bar{D} .
- (5) Expandiere das Branching \bar{B} zu einem optimalen Branching B für D .

Laufzeit: $\mathcal{O}(A \log V)$ oder $\mathcal{O}(V^2)$.

3.2.2 Arboreszenzen und das asymmetrische TSP

→analog zum symmetrischen Fall

4 Kürzeste Wege**Kürstester- (s, t) -Weg-Problem**

Gegeben sind ein Digraph $D = (V, A)$ mit Kantengewichten c_e für $e \in A$, und zwei Knoten $s, t \in V$. Zu bestimmen ist ein (s, t) -Weg in D , dessen Gesamtlänge möglichst klein ist.

4.1 Eigenschaften kürzester Wege

Definition. Im folgenden ist $s \in V$ der Startknoten, dessen kürzeste Wege berechnet werden sollen. $\delta(u, v)$ ist die Länge des kürzesten (u, v) -Weges, bzw. $+\infty$ (oder $-\infty$) wenn keiner existiert. $\pi[v]$ ist der Vorgängerknoten von v auf dem kürzesten (s, v) -Weg.

Lemma 4.1. Sei $p = (v_1v_2, v_2v_3, \dots, v_{k-1}v_k)$ ein kürzester Weg von v_1 nach v_k . Dann ist für $1 \leq i \leq j \leq k$ der Teilweg $(v_i v_{i+1}, \dots, v_{j-1} v_j)$ ein kürzester Weg von v_i nach v_j .

Lemma 4.2. Für jede Kante (u, v) gilt $\delta(s, v) \leq \delta(s, u) + c_{uv}$.

Die **Standardinitialisierung** für d ist $d[s] = 0$ und $d[v] = \infty, v \in V \setminus \{s\}$ und $\pi[v] = \perp, v \in V$.

Correct (u, v) :

Falls $d[v] > d[u] + c_{uv}$, dann setze $d[v] = d[u] + c_{uv}$ und $\pi[v] = u$.

Lemma 4.3. Nach Initialisierung gilt $d[v] \geq \delta(s, v)$, für alle $v \in V$. Dies gilt auch nach Ausführung einer beliebigen Folge von **Correct** (\cdot) -Aufrufen. Falls einmal $d[v] = \delta(s, v)$ gilt, so kann $d[v]$ nicht mehr verändert werden.

Lemma 4.4. D enthalte keine von s aus erreichbaren negativen Kreise und die Standardinitialisierung sei ausgeführt. Für jede Folge von Korrekturoperationen bildet der durch π bestimmte Digraph D_π eine Arborisierung mit Wurzel s .

4.2 Der Algorithmus von Dijkstra

Der Algorithmus berechnet alle kürzesten Wege von s , wenn $c_{uv} \geq 0$, für alle $(u, v) \in A$.

Dijkstra (D, c, s) :

- (1) Setze $d[v] = \infty$, für alle $v \in V \setminus \{s\}$, $d[s] = 0$ und $\pi[v] = \perp$ für alle $v \in V$.
- (2) $S = \emptyset, Q = V$.
- (3) Solange $Q \neq \emptyset$:
 - (3.1) Bestimme u mit $d[u] = \min_v \{d[v] \mid v \in Q\}$.
 - (3.2) $S = S \cup \{u\}, Q = Q \setminus \{u\}$
 - (3.3) Für jede Kante $(u, v) \in A$ mit $v \in Q$ führe **Correct** (u, v) aus.

Laufzeit (naiv): $\mathcal{O}(V^2)$

Laufzeit (Heap): $\mathcal{O}(A \log V)$ (besser für dünnbesetzte Graphen)

Laufzeit (Fib-Heap): $\mathcal{O}(A + V \log V)$

Satz 4.5. Nach Ausführung des Algorithmus von Dijkstra gilt $d[v] = \delta(s, v)$ für alle $v \in V$.

4.3 Der Algorithmus von Bellman und Ford

BellmanFord (D, c, s) :

- (1) Initialisiere $d[v] = \infty$, für alle $v \in V \setminus s$, $d[s] = 0, \pi[v] = \perp$ für alle $v \in V$.
- (2) Führe die folgende Schleife $(|V| - 1)$ -mal aus:
 - (2.1) Für alle $(u, v) \in A$ führe **Correct** (u, v) aus.
- (3) Für alle $(u, v) \in A$:
 - (3.1) Falls $d[v] > d[u] + c_{uv}$ dann **Stop**(„Graph enthält negativen Kreis“).
- (4) **Stop**(„Der Kürzeste-Wege-Baum wurde berechnet“).

Laufzeit: $\mathcal{O}(VA)$

Lemma 4.6. *D enthalte keinen von s aus erreichbaren negativen Kreis. Dann gilt bei Terminierung des Bellman-Ford-Algorithmus $d[v] = \delta(s, v)$ für alle Knoten in v, die von s aus erreichbar sind.*

Korollar 4.7. *Es gibt einen Weg von s nach v genau dann, wenn der Bellman-Ford-Algorithmus mit $d[v] < \infty$ terminiert.*

Satz 4.8. *Der Bellman-Ford-Algorithmus arbeitet korrekt, das heißt entweder er stellt fest, dass D einen negativen Kreis enthält, oder er berechnet die Arboreszenz der kürzesten Wege von s zu allen erreichbaren Knoten.*

Yen-Variante Ersetze (2.1) durch:

(2.1) Für $i = 1, \dots, n$: *Correct*(v_i, v_k) für alle $(v_i, v_k) \in A$ mit $i < k$.

(2.2) Für $i = n, n-1, \dots, 1$: *Correct*(v_i, v_j) für alle $(v_i, v_j) \in A$ mit $i > j$.

Variante von d'Esopo und Pape Bevorzugt Knoten, die sich verbessert haben. Exponentielle Worst-Case-Laufzeit, aber oft effizienter für dünne Graphen.

Modifikation der *Correct*-Operation:

Correct(u, v):

Falls $d[v] > d[u] + c_{uv}$, dann:

(1) Setze $d[v] = d[u] + c_{uv}$ und $\pi[v] = u$.

(2) Falls v noch nicht in Q war, setze v an das Ende von Q .

(3) Falls v schon in Q war, aber gegenwärtig nicht Q ist, dann setze v an den Anfang von Q .

Terminiert, wenn Q leer.

Verfahren von Nicholson Sucht kürzesten (s, t) -Weg gleichzeitig von s und t .

A*-Verfahren Verwendet Schätzwerte $s(v, t)$ für die Entfernung zum Zielknoten, priorisiert Knoten mit niedrigem $d[v] + s(v, t)$. Terminiert, wenn $d[t]$ und $s(s, t)$ nag genug beieinander.

4.4 Kürzeste Wege in azyklischen Digraphen

DAGShortestPath(D, c, s):

(1) Sei $t[1], \dots, t[n]$ eine topologische Sortierung der Knoten von D .

(2) Setze $d[v] = \infty$, für alle $v \in V \setminus \{s\}$, $\pi[v] = \perp$, für alle $v \in V$, und $d[s] = 0$.

(3) Für $i = 1, 2, \dots, n$:

(3.1) Setze $u = t[i]$.

(3.2) Für alle $v \in V$ mit $(u, v) \in A$ führe *Correct*(u, v) aus.

Laufzeit: $\mathcal{O}(V + A)$

4.4.1 Berechnung optimaler Lösungen des Knapsack-Problems

4.4.2 Eine Anwendung in der Tourenplanung

4.4.3 Längste Wege

Dies ist nur in azyklischen Digraphen möglich, oder wenn alle Kantengewichte negativ oder 0 sind.

Entweder man komplementiert alle Gewichte oder ändert die Initialisierung von d in $-\infty$ und dreht den Vergleich in der *Correct*-Operation um.

4.5 Kürzeste Wege zwischen allen Knotenpaaren

Eine Möglichkeit wäre das $|V|$ -malige Anwenden von Dijkstra oder Bellman-Ford, was in der $|V|$ -fachen Komplexität resultiert.

4.5.1 Der Algorithmus von Floyd und Warshall

D habe ganzzahlige Gewichte und keine negativen Kreise.

AllPairsShortestPaths(D, c):

- (1) Setze $d[i, j] = +\infty$, für alle $i, j \in V$, und $d[i, i] = 0$, für alle $i \in V$.
- (2) Für alle $(i, j) \in A$, setze $d[i, j] = c_{ij}$.
- (3) Solange es Knoten $i, j, k \in V$ gibt, mit $d[i, j] > d[i, k] + d[k, j]$ setze $d[i, j] = d[i, k] + d[k, j]$.

Laufzeit: $\mathcal{O}(|V|^2 (2|V|C))$ mit $C = \max\{|c_{ij}| \mid (i, j) \in A\} \rightarrow$ nicht poly!

Eine Verbesserung ergibt sich wie folgt:

Wenn alle kürzesten Wege bekannt sind, die nur die ersten $k - 1$ Knoten als Zwischenknoten verwenden, lässt sich ein kürzester Weg, der den k -ten Knoten (und möglicherweise seine Vorgänger) als Zwischenknoten nutzt durch $d(i, k) + d(k, j)$ bestimmen.

Sei $D^{(k)}$ eine Matrix mit $d_{ij}^{(k)}$ als kürzester Weg zwischen i und j , der nur die ersten k Knoten als Zwischenknoten nutzt:

FloydWarshall(C):

- (1) Initialisiere $D^{(0)} = C$, mit $c_{ij} = +\infty$ wenn keine Kante (i, j) existiert.
- (2) Für $k = 1, 2, \dots, |V|$:
 - (2.1) Für $i = 1, 2, \dots, |V|$:
 - (2.1.1) Für $j = 1, 2, \dots, |V|$:

$$d_{ij}^{(k)} = \min \left\{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right\}$$
- (3) $D^{(|V|)}$ enthält die Längen der kürzesten Wege.

Laufzeit: $\mathcal{O}(V^3)$

Johnson(D, c):

- (1) Bilde $D' = (V', A')$ mit $V' = V \cup \{s\}$, $A' = A \cup \{(s, v) \mid v \in V\}$ und setze $c_{sv} = 0$, für alle $v \in V$.
- (2) Führe *Bellman-Ford*(D', c, s) aus.
- (3) Enthält D' negative Kreise, *Stop*(„Algorithmus nicht anwendbar“)
- (4) Setze für jeden Knoten $v \in V$: $h[v] = \delta(s, v)$.
- (5) Setze für jede Kante $(u, v) \in A$: $\bar{c}_{uv} = c_{uv} + h[u] - h[v]$.
- (6) Für jeden Knoten $u \in V$:
 - (6.1) Führe *Dijkstra*(D, \bar{c}, u) aus. Sei $\bar{\delta}(u, v)$ die Kürzeste-Wege-Distanz von u nach $v \in V \setminus \{u\}$ (bzgl. \bar{c}).
 - (6.2) Für jedes $v \in V \setminus \{u\}$ ist die Länge des kürzesten (u, v) -Weges $\bar{\delta}(u, v) + h[v] - h[u]$.

Laufzeit: $\mathcal{O}(VA \log V)$ oder $\mathcal{O}(VA + V^2 \log V)$ je nach Dijkstra-Implementierung

4.6 Varianten kürzester Wege

4.6.1 Bottleneck-Probleme

In einem Graphen wird ein (s, t) -Weg gesucht, dessen kürzeste Kante größtmögliche Länge hat.

Initialisierung $d[v] = -\infty$, für alle $v \in V \setminus \{s\}$, $d[s] = +\infty$, $\pi[v] = \perp$, für alle $v \in V$.

Korrektur Falls $d[v] < \min\{d[u], c_{uv}\}$, dann setze $d[v] = \min\{d[u], c_{uv}\}$ und $\pi[v] = u$.

4.6.2 Netzwerke mit Gewinnen und Verlusten

Das **Arbitrage-Problem** funktioniert wie ein Kürzeste-Wege-Problem, nur dass Kanten auf einem Weg multipliziert werden. Entweder Initialisierung und Correct werden angepasst, oder alle Kantengewichte logarithmiert.

4.7 Kreise mit bestem Kosten-Zeit-Verhältnis

Gesucht ist $\mu^* = \min_{B \text{ Kreis in } D} \mu(B)$ mit $\mu(B) = \frac{\sum_{(i,j) \in B} c_{ij}}{\sum_{(i,j) \in B} \tau_{ij}}$.

μ ist ein Schätzwert für $\mu^* \in [-|V|C, +|V|C]$, welcher per Intervallschachtelung verbessert wird. ($C = \max c_{ij}$)

Es wird ein Digraph \overline{D} erstellt mit $l_{ij} = c_{ij} - \mu\tau_{ij}$ für alle $(i, j) \in A$.

- Enthält \overline{D} einen negativen Kreis, ist μ zu groß.
- Enthält \overline{D} nur positive Kreise, ist μ zu klein.
- Enthält \overline{D} einen Kreis der Länge 0, minimiert dieser die Zielfunktion.

Das Verfahren kann auch abgebrochen werden, wenn die Intervallgrenzen weniger als $\frac{1}{n^2 T^2}$ auseinander liegen. ($T = \max \tau_{ij}$)

Laufzeit: $\mathcal{O}(VA \log(CVT))$

5 Das Zuordnungsproblem

5.1 Anwendungen und Grundlagen

Perfektes-Matching-Problem in bipartiten Graphen

Sei $K_{n,n}$ der vollständige bipartite Graph mit jeweils n Knoten in den zwei Teilmengen V_1 und V_2 mit $V_1 \cup V_2 = V$ und Kantengewichten c_{ij} für $i \in V_1$ und $j \in V_2$. Gesucht ist ein perfektes Matching ($\forall v \in V : |\delta(v)| = 1$) in $K_{n,n}$ mit minimalem Gesamtgewicht.

Das Problem wird mit Hilfe linearer Programmierung gelöst werden. Es lässt sich wie folgt als lineares Programm darstellen:

$$\begin{aligned} \min \quad & \sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij} \\ \text{(AP)} \quad & \sum_{j=1}^n x_{ij} = 1, \text{ für } i = 1, 2, \dots, n \\ & \sum_{i=1}^n x_{ij} = 1, \text{ für } j = 1, 2, \dots, n \\ & x_{ij} \in \{0, 1\}, \text{ für } i = 1, 2, \dots, n, j = 1, 2, \dots, n \end{aligned}$$

Das duale Programm dazu ist:

$$(\text{AP}_D) \max \sum_{i=1}^n u_i + \sum_{j=1}^n v_j$$

$$u_i + v_j \leq c_{ij}, \text{ für } 1, 2, \dots, n, j = 1, 2, \dots, n.$$

Satz. *Schwacher Dualitätssatz (am Beispiel)*

Wenn x zulässig für (AP) ist und (u, v) zulässig für (AP_D) sind, gilt:

$$\sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij} \geq \sum_{i=1}^n u_i + \sum_{j=1}^n v_j$$

Satz. *Satz vom komplementären Schlupf (am Bsp.)*

Ein Paar von zulässigen Lösungen x für (AP) und (u, v) für (AP_D) sind genau dann optimal, wenn gilt

$$u_i + v_j < c_{ij} \Rightarrow x_{ij} = 0$$

bzw.

$$x_{ij} > 0 \Rightarrow u_i + v_j = c_{ij}$$

5.2 Die Ungarische Methode

HungarianMethod(n, C):

- (1) Bestimme eine dual zulässige Startlösung (u^0, v^0) , setze $i = 0$.
- (2) Konstruiere zu (u^i, v^i) einen 0/1-Vektor x^i , so dass die Bedingungen vom komplementären Schlupf erfüllt sind.
- (3) Beschreibe x^i eine Zuordnung, *Stop*(„Zuordnung optimal“)
- (4) Berechne eine neue dual zulässige Lösung (u^{i+1}, v^{i+1}) , setze $i = i + 1$, gehe zu (2).

(Dies ist die grobe Idee des Algorithmus)

Definition 5.1. Folgende Definitionen werden für die Methode benötigt:

- a) Für eine dual zulässige Lösung (u, v) heißt die Matrix $\bar{C} = (\bar{c}_{ij})$ mit $\bar{c}_{ij} = c_{ij} - u_i - v_j$ die **reduzierte Matrix**.
- b) Sei \bar{C} eine reduzierte Matrix. eine Menge $N \subseteq \{1, \dots, n\} \times \{1, \dots, n\}$ heißt Menge von **unabhängigen Nullen**, falls
 - (1) $\bar{c}_{ij} = 0$, für alle $(i, j) \in N$,
 - (2) $|\{j \mid (i, j) \in N\}| \leq 1$, für alle $i = 1, \dots, n$,
 - (3) $|\{i \mid (i, j) \in N\}| \leq 1$, für alle $j = 1, \dots, n$.
- c) Eine **Überdeckung** einer reduzierten Matrix ist eine Menge von Zeilen- und Spaltenindizes, so dass die zugehörigen Zeilen und Spalten sämtliche Nullelemente von \bar{C} enthalten.

Satz 5.2. Sei \bar{C} eine reduzierte Matrix. Die maximale Kardinalität einer Menge von unabhängigen Nullen ist gleich der minimalen Kardinalität einer Überdeckung.

(In einem bipartiten Graphen ist die maximale Kardinalität eines Matchings gleich der minimalen Kardinalität einer Kantenüberdeckung mit Knoten.)

5.2.1 Bestimmung einer Startlösung

Start(n, C):

- (1) Berechne Zeilenminima $u_i = \min \{c_{ij} | j = 1, \dots, n\}$, für alle $i = 1, \dots, n$.
- (2) Setze $\bar{c}_{ij} = c_{ij} - u_i$, für alle $i = 1, \dots, n, j = 1, \dots, n$.
- (3) Berechne Spaltenminima $v_j = \min \{c_{ij} | i = 1, \dots, n\}$, für alle $j = 1, \dots, n$.
- (4) Setze $\bar{c}_{ij} = \bar{c}_{ij} - v_j$, für alle $i = 1, \dots, n, j = 1, \dots, n$.

Wir gehen nun spaltenweise vor und zeichnen die Null mit dem niedrigsten Zeilenindex als unabhängige Null aus.

5.2.2 Bestimmung einer minimalen Überdeckung

MinimusCover(n, \bar{C}):

- (1) Alle Zeilen und Spalten von \bar{C} seien unmarkiert und nicht überprüft. Sind n unabhängige Nullen vorhanden gehe zu (5).
- (2) Alle Zeilen ohne unabhängige Null erhalten die Markierung „(-)“.
- (3) Gibt es keine markierte, noch nicht überprüfte Zeile, so gehe zu (4).
Andernfalls sei i eine solche Zeile.
 - (3.1) Erkläre i für überprüft.
 - (3.2) Existiert keine unmarkierte Spalte, die in Zeile i eine Null enthält, gehe zu (3).
Andernfalls sei j eine solche Spalte.
 - (3.3) Markiere j mit „(i)“.
 - (3.4) Besitzt j schon eine unabhängige Null gehe zu (3.2).
Andernfalls kann durch Verfolgen der Markierungen ausgehend von j eine alternierende Kette von abhängigen und unabhängigen Nullen konstruiert werden. Ändere die Klassifizierung dieser Nullen und gehe zu (1).
- (4) Gibt es keine markierte, noch nicht überprüfte Spalte, so gehe zu (5).
Andernfalls sei j eine solche Spalte.
 - (4.1) Erkläre j für überprüft.
 - (4.2) Markiere die Zeile, in der die unabhängige Null von Spalte j steht, mit „(j)“.
 - (4.3) Gehe zu (3).
- (5) Die Maximalzahl unabhängiger Nullen ist berechnet. Die zugehörige minimale Überdeckung ergibt sich durch Zeilen, die nicht markiert sind und Spalten, die markiert sind.

5.2.3 Korrektur der Duallösung

I bezeichnet die Indizes der Zeilen der minimalen Überdeckung, J die der Spalten.

DualUpdate(n, \bar{C}, I, J):

- (1) Setze $\delta = \min \{\bar{c}_{ij} | i \notin I \wedge j \notin J\}$.
- (2) Setze $u_i = \begin{cases} u_i + \delta, & \text{für } i \notin I, \\ u_i & \text{sonst,} \end{cases} \quad v_j = \begin{cases} v_j - \delta, & \text{für } j \notin J, \\ v_j & \text{sonst.} \end{cases}$
- (3) Setze $\bar{c}_{ij} = \begin{cases} \bar{c}_{ij} - \delta, & \text{für } i \notin I, j \notin J, \\ \bar{c}_{ij} + \delta, & \text{für } i \in I, j \in J, \\ \bar{c}_{ij}, & \text{sonst.} \end{cases}$

Satz 5.3. Die Ungarische Methode löst Zuordnungsprobleme in der Zeit $\mathcal{O}(n^3)$.

5.3 Ein dualer Algorithmus

Im Folgenden werden diese Notationen genutzt:

- $w_{ij} = c_{ij} - u_i - v_j$
- $I = \{1, 2, \dots, n\}$ ist die Menge der Zeilenknoten
- $J = \{1, 2, \dots, n\}$ ist die Menge der Spaltenknoten
- $G(I, J)$ ist der vollständige, bipartite Graph mit Kantenmenge $\{(i, j) | i \in I, j \in J\}$.

Definition 5.4. Sei F ein Subgraph von $G(I, J)$.

- a) F heißt dual zulässig, falls es einen dual zulässigen Vektor (u, v) gibt, so dass $w_{ij} = 0$, für alle $(i, j) \in F$.
- b) F heißt primal zulässig, falls es einen primal zulässigen Vektor x gibt, so dass $x_{ij} = 0$, für alle $(i, j) \in F$.

Lemma 5.5. Ist F ein primal und dual zulässiger Subgraph von $G(I, J)$, dann sind die zugehörigen Vektoren (u, v) und x optimal für (AP) bzw. (AP_D).

Definition. Sei F ein Subgraph von $G(I, J)$. Ein Knoten hat **Valenz** d (bzgl. F), falls sein Knotengrad in F gleich d ist.

Definition 5.6. Ein Subgraph F von $G(I, J)$ heißt **Superwald** mit einer Menge von Wurzeln aus J , falls gilt:

- F ist ein aufspannender Wald
- Jede Komponente von F enthält genau eine Wurzel
- Jeder Knoten aus J , der keine Wurzel ist, hat Valenz 2.

Definition. Ein Superwald F lässt sich in den **Überschuss-Wald** F^S und den **Defizit-Wald** F^D partitionieren, mit

F^S = Vereinigung der Komponenten von F , deren Wurzeln Valenz ≥ 2 haben,

F^D = Vereinigung der Komponenten von F , deren Wurzeln Valenz 0 oder 1 haben.

Lemma 5.7. Ist F dual zulässiger Superwald mit $F^S = \emptyset$, dann ist der zugehörige Vektor (u, v) eine Optimallösung von (AP_D) und F enthält eine optimale Zuordnung.

AKP(C):

- (1) Bestimme einen dual zulässigen Superwald F_0 mit Dualvariablen (u^0, v^0) und setze $i = 0$.
- (2) Falls F_i dual zulässig und F_i^S leer ist, Stop („ F_i enthält eine optimale Zuordnung“).
- (3) Iteration i :
 - (3.1) Bestimme eine geeignete Kante (g, h) .
 - (3.2) Konstruiere mit Hilfe von (g, h) einen neuen Superwald F_{i+1} .
 - (3.3) Berechne neue Dualvariablen (u^{i+1}, v^{i+1}) .
 - (3.4) Setze $i = i + 1$ und gehe zu (2).

5.3.1 Bestimmung einer Startlösung

Bestimme u_i und v_j als Zeilen- und Spaltenminima. Erstelle Subgraph $F = \{(i, j) \mid w_{ij} = 0\}$. Entferne Kanten und zeichne Knoten aus J als Wurzeln aus, bis F ein Subgraph ist.

5.3.2 Korrektur des Superwaldes

Wähle die Kante (g, h) mit kleinstem w_{ij} , die einen I -Knoten im Überschuss-Wald und einen J -Knoten im Defizit-Wald verbindet und füge sie dem Superwald hinzu:

$$\delta = w_{gh} = \min \{w_{ij} \mid i \in I \cap F_m^S, j \in J \cap F_m^D\}$$

Die Superwald-Eigenschaft muss wiederhergestellt werden. Es können 4 Fälle auftreten:

5.3.3 Korrektur der Dualvariablen

F_m^* ist anschaulich die Komponente, die „umgehängt“ wurde.

Ist Fall 1 oder 2 eingetreten, gilt:

$$u_i = \begin{cases} u_i - \delta, & \text{falls } i \in F_m^* \\ u_i & \text{sonst,} \end{cases}$$

$$v_j = \begin{cases} v_j - \delta, & \text{falls } j \in F_m^* \\ v_j & \text{sonst.} \end{cases}$$

In Fall 3 und 4 sind $+$ und $-$ vertauscht.

Lemma 5.8. Falls die Teilwälder F_m^D und F_m^S dual zulässig sind, dann gilt $\delta_{m+1} \geq \delta_m$ und $\delta_{m+1} \geq -\delta_m$.

Lemma 5.9. Falls F_m^D und F_m^S dual zulässig sind, dann auch F_{m+1}^D und F_{m+1}^S .

Satz 5.10. Der AKP-Algorithmus arbeitet korrekt und löst das Zuordnungsproblem mit Zeitkomplexität $\mathcal{O}(n^3)$.