

Panikzettel EMS

Kevin Conrads

Version 1.1.1 - 13.03.2020

Inhaltsverzeichnis

1	Einleitung	3
1.1	Versionshinweis	3
2	Stabile Matchings	4
2.1	Das Stable Marriage Problem	4
2.2	Exkurs: Stable Roommate Problem	5
2.3	Algorithmen zum Lösen des Stable Marriage Problems	5
2.3.1	Seitensprung-Dynamik	6
2.3.2	Gale-Shapley Algorithmus	6
2.4	Kontext des Stable Marriage Problems: College Admission	8
3	Sortieren	9
3.1	Insertion Sort und Merge Sort	9
3.2	Selection Sort und Bubble Sort	9
4	Laufzeit	10
4.1	Laufzeitanalyse	10
4.2	O-Notation	10
5	Interval-Scheduling und Partitioning	12
5.1	Greedy Algorithmen	12
5.2	Cashier's Algorithmus	12
5.3	Interval Scheduling	12
5.3.1	Interval Scheduling Algorithmus	13
5.3.2	Optimaler Inverall Scheduling Algorithmus	13
5.4	Intervall Partitioning	14
5.4.1	Intervall Partitioning Algorithmus	14
5.5	Scheduling zur Minimierung von Verspätung	15
6	Graphentheorie	17
6.1	Der langweilige Teil - Definitionen	17
6.1.1	Grundlegende Notation	17
6.1.2	Wege, Kreise, Bäume	18
6.1.3	Darstellung von Graphen	18
6.2	Der spaßige Teil - Algorithmen	19
6.2.1	Breitensuche	19

6.2.2	Dijkstra	20
7	Minimale Spannbäume	22
7.1	Problemdefinition	22
7.2	Hilfreiche Eigenschaften von Bäumen und der Austauschatz	22
7.3	Algorithmen zum Finden eines MSTs	22
7.3.1	Kruskal	23
7.3.2	Prim	23
7.3.3	Key-Lemma	24
7.4	Clustering	25
7.4.1	Das Clustering Problem	25
7.4.2	Max-k-Spacing-Clustering Algorithmus	26
8	Einführung in Spieltheorie	27
8.1	Spieltheorie?	27
8.2	Spieltheorie!	27
8.2.1	Strategische Spiele	27
8.2.2	Nash Gleichgewichte	28
8.2.3	Routing-Spiele	29
8.2.4	Potenzialfunktion für Routingspiele	30
8.3	Kooperative Spiele	30
8.3.1	Facility Location Game	31
8.3.2	Der Core von Kooperativen Spielen	31
8.3.3	Owen's Lineares Produktionsspiel - Beschreibung	32
9	Networkflow und Project Selection	33
9.1	Netzwerkflüsse	33
9.2	Project Selection	33
9.2.1	Project Selection Problemformulierung	33
9.2.2	Precedence Graph und Hasse Diagramm	33
9.2.3	Min Cut Formulierung	34
10	Matching Markets	36
11	Komplexitätstheorie	36

1 Einleitung

Dieser Panikzettel für das Fach “Einführung in Management Science: Design und Analyse von Algorithmen”, gehalten im WS19/20 von Prof. Peis, ist ein freiwilliges Projekt, dass ich parallel zu meiner Tutortätigkeit in diesem Fach erstellt habe. Die Struktur folgt dabei in der Reihenfolge den Vorlesungsthemen, kann jedoch an sinnvollen Stellen davon abweichen.

Dieses Dokument hat *nicht* zum Ziel, alleinige Ressource zum Lernen und Verstehen des Vorlesungsstoffes zu sein, sondern eher begleitende bzw. ergänzende Rolle als “Quick-Reference” zu den Materialien aus der Vorlesung und Übung einzunehmen. Trotz größter Sorgfalt können Fehler, die Inhalt, Form, Notation, etc. betreffen, nicht ausgeschlossen werden. Es liegt daher in der Verantwortung des/der Leser/Leserin, sich im Zweifel mit den betreffenden Lehrmaterialien der Vorlesung auseinanderzusetzen. Außerdem hafte ich nicht für falsche Antworten in z.B. der Klausur, aufgrund der Informationen in diesem Dokument getätigt wurden.

Es sei außerdem auf das “Mathe-light” und “Pseudocode light” im Moodle Lernraum hingewiesen, dass wesentliche Inhalte der Veranstaltung noch einmal deutlicher aufzeigt.

1.1 Versionshinweis

Das Dokument ist noch im Aufbau und wird sich unter Umständen noch ein wenig verändern. Lob, Kritik und Änderungsvorschlägen sind ausdrücklich erwünscht und erreichen mich am besten durch eine Email an kevin.conrads@rwth-aachen.de. Es sind weitere Beispiele zu den vielen Definitionen geplant, falls jedoch auch Anwenden von Algorithmen bzw. Vorrechnen von Aufgaben gewünscht ist, bitte ebenso eine Email an mich schreiben.

Eine neue Version wird zum einen im Moodle-Lernraum veröffentlicht, kann aber auch [hier](#) heruntergeladen werden. Es empfiehlt sich ein regelmäßiger Check nach einer neuen Version (hiermit seien auch die “Studydrivler” begrüßt ;)).

Changelog

- 14.02.2020 - Version 1: Kapitel 1 - 8 der Vorlesung eingepflegt
- 16.02.2020 - Version 1.1: Kleinere Fehlerverbesserungen und Layout angepasst
- 13.03.2020 - Version 1.1.1: Kapitel 9 hinzugefügt

Hinweis: Das “Aussehen” dieses Dokumentes (z.B. die farbigen Boxen) für diesen Panikzettel sowie das Konzept “Panikzettel” an sich sind übernommen bzw. angelehnt an das [Panikzettel-Projekt](#). Dort (und auf der Website <https://panikzettel.philworld.de/>) finden sich zwar hauptsächlich Panikzettel für Informatikveranstaltungen, aber auch für manche BWL-Fächer z.B. für Entscheidungslehre. Es lohnt sich also, mal durch die PDFs zu klicken. Außerdem freuen sich die Leute hinter dem Projekt immer über eine Email als Dank für ihre Arbeit.

So, jetzt gehts aber los. Alles anschnallen, jeder nur ein Kreuz, Eltern haften für ihre Kinder.

2 Stabile Matchings

Dieses erste Kapitel der Veranstaltung setzt einige Kenntnisse der Graphentheorie voraus. Wer eine Auffrischung gebrauchen kann, dem sei das Mathe-Light im Moodle oder das Kapitel 6 als Einstieg empfohlen.

2.1 Das Stable Marriage Problem

Definition: Stable Marriage Problem

Das Stable Marriage Problem beschreibt, wie eine Menge von Männern und eine gleichgroße Menge von Frauen miteinander verlobt (*gematched*) werden sollen, so dass keiner einen Anreiz zu einem Seitensprung hat.

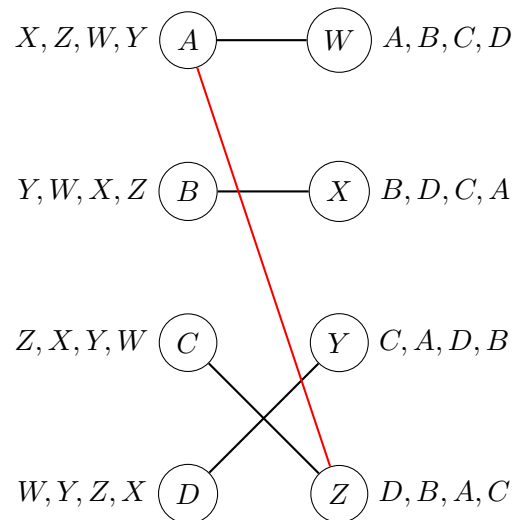
- Es gibt eine Menge an n Männern und eine Menge an n Frauen
- Jeder Mann listet die Frauen nach absteigender Präferenz
- Jede Frau listet die Männer nach absteigender Präferenz
- Gesucht ist ein Arrangement an n Hochzeiten, das heißt ein *perfektes Matching*
- Das Matching soll möglichst stabil sein: Keiner sollte einen Anreiz zu einem Seitensprung haben

Definition: Matchings in Graphen

Sei $G = (V, E)$ ein ungerichteter Graph

- Eine Teilmenge M der Kanten wird *Matching* genannt, wenn keine zwei Kanten in M einen gemeinsamen Endknoten haben.
- Die Kardinalität $|M|$ (Anzahl der Kanten in M) eines Matchings M ist nie größer als $\frac{|V|}{2}$
- Ein Matching M mit $|M| = \frac{|V|}{2}$ heißt *perfekt*

Graph-Beispiel für das Stable Marriage Problem:



Hier liegt ein instabiles Matching vor: $\{A, Z\}$ sind unzufrieden mit der Partnerwahl; sie finden sich gegenseitig besser als ihre jetzigen Partner und hätten damit Anlass für einen Seitensprung.

Definition: Bipartite Graphen

Ein ungerichteter Graph $G = (V, E)$ heißt *bipartit*, wenn sich die Knotenmenge V so in zwei Hälften aufteilen lässt, dass

- $V = L \cup R$ (jeder Knoten ist entweder in L oder in R),
- $L \cap R = \emptyset$ (kein Knoten liegt in beiden Mengen) und
- dass jede Kante einen Endknoten in L und einen in R besitzt.

Definition: Blockierende Kante und Stabiles Matching

Eine Kante u, v heißt *blockierend* (für ein Matching M), wenn der Knoten u den Knoten v gegenüber seinem jetzigen Partner präferiert und umgekehrt.

Ein Matching M ist *stabil*, wenn es keine blockierende Kante im Graphen gibt.

(Im Beispiel Graph oben ist die rot gezeichnete Kante $\{A, Z\}$ eine blockierende Kante für das Matching.)

Hier sollte deutlich werden, dass das Stable Marriage Problem äquivalent zum Stable Matching Problem in bipartiten Graphen ist.

Reduktion Das “Reduzieren” von Problem aus der Realität auf Probleme z.B. der Graphentheorie ist eine weit verbreitete Technik. Man folgt der Idee: Wenn man einen Algorithmus A hat, der ein (abstraktes) (Graphen-)Problem X löst, und ein reales Problem Y auf das Problem X reduzierbar (oder äquivalent) ist, dann kann man auch Problem Y mit dem Algorithmus A lösen.

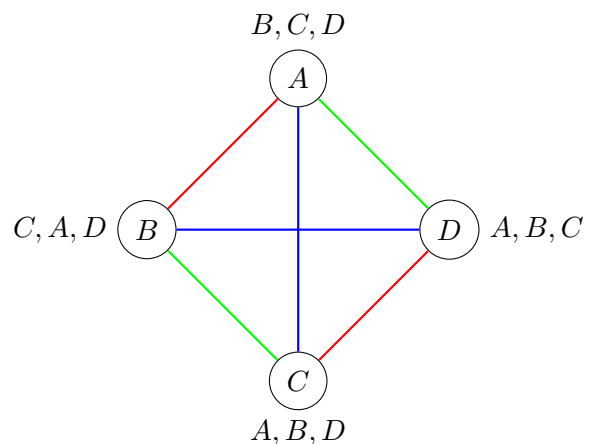
2.2 Exkurs: Stable Roommate Problem

Definition: Stable Roommate Problem

Das Stable Roommate Problem beschreibt, wie eine Menge von Personen anhand von Präferenzen in Paare aufgeteilt werden kann, sodass das resultierende Matching stabil ist.

- Gegeben: $2n$ Personen; jede Person sortiert die anderen $2n - 1$ Personen in einer Präferenzlist
- Gesucht: Eine Aufteilung der $2n$ Personen in n Paare, so dass das entsprechende (perfekte) Matching stabil ist.

Beim Stable Roommate Problem gibt es nicht immer ein stabiles perfektes Matching (siehe Beispiel rechts).



Alle Möglichkeiten an Matchings (rot, blau, grün) sind nicht stabil.

2.3 Algorithmen zum Lösen des Stable Marriage Problems

Es gibt zwei Algorithmen, die hier betrachtet werden: Die Seitensprung-Dynamik und Gale-Shapley-Algorithmus.

2.3.1 Seitensprung-Dynamik

Algorithmus:

Eingabe: Beliebiges Matching M

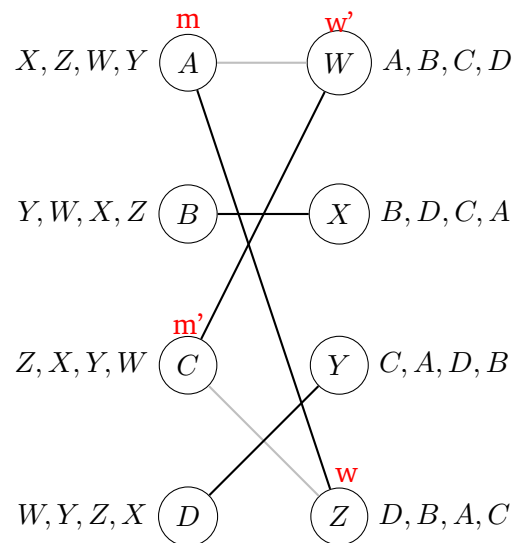
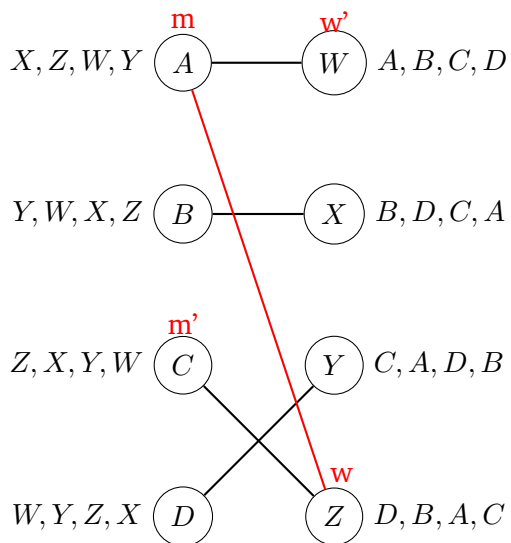
Ausgabe: Stabiles, perfektes Matching M^* .

WHILE (unzufriedenes Paar $\{m, w\}$ existiert)

Ersetze die Matchingkanten $\{m, w'\}$ und $\{m', w\}$ durch $\{m, w\}$ und $\{m', w'\}$

RETURN M^*

Folgender Graph zeigt eine Iteration innerhalb der **WHILE** Schleife:



$\{A, Z\}$ sind unzufrieden. Löse Verbindung $\{A, W\}$ und $\{C, Z\}$ auf und ...

verbinde $\{A, Z\}$ und $\{C, W\}$. Die aufgelösten Kanten sind hier grau gefärbt.

Satz: Korrektheit der Seitensprung-Dynamik

Die Seitensprung-Dynamik terminiert nicht für jede Eingabe. Das heißt es wird nicht für alle Eingaben ein stabiles Matching gefunden. (Wenn man das Beispiel oben weiterführt, landet man irgendwann wieder an der Ausgangsposition).

2.3.2 Gale-Shapley Algorithmus

Es gibt jedoch einen Algorithmus, der stets ein stabiles Matching findet:

Algorithmus: Gale-Shapley

Eingabe: Ein bipartiter Graph G mit einer Präferenzliste an jedem Knoten

Ausgabe: Stabiles, perfektes Matching M^* .

Zu Beginn sind alle Männer und Frauen ungematched

WHILE (es noch einen freien Mann gibt, der noch nicht allen Frauen einen Antrag gemacht hat)

 Wähle einen solchen freien Mann m

 Lasse diesen der besten Frau w auf seiner Liste einen Antrag machen

IF (Frau w ist noch frei)

 Verlobe m und w

ELSE

IF (w findet m besser als ihren aktuellen Verlobten m')

 Hebe die Verlobung von w' und m' auf

 Verlobe m und w

 Streiche w' von der Liste von m'

ELSE

 Streiche w von der Liste von m

Verheirate alle verlobten Paare

RETURN das resultierende Matching M^* von verheirateten Paaren

Man beachte, dass ein Mann m nur dann mit einer Frau w verlobt wird, wenn entweder w noch keinen Partner hat, oder wenn w denn Mann m besser als ihren bisherigen Verlobten m' findet. Im Umkehrschluss bedeutet dies, dass ein bereits verlobter Mann m' seine Verlobte dadurch verlieren kann und wieder in die Liste der Männer hinzugefügt wird, die noch nicht allen Frauen einen Antrag gemacht hat.

Ein perfektes Matching ist stabil, wenn entweder

- jeder Mann seine erste Wahl zur Partnerin hat, oder
- jede Frau ihre erste Wahl als Partner hat,

da so niemand den Anreiz zu einem Seitensprung hätte (dieser Fall tritt jedoch selten ein). Für Gale-Shapley gilt unabhängig davon:

Satz: Korrektheit von Gale-Shapley

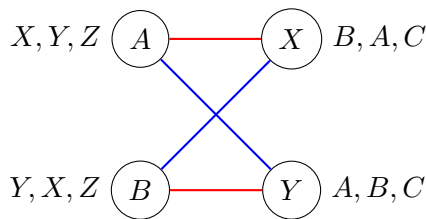
Gale-Shapley berechnet stets ein stabiles perfektes Matching.

Der Algorithmus hat eine Laufzeit von $\mathcal{O}(n^2)$, wobei n der Anzahl der Männer bzw. der Frauen entspricht.

Unter Umständen kann es mehrere stabile Matchings in einem Graphen geben, daher stellt sich die Frage, welches Gale-Shapley findet. Man stellt zunächst fest, dass Gale-Shapley die Männer bevorzugt, da diese aktiv auswählen und die Frauen nur passiv abwarten.

Definition: Zulässige Partnerin

Eine Frau w ist eine *zulässige Partnerin* von Mann m , wenn es ein stabiles perfektes Matching gibt, bei dem m und w ein Paar bilden.



- X und Y sind zulässige Partnerinnen von A
- X und Y sind zulässige Partnerinnen von B
- nur Z ist zulässige Partnerin von C . Jedes perfekte stabile Matching muss deshalb die Kante $\{C, Z\}$ enthalten

Definition: Mann-optimal

Ein Matching wird *Mann-optimal* genannt, wenn jeder Mann mit seiner bevorzugten zulässigen Partnerin zusammen ist



Satz: Gale-Shapley und Matchings

- Mann-optimale Matchings sind perfekt und stabil.
- Gale-Shapley liefert stets ein Mann-optimales Matching.
- Gale-Shapley weist jeder Frau den für sie schlechtesten zulässigen Partner zu

2.4 Kontext des Stable Marriage Problems: College Admission

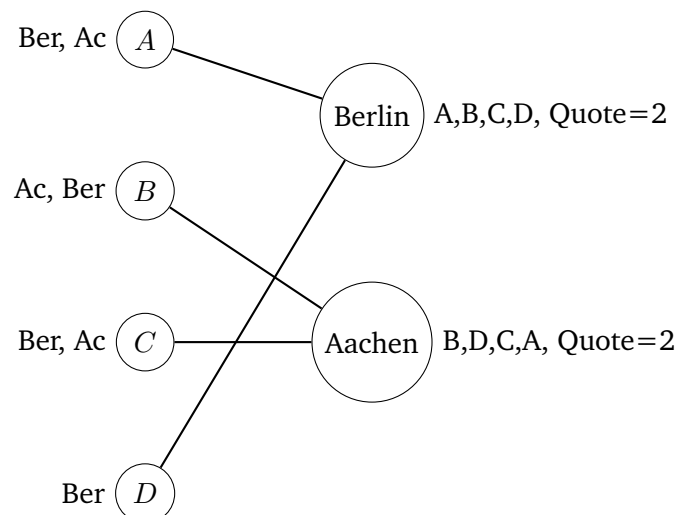
Definition: College Admission Problem

Es gibt Studierende und Universitäten mit jeweiligen Präferenzen, die einander zugeteilt werden sollen.

- Studierende bzw. Universitäten können als inakzeptabel deklariert werden
- Es gibt nicht unbedingt gleich viele Studierende wie Unis
- Jede Uni hat eine Quote, die die maximale Anzahl zugewiesener Studis festlegt

Eine Zuweisung (hier spricht man nicht mehr von Matching, wie es oben definiert wurde) ist zulässig, wenn jeder Studi einer akzeptable Uni zugewiesen wurde und jeder Uni nur akzeptable Studis zugewiesen wurde. Auch hier kann es analog "blockierende" Kanten geben.

Graph-Beispiel für das College Admission Problem:



3 Sortieren

Grundsätzlich in diesem Kapitel, sowie in allen folgenden, beginnen Arrays bzw. Listen immer mit dem Index 0. Das heißt, in einem Array A mit z.B. 3 Elementen, gibt es die Position $A[0]$, $A[1]$ und $A[2]$. Das erste Element eines n -elementigen Arrays A ist also $A[0]$ und das letzte $A[n - 1]$.

3.1 Insertion Sort und Merge Sort

Insertion Sort ist ein Sortiervorgehen, dass jedes Element eines Arrays an die richtige Stelle im bereits sortierten Teilarray einfügt ("insertion"). Merge Sort hingegen teilt das Array rekursiv in zwei Teillisten und "mergt" diese dann zu sortierten Teillisten zusammen.

Algorithmus: Insertion Sort

Eingabe: Array der Länge n

Ausgabe: Aufsteigend sortiertes Array A

```
FOR  $j = 1$  TO  $n - 1$ 
     $key = A[j]$ 
     $i = j - 1$ 
    WHILE ( $i > 0$  AND  $A[i] > key$ )
         $A[i + 1] = A[i]$ 
         $i = i - 1$ 
     $A[i + 1] = key$ 
RETURN  $A$ 
```

Algorithmus: Merge Sort

Eingabe: Array der Länge n

Ausgabe: Aufsteigend sortiertes Array A

```
IF ( $n = 1$ )
    RETURN  $A$ 

Sortiere  $A[1 \dots \frac{n}{2}]$  und  $A[\frac{n}{2} + 1 \dots n]$  rekursiv, wende also Merge Sort auf  $A[1 \dots \frac{n}{2}]$  und auf  $A[\frac{n}{2} + 1 \dots n]$  an.

Merge die beiden sortierten Teilarrays
RETURN  $A$ 
```

3.2 Selection Sort und Bubble Sort

Algorithmus: Selection Sort

Eingabe: Array der Länge n

Ausgabe: Aufsteigend sortiertes Array A

```
FOR  $j = 0$  TO  $n - 1$ 
     $i_{\min} = j$ 
    FOR  $i = j + 1$  TO  $n - 1$ 
        IF ( $A[i] < A[i_{\min}]$ )
             $i_{\min} = i$ 
    IF ( $i_{\min} \neq j$ )
        Tausche  $A[j]$  und  $A[i_{\min}]$ 
RETURN  $A$ 
```

Algorithmus: Bubble Sort

Eingabe: Array der Länge n

Ausgabe: Aufsteigend sortiertes Array A

```
FOR  $i = n$  DOWN1 TO 2
    FOR  $j = 0$  TO  $i - 1$ 
        IF ( $A[j] > A[j + 1]$ )
            Tausche  $A[j]$  und  $A[j + 1]$ 
RETURN  $A$ 
```

4 Laufzeit

4.1 Laufzeitanalyse

Die Laufzeitanalyse untersucht die Dauer, die ein Algorithmus oder Programm zum Lösen eines Problems bzw. einer Eingabe braucht. Da es verschieden schnelle Rechner in verschiedenen Anwendungsgebieten gibt, (z.B. Microcontroller vs Rechencluster), hat man sich darauf geeinigt, die Dauer eines Algorithmus nicht in Sekunden oder Minuten anzugeben, sondern in Abhängigkeit von der Länge der Eingabe. Dabei kommt es auch darauf an, wie “gut” die Eingabe codiert ist, aber das würde an dieser Stelle über das Thema hinaus gehen.

Es gibt drei häufig angewandte Analyse Methoden für Algorithmen:

- Worst-Case Analyse: Wie viele Rechenschritte bzw. wie viel Rechendauer benötigt der Algorithmus für die schlimmst-möglichen Eingabe? Diese Analyseart ist die gängigste und oft auch die aussagekräftigste.
- Average-Case Analyse: Wie viele Rechenschritte benötigt der Algorithmus im Durchschnitt für jede Eingabe?
- Best-Case Analyse: Wie viele Rechenschritte benötigt der Algorithmus für die best-mögliche Eingabe? Hier sollte man aufpassen, denn ein “guter” Algorithmus ist nicht nur auf ein paar wenigen Instanzen schnell bzw. effizient.

Diese Analysearten sind analog zu den nun folgenden mathematischen Beschreibungen in Form von unteren bzw. oberen Schranken.

4.2 O-Notation

Im folgenden wird meist von der Größe oder Länge der Eingabe als n gesprochen. Die Laufzeitanalyse untersucht dann, in welcher Größenordnung sich die Laufzeitfunktion $T(n)$ bei wachsender Eingabegröße n bewegt. Um dies geeignet (aber zugegeben nicht sofort verständlich) darzustellen, wird im Allgemeinen die “Big-O”-Notation verwendet.

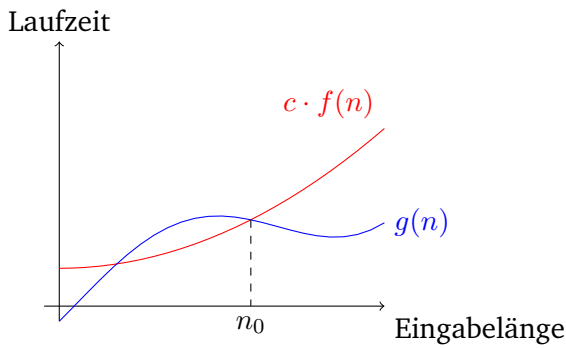
Definition: $\mathcal{O}(f)$

$\mathcal{O}(f)$ ist die Menge der Funktionen g , die nicht schneller wachsen als f . Wenn $g \in \mathcal{O}(f)$, ist $c \cdot f(n)$ eine obere Schranke für $g(n)$. Formal aufgeschrieben:

$$\mathcal{O}(f) := \{g \in \mathbb{R}_+^{\mathbb{N}} \mid \exists c > 0. \exists n_0 \in \mathbb{N}. \forall n \geq n_0. g(n) \leq c \cdot f(n)\}$$

Dabei ist c ein beliebiger Faktor, und n_0 ein Wert, ab dem für irgendein $n \geq n_0$ $c \cdot f(n)$ immer größer-gleich $g(n)$ ist. $\mathbb{R}_+^{\mathbb{N}}$ liest man als die Menge von Funktionen, die von \mathbb{N} (der Eingabelänge) nach den nicht-negativen reellen Zahlen \mathbb{R}_+ (die Laufzeit) auswerten.

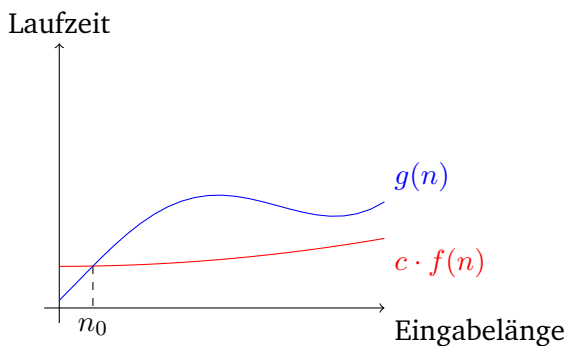
¹Das **DOWN TO** entscheidet sich von den bisherigen **FOR ... TO** nur dadurch, dass wir die Variable anstatt hochzuzählen, bei einem **DOWN TO** nach jeder Iteration der Schleife um eins runterzählen.



Definition: $\Omega(f)$

$\Omega(f)$ ist die Menge der Funktionen g , die nicht langsamer wachsen als f . Wenn $g \in \Omega(f)$, ist $c \cdot f(n)$ eine untere Schranke für $g(n)$. Formal aufgeschrieben:

$$\Omega(f) := \{g \in \mathbb{R}_+^{\mathbb{N}} \mid \exists c > 0. \exists n_0 \in \mathbb{N}. \forall n \geq n_0. g(n) \geq c \cdot f(n)\}$$

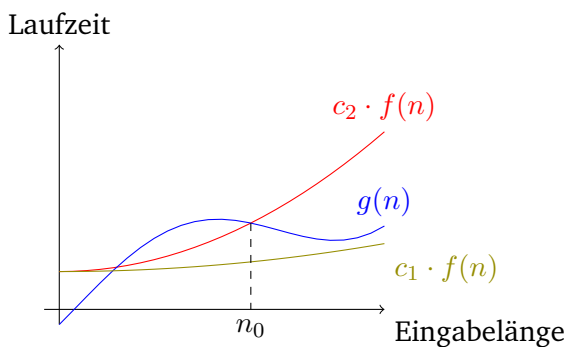


Definition: $\Theta(f)$

$\Theta(f)$ ist die Menge der Funktionen g , die nicht langsamer wachsen als f . Wenn $g \in \Theta(f)$, ist $c \cdot f(n)$ eine untere Schranke für $g(n)$. Formal aufgeschrieben:

$$\Theta(f) := \{g \in \mathbb{R}_+^{\mathbb{N}} \mid \exists c_1, c_2 > 0. \exists n_0 \in \mathbb{N}. \forall n \geq n_0. c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}$$

Alternativ gilt: $g \in \Theta(f) \Leftrightarrow g \in \mathcal{O}(f) \wedge g \in \Omega(f)$



Laufzeit einiger Algorithmen:

- Merge Sort: $\mathcal{O}(n \log(n))$, $\Theta(n \log(n))$
- Insertion Sort, Selection Sort: $\mathcal{O}(n^2)$, $\Omega(n)$

5 Interval-Scheduling und Partitioning

5.1 Greedy Algorithmen

Greedy (zu deutsch “gierig”) Algorithmen ist die Bezeichnung für solche Algorithmen, die versuchen, in jedem Schritt möglichst nah an die optimale Lösung heranzukommen und dabei Stück für Stück eine bessere Lösung generieren. Diese Art der Lösungsfindung funktioniert für einige Probleme gut (z.B. Scheduling, Partitioning), für andere jedoch nicht immer gut (z.B. Cashier’s Algorithmus).

5.2 Cashier’s Algorithmus

Der Cashier’s Algorithmus gibt für einen Geldbetrag, das mit Münzen verschiedener Münzwerte ausgezahlt werden soll, eine Menge von Münzen aus, die in Summe gleich dem Geldbetrag sind.

Algorithmus: Cashier’s Algorithmus

Eingabe: Betrag x , Münzwerte c_1, \dots, c_n

Ausgabe: Menge S an Münzen mit Gesamtwert x

Sortiere die n Münzwerte aufsteigend, sodass $c_1 \leq \dots \leq c_n$

$S = \{\}$

WHILE ($x > 0$)

 Sei k der größte Index mit $c_k \leq x$

IF (kein solches k existiert)

RETURN “keine Lösung”

ELSE

$x = x - c_k$

 Füge eine Münze von Wert c_k zu S hinzu

RETURN S

Satz: Nicht-Optimalität des Chashier’s Algorithmus

Der Cashier’s Algorithmus liefert *nicht* immer die optimale Lösung, z.B. würde der Algorithmus nur mit Münzen von Wert 3 und 5 bei einem Betrag von $x = 9$ “keine Lösung” ausgeben, obwohl drei 3-Münzen möglich sind.

5.3 Interval Scheduling

Beim Interval Scheduling geht es darum, Anfragen oder Jobs für eine Ressource so auszuwählen, dass sich die Jobs überlappungsfrei sind. Jeder Job i hat dazu eine Startzeit s_i und eine Endzeit f_i , oft kurz als (s_i, f_i) angegeben, wobei hier offene Intervalle verwendet werden (d.h., Job (1,3) und Job (3,5) sind überlappungsfrei). Das Ziel ist es dabei, möglichst viele kompatible Jobs auszuwählen.

5.3.1 Interval Scheduling Algorithmus

Es gibt u.A. die folgenden Auswahlregeln:

- (R1) frühester Startzeitpunkt s_i
- (R2) kürzester Belegungszeitraum $f_i - s_i$
- (R3) wenigste Überlappungen mit anderen Intervallen
- (R4) frühester Fertigstellungszeitpunkt f_i

Der grundsätzliche Algorithmus sieht wie folgt aus:

Algorithmus: Intervall Scheduling

Eingabe: Anfragen $M = \{1, \dots, n\}$ für Zeitintervalle $(s_1, f_1), \dots, (s_n, f_n)$

Ausgabe: Kompatible Teilmenge A der Menge M

$A = \{\}$

WHILE (Die Menge M nicht leer ist)

 Wähle eine Anfrage i aus M gemäß einer Auswahlregel (R_*)

 Füge i zu A hinzu

 Lösche alle mit i überlappenden Intervalle aus M , inklusive i

RETURN A

Es wird hier davon ausgegangen, dass innerhalb des Algorithmus nicht mehrere, sondern immer nur eine Auswahlregel angewandt wird.

5.3.2 Optimaler Inverall Scheduling Algorithmus

Der Earliest-Finish-Time-First Algorithmus, der der Auswahlregel (R4) von oben entspricht, liefert stets eine optimale Lösung. Die folgende Implementierung hat dabei die Laufzeit von $\mathcal{O}(n \cdot \log(n))$:

Algorithmus: Earliest-Finish-Time-First

Eingabe: Anfragen $M = \{1, \dots, n\}$ für Zeitintervalle $(s_1, f_1), \dots, (s_n, f_n)$

Ausgabe: Kompatible Teilmenge A der Menge M

Sortiere und re-indiziere die Intervalle, sodass $f_1 \leq \dots \leq f_n$

$A = \{\}$

FOR ($j = 1$ **TO** n)

IF (j ist kompatibel mit A)

RETURN A

An dieser Stelle nicht davon verwirren lassen, dass wir zwei verschiedene Implementierungen für den gleichen Algorithmus verwenden. Oft gibt es mehrere Algorithmen, die das gleiche Konzept umsetzen. Hier unterscheiden sich die beiden Varianten dadurch, dass bei der zweiten die Intervalle vorher noch sortiert werden, was bei der ersten nicht gemacht wird.

5.4 Intervall Partitioning

Das Intervall Partitioning Problem ist ähnlich zu dem Intervall Scheduling Problem, jedoch ist es hier das Ziel, *alle* Jobs so auf mehrere (nicht nur eine) Ressource aufzuteilen, dass alle Jobs auf einer Ressource kompatibel sind und wir möglichst wenig Ressourcen verwenden.

5.4.1 Intervall Partitioning Algorithmus

Analog zum Scheduling gibt es hier auch folgende Auswahlregeln:

- (R1) frühester Startzeitpunkt s_i
- (R2) kürzester Belegungszeitraum $f_i - s_i$
- (R3) wenigste Überlappungen mit anderen Intervallen
- (R4) frühester Fertigstellungszeitpunkt f_i

Der grundsätzliche Algorithmus sieht dabei wie folgt aus:

Algorithmus: Intervall Partitioning

Eingabe: Anfragen $M = \{1, \dots, n\}$ für Zeitintervalle $(s_1, f_1), \dots, (s_n, f_n)$

Ausgabe: Schedule S bei dem alle paarweise überlappenden Intervalle unterschiedlichen Ressourcen zugewiesen wurden

Sortiere und re-indiziere die Anfragen in M gemäß der ausgewählten Prioritätsregel (R_*). Nummeriere die Anfragen dann mit $1, \dots, n$ durch.

Setze $d = 0$

FOR ($i = 1$ **TO** n)

IF (i kann einer der Ressourcen d kompatibel zugewiesen werden)

 Weise i einer solchen Ressource zu

ELSE

 Stelle eine weitere Ressource $d + 1$ zur Verfügung

 Weise i dieser neuen Ressource $d + 1$ zu

$d = d + 1$

RETURN den resultierenden Schedule S

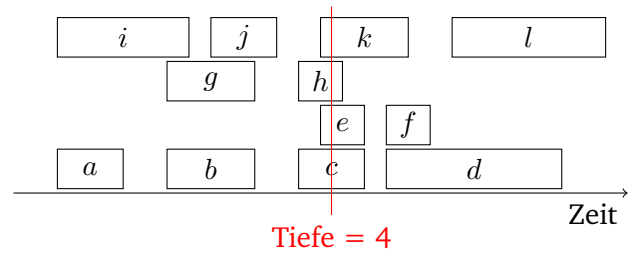
Satz: Optimaler Intervall Partitioning Schedule

Die Auswahlregel (R1)- frühester Startzeitpunkt - liefert stets einen optimalen Schedule. Sie wird auch *Earliest-Start-Time-First* genannt. Bei den anderen Auswahlregeln wird *nicht* immer ein optimale Schedule bestimmt.

Definition: Tiefe einer Instanz

Die *Tiefe* einer Instanz M (kurz: $\text{Tiefe}(M)$) ist die maximale Kardinalität einer Teilmenge A der Anfragen aus M , sodass es einen Zeitpunkt t gibt, der in allen Intervallen aus A enthalten ist.

Also gibt die Tiefe die Anzahl der Ressourcen, die wir *mindestens* für einen Schedule benötigen.



5.5 Scheduling zur Minimierung von Verspätung

Als letztes Thema in diesem Kapitel betrachten wir folgendes Problem: Es gibt wie vorher auch n Jobs, die aber wieder nur auf einer Ressource abgearbeitet werden sollen. Es können die Jobs nur nacheinander abgearbeitet werden, nicht gleichzeitig und alle Jobs können jederzeit begonnen werden, haben also keinen Startzeitpunkt wie zuvor. Jeder Job j hat eine Bearbeitungszeit p_j und eine Deadline d_j , zu der er möglichst fertig sein soll. Wenn ein Job j also zum Zeitpunkt s_j beginnt, ist er zum Zeitpunkt $f_j = s_j + p_j$, also Startzeit plus Bearbeitungszeit, fertig. Falls der Job nach seiner Deadline abgeschlossen wurde, hat er eine Verspätung von $f_j - d_j$, also Fertig-Zeitpunkt minus Deadline. Ist der Job vor seiner Deadline fertig geworden, weisen wir ihm die Verspätung 0 zu. Kurz gefasst hat jeder Job also die Verspätung $l_j = \max\{0, f_j - d_j\}$.

Das Ziel ist es nun, einen Schedule zu finden, sodass die maximale Verspätung, die irgendein Job j hat, zu minimieren (nicht die Summe der Verspätungen aller Jobs!). Mathematisch: Minimiere $L = \max_{j \in J} \{l_j\}$. Es ist hier also *besser*, wenn alle Jobs eine Verspätung von 2 Zeiteinheiten hätten, als wenn ein Job eine Verspätung von 4 Einheiten hat, aber alle anderen vor ihrer Deadline fertig geworden sind.

Wir haben erneut Auswahlregeln, aber da wir nun keine festen Startzeitpunkte haben, fällt "Earliest-Start-Time-First" weg:

- Shortest-Processing-Time-First: Schedule die Aufträge aufsteigend in der Reihenfolge ihrer Bearbeitungszeiten p_j , angefangen beim kürzesten
- Earliest-Deadline-First: Schedule die Aufträge aufsteigend in der Reihenfolge ihrer Deadlines d_j
- Smallest-Slack: Schedule die Aufträge aufsteigend in der Reihenfolge ihres "Slack" $d_j - p_j$

Satz: Optimalität von Earliest-Deadline-First

Für die Minimierung von Verspätung ist Earliest-Deadline-First die optimale Auswahlregel, die anderen beiden generieren nicht immer optimale Schedules

Algorithmus: Earliest-Deadline-First (Minimierung von Verspätungen)

Eingabe: Aufträge $M = \{1, \dots, n\}$ mit Processing Time p_j und Deadline d_j

Ausgabe: Schedule S

Sortiere und re-indiziere die n Aufträge in aufsteigender Reihenfolge der Deadlines d_j

Setze $t = 0$

FOR ($i = 1$ **TO** n)

 Schedule Auftrag j im Intervall $[t, t + p_j]$

$s_j = t$ // speichere Start- und

$f_j = s_j + p_j$ // Endzeitpunkt ab

$t = t + p_j$ // setze t auf den nächsten freien Zeitpunkt

RETURN den resultierenden Schedule S

6 Graphentheorie

6.1 Der langweilige Teil - Definitionen

6.1.1 Grundlegende Notation

Es folgen ein paar langweilige Definitionen.

Definition: Gerichteter Graph

Ein gerichteter Graph (auch: Digraph) G ist ein Paar (V, A) mit:

- Einer endlichen Menge V von Knoten ("Vertices")
- Einer Menge A von *geordneten* Paaren (Tupel) von Knoten, die (gerichtete) Kanten oder Bögen ("arcs") genannt werden.

Man verwendet gerichtete Graphen zur Darstellung asymmetrischer Beziehungen zwischen zwei Knoten, z.B. Einbahnstraßen

Definition: Ungerichteter Graphen

Ein ungerichteter Graph G ist ein Paar (V, E) mit:

- Einer endlichen Menge V von Knoten ("Vertices")
- Einer Menge A von *ungeordneten* Paaren von Knoten, die (gerichtete) Kanten ("edges") genannt werden.

Man verwendet ungerichtete Graphen zur Darstellung symmetrischer Beziehungen zwischen zwei Knoten, z.B. Freundschaften

Definition: Adjazent/benachbart

Sind u und v die beiden Endknoten einer [gerichteten] Kante $e = \{u, v\}$ [$a = (u, v)$], so ist v *adjazent* oder "*benachbart*" zu u .

Die Knoten u und v sind dann *inzident* zur [gerichteten] Kante $e = \{u, v\}$ [$a = (u, v)$].

Definition: Grad eines Knoten

Der *Grad* $\deg(u)$ eines Knoten ist die Anzahl der Kanten, die an ihn angrenzen. Schleifen werden doppelt gezählt

Bei gerichteten Graphen unterscheidet man zwischen der Anzahl der eingehenden Kanten $\deg^-(u)$ und der ausgehenden Kanten $\deg^+(u)$.

Definition: induzierter Teilgraph

Ein *Teilgraph* eines Graphen $G = (V, E)$ ist ein Graph $G' = (V', E')$ mit V' ist eine Teilmenge von V und E' ist eine Teilmenge von E .

Der Teilgraph $G' = (V', E')$ heißt *induzierter Teilgraph*, wenn E' alle Kanten aus E enthält, deren beide Endpunkte in V' sind.

Definition: Vollständig/Symmetrisch

Ein Graph G heißt *vollständig*, wenn jedes Knotenpaar durch eine Kante verbunden ist.

Ein gerichteter Graph heißt *symmetrisch*, wenn für jede Kante $(u, v) \in A$ auch die "Rückweg-Kante" $(v, u) \in A$.

Satz: Handshaking Lemma

In einem ungerichteten Graphen $G = (V, E)$ gilt:

$$\sum_{v \in V} \deg(v) = 2 \cdot |E|$$

Definition: Schleife

Eine Kante mit zwei identischen Endknoten wird *Schleife* oder *Schlinge* genannt.

6.1.2 Wege, Kreise, Bäume

Es folgen weitere, noch mehr langweilige Definitionen

Definition: Weg/Pfad

Ein *Weg* (oder *Pfad*) in einem [gerichteten] Graphen $G = (V, E)$ [$G = (V, A)$] von einem Knoten v zu einem Knoten w ist eine Sequenz an Knoten $v, x_1, x_2, \dots, x_n, w$, wobei je zwei aufeinanderfolgende Knoten durch eine [gerichtete] Kante verbunden sind.

Definition: Einfache Wege

Ein Weg ist *einfach*, wenn alle Knoten auf dem Weg paarweise unterschiedlich sind.

Definition: Zusammenhang (ungerichtete Graphen)

Ein Knoten w ist vom Knoten v aus *erreichbar*, wenn es einen Weg von v nach w gibt.

Ein ungerichteter Graph $G = (V, E)$ ist *zusammenhängend*, wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist.

Eine *Zusammenhangskomponente* eines ungerichteten Graphen G ist ein maximal zusammenhängender Teilgraph.

Definition: Zusammenhang (gerichtete Graphen)

Ein Knoten w ist vom Knoten v aus *erreichbar*, wenn es einen Weg von v nach w gibt.

Ein gerichteter Graph $G = (V, A)$ ist *stark zusammenhängend*, wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist.

Eine *starke Zusammenhangskomponente* eines gerichteten Graphen G ist ein maximal stark zusammenhängender Teilgraph.

Definition: Kreise

Ein *Kreis* in einem ungerichteten Graph $G = (V, E)$ ist ein Weg mit gleichem Start- und Endknoten

Ein Kreis ist *einfach*, wenn die Zwischenknoten paarweise verschieden sind.

Definition: Bäume und Wälder

Ein ungerichteter (Teil-)Graph $G = (V, E)$ wird *Baum* genannt, wenn G kreisfrei und zusammenhängend ist.

Ein kreisfreier ungerichteter Graph $G = (V, E)$ wird *Wald* genannt.

Satz: Relation von Bäumen und Wäldern

Ein Baum-Graph ist auch stets ein Wald, aber ein Wald-Graph ist nicht immer auch ein Baum, da ein Wald aus mehreren Bäumen bestehen kann.

Satz: Knotenzahl eines Baumes

Ein Baum mit n Knoten hat $n - 1$ Kanten

6.1.3 Darstellung von Graphen

Es gibt mehrere Möglichkeiten, Graphen zu kodieren, das heißt einfach verständlich für Menschen bzw. Computer aufzuschreiben:

Definition: Adjazenzmatrix

Graphen können mittels *Adjazenzmatrizen* repräsentiert und gespeichert werden:

- Die Adjazenzmatrix zum Graphen $G = (V, E)$ [$G = (V, A)$] mit Knotenmenge $V = \{1, \dots, n\}$ ist eine Matrix, deren Einträge entweder 0 oder 1 sind, und die n Zeilen und n Spalten hat, wobei
- der Eintrag in Zeile i , Spalte j genau dann 1 ist, wenn die beiden Knoten i und j durch eine ungerichtete Kante verbunden sind bzw. eine gerichtete Kante von i nach j verläuft.

Adjazenzmatrizen eignen sich gut für die Verarbeitung in Programmen.

Definition: Explizite Angabe

Die explizite Angabe meint, dass man einen Graphen $G = (V, E)$ durch explizites Aufschreiben der Mengen V und E [bzw. A] angibt.

Beispiel:

Definition: Skizze

Die Angabe durch Skizze meint, den Graphen aufzuzeichnen. Diese Art eignet sich sehr gut für Menschen.

Beispiel:

Es gibt außerdem noch die Möglichkeit, statt Adjazenzmatrizen auch Adjazenzlisten zu verwenden, die etwas effizienter im Speicherverbrauch sind (folgt eventuell noch an dieser Stelle).

6.2 Der spaßige Teil - Algorithmen

6.2.1 Breitensuche

Breitensuche ist für viele ("kleine") Graphenprobleme die Allzweckwaffe schlechthin. Anwendungen sind zum Beispiel die Bestimmung von Zusammenhangskomponenten, Topologische Sortierung, Kritische-Pfad-Analyse, kürzeste-Wege Suche (*nur* wenn alle Kanten Gewicht 1 haben) oder die Verifikation von Regular Safety Properties für endliche Transitionssysteme (*letzteres ist nicht Teil dieser Veranstaltung*). Die Idee ist, ausgehend von einem Startknoten s , nacheinander alle Knoten Schicht für Schicht hinzuzufügen.

Algorithmus: Breitensuche (BFS)

Eingabe: Ein Graph G sowie einen Startknoten s

Ausgabe: Knoten in den einzelnen Schichten ausgehen von Startknoten s

Markiere alle Knoten als “unbesucht”

Markiere Startknoten s als “besucht” und setze $L_0 = \{s\}$

$i = 0$

WHILE (es noch unbesuchte Nachbarn von Knoten in L_i gibt)

$L_{i+1} = \{\text{alle unbesuchten Nachbarn von } L_i\}$

Markiere die Knoten in L_{i+1} als “besucht”

$i = i + 1$

RETURN $L_k \forall k \in \{0, \dots, i\}$

6.2.2 Dijkstra

Der Dijkstra-Algorithmus steht der Breitensuche im Coolness-Faktor um nichts nach, und wird vor allem zum Lösen des kürzeste-Wege Problem benutzt:

Definition: Kürzeste-Wege Problem

- Gerichteter Graph $G = (V, A)$
- Kantenlänge $c(a) \geq 0$ auf allen Kanten $a \in A$
- Ein Startknoten $s \in V$ und einen Zielknoten $t \in V$

Gesucht ist ein kürzester Weg von s zu t .

Damit Dijkstra jedoch seine ganze Coolness entfalten kann, müssen ein paar Bedingungen für einen Graphen G gelten:

Der Algorithmus verwendet außerdem eine eigene Notation (weil er cool ist):

Satz: Voraussetzungen für Dijkstra

- Alle Knoten in $G = (V, A)$ sind von s aus erreichbar
- Es gibt weder Schleifen, noch parallele Kanten
- Es gibt keine negativen Kantengewichte $c(a)$ für $a \in A$

Die ersten beiden Voraussetzungen sind durch Preprocessing immer möglich herzustellen.

Definition: Notation für Dijkstra

- S : Alle Knoten $u \in V$, für die bereits ein kürzester $s-u$ -Weg berechnet wurde
- $d(u)$: Länge eines kürzesten $s-u$ -Weges (Distanz)
- $N(S)$: Nachbarn von S , d.h. diejenigen Knoten, die von einem Knoten S aus über eine Kante erreichbar sind

Wir kommen zum eigentlichen Algorithmus (der auch die kürzesten Wege ausgibt):

Algorithmus: Dijkstra-Algorithmus

Eingabe: Ein gerichteter Graph $G = (V, A)$ mit zugeh. Kantenkosten $c(a)$ und einen Startknoten s

Ausgabe: Den kürzesten Weg $p(v)$ vom Startknoten s zum Knoten v und seine Länge $d(v)$ für jeden Knoten $v \in V$

Initialisiere $S = \{s\}$ und $d(s) = 0$

WHILE ($S \neq V$)

FOR (jeden Nachbarn $n \in N(S)$)

 Berechne vorläufige Distanz $d'(n) = \min\{d(u) + c(u, n) | u \in S, (u, n) \in A\}$

 Wähle Knoten n mit minimaler vorläufiger Distanz $d'(n)$

 Füge n zu S hinzu

 Setze $d(n) = d'(n)$ und $p(n) = u$ // *Speichere den Vorgänger von n in $p(n)$ ab*

RETURN $d(v), p(v) \forall v \in V$

Es bietet sich an, die Zwischenschritte von Dijkstra in einer geeigneten Tabelle festzuhalten.

Satz: Korrektheit von Dijkstra

Zu jeder Zeit während des Algorithmus gilt, für einen Knoten $u \in S$, dass der Wert $d(u)$ der Länge eines kürzesten $s - u$ -Weges entspricht.

7 Minimale Spannbäume

7.1 Problemdefinition

Definition: Das Minimal-Spanning-Tree (MST) Problem

Gegeben ist ein ungerichteter zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantenkosten c_e für alle $e \in E$

Gesucht ist eine Kanten-Teilmenge $T \subseteq E$ mit minimalen Kosten $\sum_{e \in T} c_e$, sodass der Teilgraph $G[T] = (V, T)$ zusammenhängend ist.

Dabei gibt es stets eine optimale Lösung $T \subseteq E$, wodurch $G[T] = (V, E)$ ein Baum ist. Deswegen spricht man von Minimalen Spannbäumen.

7.2 Hilfreiche Eigenschaften von Bäumen und der Austauschsatz

Satz: Eigenschaften eines Baumes

Die folgenden vier Aussagen sind für einen ungerichteten Graphen $G = (V, E)$ äquivalent (d.h. jede Aussage impliziert alle anderen):

1. $G = (V, E)$ ist ein Baum
2. Je zwei Knoten in V sind durch genau einen Weg verbunden
3. $|T| = |V| - 1$ und der Graph $G = (V, E)$ ist zusammenhängend
4. $|T| = |V| - 1$ und der Graph $G = (V, E)$ ist kreisfrei

Definition: Abkürzende Schreibweise

Sei F eine Teilmenge von E . Weiter seien $e \in F$ und $f \in E \setminus F$. Wir bezeichnen mit:

- $F + e$ die Menge, die aus F entsteht, wenn man e hinzufügt,
- $F - f$ die Menge, die aus F entsteht, wenn man f entfernt

Definition: Austauschlemma

Sei (V, T) ein aufspannender Baum im ungerichteten Graphen $G = (V, E)$. Sei $e \in E$ eine Kante, die nicht in T liegt. Dann gilt:

- Der Teilgraph $(V, T + e)$ enthält genau einen Kreis C
- Für jede Kante f aus dem Kreis C gilt, dass $(V, T + e - f)$ ein aufspannender Baum ist.

7.3 Algorithmen zum Finden eines MSTs

Zum Finden eines MSTs betrachten wir im folgenden zwei Algorithmen, Kruskal und Prim. Kruskal wählt streng nach dem greedy-Prinzip die Kanten in der aufsteigenden Reihenfolge ihrer Kosten und prüft dabei, ob eine neue Kante ein Kreis verursachen würde. Prim hingegen baut den MST anhand von bereits gewählten Kanten mit Hilfe des Schnittes auf.

7.3.1 Kruskal

Algorithmus: Kruskal

Eingabe: Ein ungerichteter zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantenkosten $c(e) \forall e \in E$

Ausgabe: Eine kostengünstigste Teilmenge T der Kanten, sodass der Teilgraph $G[T] = (V, E)$ zusammenhängend ist

$T = \{\}$ // T enthält diejenigen Kanten, die der Algorithmus wählt

$E' = E$ // E' enthält die Kanten, die noch nicht betrachtet wurden

WHILE ($|T| < |V| - 1$)

 Wähle günstigste Kante $e \in E'$

 Entferne e aus E'

IF ($T + e$ kreisfrei)

 Füge e zu T hinzu

RETURN T

7.3.2 Prim

Für den Algorithmus von Prim benötigen wir vorher noch die Definition des Schnitts einer Knotenmenge

Definition: Schnitt

Sei $G = (V, E)$ ein ungerichteter Graph und S eine Teilmenge der Knoten. Die Menge aller Kanten aus E , die genau einen Endknoten in S besitzen, wird auch *Schnitt von S* oder *Schnitt(S)* genannt.

Satz: Schnitte und Kreise

Schnitte und Kreise haben stets eine gerade Anzahl an gemeinsamen Kanten

Algorithmus: Prim

Eingabe: Ein ungerichteter zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantenkosten $c(e) \forall e \in E$

Ausgabe: Eine kostengünstigste Teilmenge T der Kanten, sodass der Teilgraph $G[T] = (V, E)$ zusammenhängend ist

$T = \{\}$ // T enthält diejenigen Kanten, die der Algorithmus wählt

Wähle einen beliebigen Startknoten s und setze $S = \{s\}$

WHILE $(|T| < |V| - 1)$

 Wähle günstigste Kante e aus dem Schnitt von S

 Füge e zu T hinzu

 Erweitere S um den fehlenden Endknoten von e

RETURN T

7.3.3 Key-Lemma

Der folgende Abschnitt behandelt das Key-Lemma, welches für den Beweis der Korrektheit von Kruskal und Prim verwendet wird. Die diese Beweise jedoch über den Umfang dieses Horizont hinaus gehen, soll das Key-Lemma hier nur kurz erwähnt werden.

Definition: Key-Lemma

Eine Teilmenge F der Kantenmenge E ist *MST-geeignet*, wenn es einen MST gibt, der alle Kanten aus F enthält.

Sei F eine MST-geeignete Kantenmenge bzgl. eines Graphen $G = (V, E)$ mit Kantenkosten c . dann gilt für jede Knotenmenge U , die eine Zusammenhangskomponente des Graphen (V, F) induziert:

Ist e eine günstige Kante im Schnitt von U , dann ist $F + e$ MST-geeignet.

Satz: Korrektheit Kruskal und Prim

Kruskal und Prim liefern garantiert immer einen MST.

7.4 Clustering

7.4.1 Das Clustering Problem

Definition: Das Clustering Problem

Ziel: Gruppiere eine Menge an (vergleichbaren) Objekten, z.B. Websites oder Bilder.

Annahmen:

- Es gibt für zwei Objekte u und v einen Distanzwert $d(u, v)$, der die Ähnlichkeit bzw. Unterschiedlichkeit zwischen u und v beschreibt
- Das Distanzmaß ist symmetrisch, d.h. $d(u, v) = d(v, u)$ für alle Objektpaare $\{u, v\}$
- Die Objekte sollen in k Cluster eingeteilt werden

Um ein Merkmal der “Qualität” einer Aufteilung von Objekten in k Cluster (“Clustering”) zu haben, betrachten wir hier zunächst das *Max-k-Spacing-Clustering* Problem, um dann die Qualität eines Clustering zu definieren:

Definition: Max-k-Spacing-Clustering

Gegeben ist eine Menge $U = 1, \dots, n$ an Objekten, eine natürliche Zahl k und die Distanzen $d(u, v)$ für alle Objekte u, v aus U

Gesucht ist ein Clustering der Objekte in U in k nicht-leere Cluster, sodass die minimale Distanz zwischen zwei Objekten aus unterschiedlichen Clustern möglichst groß ist.

Das Ziel ist es also, die Cluster so festzulegen, dass die (kürzeste) Distanz zwischen zwei Cluster möglichst groß ist

Definition: Spacing

Sei $\mathcal{C} = C_1 \cup \dots \cup C_k$ ein Clustering der Objekte aus U in k paarweise disjunkte Cluster. \mathcal{C} wird dann auch ein k -Clustering der Menge U genannt.

Als *spacing* (oder “Qualität”) eines Clustering $\mathcal{C} = C_1 \cup \dots \cup C_k$ wird die minimale Distanz zweier Objekte aus unterschiedlichen Clustern bezeichnet, d.h.,

$$\text{spacing}(\mathcal{C}) = \min\{d(u, v) \mid u \in C_i, v \in C_j, \text{ für } i \neq j\}$$

7.4.2 Max-k-Spacing-Clustering Algorithmus

Algorithmus: Max-k-Spacing-Clustering Algorithmus

Eingabe: Menge U an n Objekten, natürliche Zahl k , Distanz $d(u, v)$ zwischen allen Knoten $u, v \in U$

Ausgabe: Eine Aufteilung der Objekte in U in k nicht-leere Cluster

Konstruiere einen Graphen mit n Knoten, die die Objekte in U repräsentieren und ohne Kanten.

// Der initiale Graph besteht somit aus n Zusammenhangskomponenten

WHILE (es mehr als k Zusammenhangskomponenten (Cluster) gibt)

 Wähle zwei Knoten aus unterschiedlichen Clustern, deren Distanz minimal ist

 Verbinde diese Knoten durch eine Kante

// Somit gibt es ein Cluster weniger

RETURN die k Cluster C_1, \dots, C_k

Satz: Äquivalenz des Algorithmus zu Kruskal

Der max-k-Spacing-Algorithmus entspricht genau Kruskals-Algorithmus, außer, dass er abbricht, sobald der Graph aus k Clustern besteht.

Alternativ kann man auch einen MST berechnen und dann die $k - 1$ teuersten Kanten entfernen.

Satz: Korrektheit des Clustering Algorithmus

Der Clustering-Algorithmus liefert garantiert immer eine Aufteilung in k Cluster mit maximalen Spacing

8 Einführung in Spieltheorie

8.1 Spieltheorie?

Spieltheorie ist ein Werkzeug mit dem viel Situationen des realen Lebens modelliert werden können. Spieltheorie eignet sich besonders dann, wenn mehrere Akteure den gleichen Regeln unterliegen und versuchen, den für sie bestmöglichen Ertrag zu erreichen (zum Beispiel Wegfindung in Verkehrsnetzwerken).

8.2 Spieltheorie!

8.2.1 Strategische Spiele

Definition: Modell Strategische Spiele

Strategische Spiele lassen sich folgendermaßen charakterisieren:

- Es gibt eine endliche Spielermenge $N = \{1, \dots, n\}$
- Jeder Spieler $i \in N$ verfügt über eine Strategiemenge X_i , also die Menge an Optionen, die der Spieler hat
- Jeder Spieler i wählt jeweils eine Strategie x_i aus X_i , ohne zu wissen, welche Strategie die übrigen Spieler wählen
- Wenn jeder Spieler $i \in N$ eine Strategie x_i gewählt hat, entsteht so eine Spielsituation (Profil) $x = (x_1, \dots, x_n)$, welche die Spieler unterschiedlich gut oder schlecht für sich einschätzen
- Diese Einschätzung drückt jeder Spieler $i \in N$ mittels seiner Bewertungsfunktion u_i aus, die einfach ausgedrückt jedem (möglichen) Profil x einen reellen Wert $u_i(x)$ zuweist
- Die Bewertungsfunktion u_i liefert also Aussagen darüber, welche Profile ein Spieler bevorzugt (und auch, welche Strategie ein Spieler möglicherweise wählt, um den gewünschten Spielausgang zu erreichen)

Hier nochmal absolute Klarheit über die verwendete Notation:

- X_i ist die Menge aller Strategien für einen Spieler i (Welche Optionen hab ich?)
- x_i ist eine bestimmte Strategie aus der Menge der Strategien für Spieler i
- $x = (x_1, \dots, x_n)$ ist ein Profil, eine Spielsituation, die durch Wahl der Strategien x_1, \dots, x_n der jeweiligen Spieler $1, \dots, n$ entsteht

Kurzgefasst werden strategische Spiel auch mit $G = (N, (X_i)_{i \in N}, (u_i)_{i \in N})$ notiert, also der Spielermenge N , der Strategiemenge X_i und der Bewertungsfunktion u_i für jeden Spieler $i \in N$.

Alles klar? Dann können wir ja weitermachen.

Definition: Kosten- vs. Nutzenspiele

Bei einem strategischen Spiel $G = (N, (X_i)_{i \in N}, (u_i)_{i \in N})$ kann die Bewertungsfunktion u_i je nach Kontext eine *Kostenfunktion* oder *Nutzenfunktion* sein, aber innerhalb eines Spiels immer nur eine Art von beiden.

Achtung! Kostenfunktionen wollen von den Spielern stets minimiert werden und Nutzenfunktionen stets maximiert werden. Um welche Art der Bewertungsfunktion es sich handelt, geht meist aus der Beschreibung des Spiels hervor. Die Kenntnis über den Unterschied über diese beiden Arten wird jedoch ab jetzt vorausgesetzt.

Satz: Umwandlung Kosten- in Nutzenspiel und umgekehrt

Jedes Kostenspiel (das heißt ein Spiel, wo die Bewertungsfunktionen u_i Kostenfunktionen sind) kann in ein Nutzenspiel umgewandelt werden (ebenso umgekehrt), in dem man die Kosten- bzw. Nutzenfunktion mit -1 multipliziert.

Außerdem wird angenommen, dass die Werte der Bewertungsfunktion stets nicht-negativ sind.

8.2.2 Nash Gleichgewichte

Definition: Reines Nash Gleichgewicht

Ein Profil $x = (x_1, \dots, x_n)$ ist ein reines Nash Gleichgewicht, wenn kein Spieler ein Anreiz hat, seine Strategie zu wechseln, unter der Annahme, dass die übrigen Spieler bei ihren aktuellen Strategien bleiben. Mathematisch ausgedrückt:

Ein Profil $x = (x_1, \dots, x_n)$ ist ein reines Nash Gleichgewicht für ein *Kostenspiel*, wenn für jeden Spieler $i \in N$ gilt:

$$u_i(x) \leq u_i(x'_i, x_{-i}) \text{ für alle } x'_i \in X_i$$

Bei Nutzenspielen wird das Ungleich-Zeichen entsprechend zu \geq .

x_{-i} bezeichnet das Profil x , aus der die Strategie von Spieler i "gestrichen" wurde (deswegen das $-i$). (x'_i, x_{-i}) ist dann das Profil, aus dem die alte Strategie x_i von Spieler i gestrichen wurde und eine andere x'_i hinzugefügt wurde.

Das Profil x ist also dann ein Nash Gleichgewicht, wenn für jeden Spieler i die Bewertungsfunktion im Profil x besser oder gleich gut ist, als wenn Spieler i eine alternative Strategie wählen würde.

Satz: Reines Nash Gleichgewicht in Strategischen Spielen

Nicht jedes Strategische Spiel hat ein reines Nash Gleichgewicht

8.2.3 Routing-Spiele

Definition: Routing-Spiele

Routing-Spiele sind eine Spezialklasse von strategischen Spielen. Sie bestehen aus:

- Netzwerk $G = (V, E)$
- Startknoten s_i und Endknoten t_i für jeden Spieler $i \in N = \{1, \dots, n\}$
- Kostenfunktion c_e für jede Kante $e \in E$, die jeder natürlichen Zahl x einen Wert $c_e(x)$ zuweist

Jeder Spieler i wählt als Strategie einen der möglichen $s_i - t_i$ -Pfade im Graphen G . Das Profil $P = (P_1, \dots, P_n)$ besteht dann aus den jeweiligen Pfaden P_i , den jeder Spieler i gewählt hat.

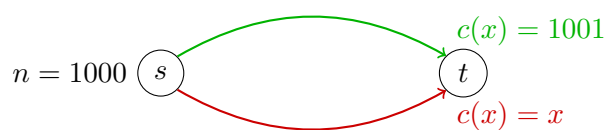
Weiter wird mit $I_e(P)$ die "Last", die auf Kante e liegt, bezeichnet und ist die Anzahl aller Spieler i , die die Kante e in ihrem Pfad P_i haben.

Satz: Beispiel von Pigou

Das nebenstehende Beispiel, benannt nach Pigou, zeigt, dass eigennütziges Verhalten von Spielern zu einer schlechteren Situation für alle führen kann:

- 1000 Spieler wollen von s nach t reisen
- Die grüne obere Route dauert für jeden Spieler, unabhängig von der Anzahl der Spieler, stets 1001 Zeiteinheiten
- Die Reisezeit auf der unteren Route wächst linear mit der Anzahl der Spieler, die diese Route nehmen. Jeder Spieler benötigt also x Zeiteinheiten.
- Da jeder Spieler eigensinnig denkt, nimmt jeder den roten Pfad, da die Zeit $x = 1$ niedriger ist als 1001 (Ein einzelner Spieler weiß ja nicht, welchen Weg die anderen wählen, darum nimmt er an, er ist der einzige, der den roten Pfad nimmt).

Graphbeispiel Pigou:



Definition: Soziales Optimum und Soziale Kosten

Das soziale Optimum eines Routingspiels ist das Profil, wo die Kosten aller Spieler aufsummiert minimal sind. Allgemein nennt man die Summe der Kosten aller Spieler auch *Soziale Kosten*.

Definition: Price of Anarchy

Der *Price of Anarchy (PoA)* (etwa: Preis der Eigensinnigkeit) ist das Verhältnis von den maximalen Sozialen Kosten eines Nash-Gleichgewichts zu dem optimalen sozialen Kosten.

In dem Graphbeispiel sieht man folgendes:

- Alle Spieler wählen die untere Route, und jeder Spieler hat Kosten von 1000 Zeiteinheiten. Die Sozialen Kosten betragen $1000 \cdot 1000 = 1.000.000$
- Das Soziale Optimum hingegen ist, wenn sich die Spieler gleichmäßig auf beide Routen aufteilen. Dann wären die sozialen Kosten $500 \cdot 1001 + 500 \cdot 500 = 750.500$
- Der PoA ist hier also $\frac{1.000.000}{750.500} \approx 1,33245$

Satz: PoA in Routing Spielen

Der PoA in Routing Spielen, wo die Kostenfunktion jeder Kante e linear ist, also von der Form $c_e(x) = a_e(x) + b_e$, ist niemals größer als $\frac{4}{3}$.

Das obige Beispiel ist ein extremer Fall und ziemlich nah an $\frac{4}{3}$, aber dennoch kleiner.

Satz: Gleichgewicht in Routing-Spielen

Das hier betrachtete Modell von Routing-Spielen geht auf Rosenthals “Congestion Games” zurück und wird daher auch das *Rosenthal-Modell* genannt.

Routing-Spiele im Rosenthal-Modell besitzen stets ein Gleichgewicht.

8.2.4 Potenzialfunktion für Routingspiele

Die folgende Potenzialfunktion ϕ wird für den Beweis des vorangegangenen Satzes verwendet. Der Beweis wird hier jedoch weggelassen.

Definition: Potenzialfunktion

Die folgende Potenzialfunktion Φ weist jedem Profil $P = (P_1, \dots, P_n)$ einen Wert $\Phi(P)$ zu:

$$\Phi(P) = \sum_{e \in E} \sum_{t=0}^{l_e(P)} c_e(t)$$

8.3 Kooperative Spiele

Im folgenden wird davon ausgegangen, dass die Spieler nicht eigennützig denken, sondern grundsätzlich bereit zur Kooperation sind, solange es ihnen einen Vorteil verschafft.

Definition: Modell kooperativer Spiele

- Es gibt eine endliche Menge $N = \{1, \dots, n\}$ an Spielern
- Zu jeder Teilmenge (*Koalition*) S der Spielermenge N gibt es einen Wert $v(s)$

Je nach Kontext gilt für den Wert $v(S)$:

- Kooperatives Kostenspiel: der Wert $v(S)$ beschreibt die Kosten, die allein durch die Spieler der Menge S verursacht werden
- Kooperatives Nutzenspiel: der Wert $v(S)$ beschreibt den Gewinn, der allein durch die Spieler der Menge S erwirtschaftet werden kann

8.3.1 Facility Location Game

Definition: Facility Location Game

Das Facility Location Game (FLG) ist ein kooperatives Kostenspiel

- n Kunden $N = \{1, \dots, n\}$ möchten an offene Einrichtungen (Facilities) angebunden werden
- Es gibt eine Menge \mathcal{F} an möglichen Standorten, an denen eine Einrichtung eröffnet werden kann
- Die Kosten für die Eröffnung der Einrichtung j betragen f_j Geldeinheiten
- Die Kosten um Kunde i an Einrichtung j anzubinden betragen $c_{i,j}$ Geldeinheiten
- Für jede Teilmenge S der Spielermenge N bezeichnet $v(S)$ die minimale Summe aus Eröffnungs- und Anbindungskosten, wenn alle Spieler der Koalition S an offene Einrichtungen angebunden werden.

Die Frage, die dann aufkommt, ist: Wie sollen die Gesamtkosten $v(N)$, also die Kosten, wenn alle Spieler angebunden werden sollen, auf faire Weise aufgeteilt werden?

8.3.2 Der Core von Kooperativen Spielen

Um die vorstehende Frage zu beantworten, betrachten wir das Konzept des *Core*. Teile die Gesamtkosten [bzw. Gesamtgewinn] $v(N)$ so unter den Spielern in N auf, dass keine Koalition S den Anreiz hat, sich von der großen Koalition N abzuspalten. Wir beschreiben die Aufteilung der Kosten als Vektor x mit n Einträgen, wo der i -te Eintrag den Anteil beschreibt, den Spieler i zahlen muss [bzw. erhält], und für den gilt:

- Summiert man die Einträge aller Spieler auf, so ergibt sich der Wert $v(N)$ und
- Summiert man die Einträge aller Spieler einer Koalition S auf, so ergibt sich ein Wert $x(S)$, der nicht größer [bzw. nicht kleiner] ist als $v(S)$ (sonst hätte die Koalition einen Anreiz, "ihr eigenes Ding" zu machen)

Definition: Core

Core eines kooperativen **Kostenspiels** (N, v) :

$$\text{Core}(N, v) = \{x \in \mathbb{R}_+^n \mid \sum_{i \in N} x_i = v(N) \wedge \sum_{i \in S} x_i \leq v(S) \forall S \subseteq N\}$$

Core eines kooperativen **Nutzenspiels** (N, v) :

$$\text{Core}(N, v) = \{x \in \mathbb{R}_+^n \mid \sum_{i \in N} x_i = v(N) \wedge \sum_{i \in S} x_i \geq v(S) \forall S \subseteq N\}$$

8.3.3 Owen's Lineares Produktionsspiel - Beschreibung

Definition: Owen's Lineares Produktionsspiels (OLP)

- n Firmen $N = \{1, \dots, n\}$ ziehen in Erwägung zu kooperieren um durch Zusammenfügen ihrer Ressourcen insgesamt höhere Gewinne zu erzielen
- Es gibt k verschiedene Artikeltypen, die produziert werden können
- Es gibt m verschiedene Typen an Ressourcen, die zur Produktion benötigt werden
- Zur Produktion einer Einheit von Artikel j werden $a_{r,j}$ Einheiten der Ressource r benötigt.
- Firma i verfügt über $b_{i,r}$ Einheiten der Ressource r .
- Eine Einheit von Artikel j erzeugt einen Gewinn von c_j Geldeinheiten
- Für jede Koalition S der Firmenmenge N kann mittels des folgenden linearen Programms der Zielfunktionswert $v(S)$ eines optimalen (fraktionalen) Produktionsplans bestimmt werden.

Definition: OLP: LP zur Bestimmung des Produktionsplans

$$\begin{aligned} \max \quad & \sum_{j=1}^k c_j x_j \\ \text{s.t.} \quad & \sum_{j=1}^k a_{rj} x_j \leq \sum_{i \in S} b_{ir} \quad \forall r \in \{1, \dots, m\} \\ & x \geq 0 \quad \forall j \in \{1, \dots, k\} \end{aligned}$$

Auch hier stellt sich die Frage, wie der Gewinn fair auf die einzelnen Firmen aufgeteilt werden kann. Dazu betrachten wir das *dual* LP zum obigen LP:

Definition: Berechnung eines Core-Vektors in OLP

Ein Core-Vektor für OLP kann wie folgt berechnet werden. Sei $b_r(S) = \sum_{i \in S} b_{i,r}$ für Rohstoff r und Koalition $S \subseteq N$.

1. Betrachte die LPs

$$P(S) : v(S) = \max_{x \geq 0} \left\{ \sum_{j=1}^k c_j x_j \mid \sum_{j=1}^k a_{rj} x_j \leq b_r(S) \quad \forall r = 1, \dots, m \right\}$$

und zugehörigen dualen LPs:

$$D(S) : w(S) = \min_{y \geq 0} \left\{ \sum_{r=1}^m b_r(S) y_r \mid \sum_{r=1}^m a_{rj} y_r \leq c_j \quad \forall j = 1, \dots, k \right\}$$

und nehme die optimale Lösung (y_1^*, \dots, y_m^*) für das LP $D(N)$.

2. Gebe jedem Spieler $z_i^* = \sum_{r=1}^m b_{ir} y_r^*$.

Dieser Vektor z^* liegt im Core.

9 Networkflow und Project Selection

9.1 Netzwerkflüsse

Netzwerkflüsse werden oft benutzt, wenn es darum geht, einen “Fluss” von Gütern, Fahrzeugen, Nachrichten usw. in einem “Netzwerk” bestehend aus z.B. Bahnhöfen, Häfen, Sendemasten, Straßen(-Kreuzungen) zu modellieren. Das folgende Kapitel über Project Selection baut auf diesem Konzept auf. Die Grundlagen für Maximale (Netzwerk-)Flüsse und den Zusammenhang mit dem minimalen Schnitt werden an dieser Stelle vorausgesetzt, können aber bei Bedarf z.B. in den Folien der Veranstaltung “Quantitative Methoden” oder auch in der Vorlesung zu Netzwerkflüssen in dieser Veranstaltung nachgeschlagen werden.

Der wichtigste Satz sei hier jedoch noch einmal erwähnt:

Satz: Max-Flow-Min-Cut

Für einen Graphen $G = (V, E)$ mit Quelle s und Senke t und einem Fluss f , sowie D_f der zugehörige Residualgraph von G , gilt:

Fluss f hat maximalen Wert \Leftrightarrow kein augmentierender Pfad im Residualgraphen D_f

Darüber hinaus gilt:

$$\max\{val(f) \mid f \text{ ist s-t-Fluss}\} = \min\{cap(S) \mid S \subseteq V \setminus \{t\}, s \in S\}$$

9.2 Project Selection

9.2.1 Project Selection Problemformulierung

Definition: Projekte mit Vorgängerbeziehungen

- Menge P Projekten, die potenziell bearbeitet werden können
- Jedes Projekt j ist ein Wert (Revenue) $w_j \in \mathbb{R}$ zugewiesen
- $w_j < 0$ werden als Kosten und $w_j \geq 0$ als Gewinn des Projekts $j \in P$ interpretiert
- *Precedence constraints* vom Typ $i \prec j$ zwischen Projekten i, j meint, dass wenn $i \prec j$ gilt, dann kann Projekt j nur dann ausgewählt werden, wenn auch Projekt i ausgewählt wird. Projekt i ist dann der *Vorgänger* (*predecessor*) von j genannt
- Eine Projektauswahl $S \subseteq P$ ist *zulässig*, wenn für jedes $j \in S$ auch alle Vorgänger von j ebenfalls zu S gehören

Das Ziel im Project Selection (Max Weight Closure) Problem ist es nun, aus einer Menge P an Projekten wie oben definiert eine zulässige Teilmenge $S \subseteq P$ zu wählen, sodass der Gewinn $\sum_{i \in S} w_i$ maximiert wird.

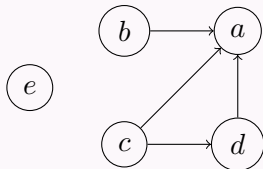
9.2.2 Precedence Graph und Hasse Diagramm

Wir machen uns nun die obige Definition zu nutze in dem wir das Project Selection Problem zu einem Graph Problem umwandeln (siehe auch [Reduktion](#)). Dafür brauchen wir zunächst den “precedence graph (PG)”, der die Projekte aus P als Knoten darstellt und genau dann eine gerichtete Kante (j, i) von einem Knoten j zu einem Knoten i enthält, wenn $i \prec j$ gilt.

Wichtig: $i \prec j$ meint, dass Projekt j nur dann ausgewählt werden kann, wenn auch i ausgewählt wird. Im PG wird jedoch eine Kante (j, i) dafür gezogen. Das heißt, dass die ausgehenden Kanten eines Knoten j auf alle Projekte zeigen, die ausgewählt werden müssen, damit auch j ausgewählt werden kann. Stehen zwei Projekte nicht in Relation durch \prec , so existiert auch keine Kante im PG zwischen diesen.

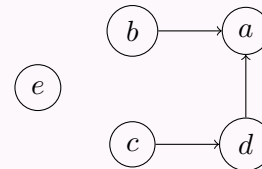
Beispiel: Project Selection - Precedence Graph

Sei $P' = \{a, b, c, d, e\}$ eine Projektmenge mit $a \prec b$, $a \prec c$, $a \prec d$ und $d \prec c$. Der PG von P' sieht dann wie folgt aus:



Beispiel: Project Selection - Hasse Diagramm

Sei $P' = \{a, b, c, d, e\}$ eine Projektmenge mit $a \prec b$, $a \prec c$, $a \prec d$ und $d \prec c$. Das HD von P' sieht dann wie folgt aus:



Wie man in dem PG links sieht, hängt c von d und a ab und d von a . Wir können die Kante (c, a) weglassen, weil die \prec -Relation transitiv ist: c kann nur ausgewählt werden, wenn wir a und d auswählen, aber d fordert auch die Wahl von a . Wenn also d auswählen wollen, ist a nach Definition auch auszuwählen. Dadurch können wir auch c auswählen. Die Kantenfolge $(c, d), (d, a)$ drückt also implizit auch (c, a) aus.

Die Version des PGs auf der rechten, die man auch *Hasse Diagramm (HD)* nennt, spart uns die Kante (c, a) , da diese keinen Mehrwert an Information liefert. Ein HD ist also ein PG ohne transitive Kanten.

9.2.3 Min Cut Formulierung

Um nun eine Gewinn-maximierende Auswahl an Projekten zu finden, erweitern wir die Konstruktion des Hasse Diagramms um weitere Komponenten zu einem Flussnetzwerk. Dazu verwenden wir die *min cut formulation*:

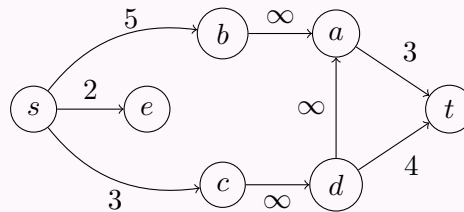
Definition: Min Cut Formulation

- Kapazität ∞ auf allen Kanten des **Hasse Diagramms**
- Füge eine Quelle s und eine Senke t hinzu
- Füge Kanten (s, j) mit Kapazität w_j für jedes Projekt j aus $P^+ := \{j \in P | w_j \geq 0\}$ - die Menge alle Projekte, die Gewinn bringen - hinzu.
- Füge Kanten (j, t) mit Kapazität $-w_j$ für jedes Projekt j aus $P^- := \{j \in P | w_j \geq 0\}$ - die Menge alle Projekte, die Verlust bringen - hinzu.
- Zur Vereinfachung der Notation ist $w_s = w_t = 0$

Angewendet auf unser vorheriges Beispiel P' , wobei nun auch die Revenues definiert werden, ergibt sich also folgendes Flussnetzwerk:

Beispiel: Project Selection: Flussnetzwerk

Sei $P' = \{a, b, c, d, e\}$ eine Projektmenge mit $a \prec b$, $a \prec c$, $a \prec d$ und $d \prec c$. Außerdem sei $w_a = -3$, $w_b = 5$, $w_c = 3$, $w_d = -4$, $w_e = 2$. Das Flussnetzwerk von P' in der min-cut-Formulierung sieht dann wie folgt aus:



Als letzter Schritt bleibt nur noch, aus diesem Flussnetzwerk eine zulässige Projektauswahl zu erhalten, die den Gewinn maximiert. Das erreichen wir, wie die Flusskonstruktion ja schon andeutet, in dem wir in einem Flussnetzwerk den Maximal Fluss bestimmen und daraus den minimalen Schnitt ablesen. Die optimale zulässige Projektauswahl ist dann die Menge S , die den Schnitt induziert (ohne den Hilfsknoten s). Für die Optimalität gilt dann:

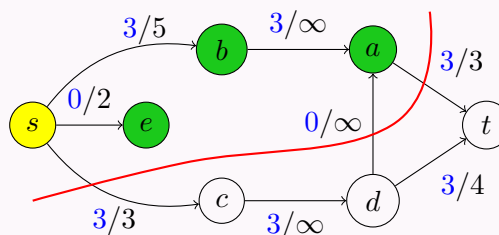
Satz: Optimale Projektauswahl via Min-Cut Berechnung

$S \subseteq P$ ist eine zulässige Projektauswahl mit maximalem Revenue $\sum_{j \in S} w_j \Leftrightarrow S \cup \{s\}$ ist ein s-t Schnitt in der Flussnetzwerk-Konstruktion mit minimaler Kapazität.

Warum funktioniert diese Konstruktion und berechnet eine optimale Projektauswahl? Dazu schaut man sich erneut das Beispiel an (in dem schon ein maximaler Fluss bestimmt wurde):

Beispiel: Project Selection: Flussnetzwerk

Sei $P' = \{a, b, c, d, e\}$ eine Projektmenge mit $a \prec b$, $a \prec c$, $a \prec d$ und $d \prec c$. Außerdem sei $w_a = -3$, $w_b = 5$, $w_c = 3$, $w_d = -4$, $w_e = 2$. Das Flussnetzwerk von P' in der min-cut-Formulierung und die zugehörige Berechnung des minimalen Schnittes sehen dann wie folgt aus:



In rot ist dabei der minimale Schnitt und in Grün die daraus folgende optimale Projektauswahl zu sehen.

Man kann sich den Fluss in diesem Netzwerk grundsätzlich als zu investierende Kosten vorstellen: Das Projekt c bringt 5 Geldeinheiten Profit, weswegen die Kante (s, c) mit Kapazität 5 existiert. Es kann c nur ausgewählt werden, wenn auch d ausgewählt wird (durch die Kante (c, d) ausgedrückt). Das Projekt d hingegen verursacht Kosten in Höhe von 4, dargestellt durch die Kante (d, t) mit Kapazität 4. Es fließen also Flusseinheiten von s zu gewinnbringenden Knoten und von verlustbringenden Knoten zu t . Da c uns 3 Profit bringt, wählen wir das Projekt temporär einmal aus. Um nun aber zu schauen,

ob die Abhängigkeiten von c eventuell insgesamt zu Verlust führen würden, versuchen wir also, den maximalen Fluss zu bestimmen, der über die Kante (s, c) laufen kann. Wenn die Kante voll ausgelastet sein sollte (wie oben im Beispiel), heißt das, dass alle drei Einheiten an Profit von c aus zu t fließen können, also das sämtlicher Profit von c durch die Abhängigkeiten von c “aufgebraucht” wurde, denn es kann ja nur von Projekten, die Verlust bringen etwas zu t fließen (oben 3 Einheiten über (d, t) , die ursprünglich von c kommen. Die Wahl von c würde also durch die Kosten von d nicht zu Profit führen.

Der Knoten b stellt dabei das gleiche Szenario dar, jedoch sind die Kosten durch die Wahl von b kleiner als der Profit durch b , weswegen hier $5 - 3 = 2$ Profit durch b verursacht werden. Als letztes bleibt das Projekt e , das keinerlei Abhängigkeiten hat, weswegen auch $2 - 0 = 2$ Profit durch die Wahl von e verursacht werden.

10 Matching Markets

... folgt zeitnah ...

11 Komplexitätstheorie

... folgt irgendwann ...