

BKA Zusammenfassung

Uni Hamburg - Sommersemester 2023

Inhaltsverzeichnis

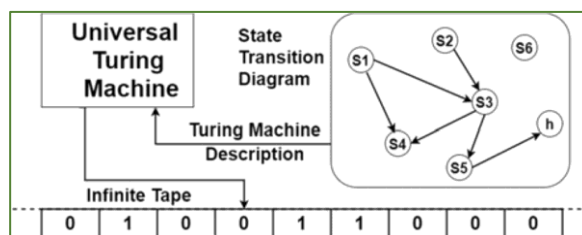
Church Turing These	3
Turingmaschinen	3
Konfigurationen	3
Sprachen	4
Mehrband-Turingmaschinen	4
Nichtdeterministische Turingmaschine	5
 Entscheidbarkeit	 5
Entscheidbare Sprachen	5
Unentscheidbare Sprachen	6
Unentscheidbarkeit von A_{TM}	6
Satz von Rice	7
 Reduktionen.....	 8
 Zeitkomplexität	 9
Laufzeitklassen	9
Die Klasse P.....	9
Die Klasse NP	9
Die Klasse NPC.....	10
Die Klasse co-NP	10
$P = NP$ oder $P \neq NP$?	10
 Wichtige Probleme in NPC.....	 11
Boolesche Formeln	11
SAT	11
3-SAT	11
Graphen.....	11
Clique.....	11

Independent Set	11
Vertex Cover	12
Traveling Salesman	12
Mengen	12
Set Cover.....	12
Subset-Sum.....	12
Bin Packing.....	13
Knapsack.....	13
Dreiecksungleichung	13
Approximation	14
Approximationsfaktor	14
Approximations-Schema	14
PTAS	15
FPTAS	15
Lokale Suche	16
Metropolis-Algorithmus	16
Simulated Annealing.....	17
Hopfield Networks.....	17
Maximum-Cut.....	18
k-Median.....	19

Church Turing These

Turingmaschinen

Turingmaschinen (TMs) sind simple theoretische Rechenmaschinen, die trotzdem komplex genug sind, um alle (berechenbaren) Probleme zu lösen. Es gibt jedoch Probleme, die nicht von TMs gelöst werden können (nicht berechenbare Probleme).



Formell sind TMs ein 7-Tupel $(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$:

- Q ist eine endliche Menge von Zuständen
- Σ ist das Eingabealphabet, $_ \notin \Sigma$ ($_$ ist das Blank-Symbol)
- Γ ist das Bandalphabet, $_ \in \Gamma$ und $\Sigma \subset \Gamma$
- δ ist die Übergangsfunktion: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{R, L\}$
- $q_0 \in Q$ ist der Startzustand
- $q_a \in Q$ ist der akzeptierende Zustand, $q_a \neq q_r$
- $q_r \in Q$ ist der ablehnende Zustand, $q_r \neq q_a$

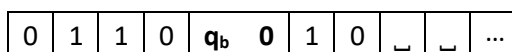
Zu Beginn einer Berechnung wird das Eingabewort w auf die ersten Zellen von links geschrieben, alle anderen Zellen bleiben leer (gefüllt mit $_$). Der Kopf der Turingmaschine startet am linken Ende des Bands, auf dem ersten Zeichen von w . Daraufhin werden kontinuierlich das momentane Bandzeichen sowie der aktuelle Zustand geprüft und basierend darauf wird ein neues Zeichen geschrieben, ein neuer Zustand betreten und der Kopf um eine Zelle nach links oder rechts bewegt. Wenn der Kopf versucht sich über die linke Kante hinaus zu bewegen, passiert nichts.

Die Berechnung endet, falls entweder Zustand q_a oder q_r betreten werden. Wenn die TM auf Wort w irgendwann den Zustand q_a erreicht, sagt man, dass die TM w **akzeptiert**. Umgekehrt, wenn der Zustand q_r erreicht wird, **lehnt** die TM das Wort w ab. Es ist aber auch möglich, dass die TM in einer Endlosschleife gefangen wird und nie terminiert. Zu wissen, ob eine TM terminieren wird oder nicht ist unberechenbar (\rightarrow Halteproblem).

Es gibt verschiedene Varianten von TMs, die jedoch alle gleichmächtig zu den regulären TMs sind, d.h. sie akzeptieren alle die gleiche Sprachklasse.

Konfigurationen

Man gibt den momentanen Zustand einer TM als Konfiguration an. Bspw. bedeutet 0110 q_b 010:



Die TM ist gerade im Zustand q_b

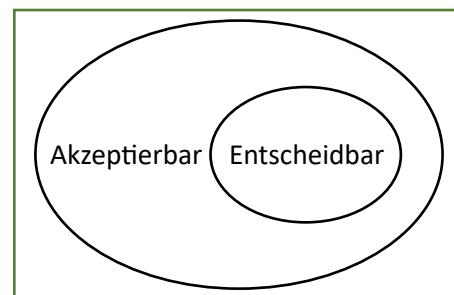
Der Kopf der TM ist gerade bei dieser 0

Sprachen

Alle Wörter, die eine TM M akzeptiert, bilden deren Sprache $L(M)$.

Akzeptierbar Eine Sprache ist (TM-)akzeptierbar oder rekursiv aufzählbar, wenn eine TM existiert, die alle ihrer Wörter akzeptiert (und keine der Wörter, die nicht in der Sprache sind)

Entscheidbar Eine Sprache ist entscheidbar, wenn sie akzeptierbar ist und außerdem auf jeder Eingabe in endlich vielen Schritten hält



$\text{Entscheidbar} \subset \text{Akzeptierbar}$

	Wort ist in Sprache	Wort ist nicht in Sprache
Akzeptierbar	TM muss terminieren (in q_a)	TM darf unendlich laufen
Entscheidbar	TM muss terminieren (in q_a)	TM muss terminieren (in q_r)

Wichtig: Eine Sprache A ist entscheidbar, wenn A und \bar{A} akzeptierbar sind.

Häufig gibt man anstelle eines Problems die Sprache aller Wörter an, die die Problemdefinition erfüllen. Wir unterscheiden dabei zwischen **Entscheidungsproblemen** und **Optimierungsproblemen**. Optimierungsprobleme sind entweder Maximierungs- oder Minimierungsprobleme (größten/kleinsten möglichen Wert finden).

Entscheidungsproblem	Optimierungsproblem
Ja/Nein	Optimalen Wert finden
Das Problem „Ist eine Zahl eine Primzahl“ kann in die Sprache aller Primzahlen umgewandelt werden.	Das Problem „Wie lang ist der längste Teilstring eines Wortes der außerdem ein Palindrom ist“ kann in die folgende Sprache umgewandelt werden:
$PRIM = \{n \mid n \text{ ist eine Primzahl}\}$	$PAL = \{\langle w, n \rangle \mid w \text{ ist ein Wort und } n \text{ die Länge des längsten Palindroms in } w\}$

Mehrband-Turingmaschinen

Eine Mehrband-TM hat eine konstante, feste Anzahl von k Bändern.

- Die Eingabe steht auf Band 1, alle anderen Bänder sind leer (gefüllt mit $_$)
- Es gibt einen Kopf pro Band, Köpfe können sich unabhängig voneinander bewegen
- Eine Konfiguration gibt nun als erstes den Zustand an, dann die Bandinhalte der Bänder mit einem Punkt über dem Zeichen, auf dem einer der Köpfe steht (mit Trennzeichen zwischen den Bändern, häufig #)

Die Übergangsfunktion sieht etwas anders aus:

$$\delta: Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$$

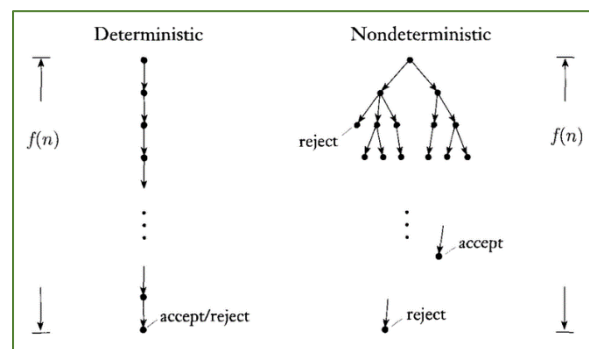
Es ist möglich, Mehrband-TMs auf normalen TMs zu simulieren (wie genau ist nicht so wichtig, relevant ist, dass es geht). Eine Mehrband-Maschine, die ein Problem in $T(n)$ entscheidet, kann von einer Einband-Maschine in $O(T(n)^2)$ simuliert werden.

Nichtdeterministische Turingmaschine

Bei nichtdeterministischen TMs ist es möglich, dass die Übergangsfunktion mehrere mögliche Übergänge produziert. Die Übergangsfunktion sieht dann so aus:

$$\delta: Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

Dabei entsteht ein Berechnungsbaum von möglichen Übergangspfaden. Eine NTM akzeptiert ein Wort, wenn mindestens ein Pfad in **accept** endet.



Auch NTMs sind komplexitätstechnisch gleichwertig zu deterministischen TMs, d.h. auch sie akzeptieren dieselbe Sprachklasse. Darum ist es möglich, NTMs mithilfe von DTMs zu simulieren.

Wichtig: Bei der Simulation wird der Berechnungsbaum **breadth-first** abgearbeitet (Breitensuche), da bei depth-first möglich ist, dass manche Berechnungspfade in unendlichen Schleifen hängen bleiben. Dies nennt man *kontrolliert laufen lassen*.

Eine NTM die ein Problem in $\mathbf{T(n)}$ entscheidet, kann von einer DTM in $2^{O(T(n))}$ simuliert werden.

Entscheidbarkeit

Entscheidbare Sprachen

Die folgenden Sprachen sind entscheidbar:

X generiert/akzeptiert Y:

$A_{DFA} = \{\langle B, w \rangle \mid B \text{ ist ein DFA der die Eingabe } w \text{ akzeptiert}\}$

$A_{NFA} = \{\langle B, w \rangle \mid B \text{ ist ein NFA der die Eingabe } w \text{ akzeptiert}\}$

$A_{REG} = \{\langle R, w \rangle \mid R \text{ ist ein regulärer Ausdruck der } w \text{ generiert}\}$

$A_{KFG} = \{\langle G, w \rangle \mid G \text{ ist eine kontextfreie Grammatik die } w \text{ generiert}\}$

Die Sprache von X ist leer:

$E_{DFA} = \{\langle A \rangle \mid A \text{ ist ein DFA und } L(A) = \emptyset\}$

$E_{KFG} = \{\langle G \rangle \mid G \text{ ist eine kontextfreie Grammatik und } L(G) = \emptyset\}$

Die Sprachen von X und Y sind gleich:

$EQ_{DFA} = \{\langle A, B \rangle \mid A \text{ und } B \text{ sind DFAs und } L(A) = L(B)\}$

Unentscheidbare Sprachen

Die folgenden Sprachen sind unentscheidbar, wobei die meisten versuchen, Aussagen über Turingmaschinen treffen:

$$EQ_{KFG} = \{\langle G, H \rangle \mid G \text{ und } H \text{ sind kontextfreie Grammatiken und } L(G) = L(H)\}$$

$$ALL_{KFG} = \{\langle G \rangle \mid G \text{ ist eine kontextfreie Grammatik und } L(G) = \Sigma^*\}$$

$$A_{TM} = \{\langle M, w \rangle \mid M \text{ ist eine TM und } M \text{ akzeptiert } w\}$$

$$E_{TM} = \{\langle M \rangle \mid M \text{ ist eine TM und } L(M) = \emptyset\}$$

$$EQ_{TM} = \{\langle M, M' \rangle \mid M \text{ und } M' \text{ sind TMs und } L(M) = L(M')\}$$

$$REG_{TM} = \{\langle M \rangle \mid M \text{ ist eine TM und } L(M) \text{ ist eine reguläre Sprache}\}$$

$$HALT_{TM} = \{\langle M, w \rangle \mid M \text{ ist eine TM und hält auf } w\}$$

Über Cantors Diagonalisierungsverfahren wissen wir, dass die Menge aller Sprachen *nicht abzählbar unendlich*, aber die Menge aller TMs nur *abzählbar unendlich* ist. Es gibt also mehr Sprachen als TMs, was bedeutet, dass Sprachen existieren, die von keinen TMs akzeptiert werden.

Wir wissen: eine Sprache A ist entscheidbar, wenn A und \bar{A} akzeptierbar sind. Wenn A nicht entscheidbar ist, ist also entweder A oder \bar{A} nicht akzeptierbar (oder beide!).

Unentscheidbarkeit von A_{TM}

Hier ist ein Beweis, der zeigt, warum A_{TM} nicht entscheidbar ist:

Wir gehen davon aus, dass A_{TM} entscheidbar ist und erhalten einen Widerspruch. H sei eine TM, die A_{TM} entscheiden soll:

$$H(\langle M, w \rangle) = \begin{cases} \text{accept,} & \text{falls } w \text{ von } M \text{ akzeptiert wird} \\ \text{reject,} & \text{falls } w \text{ von } M \text{ abgelehnt wird} \end{cases}$$

Wir definieren nun eine neue TM D , die H nutzt:

$$D(\langle M \rangle) = \begin{cases} \text{accept,} & \text{falls } \langle M \rangle \text{ von } M \text{ abgelehnt wird} \\ \text{reject,} & \text{falls } \langle M \rangle \text{ von } M \text{ akzeptiert wird} \end{cases}$$

bzw.

$$D(\langle M \rangle) = \begin{cases} \text{accept,} & \text{falls } H(\langle M, \langle M \rangle \rangle) \text{ ablehnt} \\ \text{reject,} & \text{falls } H(\langle M, \langle M \rangle \rangle) \text{ akzeptiert} \end{cases}$$

Jetzt entsteht ein Widerspruch, wenn D mit $\langle D \rangle$ aufgerufen wird:

$$D(\langle D \rangle) = \begin{cases} \text{accept,} & \text{falls } \langle D \rangle \text{ von } D \text{ abgelehnt wird} \\ \text{reject,} & \text{falls } \langle D \rangle \text{ von } D \text{ akzeptiert wird} \end{cases}$$

D wird also gezwungen, genau das Gegenteil von sich zu tun, was zu einem Widerspruch führt. Dadurch kann D nicht existieren, und H auch nicht. Es kann also keine TM existieren, die A_{TM} entscheidet.

Satz von Rice

Der Satz von Rice sagt, dass alle **nicht-trivialen Eigenschaften** einer (erkennbaren) Sprache **nicht entscheidbar** sind. Der Satz kann also angewendet werden, um zu zeigen, dass eine Sprache unentscheidbar ist (sagt aber nichts über Erkennbarkeit aus!).

Eine Eigenschaft ist trivial, wenn entweder für alle oder gar keine erkennbaren Sprachen gilt. Für eine nicht triviale Eigenschaft existiert also mindestens eine erkennbare Sprache, die diese Eigenschaft hat, und mindestens erkennbare Sprache, die diese nicht hat. Es gibt kein allgemeines Verfahren mit dem bestimmt werden kann, ob eine erkennbare Sprache eine nicht triviale Eigenschaft besitzt.

Triviale Eigenschaften sind immer entscheidbar, nicht-triviale Eigenschaften sind nicht entscheidbar.

Für eine Eigenschaft P sind zwei Bedingungen zu erfüllen, dass sie als nicht-trivial gilt:

1. Es existiert mindestens eine TM M_{Wahr} deren erkannte Sprache die Eigenschaft P hat und mindestens eine TM M_{Falsch} deren erkannte Sprache nicht die Eigenschaft P hat.
2. Wenn zwei TMs M_1 und M_2 die gleiche Sprache erkennen ($L(M_1) = L(M_2)$) muss gelten:
 - a. Entweder beide Sprachen besitzen die Eigenschaft P
 - b. Oder keine Sprache besitzt die Eigenschaft P

Beispiel 1:

Eigenschaft: Die Sprache einer TM hat mindestens 10 Wörter

Sprache: $L_1 = \{\langle M \rangle \mid L(M) \text{ hat mindestens 10 Wörter}\}$

- **Bedingung 1 gilt:** es existiert eine TM M_{Wahr} die mehr als 10 Wörter hat und M_{Falsch} die weniger als 10 Wörter hat
- **Bedingung 2 gilt:** Wenn die Sprache von zwei TMs gleich ist, müssen sie dieselbe Anzahl von Wörtern haben, woraus folgt, dass beide entweder in der Sprache sind oder nicht

⇒ Hier kann man den Satz von Rice anwenden

⇒ Damit ist L_1 **unentscheidbar**

Beispiel 2:

Eigenschaft: Die TM hält nach höchstens 17 Schritten

Sprache: $L_2 = \{\langle M \rangle \mid M \text{ hält auf jeder Eingabe nach höchstens 17 Schritten}\}$

- **Bedingung 1 gilt:** es existiert eine TM M_{Wahr} die in 10 Schritten stoppt und M_{Falsch} die in 20 Schritten stoppt (10 und 20 sind willkürlich gewählt)
- **Bedingung 2 gilt nicht:** Es kann zwei TMs geben die dieselben Sprachen erkennen, wobei aber womöglich eine in 15 Schritten und die andere in 30 Schritten terminiert.

⇒ Hier kann man den Satz von Rice **nicht anwenden**

Reduktionen

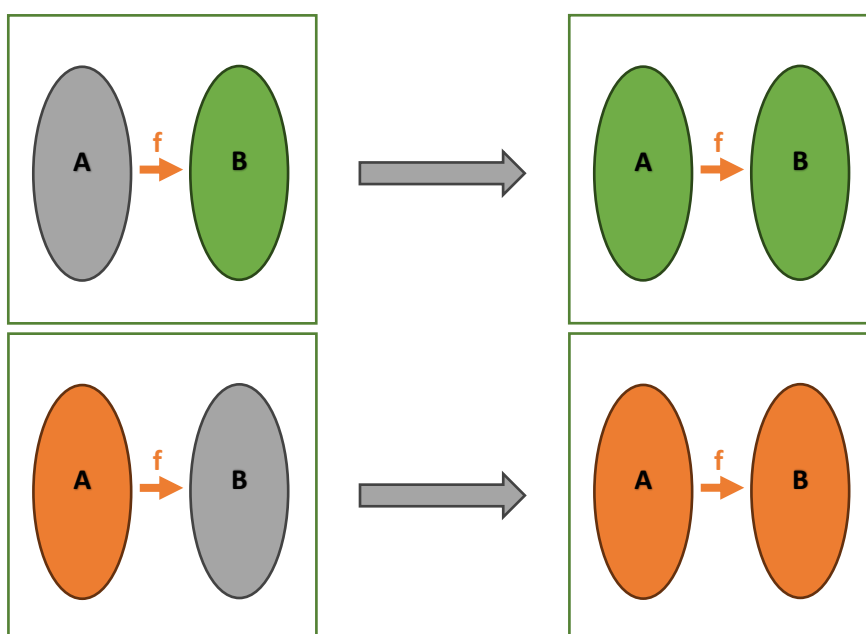
Eine Reduktion überführt ein Problem A in ein anderes Problem B. Danach kann die Lösung von B benutzt werden, um A zu lösen. Geschrieben wird dies so: $A \leq B$ (heißt, A wird auf B reduziert). Damit so eine Reduktion existieren kann, muss eine TM existieren, die diese berechnet, d.h. die Reduktion muss eine *berechenbare* Funktion sein.

Reduktionen können auf zwei Arten angewendet werden:

- Wenn B **entscheidbar** und A auf B reduzierbar ist, ist A auch **entscheidbar**
- Wenn A **unentscheidbar** und A auf B reduzierbar ist, ist B auch **unentscheidbar**

Dasselbe gilt analog auch für **Erkennbarkeit**:

- Wenn B **erkennbar** und A auf B reduzierbar ist, ist A auch **erkennbar**
- Wenn A **unerkenbar** und A auf B reduzierbar ist, ist B auch **unerkenbar**



Formell ist eine Reduktion eine berechenbare Funktion f , die eine Instanz eines Problems A auf die Instanz eines anderen Problems B abbildet. f heißt dann *Reduktionsfunktion*:

$$f: \Sigma^* \rightarrow \Sigma^*$$

$$\forall w \in \Sigma^*: w \in A \Leftrightarrow f(w) \in B$$

Wenn f jede Instanz in Polynomialzeit reduziert, ist f eine *Polynomialzeitreduktion*. Diese Art von Reduktion wird so angegeben: $A \leq_p B$

Leider gibt es kein gängiges Verfahren, um Reduktionen zu bestimmen. Hierbei muss man kreativ sein und eine gute Intuition für die Probleme und bekannte Reduktionen haben. Ich kann aber den folgenden YouTube Kanal empfehlen. Dort sind sehr gute Videos zur theoretischen Informatik, darunter auch Reduktionen:

<https://www.youtube.com/watch?v=-5TAx4tZlro&list=PLyITVsqZiRXMqnOcAhCYCzGGEhhbfe-xV>

Zeitkomplexität

Die Laufzeit (Zeit-Komplexität) einer TM ist die Anzahl der Berechnungsschritte, die bei einer Berechnung gemacht werden. Die Laufzeit wird immer in Abhängigkeit der Eingabegröße n angegeben.

- Best Case Laufzeit** Beste Laufzeit über alle Eingaben der Länge n
- Worst Case Laufzeit** Längste Laufzeit über alle Eingaben der Länge n
- Average Case Laufzeit** Durchschnittliche Laufzeit über alle Eingaben der Länge n

Wir betrachten aber eigentlich nur die **Worst Case Laufzeit**.

Laufzeitklassen

Hier sind die wichtigsten Laufzeitklassen. Die Laufzeit wird über die Laufzeitfunktion f angegeben. $f(n)$ gibt jeweils die Worst Case Laufzeit für eine Eingabegröße n an.

Notation	Bezeichnung	Beispiele
$f \in \mathcal{O}(1)$	f wächst konstant	$2, 6, \pi, 1.000.000$
$f \in \mathcal{O}(\log(n))$	f wächst logarithmisch	$\log(n), \log(n^2)$
$f \in \mathcal{O}(\sqrt{n})$	f wächst wie die Wurfelfunktion	$\sqrt{n}, \sqrt[3]{n}, \sqrt[40]{5n}$
$f \in \mathcal{O}(n)$	f wächst linear	$n, 4n, 10n + 7$
$f \in \mathcal{O}(n \log(n))$	f hat Wachstum $n \log(n)$	$n \log(n), \log(n!)$
$f \in \mathcal{O}(n^m)$	f wächst polynomiell (für $m > 1$)	$4n^2, n^3 + 4n, n^{1.000}$
$f \in \mathcal{O}(m^n)$	f wächst exponentiell (für $m > 1$)	$1.1^n, 10^n$
$f \in \mathcal{O}(n!)$	f wächst faktoriell	$n!, 4n! + n^2$

Die Klasse P

P ist die Klasse aller Sprachen, die von einer **deterministischen 1-Band TM** in **polynomieller Zeit** entschieden werden können. Diese Probleme lassen sich noch realistisch (von realistischen Computern in realistischer Zeit) lösen.

- Enthält keine unentscheidbaren Probleme
- Enthält keine Probleme, die nur von nichtdeterministischen oder Mehrband-TMs in polynomieller Zeit gelöst werden können

Wichtig: Die Laufzeit hängt stark von der **Codierung** ab. Eine Codierung kann zu einer polynomiellen Laufzeit führen, eine andere zu einer exponentiellen. Wir nehmen aber immer realistische Codierungen, d.h. keine unäre Codierung.

Die Klasse NP

NP ist die Klasse aller Probleme, die in Polynomialzeit verifiziert werden können. Wir haben zwei Möglichkeiten zu prüfen, ob eine Sprache A in NP liegt:

1. Es existiert ein **Polynomialzeitverifizierer** zu A. Ein Verifizierer für eine Sprache A ist ein Algorithmus, der als Eingabe eine Instanz w und ein Zertifikat (=Lösung) c (häufig $\langle w, c \rangle$) nimmt und zurückgibt, ob die Lösung korrekt ist. Dies muss in Polynomialzeit geschehen.
2. Es existiert eine **nichtdeterministische TM** die A in polynomieller Zeit entscheidet. Hier kann man häufig nichtdeterministisch eine Lösung „raten“ und dann überprüfen.

Die Klasse NPC

Probleme in NPC (oder auch NP-Vollständig) sind die „schwersten“ Probleme in NP. Jedes Problem in NP (und NPC) kann auf jedes Problem in NPC reduziert werden.

Um zu zeigen, dass eine Sprache B in NPC liegt, müssen zwei Bedingungen gelten:

1. $B \in \mathcal{NP}$
2. Für jede Sprache $A \in \mathcal{NP}$ gilt $A \leq_p B$ (heißt, jedes Problem in NP lässt sich auf B reduzieren)

Wenn nur Bedingung 2 gilt aber nicht unbedingt 1, dann liegt ein Problem in der Klasse **NP-Schwer**.

Wenn $B \in \mathcal{NPC}$ und $B \leq_p C$ für eine Sprache $C \in \mathcal{NP}$, dann folgt $C \in \mathcal{NPC}$.

Die Klasse co-NP

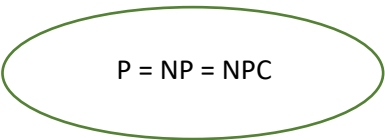
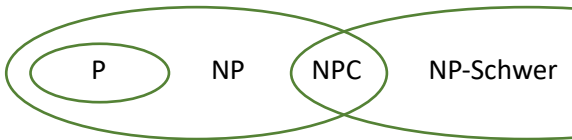
Ein Problem liegt in co-NP, wenn ihr Komplement in NP liegt. Das Komplement einer Sprache A ist wie folgt definiert:

$$\bar{A} = \Sigma^* - A$$

Heißt, alle (über einem Alphabet) möglichen Wörter, die nicht in A sind.

$P = NP$ oder $P \neq NP$?

Die Frage, ob $P = NP$ oder $P \neq NP$ ist (momentan) noch unbekannt. Es gibt jedoch zwei Szenarien:

$P = NP$	$P \neq NP$
 <p>P, NP und NPC sind alle dieselbe Sprachklasse.</p>	 <p>P ist eine Teilmenge von NP, und NPC die Schnittmenge zwischen NP und NP-Schwer.</p>

Wichtig: in beiden Szenarien ist P immer eine Teilmenge von NP ($P \subseteq NP$). Das heißt, jedes Problem in P ist auch ein NP Problem.

Wichtige Probleme in NPC

Hier werden die wichtigsten NP-Vollständigen Probleme aus der Vorlesung nochmal aufgeführt. Wie genau ein Problem auf ein anderes reduziert werden kann ist hier nicht, das steht aber alles in den Folien. Für viele Probleme gab es genaue Sprachdefinitionen, aber nicht für alle. Wenn keine in den Folien enthalten waren, habe ich sie selbst analog zur Vorlesung definiert.

Boolesche Formeln

SAT

SAT ist das Problem zu prüfen, ob eine boolesche Formel erfüllbar ist. Eine erfüllbare Formel ist eine Formel, für die mindestens eine wahre Belegung existiert.

$$SAT = \{\langle \phi \rangle \mid \phi \text{ ist eine erfüllbare boolesche Formel}\}$$

Bsp. $\phi_1 = (x_1 \vee x_2) \wedge x_3$ ist erfüllbar $\phi_1 \in SAT$
 $\phi_2 = x_1 \wedge \overline{x_1}$ ist nicht erfüllbar $\phi_2 \notin SAT$

3-SAT

3-SAT ist das Problem zu prüfen, ob eine boolesche Formel, in der jede Klausel genau 3 Literale hat, erfüllbar ist. Ein Literal ist eine negierte oder nicht negierte Variable. Eine Klausel ist die OR-Verknüpfung von Literalen. Eine Formel ist in 3-SAT, wenn jede Klausel genau 3 Literale hat.

$$3-SAT = \{\langle \phi \rangle \mid \phi \text{ ist eine erfüllbare boolesche Formel in der jede Klausel genau 3 Literale hat}\}$$

Jede SAT Formel lässt sich auf eine 3-SAT Formel reduzieren.

Graphen

Clique

Eine Clique eines gerichteten Graphens $G = (V, E)$ ist eine Teilmenge $V' \subseteq V$, sodass alle Knoten in V' voll verbunden sind ((V', E) ist ein kompletter Graph). Eine k-Clique ist eine Clique der Größe k.

$$Clique = \{\langle G, k \rangle \mid G \text{ ist ein Graph mit einer } k\text{-Clique}\}$$

3-SAT lässt sich auf Clique reduzieren.

Independent Set

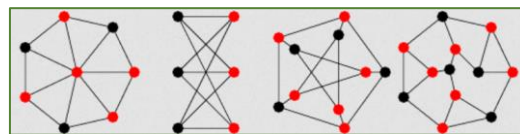
Ein Independent Set (IS) eines gerichteten Graphens $G = (V, E)$ ist eine Teilmenge $V' \subseteq V$, sodass jedes Knotenpaar in V' nicht durch eine Kante in E Verbunden ist. Analog zur Clique ist ein k-Independent Set ein Independent Set der Größe k.

$$Independent\ Set = \{\langle G, k \rangle \mid G \text{ ist ein Graph mit einem } k\text{-Independent Set}\}$$

Ein Graph hat ein IS, wenn es eine Clique hat (und umgekehrt). Die beiden Probleme lassen sich aufeinander reduzieren. Eine Clique in einem Graphen G ist ein IS in dem *Komplementärgraphen* von G und umgekehrt.

Vertex Cover

Ein Vertex Cover eines ungerichteten Graphen $G = (V, E)$ ist eine Teilmenge $V' \subseteq V$, sodass jede Kante mindestens einen Endknoten in V' enthält. Gesucht wird meistens ein minimal Vertex Cover, d.h. eine möglichst kleine Teilmenge von Knoten für die dies gilt.



Die rot markierten Knoten bilden
Vertex Cover ihrer Graphen

$$\text{Vertex Cover} = \{\langle G, k \rangle \mid G \text{ ist ein Graph mit einem (minimalen) } k\text{-Vertex Cover}\}$$

Traveling Salesman

Das Traveling Salesman Problem beschreibt das Problem, einen hamiltonischen Zyklus mit minimalen Kosten in einem vollständigen, ungerichteten Graphen $G = (V, E)$ mit nicht-negativen ganzzahligen Kosten $c(u, v)$ für jede Kante zu finden. Heißt, ein Kreis der alle Knoten besucht mit minimalen Kosten.

$$\text{Traveling Salesman} = \{\langle G, k \rangle \mid G \text{ ist ein Graph mit minimalen Traveling Salesman Kosten } k\}$$

Mengen

Set Cover

Gegeben sind ein Set von Elementen $U = \{1, 2, \dots, n\}$ und eine Menge von Sets S die jeweils Teilmengen von U sind. Gesucht ist die minimale Anzahl von Sets aus S , die jedes Element aus U enthalten.

$$\text{Set Covering} = \{\langle U, S, k \rangle \mid \langle U, S \rangle \text{ ist eine Instanz von Set Cover mit Lösungsgröße } k\}$$

Bsp.: $U = \{1, 2, 3, 4, 5\}$, $S = \{\{1, 2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}\}$

Hier ist die optimale Lösung $\{\{1, 2, 3\}, \{4, 5\}\} \subseteq S$, mit $k = 2$

Subset-Sum

Eine Instanz des Subset-Sum Problems ist ein Paar $\langle S, t \rangle$, wobei S eine Menge von positiven ganzen Zahlen und t eine positive ganze Zahl ist. Subset-Sum lässt sich als Entscheidungs- und Optimierungsproblem interpretieren.

Entscheidungsproblem Gibt es eine Teilmenge $S' \subseteq S$ dessen Summe gleich t ist?

$$\text{Subset-Sum}_E = \{\langle S, t \rangle \mid \text{In } S \text{ existiert eine Teilmenge von Zahlen mit Summe } t\}$$

Optimierungsproblem Finde die Teilmenge $S' \subseteq S$ dessen Summe größtmöglich aber höchstens t ist

$$\text{Subset-Sum}_O = \{\langle S, t, t' \rangle \mid S \text{ ist eine Menge von positiven Zahlen. Eine Teilmenge von } S \text{ lässt sich höchstens zu } t' \text{ mit } t' \leq t \text{ aufsummieren}\}$$

Bin Packing

Gegeben ist eine Liste von n Gegenständen mit Größen $a_1, \dots, a_n, a_i \in (0,1]$. Ein Bin ist ein Behälter mit Kapazität 1, es passen also nur Gegenstände mit einer Summe ≤ 1 rein. Was ist die minimale Anzahl von Bins, sodass alle Gegenstände aufgenommen werden können?

$\text{Bin Packing} = \{\langle A, k \rangle \mid A \text{ ist eine Liste von Gegenständen mit Größen die in } k \text{ Bins passen}\}$

Knapsack

Gegeben ist eine Menge $S = \{a_1, \dots, a_n\}$ von Objekten mit Gewichten $s_1, \dots, s_n \in \mathbb{Z}^+$ und Werten $p_1, \dots, p_n \in \mathbb{Z}^+$ sowie eine Kapazität B . Finde eine Teilmenge der Objekte deren Gesamtgewicht durch B begrenzt und Gesamtwert maximal ist.

$\text{Knapsack} = \{\langle S, B, k \rangle \mid S \text{ ist eine Liste von Objekten mit Gewichten und Werten.}$
 $k \text{ ist die maximale Summe der Werte der Objekte deren Gewichte } B \text{ nicht überschreiten}\}$

Dreiecksungleichung

Die Dreiecksungleichung ist zwar kein eigenes Problem, wird aber häufig zur Berechnung von Approximationsfaktoren genutzt und ist auch klausurrelevant. Außerdem kann man viele Probleme in zwei Varianten unterteilen: die, bei denen die Dreiecksungleichung gilt, und die bei denen sie nicht gilt.

Sei $G = (V, E)$ ein Graph mit positiven, ganzzahligen Kosten $c(u, v)$ für jede Kante. Die Kostenfunktion c erfüllt die Dreiecksungleichung, wenn die folgende Ungleichung für alle Kanten gilt:

$$c(u, v) + c(v, w) \geq c(u, w)$$

Heißt, es ist immer am billigsten direkt von einem Knoten u zu einem anderen Knoten w zu gehen, anstatt über einen weiteren Knoten v .

Approximation

Für viele Probleme ist nicht möglich in polynomieller Zeit die optimale Lösung zu finden (außer $P=NP!$), häufig ist diese exponentiell, oder noch schlimmer, Faktoriell. Stattdessen sucht man dann nicht nach der optimalen Lösung, sondern nach einer Lösung die *gut genug* ist.

*Bspw. anstatt der **maximalen Clique** suchen wir nach einer **ziemlich großen Clique***

Approximationsfaktor

Wie gut die approximierte Lösung eines Algorithmus (im Worst Case) zur optimalen Lösung ist, wird mit dem Approximationsfaktor $\rho(n)$ angegeben. Ein Approximationsfaktor ist immer ≥ 1 , egal ob es ein Maximierungs- oder Minimierungsproblem ist.

Wenn für eine feste Eingabegröße n fast alle Lösungen einen Approximationsfaktor 1 aber eine Lösung den Approximationsfaktor 3 hat, dann ist der Approximationsfaktor $\rho(n) = 3$.

Approximationsfaktor	Beschreibung
1-Approximations-Algorithmus	Der Algorithmus bestimmt die optimale Lösung.
2-Approximations-Algorithmus	Die Lösung ist im Worst Case doppelt so schlecht wie die optimale Lösung
3-Approximations-Algorithmus	Die Lösung ist im Worst Case dreifach so schlecht wie die optimale Lösung
...	...

Für viele Probleme ist der Approximationsfaktor konstant, also unabhängig von der Eingabegröße. Es ist aber möglich, dass dieser abhängig von der Eingabegröße ist, d.h. er kann sich für kleiner oder große Instanzen unterscheiden.

Leider gibt es auch hier kein allgemeines Verfahren, um den Approximationsfaktor eines Algorithmus zu bestimmen. Hier hilft eine gute Intuition von Aufgaben und Übung. Die Folien enthalten außerdem gute Beispiele für Berechnungen von Approximationsfaktoren.

Approximations-Schema

Es ist möglich, dass man bessere Lösungen findet, wenn man mehr Rechenzeit aufwendet. Die besseren Lösungen drücken sich in Form eines neuen Parameters ϵ aus, der die Abweichung zur optimalen Lösung angibt.

Ein Approximations-Schema für ein Optimierungsproblem ist eine Familie von Approximationsalgorithmen, der als Eingabe eine Instanz und einen Parameter $\epsilon > 0$ erhält. Basierend darauf wird für eine $(1+\epsilon)$ -Approximation berechnet.

Achtung! Die Laufzeit ist jetzt von der Eingabegröße n und dem ϵ abhängig, bspw. $T(n) = n^{2/\epsilon}$. Dabei ist es möglich, dass die Laufzeit sehr schnell für kleiner werdende ϵ zunimmt.

n	ϵ	2	1	$1/2$	$1/4$	$1/100$
	$T(n)$	n	n^2	n^4	n^8	n^{200}
10^1		10^1	10^2	10^4	10^8	10^{200}
10^2		10^2	10^4	10^8	10^{16}	10^{400}
10^3		10^3	10^6	10^{12}	10^{24}	10^{600}
10^4		10^4	10^8	10^{16}	10^{32}	10^{800}

Die Laufzeiten für $T(n) = n^{2/\epsilon}$

PTAS

PTAS steht für **poly-time Approximation-Schema**. Die Eingabegröße darf dabei für ein festes ϵ höchstens polynomiell zu n sein.

	PTAS	FPTAS
n	Polynomiell	Polynomiell
ϵ	Beliebig	Polynomiell

FPTAS

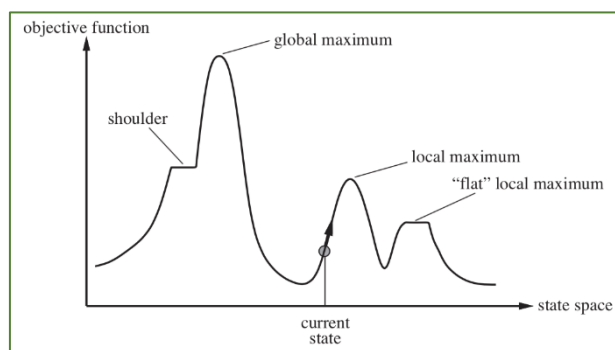
FPTAS steht für **fully polynomial Approximation-Schema**. Die Eingabegröße darf dabei höchstens polynomiell zu n sowie zu $1/\epsilon$ sein. Jedes FPTAS ist gleichzeitig auch ein PTAS. Für manche Probleme existieren keine PTAS und FPTAS Schemata (außer, $P=NP$).

Hier ein paar Beispiele:

Laufzeit	n	$1/\epsilon$	PTAS?	FPTAS?
$n^{2/\epsilon}$	Polynomiell	Exponentiell	Ja	Nein
$4n^{1/\epsilon}$	Polynomiell	Exponentiell	Ja	Nein
$2^n \cdot 1/\epsilon$	Exponentiell	Linear	Nein	Nein
$n \cdot 8/\epsilon^2$	Linear	Polynomiell	Ja	Ja
$\log(n) \cdot 1/\epsilon^4$	Logarithmisch	Polynomiell	Ja	Ja

Lokale Suche

Eine Methode zur Lösung NP-schwerer Probleme die genutzt wird, um komplizierte **Optimierungsprobleme** näherungsweise zu lösen. Das Verfahren basiert darauf eine bessere Lösung zu finden, indem man inkrementell eine bessere Lösung aus der Nachbarschaft der gerade betrachteten Lösung nimmt. Damit findet man *gute*, aber nicht unbedingt *optimale* Lösungen.



Die momentane Lösung „current state“ wird sich dem local maximum nähern

- Sei \mathcal{S} die Menge aller möglichen Lösungen
- Eine bestimmte Lösung $S \in \mathcal{S}$ hat die Kosten $C(S)$
- Wir suchen eine Lösung S' in der Nähe von S mit niedrigeren Kosten als S ($C(S') < C(S)$)
- Wir testen dafür ähnliche Versionen von S , indem wir leicht dessen Parameter beeinflussen
- Das wiederholen wir, bis S die beste Lösung unter seinen Nachbarn ist (\Rightarrow Maximum)

Gradient Descent nimmt dabei immer den Nachbarn, der die Kosten **minimiert**. Da immer nur die lokale Nachbarschaft betrachtet wird, passiert es häufig, dass der Algorithmus in einem **lokalen Maximum** stecken bleibt. Außerdem wird eine Kostenfunktion C benötigt, die allen Lösungen bestimmte Kosten zuweist.

Metropolis-Algorithmus

Der Metropolis-Algorithmus ist ein lokaler Suchalgorithmus mit parallelen zu thermodynamischen Systemen. Der Zustandsübergang zwischen zwei Zuständen S und S' basiert dabei mit auf der *Gibbs-Boltzmann-Funktion*, die im Kontext der Physik für feste Parameter k (*Boltzman-Konstante*) und T (*Temperatur*) die Wahrscheinlichkeit angibt, ein thermodynamisches System in einem Zustand mit Energie E zu finden:

$$G(E) = e^{-E/(kT)}$$

Wenn T hoch ist, sind die Unterschiede zwischen den Wahrscheinlichkeiten für Zustände mit hohen und niedrigen Energien klein, d.h., das System "akzeptiert" wahrscheinlicher auch schlechtere Lösungen. Dies ermöglicht eine **breitere Suche** im Lösungsraum und verhindert, dass der Algorithmus zu früh in einem lokalen Optimum hängen bleibt.

Wenn T jedoch klein ist, ist die Wahrscheinlichkeit, dass sich das System in einem Zustand mit niedriger Energie befindet, deutlich größer als die Wahrscheinlichkeit für einen Zustand mit hoher Energie, was zu einer **schmaleren Suche** in der Nähe der aktuell besten Lösung führt.

k hat ähnliche Auswirkungen auf das System wie T , ist aber im Kontext von thermodynamischen Systemen eine fundamentale Konstante.

Der Metropolis-Algorithmus nimmt als Eingabe eine Anfangslösung S_0 und Parameter k und T und arbeitet wie folgt:

1. S ist die aktuelle Lösung
2. Nehme uniform zufällig eine Lösung S' aus der Nachbarschaft von S
3. Wenn $C(S') \leq C(S)$, dann $S = S'$
4. Wenn $C(S') > C(S)$, dann $S = S'$ mit einer Wahrscheinlichkeit von $e^{-\frac{C(S')-C(S)}{kT}}$

Simulated Annealing

Simulated Annealing ist fast identisch zum Metropolis-Algorithmus, jedoch verringert sich die Temperatur T über Zeit. Wir starten mit einer hohen Temperatur und verringern diese im Laufe der Zeit.

Simulated Annealing nimmt als Eingabe eine Anfangslösung S_0 und Parameter k und T . Wir haben außerdem eine **cooling schedule** $\tau: \mathbb{N} \rightarrow \mathbb{R}^+$, die jeder Iteration eine Temperatur zuordnet. Die Temperatur muss über Zeit fallen und auf 0 enden.

1. Wähle T gemäß der cooling schedule
2. S ist die aktuelle Lösung
3. Nehme uniform zufällig eine Lösung S' aus der Nachbarschaft von S
4. Wenn $C(S') \leq C(S)$, dann $S = S'$
5. Wenn $C(S') > C(S)$, dann $S = S'$ mit einer Wahrscheinlichkeit von $e^{-\frac{C(S') - C(S)}{kT}}$

Zu Beginn springen wir zwischen allen Lösungen hin und her, gegen Ende pendelt sich die Suche in ein lokales oder sogar das globale Maximum ein.

Es gibt leider keine gängige Methode, die optimalen Parameter, eine optimale cooling schedule oder eine optimale Auswahl von Nachbarschaftskandidaten zu finden. Hier muss man für gewöhnlich einfach Parameter ausprobieren und testen, ob sie zu guten Ergebnissen führen.

Hopfield Networks

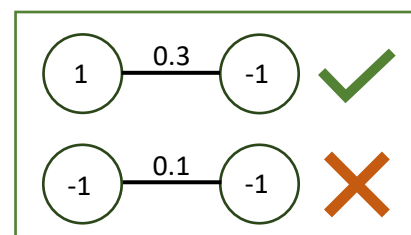
Ein Hopfield-Netzwerk ist ein ungerichteter Graph $G = (V, E)$ mit ganzzahligen Gewichten w_e an den Kanten. Eine Konfiguration des Netzwerks weist jedem Knoten in V einen Zustand $s_u \in \{-1, 1\}$ zu. So gesehen sind Hopfield-Netzwerke eine sehr simple Variante von neuronalen Netzwerken.

- Gewicht w_e von $(u, v) \in E$ ist **negativ**: u und v bevorzugen den gleichen Zustand
- Gewicht w_e von $(u, v) \in E$ ist **positiv**: u und v bevorzugen unterschiedliche Zustände
- $|w_e|$ gibt die **Stärke** dieser Bevorzugung an

Kanten können entweder gut oder schlecht sein:

- Eine Kante ist **gut** wenn
 - Entweder $w_e < 0$ und $s_u = s_v$
 - Oder $w_e > 0$ und $s_u \neq s_v$
- Gilt keins von beiden, ist die Kante **schlecht**

Daraus folgt: eine Kante $e = (u, v)$ ist gut, wenn $w_e \cdot s_u \cdot s_v < 0$



Ein Knoten kann entweder zufrieden oder unzufrieden sein:

- Ein Knoten v ist **zufrieden**, wenn das Gesamtgewicht seiner guten Kanten mindestens so groß wie das seiner schlechten Kanten ist. Dies ist äquivalent zu:

$$\sum_{v: e=(u,v) \in E} w_e \cdot s_u \cdot s_v \leq 0$$

Der Zustand des gesamten Netzwerks wird **stabil** genannt, wenn alle Knoten zufrieden sind. Jedes Hopfield-Netzwerk hat mindestens einen stabilen Zustand. Ein stabiler Zustand kann über den folgenden Algorithmus in polynomieller Zeit gefunden werden:

1. Wähle einen unzufriedenen Knoten v
2. Flippe den Zustand von v (also -1 wird zu 1 und 1 wird zu -1)
3. Wiederhole die Schritte bis alle Knoten zufrieden sind

Maximum-Cut

Gegeben sei ein ungerichteter Graph $G = (V, E)$ mit positiven ganzzahligen Gewichten w_e für jede Kante $e \in E$. Für zwei Teilmengen der Knoten $A, B \subseteq V$ definieren wir

$$w(A, B) = \sum_{e=(u,v), u \in A, v \in B} w_e$$

Das bedeutet, $w(A, B)$ ist die Summe aller Kantengewichte zwischen A und B . Wir suchen eine Partition A, B von V (heißt, ein Knoten ist entweder in A oder B , $A \cup B = V$), sodass $w(A, B)$ maximiert/minimiert wird.

- **Maxcut** beschreibt das Problem, eine **maximale** Summe von $w(A, B)$ zu finden (liegt in NPC)
- **Mincut** beschreibt das Problem, eine **minimale** Summe von $w(A, B)$ zu finden (liegt in P)

Da Maxcut und Hopfield eng verwandte Probleme sind (Knoten mit -1 sind A und Knoten mit 1 sind B), lässt sich hier derselbe Algorithmus aus Hopfield anwenden. Die Nachbarschaftszustände des Zustands (A, B) sind dabei alle Zustände, die durch Verschieben eines Knotens in die andere Menge erreicht werden können.

Ein Knoten ist hier zufrieden, wenn die Summe der Kantengewichte zu Knoten in derselben Partition nicht größer ist als die Summe der Gesamtgewichte zu den Knoten einer anderen Partition.

Der Algorithmus sieht so aus:

1. Wähle einen unzufriedenen Knoten v
2. Flippe den Zustand von v (wenn $v \in A$ dann wechsele v zu B und umgekehrt)
3. Wiederhole die Schritte bis alle Knoten zufrieden sind

Wenn (A, B) eine Partition in einem lokalen Maximum und (A^*, B^*) die global optimale Partition ist, gilt:

$$W(A, B) \geq \frac{1}{2} \cdot W(A^*, B^*)$$

Somit bestimmt der Algorithmus also eine 2-Approximation von Maxcut. Da dieser aber nicht polynomiell in n da er außerdem linear in W ist, lässt sich dieser leicht zu einem Approximationsschema umwandeln. Die Idee ist dabei, den Algorithmus zu stoppen, wenn keine „großen“ Verbesserungen mehr möglich sind. Eine „große“ Verbesserung bedeutet hier, dass sie den Wert eines Cuts (A, B) um mindestens den folgenden Wert erhöht:

$$\frac{2\epsilon}{n} \cdot W(A, B)$$

Das neue Approximationsschema nimmt nun außerdem ein ϵ als Parameter und sieht wie folgt aus:

1. Wähle einen nicht zufriedenen Knoten v der eine große Verbesserung bewirkt
2. Flippe den Zustand von v (wenn $v \in A$ dann wechsele v zu B und umgekehrt)
3. Wiederhole die Schritte bis alle Knoten zufrieden sind

Die Güte des Approximationsschemas ist nun:

$$(2 + \epsilon) \cdot W(A, B) \geq W(A^*, B^*)$$

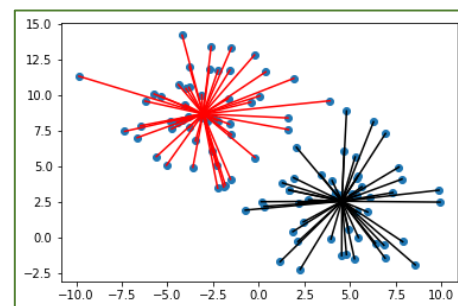
Damit ist dieses Schema bestenfalls leicht schlechter als eine 2-Approximation. Der Algorithmus endet nach $O(1/\epsilon \cdot n \cdot \log(W))$ vielen Flips, wobei W die Summe aller Kantengewichte ist ($W = \sum w_i$). Das Approximationsschema ist dadurch außerdem ein FPTAS.

k-Median

k Facilities (Bspw. Supermärkte, Geldautomaten, ...) sollen möglichst optimal in der Nähe aller n Kunden platziert werden. Allgemein sollen k Punkte möglichst optimal unter allen n Datenpunkten platziert werden, sodass die Summe der Distanzen der Datenpunkte zu ihrer nächsten Facility minimiert wird. Dieses Problem ist NP-Schwer.

Gegeben ist:

- Menge aller möglichen Orte für Facilities F , $|F| = m$
- Menge an Kunden D , $|D| = n$
- Distanzfunktion $d: D \times F \rightarrow \mathbb{R}$
 $d(i, j)$ sind die Kosten, falls Kunde i eine Facility an Ort j nutzt
- k ist die Anzahl an Facilities die geöffnet werden



Hier wurden zwei Facilities ($k=2$) möglichst optimal platziert. Die Linien zeigen die Distanzen der Kunden zu ihrer nächsten Facility

Wir suchen eine Teilmenge $S \subset F$ mit $|S| \leq k$, sodass die folgende Gleichung minimiert wird:

$$\sum_{i \in D} \min_{j \in S} d(i, j)$$

Dabei ist $\min_{j \in S} d(i, j)$ die Distanz eines Kunden i zu seiner nächstgelegenen Facility.

Wir betrachten eine leicht angepasste Variante des Problems. Hier unterscheiden wir nicht zwischen Facilities und Kunden, bzw. jeder Punkt ist gleichzeitig Kunde als auch Ort einer möglichen Facility:

- N ist eine Menge von n möglichen Orten
- Distanzfunktion $d: N \times N \rightarrow \mathbb{R}$, $d(i, j) = d(j, i)$

Der Approximationsalgorithmus dazu sieht so aus:

1. Wähle k zufällige Punkte für S
2. Solange ein Punkt $s' \notin S$ existiert und dieser Punkt eine Verbesserung der Gesamtkosten $C(S)$ erzielt, falls er mit einem Punkt $s \in S$ ausgetauscht wird, ersetze s mit s'

Die Nachbarschaft einer Menge S ist eine Menge S' , in der ein Punkt $s \in S$ mit einem anderen Punkt $s' \notin S$ ausgetauscht wird. Dies wird auch *Swap-Nachbarschaft* genannt. Jede Lösung, die unter der Swap-Nachbarschaft *lokal optimal* ist, ist höchstens 5 mal größer als die optimale Lösung. Damit berechnet dieser Algorithmus eine **5-Approximation**.

Dieser Algorithmus hat jedoch nicht unbedingt eine polynomielle Laufzeit. Hier können wir denselben Trick wie bei [Maxcut](#) anwenden: wir erlauben Swaps nur, wenn sie eine Verbesserung um den Faktor $(1 - \delta)$ bewirken. So erhält man einen $(5+\epsilon)$ -Approximationsalgorithmus mit polynomieller Laufzeit.