

2. Test Zusammenfassung

Vorwort

Diese Zusammenfassungen wurden basierend auf den Vorlesungsfolien erstellt und liegen als **PDF-Dateien** vor. Sie dienen als stichwortartige Gedächtnisstütze und orientieren sich stark an den Inhalten der Folien.

Meiner Meinung nach sind die **Slides bereits sehr gut aufbereitet**, um den Stoff zu lernen, und diese Zusammenfassungen dienen primär der schnellen Wiederholung und dem Überblick. Daher wurden viele Formulierungen und die Struktur der Slides übernommen.

Legende

 Aufzählungen / Abläufe / etc.

 Definitionen / anderes Wichtiges

 Sätze / Informationen / Lemmas / Bemerkungen / ...

 Beispiele

 Fragestellungen / Problemstellungen / ...

 Beweise

 Anmerkungen

Inhalt

ⓘ Inhalt

- 9. Polynominalzeitreduktionen
- 10. NP-Vollständigkeit Spezialfälle
- 11. Branch and Bound
- 12. Dynamische Programmierung
- 13. Approximation
- 14. Heuristiken und Lokale Suche

9. Polynominalzeitreduktionen

Einleitung

Effizient lösbare Probleme (Wiederholung)

- **Definition:** Ein Problem gilt als **effizient lösbar**, wenn es durch einen Algorithmus gelöst werden kann, dessen Laufzeit durch ein **Polynom** der Eingabegröße beschränkt ist.
 - Laufzeit: $O(n^c)$
 - n : Eingabegröße (z.B. Anzahl der Bits)
 - c : Konstanter Exponent
- **Synonym:** Effizient lösbare Probleme werden auch als **handhabbar (tractable)** bezeichnet.
- **Cobham-Edmonds-Annahme:**
 - Vorschlag von Alan Cobham und Jack Edmonds in den 1960er-Jahren.
 - **Gleichsetzung von Handhabbarkeit mit Lösbarkeit in Polynomialzeit.**
 - Hat die Informatikforschung der letzten 50 Jahre maßgeblich beeinflusst und sich weitgehend durchgesetzt.

Diskussion zur Cobham-Edmonds-Annahme

- **Rechtfertigung der Annahme:**
 - **Praxisbezug:** Polynomielle Algorithmen weisen in der Regel kleine Konstanten und niedrige Exponenten auf.
 - **Strukturelle Einsicht:** Der Übergang von exponentiellen (z.B. Brute-Force) zu polynomiellen Algorithmen deutet oft auf das Erkennen einer fundamentalen Struktur des Problems hin.
- **Ausnahmen/Kritik:**
 - **Ineffiziente polynomielle Algorithmen:** Es existieren polynomielle Algorithmen mit sehr großen Konstanten oder hohen Exponenten, die in der praktischen Anwendung unbrauchbar sein können.
 - **Praktische Relevanz exponentieller Algorithmen:** Algorithmen mit exponentieller oder noch schlechterer Laufzeit finden dennoch Anwendung, wenn:
 - Worst-Case-Eingaben extrem selten auftreten.
 - Die Größe der zu lösenden Problemfälle ausreichend klein ist.

Probleme klassifizieren: P or not P?

Ziel der Klassifizierung von Problemen

- Unterscheidung zwischen Problemen, die **in Polynomialzeit lösbar** sind, und solchen, die **nicht in Polynomialzeit lösbar** sind (NP-Probleme)

☰ Beispiele für Probleme, die nachweislich mehr als polynomiale Zeit erfordern:

- Halteproblem mit Schranke:** Hält eine gegebene Turingmaschine nach höchstens k Schritten?
- Verallgemeinertes Schachgewinnproblem:** Gegeben sei eine Brettbelegung für eine $n \times n$ Generalisierung von Schach. Kann Schwarz garantiert gewinnen?

☰ Probleme, deren Klassifizierung (P oder NP?) unbekannt ist

- Maximum Independent Set:** Gegeben ein Graph G und eine Zahl k , enthält G mindestens k Knoten, die paarweise nicht adjazent sind?
- 3-Färbbarkeit:** Lassen sich die Knoten eines gegebenen Graphen mit 3 Farben färben, sodass Paare adjazenter Knoten unterschiedliche Farben haben?
- SAT (Erfüllbarkeitsproblem der Aussagenlogik):** Ist eine gegebene aussagenlogische Formel erfüllbar?

Anmerkung: Für viele fundamentale Probleme konnte noch keine eindeutige Klassifizierung (polynomial oder exponentiell) gefunden werden. Dies ist ein unbefriedigender Zustand in der theoretischen Informatik.

Umgang mit Problemen, die nicht in Polynomialzeit lösbar sind:

- Wie gehen wir damit um, wenn wir ein Problem nicht in Polynomialzeit lösen können?
- Man soll nicht sagen, dass man zu blöd ist den Algo zu finden, sondern soll sagen, dass „am selbst + die ganzen anderen Personen nichts gefunden haben.“

Ja/Nein Probleme

siehe hier

- Vereinfachung:** Zur einfacheren Betrachtung konzentrieren wir uns auf **Ja/Nein-Probleme (decision problems)**.
- Definition Ja/Nein-Problem:** Ein Problem, dessen Lösung entweder **Ja** oder **Nein** ist.
- Unterscheidung zu anderen Problemtypen:**
 - Funktionales Problem:** Liefert eine **Lösung** oder eine **Lösungsmenge** als Antwort.

- **Optimierungsproblem:** Ziel ist es, eine **optimale Lösung** (z.B. Minimum oder Maximum) zu finden.

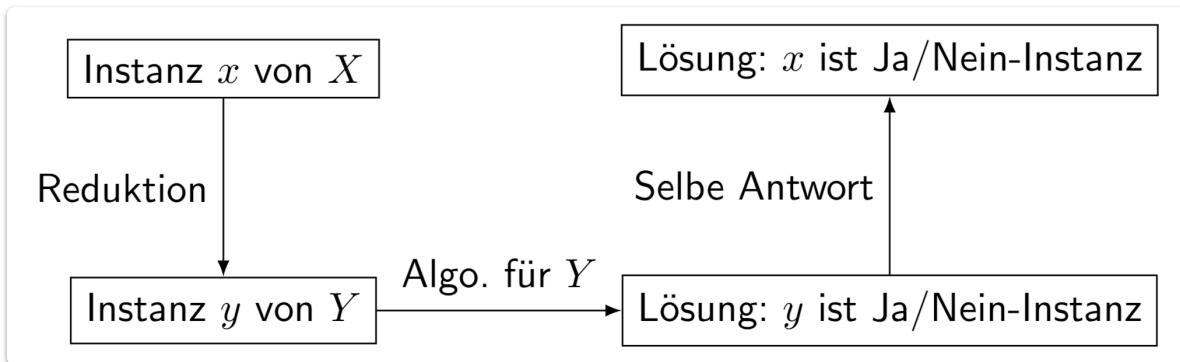
☰ Beispiel und Unterscheidung:

- **Ja/Nein-Problem:** Gibt es für einen gewichteten Graphen einen Spannbaum mit Kosten $\leq k$? (Antwort: Ja oder Nein)
- **Funktionales Problem:** Finde in einem gewichteten Graphen einen Spannbaum mit Kosten $\leq k$. (Antwort: Der Spannbaum selbst, falls er existiert)
- **Optimierungsproblem:** Finde in einem gewichteten Graphen einen Spannbaum mit minimalen Kosten (MST). (Antwort: Der Spannbaum mit den geringsten Gesamtkosten)

Polynomialzeitreduktionen

Wir haben 2 Probleme und wollen das Problem X auf das Problem Y polynomiell reduzieren.

- **Informelle Beschreibung:** Wenn ein Problem X **polynomialzeitreduzierbar** auf ein Problem Y ist, bedeutet das:
 - Falls wir einen **effizienten Algorithmus** (Polynomialzeit) für Y haben,
 - dann können wir auch X **effizient lösen**.
- **Vorgehensweise der Reduktion:**
 1. Gegeben eine **Instanz x von Problem X** .
 2. Transformiere x in eine **Instanz y von Problem Y** in Polynomialzeit.
 3. Löse die Instanz y von Y mit dem effizienten Algorithmus für Y .
 4. **Schlussfolgerung:**
 - Wenn y eine **Ja-Instanz** von Y ist, dann ist auch x eine **Ja-Instanz** von X .
 - Wenn y eine **Nein-Instanz** von Y ist, dann ist auch x eine **Nein-Instanz** von X .



⌚ Definition

siehe

Eine **Polynomialzeitreduktion** von Problem X auf Problem Y ist ein Algorithmus R , der für jede Instanz x von X eine Instanz y von Y berechnet, so dass die folgenden zwei Bedingungen erfüllt sind:

1. **Äquivalenz der Ja-Instanzen:** x ist eine Ja-Instanz von X genau dann, wenn y eine Ja-Instanz von Y ist.
2. **Effizienz der Reduktion:** Der Algorithmus R hat eine **Laufzeit in Polynomialzeit**. Das bedeutet, es existiert eine Konstante c , sodass R die Instanz y in einer Zeit von $O(n^c)$ berechnet, wobei n die Eingabegröße der Instanz x ist.

Notation

Wir schreiben $X \leq_P Y$, um auszudrücken, dass es eine Polynomialzeitreduktion von X auf Y gibt. In diesem Fall sagen wir auch: " **X ist auf Y polynomiell reduzierbar**".

Hinweis zur Interpretation

Falls $X \leq_P Y$ gilt, kann man auch sagen, dass "Y mindestens so schwer ist wie X"

Lösung von Problemen durch Reduktion

siehe

- **Idee:** Nutze eine Polynomialzeitreduktion auf ein bereits als handhabbar bekanntes Problem.
- **Grundsatz:** Wenn $X \leq_P Y$ und Y in Polynomialzeit lösbar ist, dann ist auch X in Polynomialzeit lösbar.
- **Beweis:**
 - Sei R der Reduktionsalgorithmus von X nach Y mit Laufzeit $O(n^a)$ für eine Instanz x von X der Größe n ($a \geq 1$).
 - Sei A der Algorithmus zum Lösen von Y mit Laufzeit $O(n^b)$ für eine Instanz y von Y der Größe n ($b \geq 1$).
 - Gegeben eine Instanz x von X der Größe n :
 1. Anwenden von R auf x erzeugt eine Instanz y von Y in $O(n^a)$ Zeit.
 2. Die Größe von y ist höchstens $O(n^a)$, da sie in dieser Zeit erzeugt wurde.
 3. Lösen von y mit A benötigt $O((n^a)^b) = O(n^{ab})$ Zeit.
 - Die Gesamtlaufzeit zur Lösung von x ist $O(n^a) + O(n^{ab}) = O(n^{ab})$, was ein Polynom in n ist. \square

Nachweis der Nicht-Handhabbarkeit durch Reduktion

siehe

- **Idee:** Reduziere ein bereits als nicht handhabbar bekanntes Problem auf das zu untersuchende Problem.
- **Grundsatz:** Wenn $X \leq_P Y$ und X nicht in Polynomialzeit lösbar ist, dann kann auch Y nicht in Polynomialzeit lösbar sein.
- **Beweis (durch Widerspruch):**
 - Angenommen, Y wäre in Polynomialzeit lösbar.
 - Da $X \leq_P Y$, existiert eine Polynomialzeitreduktion von X auf Y.
 - Durch die Kombination der Polynomialzeitreduktion und des Polynomialzeitalgorithmus für Y könnte X ebenfalls in Polynomialzeit gelöst werden.
 - Dies widerspricht der Annahme, dass X nicht in Polynomialzeit lösbar ist.

⌚ Definition - Polynomialzeit-Äquivalenz

Wenn $X \leq_P Y$ und $Y \leq_P X$, dann schreiben wir $X \equiv_P Y$. Dies bedeutet, dass X und Y bezüglich ihrer Schwierigkeit in Polynomialzeit äquivalent sind.

Satz - Transitivität der PZR

siehe

- Ist $X \leq_P Y$ und $Y \leq_P Z$, dann folgt daraus $X \leq_P Z$.
- **Beweis:**
 - Sei R_1 der Reduktionsalgorithmus von X nach Y mit Laufzeit $O(n^a)$ ($a \geq 1$).
 - Sei R_2 der Reduktionsalgorithmus von Y nach Z mit Laufzeit $O(n^b)$ ($b \geq 1$).
 - Sei x eine Instanz von X der Größe n .
 - $R_1(x)$ erzeugt eine Instanz x' von Y in höchstens $O(n^a)$ Zeit, wobei die Größe von x' $O(n^a)$ ist.
 - $R_2(x')$ erzeugt eine Instanz z von Z in $O((n^a)^b) = O(n^{ab})$ Zeit.
 - Somit existiert eine Polynomialzeitreduktion (die Komposition von R_1 und R_2) von X nach Z mit einer Laufzeit von $O(n^a) + O(n^{ab}) = O(n^{ab})$, was polynomiell in n ist.

Angeben von Polynomialzeitreduktion

siehe

Beim Definieren einer Polynomialzeitreduktion von einem Problem X auf ein Problem Y müssen zwei zentrale Eigenschaften nachgewiesen werden:

1. Korrektheit der Reduktion:

- Ja-Instanzen von X müssen auf Ja-Instanzen von Y abgebildet werden.
- Nein-Instanzen von X müssen auf Nein-Instanzen von Y abgebildet werden.

2. Polynomialität der Reduktion:

- Der Reduktionsalgorithmus muss in Polynomialzeit ausführbar sein.

Die Schwierigkeit des Nachweises kann variieren: In manchen Fällen ist die Korrektheit offensichtlich, während in anderen die Polynomialität leichter zu zeigen ist.

Independent set und Vertex Cover

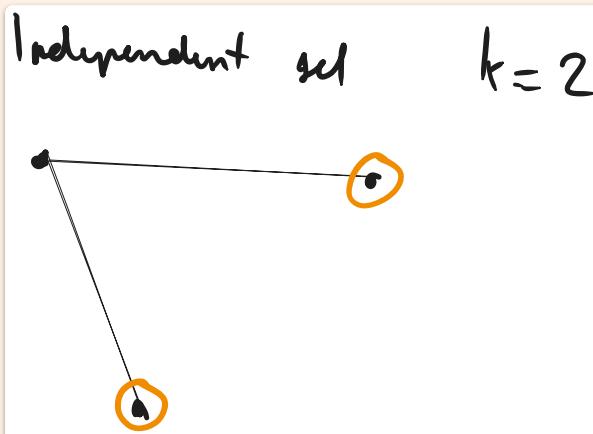
Independent set

⌚ Definition

- Ein *Independent Set* (oder auch *unabhängige Menge*) eines Graphen $G = (V, E)$ ist eine Teilmenge $S \subseteq V$ der Knoten, in der es **keine zwei adjazenten Knoten** gibt.
- **Beispiel:** Wenn Knoten A und B durch eine Kante verbunden sind, können nicht beide in einem Independent Set sein.

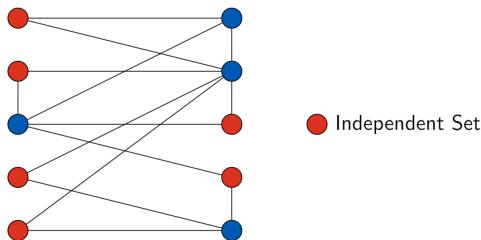
⌚ Problemstellung

- **Gegeben:** Ein Graph $G = (V, E)$ und eine ganze Zahl k .
- **Frage:** Gibt es ein Independent Set S , sodass $|S| \geq k$ gilt?
 - **Wichtig:** Die Zahl k ist Teil der Eingabe und keine Konstante. Sie variiert also von Problem zu Problem.
- **Hinweis:** Ja/Nein-Probleme werden ab jetzt mit dieser Schreibweise (z.B. INDEPENDENT SET) gekennzeichnet.



☰ Weiteres Beispiel

INDEPENDENT SET: Gegeben sei ein Graph $G = (V, E)$ und eine ganze Zahl k . Gibt es ein Independent Set S , sodass $|S| \geq k$ gilt?



Beispiel: Existiert ein Independent Set der Größe ≥ 6 ? Ja.

Beispiel: Existiert ein Independent Set der Größe ≥ 7 ? Nein.

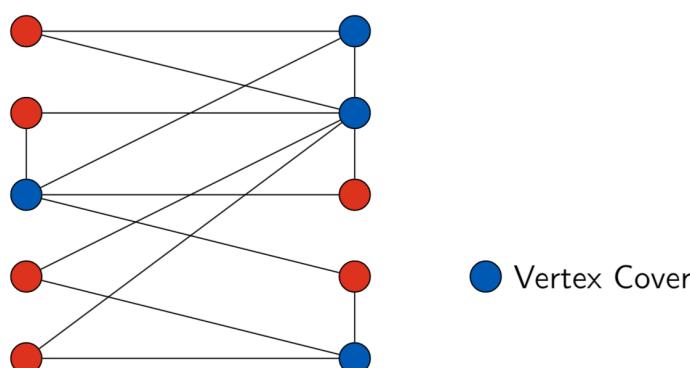
Vertex Cover

⌚ Definition

- Ein *Vertex Cover* (oder auch *Knotenüberdeckung*) eines Graphen $G = (V, E)$ ist eine Menge $S \subseteq V$ von Knoten, sodass jede Kante des Graphen zu mindestens einem Knoten aus S inzidiert ist.
 - Anders ausgedrückt: Wenn man alle Knoten in S markiert, muss jede Kante im Graphen mindestens einen markierten Endpunkt haben.

(?) Problemstellung

- Gegeben:** Ein Graph $G = (V, E)$ und eine ganze Zahl k .
- Frage:** Gibt es ein Vertex Cover S von G , sodass $|S| \leq k$ gilt?



- Beispiel für die Abbildung:**

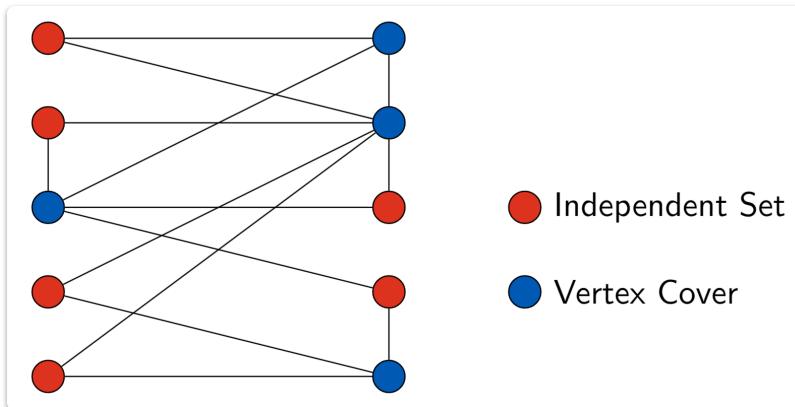
- Existiert ein Vertex Cover der Größe ≤ 4 ? Ja.
- Existiert ein Vertex Cover der Größe ≤ 3 ? Nein.

Konversionslemma: Vertex Cover und Independent Set

Wir wollen zeigen, dass $\text{Vertex Cover} \equiv_P \text{Independent Set}$. Dazu zeigen wir zuerst:

Sei $G = (V, E)$ ein ungerichteter Graph mit Knotenmenge V und Kantenmenge E . Sei $S \subseteq V$ eine Teilmenge der Knoten und $C = V - S$ das Komplement von S in V . Dann gilt:

S ist ein Independent Set von G genau dann, wenn C ein Vertex Cover von G ist.



✓ Beweis

$\Rightarrow:$

- Betrachte eine beliebige Kante $(u, v) \in E$. Wir wollen zeigen, dass mindestens einer der beiden Knoten u, v in C liegt.
- Weil S ein Independent Set ist, muss mindestens einer der beiden Knoten u, v nicht in S liegen. Also liegt mindestens einer der beiden Knoten in C .
- Daher ist C ein Vertex Cover. \square

$\Leftarrow:$

- Betrachte zwei Knoten $u \in S$ und $v \in S$. Wir wollen zeigen, dass u und v nicht adjazent sind.
- Aus $u \in S$ und $v \in S$ folgt $u \notin C$ und $v \notin C$.
- u und v können nicht adjazent sein, ansonsten wäre C kein Vertex Cover (weil es die Kante (u, v) nicht überdeckt).
- Also ist S ein Independent Set. \square

Also gilt das Lemma.

Vertex Cover und Independent Set

Es gilt: $\text{VERTEX COVER} \equiv_P \text{INDEPENDENT SET}$.

Dies bedeutet, dass die Probleme **Vertex Cover** und **Independent Set** äquivalent sind. Das

bedeutet, wenn man ein effizientes Verfahren (z.B. eine polynomielle Zeitlösung) für das eine Problem hat, kann man es auch für das andere Problem anwenden.

✓ Beweis der Äquivalenz

Um die Äquivalenz zu zeigen, müssen wir beide Richtungen beweisen:

Beweisidee: Wir zeigen, dass eine Instanz vom einen ins andere umgewandelt werden kann, sodass die Lösung vom umgewandelten die Lösung vom vorigen zeigt.

1. $\text{VERTEX COVER} \leq_P \text{INDEPENDENT SET}$

- Sei (G, k) eine Instanz von VERTEX COVER .
 - G ist ein Graph.
 - k ist die gewünschte Größe des Vertex Covers.
- Sei n die Anzahl der Knoten von G .
- In **Polynomialzeit** generieren wir $(G, n - k)$ als Instanz von INDEPENDENT SET .
 - Das bedeutet, wir suchen ein Independent Set der Größe $n - k$ im selben Graphen G .
- **Die Reduktion ist korrekt:**
 - Ein Graph G hat genau dann ein G Vertex Cover der Größe $\leq k$, wenn G ein Independent Set der Größe $\geq n - k$ hat.
 - (Dies folgt aus dem Konversionslemma, welches die Beziehung zwischen Vertex Cover und Independent Set herstellt.)
- **Die Reduktion ist klarerweise polynomiell:**
 - Sie ersetzt lediglich k durch $n - k$, was in konstanter Zeit geschieht.

2. $\text{INDEPENDENT SET} \leq_P \text{VERTEX COVER}$

- Sei (G, k) eine Instanz von INDEPENDENT SET .
 - G ist ein Graph.
 - k ist die gewünschte Größe des Independent Sets.
- Sei n die Anzahl der Knoten von G .
- In **Polynomialzeit** generieren wir $(G, n - k)$ als Instanz von VERTEX COVER .
 - Das bedeutet, wir suchen ein Vertex Cover der Größe $n - k$ im selben Graphen G .
- **Die Reduktion ist korrekt:**
 - Ein Graph G hat genau dann ein G Independent Set der Größe $\geq k$, wenn G ein Vertex Cover der Größe $\leq n - k$ hat.
 - (Auch dies folgt aus dem Konversionslemma.)
- **Die Reduktion ist klarerweise polynomiell:**

- Auch hier wird lediglich k durch $n - k$ ersetzt.

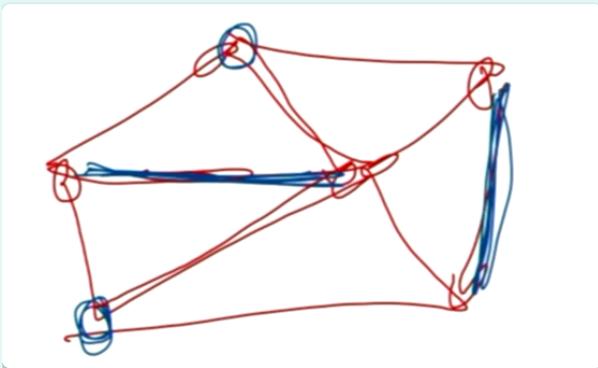
Wir haben also beide Richtungen gezeigt, und es folgt die **Äquivalenz** zwischen Vertex Cover und Independent Set.

Beispiel: Spannbäume und Nicht-Blockierer

⌚ Definition - Nicht-Blockierer

- Gegeben ist ein Graph $G = (V, E)$ mit reellwertigen Kantengewichten $c_e = c_{uv}$ für $e = (u, v) \in E$.
- Ein **Nicht-Blockierer** (N) ist eine Teilmenge der Kanten $N \subseteq E$, sodass es für alle Knotenpaare $u, v \in V$ einen $u - v$ -Pfad in G gibt, der keine Kante aus N enthält. (Ein Nicht-Blockierer "blockiert" also keinen Pfad, d.h., es gibt immer einen Pfad, der die Kanten in N meidet.)
- **Kosten eines Nicht-Blockierers:** Die Kosten eines Nicht-Blockierers sind gegeben durch $\sum_{e \in N} c_e$.
- Ein **maximaler Nicht-Blockierer** ist ein Nicht-Blockierer mit größten Kosten.

Wenn man jeden Knoten von jedem Knoten immer noch erreichen kann auch wenn man eine Kante weglassen würde. Beispielsweise bei Straßennetz und Baustelle:



⌚ Problem MNB (Maximaler Nicht-Blockierer)

- **Definition:** Gegeben ist ein gewichteter Graph G und eine Zahl k . Besitzt G einen Nicht-Blockierer mit Kosten $\geq k$?
- **Frage:** Ist das Problem MNB in Polynomialzeit lösbar?

Reduktion auf Spannbäume

Wir zeigen, dass MNB in Polynomialzeit lösbar ist, indem wir es auf ein bekanntes Problem (MST*) reduzieren.

⌚ Definition Spannbaum

- **Definition:** Gegeben ist ein Graph $G = (V, E)$ mit reellwertigen Kantengewichten $c_e = c_{uv}$ für $e = (u, v) \in E$.

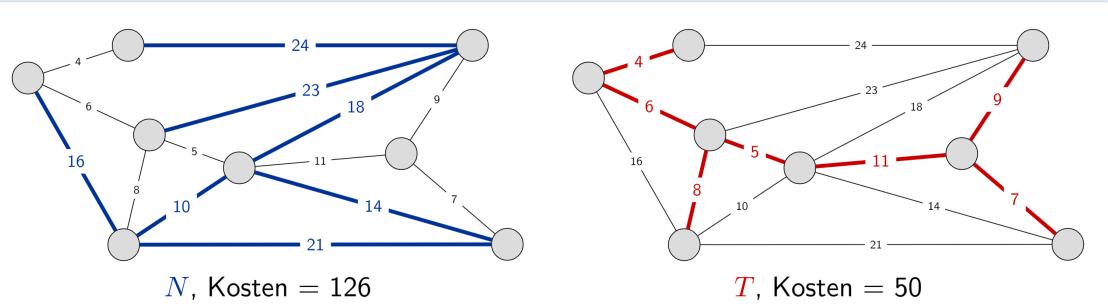
- Ein **Spannbaum** (T) ist eine Teilmenge der Kanten $T \subseteq E$, sodass $G_T = (V, T)$ ein Baum ist. (Ein Baum ist ein zusammenhängender, zyklenfreier Graph.)
- **Kosten eines Spannbaums:** Die Kosten des Baumes sind gegeben durch $\sum_{e \in T} c_e$.
- Ein **minimaler Spannbaum** ist ein Spannbaum mit kleinsten Kosten.

② Problem MST* (Minimaler Spannbaum mit Kosten)

- **Definition:** Gegeben ist ein gewichteter Graph G und eine Zahl k . Besitzt G einen Spannbaum mit Kosten $\leq k$?
- **Lösbarkeit:** Wir wissen aus dem Kapitel „Greedy-Algorithmen“, dass MST^* in Polynomialzeit gelöst werden kann (z.B. mit dem Kruskal- oder Prim-Algorithmus).

① Konversionslemma

- Sei $G = (V, E)$ ein gewichteter Graph.
- Sei $N \subseteq E$ ein Nicht-Blockierer und $T = E - N$ (d.h., T sind alle Kanten, die *nicht* im Nicht-Blockierer sind).
- **Aussage:** N ist ein maximaler Nicht-Blockierer genau dann, wenn T ein minimaler Spannbaum ist.
- **Kostenbeziehung:** Die Kosten von N sind genau $K' := \sum_{e \in E} c_e$ minus den Kosten von T .
 - $\text{Kosten}(N) = \sum_{e \in E} c_e - \text{Kosten}(T)$



Äquivalenz von MNB und MST*

Es gilt: $MNB \equiv_P MST^*$ (MNB ist polynomiell äquivalent zu MST*).

✓ Beweis der Äquivalenz

1. $MNB \leq_P MST^*$

- Wir reduzieren eine Instanz (G, k) von MNB auf die Instanz $(G, K_{\text{gesamt}} - k)$ von MST^* .
 - K_{gesamt} ist die Summe der Gewichte aller Kanten in G .

- Wenn wir einen Nicht-Blockierer mit Kosten $\geq k$ suchen, entspricht das dem Finden eines Spannbaums mit Kosten $\leq K_{gesamt} - k$.
- **Intuition:** Ein Nicht-Blockierer mit hohen Kosten bedeutet, dass die *verbleibenden* Kanten (die nicht im Nicht-Blockierer sind und den Spannbaum bilden) geringe Kosten haben müssen.

2. $MST^* \leq_P MNB$

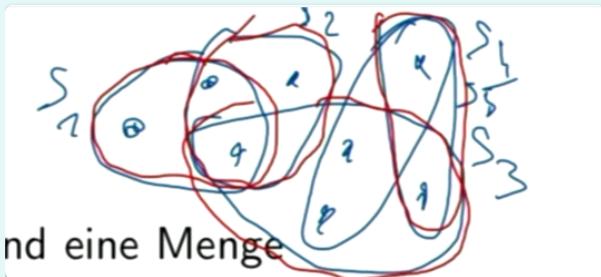
- Wir reduzieren eine Instanz (G, k) von MST^* auf die Instanz $(G, K_{gesamt} - k)$ von MNB.
 - Wenn wir einen Spannbaum mit Kosten $\leq k$ suchen, entspricht das dem Finden eines Nicht-Blockierers mit Kosten $\geq K_{gesamt} - k$.
- **Intuition:** Ein Spannbaum mit geringen Kosten bedeutet, dass die Kanten, die *nicht* im Spannbaum sind (und den Nicht-Blockierer bilden), hohe Kosten haben müssen.

Es folgt daher: **MNB ist in Polynomialzeit lösbar**, da es auf MST^* reduziert werden kann, welches in Polynomialzeit lösbar ist.

Set Cover (Mengenüberdeckungsproblem)

⌚ Definition des Set Cover

- Gegeben sei:
 - Eine Menge U von Elementen (auch "Universum" genannt).
 - Eine Menge $S = \{S_1, S_2, \dots, S_m\}$ von Teilmengen von U . Jede S_i ist also $S_i \subseteq U$.
- Ein **Set Cover** (C) ist eine Teilmenge $C \subseteq S$.
 - Dies bedeutet, C ist eine Menge von Mengen aus S .
- Die Bedingung für ein Set Cover ist, dass die **Vereinigung aller Mengen in C dem Universum U entspricht**.
 - Formal: $\bigcup_{X \in C} X = U$.
- Man sagt auch: C ist ein Set Cover von S .



(?) Das Set Cover Problem (Entscheidungsproblem)

- **Gegeben:**
 - Eine Menge U von Elementen.
 - Eine Menge $S = \{S_1, S_2, \dots, S_m\}$ von Teilmengen von U .
 - Eine ganze Zahl k .
- **Frage:** Existiert eine Teilmenge $C \subseteq S$ mit $|C| \leq k$, sodass die Vereinigung von C gleich U ist?
 - Das Problem fragt also, ob es ein Set Cover gibt, das aus **höchstens k** der gegebenen Teilmengen besteht.

☰ Beispielhafte Anwendungen des Set Cover Problems

Das Set Cover Problem ist ein klassisches Problem in der Informatik und hat viele praktische Anwendungen.

- **Szenario:** Softwareentwicklung
 - Es gibt m verfügbare Softwarekomponenten.

- Menge U besteht aus n Eigenschaften, die unser Softwaresystem haben sollte. (z.B. "Datenbankzugriff", "Benutzeroauthentifizierung", "Reporting-Funktion").
 - Die i -te Softwarekomponente bietet eine Menge $S_i \subseteq U$ von Eigenschaften an. (z.B. Komponente 1 bietet "Datenbankzugriff" und "Caching", Komponente 2 bietet "Benutzeroauthentifizierung" und "Verschlüsselung").
 - **Ziel:** Erreiche alle n Eigenschaften mit *maximal* k Komponenten.
 - Dies entspricht genau der Fragestellung des Set Cover Problems: Finde die kleinste Anzahl von Komponenten, deren kombinierte Eigenschaften alle gewünschten Eigenschaften abdecken.
-

Weiteres Beispiel:

Möglichst wenige Pizzen aus Speisekarten aussuchen, dass möglichst viele Zutaten drauf sind.

Weiteres Beispiel:

$$\begin{aligned} U &= \{1, 2, 3, 4, 5, 6, 7\} \\ k &= 2 \\ S_1 &= \{3, 7\} & S_4 &= \{2, 4\} \\ S_2 &= \{3, 4, 5, 6\} & S_5 &= \{5\} \\ S_3 &= \{1\} & S_6 &= \{1, 2, 6, 7\} \end{aligned}$$

Vertex Cover auf Set Cover reduzieren

Behauptung: VERTEX COVER \leq_P SET COVER.

Beweis: Gegeben sei eine Vertex Cover Instanz (G, k) mit $G = (V, E)$.

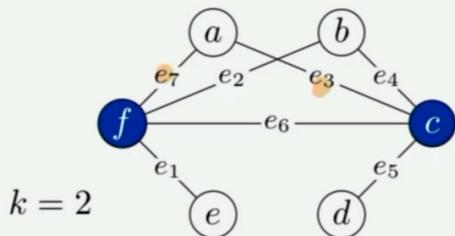
Wir konstruieren eine Instanz von SET COVER.

- $k = k$,
- $U = E$,
- Für jedes $v \in V$ erzeugen wir eine Menge $S_v = \{e \in E : e \text{ inzident zu } v\}$
- \mathcal{S} ist die Menge aller S_v für $v \in V$.
- Die Reduktion ist klarerweise *polynomiell*.

Wir wollen hier das **Vertex Cover als Set Cover darstellen**.

Für jeden Knoten nehmen wir alle Kanten die zu dem Knoten inzident sind. Beispiel Kante e_3 und e_7

VERTEX COVER



SET COVER

$$\begin{aligned} U &= \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\} \quad k = 2 \\ S_a &= \{e_3, e_7\}, \quad S_b = \{e_2, e_4\} \\ S_c &= \{e_3, e_4, e_5, e_6\}, \quad S_d = \{e_5\} \\ S_e &= \{e_1\}, \quad S_f = \{e_1, e_2, e_6, e_7\} \end{aligned}$$

und das machen wir jetzt für alle Knoten. Und das k übernehmen wir einfach und lassen wir gleich.

- **Korrektheit:** G hat ein Vertex Cover der Größe $\leq k$ genau dann wenn \mathcal{S} ein Set Cover der Größe $\leq k$ hat.
- \Rightarrow : Sei $C = \{v_1, \dots, v_k\} \subseteq V$ ein Vertex Cover von G . Dann ist $\mathcal{C} = \{S_{v_1}, \dots, S_{v_k}\}$ ein Set Cover von \mathcal{S} .
- \Leftarrow : Sei $\mathcal{C} = \{S_{v_1}, \dots, S_{v_k}\}$ ein Set Cover von \mathcal{S} . Dann ist $C = \{v_1, \dots, v_k\} \subseteq V$ ein Vertex Cover von G . \square

i Bemerkung

Jede Setcover Instanz die wir durch Reduktion bekommen, hat bestimmte Eigenschaften.

Nicht jede Instanz wird diese Eigenschaften haben. Eine die wir bei unserem Beispiel herauslesen können ist, dass jede Kante in genau 2 Mengen vorkommt. Weil wir für jeden Knoten die Menge aller inzidenten Kanten finden und jede ist zu genau 2 inzident also wird das stimmen. Wenn wir aber im allgemeinen das machen, muss das nicht stimmen.

- Nicht jede Set Cover Instanz kann durch die Reduktion von Vertex Cover entstehen.
- Daher sprechen wir hier von einer "Reduktion eines Spezialfalls auf den allgemeinen Fall".

Reduktion mit Gadgets

Mit Gadgets sind in diesem Fall diese Dreiecke gemeint und ist ein allgemeiner Begriff um kleine Bausteine zu beschreiben, die dann eine ganze Instanz implementieren. Aber es gibt keine genau Definition von Gadgets.

Erfüllbarkeitsproblem (satisfiability)

siehe [GDS](#)

- **Literal:** Eine boolesche Variable (x_i) oder ihre Negation ($\neg x_i$).
 - Beispiele: $x_i, \neg x_i$
- **Klausel:** Eine Disjunktion (logisches ODER, \vee) von Literalen.
 - Beispiel: $C_j = x_1 \vee x_2 \vee x_3$
- **Konjunktive Normalform (KNF):** Eine aussagenlogische Formel Φ , bei der Klauseln konjunktiv (logisches UND, \wedge) verknüpft werden.
 - Beispiel: $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$
- **Wahrheitsbelegung (truth assignment):** Eine Wahrheitsbelegung ist eine Funktion f , die jeder Variable einen Wahrheitswert `true` oder `false` zuordnet.
- **Erfüllen einer Formel:** Eine Wahrheitsbelegung f erfüllt eine KNF-Formel Φ , falls jede Klausel von Φ mindestens eine Variable x mit $f(x) = \text{true}$ oder eine negierte Variable $\neg x$ mit $f(x) = \text{false}$ enthält.

⌚ Das Erfüllbarkeitsproblem (SAT)

- **SAT (satisfiability):** Gegeben ist eine KNF-Formel Φ . Gibt es eine Wahrheitsbelegung, die Φ erfüllt?
- **3-SAT:** SAT, bei dem jede Klausel genau 3 Literale enthält.
 - Jedes Literal muss sich auf eine unterschiedliche Variable beziehen.
 - Beispiel: $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$
 - **Erfüllende Wahrheitsbelegung:** $f(x_1) = \text{true}, f(x_2) = \text{true}, f(x_3) = \text{false}$.

Bedeutung von SAT

Das Erfüllbarkeitsproblem (SAT) ist in zweierlei Hinsicht von großer Bedeutung:

1. **Mächtige heuristische Algorithmen (SAT-Solver):** Für SAT existieren leistungsstarke heuristische Algorithmen (SAT-Solver), die große, "strukturierte" Instanzen lösen können.
 - → Daher ist SAT ein beliebtes Problem, um andere Probleme darauf zu reduzieren (Lösung durch Reduktion).
2. **Nicht-Handhabbarkeit für allgemeine Instanzen:** Andererseits wird SAT für allgemeine Instanzen als nicht-handhabbar angesehen (SAT ist "NP-vollständig", was auf den

nächsten Folien erläutert wird).

- → Daher ist SAT ein beliebtes Problem, das auf andere Probleme reduziert wird, um deren Nicht-Handhabbarkeit zu zeigen.

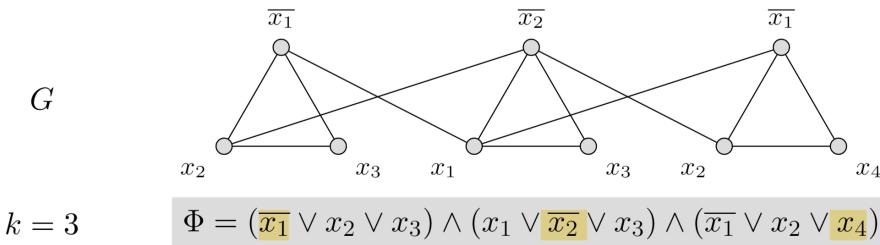
3-Sat auf Independent Set reduzieren

Behauptung: $3\text{-SAT} \leq_P \text{INDEPENDENT SET}$.

Beweis: Gegeben sei eine Instanz Φ von 3-SAT mit k Klauseln. Wir konstruieren eine Instanz (G, k) von INDEPENDENT SET.

- G enthält 3 Knoten für jede Klausel (einen für jedes Literal).
- Verbinde 3 Literale in einer Klausel zu einem Dreieck.
- Verbinde ein Literal mit jeder seiner Negationen.

Die Reduktion ist klarerweise *polynomiel*.



- Als k definieren wir immer die **Anzahl der Klauseln**

Korrektheit: G enthält ein Independent Set der Größe k genau dann wenn ϕ erfüllbar ist.

✓ Beweis

(\Rightarrow) Sei S ein Independent Set der Größe k .

- S enthält *genau einen Knoten pro Dreieck* (S kann höchstens einen Knoten pro Dreieck enthalten, ansonsten ist S nicht unabhängig; S muss mindestens einen Knoten pro Dreieck enthalten, da $|S| = k$ und es k Dreiecke gibt).
- Wir konstruieren eine Wahrheitsbelegung der Variablen, indem wir $x = \text{true}$ setzen, falls $x \in S$, und $x = \text{false}$ setzen, falls $\neg x \in S$.
- Wegen der Kanten zwischen den Dreiecken kann es zu *keinen widersprüchlichen Belegungen ein und derselben Variable* kommen.
- Wir setzen die *übrigen Variablen* beliebig auf true oder false .
- Diese *Wahrheitsbelegung erfüllt alle Klauseln*.

(\Leftarrow) Gegeben sei eine Wahrheitsbelegung f , die Φ erfüllt.

- Wir konstruieren ein Independent Set S , indem wir von jedem Dreieck einen Knoten l mit $l = x$ und $f(x) = \text{true}$ oder einen Knoten l mit $l = \neg x$ und $f(x) = \text{false}$ wählen. (So einen Knoten l gibt es immer, da f erfüllend).

- S ist unabhängig, weil die Kanten zwischen den Dreiecken jeweils zwischen einer Variable und ihrer Negation verlaufen.
- $|S| = k$, weil wir von *jedem Dreieck einen Knoten wählen*.

Optimierungsprobleme

Ja/Nein-Problem: Existiert ein Vertex Cover der Größe $\leq k$?

Optimierungsproblem: Finde kleinstes Vertex Cover.

Klar:

- Ja/Nein-Problem kann mit dem Optimierungsproblem gelöst werden.
- Berechne kleinstes Vertex Cover C und antworte „Ja“ falls $|C| \leq k$ ist.

Umgekehrte Richtung: Löse Optimierungsproblem mittels (mehrmaligem) Lösen des Ja/Nein-Problems.

Können wir die beiden Probleme jetzt aufeinander reduzieren?

Das Ja/Nein Problem kann sehr einfach auf das Optimierungsproblem reduzieren, weil man einfach den Algorithmus ausführen kann von dem OP und dann schauen ob k kleiner/größer ist, aber umgekehrt ist das schon bisschen schwerer.

Optimierungsproblem für Vertex Cover lösen

Ausgangslage: Es existiert ein Algorithmus $VC(G, k)$, der das Ja/Nein-Problem für ein Vertex Cover der Größe $\leq k$ löst.

Ziel: Wir möchten mittels $VC(G, k)$ ein kleinstes Vertex Cover finden.

Verwendete Eigenschaften für jeden Graphen $G = (V, E)$:

1. Sei $v \in V$. Falls C ein Vertex Cover von $G - v$ ist, dann ist $C \cup \{v\}$ ein Vertex Cover von G .
2. Falls G ein Vertex Cover der Größe $k \geq 1$ besitzt, dann gibt es ein $v \in V$, sodass $G - v$ ein Vertex Cover der Größe $k - 1$ besitzt.

Der Algorithmus OptVC(G)

Der Algorithmus `OptVC(G)` berechnet ein kleinstes Vertex Cover von G mittels mehrmaligen Aufrufs von $VC(G, k)$.

```
OptVC( $G$ ):  

for  $k \leftarrow 0$  bis  $n - 1$   

    if VC( $G, k$ )  

        return FindVC( $G, k$ )
```

```
FindVC( $G, k$ ):  

if  $k = 0$   

    return  $\emptyset$   

else  

    foreach  $v \in V(G)$   

        if VC( $G - v, k - 1$ )  

            return  $\{v\} \cup$  FindVC( $G - v, k - 1$ )
```

Komplexität: Insgesamt wird $VC(G, k)$ höchstens $O(n) + O(n^2) = O(n^2)$ mal aufgerufen.

- Analog kann für viele andere Optimierungsprobleme vorgegangen werden.
- Das rechtfertigt unseren Fokus auf Ja/Nein Probleme.

Definition von NP

NP-Probleme

⌚ Definition eines NP-Problems

- Ein Ja/Nein-Problem (oder Entscheidungsproblem) ist ein **NP-Problem**, falls wir Ja-Instanzen mit Hilfe eines **Zertifikats** effizient überprüfen können.
 - "Effizient" bedeutet hier in Polynomialzeit.

Zertifikat

- Ein **Zertifikat** t für eine Instanz x ist ein beliebiger Input.
- Die Größe m des Zertifikats t muss polynomiell in der Größe n von x beschränkt sein, das heißt $m \leq p(n)$ für ein Polynom p .
 - Dies stellt sicher, dass das Zertifikat nicht übermäßig groß ist und somit effizient verarbeitet werden kann.

Zertifizierer

- Ein **Zertifizierer** $C(x, t)$ ist ein Polynomialzeitalgorithmus.
- Er überprüft, ob eine Ja-Instanz x mit Hilfe eines Zertifikats t gültig ist.
 - Der Zertifizierer nimmt die Instanz x und das potenzielle Zertifikat t als Eingabe und entscheidet, ob t ein gültiger "Beweis" dafür ist, dass x eine Ja-Instanz ist.

ⓘ Anmerkung zur Notation

- **NP** steht für „nicht-deterministisch polynomielle“ Zeit.
 - Dies bezieht sich auf die ursprüngliche Definition, bei der ein nicht-deterministischer Turing-Automat das Problem in Polynomialzeit lösen kann. Die hier gegebene Definition über Zertifikate ist jedoch äquivalent und oft intuitiver.
- Für ein NP-Problem X sagen wir auch: „ X ist in NP“.

Eigenschaften des Zertifizierers $C(x, t)$

Genauer gesagt, der Zertifizierer $C(x, t)$ soll folgende Eigenschaften haben:

- **Für jede Ja-Instanz x gibt es ein Zertifikat t** (von polynomieller Länge), welches den Zertifizierer zum Akzeptieren bringt.

- Wenn eine Instanz tatsächlich eine "Ja"-Antwort hat, muss es einen "Beweis" (Zertifikat) geben, den der Zertifizierer als gültig erkennt.
- Man kann sich das so vorstellen: "Der Zertifizierer kann überzeugt werden."
- Für keine Nein-Instanz x gibt es ein Zertifikat t , welches den Zertifizierer zum Akzeptieren bringt.
- Wenn eine Instanz eine "Nein"-Antwort hat, darf es keinen "Beweis" (Zertifikat) geben, der den Zertifizierer fälschlicherweise dazu bringt, die Instanz als "Ja" zu akzeptieren.
- Man kann sich das so vorstellen: "Der Zertifizierer kann nicht ausgetrickst werden."

☰ Beispiel dazu: SAT und HAM-Cycle

SAT: Gegeben sei eine Formel Φ in konjunktiver Normalform. Ist diese Formel erfüllbar?

Zertifikat: Eine Wahrheitsbelegung f für die n booleschen Variablen, die Φ erfüllt.

Zertifizierer: Überprüfe, ob f die Formel Φ erfüllt.

Beispiel:

$\Phi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4)$; Instanz s
 $f(x_1) = \text{true}, f(x_2) = \text{true}, f(x_3) = \text{false}, f(x_4) = \text{true}$ Zertifikat t

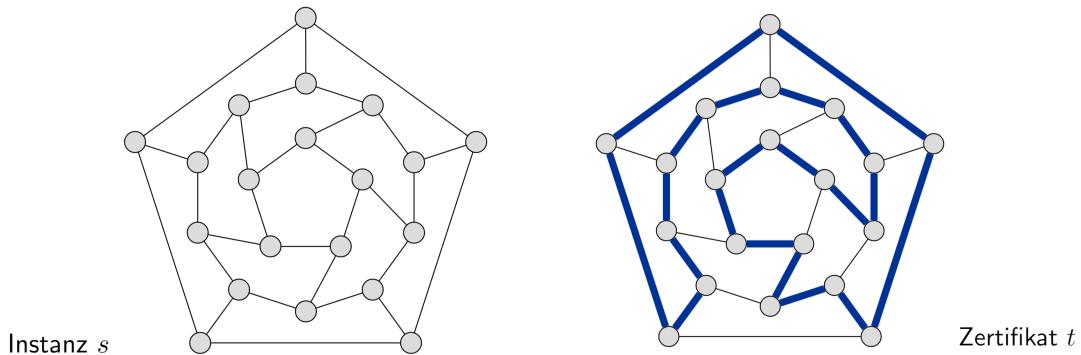
Schlussfolgerung: SAT ist in NP.

HAM-CYCLE: Gegeben sei ein ungerichteter Graph G . Existiert ein Kreis C in G , der alle Knoten von G genau einmal enthält? So ein Kreis wird als Hamiltonkreis bezeichnet.

Zertifikat: Ein Hamiltonkreis.

Zertifizierer: Überprüfe, ob der Hamiltonkreis jeden Knoten in V genau einmal enthält und dass es eine Kante zwischen jedem Paar von direkt aufeinander folgenden Knoten in dem Hamiltonkreis und auch vom ersten zum letzten Knoten gibt.

Beispiel:



Schlussfolgerung: HAM-CYCLE ist in NP.

Quiz zur Definition:

Frage 6: Welche der folgenden Probleme sind in NP?

- ✓ (A) INDEPENDENT SET
- ✗ (B) Gegeben ein Graph G , finde ein kleinstes Vertex Cover von G .
- ✗ (C) Gegeben ein Graph G und $k > 0$. Hat jedes Vertex Cover von G mindestens k Knoten?
- ✓ (D) Gegeben die Präferenzen von n Kindern und n Gastfamilien, existiert ein Stable Matching?

- B stimmt nicht weil kein ja/nein
- C stimmt nicht, weil man nicht Existenz beweisen muss --> alle anschauen --> nicht zertifizierbar.

Weitere Eigenschaften zu P und NP

ⓘ Unterscheidung P und NP

P: Ja/Nein-Probleme, für die polynomielle Algorithmen existieren.

NP: Ja/Nein-Probleme, für die polynomielle Zertifizierer existieren.

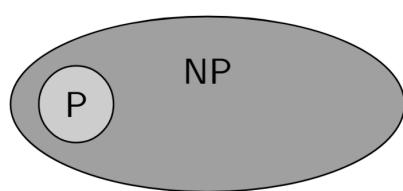
Es gilt: $P \subseteq NP$.

Beweis: Wir betrachten ein beliebiges Problem X in P.

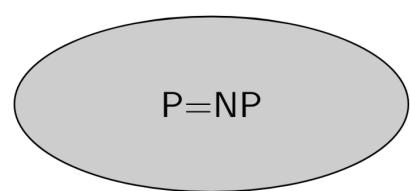
- Nach Definition existiert ein Polynomialzeit-Algorithmus $A(s)$, der X löst.
- Zertifikat: $t = \emptyset$, Zertifizierer $C(s, t) = A(s)$. \square

(?) Gilt $P = NP$? [Cook 1971, Levin 1973]

- Ist das Ja/Nein-Problem so leicht wie das Zertifizierungsproblem?
- Für die Beantwortung der Frage ist 1 Million US Dollar ausgeschrieben (Clay Mathematics Institute).



Falls $P \neq NP$



Falls $P = NP$

- **Falls ja:** Effiziente Algorithmen für Vertex Cover, Ham-Cycle, TSP*, SAT, ...
- **Falls nein:** Keine effizienten Algorithmen für Vertex Cover, Ham-Cycle, TSP*, SAT, ...

Vorherrschende Meinung zu P = NP: Wahrscheinlich "nein".

NP-Vollständigkeit

⌚ Definition - NP-Schwer

- Ein Ja/Nein-Problem Y ist **NP-schwer**, falls für jedes Problem X in NP gilt, dass $X \leq_p Y$.
- Das heißt, jedes NP-Problem X kann in Polynomialzeit auf Y reduziert werden.
- NP-schwere Probleme sind also gewissermaßen "mindestens so schwer" wie alle Probleme in NP.

⌚ Definition - NP-vollständig

- Ein Problem Y ist **NP-vollständig**, falls es sowohl in NP liegt als auch NP-schwer ist.
- Die NP-vollständigen Probleme sind also gewissermaßen die "schwersten" Probleme in NP.
- Nach dieser Definition können nur Ja/Nein-Probleme NP-vollständig sein.

ⓘ Theorem

Sei Y ein NP-vollständiges Problem. Y ist in polynomieller Zeit lösbar genau dann, wenn $P = NP$.

Beweis:

- (\Leftarrow) Wenn $P = NP$, dann kann Y in polynomieller Zeit gelöst werden, da Y sich in NP befindet.
- (\Rightarrow) Angenommen, Y kann in polynomieller Zeit gelöst werden. Sei X ein beliebiges Problem in NP. Da $X \leq_p Y$, können wir X in Polynomialzeit lösen (reduziere X auf Y in Polynomialzeit, löse Y in Polynomialzeit). Das impliziert $NP \subseteq P$. Wir wissen bereits, dass $P \subseteq NP$. Daher $P = NP$.

Gibt es ein NP-vollständiges Problem?

Allgemeine Überlegung: Das ist nicht von vornherein klar. Es könnte z.B. mehrere schwerste NP-Probleme geben, die nicht jeweils aufeinander reduzierbar sind.

Theorem: SAT ist NP-vollständig. [Cook 1971, Levin 1973]

NP-Vollständigkeit nachweisen

- **Anmerkung:** Sobald ein Problem als NP-vollständig nachgewiesen wurde, können andere Probleme einfacher als NP-vollständig erkannt werden, da sie wie Dominosteine "umfallen".

Rezept zum Nachweis der NP-Vollständigkeit eines Problems Y :

1. **Schritt 1:** Zeige, dass Y in NP ist.
2. **Schritt 2:** Wähle ein NP-vollständiges Problem X .
3. **Schritt 3:** Beweise, dass $X \leq_p Y$.

Rechtfertigung:

- Wenn X ein NP-vollständiges Problem ist und Y ein Problem in NP mit der Eigenschaft $X \leq_p Y$, dann ist Y NP-vollständig.

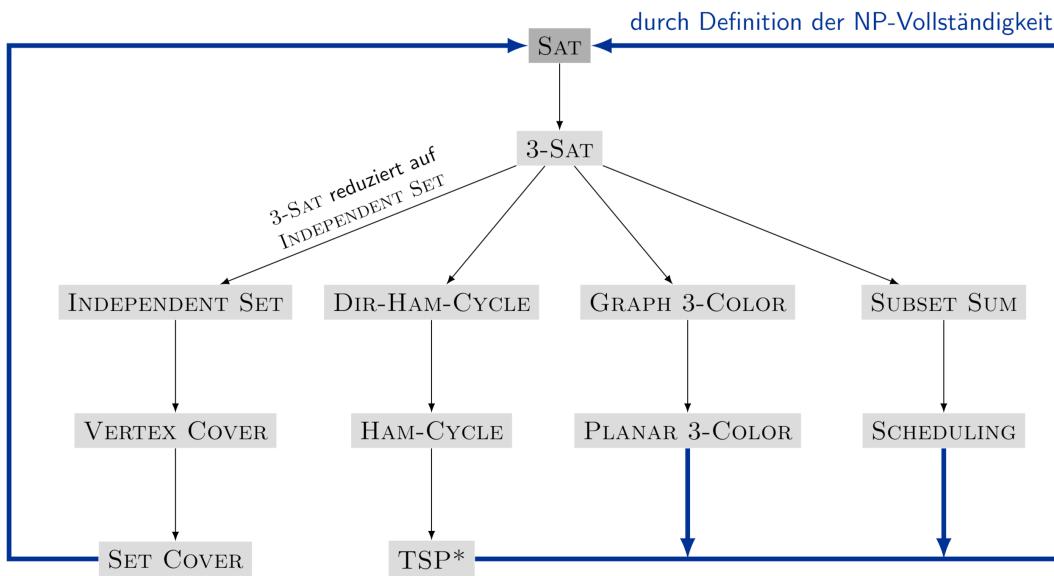
Beweis

Sei W ein beliebiges Problem in NP.

Dann gilt $W \leq_p X \leq_p Y$.

- $W \leq_p X$: durch Definition von NP-Vollständigkeit von X (da X NP-vollständig ist, kann jedes Problem in NP polynomial-zeitreduziert werden auf X).
- $X \leq_p Y$: durch Annahme (unser Beweisschritt 3).
- Durch Transitivität gilt: $W \leq_p Y$.
 - (Transitivität bedeutet hier, wenn W auf X reduziert werden kann und X auf Y , dann kann W auch direkt auf Y reduziert werden.)
- Daher ist Y NP-vollständig.

Beobachtung: Alle Probleme in der Abbildung sind NP-vollständig und lassen sich polynomiell auf einander reduzieren.



3-Color (3-Knotenfärbung)

Da wir nur bis hier in der vo gekommen sind werde ich hier diese Folien nicht behandeln:

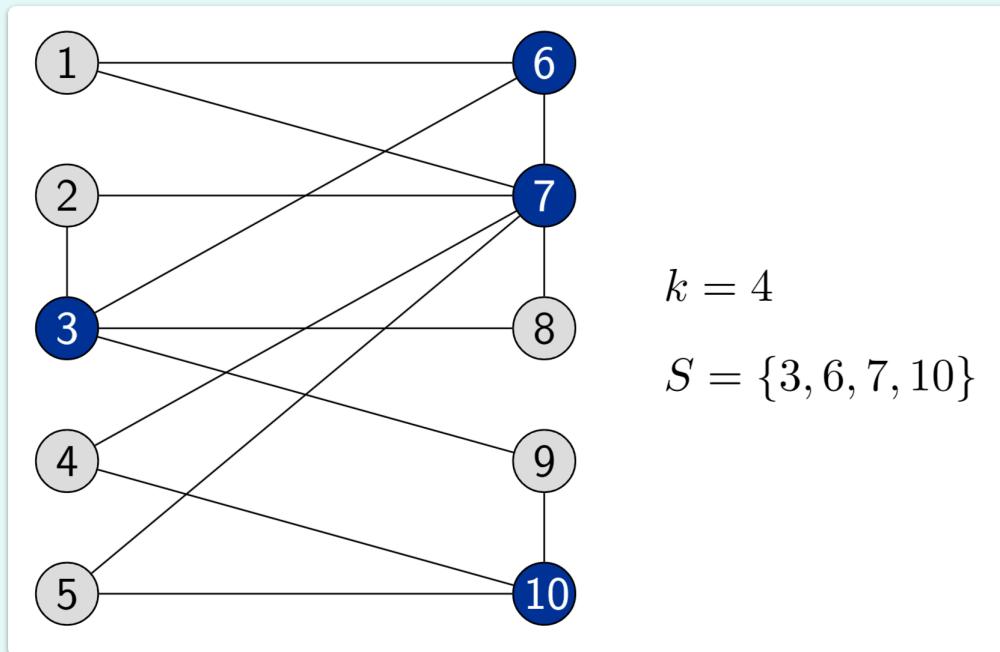
- [AD_09_PolynomialzeitreduktionUndNP, p.64](#)
- [AD_09_PolynomialzeitreduktionUndNP, p.65](#)
- [AD_09_PolynomialzeitreduktionUndNP, p.66](#)
- [AD_09_PolynomialzeitreduktionUndNP, p.67](#)
- [AD_09_PolynomialzeitreduktionUndNP, p.68](#)
- [AD_09_PolynomialzeitreduktionUndNP, p.69](#)
- [AD_09_PolynomialzeitreduktionUndNP, p.70](#)
- [AD_09_PolynomialzeitreduktionUndNP, p.71](#)
- [AD_09_PolynomialzeitreduktionUndNP, p.72](#)
- [AD_09_PolynomialzeitreduktionUndNP, p.74](#)
- [AD_09_PolynomialzeitreduktionUndNP, p.75](#)
- [AD_09_PolynomialzeitreduktionUndNP, p.76](#)
- [AD_09_PolynomialzeitreduktionUndNP, p.77](#)

10. NP-Vollständigkeit Spezialfälle

Finden von kleinen Vertex Covers

💡 Vertex Cover (Knotenüberdeckung)

- **Gegeben:**
 - Ein Graph $G = (V, E)$
 - V : Menge der Knoten
 - $|V| = n$: Anzahl der Knoten
 - E : Menge der Kanten
 - Eine ganze Zahl k
- **Frage:** Existiert eine Teilmenge von Knoten $S \subseteq V$ mit folgenden Eigenschaften?
 - **Größe:** $|S| \leq k$
 - **Überdeckung:** Für jede Kante $(u, v) \in E$ gilt:
 - $u \in S$ oder
 - $v \in S$
 - (oder beides)
- **Ziel:** kleines VC mit $k << n$



S ist VC \Leftrightarrow wenn V/S ist independent Set.

Finden kleiner Vertex Covers

Frage: Was ist möglich, wenn k klein ist?

Brute-Force Ansatz

- **Laufzeit:** $O(kn^{k+1})$

- **Vorgehen:**

- Probiere alle $\binom{n}{k} = O(n^k)$ Teilmengen der Größe k aus. (Das ist die Anzahl der Möglichkeiten, k Elemente aus n Elementen auszuwählen)
- Benötigt $O(kn)$ Zeit, um zu überprüfen, ob eine Teilmenge ein Vertex Cover ist.
- **Hier:** $E = |m| \leq kn$, da jeder Knoten in S maximal $n - 1$ Nachbarn hat (Hierbei ist E die Anzahl der Kanten und S die Menge der Knoten im Graphen. Die Ungleichung $E \leq kn$ gibt eine obere Schranke für die Anzahl der Kanten an, wenn die Anzahl der Knoten n und eine bestimmte Eigenschaft k gegeben ist. Die Aussage $E \leq kn$ ist eine Vereinfachung oder Annahme, die sich auf die Überprüfung bezieht).

Zu der Zeit kommt es dadurch:

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdot \dots \cdot (n-k+1)}{k!} \implies \binom{n}{k} \leq n^k$$

Ziel

- Beschränke die exponentielle Abhängigkeit von k .

:≡ Beispiel

- $n = 1000, k = 10$
 - Brute-Force: $kn^{k+1} = 10 \cdot 1000^{11} = 10 \cdot (10^3)^{11} = 10 \cdot 10^{33} = 10^{34} \rightarrow$ **undurchführbar**.
 - Besser: $2^k kn \approx 2^{10} \cdot 10 \cdot 1000 = 1024 \cdot 10000 \approx 10^4 \cdot 10^4 = 10^8 \rightarrow 2^k kn \approx 10^7$ (Dies ist eine Optimierung oder ein verbesserter Algorithmus im Vergleich zum Brute-Force, der eine kleinere Laufzeit hat, aber immer noch von k abhängt).

Anmerkung

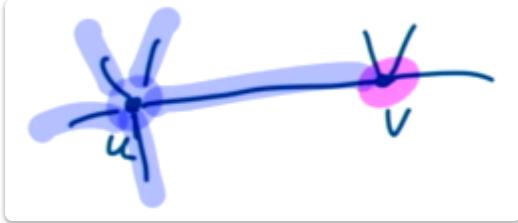
- Wenn k eine **Konstante** ist, dann ist der Algorithmus **polynomiell**.
- Wenn k eine **kleine Konstante** ist, dann ist er auch **praktikabel**.

Eine aktuelle CPU schafft so ca 1.000.000 MIPS (Millionen Operationen pro Sekunde) also 10^{12} Ops pro Sek. $\implies 10^{22}$ Sekunden Rechenzeit um das mit Brute-Force auszurechnen. (Schätzung: Das Universum ist 10^{18} Sekunden alt)

Beweis

Behauptung: Sei (u, v) eine Kante von Graph G . G hat ein Vertex Cover der Größe $\leq k$ genau dann, wenn zumindest einer der Graphen $G - \{u\}$ und $G - \{v\}$ ein Vertex Cover der Größe $\leq k - 1$ hat.

Man muss sich dann entscheiden entweder u oder v ins VC hinzuzufügen:



✓ Beweis

⇒ (Hinrichtung)

- Angenommen G hat ein Vertex Cover S der Größe $\leq k$.
- S enthält entweder u oder v (oder beide). Angenommen, S enthält u .
- Dann ist $S - \{u\}$ ein Vertex Cover von $G - \{u\}$.

⇐ (Rückrichtung)

- Angenommen S ist ein Vertex Cover von $G - \{u\}$ der Größe $\leq k - 1$.
- Dann ist $S \cup \{u\}$ ein Vertex Cover von G .

Eigenschaften von Vertex Cover

Behauptung: Wenn G ein Vertex Cover der Größe k hat, dann hat G höchstens $|E| \leq k(n - 1)$ Kanten.

Beweis: Jeder Knoten überdeckt höchstens $n - 1$ Kanten.

falls $|E| > k * (n - 1)$ gibt es kein k-VC

Algorithmus

Behauptung: Der folgende Algorithmus bestimmt mit einer Laufzeit in $O(2^k kn)$, ob G ein Vertex Cover der Größe $\leq k$ hat.

```

Vertex-Cover( $G, k$ ):
  if  $G$  enthält keine Kanten
    return true
  if  $G$  enthält  $> k(n - 1)$  Kanten
    return false
  Sei  $(u, v)$  eine beliebige Kante von  $G$ 
   $a \leftarrow \text{Vertex-Cover}(G - \{u\}, k - 1)$ 
   $b \leftarrow \text{Vertex-Cover}(G - \{v\}, k - 1)$ 
  return  $a \text{ OR } b$ 
```

Beweis:

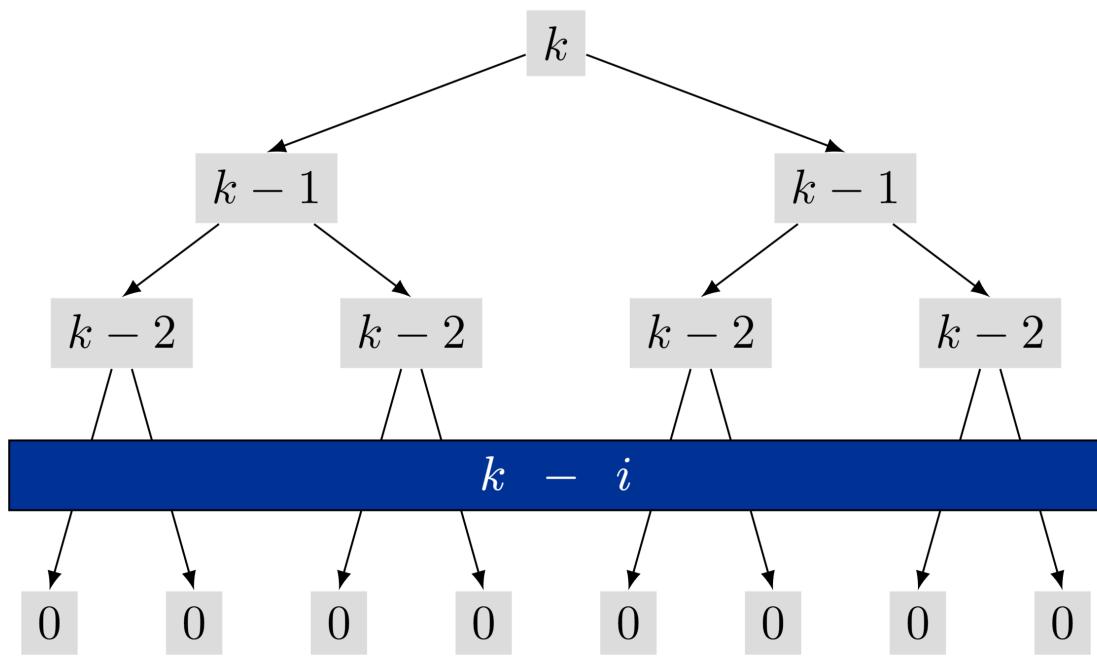
- Korrektheit folgt aus den zwei vorherigen Behauptungen.
- Es existieren $\leq 2^{k+1}$ Knoten im Rekursionsbaum. Jeder Aufruf benötigt $O(kn)$ Zeit. \square

Am Ende geben wir den zurück der `true` zurückgibt.

Wenn wir k Rekursionsschritte gemacht haben sind wir fertig.

Rekursionsbaum

$$T(n, k) \leq \begin{cases} c & \text{falls } k = 0 \\ cn & \text{falls } k = 1 \\ 2T(n - 1, k - 1) + ckn & \text{falls } k > 1 \end{cases} \Rightarrow T(n, k) \leq 2^k ckn$$



In jedem Schritt reduzieren wir den Wert k um 1, also sind wir nach $k + 1$ Level da wo $k = 0$ ist und können im Worst Case da abbrechen. Jede Verzweigung stellt ein wählen zwischen Kante u oder Kante v dar.

Das kann man auch noch mit vollständiger Induktion beweisen

Beweis mit vollständiger Induktion über k Beweis mit vollständiger Induktion über K

Ind. anfangs $K=1$ $c \cdot n \leq 2^1 \cdot c \cdot 1 \cdot n = 2cn$

Annahme Beh. gilt für alle $j < k$

Ind.-schluss: Zeige Beh. für k

$$\begin{aligned} T(n, k) &\leq 2 \cdot T(n-1, k-1) + c \cdot k \cdot n \\ &\leq 2 \cdot 2^{k-1} \cdot c \cdot (k-1) \cdot n + ck^n \\ &\leq 2^k c \cdot n \cdot k - \underbrace{2^k \cdot c \cdot n + k \cdot c \cdot n}_{< 0} \leq 2^k \cdot c \cdot k \cdot n \end{aligned}$$

Lösen NP-vollständiger Probleme auf Bäumen

⌚ Definition

Gegeben ist ein ungerichteter Graph $G = (V, E)$.

Eine **Independent Set** ist eine Teilmenge $I \subseteq V$ der Knotenmenge, sodass **keine zwei Knoten in I** durch eine Kante verbunden sind.

Anders gesagt: Für alle $u, v \in I$ gilt: $E(u, v) \notin E$.

☰ Beispiel

Graph:

A --- B --- C

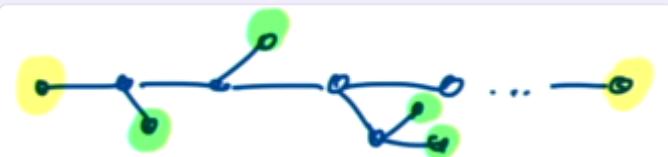
Kanten: $\{(A,B), (B,C)\}$

Mögliche **Independent Sets**:

- $\{A\}$
- $\{C\}$
- $\{A,C\} \rightarrow$ **größtes Independent Set** in diesem Graph
- \emptyset

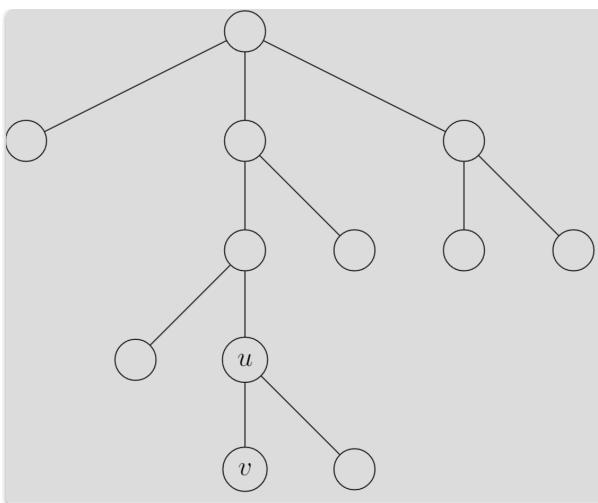
Nicht erlaubt:

- $\{A,B\} \rightarrow A$ und B sind verbunden
- $\{B,C\} \rightarrow$ ebenfalls verbunden



Independent Set auf Bäumen

- **Tatsache:** Ein Baum mit mindestens zwei Knoten besitzt immer mindestens zwei Blätter.
 - Ein **Blatt** ist ein Knoten mit einem Grad von 1 (d.h., er ist nur mit einer einzigen anderen Kante verbunden).
- **Beobachtung:** Wenn v ein Blatt ist, dann existiert ein maximales Independent Set, welches v enthält.



Anmerkung

Nicht jedes max. Independent-Set muss v enthalten, aber es gibt eines, welches den Knoten v enthält.

✓ Beweis

- Austauschargument:

Wir gehen von einem **maximalen Independent Set** S aus (dies ist eine unabhängige Menge, die nicht weiter vergrößert werden kann, ohne die Unabhängigkeit zu verlieren).

- **Fall 1:** Wenn $v \in S$, sind wir fertig, da S bereits v enthält.
 - **Fall 2:** Wenn $v \notin S$ und der einzige Nachbar u von v ebenfalls nicht in S ist ($u \notin S$), dann ist $S \cup \{v\}$ eine unabhängige Menge. Dies würde bedeuten, dass S nicht maximal war, was ein Widerspruch zur Annahme ist.
 - **Fall 3:** Wenn $v \notin S$ und $u \in S$ (der einzige Nachbar von v ist in S), dann ist $S \cup \{v\} - \{u\}$ eine unabhängige Menge.
 - Begründung: Da u der einzige Nachbar von v ist, und u aus S entfernt wird, kann v zu der Menge hinzugefügt werden, ohne eine Kantenverbindung innerhalb der Menge zu schaffen. Alle anderen Knoten in S sind nicht mit v verbunden, da sie nicht u sind.
 - Die Größe der Menge bleibt gleich: $|S \cup \{v\} - \{u\}| = |S|$.
 - Diese neue Menge ist ebenfalls ein maximales Independent Set, das v enthält.

Greedy Algorithmus:

Theorem: Der nachfolgende Greedy-Algorithmus findet ein maximales Independent Set in einem **Wald** $T = (V, E)$ (jede **Zusammenhangskomponente** des Graphen ist ein **Baum**).

```
Independent-Set-In-A-Forest( $T$ ):
 $S \leftarrow \emptyset$ 
while  $T$  hat zumindest eine Kante
    Sei  $e = (u, v)$  eine Kante, sodass  $v$  ein Blatt ist
    Füge  $v$  zu  $S$  hinzu
    Lösche aus  $V$  die Knoten  $u$  und  $v$  (und alle
    zu diesen Knoten inzidenten Kanten)
 $S \leftarrow S \cup V$ 
return  $S$ 
```

Beweis: Korrektheit folgt aus der vorherigen Beobachtung. \square

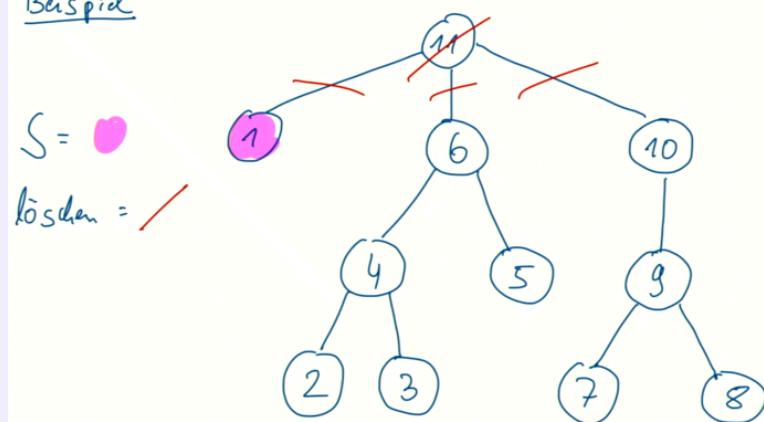
Anmerkung: Kann man in $O(n)$ Zeit implementieren, indem man die Knoten in Postorder durchmustert.

- So ein v welches ein Blatt ist gibt es immer!

:≡ Beispiel

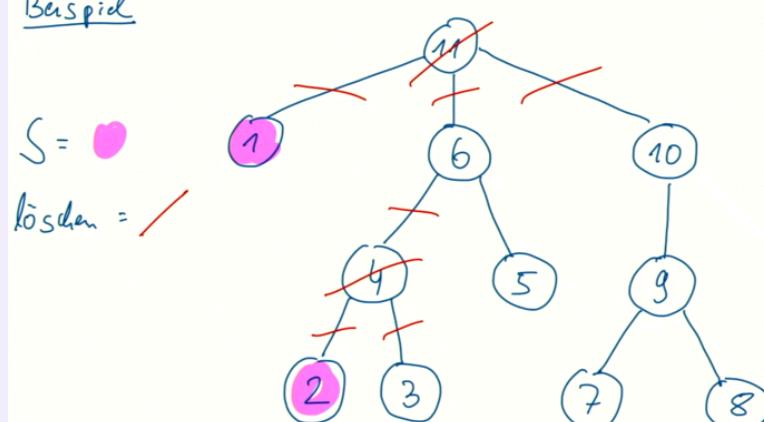
Wir fügen 1 zu unserem Set hinzu, und löschen die jeweiligen dinge raus:

Beispiel

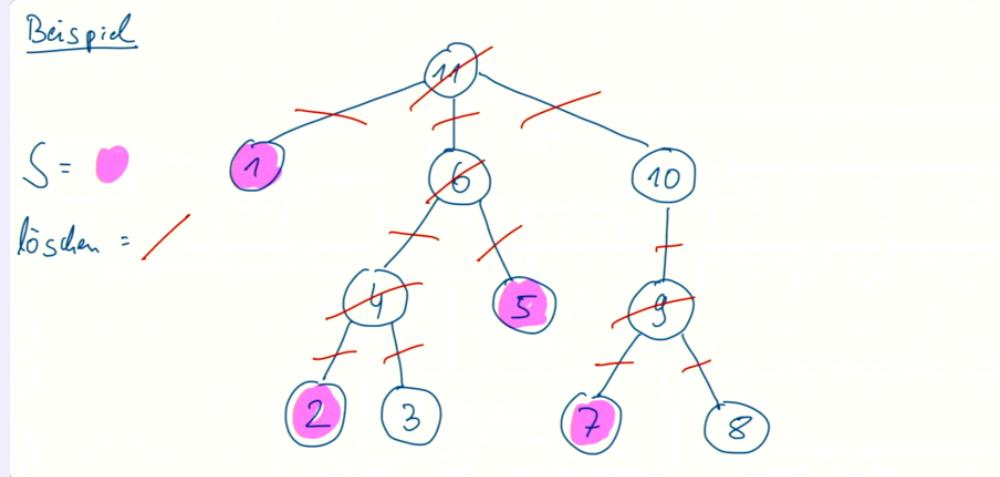


Das selbe für 2:

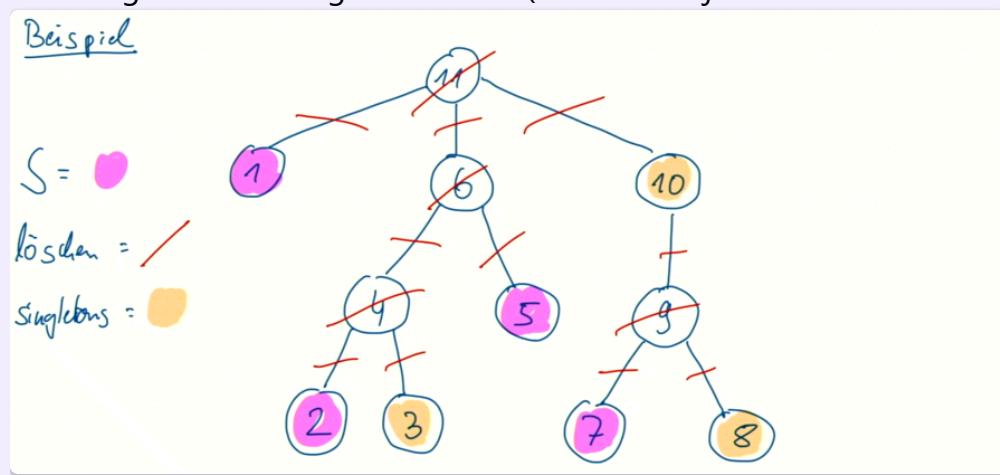
Beispiel



3er ist ein Single Knoten also gehen wir zum 5er und so weiter bis es keine Kanten mehr gibt:



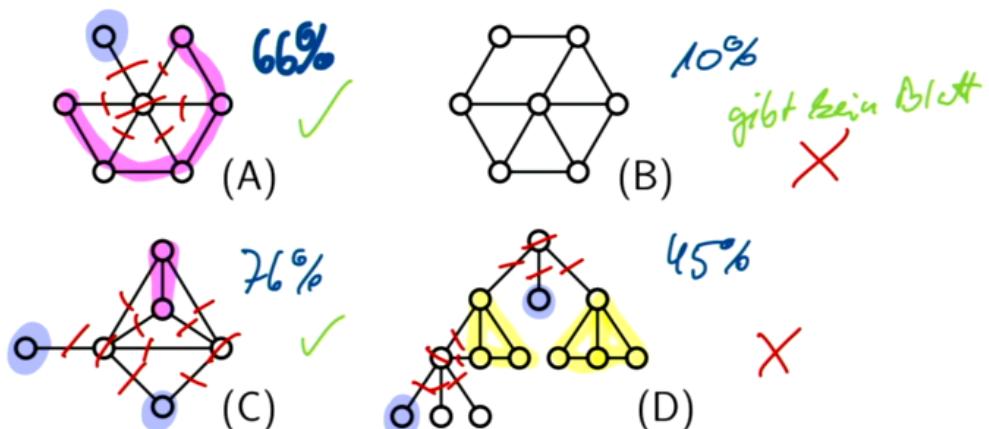
Dann fügt man alle singletons hinzu (also die die jetzt keine Kanten haben):



Am Ende haben wir eine Menge S gefunden mit 7 Knoten. Wir wissen auch, dass der Graph kein größeres Independent Set beinhaltet.

☰ Slido Beispiel

Frage 2: Für welche der folgenden Graphen G liefert der Greedy-Algorithmus Independent-Set-In-A-Forest(G) eine korrekte Lösung, obwohl G kein Wald ist?



Bei D stimmt der Algorithmus für die ersten beiden Schritte, aber dann haben wir kein Blatt mehr welches am restlichen Graphen liegt.

Anmerkung: Graphen die **immer einen Blattknoten** besitzen, auch wenn man diesen iterativ entfernt heißen **1-degeneriert**, und dann funktioniert auch der Algorithmus

Gewichtetes Independent Set auf Bäumen

Bis jetzt hatten all unsere Knoten das Gewicht $w_v = 1$ war also ein ungewichtetes Independent Set, da alle Knoten/Kanten gleich viel Wert waren.

Jetzt:

② Problemstellung

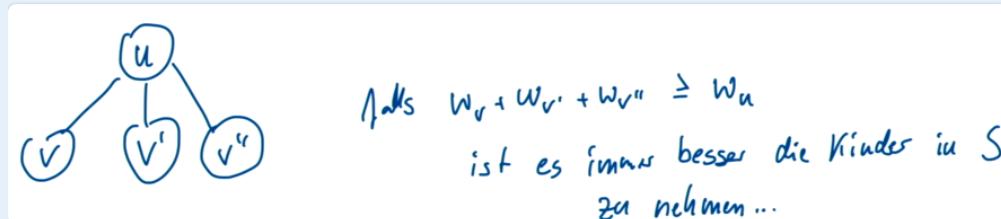
Gegeben sei ein Baum und Knotengewichte $w_v > 0$ für jeden Knoten v . Finde ein Independent Set S , das die Summe der Gewichte aller Knoten in S maximiert (

$$\sum_{v \in S} w_v$$

).

i Beobachtung

Wenn (u, v) eine Kante ist, sodass v ein Blatt ist (d.h., v ist ein Knoten mit Grad 1 und u ist sein einziger Nachbar), dann beinhaltet die optimale Lösung (das maximale gewichtete Independent Set) entweder den Knoten u oder sie enthält alle Blattknoten, die zu u inzident sind (d.h., alle Blätter, die mit u verbunden sind).



Lösung mit Dynamischer Programmierung

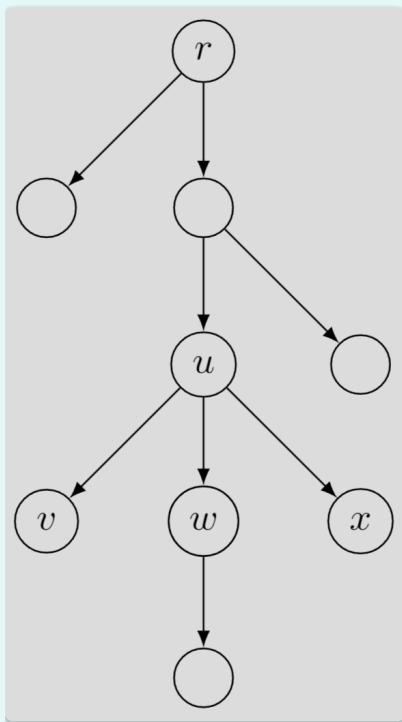
Um das gewichtete Independent Set Problem auf Bäumen zu lösen, kann man Dynamische Programmierung (DP) verwenden.

�� Vorgehensweise

Wähle einen beliebigen Knoten als Wurzel des Baumes aus, z.B. Knoten r . Die Problemstellung wird dann rekursiv von der Wurzel zu den Blättern hin gelöst, oder Bottom-up von den Blättern zur Wurzel.

⌚ Definition

- **Definitionen der DP-Zustände für einen Knoten u :**
 - $OPT_{in}(u)$: Stellt das **maximale Gewicht** eines Independent Sets des Unterbaums dar, der bei u verwurzelt ist, *unter der Bedingung, dass u selbst im Independent Set enthalten ist*.
 - $OPT_{out}(u)$: Stellt das **maximale Gewicht** eines Independent Sets des Unterbaums dar, der bei u verwurzelt ist, *unter der Bedingung, dass u selbst nicht im Independent Set enthalten ist*.



- **Nachfolger von u :** Im Beispielbild sind die Nachfolger (Kinder) von u die Knoten v, w, x .

Formulierung:

$$\begin{aligned} OPT_{in}(u) &= w_u + \sum_{v \in \text{Nachfolger}(u)} OPT_{out}(v) \\ OPT_{out}(u) &= \max_{v \in \text{Nachfolger}(u)} \{OPT_{in}(v), OPT_{out}(v)\} \end{aligned}$$

↑ freie Wahl für die Kinder von u

Falls wir den Knoten U aufnehmen, dürfen wir seine Kinder nicht mehr aufnehmen. Wir nehmen w_u und die Summe aller Nachfolgerknoten von OPT_{out} , dann sind die Kinder nicht in der Lösung

Das 2. da nehme ich nur die Kinder auf und nicht den Parent Knoten.

Implementierung

```

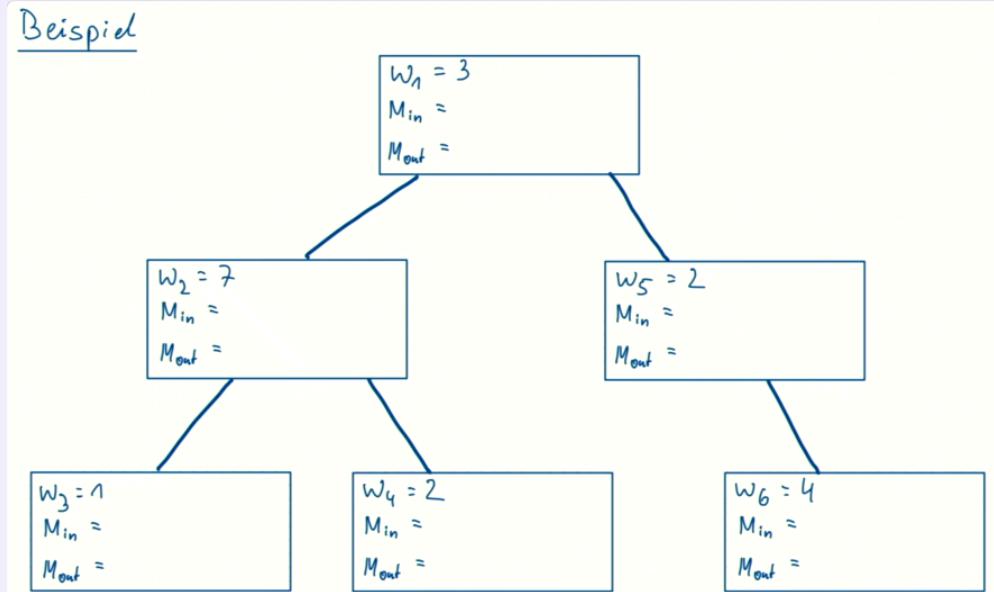
Weighted-Independent-Set-In-A-Tree( $T$ ):
Wähle eine Wurzel  $r$  aus
foreach Knoten  $u$  von  $T$  in Postorder
    if  $u$  ist ein Blatt
         $M_{in}[u] \leftarrow w_u$ 
         $M_{out}[u] \leftarrow 0$ 
    else
         $M_{in}[u] \leftarrow w_u + \sum_{v \in \text{Nachfolger}(u)} M_{out}[v]$ 
         $M_{out}[u] \leftarrow \max\{M_{in}[v], M_{out}[v]\}$ 
return  $\max\{M_{in}[r], M_{out}[r]\}$ 

```

- Stellt sicher, dass ein Knoten nach seinen Nachfolgern besucht wird.

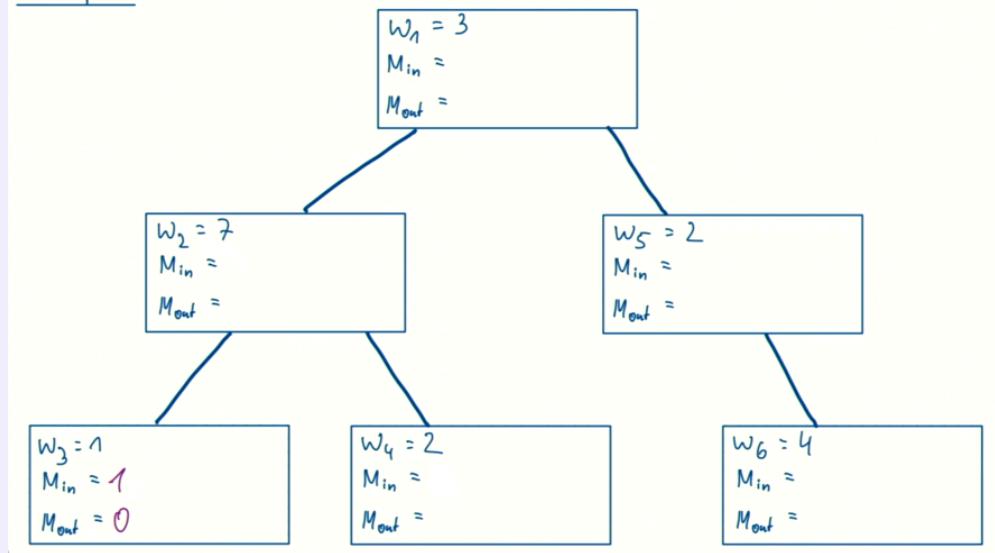
Tabelle M_{in} und M_{out} mit $n = |V|$ Einträgen

Beispiel



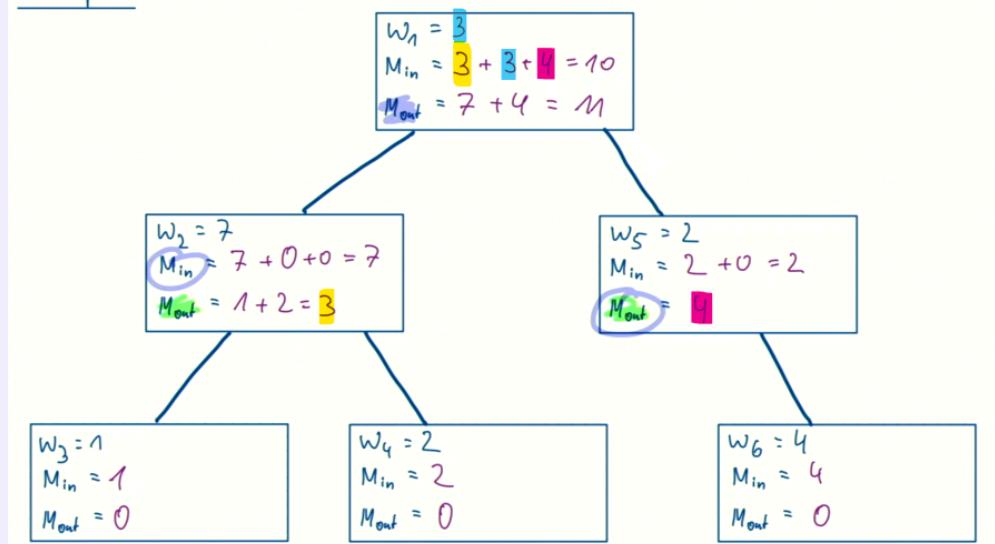
Wir sind in Postorder heißt wir beginnen links unten:

Beispiel



Dann geht's einfach in Postorder weiter und berechnen was sich mehr lohnt:

Beispiel



Am Ende entscheidet man sich dann für das größere Ergebnis also das M_{out} wo 11 rauskommt.

i) Laufzeit vom Algorithmus

- **Aussage:** Der Algorithmus findet ein **maximal gewichtetes Independent Set** in einem Baum.
- **Laufzeit:** $O(n)$ (wobei n die Anzahl der Knoten ist).
 - Findet sowohl den **Wert** als auch das **Independent Set** selbst.
- **Beweis der Laufzeit:**
 - Benötigt $O(n)$ Zeit.
 - Knoten werden in **Postorder** durchmustert.

- Jede Kante wird genau einmal überprüft.

Kontext

Independent Set auf Bäumen: Spezialfall

- Dieser strukturierte Spezialfall (Independent Set auf Bäumen) ist **handhabbar**, weil wir einen Knoten finden können, der die **Verbindung zwischen den Subproblemen in verschiedenen Subbäumen unterbrechen kann**.

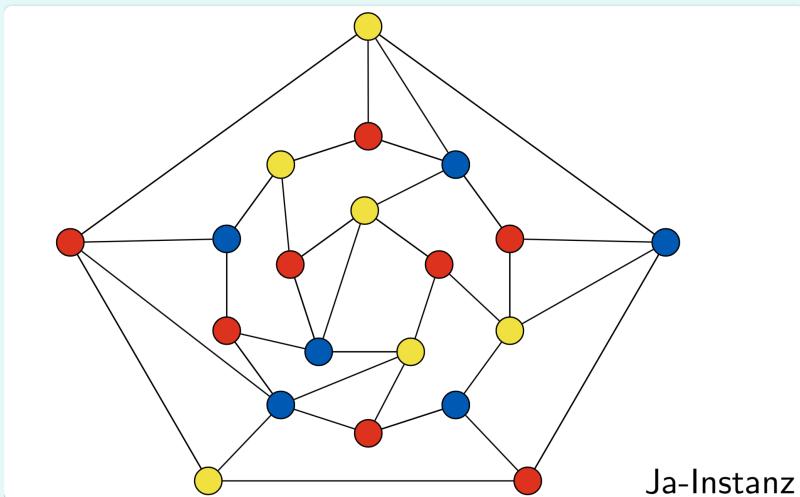
Graphen mit beschränkter Baumweite (elegante Generalisierung von Bäumen)

- **Baumweite** ist ein Maß dafür, wie "baumartig" ein Graph ist. Bäume haben eine Baumweite von 1.
- Graphen mit beschränkter Baumweite:
 - Erfassen eine **reichhaltige Klasse von Graphen**, die in der Praxis häufig auftreten.
 - Erlauben die **Aufteilung in unabhängige Teile** (ähnlich wie bei Bäumen, aber allgemeiner), was die Anwendung von dynamischer Programmierung ermöglicht.

Knotenfärben in Intervallgraphen

⌚ 3-Color - Definition/Wiederholung

Gegeben ist ein ungerichteter Graph G . Kann man die Knoten des Graphen in 3 Farben so einfärben, dass benachbarte Knoten nicht die gleiche Farbe besitzen?



Jetzt betrachten wir diese Version:

⌚ k -Färbungsproblem

- **Ziel:** Bei einem ungerichteten Graphen G , zu entscheiden, ob seine Knoten mit $k \geq 3$ Farben gefärbt werden können.
- **Bedingung:** Keine zwei direkt miteinander verbundenen Knoten dürfen dieselbe Farbe haben.
- **Schwierigkeit:** Das k -Färbungsproblem ist **NP-vollständig**, wenn $k \geq 3$.

Das ist selbst für kleine k schon schwer.

Bei $k = 2$ also bei bipartite Graphen kann man das noch leicht unterscheiden

⌚ Opt-COLOR (Optimierungsproblem der Färbung)

- **Problemstellung:** Gegeben ist ein ungerichteter Graph G . Ziel ist es, die Knoten des Graphen mit der **minimalen Anzahl an Farben** einzufärben, sodass benachbarte Knoten keine gleiche Farbe erhalten.
 - Dies ist die Optimierungsvariante des k -Färbungsproblems, bei der nicht nur gefragt wird, ob eine Färbung mit k Farben existiert, sondern die **kleinstmögliche** k gesucht wird.
- **Ziel der weiteren Betrachtung:** Fokussierung auf **Intervallgraphen**, da diese einen effizient lösbar Spezialfall des Färbungsproblems darstellen.

Also unsere Frage ist, wie viele Farben wir mindestens brauchen um das so färben zu können.

Intervallgraphen

⌚ Definition

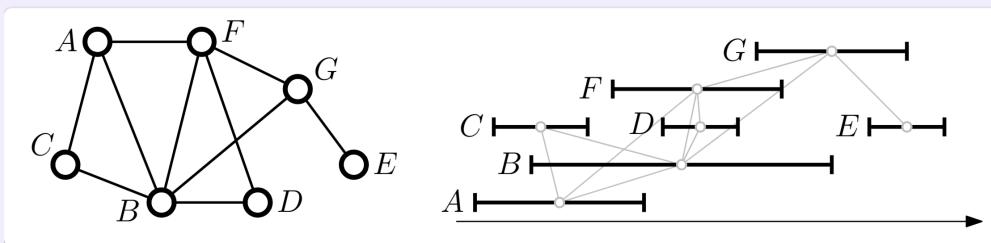
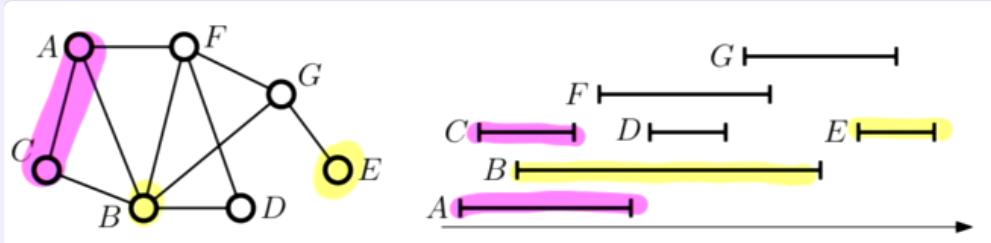
Intervallgraphen sind Graphen, die als **Schnittgraph von Intervallen in \mathbb{R}** dargestellt werden können.

- **Bestandteile eines Intervallgraphen:**

- Ein ungerichteter Graph $G = (V, E)$.
- Eine Intervallmenge $\mathcal{I} = \{I_v = [a_v, b_v] \subset \mathbb{R} \mid v \in V\}$. Jedem Knoten $v \in V$ wird also ein Intervall I_v auf der reellen Zahlengerade zugewiesen.
- Eine Kante $(u, v) \in E$ existiert genau dann, wenn sich die den Knoten u und v zugewiesenen Intervalle I_u und I_v **überschneiden** (d.h., ihr Schnitt ist nicht leer: $I_u \cap I_v \neq \emptyset$).

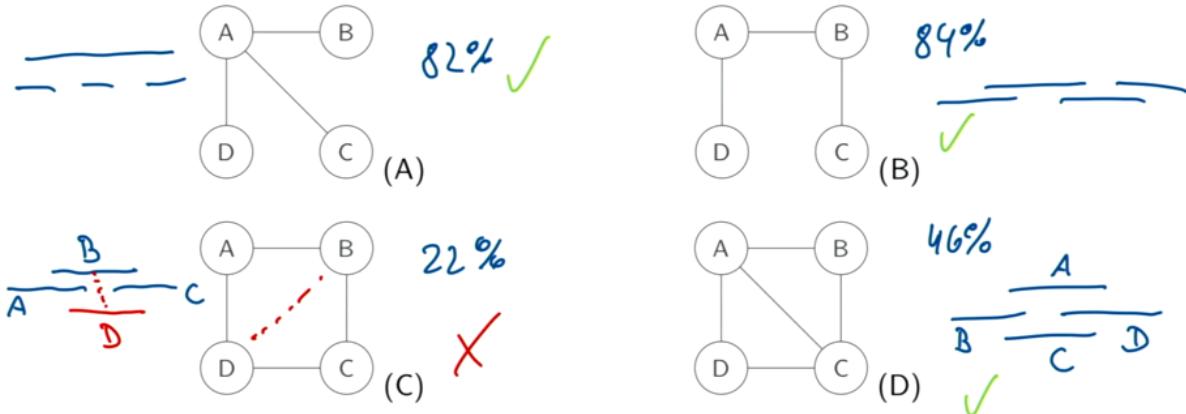
☰ Beispiel

Die Graphen dürfen sich nur überlappen, wenn sie eine Verbindung haben:



☰ Beispiel2 (Slido)

Frage 3: Welche der folgenden Graphen sind Intervallgraphen?

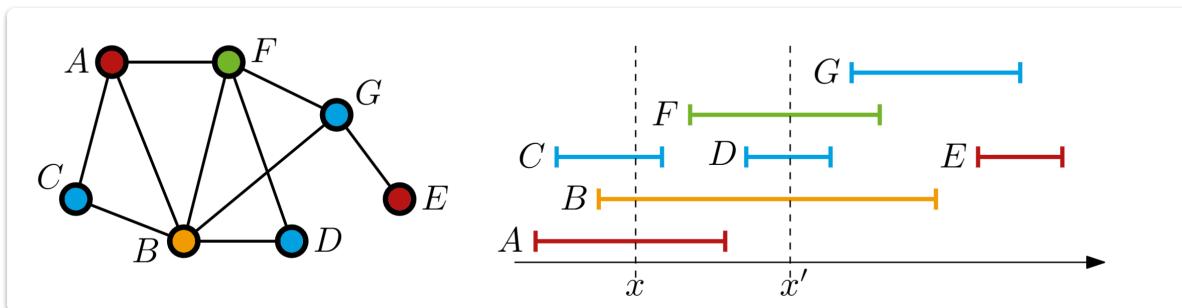


Hinweis, man kann das in Linearzeit auch algorithmisch testen (ob ein Graph ein Intervallgraph ist)

Färben von Intervallgraphen

Färben mit einer Farbe

- Äquivalent zu Maximum Independent Set
- Äquivalent zu Interval Scheduling (Kap. Greedy-Algorithmen)
- Greedy-Algorithmus EDF (earliest deadline first)?



Hier bräuchte man 4 Farben, aber kann man auch weniger brauchen?

① Beobachtung

- Wenn ein Punkt $x \in \mathbb{R}$ in k verschiedenen Intervallen gleichzeitig liegt, dann benötigt man für die Färbung der entsprechenden k Knoten **mindestens k Farben**.
 - Begründung: Alle k Intervalle überschneiden sich an Punkt x , was bedeutet, dass die k zugehörigen Knoten paarweise zueinander benachbart sind (sie bilden eine k -Clique). Eine k -Clique erfordert immer mindestens k verschiedene Farben.

ⓘ Beobachtung

- Definiere die **Tiefe d** als den maximalen Wert von k für alle möglichen Punkte $x \in \mathbb{R}$, d.h., $d := \max_{x \in \mathbb{R}} \{|\{I \in \mathcal{I} \mid x \in I\}|\}$.
 - Dies bedeutet, d ist die maximale Anzahl von Intervallen, die sich an einem einzelnen Punkt auf der reellen Achse überschneiden.
 - Für die Färbung des Graphen G (der durch die Intervalle definiert ist) benötigt man **mindestens d Farben**.
 - Dies folgt direkt aus der ersten Beobachtung: Die d Intervalle, die sich am Punkt der maximalen Überlappung treffen, bilden eine Clique der Größe d
 - .

ⓘ Frage

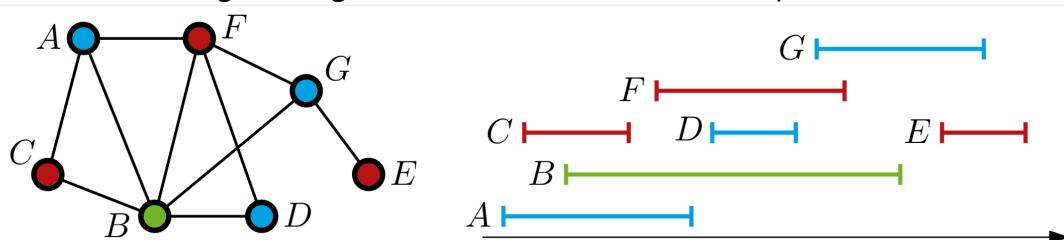
- Geht es auch immer mit genau d Farben? (Kann man einen Intervallgraphen immer mit d Farben färben, wobei d die maximale Tiefe der Intervallüberlappung ist?)

⇒ Ja es geht auch nur mit d Farben

Algorithmus:

```
Interval-Coloring( $\mathcal{I}$ ):
Sortiere Intervallgrenzen aufsteigend in Liste  $\mathcal{L}$ 
 $\text{col}_{\max} \leftarrow 0$ 
Initialisiere leere Queue  $Q$ 
foreach Punkt  $a$  von  $\mathcal{L}$ 
  if  $a$  ist ein Startpunkt
    if  $Q$  ist leer
      färbe  $I = [a, b]$  mit Farbe  $\text{col}_{\max}$ 
       $\text{col}_{\max} \leftarrow \text{col}_{\max} + 1$ 
    else
      entferne  $c = Q.\text{head}$  und färbe  $I = [a, b]$  mit Farbe  $c$ 
  else // $a$  ist ein Endpunkt
    füge Farbe von  $I = [b, a]$  in  $Q$  ein
return Färbung von  $\mathcal{I}$  und Anzahl  $\text{col}_{\max}$ 
```

Nach Anwendung des Algorithmus sieht dann unser Graph so aus:



Hier brauchen wir nicht mehr 4 sondern nur noch 3 Farben, was auch unsere Tiefe d war.

Beweis

✓ Theorem: Algorithmus Interval-Coloring

- **Aussage:** Der Algorithmus "Interval-Coloring" berechnet eine **minimale Färbung** für eine Intervallmenge \mathcal{I} und den Graphen G mit n Knoten.
- **Laufzeit:** $O(n \log n)$.
- **Beweis / Funktionsweise:**
 - Algorithmus verwaltet "freie" Farben (z.B. in einer Queue Q).
 - Wenn keine Farbe frei ist, wird eine neue Farbe gewählt.
 - Am Ende jedes Intervalls wird dessen Farbe freigegeben.
 - **Gültigkeit:** Überlappende Intervalle erhalten verschiedene Farben; mindestens d Farben benötigt.
 - **Anzahl Farben:** Es werden **genau d Farben** verwendet ($0, \dots, d - 1$).
 - Widerspruchsbeweis: Wäre Farbe d notwendig, gäbe es d überlappende Intervalle, was der Definition von d als maximale Tiefe widerspräche.
 - **Laufzeit:** Die eigentliche Färbung ist $O(n)$, aber die **Sortierung der Intervallgrenzen** dominiert mit $O(n \log n)$.

11. Branch and Bound

Einleitung

Kombinatorische Optimierung

⌚ Definition

- Ziel: Konstruktion einer Teilmenge aus einer (großen) Menge diskreter Elemente (Gegenstände, Orte usw.).
- Erfüllung gewisser Nebenbedingungen.
- Optimalität bezüglich einer Kostenfunktion (kleinstes Gewicht, kürzeste Strecken, ...).

Optimierung: Schwere Probleme

Schwere Probleme:

- In dieser Lehrveranstaltung wurden bereits Probleme behandelt, für die es **unwahrscheinlich** ist, dass Lösungsverfahren existieren, die **alle möglichen Instanzen** (siehe hier) eines bestimmten Problems in **polynomieller Zeit** lösen können.

Anwendung:

- In den folgenden Einheiten werden wir uns mit Verfahren beschäftigen, die bei solchen schweren Problemen grundsätzlich angewendet werden können.

Verfahren für Optimierungsprobleme:

- Branch-and-Bound
- Dynamische Programmierung
- Approximations(algorithmen)
- Heuristische Verfahren

Diese haben immer unterschiedliche Trade-offs

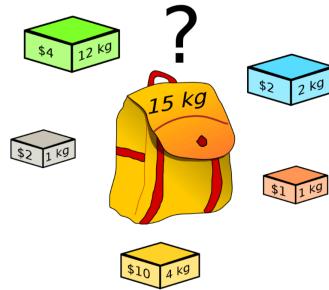
Branch and Bound

Beschränke eine auf Divide-and-Conquer basierende systematische Durchmusterung aller Lösungen mit Hilfe von Methoden, die **untere und obere Schranken** liefern, und ermittle eine optimale Lösung.

Trade-Off hier: **Verzicht auf Garantie der Polynomialzeit**.

Rucksackproblem

Gegeben: n Gegenstände mit positiven rationalen Gewichten g_1, \dots, g_n und Werten w_1, \dots, w_n und eine Kapazität G (auch positiv rational).



Gesucht: Teilmenge S der Gegenstände mit Gesamtgewicht $\leq G$ und maximalem Gesamtwert.

Man will eine Teilmenge der Gegenstände mit so viel Wert wie möglich mitnehmen, ohne die Gewichtsgrenze G zu überschreiten.

Welche Teilmenge ist die beste?

Mathematische Formulierung

Entscheidungsvariablen:

- Einführung von 0/1-Entscheidungsvariablen x_1, \dots, x_n für die Wahl der Gegenstände:

$$x_i = \begin{cases} 0 & \text{falls Gegenstand } i \text{ nicht gewählt wird} \\ 1 & \text{falls Gegenstand } i \text{ gewählt wird} \end{cases}$$

Mathematische Formulierung (für n Gegenstände):

- Maximiere den Gesamtwert der ausgewählten Gegenstände:

$$\sum_{i=1}^n w_i x_i$$

wobei w_i der Wert des Gegenstands i ist.

Das ist unsere Zielfunktion.

- Nebenbedingung: Das Gesamtgewicht der ausgewählten Gegenstände darf die Kapazität G des Rucksacks nicht überschreiten:

$$\sum_{i=1}^n g_i x_i \leq G$$

wobei g_i das Gewicht des Gegenstands i ist.

- Zusätzliche Bedingung: Die Entscheidungsvariablen müssen binär sein:

$$x_i \in \{0, 1\} \quad \text{für } i = 1, \dots, n$$

Enumeration (Backtracking)

Enumeration:

- Eine Enumeration aller zulässigen Lösungen für das Rucksackproblem entspricht der Aufzählung aller Teilmengen der n -elementigen Menge (bis auf diejenigen Teilmengen, die nicht in den Rucksack passen).
- Da gibt's 2^n viele mögliche Lösungen

Lösungsvektor und Zielfunktion:

- Zu jedem aktuellen Lösungsvektor $\vec{x} = (x_1, \dots, x_n)$ gehört ein Zielfunktionswert (Gesamtwert von \vec{x}) w_{curr} und ein Gesamtgewicht g_{curr} .
- Die bisher beste gefundene Lösung wird in dem globalen Vektor \vec{x}_{best} und der zugehörige Lösungswert in der globalen Variablen w_{max} gespeichert.

Prinzip:

- Wir folgen wiederum dem Prinzip des Divide-and-Conquer.

Enumerationsalgorithmus

Eingabe: Anzahl z der fixierten Variablen in \vec{x} ; Gesamtwert w_{curr} ; Gesamtgewicht g_{curr} ; aktueller Lösungsvektor \vec{x} .

```
Enum( $z, w_{curr}, g_{curr}, \vec{x}$ ):
  if  $g_{curr} \leq G$ 
    if  $w_{curr} > w_{max}$ 
       $w_{max} \leftarrow w_{curr}$ 
       $\vec{x}_{best} \leftarrow \vec{x}$ 
    for  $i \leftarrow z + 1$  bis  $n$ 
       $x_i \leftarrow 1$ 
      Enum( $i, w_{curr} + w_i, g_{curr} + g_i, \vec{x}$ )
       $x_i \leftarrow 0$ 
```

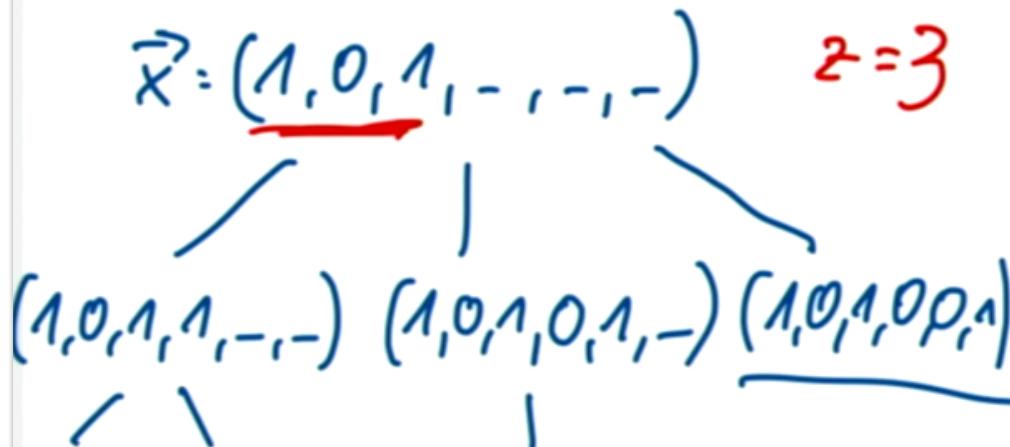
Hinweis:

- w_{max} und \vec{x}_{best} sind globale Variablen.
- Initialisierung: $w_{max} = 0$ und $\vec{x}_{best} = \vec{0}$

1. Schauen ob wir noch Platz haben
2. Schauen ob unsere jetzige Lösung besser ist als die die wir bis jetzt hatten
 1. wenn ja update
3. Rekursiver Aufruf für jeden nächsten Gegenstand den es gibt.

Aufrufbaum:

Aufrufbaum



① Ablauf des Enumerationsalgorithmus (Backtracking)

Start:

- Der Algorithmus wird mit dem Aufruf `Enum(0, 0, 0, \vec{0})` gestartet.

Rekursiver Aufruf:

- In jedem rekursiven Aufruf wird die aktuelle Lösung \vec{x} bewertet.
- Danach werden die Variablen x_1 bis x_z als fixiert betrachtet.
- Der dadurch beschriebene Teil des gesamten Suchraums wird weiter unterteilt.
- Wir betrachten alle möglichen Fälle, welche Variable x_i (mit $i = z + 1$ bis $i = n$) als nächstes auf 1 gesetzt werden kann.
- Die Variablen x_{z+1} bis x_{i-1} werden gleichzeitig auf 0 fixiert.
- Alle so erzeugten kleineren Unterprobleme werden durch rekursive Aufrufe gelöst.

① Komplexität

- Es gibt bis zu 2^n rekursive Aufrufe.
- Der Aufwand pro Aufruf (exklusive Rekursion) ist konstant.
- Daher liegt die Laufzeit in $O(2^n)$.

Branch and Bound

Rucksackproblem: Verbesserung der Enumeration

⌚ Idee zur Verbesserung

- Überprüfen von Zwischenlösungen mit $z < n$ fixierten Variablenwerten.

- Man überprüft, ob es noch möglich sein kann, aus dieser Lösung durch Hinzufügen weiterer Gegenstände eine zu erzeugen, die besser ist als die bisher beste gefundene.
- Wenn es **offensichtlich ist, dass keine neue beste Lösung abgeleitet werden kann**, dann sind weitere **rekursive Aufrufe nicht sinnvoll**.
- Das **frühzeitige Abbrechen** führt zu einer Beschneidung des rekursiven Aufrufbaums.
- Das kann eine **erhebliche Beschleunigung** bewirken.

Wie funktioniert das jetzt bei unserem Rucksackproblem

Ansatz

- Berechne obere Schranke U' , und führe den Aufruf nur durch, wenn der Wert $U' > w_{\max}$.
- Sortiere die Gegenstände nach nicht-steigenden Werten $\frac{w_i}{g_i}$.

Wenn w_{\max} größer oder gleich U' ist, dann wissen wir, dass wir nichts besseres mehr finden können und können abbrechen.

Wir wollen die Gegenstände mitnehmen, die vom Wert-Gewichtsquotienten sich am meisten lohnen.

```
Enum( $z, w_{\text{curr}}, g_{\text{curr}}, \vec{x}$ ):
  if  $g_{\text{curr}} \leq G$ 
    if  $w_{\text{curr}} > w_{\max}$ 
       $w_{\max} \leftarrow w_{\text{curr}}$ 
       $\vec{x}_{\text{best}} \leftarrow \vec{x}$ 
    for  $i \leftarrow z + 1$  bis  $n$ 
       $U' \leftarrow w_{\text{curr}} + (G - g_{\text{curr}}) \cdot \frac{w_i}{g_i}$ 
      if  $U' > w_{\max}$ 
         $x_i \leftarrow 1$ 
        Enum( $i, w_{\text{curr}} + w_i, g_{\text{curr}} + g_i, \vec{x}$ )
         $x_i \leftarrow 0$ 
```

(Das Rote ist neu!)

Obere Schranke (U'):

- Berechnung der oberen Schranke für den Wert der optimalen Lösung in einem Teilproblem:

$$U' \leftarrow w_{\text{curr}} + (G - g_{\text{curr}}) \cdot \frac{w_i}{g_i}$$

- w_{curr} : Wert der bisherigen Zuteilung.

- $G - g_{curr}$: Verbleibende Kapazität im Rucksack.
- $\frac{w_i}{g_i}$: Wert pro Gewichtseinheit des aktuell untersuchten Gegenstands i .

Erläuterung:

- Die verbleibende Kapazität wird mit dem aktuell untersuchten Gegenstand i komplett (möglicherweise auch mehrmals) aufgefüllt.
- Hierbei kann es auch zu **teilweisen Zuteilungen** kommen (z.B. Gegenstand i wird 1.7 mal eingepackt).
- Da die Gegenstände nach **nicht-steigenden Werten** $\frac{w_i}{g_i}$ **sortiert** sind, haben alle Gegenstände $i + 1, i + 2, \dots, n$ einen relativen Wert kleiner oder gleich dem von i .
- Damit ist die obere Schranke U' **garantiert größer oder gleich** dem Wert der optimalen Lösung für dieses Teilproblem.

Prinzip von Branch-and-Bound: Maximierungsproblem

⌚ Branching

- Wie bei der Enumeration üblich wird das Problem rekursiv in kleinere Teilprobleme partitioniert → *Divide-and-Conquer-Prinzip*.

⌚ Bounding

- Für jedes Teilproblem wird berechnet:
 - Eine **lokale obere Schranke** U' (upper bound) (liefert uns einen **best case**).
 - Eine **lokale untere Schranke** L' (lower bound) (liefert uns einen **worst case**).
 - Zwischen U' und L' ist eine kleine Lücke gut für den Algorithmus

⌚ Abbruch

- Teilprobleme mit $U' \leq L$ (wobei L einer **globalen unteren Schranke** entspricht) brauchen nicht weiter verfolgt zu werden!

⌚ Schranken

- Der Wert jeder gültigen Lösung ist eine **untere Schranke**.
- Obere Schranken werden i. A. separat mit einer sogenannten **Dualheuristik** ermittelt.

Rucksackproblem: Verbesserte Schranken

① Verbesserte untere Schranke

Sortierung:

- Die Gegenstände werden nicht-steigend nach ihrem relativen Wert $\frac{w_i}{g_i}$ sortiert.

Untere Schranke:

- Man durchläuft alle Gegenstände, deren Variablen noch nicht festgelegt sind, in der sortierten Reihenfolge und packt den jeweils aktuellen Gegenstand ein, falls noch Platz im Rucksack ist (Greedy-Algorithmus).

② Verbesserte obere Schranke

Einfache obere Schranke (wurde weiter vorne beschrieben).

Mögliche Verbesserung:

- Alle Gegenstände, deren Variablen noch nicht festgelegt sind, werden in der sortierten Reihenfolge durchlaufen.
- Man packt alle Gegenstände ein, bis man zu dem ersten Gegenstand kommt, der nicht mehr in den Rucksack passt.
- Sei r die noch freie Kapazität des Rucksacks. Dann zählt man $r \cdot \frac{w_i}{g_i}$ noch zu dem Wert der Gegenstände im Rucksack dazu.
- Der letzte Gegenstand wird daher nur **teilweise eingepackt**.
- Alle verbleibenden Gegenstände werden ignoriert.

Lösung:

- Diese Vorgehensweise liefert in der Regel eine **obere Schranke**, die keine gültige Lösung des Rucksackproblems entspricht.
 - Das macht aber nichts, da wir trz dann eine echte obere Schranke haben
- Falls diese Vorgehensweise zu einer gültigen Lösung führt, dann ist die Lösung (für das betrachtete Teilproblem) **optimal**.

☰ Beispiel - Rucksackproblem

Gegeben: 4 Gegenstände, Rucksackkapazität = 100

Gegenstand	1	2	3	4
Gewicht g_i	32	16	21	50
Wert w_i	80	20	63	100
Verhältnis w_i/g_i	2.5	1.25	3	2

Sortierung:

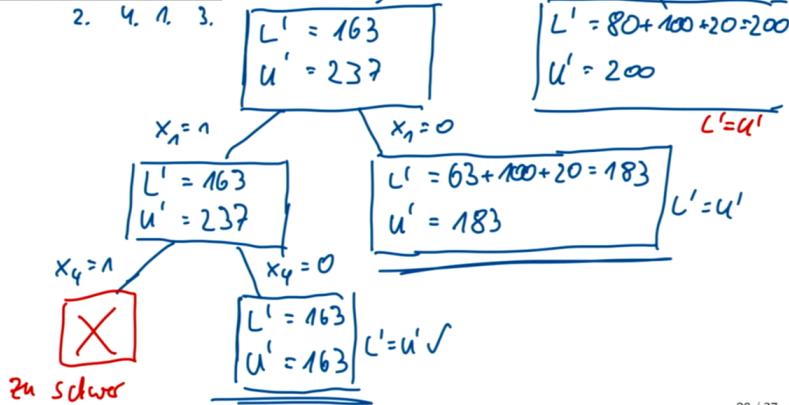
- Für jeden Gegenstand i das Verhältnis $\frac{w_i}{g_i}$ berechnen.
- Sortierte Reihenfolge der Gegenstände: 3 (3), 1 (2.5), 4 (2), 2 (1.25)

Rucksackproblem: Beispiel

Branch-and-Bound-Baum:

4 Gegenstände, Rucksackkapazität = 100

Gegenstand	1	2	3	4
Gewicht g_i	32	16	21	50
Wert w_i	80	20	63	100
Verhältnis w_i/g_i	2.5	1.25	3	2



Erläuterung:

- 1 (Start):** Noch keine Variablen fixiert. Der gesamte Suchraum wird betrachtet.
- 2:** Es wird angenommen, dass Gegenstand 3 (x_3) fixiert ist (z.B. auf 1 gesetzt), und nur die anderen Gegenstände können ausgewählt werden. Dies führt zu einem Unterbaum des Suchraums.
- Eine Fixierung von Gegenstand 3, 1 und 4 würde zu einer unmöglichen Lösung führen, da sie eine Kapazität von 103 benötigen würde (Annahme: Rucksackkapazität ist geringer). Dieser Pfad im Suchbaum würde frühzeitig verworfen (implizites Bounding).
- 4 und 5:** In diesen Knoten gilt $L = U$ (lokale untere Schranke gleich lokaler oberer Schranke). Dies bedeutet, dass die bestmögliche Lösung in diesem Unterbaum bereits gefunden wurde. Daher brauchen in diesem Unterbaum keine weiteren Gegenstände hinzugefügt werden → **Beschneidung des rekursiven Aufrufbaums**.
- 6:** Hier gilt ebenfalls $L = U$. Der Unterbaum braucht nicht weiter untersucht zu werden. Der Wert der Lösung in diesem Unterbaum (L bzw. U) ist am größten im Vergleich zu anderen Endknoten mit $L = U$.

- **Ergebnis:** Da in Knoten 6 die beste Lösung mit $L = U$ gefunden wurde, und in diesem Fall $x_3 = 0$ war (Gegenstand 3 wurde nicht eingepackt), werden die Gegenstände 1, 2 und 4 eingepackt.

Maximierungsproblem: Vorgehen

Allgemeines Vorgehen

- durch das Fixieren von Variablen oder Hinzufügen von Randbedingungen in Unterprobleme zerteilt --> Lösungsraum wird partitioniert.
- Ist für eine (oder mehrere) dieser Teilmengen die für sie berechnete **obere Schranke U' nicht größer** als die **beste überhaupt bisher gefundene untere Schranke L** (= Wert der bisher besten Lösung), braucht man die Lösungen in dieser Teilmenge nicht mehr beachten (**Pruning**).
- Ist die obere Schranke U' **größer** als die beste gegenwärtige untere Schranke L , muss man die Teilmenge weiter zerkleinern (**Branching**).
- Man fährt solange mit der Zerteilung fort, bis für alle Lösungsteilmengen die obere Schranke U' nicht mehr größer ist als die (globale) beste untere Schranke L .

Allgemeiner Algorithmus

Eingabe: Instanz I

```

Branch-and-Bound-Max( $I$ ):
 $L \leftarrow -\infty$  oder Wert einer initialen heuristischen Lösung
 $U \leftarrow \infty$  oder obere Schranke für  $I$  aus Dualheuristik
 $\Pi \leftarrow \{(I, L, U)\}$ 
while  $\exists(I', L', U') \in \Pi$ 
    Entferne  $(I', L', U')$  aus  $\Pi$ 
    if  $U' > L$ 
        Partitioniere  $I'$  in Teilinstanzen  $I_1, \dots, I_k$ 
        Berechne zugehörige gültige heuristische Lösungen → untere Schranken  $L_1, \dots, L_k$ 
        Berechne zugehörige lokale obere Schranken  $U_1, \dots, U_k$  mit Dualheuristik
         $\Pi \leftarrow \Pi \cup \{(I_1, L_1, U_1), \dots, (I_k, L_k, U_k)\}$ 
        if  $\max\{L_1, \dots, L_k\} > L$ 
             $L \leftarrow \max\{L_1, \dots, L_k\}$ 
    return beste gefundene Lösung mit Wert  $L$ 

```

- █ *Bounding – Fall $U' \leq L$ nicht weiter interessant.*
- █ *Branching.*

Maximierungsproblem: Allgemeines Verfahren

Allgemeines Verfahren

- Branch-and-Bound = allgemeines **Metaverfahren**
- Anwendbar auf viele diskrete Optimierungsprobleme

Effizienzfaktoren:

- Wahl der Heuristiken für obere Schranke U' und untere Schranke L'
- Branching-Strategie (wie erfolgt die Zerteilung in Teilprobleme)

- Auswahlregel für nächste Teilinstanz (z.B. nach bestem U')

Branch-and-Bound: Auswahl des nächsten Teilproblems

Auswahl des nächsten Teilproblems:

- Auswahl aus offenen Problemen hat **keinen Einfluss auf Korrektheit** von Branch-and-Bound
- Aber: **großer Einfluss auf praktische Laufzeit**

Beispiele für Strategien:

- **Best-first:** Auswahl des Teilproblems mit der besten (z.B. höchsten bei Maximierung) oberen Schranke.
- **Depth-first:** Auswahl des zuletzt erzeugten Teilproblems.

⌚ Best-first

- Wählt Teilproblem mit **bester dualer Schranke** (größte obere Schranke bei Maximierung)
- Ziel: **minimale Anzahl** bearbeiteter Teilprobleme bis zur Optimalität

⌚ Depth-first

- Bearbeitet **zuletzt erzeugtes** Teilproblem (analog zu Tiefensuche)
- Liefert oft **schnell gültige Näherungslösung** → dient als initiale untere Schranke
- Kombination möglich: zuerst Depth-first (für L'), dann Best-first (für Effizienz)

Branch and Bound für Minimales Vertex Cover

Allgemeiner Algorithmus

Eingabe: Instanz I

```

Branch-and-Bound-Min( $I$ ):
 $U \leftarrow \infty$  oder Wert einer initialen heuristischen Lösung
 $L \leftarrow -\infty$  oder untere Schranke für  $I$  aus Dualheuristik
 $\Pi \leftarrow \{(I, L, U)\}$ 
while  $\exists (I', L', U') \in \Pi$ 
    Entferne  $(I', L', U')$  aus  $\Pi$ 
    if  $L' < U$ 
        Partitioniere  $I'$  in Teilinstanzen  $I_1, \dots, I_k$ 
        Berechne zugehörige gültige heuristische Lösungen  $\rightarrow$  obere Schranken  $U_1, \dots, U_k$ 
        Berechne zugehörige lokale untere Schranken  $L_1, \dots, L_k$  mit Dualheuristik
         $\Pi \leftarrow \Pi \cup \{(I_1, L_1, U_1), \dots, (I_k, L_k, U_k)\}$ 
        if  $\min\{U_1, \dots, U_k\} < U$ 
             $U \leftarrow \min\{U_1, \dots, U_k\}$ 
    return beste gefundene Lösung mit Wert  $U$ 

```

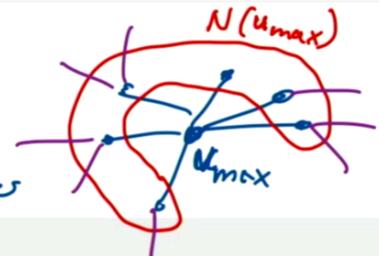
- Bounding - Fall $L' \geq U$ nicht weiter interessant.
- Branching.

Minimales Vertex Cover

Branch-and-Bound: Minimales Vertex Cover

Eingabe: Graph $G = (V, E)$ und Knotenmenge $C = \emptyset$

Knoten fürs Vertex Cover



```

MinVertexCover-BranchAndBound( $G, C$ ):
 $U \leftarrow$  gültige heuristische Lösung für  $(G, C)$  mit Greedyheuristik
 $L \leftarrow$  untere Schranke für  $(G, C)$  mit Matchingheuristik
 $\Pi \leftarrow \{((G, C), L, U)\}$ 
while  $\exists I' \in \Pi$ 
    Entferne  $I' = ((G', C'), L', U')$  aus  $\Pi$ 
    if  $L' < U$ 
         $u_{\max} \leftarrow$  Knoten mit maximalem Grad in  $G'$ 
        Erzeuge Teilinstanzen  $I_1 = (G' - \{u_{\max}\}, C \cup \{u_{\max}\})$  und
         $I_2 = (G' - \{u_{\max}\} - N(u_{\max}), C \cup N(u_{\max}))$  → wähle alle Nachbarn von  $u_{\max}$ 
        Berechne für  $I_1, I_2$  gültige heuristische Lösungen mit Greedyheuristik  $\rightarrow$  obere Schranken  $U_1, U_2$ 
        Berechne für  $I_1, I_2$  lokale untere Schranken  $L_1, L_2$  mit Matchingheuristik
         $\Pi \leftarrow \Pi \cup \{(I_1, L_1, U_1), (I_2, L_2, U_2)\}$ 
        if  $\min\{U_1, U_2\} < U$ 
             $U \leftarrow \min\{U_1, U_2\}$ 
    return beste gefundene Lösung mit Wert  $U$ 

```

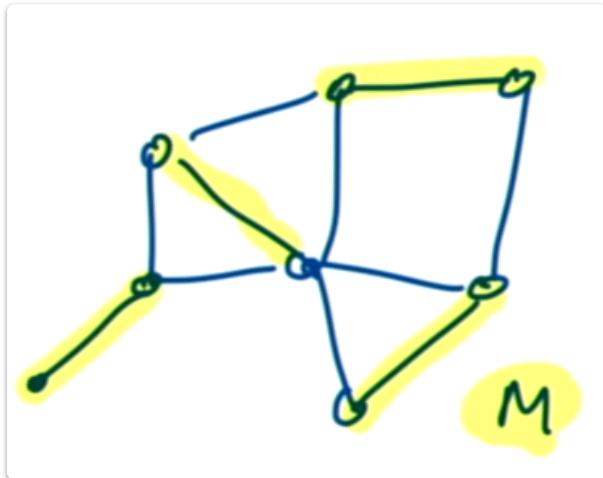
- alle Nachbarknoten von u_{\max}

Untere Schranke: Wird mit Hilfe eines Matchings bestimmt.

- Sei ein Graph $G = (V, E)$ gegeben.
- Eine Menge $M \subseteq E$ heißt Matching, wenn keine zwei Kanten aus M einen Knoten gemeinsam haben.

Nicht erweiterbares Matching (maximales Matching):

- Ein Matching M ist **nicht erweiterbar** (maximal), wenn es keine Kante $e \in E \setminus M$ gibt, sodass $M \cup \{e\}$ ein gültiges Matching ist.
- Ein nicht erweiterbares Matching ist **nicht notwendigerweise** ein größtes Matching (Maximum Matching).
- Ein nicht erweiterbares Matching kann mit einem Greedy-Verfahren gefunden werden.

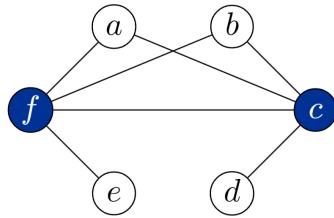


Berechnung der unteren Schranke L' für die Instanz (G', C') :

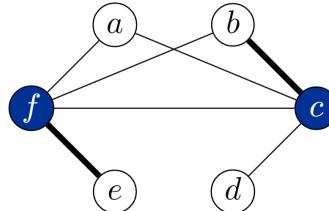
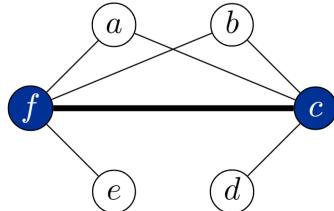
- Man wählt für L' die Größe eines nicht erweiterbaren Matchings.
 - Dabei wird zunächst eine beliebige Kante $e = (u, v)$ gewählt und dann die Knoten u und v und ihre inzidenten Kanten aus G' entfernt.
 - Man fährt mit dieser Prozedur fort, bis keine Kante mehr vorhanden ist.
 - Die Anzahl der gewählten Kanten entspricht der Größe des Matchings.
- Kanten in einem Matching haben keine Knoten gemeinsam.
- Ein Vertex Cover muss zumindest einen Knoten für jede Kante in einem Matching wählen.
- Daher ist die Größe eines Matchings von G' eine untere Schranke für die Größe eines Vertex Covers der Instanz (G', C') .

☰ Beispiel für untere Schranke

Vertex Cover: Minimales Vertex Cover mit $k = 2$



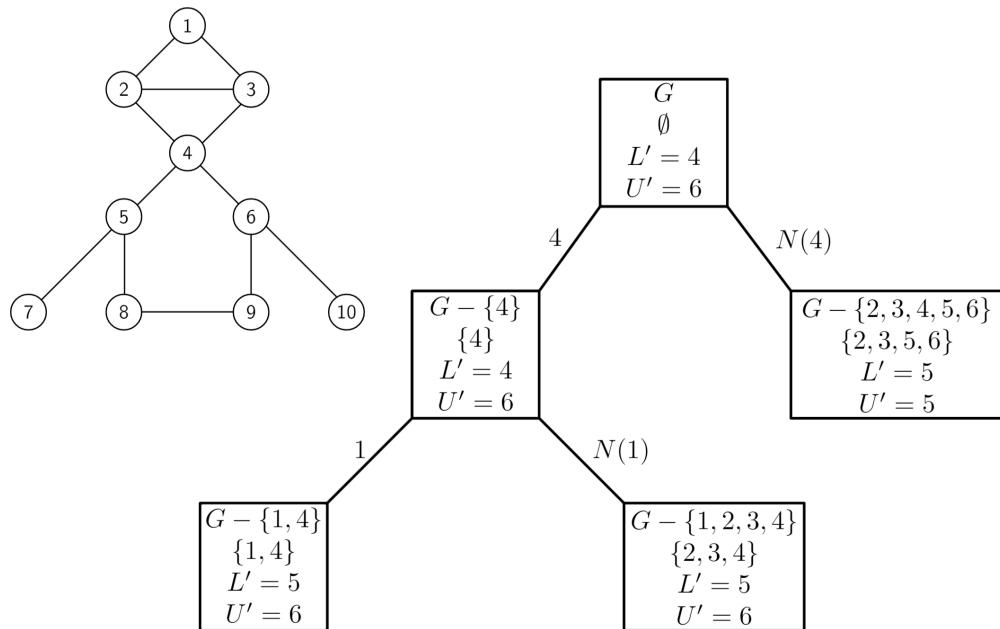
Greedy-Matching: 2 Beispiele für Greedy-Matching (fett eingezeichnete Kanten).



Obere Schranke U' : Wird mit Hilfe eines Greedy-Algorithmus bestimmt.

- Sei ein Graph $G' = (V', E')$ und eine Knotenmenge C' gegeben (die bereits im Vertex Cover enthaltene Knoten der aktuellen Teilstanz).
- Initialisiere eine Menge $S \leftarrow \emptyset$.
- Sortiere die Knoten in V' nicht-steigend nach dem Knotengrad (Anzahl der inzidenten Kanten).
- Durchlaufe V' in dieser Reihenfolge, solange der Graph G' noch Kanten enthält.
 - Füge den Knoten u mit dem höchsten Knotengrad zu S hinzu.
 - Entferne u und alle seine inzidenten Kanten aus G' .
 - Passe die Reihenfolge der verbleibenden Knoten in V' gegebenenfalls an (da sich Knotengrade ändern können).
- Die Menge S ist ein Vertex Cover für den verbleibenden Graphen G' .
- Daher ist $|C'| + |S|$ eine obere Schranke für die Größe eines minimalen Vertex Covers der Teilstanz (G', C') (und damit auch des Eingabegruppen G).

☰ Minimales Vertex Cover Beispiel



Branch-and-Bound: Zusammenfassung

- Branch-and-Bound ist eine allgemein für kombinatorische Optimierungsprobleme einsetzbare Technik zur Berechnung exakter (optimaler) Lösungen.
- Sie funktioniert sowohl für Maximierungs- als auch für Minimierungsprobleme.
- Praktisch lassen sich oft hohe Beschleunigungen erreichen, die worst-case Laufzeit bleibt jedoch wie bei der Enumeration aller möglichen Lösungen (oft exponentiell).

Vorgehen beim Entwurf von Branch-and-Bound Algorithmen

- Wie lassen sich (Teil-)Instanzen des Problems ausdrücken?
- Was sind gute Heuristiken für untere und obere Schranken?
- Wie wird eine (Teil-)Instanz in weitere Teilinstanzen partitioniert (Branching)?
- Welche Teilinstanz wird im nächsten Schritt ausgewählt?

12. Dynamische Programmierung

Wir haben in [11. Branch and Bound](#) schon gelernt wie man mit bestimmten Optimierungsproblemen umgehen kann, jetzt machen wir weiter bei der Optimierung von Dynamischer Programmierung, mit einer art Divde-and-Conquer Optimierung.

Einleitung

Grundlagen Dynamische Programmierung

- **Dynamische Programmierung:**
 - Zerlege ein Problem in eine Folge von **überlappenden Teilproblemen**.
 - Erstelle und speichere Lösungen für diese Teilprobleme.
 - Verwende die gespeicherten Lösungen, um Lösungen für immer größere Teilprobleme zu konstruieren.
- **Optimalitätsprinzip von Bellman:**
 - Dynamische Programmierung führt zu einem optimalen Ergebnis, **genau dann**, wenn sich die optimale Lösung des Gesamtproblems aus den optimalen Lösungen seiner Subprobleme zusammensetzt.
- **Effizienz:**
 - Hängt stark von der Vorgehensweise bei der Aufteilung des Problems und der Ermittlung der Lösungen für die einzelnen Teilprobleme ab.
- **Wesentlicher Aspekt:**
 - **Speicherung (Memoization)** von Ergebnissen für bereits gelöste Subprobleme zur Wiederverwendung. Dies vermeidet redundante Berechnungen und trägt zur Effizienz bei.

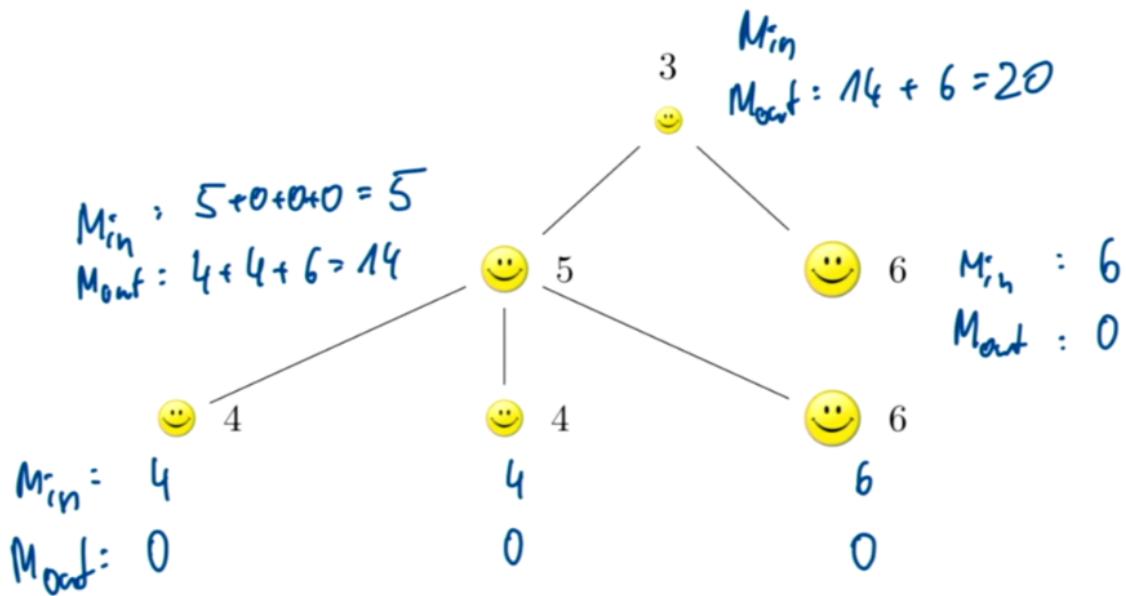
--> Es ist ähnlich zu [Divide and Conquer](#) Techniken

--> Oft exponentiell große Suchräume, aber wir müssen nicht immer alles betrachten

Weighted Independent Set auf Bäumen

:≡ Beispiel

[siehe hier](#)



- Für jeden Knoten sind zwei Werte notiert:
 - M_{in} : Das maximale Gewicht eines Independent Sets im Unterbaum, **inklusive** des aktuellen Knotens. Wenn der aktuelle Knoten im Independent Set ist, dürfen seine direkten Kinder nicht im Set sein.
 - M_{out} : Das maximale Gewicht eines Independent Sets im Unterbaum, **ohne** den aktuellen Knoten. In diesem Fall können die direkten Kinder des aktuellen Knotens entweder im Independent Set sein oder nicht, je nachdem, was das maximale Gewicht ergibt.
- **Blattknoten (Gewicht w):**
 - $M_{in} = w$ (Der Knoten selbst bildet ein Independent Set)
 - $M_{out} = 0$ (Wenn der Knoten nicht im Set ist, trägt er kein Gewicht bei)
- **Innere Knoten (am Beispiel des Knotens mit Gewicht 5):**
 - Um M_{in} für den Knoten mit Gewicht 5 zu berechnen:
 - Wir nehmen das Gewicht des Knotens selbst (5).
 - Da der Knoten im Independent Set ist, dürfen seine direkten Kinder (mit Gewichten 4, 4, 6) nicht im Set sein.
 - Daher addieren wir die M_{out} -Werte der Kinder: $0 + 0 + 0 = 0$.
 - $M_{in} = 5 + 0 + 0 + 0 = 5$.
 - Um M_{out} für den Knoten mit Gewicht 5 zu berechnen:
 - Der Knoten selbst ist nicht im Independent Set.
 - Für jedes Kind wählen wir den maximalen Wert zwischen M_{in} und M_{out} des Kindes, da das Kind entweder im Independent Set sein kann oder nicht.
 - Kinder mit Gewicht 4: $\max(M_{in} = 4, M_{out} = 0) = 4$
 - Kind mit Gewicht 4: $\max(M_{in} = 4, M_{out} = 0) = 4$
 - Kind mit Gewicht 6: $\max(M_{in} = 6, M_{out} = 0) = 6$

- $M_{out} = 4 + 4 + 6 = 14.$
- **Wurzelknoten (Gewicht 3):**
 - $M_{in} = 3 + M_{out}(\text{linkes Kind}) + M_{out}(\text{rechtes Kind}) = 3 + 14 + 0 = 17$
(Anmerkung: Im Screenshot steht hier fälschlicherweise $M_{in} = 6$, dies sollte $3 + 14 = 17$ sein, da das rechte Kind mit Gewicht 6 ein M_{out} von 0 hat, wenn der Wurzelknoten im Set ist)
 - $M_{out} = \max(M_{in}(\text{linkes Kind}), M_{out}(\text{linkes Kind})) + \max(M_{in}(\text{rechtes Kind}), M_{out}(\text{rechtes Kind}))$
 - $M_{out} = \max(5, 14) + \max(6, 0) = 14 + 6 = 20.$
- Das maximale Gewicht des Weighted Independent Set für den gesamten Baum ist $\max(M_{in}(\text{Wurzel}), M_{out}(\text{Wurzel})) = \max(17, 20) = 20.$

Einführendes Beispiel

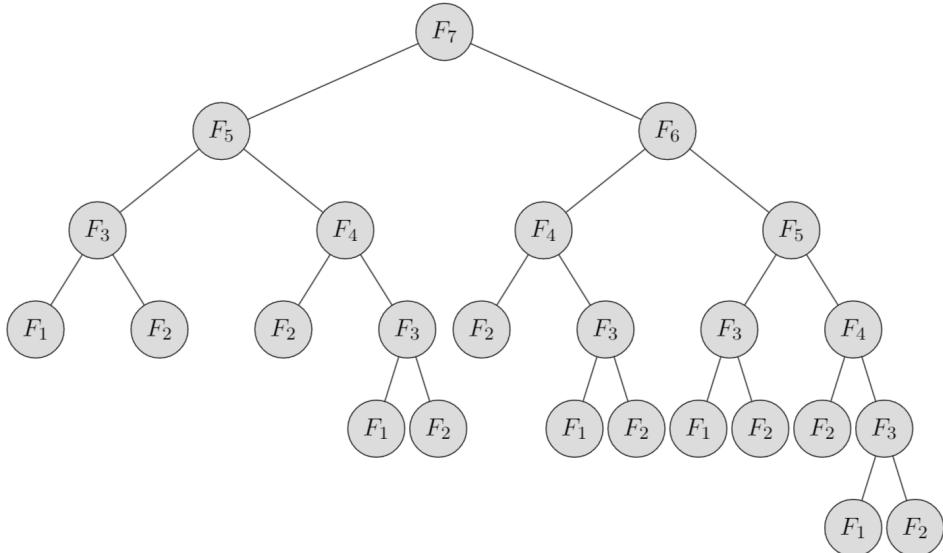
Fibonacci-Zahlen

- **Folge von Fibonacci-Zahlen:**
 - Definition: $F_1 = F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$ für alle $n \geq 3$.
 - Die Folge beginnt mit: 1, 1, 2, 3, 5, 8, 13, ...
- **Einfacher rekursiver Algorithmus:**

```
Fibonacci(n):
    if n == 1 oder n == 2:
        return 1
    else:
        return Fibonacci(n - 1) + Fibonacci(n - 2)
```

Laufzeit

- **Gesamtzahl der Aufrufe für die i -te Fibonacci-Zahl:**
 - Entspricht der i -ten Fibonacci-Zahl selbst.
 - Beispiele:
 - $F_{10} = 55$ (ca. 55 rekursive Aufrufe)
 - $F_{20} = 6765$
 - $F_{30} = 832040$
 - $F_{40} = 102334155$
- **Exponentielle Zeitkomplexität:**
 - Begründung:
 - $F_n = F_{n-1} + F_{n-2}$
 - Daraus folgt $F_n \geq 2F_{n-2}$ (da $F_{n-1} \geq F_{n-2}$ für $n \geq 3$)
 - Weiterhin gilt $F_2 = 1 = 2^0$ und $F_3 = 2 \geq 2^1$.
 - Allgemein lässt sich zeigen, dass $F_n \geq 2^{\lfloor (n-1)/2 \rfloor}$.
 - Eine genauere Analyse zeigt, dass $F_n \geq (\sqrt{2})^{n-2}$.



- Der Rekursionsbaum verzweigt sich exponentiell. Viele Teilprobleme werden redundant berechnet (z.B. `Fibonacci(3)` wird mehrfach aufgerufen bei der Berechnung von `Fibonacci(5)`).
- Der Rekursionsbaum hat fast gleich viele Blätter wie die Aufrufhöhe

Dynamische Programmierung (Rekursiv mit Memoization)

⌚ Speicherung - Memoization

- Die bereits berechneten Fibonacci-Zahlen werden in einem Array (z.B. F) zwischengespeichert.
- Vor jeder rekursiven Berechnung wird geprüft, ob das Ergebnis für das aktuelle n bereits im Array gespeichert ist. Wenn ja, wird der gespeicherte Wert direkt zurückgegeben.
- Heißt der rekursive Aufruf wird nur berechnet, wenn er noch nie berechnet wurde, also nur ein mal.

• Algorithmus:

Initialisiere ein Array F der Größe $n+1$ mit leeren Werten.

```
Fibonacci(n):
    if F[n] ist leer:
        if n == 1 oder n == 2:
            F[n] <- 1
        else:
            F[n] <- Fibonacci(n - 1) + Fibonacci(n - 2)
    return F[n]
```

• Laufzeit: $O(n)$

- Jeder Wert von $F[i]$ wird maximal einmal rekursiv berechnet.
- Es gibt n mögliche Werte für i (von 1 bis n).
- Die Überprüfung und der Zugriff auf das Array erfolgen in konstanter Zeit.

Dynamische Programmierung (Iterativ)

- **Speicherung:**
 - Die berechneten Fibonacci-Zahlen werden in einem Array (z.B. F) gespeichert und in der Berechnung wiederverwendet.
 - Die Berechnung erfolgt bottom-up, beginnend mit den Basisfällen.
- **Algorithmus:**

```
Linear-Fibonacci(n):
    F sei ein Array der Größe n+1.
    F[1] <- 1
    F[2] <- 1
    for i <- 3 bis n:
        F[i] <- F[i - 1] + F[i - 2]
    return F[n]
```

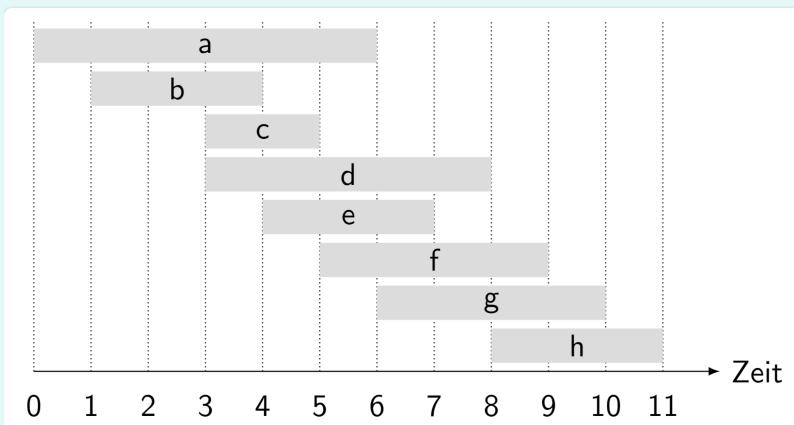
- **Laufzeit:** $O(n)$
 - Die Schleife wird $n - 2$ Mal durchlaufen.
 - Jeder Schleifendurchlauf benötigt konstanten Aufwand (Addition und Zuweisung).

Gewichtetes Intervall Scheduling

mehr dazu: [siehe hier](#)

⌚ Gewichtetes Interval Scheduling

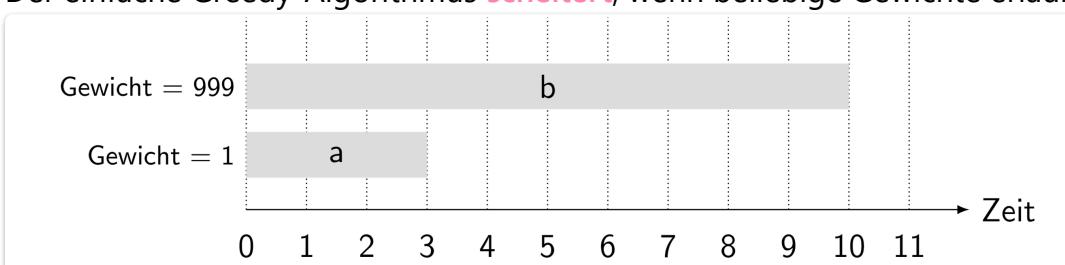
- Gegeben sind Jobs, wobei Job j startet zum Zeitpunkt s_j , endet zum Zeitpunkt f_j und hat ein Gewicht $w_j > 0$.
- Zwei Jobs sind **kompatibel**, wenn sie sich nicht überlappen.
- **Ziel:** Finde eine Teilmenge von paarweise kompatiblen Jobs mit **maximalem Gesamtgewicht**.



Da gabs die Greedy-Variante EDF (Earliest Deadline first), da wissen wir aber nicht, ob das auch das Gewicht maximiert.

Interval Scheduling: Rückblick

- **Wiederholung (einfaches Interval Scheduling):**
 - Ein Greedy-Algorithmus funktioniert, wenn alle Gewichte gleich 1 sind.
 - Vorgehensweise:
 - Berücksichtige Jobs in aufsteigender Reihenfolge der Endigungszeit.
 - Füge einen Job zur Teilmenge hinzu, wenn er kompatibel mit dem zuvor ausgewählten Job ist.
- **Beobachtung:**
 - Der einfache Greedy-Algorithmus **scheitert**, wenn beliebige Gewichte erlaubt sind.

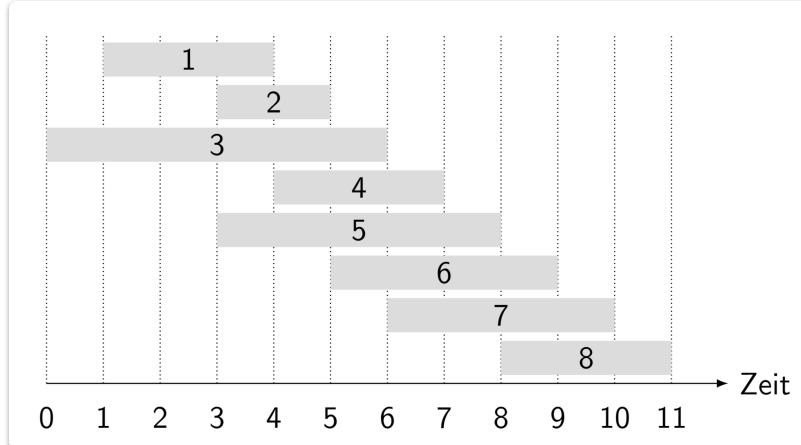


Gewichtetes Interval Scheduling

- Notation:

- Ordne die Jobs aufsteigend sortiert nach ihrer Beendigungszeit: $f_1 \leq f_2 \leq \dots \leq f_n$.
- Definition: $p(j)$ ist der größte Index $i < j$, sodass Job i kompatibel zu Job j ist (d.h., $f_i \leq s_j$). Falls kein solcher Job existiert, ist $p(j) = 0$.

- Beispiel:



- $p(8) = 5$ (Job 5 endet vor Job 8 und überlappt nicht)
- $p(7) = 3$ (Job 3 endet vor Job 7 und überlappt nicht)
- $p(2) = 0$ (Kein Job mit kleinerem Index ist mit Job 2 kompatibel)

Dynamische Programmierung: Binäre Auswahl

- Notation:

- $OPT(j)$ = Wert der optimalen Lösung für das Problem, bestehend aus den Jobs $1, 2, \dots, j$ (die nach Endzeit sortiert sind).

- Wir unterscheiden zwei Fälle für die optimale Lösung $OPT(j)$:

- Fall 1: $OPT(j)$ wird erreicht mit einer Lösung, die den Job j enthält.
- Fall 2: $OPT(j)$ wird erreicht mit einer Lösung, die den Job j nicht enthält.

- Konsequenz:

- Fall 1 (Job j ist in der optimalen Lösung):

- Die Lösung kann keine inkompatiblen Jobs $\{p(j) + 1, p(j) + 2, \dots, j - 1\}$ enthalten.
- Daher besteht die optimale Lösung in diesem Fall aus Job j und der optimalen Lösung für die Jobs $1, \dots, p(j)$.
- Der Wert dieser Lösung ist $w_j + OPT(p(j))$.

- Fall 2 (Job j ist nicht in der optimalen Lösung):

- Dann ist der Wert der optimalen Lösung für die Jobs $1, \dots, j$ derselbe wie der Wert der optimalen Lösung für die Jobs $1, \dots, j - 1$.
- Der Wert dieser Lösung ist $OPT(j - 1)$.

- Rekursive Definition für $OPT(j)$:

$$OPT(j) = \begin{cases} 0 & \text{wenn } j = 0 \\ \max\{w_j + OPT(p(j)), OPT(j - 1)\} & \text{sonst} \end{cases}$$

- **Basisfall:** Wenn keine Jobs ($j = 0$) betrachtet werden, ist der optimale Wert 0.
- **Rekursiver Schritt:** Für $j > 0$ wählen wir das Maximum zwischen:
 - Dem Gewicht des aktuellen Jobs j plus dem optimalen Wert der kompatiblen Jobs davor ($p(j)$).
 - Dem optimalen Wert ohne den aktuellen Job j (d.h., der optimale Wert bis zum vorherigen Job $j - 1$).

Gewichtetes Interval Scheduling: Brute-Force-Ansatz

Brute-Force-Algorithmus

- **Eingabe:** n Jobs mit Startzeiten s_1, \dots, s_n , Endzeiten f_1, \dots, f_n und Gewichten w_1, \dots, w_n .
- Sortiere Jobs nach Beendigungszeit, sodass $f_1 \leq f_2 \leq \dots \leq f_n$.
- Berechne $p(1), p(2), \dots, p(n)$ (der größte Index $i < j$, sodass Job i mit Job j kompatibel ist).

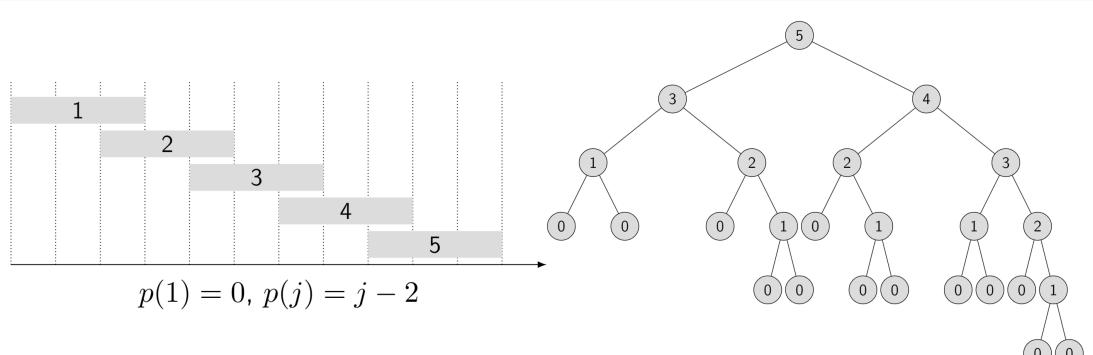
- **Rekursive Funktion zur Berechnung des optimalen Werts:**

```
Compute-Opt(j):
    if j == 0:
        return 0
    else:
        return max(w_j + Compute-Opt(p(j)), Compute-Opt(j - 1))
```

- **Beobachtung:**

- Der rekursive Algorithmus ist ineffizient wegen **redundanter** Subprobleme.
- Die Anzahl der rekursiven Aufrufe wächst **exponentiell**.

Beispiel



- Für diese spezielle Anordnung gilt: $p(1) = 0$, $p(j) = j - 2$ für $j \geq 2$.
- Der Rekursionsbaum zeigt, dass viele Compute-Opt-Aufrufe mit den gleichen Argumenten wiederholt ausgeführt werden (z.B. Compute-Opt(0) und Compute-Opt(1)).
- Die Struktur des Rekursionsbaums ähnelt der eines Binärbaums, und die Anzahl der Knoten wächst exponentiell mit n . Dies führt zu einer exponentiellen Laufzeit des Brute-Force-Ansatzes.

Gewichtetes Interval Scheduling: Speicherung (Memoization)

"top-down"

- **Speicherung:**
 - Speichere die Ergebnisse jedes Teilproblems in einem Cache (z.B. einem Array M).
 - Berechnung eines Teilproblems erfolgt nur noch, wenn dessen Ergebnis noch nicht gespeichert ist.
- **Allgemein:**
 - **Eingabe:** n Jobs mit Startzeiten s_1, \dots, s_n , Endzeiten f_1, \dots, f_n und Gewichten w_1, \dots, w_n .
 - Sortiere Jobs nach Beendigungszeit, sodass $f_1 \leq f_2 \leq \dots \leq f_n$.
 - Berechne $p(1), p(2), \dots, p(n)$.
- **Algorithmus mit Memoization:**

```

Initialisiere ein Array M der Größe n+1 mit leeren Werten.
M[0] <- 0

Compute-Opt(j):
    if M[j] ist leer:
        M[j] <- max(w_j + Compute-Opt(p(j)), Compute-Opt(j - 1))
    return M[j]

```

- **Globales Array:** M ist ein globales Array, um die berechneten optimalen Werte zu speichern.
- Sehr ähnlich zu **Brute-force-Ansatz** aber das `if-statement` wurde hinzugefügt.

Gewichtetes Interval Scheduling: Laufzeit

- **Behauptung:** Die Version mit Speicherung (Memoization) benötigt $O(n \log n)$ Zeit.
- **Begründung der Laufzeit:**
 - **Sortieren nach Beendigungszeit:** $O(n \log n)$.
 - **Berechne $p(j)$:** $O(n \log n)$ mittels binärer Suche (für jedes Intervall) auf der nach Beendigungszeit sortierten Folge der Endzeiten, um den größten kompatiblen Job

mit kleinerem Index zu finden. Für jeden der n Jobs wird eine binäre Suche in $O(\log n)$ Zeit durchgeführt.

- --> liefert Werte $p(1) \dots p(n)$
- **Compute-Opt(j)** : Jeder Aufruf benötigt $O(1)$ Zeit (ohne die rekursiven Aufrufe selbst) und
 - (i) liefert entweder einen bereits existierenden Wert aus $M[j]$ zurück.
 - (ii) oder berechnet einen neuen Eintrag für $M[j]$ und macht **zwei** rekursive Aufrufe. Da jeder Eintrag in M nur einmal berechnet wird, gibt es maximal $n + 1$ rekursive Aufrufe, die tatsächlich eine Berechnung durchführen. Jeder dieser Aufrufe benötigt konstante Zeit.
- **Maß für den Fortschritt $\varphi =$ die Anzahl der nicht leeren Einträge in M .**
 - Am Anfang gilt $\varphi = 0$, danach $\varphi \leq n$.
 - Jeder rekursive Aufruf, der einen leeren Eintrag berechnet, erhöht φ um 1. Da φ maximal n erreicht, gibt es maximal n solcher Aufrufe.
- Die gesamte Laufzeit von **Compute-Opt(n)** ist somit $O(n)$.
- **Gesamlaufzeit:** Die Gesamlaufzeit des Algorithmus wird durch das Sortieren und die Berechnung der $p(j)$ -Werte dominiert, gefolgt von den maximal n konstanten Zeitaufgaben von **Compute-Opt**. Daher ist die Gesamlaufzeit $O(n \log n)$.

Gewichtetes Interval Scheduling: Bottom-up

- **Bottom-up dynamische Programmierung: Iterative Lösung.**

Allgemein

- **Eingabe:** n Jobs mit Startzeiten s_1, \dots, s_n , Endzeiten f_1, \dots, f_n und Gewichten w_1, \dots, w_n .
- Sortiere Jobs nach Beendigungszeit, sodass $f_1 \leq f_2 \leq \dots \leq f_n$.
- Berechne $p(1), p(2), \dots, p(n)$.

- **Iterativer Algorithmus:**

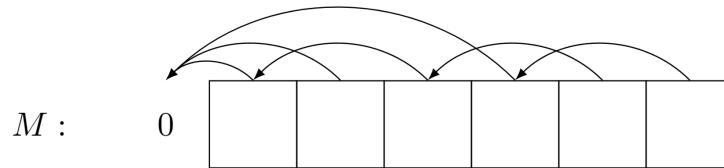
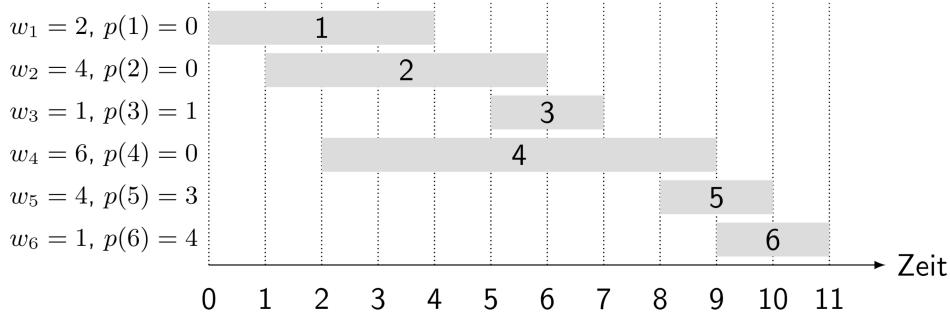
```
Iterative-Compute-Opt():
  M sei ein Array der Größe n+1.
  M[0] <- 0
  for j <- 1 bis n:
    M[j] <- max(w_j + M[p(j)], M[j - 1])
  return M[n]
```

- **Laufzeit:** Die Laufzeit von **Iterative-Compute-Opt** ist $O(n)$ (die Schleife läuft von 1 bis n). Das Sortieren der Jobs und die Vorberechnung von $p(\cdot)$ benötigen weiterhin $O(n \log n)$. Die **Gesamlaufzeit** des Bottom-up-Ansatzes beträgt somit $O(n \log n)$.

☰ Beispiel

Gegeben:

- $n = 6$ Jobs mit Gewichten $w_i, i = 1 \dots n$.
- Jobs sind schon sortiert nach Beendigungszeit.



$$w_j + M[p(j)]:$$

$$M[j - 1]:$$



$$w_j + M[p(j)]:$$

$$M[j - 1]: \quad \textcolor{red}{2}$$



$$w_j + M[p(j)]: \quad \textcolor{red}{2} \quad \textcolor{red}{4}$$

$$M[j - 1]: \quad 0 \quad 2$$

$$\begin{array}{l}
 M : \quad \boxed{\begin{array}{|c|c|c|c|c|c|} \hline 0 & 2 & 4 & 4 & 6 & \\ \hline \end{array}} \\
 w_j + M[p(j)]: \quad \begin{array}{ccccc} 2 & 4 & 3 & 6 & \end{array} \\
 M[j - 1]: \quad \begin{array}{ccccc} 0 & 2 & 4 & 4 & \end{array}
 \end{array}$$

$$\begin{array}{l}
 M : \quad \boxed{\begin{array}{|c|c|c|c|c|c|} \hline 0 & 2 & 4 & 4 & 6 & 8 \\ \hline \end{array}} \\
 w_j + M[p(j)]: \quad \begin{array}{ccccc} 2 & 4 & 3 & 6 & 8 \\ \hline \end{array} \\
 M[j - 1]: \quad \begin{array}{ccccc} 0 & 2 & 4 & 4 & 6 \\ \hline \end{array}
 \end{array}$$

$$\begin{array}{l}
 M : \quad \boxed{\begin{array}{|c|c|c|c|c|c|} \hline 0 & 2 & 4 & 4 & 6 & 8 \\ \hline \end{array}} \\
 w_j + M[p(j)]: \quad \begin{array}{ccccc} 2 & 4 & 3 & 6 & 8 & 7 \\ \hline \end{array} \\
 M[j - 1]: \quad \begin{array}{ccccc} 0 & 2 & 4 & 4 & 6 & 8 \\ \hline \end{array}
 \end{array}$$

$$\begin{array}{l}
 M : \quad \boxed{\begin{array}{|c|c|c|c|c|c|} \hline 0 & 2 & 4 & 4 & 6 & 8 \\ \hline \end{array}} \\
 w_j + M[p(j)]: \quad \begin{array}{ccccc} 2+0 & 4+0 & 1+2 & 6+0 & 4+4 & 1+6 \\ \hline \end{array} \\
 M[j - 1]: \quad \begin{array}{ccccc} 0 & 2 & 4 & 4 & 6 & 8 \\ \hline \end{array}
 \end{array}$$

so ein Beispiel haben wir auch als ue: [ue7-A2-\(b\)](#)

Gewichtetes Interval Scheduling: Finden einer Lösung

② Frage

- Der Algorithmus berechnet den optimalen Wert. Wie bekommen wir aber die Menge der ausgewählten Jobs (die eigentliche Lösung)?

- **Antwort:** Durch eine Nachbearbeitung (Backtracking) des berechneten M -Arrays.

Ablauf

1. Führe `M-Compute-Opt(n)` oder `Iterative-Compute-Opt(n)` aus, um das Array M mit den optimalen Werten bis zu jedem Job j zu füllen.
2. Führe eine rekursive Funktion `Find-Solution(n)` aus, um die Menge der ausgewählten Jobs zu rekonstruieren.

- **Algorithmus zur Rekonstruktion der Lösung:**

```
Find-Solution(j):
    if j == 0:
        Keine Ausgabe
    elseif w_j + M[p(j)] > M[j - 1]:
        Gib Job j aus
        Find-Solution(p(j))
    else:
        Find-Solution(j - 1)
```

- Die Funktion geht das Array M von hinten nach vorne durch.
- Für jeden Job j wird geprüft, ob die Einbeziehung von Job j zu einem höheren Wert führt ($w_j + M[p(j)] > M[j - 1]$).
- Wenn ja, wird Job j zur Lösung hinzugefügt und die Suche wird mit $p(j)$ fortgesetzt (da Jobs zwischen $p(j)$ und $j - 1$ nicht in der optimalen Lösung sein können, wenn j enthalten ist).
- Wenn nein, wird Job j nicht zur Lösung hinzugefügt und die Suche wird mit $j - 1$ fortgesetzt.
- **Laufzeit:** Die Anzahl der rekursiven Aufrufe von `Find-Solution` ist maximal n , da in jedem Schritt j reduziert wird. Daher ist die Laufzeit von `Find-Solution` $O(n)$.

Beispiel - Ergebnis finden

Find-Solution (6)

 $M :$

0	2	4	4	6	8	8
0	2	4	3	6	8	7
	2	4	4	6	6	8

 $w_j + M[p(j)]:$ $M[j - 1]:$

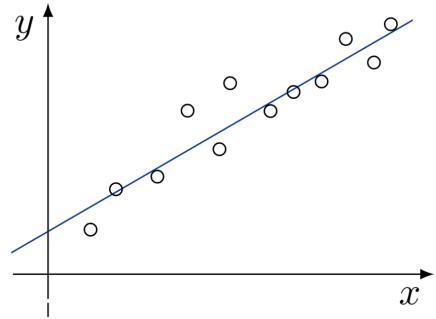
Segmented Least Squares

Least Squares

⌚ Definition

- **Fundamentales Problem in der Statistik und der Numerischen Analyse.**
- **Gegeben:** n Punkte in der Ebene: $(x_1, y_1), \dots, (x_n, y_n)$.
- **Finde:** Eine Gerade $y = ax + b$, welche die Summe der quadrierten Fehler minimiert.
- **Fehlerfunktion:** $\text{Err} = \sum_{i=1}^n (y_i - (ax_i + b))^2$

$$\text{Err} = \sum_{i=1}^n (y_i - (ax_i + b))^2$$



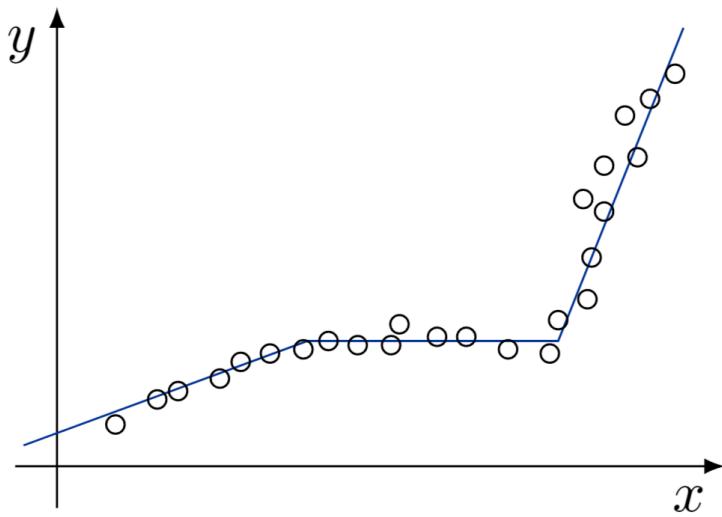
- **Analytische Lösung:** Der minimale Fehler wird erreicht, wenn:

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

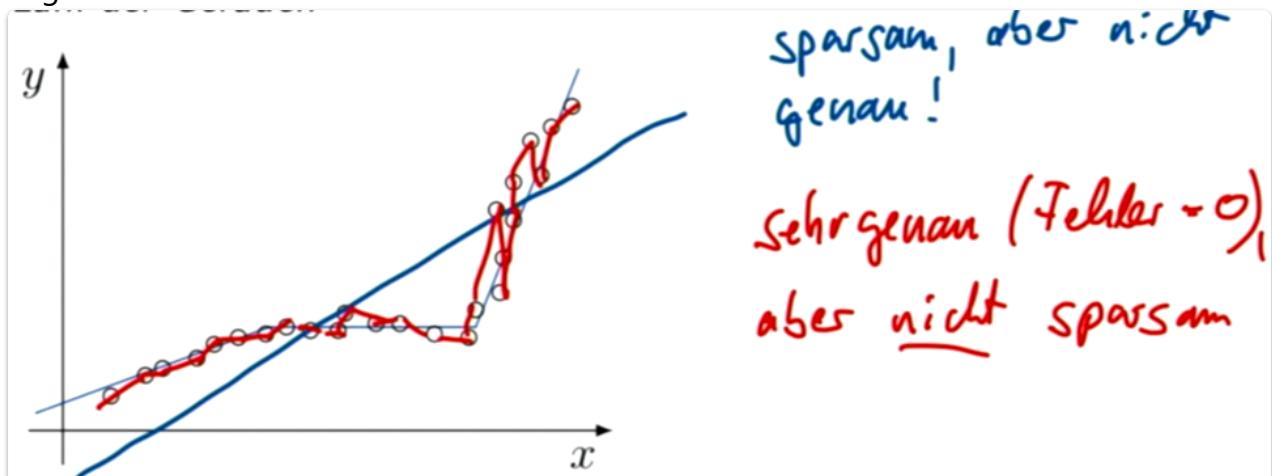
Beste Lösung (a, b) kann man in $O(n)$ berechnen

Segmented Least Squares

- **Punkte durch eine Folge von Geradensegmenten annähern.**
- **Gegeben:** n Punkte in der Ebene $(x_1, y_1), \dots, (x_n, y_n)$ so dass $x_1 < x_2 < \dots < x_n$.
- **Finde:** Eine Folge von Geraden, welche eine bestimmte Funktion $f(x)$ minimiert.
- **Frage:** Was ist eine angemessene Wahl für $f(x)$? Die Funktion $f(x)$ sollte sowohl **Genauigkeit** als auch **Sparsamkeit** gewährleisten.
 - **Genauigkeit:** Höhe des Fehlers.
 - **Sparsamkeit:** Anzahl der Geraden (Segmente).



- **Finde:** Eine Folge von Geraden welche:
 - die Summe der quadrierten Fehler E in jedem Segment
 - die Anzahl der Geraden L
 - minimiert.
- **Tradeoff Funktion:** $E + cL$, für eine Konstante $c > 0$. Der Parameter c gewichtet das Verhältnis zwischen der Reduktion des Fehlers und der Hinzufügung eines neuen Segments.



Dynamischer Ansatz: Segmented Least Squares

Notation

- $OPT(j)$ = minimale Kosten für die Punkte p_1, p_2, \dots, p_j .
- $e(i, j)$ = minimale Summe des quadrierten Fehlers für die Punkte p_i, p_{i+1}, \dots, p_j , wenn diese durch eine einzige Gerade approximiert werden.

Berechnen von $OPT(j)$

- Betrachte das letzte Segment, das die Punkte p_i, p_{i+1}, \dots, p_j für ein bestimmtes i nutzt.

- Die Kosten für diese Lösung wären die optimalen Kosten für die Punkte bis p_{i-1} plus die Kosten des letzten Segments (Fehler plus Kosten für ein neues Segment).
- Kosten = $OPT(i-1) + e(i, j) + c$.

⌚ Rekursive Definition für $OPT(j)$

$$OPT(j) = \begin{cases} 0 & \text{falls } j = 0 \\ \min_{1 \leq i \leq j} \{ OPT(i-1) + e(i, j) + c \} & \text{sonst} \end{cases}$$

alle Mögl. für Start des letzten Segm. *Lsg. für $1, \dots, i-1$* *neuer Fehler für i, \dots, j* *Kosten für ein neues Segm.*

- Basisfall:** $OPT(0) = 0$ (keine Punkte, keine Kosten).
- Rekursiver Schritt:** Um die optimalen Kosten bis zum Punkt j zu finden, betrachten wir alle möglichen letzten Segmente, die bei einem Punkt i ($1 \leq i \leq j$) beginnen und bis j gehen. Für jedes solche Segment berechnen wir die Kosten als die optimalen Kosten bis zum Punkt $i - 1$ plus den Fehler des Segments von i bis j plus die Kosten c für das Hinzufügen eines neuen Segments. Wir wählen das Minimum über alle möglichen Startpunkte i .

Segmented Least Squares: Algorithmus

```

Segmented-Least-Squares(  $P = \{p_1, p_2, \dots, p_n\}$  )
   $M[0] = 0$ 
   $O(n^2)$  Iterat. { for  $j \leftarrow 1$  bis  $n$ 
    for  $i \leftarrow 1$  bis  $j$ 
      berechne Fehler  $e(i, j)$  für Punkte  $p_i, \dots, p_j$ 
       $O(n)$  Zeit
  }
   $O(n^3)$  Vorberechnung von  $e(i, j)$ 

   $O(n)$  Iterat. { for  $j \leftarrow 1$  to  $n$ 
     $M[j] = \min_{1 \leq i \leq j} (M[i-1] + e(i, j) + c)$ 
     $O(n)$  Mögl.
  }
   $O(n^2)$ 
  return  $M[n]$ 

```

- Laufzeit:** $O(n^3)$.
 - Flaschenhals:** Das Berechnen von $e(i, j)$ für $O(n^2)$ Paare (i, j) benötigt $O(n)$ Operationen pro Paar (um die Parameter der besten Geraden und den Fehler zu berechnen).
 - Die äußere Schleife für j läuft von 1 bis n .
 - Die innere Schleife für i läuft von 1 bis j (bis zu n Mal).
 - Innerhalb der inneren Schleife wird $e(i, j)$ berechnet und das Minimum gefunden.

- **Finden einer Lösung:** Analog zum Interval Scheduling durch Rückverfolgen der Minimierung im Array M . Wir merken uns bei der Berechnung von $M[j]$, welcher Wert von i das Minimum erzeugt hat, und rekonstruieren so die Segmente.
- **Verbesserungsmöglichkeiten:** Kann mithilfe geschickterer Vorberechnung und Wiederverwendung von Zwischenergebnissen zu $O(n^2)$ verbessert werden.

Zwischenfazit:

Zwischenfazit

• wir betrachten polynomiale Anzahl Teilprobleme (Arraygröße)

• optimale Lösung lässt sich aus opt. Lösungen geeigneter Teilprobleme zusammensetzen

• Teilprobleme lassen sich von klein nach groß anfüllen und rekursiv lösen, d.h.

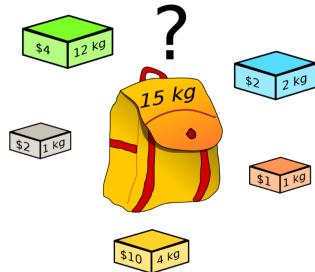
for $j = 1, \dots, n$

$M[j] = f(M[1], \dots, M[j-1])$ // z.B. binäre Auswahl oder
lineare Auswahl
return $M[n]$

• Rekonstruktion der besten Lösung (nicht nur des Wertes) durch Backtracking

Rucksackproblem

Gegeben: n Gegenstände mit positiv rationalen Gewichten g_1, \dots, g_n , Werten w_1, \dots, w_n und einer positiv rationalen Kapazität G .



Gesucht: Teilmenge S der Gegenstände mit Gesamtgewicht $\leq G$ und maximalem Gesamtwert.

Vereinfachung: Der Einfachheit halber nehmen wir im folgenden an, dass sowohl die Gewichte als auch die Kapazität positiv ganzzahlig sind.

☰ Beispiel

#	Wert	Gewicht	w_i/g_i
1	1	1	1
2	6	2	3
3	18	5	$3\frac{3}{5}$
4	22	6	$3\frac{2}{3}$
5	28	7	4

$$G = 11$$

Greedy: Füge wiederholt einen Gegenstand mit einem maximalen Verhältnis von w_i/g_i , der in den Rucksack passt, hinzu.

Beispiel: $\{3,4\}$ ergibt einen Gesamtwert von 40.

Beispiel: $\{5,2,1\}$ ergibt nur einen Gesamtwert von 35 \Rightarrow Greedy ist nicht optimal.

Idee und Motivation

- Im Kapitel über Branch and Bound wurde ein Algorithmus mit Laufzeit $O(2^n)$ vorgestellt.
- Ziel ist ein Algorithmus mit Laufzeit $O(nG)$, falls $nG < 2^n$ wesentlich effizienter ist. (G ... Eingabegröße)
- Die Intuition hinter dem Algorithmus ist, dass man nur (Teil-)Lösungen mit verschiedenen Gewichten unterscheiden muss.

Dynamische Programmierung:

⚡ Falscher Ansatz

Definition: $OPT(i)$ = Maximaler Profit für die Teilmenge von den Gegenständen $1, \dots, i$

- **Fall 1:** $OPT(i)$ wird erreicht mit einer Lösung, die den Gegenstand i nicht enthält.
 - Es gilt $OPT(i) = OPT(i - 1)$, da wir wissen, dass die Lösung den Gegenstand i nicht enthält.
- **Fall 2:** $OPT(i)$ wird erreicht mit einer Lösung, die den Gegenstand i enthält.
 - Das Akzeptieren von Gegenstand i impliziert nicht, dass wir andere Gegenstände nicht aufnehmen werden.
 - Ohne zu wissen, welche anderen Gegenstände vor i ausgewählt wurden, können wir nicht sagen, ob wir für i und die nachfolgenden Gegenstände genug Platz haben.

Lösungsansatz: Die Berechnung der Teilprobleme muss die verbleibende Gesamtkapazität berücksichtigen!

⌚ Richtiger Ansatz: Gewichtsbeschränkung

Definition: $OPT(i, g)$ = Maximaler Profit für die Teilmenge von den Gegenständen $1, \dots, i$, mit einer Gewichtsbeschränkung g .

- **Fall 1:** $OPT(i, g)$ wird erreicht mit einer Lösung, die den Gegenstand i nicht enthält.
 - Es gilt $OPT(i, g) = OPT(i - 1, g)$, da wir wissen, dass die Lösung den Gegenstand i nicht enthält und die Gewichtsbeschränkung gleich bleibt.
- **Fall 2:** $OPT(i, g)$ wird erreicht mit einer Lösung, die den Gegenstand i enthält.
 - Neue Gewichtsbeschränkung = $g - w_i$.
 - Daher gilt dann $OPT(i, g) = w_i + OPT(i - 1, g - w_i)$.

$$OPT(i, g) = \begin{cases} 0 & \text{wenn } i = 0 \\ OPT(i - 1, g) & \text{wenn } g_i > g \\ \max \{OPT(i - 1, g), w_i + OPT(i - 1, g - w_i)\} & \text{sonst} \end{cases}$$

Fall1 Fall2

Rucksackproblem: Bottom-Up

Rucksack: Befülle ein $(n + 1) \times (G + 1)$ Array.

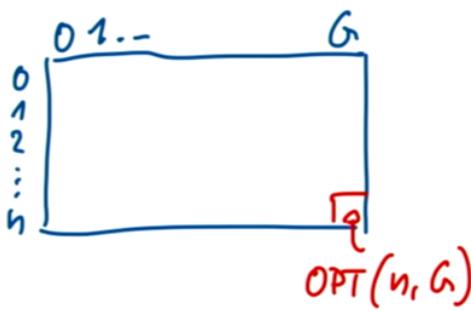
Eingabe: $n, G, g_1, \dots, g_n, w_1, \dots, w_n$

```

for  $g \leftarrow 0$  bis  $G$ 
     $M[0, g] \leftarrow 0$ 

for  $i \leftarrow 1$  bis  $n$ 
    for  $g \leftarrow 0$  bis  $G$ 
        if  $g_i > g$ 
             $M[i, g] \leftarrow M[i - 1, g]$ 
        else
             $M[i, g] \leftarrow \max\{M[i - 1, g], w_i + M[i - 1, g - g_i]\}$ 
return  $M[n, G]$ 

```



Als Platz haben wir die Tabelle mit n Zeilen und g Spalten und

☰ Beispiel

Erstellen von Tabelle

	$G + 1$											
	0	1	2	3	4	5	6	7	8	9	10	11
n + 1 ↓	\emptyset	0	0	0	0	0	0	0	0	0	0	0
	{ 1 }	0	1	1	1	1	1	1	1	1	1	1
	{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7
	{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25
	{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	40
	{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35

$OPT: \{ 4, 3 \}$
 $Wert = 22 + 18 = 40$

$G = 11$

Gegenstand	Wert	Gewicht
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Backtracking: Bestimmen der Lösung

Nach Erstellen der Tabelle → Backtracking um Lösung zu bekommen

Bestimmen der Lösung

Optimale Lösung: Mit Hilfe der Werte im Array M :

Find-Solution(M):

```

 $i \leftarrow n$ 
 $k \leftarrow G$ 
 $A \leftarrow \emptyset$ 
while  $i > 0$  und  $k > 0$ 
    if  $M[i, k] \neq M[i-1, k]$ 
         $A \leftarrow A \cup \{i\}$ 
         $k \leftarrow k - g_i$ 
     $i \leftarrow i - 1$ 
return  $A$ 
```

Rucksackproblem: Laufzeit

Laufzeit: $O(nG)$.

- Polynomiell in n .
- Aber die Laufzeit hängt auch von der Rucksackkapazität G ab.
 - G ist exponentiell in der Eingabelänge, weil Zahlen binär kodiert werden.
 - Die Laufzeit ist somit nicht polynomiell in der Eingabelänge beschränkt.

$$G = 2^{\log(G)}$$

$$\implies O(n \cdot 2^{\log(G)})$$

- „Pseudo-polynomiell.“

Allgemein: Falls $P \neq NP$ kann das Rucksackproblem nicht in Polynomialzeit gelöst werden.

Verbesserung

Speicherung: Man muss eigentlich nicht das gesamte Array speichern.

Verbesserung:

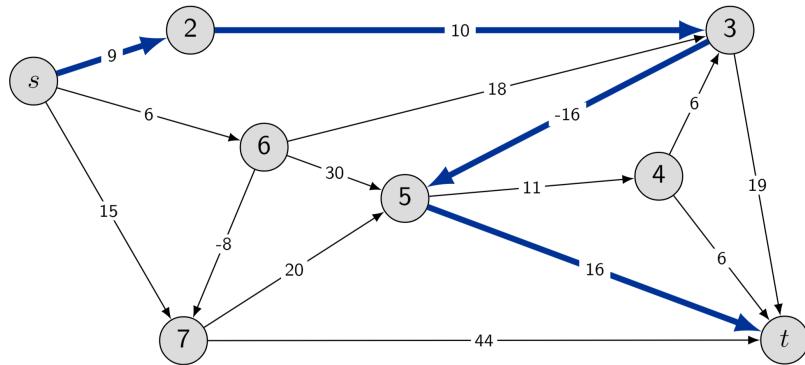
- Man merkt sich nur die letzte Zeile.
- Die Aktualisierung der Einträge erfolgt von rechts nach links.

Kürzeste Pfade

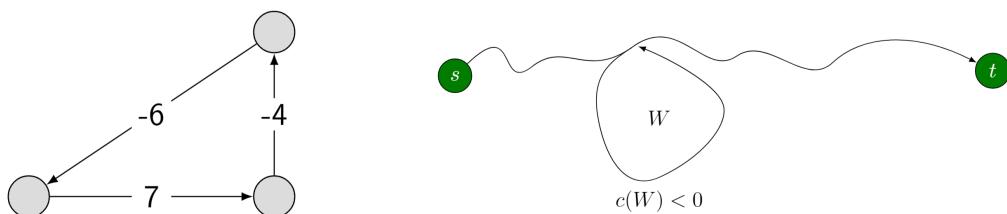
Kürzester Kantenzug: Gegeben sei ein gerichteter Graph $G = (V, E)$, mit Kantengewichten c_{vw} . Finde einen kürzesten Kantenzug zwischen Knoten s und t .

- erlaube negative Gewichte (bei Dijkstra-Algorithmus nicht erlaubt)

Beispiel:



Kreise mit negativen Kosten (negative Kreise)



Bemerkung: Im Falle von negativen Kreisen:

- keine sinnvolle Definition von kürzesten **Kantenzügen** mehr möglich.
- Definition von kürzesten **Pfaden** bleibt jedoch sinnvoll.
- **Unterschied:** In einem Pfad wird jeder Knoten maximal einmal besucht, in einem Kantenzug beliebig oft.

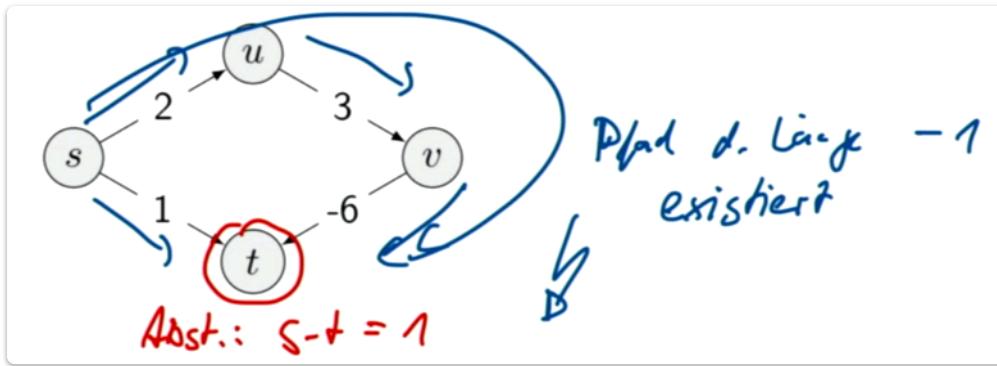
Kürzeste Pfade: Komplexität

Theorem: Das Finden eines kürzesten Pfades in einem gerichteten Graphen mit reellwertigen Kantengewichten ist **NP-vollständig**.

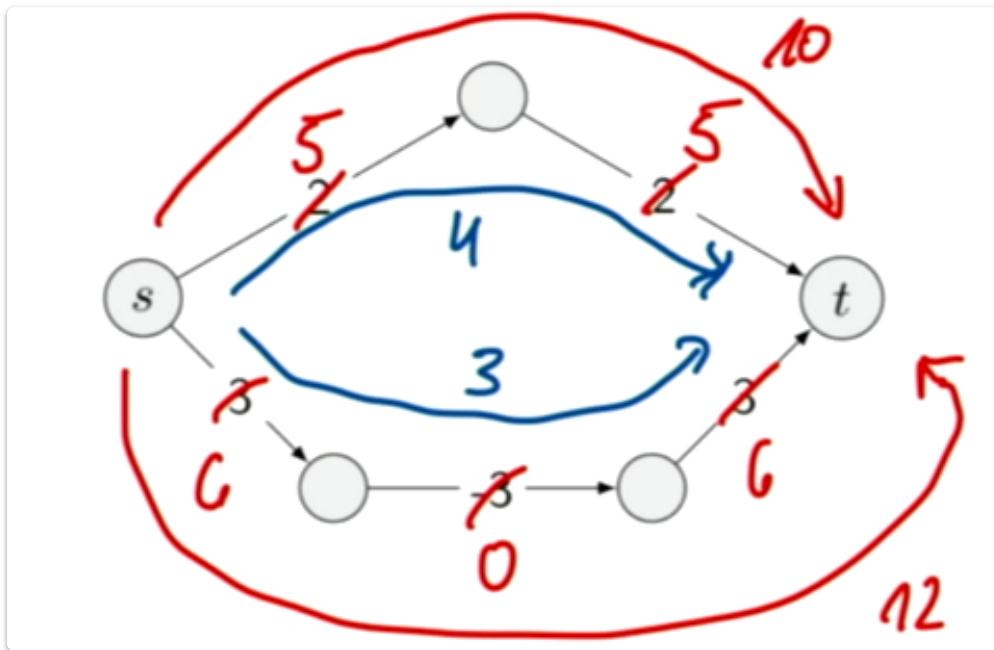
Bemerkungen: Verbietet man negative Kreise wird das Problem jedoch wieder in Polynomialzeit lösbar.

Kürzeste Pfade: Falsche Ansätze

Algorithmus von Dijkstra: Schlägt fehl bei negativen Kantengewichten.



Neugewichtung: Zu jedem Kantengewicht eine Konstante dazuzugeben schlägt fehl.



Kürzeste Pfade: Bellman's Gleichungen

Sei $G = (V, E)$ ein gerichteter Graph ohne negative Kreise mit Kantengewichten c_{vw} und $t \in V$, dann gilt für die Länge $OPT(v)$ eines kürzesten $v - t$ Pfades P in G :

$$OPT(t) = 0$$

$$OPT(v) = \min_{(v,w) \in E} \{c_{vw} + OPT(w)\}, \forall v \in V, v \neq t$$

Beweis:

Wir zeigen mit Induktion über die Anzahl Kanten auf dem kürzesten $v - t$ Pfad, dass $OPT(v)$ die Länge dieses kürzesten $v - t$ Pfades ist.

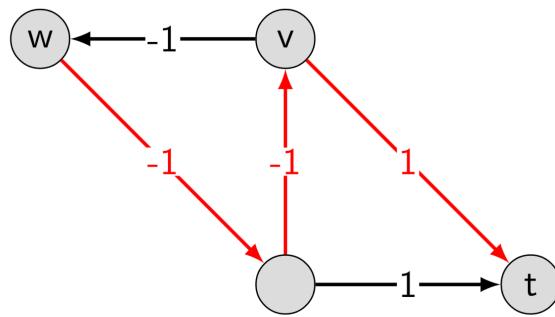
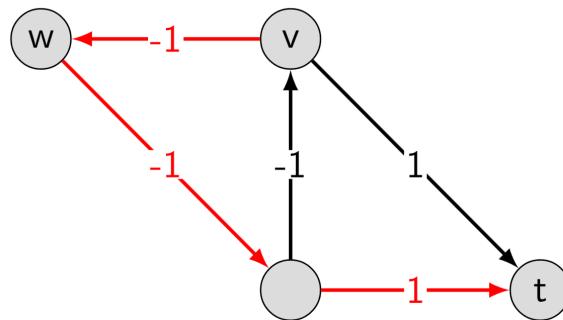
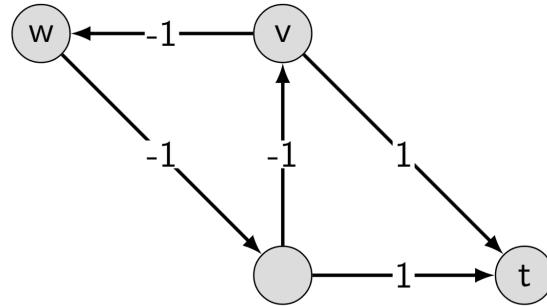
- **Basisfall:** Aussage gilt für $v = t$ mit 0 Kanten.
- **Induktionsvoraussetzung:**
 - Sei P ein kürzester $v - t$ Pfad mit ℓ Kanten und (v, w) die erste Kante von P .
 - Da es keine negativen Kreise gibt, liegt ein Knoten nie zweimal auf einem kürzesten Kantenzug.

- $P - (v, w)$ ist ein kürzester $w - t$ Pfad mit $\ell - 1$ Kanten, der v nicht enthält. Wegen der Induktionsvoraussetzung ist seine Länge $OPT(w)$.
- Damit ist aber $c_{vw} + OPT(w)$ die Länge von P und $OPT(v) \leq c_{vw} + OPT(w)$.
- Wäre $OPT(v) < c_{vw} + OPT(w)$, so gäbe es einen kürzeren $v - t$ Pfad P' über einen anderen Zwischenknoten $w' \rightarrow \dots \rightarrow t$. Widerspruch zur Annahme, dass P kürzester $v - t$ Pfad ist.

$$\implies OPT(v) = c_{vw} + OPT(w)$$

Gegenbeispiel

Falls G negative Kreise enthält, gelten Bellman's Gleichungen nicht mehr:

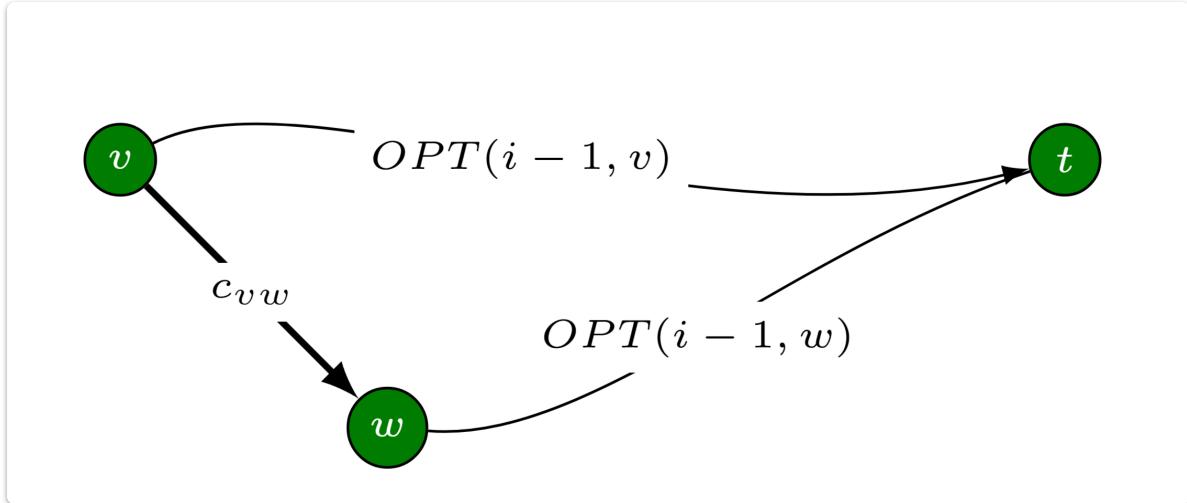


Wie man sieht, gilt hier die Bellman's Gleichung nicht mehr, da sich die Kosten durch wiederholtes Durchlaufen des negativen Kreises beliebig verringern lassen. Es existiert kein "kürzester" Pfad mehr im Sinne einer festen Länge, da man durch den negativen Kreis immer "kürzer" werden kann.

Kürzester Pfad: Dynamische Programmierung

Definition: $OPT(i, v) =$ Länge eines kürzesten $v - t$ Pfades P , der höchstens i Kanten benutzt.

- **Fall 1:** P benutzt höchstens $i - 1$ Kanten.
 - $OPT(i, v) = OPT(i - 1, v)$
- **Fall 2:** P benutzt genau i Kanten. Dann besteht der kürzeste Pfad aus einer ersten Kante (v, w) und dem besten $w - t$ Pfad, der höchstens $i - 1$ Kanten benutzt.



$$OPT(i, v) = \begin{cases} 0 & \text{wenn } i = 0 \text{ und } v = t \\ \infty & \text{wenn } i = 0 \text{ und } v \neq t \\ \min\{OPT(i - 1, v), \min_{(v,w) \in E}\{c_{vw} + OPT(i - 1, w)\}\} & \text{ansonsten} \end{cases}$$

Anmerkung: Aufgrund von Bellman's Gleichungen ist $OPT(n - 1, v) = OPT(v)$ Länge eines kürzesten $v - t$ Pfades, wenn es keine negativen Kreise gibt.

Denn ein "Pfad" mit $\geq n$ Kanten muss einen Kreis enthalten.

Maximal kann die Länge $n - 1$ sein, da keine Kante doppelt vorkommt.

Kürzester Pfad: Implementierung

```

Simple-Shortest-Path-DP( $G, s, t$ ):
foreach node  $v \in V$ 
     $M[0, v] \leftarrow \infty$ 
 $M[0, t] \leftarrow 0$ 
for  $i \leftarrow 1$  bis  $n - 1$ 
    foreach Knoten  $v \in V$ 
         $M[i, v] \leftarrow M[i - 1, v]$ 
    foreach Kante  $(v, w) \in E$ 
         $M[i, v] \leftarrow \min(M[i, v], c_{vw} + M[i - 1, w])$ 
return  $M[n - 1, s]$ 

```

Analyse: $O(mn)$ Zeit, $O(n^2)$ Platz.

$$n = |V|$$

$$m = |E|$$

$$n \leq m$$

☰ Beispiel für den nicht verbesserten Algorithmus

siehe Folien: AD_12_DynamischeProgrammierung, p.52

Man schaut sich bei jeder Iteration alle Knoten an und ob sie sich seit letzter Iteration verbessert haben / verbessern können.

Einen kürzesten Pfad finden: Verwalte den Nachfolger für jeden Knoten.

Korrektheit der Ergebnisse:

- Der beschriebene Algorithmus liefert immer eine korrekte Lösung, wenn **keine negativen Kreise im Graphen vorhanden** sind.
- Wenn negative Kreise vorhanden sind, kann der Algorithmus falsche Ergebnisse liefern.

Überprüfung auf negative Kreise:

- Nach dem Terminieren des Algorithmus.
- Für jeden Knoten v wird überprüft: Falls $OPT(n, v) < OPT(n - 1, v)$, dann gibt es einen negativen Kreis.

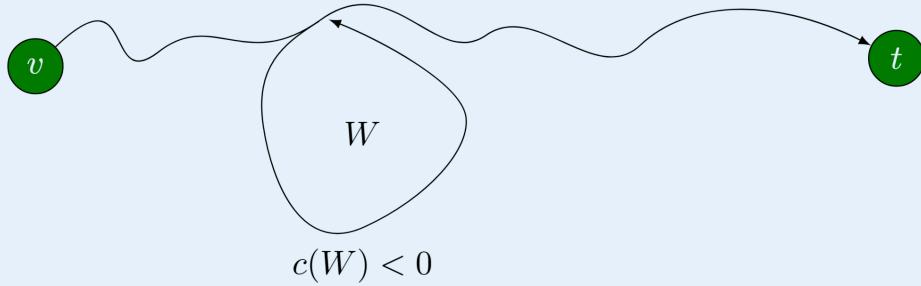
ⓘ Negative Kreise erkennen

Lemma: Wenn $OPT(n, v) < OPT(n - 1, v)$ für einen Knoten v , dann enthält (ein beliebiger) kürzester $v - t$ Kantenzug K mit maximal n Kanten einen Kreis W .

Außerdem hat W negative Kosten.

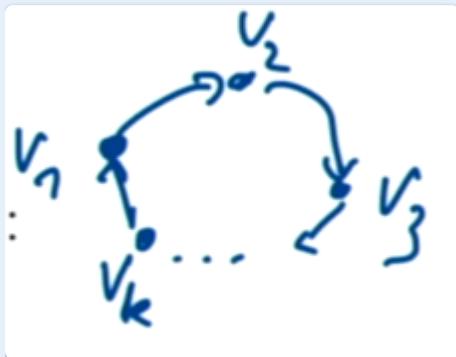
✓ Beweis: durch Widerspruch

- Da $OPT(n, v) < OPT(n - 1, v)$, wissen wir, dass K genau n Kanten hat.
- Mit Hilfe des Schubfachprinzip kann man zeigen, dass K einen gerichteten Kreis W enthalten muss.
- Löschen von W ergibt einen $v - t$ Kantenzug mit $< n$ Kanten $\rightarrow W$ hat negative Kosten.



ⓘ Umgekehrt: Wenn es einen Kreis gibt, dann muss es eine Verbesserung geben

Lemma: Falls G einen negativen Kreis enthält, der von dem aus t erreicht werden kann, dann gibt es eine Kante (v, u) , sodass $OPT(n - 1, v) > c_{vu} + OPT(n - 1, u)$ (und somit $OPT(n, v) < OPT(n - 1, v)$).



✓ Beweis

- Sei C ein beliebiger Kreis in G auf den Knoten (v_1, \dots, v_k) , dann gilt:

$$m = \sum_{i=1}^k [OPT(n - 1, v_i) - OPT(n - 1, v_{i+1 \bmod k})] = 0.$$

- Falls nun C ein negativer Kreis ist, ergibt sich $\sum_{i=1}^k c_{v_i v_{i+1 \bmod k}} < 0 = m$.
- D.h. es muss mindestens ein i existieren mit

$$c_{v_i v_{i+1 \bmod k}} < OPT(n-1, v_i) - OPT(n-1, v_{i+1 \bmod k})$$

und somit gilt für die Kante $(v_i, v_{i+1 \bmod k})$, dass

$$c_{v_i v_{i+1 \bmod k}} + OPT(n-1, v_{i+1 \bmod k}) < OPT(n-1, v_i).$$

Folgerung aus den Lemmas:

Aus den beiden vorherigen Lemmas folgt nun:

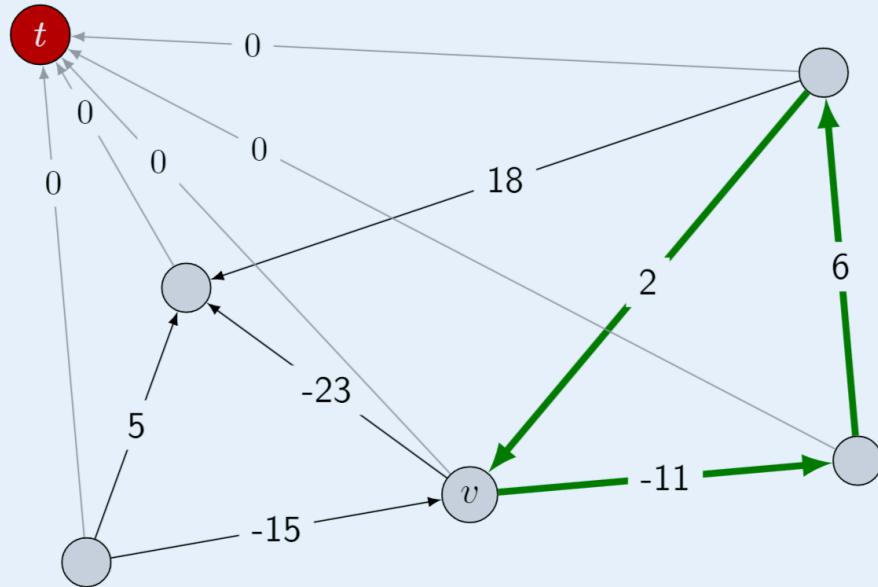
Theorem: G enthält einen negativen Kreis von dem aus t erreicht werden kann genau dann wenn ein Knoten v mit $OPT(n, v) < OPT(n-1, v)$ existiert.

① Negative Kreise erkennen

Theorem: Man kann in Zeit $O(nm)$ entscheiden, ob ein Graph einen negativen Kreis hat und diesen im positiven Fall auch ausgeben.

✓ Beweis

- Gib einen neuen Knoten t' hinzu und verbinde alle Knoten mit t' mit einer Kante mit den Kosten 0.
- Überprüfe, ob $OPT(n, v) = OPT(n-1, v)$ für alle Knoten v .
 - Wenn ja, dann gibt es keine negativen Kreise.
 - Wenn nein, dann extrahiere den Kreis aus dem kürzesten Kantenzug von v zu t' mit maximal n Kanten.



Kürzeste Pfade: Praktische Verbesserungen

Praktische Verbesserungen:

- Verwalte nur ein Array $M[v] = \text{kürzester } v - t \text{ Pfad}$, den wir bisher gefunden haben.
- Brich den Algorithmus ab, sobald sich nach einer vollen Iteration kein Eintrag in M mehr geändert hat.

ⓘ Theorem

Beim Ablauf des Algorithmus ist $M[v]$ die Länge eines $v - t$ Pfades und nach i Runden von Updates ist der Wert $M[v]$ nicht größer als die Länge eines kürzesten $v - t$ Pfades, der $\leq i$ Kanten benutzt.

Gesamte Auswirkung:

- **Speicher:** $O(m + n)$.
- **Laufzeit:** $O(mn)$ Worst-Case, aber wesentlich schneller in der Praxis.
- Dijkstra hatte Zeit $O((n + m) \cdot \log(n))$, aber dafür keine Behandlung von negativen Kanten

Effiziente Implementierung von Bellman-Ford

Push-Based-Shortest-Path-DP(G, s, t):

foreach Knoten $v \in V$

$M[v] \leftarrow \infty$

 Nachfolger[v] $\leftarrow \emptyset$

$M[t] = 0$

for $i \leftarrow 1$ bis $n - 1$:

foreach Kante $(v, w) \in E$

if $M[v] > M[w] + c_{vw}$

$M[v] \leftarrow M[w] + c_{vw}$

 Nachfolger[v] $\leftarrow w$

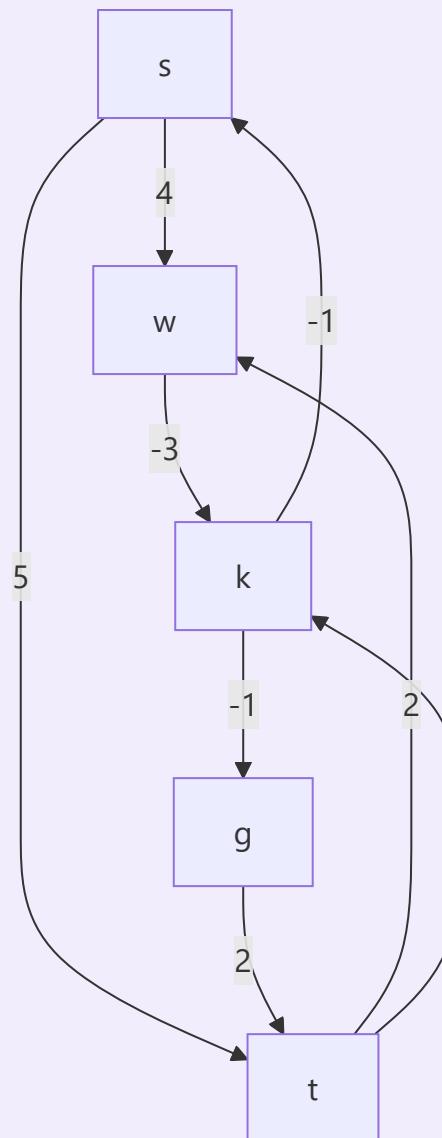
if kein $M[w]$ Wert ändert sich in Iteration i , stop.

return $M[s]$

:≡ Beispiel für den verbesserten Algorithmus

Beispiel siehe [AD_12_DynamischeProgrammierung, p.60](#)

:≡ Weiteres Beispiel aus der ue7-A4-(b)



Man schaut, mit wie viel Gewicht man zu t kommt, je nach dem wie viele Schritte man gehen kann und nimmt, falls man einen anderen Wert hat den kleineren.

	0	1	2	3	4
t	0	0	0	0	0
w	∞	∞	∞	-2	-2
k	∞	∞	1	1	1
s	∞	5	5	5	2
g	∞	2	2	2	2

Ergebnis: $s \rightarrow w \rightarrow k \rightarrow g \rightarrow t$

Finden eines kürzesten Pfades: Korrektheit

Den kürzesten Pfad finden: Verwalte den Nachfolger für jeden Knoten.

⌚ Definition

Ein Knoten w ist der Nachfolger eines Knotens v (an einer beliebigen Stelle im Algorithmus), falls $M[v]$ zuletzt auf $c_{vw} + M[w]$ geändert wurde.

Beobachtung: Falls w momentan der Nachfolger von v ist, dann gilt: $M[v] \geq c_{vw} + M[w]$.
 → wenn der Nachfolger gesetzt wird gilt $M[v] = c_{vw} + M[w]$; später kann sich $M[w]$ aber noch verringern.

ⓘ Lemma

Falls der Nachfolgergraph einen Kreis enthält (an einer beliebigen Stelle im Algorithmus), dann ist der Kreis negativ.

✓ Beweis

- Seien v_1, \dots, v_k die Knoten auf einem Kreis C im Nachfolgergraph und sei (v_k, v_1) die letzte Kante in C , die vom Algorithmus hinzugefügt wurde.
- Dann gilt $M[v_i] \geq c_{v_iv_{i+1}} + M[v_{i+1}]$ für alle i mit $1 \leq i < k$ (unmittelbar bevor die Kante (v_k, v_1) gesetzt wird) $M[v_k] > c_{v_kv_1} + M[v_1]$.
- Nach Aufsummieren aller Ungleichungen verschwinden die $M[v_i]$ -Terme und wir erhalten: $0 > \sum_{i=1}^{k-1} c_{v_iv_{i+1}} + c_{v_kv_1}$, also ist C ein negativer Kreis.

Aus dem vorherigen Lemma folgt, dass falls G keine negativen Kreise enthält, der Nachfolgergraph kreisfrei ist.

Nach Beendigung des Algorithmus auf einem Graphen G ohne negative Kreise, folgt nun:

- Jeder Knoten v von dem aus t erreichbar ist hat genau einen Nachfolger w und $OPT(n - 1, v) = c_{vw} + OPT(n - 1, w)$.
- Also enthält der Nachfolgergraph für jeden solchen Knoten genau einen Pfad nach t und dieser Pfad ist ein kürzester Pfad.

Dynamische Programmierung Zusammenfassung

Vorgehen beim Entwurf von Dynamischen Programmen

- Versteh und charakterisiere die Struktur des Problems und seiner optimalen Lösungen anhand von überlappenden Teilproblemen und optimalen Teillösungen
- Definiere rekursiv den Wert einer optimalen Lösung
- Berechne und speichere die Werte der optimalen (Teil-)Lösungen in einer Tabelle
- Konstruiere die optimale Lösung durch Rückverfolgung der Optimierungsentscheidungen des Algorithmus

Techniken der Dynamischen Programmierung

- Binäre Auswahl: z.B. gewichtetes Interval Scheduling
- Mehrfachauswahl: z.B. segmented least squares, kürzeste Pfade
- Einführen zusätzlicher Variablen: z.B. Knapsack

Top-down vs. bottom-up: rekursive vs. iterative Berechnung

13. Approximation

Einleitendes Beispiel Gütegarantie

Annahmen:

- Sei A ein Algorithmus, der für jede Instanz x eines Problems X eine gültige Lösung mit Lösungswert $c_A(x) > 0$ liefert.
- Sei $c_{opt}(x) > 0$ der Wert einer optimalen Lösung.

⌚ Definition für Minimierungsprobleme

Falls es ein $\varepsilon \geq 1$ gibt, sodass für alle Instanzen x von X gilt:

$$\frac{c_A(x)}{c_{opt}(x)} \leq \varepsilon$$

dann ist A ein ε -Approximationsalgorithmus und der Wert ε heißt Gütegarantie.

⌚ Definition für Maximierungsprobleme

Falls es ein ε mit $0 < \varepsilon \leq 1$ gibt, sodass für alle Instanzen x von X gilt:

$$\frac{c_A(x)}{c_{opt}(x)} \geq \varepsilon$$

dann ist A ein ε -Approximationsalgorithmus und der Wert ε heißt Gütegarantie.

Es folgt:

- für Minimierungsprobleme: $\varepsilon \geq 1$
- für Maximierungsprobleme: $0 < \varepsilon \leq 1$
- $\varepsilon = 1 \iff A$ ist ein exakter Algorithmus

Approximationsalgorithmus für Vertex Cover

💡 Definition - Vertex Cover

Ein **Vertex Cover** eines Graphen $G = (V, E)$ ist eine Menge $C \subseteq V$, so dass jede Kante des Graphen zu mindestens einem Knoten aus C inzident ist.

Minimales Vertex Cover: Finde für einen gegebenen Graphen ein Vertex Cover mit minimaler Größe $|C|$.

Aufwand: Ist NP-schwer, d.h. kann i.A. vermutlich nicht in polynomieller Zeit gelöst werden.

Minimales Vertex Cover: Approximationsalgorithmus

2-Approximationsalgorithmus: Findet ein Vertex Cover in einem Graphen $G = (V, E)$, welches höchstens doppelt so groß wie das Optimum ist.

Approx-Vertex-Cover(G):

Approx-Vertex-Cover(G):

$C \leftarrow \emptyset$

while $E \neq \emptyset$

 Wähle eine beliebige Kante $(u, v) \in E$

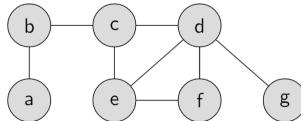
$C \leftarrow C \cup \{u, v\}$

 Entferne aus E alle Kanten, die inzident zu u oder v sind

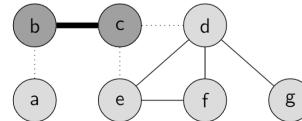
return C

☰ Beispiel

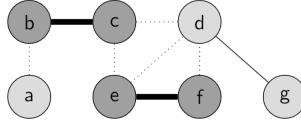
Beispielhafter Ablauf des Algorithmus, Vergleich mit optimaler Lösung.



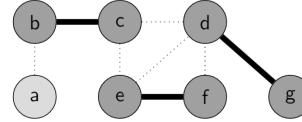
(a) Ausgangssituation



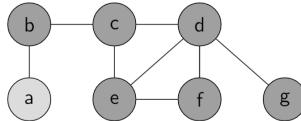
(b) 1. Kante auswählen



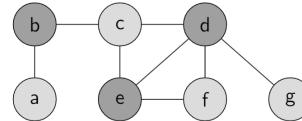
(c) 2. Kante auswählen



(d) 3. Kante auswählen



(e) Ergebnis



(f) Optimale Lösung

Hier können wir jetzt uns die untere Schranke ausrechnen:

$$\frac{6}{3} = 2$$

Minimales Vertex Cover: Gütegarantie

Theorem: Approx-Vertex-Cover ist ein polynomieller Algorithmus mit einer Gütegarantie von 2.

✓ Laufzeit

Ist polynomiell.

- In der Schleife werden nacheinander Kanten ausgewählt, zwei Knoten zu C hinzugefügt und dann alle inzidenten Kanten gelöscht.
- Mit Adjazenzlisten kann dieser Algorithmus mit Laufzeit $O(n + m)$ implementiert werden.
- Wir schreiben $n = |V|$ und $m = |E|$.

Gütegarantie:

- Sei M die Kantenmenge, die vom Algorithmus ausgewählt wird; M ist ein Matching (Matching... Teilmenge von Kanten im Graphen, sodass keine 2 Kanten in einem Knoten inzident sind, Teilmengen von den Kanten, sodass keine 2 Kanten den selben Endpunkt haben).

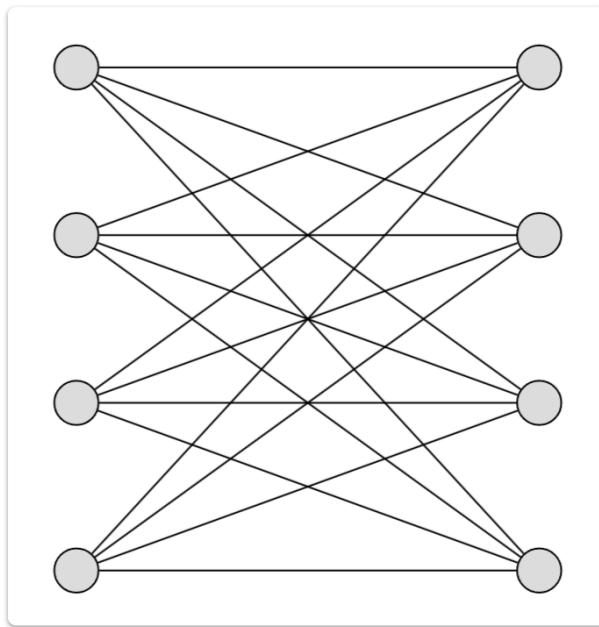
- In einem kleinsten Vertex Cover C^* muss gelten: Für jede Kante $e \in M$ existiert ein Knoten $v \in C^*$, der inzident zu e ist.
- Daher muss C^* zumindest einen der Endpunkte jeder Kante $e \in M$ enthalten.
- Es folgt, dass $|C^*| \geq |M|$.
- Da Approx-Vertex-Cover (kurz AVC) ein Vertex Cover der Größe $2|M|$ findet, gilt für alle Instanzen x :

$$c_{AVC}(x) = 2|M| \leq 2 \cdot c_{opt}(x)$$

Approx-Vertex-Cover: Für einen bipartiten vollständigen Graphen (kurz $K_{n,n}$) ist die Schranke sogar scharf.

Hinweis: Ein einfacher Graph heißt **bipartit** oder paar, wenn sich seine Knoten in zwei disjunkte Teilmengen A und B aufteilen lassen, sodass zwischen den Knoten innerhalb einer jeden Teilmenge keine Kanten verlaufen.

Beispiel: Approx-Vertex-Cover würde alle Knoten auswählen. Die optimale Lösung besteht aus den Knoten einer Seite.



Alternativer Algorithmus

Alternativer Algorithmus: Wählt immer einen Knoten mit aktuell maximalem Grad.

Approx-Vertex-Cover2(G):

Approx-Vertex-Cover2(G):

$C \leftarrow \emptyset$

while $E \neq \emptyset$

 Wähle einen Knoten u mit maximalem Grad im aktuellen Graphen

$C \leftarrow C \cup \{u\}$

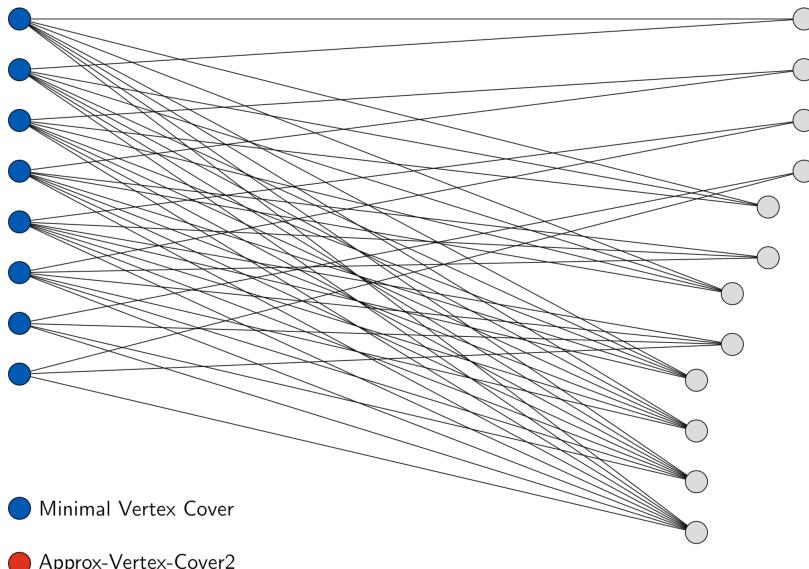
 Entferne aus E alle Kanten, die inzident zu u sind

return C

Gütegarantie: Man kann zeigen, dass dieser Algorithmus eine logarithmische Gütegarantie hat. Die scheinbar intelligenteren Auswahl führt hier nicht zu einer Verbesserung!

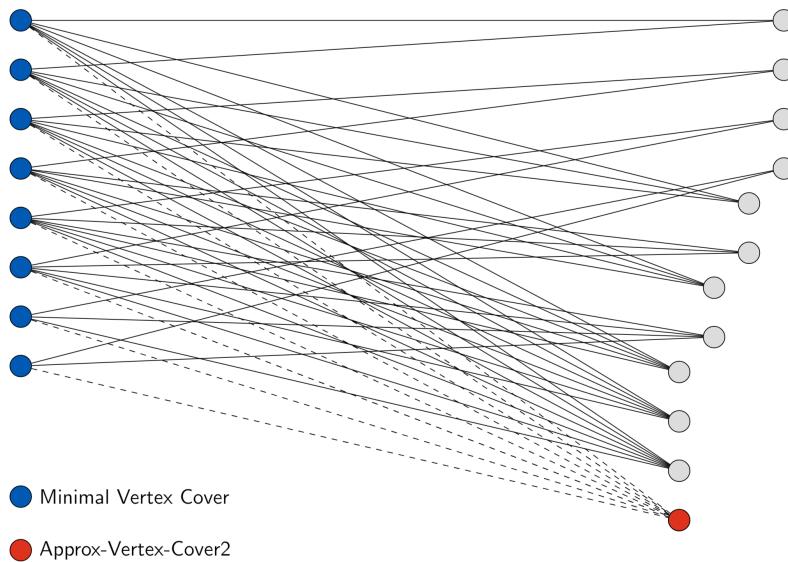
☰ Beispiel

Schlechtes Beispiel:

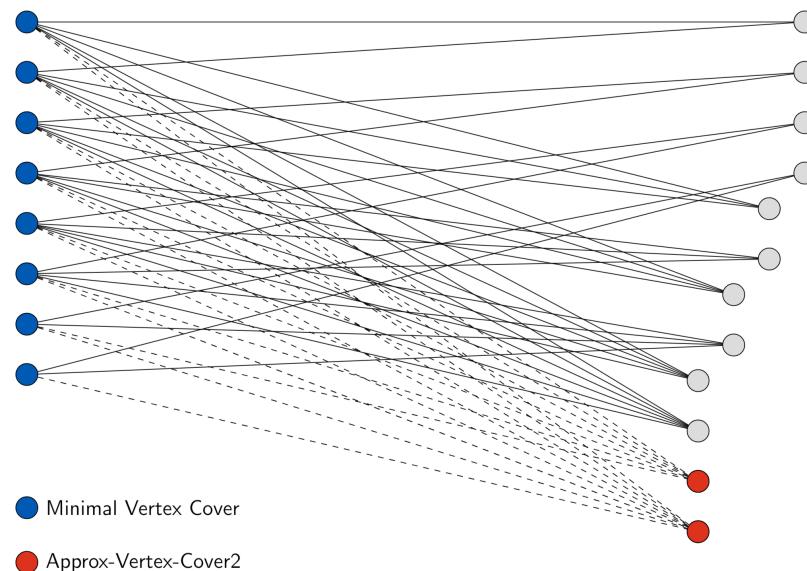


Hier wäre ja das minimale Vertex Cover, wenn man einfach die 8 auf der linken Seite nimmt, da diese alle abdecken würden. Allerdings nimmt unser Greedy Algorithmus ja den mit dem maximalen Grad.

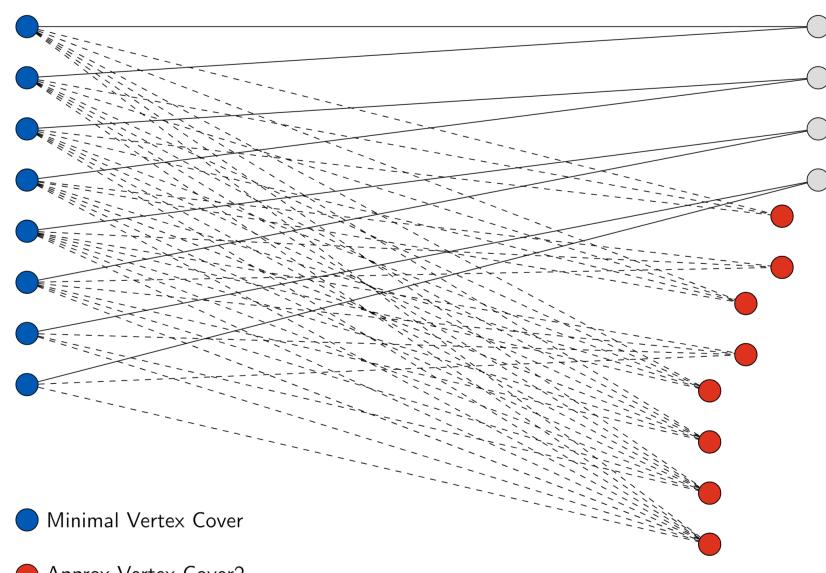
Schlechtes Beispiel:



Schlechtes Beispiel:

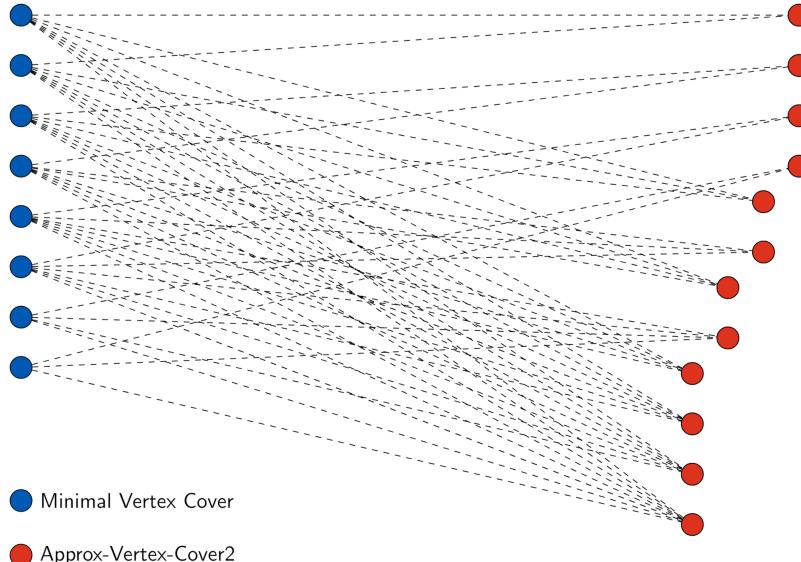


Schlechtes Beispiel:



Hier hat man dann auf der rechten Seite lauter Knoten mit Grad 2, während die linke Seite nur noch Knotengrad 1 hat, daher werden die letzten Knoten auch von rechts gewählt.

Schlechtes Beispiel:



Konstruktionsschema für schlechtes Beispiel:

- Wähle n Knoten auf der linken Seite.
- Für $i = 2, \dots, n$ geben wir jeweils eine Menge R_i mit $\lfloor \frac{n}{i} \rfloor$ Knoten auf der rechten Seite hinzu.
- Jeder Knoten aus der Menge R_i hat einen Grad i und ist jeweils mit i Knoten auf der linken Seite verbunden.

Beispiel aus vorheriger Folie:

- Links gibt es 8 Knoten.
- Rechts gibt es 12 Knoten:
 - R_2 (4 Knoten mit Grad 2)
 - R_3 (2 Knoten mit Grad 3)
 - R_4 (2 Knoten mit Grad 4)
 - R_5 (1 Knoten mit Grad 5)
 - R_6 (1 Knoten mit Grad 6)
 - R_7 (1 Knoten mit Grad 7)
 - R_8 (1 Knoten mit Grad 8)

Gütegarantie: Für n Knoten auf der linken Seite.

- Approx-Vertex-Cover2 wählt alle Knoten auf der rechten Seite, d.h.

$$\sum_{i=2}^n |R_i| = \sum_{i=2}^n \lfloor \frac{n}{i} \rfloor \geq \sum_{i=2}^n (\frac{n}{i} - 1) \geq n \sum_{i=1}^n \frac{1}{i} - 2n = n(H_n - 2)$$

Knoten. Dabei ist H_n die n -te harmonische Zahl.

- Es gilt $H_n = \ln n + \Theta(1)$.
- Minimales Vertex Cover umfasst n Knoten. Gütegarantie ist daher:

$$\frac{n(H_n - 2)}{n} = \Omega(\log n)$$

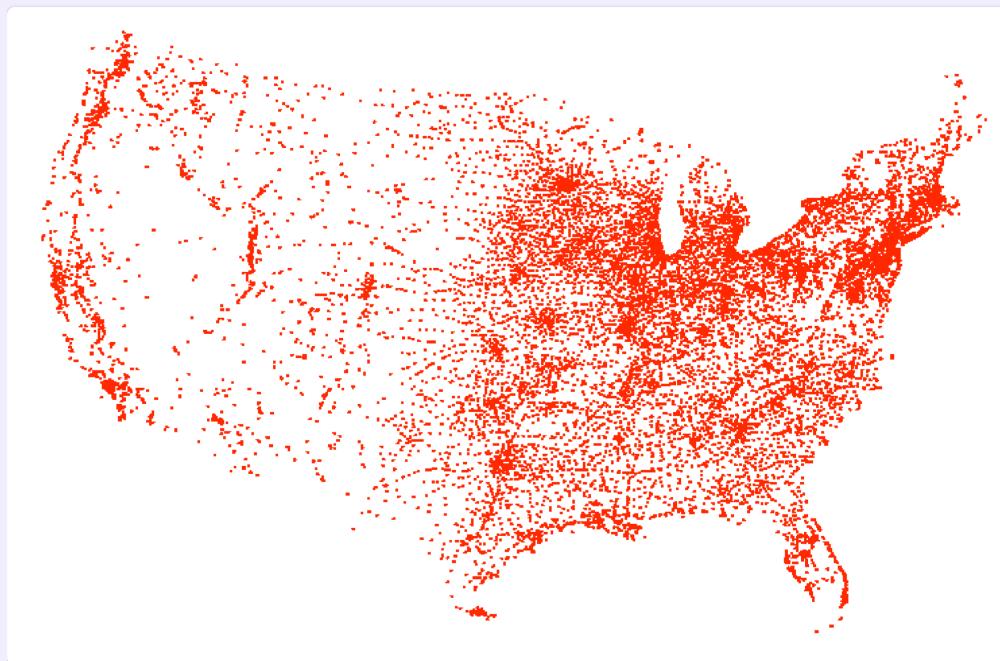
Spanning-Tree-Heuristik (ST) für das symmetrische TSP

⌚ Traveling Salesmen Problem - Wiederholung

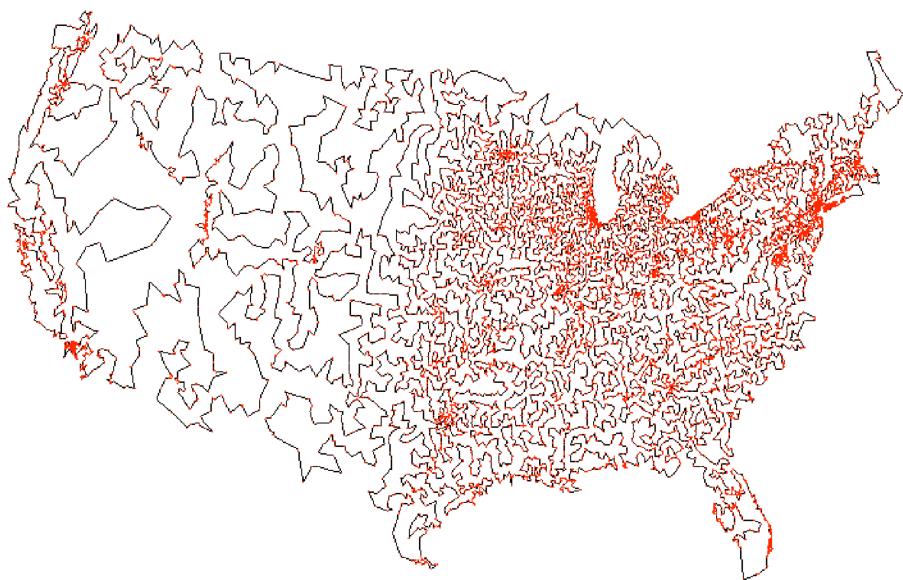
- Man sucht eine Reihenfolge (Tour) für den Besuch mehrerer Orte, sodass die gesamte Reisestrecke eines Handlungsreisenden möglichst kurz ist (Minimierungsproblem).
- Jeder Ort wird genau einmal besucht und nach dem letzten Ort wird zum ersten zurückgekehrt.

☰ Beispiel

Beispiel: Die 13509 Städte der USA mit mehr als 500 Einwohner.



Beispiel: Die 13509 Städte der USA mit mehr als 500 Einwohner: Optimale Tour



Traveling Salesperson Problem (TSP)

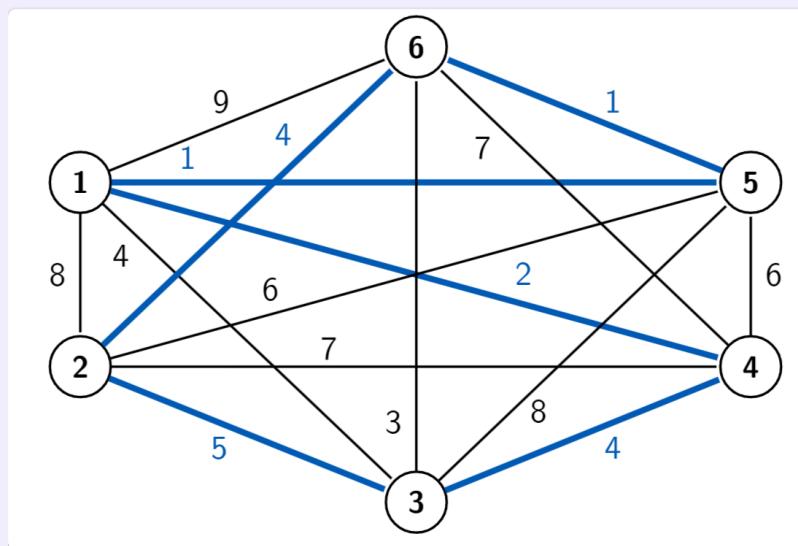
Darstellung: Die kürzesten Wege zwischen allen Knotenpaaren können durch einen gewichteten vollständigen Graphen repräsentiert werden.

Vollständiger Graph:

- Ist ein schlichter Graph, in dem jeder Knoten mit jedem anderen Knoten durch eine Kante verbunden ist.
- Ein vollständiger Graph mit n Knoten wird als K_n bezeichnet.

☰ Beispiel

6 Orte, minimale Tour der Länge 17.



Symmetrisches TSP

⌚ Symmetrisches TSP

- Gegeben ist ein ungerichteter vollständiger Graph $G = (V, E)$ mit Distanzmatrix c mit $c_{ii} = +\infty$ und $c_{ij} \geq 0$.
- Für alle Knotenpaare (i, j) sind die Distanzen in beide Richtungen identisch, d.h. es gilt $c_{ij} = c_{ji}$.
- Jede Tour hat dieselbe Länge in beide Richtungen.

Hinweis: Das TSP ist NP-schwer (Beweis durch Reduktion von HAM-CYCLE).

Minimum Spanning Tree Heuristik (MST) für das sym. TSP

Spanning-Tree-Heuristik: Tour wird aus einem Minimum Spanning Tree (MST) für den gegebenen Graphen G abgeleitet.

📋 Vorgehen

1. Bestimme einen MST (V, B_1) von G .
2. Verdopple alle Kanten in B_1 , das ergibt Graph (V, B_2) . Dieser Graph ist eulersch.
3. Bestimme eine **Euler-Tour** T im Graphen (V, B_2) . Gib dieser Tour eine Orientierung, wähle einen Knoten $p \in V$, markiere p als besucht, setze $T' \leftarrow (p)$, $i \leftarrow 0$.
4. Sind alle Knoten markiert, setze T' als Ergebnis-Tour.
5. Laufe von p entlang der Orientierung von T bis ein unmarkierter Knoten q erreicht ist. Setze $T' \leftarrow T' \cup \{(p, q)\}$, markiere q , setze $p \leftarrow q$ und gehe zu (4).

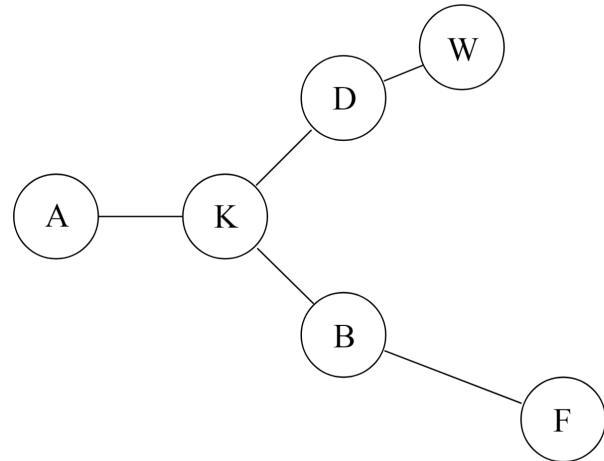
⌚ Euler-Tour

Ein Graph hat eine Euler-Tour (ein Rundgang, der jede Kante genau einmal benutzt und zum Startknoten zurückkehrt) genau dann, wenn alle seine Knoten einen geraden Grad haben. Durch das Verdoppeln der Kanten im MST stellen wir sicher, dass jeder Knoten im resultierenden Graphen einen geraden Grad hat.

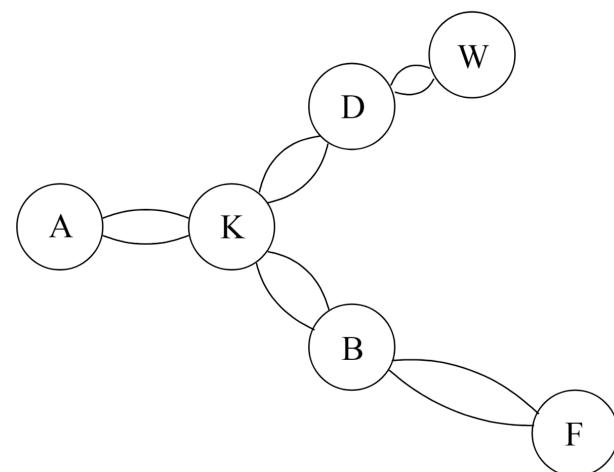
☰ Beispiel

Ausgangspunkt: Vollständiger Graph mit 6 Knoten.

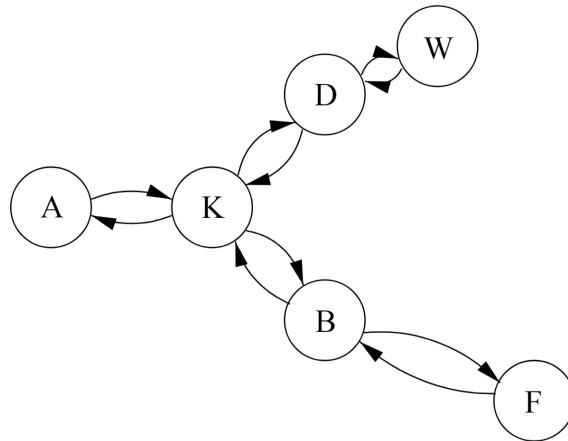
Schritt 1: MST (V, B_1) sieht beispielsweise folgendermaßen aus:



Schritt 2: Alle Kanten im MST werden verdoppelt (V, B_2) .



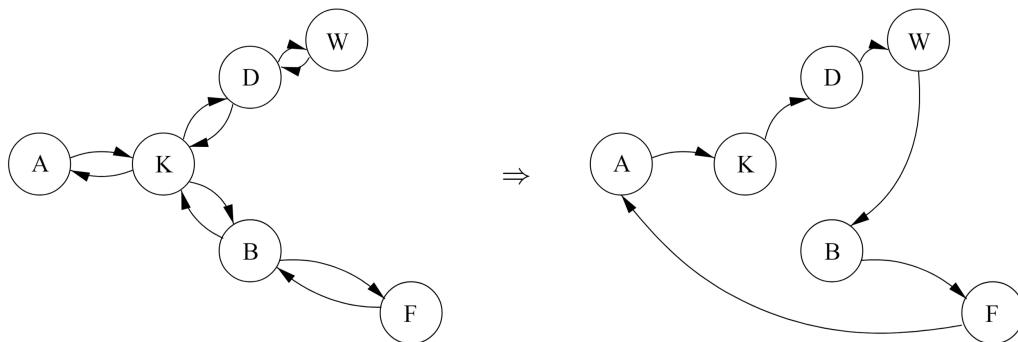
Schritt 3: Bestimme Eulertour F , die jede Kante genau einmal enthält.



Eulertour: $F = (A, K, D, W, D, K, B, F, B, K, A)$

Schritt 4 und 5: Eulertour \Rightarrow TSP-Tour

- A ist der Startknoten.
- In der Tour wird jeder Knoten nur einmal besucht.
- Wird ein Knoten besucht, dann wird er markiert.
- Der nächste TSP-Tour-Knoten ist immer der nächste unmarkierte Knoten auf der Eulertour.



ⓘ Theorem

Eine Euler-Tour existiert in einem ungerichteten Graphen genau dann, wenn er zusammenhängend ist und jeder Knoten einen geraden Grad hat.

Hinweis: Durch die Verdopplung der Kanten im MST hat jeder Knoten geraden Grad. Eine Euler-Tour existiert somit immer.

Spanning-Tree-Heuristik (ST): Gütegarantie

Laufzeit: Der erste Schritt (MST finden) kann beispielsweise mit Prims Algorithmus in Zeit $O(n^2)$ gelöst werden. Die restlichen Schritte sind nicht aufwendiger und daher läuft die gesamte Heuristik in Zeit $O(n^2)$.

Algorithmus von Hierholzer (1873)

Hinweis: Findet eine Euler-Tour (falls vorhanden) in einem Graphen G in linearer Zeit.

Grundlegende Idee:

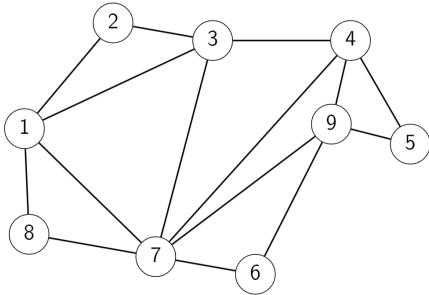
- Beginnend von einem Startknoten wird ein Pfad (nicht notwendigerweise einfach) mit einer noch nicht benutzten Kante fortgesetzt. Wenn alle Knoten geraden Grad haben, kehrt man wieder zum Ausgangsknoten zurück. Dieser geschlossene Pfad bildet einen **Zyklus** mit den Knoten $v_1, v_2, \dots, v_k, v_1$.
- Die Kanten $E(Z) = \{(v_1, v_2), (v_2, v_3), \dots, (v_{k-1}, v_k), (v_k, v_1)\}$ des gefundenen Zyklus Z werden aus dem Graphen entfernt. Im Restgraphen sind, wenn eine Euler-Tour vorhanden ist, wiederum alle Knotengrade gerade.
- Gibt es in einem Zyklus Z einen Knoten mit einem Grad größer 0, dann kann man von dort aus wiederum einen Zyklus bilden.

Vorgehen

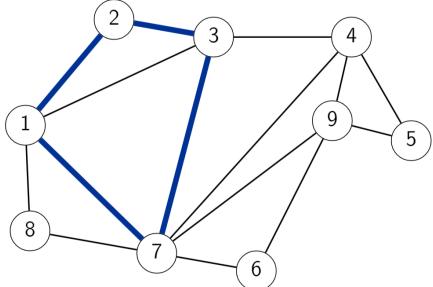
1. Wähle einen beliebigen Knoten v_0 des Graphen und konstruiere von v_0 ausgehend einen geschlossenen Pfad Z in G , der keine Kante in G zweimal durchläuft. Wir nennen Z einen Zyklus.
2. Wenn Z eine Euler-Tour ist (also alle Knoten beinhaltet), terminiere. Andernfalls:
3. Lösche nun alle Kanten aus G .
4. An einem Knoten v des Zyklus Z , der dessen Grad im aktuellen (Rest-)Graphen größer 0 ist, startet man nun einen neuen Zyklus Z' .
5. Füge Z' in den Zyklus Z ein, indem der Startpunkt von Z' beim ersten Auftreten in Z durch alle Knoten von Z' in der durchlaufenden Reihenfolge ersetzt wird.
6. Fahre bei Schritt 2 fort.

Beispiel

Ausgangsgraph:

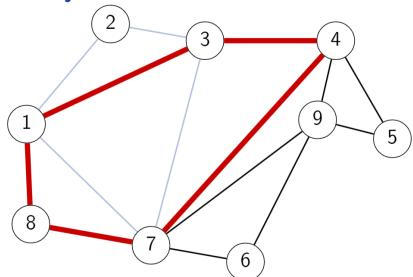


1. Zyklus:



$$Z = (1, 2, 3, 7, 1)$$

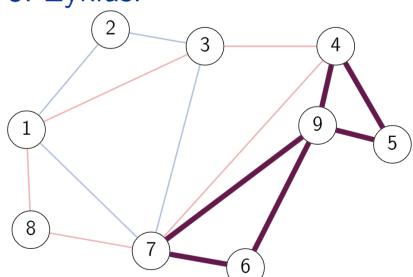
2. Zyklus:



$$Z' = (1, 3, 4, 7, 8, 1)$$

$$Z = (1, 3, 4, 7, 8, 1, 2, 3, 7, 1)$$

3. Zyklus:

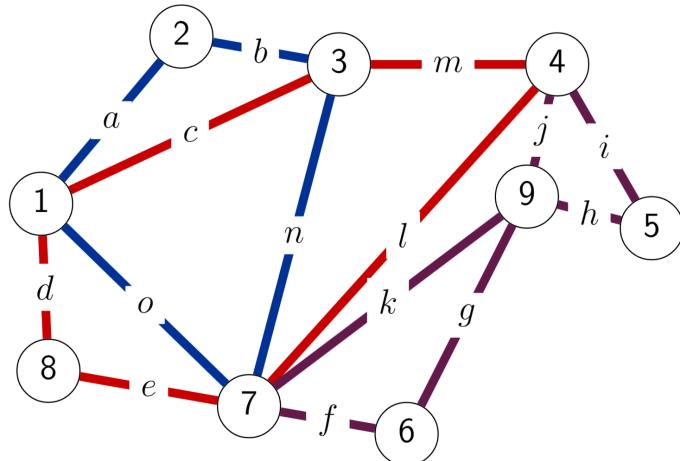


$$Z' = (7, 6, 9, 5, 4, 9, 7)$$

$$Z = (1, 3, 4, 7, 6, 9, 5, 4, 9, 7, 8, 1, 2, 3, 7, 1)$$

Mögliche Eulertour: Weiteres Beispiel

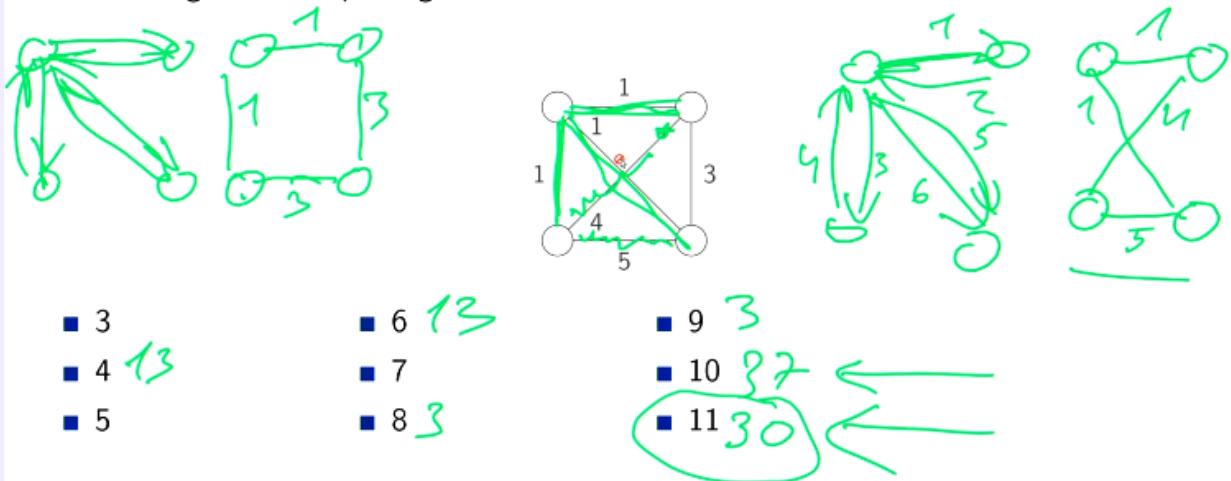
- Kantenfolge: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o
- Knotenfolge: 1, 2, 3, 1, 8, 7, 6, 9, 5, 4, 9, 7, 4, 3, 7, 1



Laufzeit: Liegt in $O(n + m)$.

☰ Beispiel

Frage 3: Welche Länge hat die schlechteste Tour, die von der Spanning-Tree-Heuristik auf dem folgenden Graphen gefunden wird?



Ist das symmetrische TSP 2-approximierbar?

- **Theorem:** Angenommen $P \neq NP$. Dann gibt es keinen polynomiellen 2-Approximationsalgorithmus für das symmetrische TSP (Travelling Salesperson Problem).

✓ Beweis (durch Widerspruch)

1. **Annahme:** Es existiert ein polynomieller Approximationsalgorithmus A mit einer Gütegarantie 2 für das symmetrische TSP.

2. Sei $G = (V, E)$ ein Graph, für den wir bestimmen wollen, ob er einen Hamiltonkreis enthält (dies ist ein NP-vollständiges Problem).
3. Wir möchten nun Algorithmus A benutzen, um den Hamiltonkreis effizient zu finden.
4. Dazu wird Graph G in eine TSP-Instanz $G' = (V, E', c)$ transformiert.

- Transformation von G in eine TSP-Instanz $G' = (V, E', c)$:

- Sei (V, E') der vollständige Graph, d.h. $E' = \{(u, v) : u, v \in V, u \neq v\}$.

$$c_{u,v} = \begin{cases} 1 & \text{wenn } (u, v) \in E \\ 2 \cdot n + 1 & \text{sonst} \end{cases}$$

Die Kantenkosten $c_{u,v}$ sind definiert als:

1, wenn die Kante (u, v) in E (dem ursprünglichen Graphen) enthalten ist.

$2 \cdot n + 1$, wenn die Kante (u, v) nicht* in E enthalten ist. Hierbei ist n die Anzahl der Knoten in V .

- Graph G kann in polynomieller Zeit in G' transformiert werden.

- Fall 1: G enthält einen Hamiltonkreis H

- Wenn G einen Hamiltonkreis H enthält, dann werden jeder Kante von H die Kosten 1 zugewiesen.
- G' enthält somit eine Tour mit den Kosten n (da ein Hamiltonkreis n Kanten hat und jede Kante Kosten 1 hat).

- Fall 2: G enthält keinen Hamiltonkreis

- Wenn G keinen Hamiltonkreis enthält, dann besitzt jede Tour in G' mindestens eine Kante, die sich *nicht* in E befindet (da jede Tour, die nur Kanten aus E enthält, per Definition ein Hamiltonkreis in G wäre).
- Solch eine Tour hat aber Kosten von mindestens:
 - $(2n + 1)$ für die eine Kante, die nicht in E ist.
 - Plus $(n - 1)$ für die restlichen $n - 1$ Kanten (die mindestens Kosten 1 haben).
 - Gesamtkosten: $(2n + 1) + (n - 1) = 2n + n = 3n > 2n$.

- Daher sind die Kosten einer Tour in G' , die *kein* Hamiltonkreis in G ist, zumindest um den Faktor 3 größer als die einer Tour, die *einen* Hamiltonkreis in G ist.
 - (Begründung: Kosten einer Hamiltonkreis-Tour sind n . Kosten einer Nicht-Hamiltonkreis-Tour sind $3n$. Das Verhältnis ist $3n/n = 3$).
- Nun wenden wir Algorithmus A auf die TSP-Instanz G' an.
- Algorithmus A liefert garantiert eine Tour zurück, deren Kosten höchstens um den Faktor 2 über denen einer optimalen Tour liegen (dies ist die definierte Gütegarantie 2 von Algorithmus A).

- **Fall 1: Wenn G einen Hamiltonkreis enthält,** dann muss Algorithmus A ihn zurückliefern.
 - Begründung: Die optimale Tour in G' hätte Kosten n (den Hamiltonkreis). Da A eine Tour liefert, deren Kosten höchstens $2 \cdot n$ betragen, und die einzige andere Option (eine Tour ohne Hamiltonkreis in G) Kosten von $3n$ oder mehr hat, muss A den Hamiltonkreis finden.
- **Fall 2: Wenn G keinen Hamiltonkreis enthält,** dann liefert Algorithmus A eine Tour mit Kosten größer als $2 \cdot n$.
 - Begründung: Wenn G keinen Hamiltonkreis enthält, ist die optimale Tour in G' eine, die mindestens eine Kante mit hohen Kosten ($2n + 1$) enthält. Die Kosten einer solchen Tour sind mindestens $3n$. Da A eine Tour mit Kosten von *höchstens* $2 \times$ (Kosten der optimalen Tour) liefert, kann A keine Tour mit Kosten $\leq 2n$ finden, wenn die optimale Tour Kosten $\geq 3n$ hat.
- Daraus folgt: Algorithmus A kann benutzt werden, um einen Hamiltonkreis in polynomieller Zeit zu finden.
 - **Dies ist aber ein Widerspruch dazu, dass HAM-CYCLE NP-schwer ist.**
 - Dieser Widerspruch tritt nur auf, wenn $P = NP$ gilt.
 - Da wir von $P \neq NP$ ausgegangen sind, ist die ursprüngliche Annahme (Existenz eines 2-Approximationsalgorithmus) falsch.
 - Daher kann es keinen polynomiellen 2-Approximationsalgoritmus für das symmetrische TSP geben, *wenn* $P \neq NP$.

Spanning-Tree-Heuristik (ST): Gütegarantie

-
- **Theorem:** Ähnlich wie für den Fall der 2-Approximation kann für ein **allgemeines** $\varepsilon > 1$ gezeigt werden, dass es keinen polynomiellen ε -Approximationsalgoritmus für das symmetrische TSP gibt.
 - Das symmetrische TSP ist somit "**nicht approximierbar**" (im allgemeinen Fall, wenn $P \neq NP$ gilt).

✓ Beweis

- Der Beweis folgt dem gleichen Prinzip wie auf den vorherigen Folien (für die 2-Approximation).
- Der einzige Unterschied liegt in der Definition der Kantenkosten:
 - Nun gilt $c(u, v) = \varepsilon \cdot n + 1$, wenn $(u, v) \notin E$ (wobei ε ein beliebiger Wert größer 1 ist).

② Frage

- Wozu dann der ganze Aufwand, wenn eine beliebige Approximation (d.h. mit einem beliebigen ε) nicht effizient möglich ist?

Antwort:

- Unter einer **einfachen Voraussetzung** wird das Ergebnis besser (d.h. es gibt Fälle, in denen eine Approximation doch möglich oder effizienter ist).

Metrisches TSP

⌚ Metrisches TSP

- Ein TSP (Travelling Salesperson Problem) heißt **metrisch**, wenn für die Distanzmatrix C die **Dreiecksungleichung** gilt.
- Das bedeutet, für alle Knoten i, j, k muss die Bedingung erfüllt sein:
 - $c_{ik} \leq c_{ij} + c_{jk}$
 - (Intuitiv: Der direkte Weg von i nach k ist nie länger als der Weg von i über j nach k . Man kann keinen "Short-Cut" über einen dritten Punkt finden.)
- **Hinweis:** Insbesondere ist auch das **Euklidische TSP** metrisch. Hierbei entsprechen die Knoten Punkten in der euklidischen Ebene, und die Distanzen sind die euklidischen Distanzen.

ⓘ Theorem

- Das metrische TSP besitzt einen polynomiellen Approximationsalgorithmus mit einer **Gütegarantie von 2**.
- Das bedeutet, für eine gegebene TSP-Instanz x :
 - $\frac{c_{ST}(x)}{c_{opt}(x)} \leq 2$
 - Hierbei ist $c_{ST}(x)$ die Kosten der Tour, die von der Spanning-Tree-Heuristik gefunden wird, und $c_{opt}(x)$ sind die Kosten der optimalen Tour.
 - (Die von der ST-Heuristik gefundene Tour ist also maximal doppelt so lang wie die optimale Tour.)

Laufzeit:

- Die Spanning-Tree-Heuristik läuft in **polynomieller Zeit**.

- Der **aufwändigste Schritt** ist die Ermittlung des **MST** (Minimum Spanning Tree) im ersten Schritt.

✓ Beweis für Gütegarantie

Beweis für Gütegarantie: Es gilt

$$c_{\text{ST}}(x) \leq c_{B_2}(x) = 2c_B(x) \leq 2c_{\text{opt}}(x)$$

- Gilt wegen der Dreiecksungleichung.
- Gilt, da B_2 durch Verdopplung der Kanten aus B entsteht. Gilt, da ein MST die kürzeste Möglichkeit ist, in einem Graphen alle Knoten zu verbinden.

- Begründung der Ungleichungen:**

- $c_{\text{ST}}(x) \leq c_{B_2}(x)$: Gilt wegen der Dreiecksungleichung.
 - (Dies bezieht sich auf die Umwandlung eines "doppelt durchlaufenen" MST in eine Tour, wobei Kanten unter Umständen abgekürzt werden können, da die Dreiecksungleichung gilt.)
- $c_{B_2}(x) = 2c_B(x)$: Gilt, da B_2 durch Verdopplung der Kanten aus B entsteht.
 - (Hier ist B der minimale Spannbaum und B_2 ein Graph, in dem jede Kante des MST zweimal vorkommt, was einen Eulerkreis ermöglicht, der dann in eine Tour umgewandelt wird.)
- $c_B(x) \leq c_{\text{opt}}(x)$: Gilt, da ein MST die kürzeste Möglichkeit ist, in einem Graphen alle Knoten zu verbinden (ohne Zyklen).
 - (Jede TSP-Tour enthält einen Spannbaum. Der MST ist der "billigste" Spannbaum, daher sind seine Kosten kleiner oder gleich den Kosten jedes anderen Spannbaums, einschließlich des Spannbaums, der Teil einer optimalen TSP-Tour ist.)

Anschaulich siehe: [Beispiel dazu](#)

Lastverteilung (Load Balancing)

- **Eingabe:**

- m identische Maschinen
- n Jobs (Aufgaben)
- Jeder Job j hat eine Bearbeitungszeit t_j .
- Jeder Job j muss ununterbrochen auf einer Maschine ausgeführt werden.
- Eine Maschine kann nur einen Job auf einmal ausführen.

 **Definition: Last einer Maschine**

- Sei J_i die Teilmenge von Jobs, die Maschine i zugewiesen wurde.
- Die Last der Maschine i ist

$$L_i = \sum_{j \in J_i} t_j$$

. (Summe der Bearbeitungszeiten aller Jobs auf Maschine i).

 **Definition: Bearbeitungsdauer (Makespan)**

- Die Bearbeitungsdauer (oder Makespan) ist die maximale Last auf irgendeiner Maschine.
- $L = \max_{i=1, \dots, m} L_i$.

- **Ziel der Lastverteilung:**

- Teile jeden Job einer Maschine so zu, dass die Bearbeitungsdauer (Makespan) minimiert wird.

Lastverteilung: List-Scheduling

List-Scheduling-Algorithmus:

- Berücksichtigt n Jobs mit einer fixen Ordnung (d.h. die Jobs werden in einer bestimmten, vorher festgelegten Reihenfolge verarbeitet).
- **Greedy-Algorithmus:** Teile Job j einer Maschine mit der aktuell kleinsten Last zu (lokale optimale Entscheidung).
 - L_i : Aktuelle Last auf Maschine i .
 - J_i : Menge der Jobs, die Maschine i zugewiesen wurden.

List-Scheduling($m, n, t_1, t_2, \dots, t_n$):

```

List-Scheduling( $m, n, t_1, t_2, \dots, t_n$ ):
  for  $i \leftarrow 1$  bis  $m$ 
     $J_i \leftarrow \emptyset$ 
     $L_i \leftarrow 0$ 
  for  $j \leftarrow 1$  bis  $n$ 
     $i = \operatorname{argmin}_{k=1, \dots, m} L_k$ 
     $J_i \leftarrow J_i \cup \{j\}$ 
     $L_i \leftarrow L_i + t_j$ 
  return  $J_1, \dots, J_m$ 
```

■ Maschine i hat geringste Last

Laufzeit: $O(n \log m)$ mit priority Queue

☰ Beispiel

Frage 4: Gegeben sind drei Maschinen und die folgenden Jobs:

Job	j_1	j_2	j_3	j_4	j_5	j_6
Bearbeitungszeit	2	2	5	5	6	10



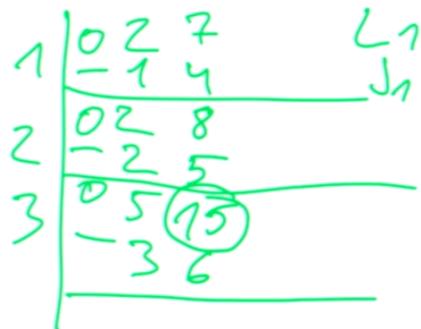
Für welche Reihenfolge an Jobs liefert der List-Scheduling Algorithmus das **schlechteste** Ergebnis?

- ✗ (A) $j_1, j_2, j_3, j_4, j_5, j_6$
- ✗ (B) $j_6, j_5, j_4, j_3, j_2, j_1$
- ✓ (C) $j_1, j_2, j_5, j_4, j_3, j_6$
- ✗ (D) $j_1, j_4, j_3, j_2, j_5, j_6$

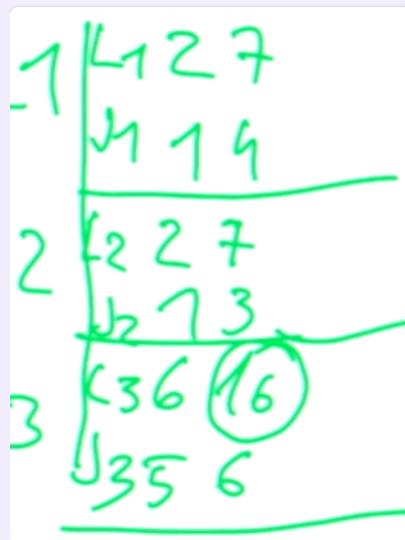
(A) Liefert uns das Ergebnis 15

Für welche Reihenfolge an Jobs liefert der List-Scheduling Algorithmus das schlechteste Ergebnis?

- (A) $j_1, j_2, j_3, j_4, j_5, j_6$ ←
- (B) $j_6, j_5, j_4, j_3, j_2, j_1$
- (C) $j_1, j_2, j_5, j_4, j_3, j_6$
- (D) $j_1, j_4, j_3, j_2, j_5, j_6$



(C) liefert uns das Ergebnis 16



Die anderen muss man sich genau so anschauen

Theorem:

- [Graham, 1966] List-Scheduling ist ein **2-Approximationsalgorithmus**.
- Dies ist die erste Worst-Case-Analyse eines Approximationsalgorithmus.
- Dazu muss die resultierende Lösung (vom List-Scheduling) mit der **optimalen Bearbeitungsdauer L^*** verglichen werden.

① Lemma 1

Für die optimale Dauer L^* gilt in jedem Fall: $L^* \geq \max_j t_j$.

✓ Beweis

Eine Maschine muss den aufwendigsten Job (j) mit der Bearbeitungszeit t_j verarbeiten. Da die optimale Dauer L^* die **maximale** Last auf einer Maschine ist,

muss sie mindestens so groß sein wie die Bearbeitungszeit des längsten einzelnen Jobs.

ⓘ Lemma 2

Für die optimale Dauer L^* gilt in jedem Fall:

$$L^* \geq \frac{1}{m} \sum_j t_j$$

✓ Beweis

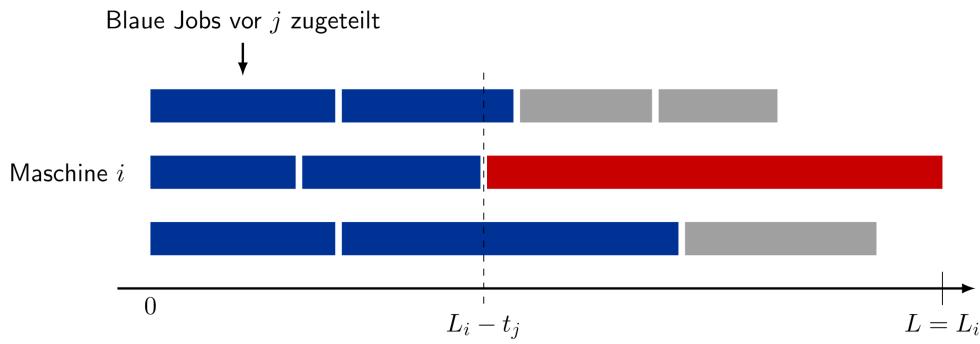
- Die gesamte Verarbeitungszeit aller Jobs ist $\sum_j t_j$.
- Eine der m Maschinen muss zumindest den $1/m$ -ten Teil der gesamten Arbeit machen (im Durchschnitt). Da L^* die maximale Last ist, muss sie mindestens so groß sein wie der Durchschnitt der Lasten über alle Maschinen.

Theorem:

List-Scheduling ist ein 2-Approximationsalgorithmus.

✓ Beweis

- Wir betrachten die Last L_i einer Maschine i , die einen Engpass darstellt (d.h. die Maschine mit der maximalen Last, also $L_i = L$).
- Sei j der letzte zugeteilte Job auf Maschine i .
- **Wenn Job j Maschine i zugewiesen wird, hat i die geringste Last.**
 - Die Last auf Maschine i vor der Zuteilung von Job j war $L_i - t_j$.
 - Da Maschine i zu diesem Zeitpunkt die geringste Last hatte, muss gelten: $L_i - t_j \leq L_k$ für alle $1 \leq k \leq m$.
 - Somit ist $L_i - t_j \leq \min_{k=1,\dots,m} L_k$.



- Die Abbildung zeigt, dass die Summe der "blauen" Jobs (also die Jobs auf allen Maschinen vor der Zuteilung des letzten Jobs j auf Maschine i) mindestens so groß ist wie $m \cdot (L_i - t_j)$, da jede Maschine zu diesem Zeitpunkt eine Last von mindestens $L_i - t_j$ hatte.
- Wir betrachten die Last L_i einer Maschine i , die einen Engpass darstellt (d.h., L_i ist die maximale Last L).
- Sei j der letzte zugeteilte Job auf Maschine i .
- Wenn Job j Maschine i zugewiesen wird, hatte i die geringste Last.
 - Die Last auf Maschine i vor der Zuteilung von Job j war $L_i - t_j$.
 - Es gilt: $L_i - t_j \leq L_k$ für alle Maschinen k (da i die geringste Last hatte).
- Wir summieren alle Ungleichungen über alle Maschinen k und dividieren durch m :

$$\begin{aligned} L_i - t_j &\leq \frac{1}{m} \sum_k L_k \\ &= \frac{1}{m} \sum_j t_j \\ &\leq L^* \end{aligned}$$

Nun ist $L_i = \underbrace{(L_i - t_j)}_{\leq L^*} + \underbrace{t_j}_{\leq L^*} \leq 2L^*$. \square

□ Lemma 2 □ Lemma 1

Center Selection

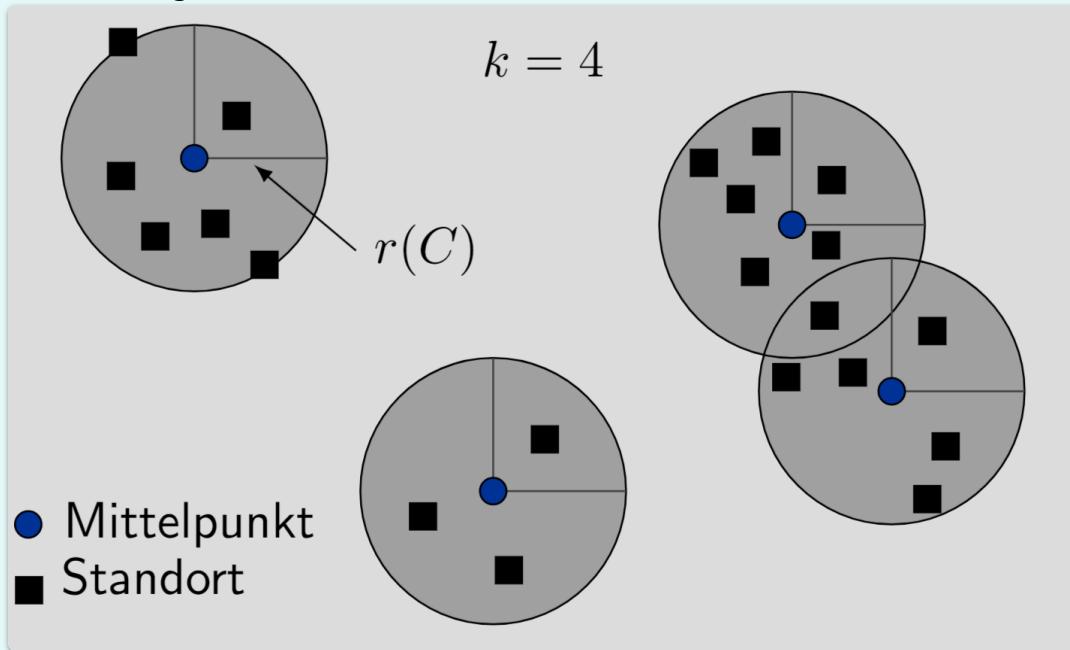
Definition

Eingabe:

- Eine Menge von n Standorten s_1, \dots, s_n .
- Eine ganze Zahl $k > 0$.

Problem

- Wähle k Mittelpunkte C , sodass die **maximale Distanz** von einem Standort zu einem nächsten Mittelpunkt minimiert wird.
- (Ziel ist es, k "Zentren" so zu platzieren, dass der am weitesten entfernte Standort so nah wie möglich an einem Zentrum ist.)



Notation:

- $\text{dist}(x, y)$: Distanz zwischen x und y .
- $\text{dist}(s_i, C)$: Distanz von Standort s_i zu seinem nächsten Mittelpunkt.
 - Formal: $\text{dist}(s_i, C) = \min_{c \in C} \text{dist}(s_i, c)$.
- $r(C)$: Der kleinste überdeckende Radius für eine gegebene Menge von Mittelpunkten C .
 - Formal: $r(C) = \max_{s_i \in S} \text{dist}(s_i, C)$.
 - (Dies ist die Metrik, die minimiert werden soll – der Radius des kleinsten Kreises, der alle Standorte überdeckt, wenn die Kreise um die Mittelpunkte liegen.)

Ziel:

- Finde eine Menge von Mittelpunkten C , die $r(C)$ minimiert, unter Berücksichtigung, dass die Kardinalität von C gleich k sein muss ($|C| = k$).

Eigenschaften der Distanzfunktion:

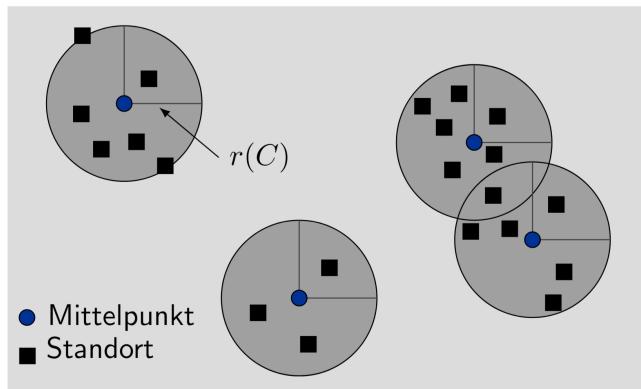
Eine gültige Distanzfunktion (Metrik) erfüllt folgende Eigenschaften:

- $\text{dist}(x, x) = 0$ (Identität: Die Distanz von einem Punkt zu sich selbst ist Null).
- $\text{dist}(x, y) = \text{dist}(y, x)$ (Symmetrie: Die Distanz von x nach y ist gleich der Distanz von y nach x).
- $\text{dist}(x, y) \leq \text{dist}(x, z) + \text{dist}(z, y)$ (Dreiecksungleichung: Der direkte Weg ist nie länger als ein Umweg über einen dritten Punkt).

:≡ Beispiel

Beispiel: Jeder Standort ist ein Punkt in der Ebene, ein Mittelpunkt kann jeder Punkt in der Ebene sein, $\text{dist}(x, y) =$ Euklidische Distanz.

Anmerkung: Es gibt unendlich viele potentielle Lösungen!



Greedy-Algorithmus

⚡ Falsche Ansatz

- **Greedy-Algorithmus (falscher Ansatz):**
 1. Wähle für den ersten Mittelpunkt einen "besten" Platz für einen Mittelpunkt.
 2. Danach füge iterativ Mittelpunkte hinzu, sodass der Abdeckradius immer am meisten reduziert wird.
- **Anmerkung:** Dieser Ansatz kann beliebig schlecht sein!



Mittelpunkt 1



Schlechte Wahl für
 $k = 2$ Mittelpunkte

● Mittelpunkt
■ Standort

- Das Bild zeigt ein Beispiel, wo ein initialer Mittelpunkt "intuitiv" in die Mitte einer Gruppe gesetzt wird. Wenn dann ein zweiter Mittelpunkt hinzugefügt werden soll, kann es sein, dass dieser nicht optimal platziert wird, weil die erste Wahl schon eine schlechte Ausgangsbasis geschaffen hat.

Richtiger Ansatz

Greedy-Algorithmus (korrekter Ansatz für Center Selection)

- Wähle den ersten Mittelpunkt **beliebig** aus der Menge aller Standorte.
- Wähle wiederholt als nächsten Mittelpunkt einen Standort, der am **weitesten von jedem existierenden Mittelpunkt entfernt ist**.
 - (Dies ist eine Max-Min-Strategie: Maximiere den minimalen Abstand zum nächsten Zentrum, um eine gute Verteilung zu erzielen.)

Greedy-Center-Selection($k, n, s_1, s_2, \dots, s_n$):

```

Greedy-Center-Selection( $k, n, s_1, s_2, \dots, s_n$ )
 $C = \{s_1\}$ 
wiederhole  $k - 1$  mal
  Wähle einen Standort  $s_i$  mit maximaler  $\text{dist}(s_i, C)$ 
  Füge  $s_i$  zu  $C$  hinzu
return  $C$ 
```

- Standort am weitesten von jedem Mittelpunkt:** Das bedeutet, der Standort s_t wird so gewählt, dass $\min_{c \in C} \text{dist}(s_t, c)$ maximiert wird.

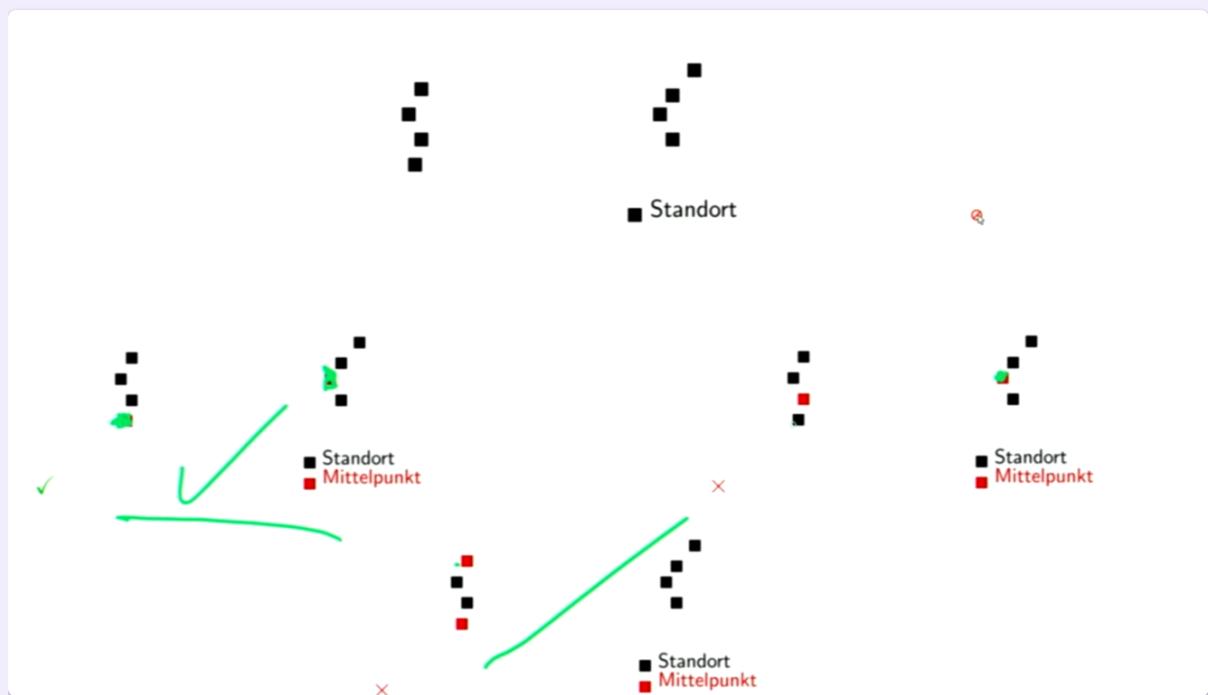
Beobachtung:

- Nach der Terminierung sind alle Mittelpunkte in C paarweise zumindest $r(C)$ voneinander entfernt.
 - (Dies ist eine wichtige Eigenschaft des Algorithmus: Die Mittelpunkte sind gut verteilt und nicht zu nah beieinander.)

✓ Beweis

- Durch Konstruktion des Algorithmus (indem immer der am weitesten entfernte Punkt gewählt wird, stellt man sicher, dass die neu hinzugefügten Mittelpunkte einen gewissen Mindestabstand zu den bestehenden haben).

☰ Beispiel



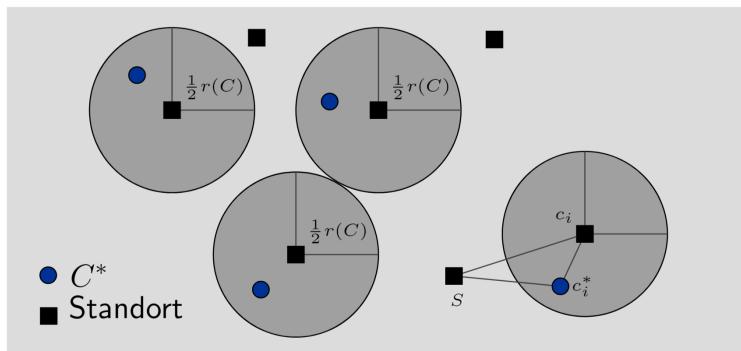
Nur das linke stimmt

Greedy-Algorithmus: Analyse (für das Center Selection Problem)

Theorem: Sei C^* eine optimale Menge von Mittelpunkten. Dann ist $r(C) \leq 2r(C^*)$.

Beweis: (durch Widerspruch) Angenommen $r(C^*) < \frac{1}{2}r(C)$.

- Für jeden Standort c_i in C betrachte den Radius $\frac{1}{2}r(C)$ um ihn herum.
- Nur ein $c_i^* \in C^*$ in jedem Radius; sei c_i der Standort mit c_i^* .
- Betrachte einen beliebigen Standort s und seinen nächsten Mittelpunkt c_i^* in C^* .
- $\text{dist}(s, C) \leq \text{dist}(s, c_i) \leq \text{dist}(s, c_i^*) + \text{dist}(c_i^*, c_i) \leq 2r(C^*)$.
- Daher $r(C) \leq 2r(C^*)$. \square
- Δ -Ungleichung $\leq r(C^*)$ da c_i^* der nächste Mittelpunkt ist.



Theorem: Optimalität der Menge von Mittelpunkten

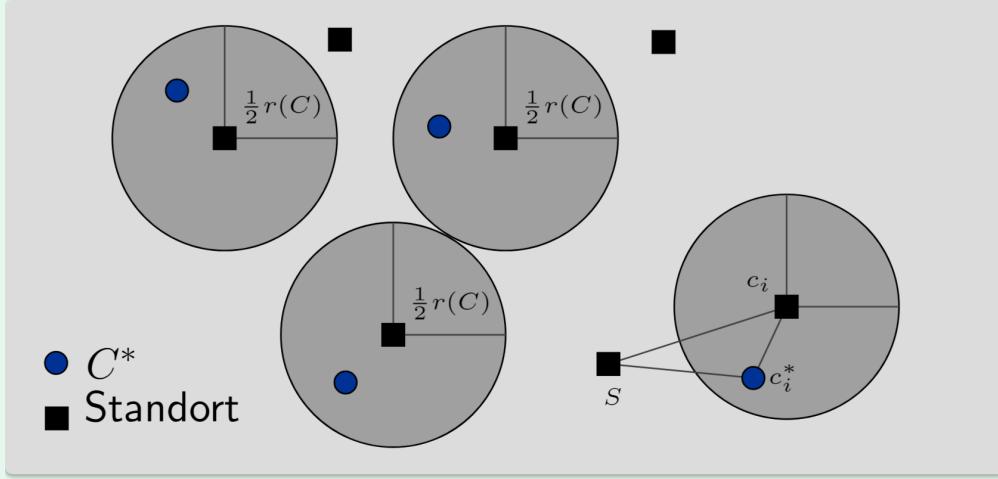
Sei C^* eine optimale Menge von Mittelpunkten. Dann gilt: $r(C) \leq 2r(C^*)$.

✓ Beweis (durch Widerspruch)

Angenommen, das Gegenteil ist der Fall, d.h., $r(C^*) < \frac{1}{2}r(C)$.

- Für jeden Standort c_i in C (der Menge der Standorte) wird der Radius $\frac{1}{2}r(C)$ um ihn herum betrachtet.
- Nur ein Mittelpunkt $c_j^* \in C^*$ befindet sich in jedem dieser Radien; sei c_i^* der Mittelpunkt, der zu c_i gehört (also der nächste Mittelpunkt zu c_i ist).
- Betrachte einen beliebigen Standort s und seinen nächsten Mittelpunkt c_i^* in C^* .
- Es gilt die Dreiecksungleichung: $\text{dist}(s, c_i) \leq \text{dist}(s, c_i^*) + \text{dist}(c_i^*, c_i) \leq 2r(C^*)$.
 - Erläuterung zur Ungleichung:
 - $\text{dist}(s, c_i)$ ist der Abstand von s zum nächsten Standort c_i in C .
 - $\text{dist}(s, c_i^*)$ ist der Abstand von s zu seinem nächsten Mittelpunkt c_i^* in C^* .
 - $\text{dist}(c_i^*, c_i)$ ist der Abstand zwischen dem Mittelpunkt c_i^* und dem Standort c_i .
 - Die Dreiecksungleichung besagt, dass der direkte Weg zwischen zwei Punkten ($\text{dist}(s, c_i)$) nie länger ist als der Umweg über einen dritten Punkt ($\text{dist}(s, c_i^*) + \text{dist}(c_i^*, c_i)$).

- Der Ausdruck $\leq 2r(C^*)$ kommt daher, dass sowohl $dist(s, c_i^*)$ als auch $dist(c_j^*, c_i)$ maximal $r(C^*)$ betragen können, da $r(C^*)$ der *maximale Radius* ist, der nötig ist, um alle Standorte von einem Mittelpunkt in C^* aus abzudecken.
- Daraus folgt, dass $r(C) \leq 2r(C^*)$. (Dies widerspricht der Annahme $r(C^*) < \frac{1}{2}r(C)$.)



Theorem: Gütegarantie des Greedy-Algorithmus

Theorem: Optimalität und Greedy-Algorithmus

- Sei C^* eine optimale Menge von Mittelpunkten. Dann gilt: $r(C) \leq 2r(C^*)$.
- Der Greedy-Algorithmus ist ein 2-Approximationsalgorithmus für das Center-Selection-Problem.

Anmerkung: Greedy-Algorithmus und Gütegarantie

- Der Greedy-Algorithmus platziert Mittelpunkte immer auf Standorten.
- Trotzdem erreicht er eine Gütegarantie von 2 gegenüber einer optimalen Lösung, selbst wenn die optimalen Mittelpunkte überall platziert werden dürfen (z.B. Punkte in einer Ebene).

Frage & Theorem: Bessere Approximationsalgorithmen

- Gibt es Hoffnung auf einen 3/2- oder 4/3-Approximationsalgorithmus? Eher nein!
- Ein ϵ -Approximationsalgorithmus für das Center-Selection-Problem mit beliebigem $\epsilon < 2$ existiert nur, wenn P = NP gilt.
 - **Bedeutung:** Eine bessere Approximationsgarantie als 2 ist so schwer wie die Frage, ob P=NP, was das Problem als NP-schwer einstuft.

14. Heuristiken und Lokale Suche

Verfahren:

- Konstruktionsverfahren
- Verbesserungsheuristiken – Lokale Suchverfahren
- Metaheuristiken
 - Simulated Annealing
 - Tabu Suche
 - Evolutionäre Algorithmen

Konstruktionsverfahren

Eigenschaften:

- Meist sehr **problemspezifisch** (d.h. sie sind oft nur für ein bestimmtes Problem oder eine bestimmte Art von Problemen konzipiert).
- Häufig **intuitiv gestaltet**.
- Basiert oft auf dem **Greedy-Prinzip**

Greedy-construction-heuristic:

Greedy-construction-heuristic:

$x \leftarrow$ leere Lösung

while x ist keine vollständige Lösung

$e \leftarrow$ die aktuell nützlichste Erweiterung von x

$x \leftarrow x \oplus e$

- Beginnt mit einer leeren Lösung x .
- In einer `while`-Schleife wird die Lösung schrittweise erweitert, solange sie nicht vollständig ist.
- In jedem Schritt wird die aktuell nützlichste Erweiterung e für die Lösung x bestimmt.
- Die Lösung x wird um diese Erweiterung e ergänzt ($x \leftarrow x \oplus e$).

☰ Beispiele für Konstruktionsverfahren

Wir haben bereits einige Konstruktionsverfahren kennengelernt:

- **Minimaler Spannbaum:**
 - Greedy Algorithmen von Prim und Kruskal
 - Berechnen ein **globales Optimum** und sind daher **exakte Algorithmen**.

- **0/1-Rucksackproblem:**

- *First-Fit-Heuristik*: Berücksichtigt alle Gegenstände in nicht aufsteigendem Verhältnis von Wert zu Gewicht (d.h., absteigend nach Wert/Gewicht sortiert) und packt jeden passenden Gegenstand ein.

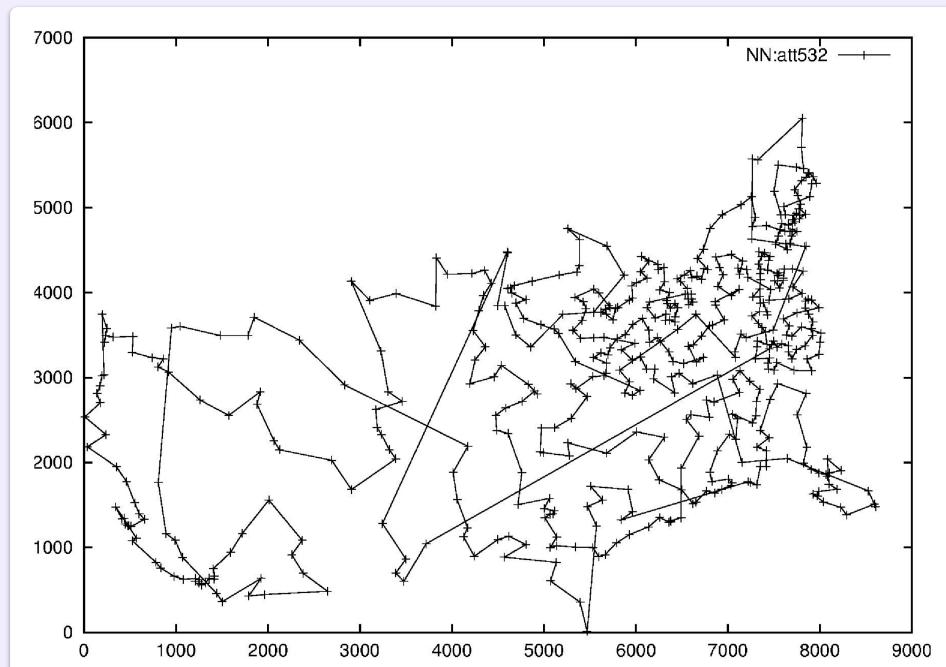
- **Vertex Cover Problem**

- **Lastverteilung:**

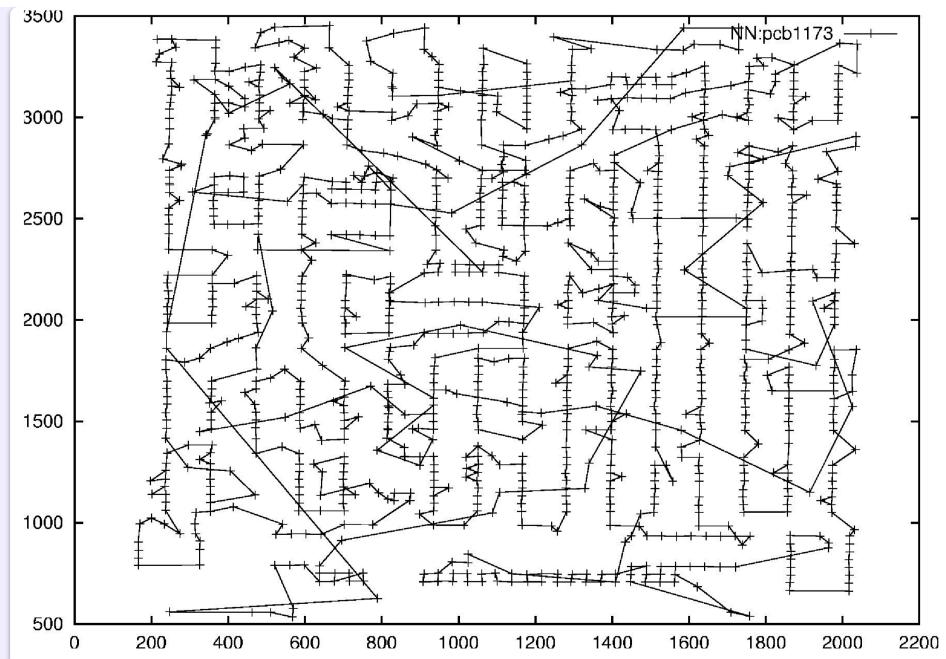
- *List Scheduling Algorithmus*
- Gütegarantie: 2 (d.h., die Lösung ist maximal doppelt so schlecht wie die optimale Lösung).

- **TSP (Traveling Salesperson Problem):**

- *Spanning-Tree-Heuristik*:
 - Gütegarantie: 2 für metrisches TSP (wenn die Dreiecksungleichung gilt, d.h. der direkte Weg ist nie länger als ein Umweg).
- *Nearest-Neighbor-Heuristik*:
 - Starte bei einem Knoten und gehe immer zum nächsten nicht besuchten Nachbarn.



(Man nimmt immer die nächste Stadt die noch nicht genommen wurde)



(Printerplatte wo Chips draufgelötet werden. Bei allen Kreuzern müssen Löcher gemacht werden)

☰ Weiteres Beispiel: Insertion-Heuristiken für das TSP

Vorgehen:

- Starte mit einer Tour von 3 Knoten oder im Fall von Euklidischen Instanzen mit der konvexen Hülle.
- Füge die restlichen Knoten schrittweise zur Tour hinzu, bis alle Knoten eingebunden sind.

Unterschiedliche Strategien für die Auswahl des nächsten einzufügenden Knotens
(keine ist immer am besten):

- **Nearest Insertion:** Wähle den Knoten, der zu einem bereits in der Tour befindlichen Knoten am nächsten ist.
- **Cheapest Insertion:** Wähle den Knoten, dessen Einfügen die Tourlänge am wenigsten erhöht.
- **Farthest Insertion:** Wähle den Knoten, der zu einem bereits in der Tour befindlichen Knoten am weitesten entfernt ist.
- **Random Insertion:** Wähle Knoten zufällig.

Strategien für das Einfügen des neuen Knotens (sobald der nächste Knoten ausgewählt ist):

- Füge den neuen Knoten nach dem Knoten in der bestehenden Tour ein, der ihm am nächsten ist (oder nach dem nächstbesten Knoten).

- Füge den neuen Knoten so ein, dass eine minimale Zunahme der Tourlänge verursacht wird.

Praktische Ergebnisse:

- Oft *10-20% über dem Optimum*.
- *Keine konstante Gütegarantie* für das metrische TSP (d.h., die Abweichung vom Optimum kann beliebig groß werden, auch wenn die Dreiecksungleichung gilt).

Lokale Suche

Verbesserungsheuristiken

Konstruktionsheuristiken sind oft intuitiv und schnell, liefern aber häufig keine ausreichend guten Lösungen.

Idee der Lokalen Suche:

- Versuche eine Ausgangslösung durch kleine Änderungen iterativ zu verbessern.
- **i.A. (in aller Regel) keine Gütegarantien.**
- In der Praxis aber oft deutliche Verbesserung von Lösungen.
- Für die meisten Anwendungen akzeptable Laufzeiten.

Lokale Suche

Prinzip:

```

 $x \leftarrow$  Ausgangslösung
while Abbruchkriterium nicht erfüllt
    Wähle  $x' \in N(x)$ 
    if  $x'$  besser als  $x$ 
         $x \leftarrow x';$ 

```

- $N(x)$: Nachbarschaft von x (Menge aller Lösungen, die durch eine "kleine" Änderung aus x entstehen).

Komponenten, die für eine lokale Suche wichtig sind

- **Lösungsrepräsentation:** Wie wird eine mögliche Lösung dargestellt?
- **Nachbarschaftsstruktur:** Welche Lösungen werden von einer aktuellen ausgehend unmittelbar in Erwägung gezogen? (Definition von $N(x)$)
- **Schrittfunktion:** Wie wird die Nachbarschaft durchsucht und welche Nachbarlösung wird als Nächstes gewählt?
- **Terminierungskriterium:** Wann wird die Suche beendet? (z.B. maximale Iterationen, keine Verbesserung über eine bestimmte Anzahl von Iterationen, Erreichen eines bestimmten Qualitätsniveaus).

Definition - Nachbarschaftsstruktur

Eine Nachbarschaftsstruktur ist eine Funktion $N : S \rightarrow 2^S$, die jeder gültigen Lösung $x \in S$ (wobei S der Lösungsraum ist) eine Menge von Nachbarn $N(x) \subseteq S$ zuweist.

- $N(x)$ wird auch **Nachbarschaft von x** genannt.

Eigenschaften:

- Die Nachbarschaft ist üblicherweise implizit durch *mögliche Veränderungen (Züge, Moves)* definiert. (Das heißt, man beschreibt, welche kleinen Änderungen man an einer Lösung vornehmen darf, um eine Nachbarlösung zu erhalten, anstatt explizit alle Nachbarn aufzulisten.)
- **Darstellung als Nachbarschaftsgraph möglich:**
 - Knoten entsprechen den Lösungen.
 - Eine Kante (u, v) existiert, wenn $v \in N(u)$ (d.h., v ist ein Nachbar von u).
- Es gilt, die **Größe der Nachbarschaft vs. Suchaufwand** abzuwägen.
 - Eine größere Nachbarschaft kann bessere Lösungen finden, erfordert aber mehr Rechenzeit pro Iteration.
 - Eine kleinere Nachbarschaft ist schneller zu durchsuchen, läuft aber Gefahr, in lokalen Optima stecken zu bleiben.

Lokale Suche: Vertex Cover

Vertex Cover (VC):

- Gegeben sei ein Graph $G = (V, E)$.
- Finde eine minimale Teilmenge $C \subseteq V$ von Knoten, sodass für jede Kante $(u, v) \in E$ entweder $u \in C$ oder $v \in C$ (oder beide) gilt. (Das heißt, jeder Kante muss mindestens ein Endpunkt im Vertex Cover haben.)

Nachbarschaftsstruktur für Vertex Cover:

- Eine Lösung $C' \in N(C)$ (Nachbar von C) existiert, wenn C' aus C durch Löschen eines einzigen Knotens erzeugt werden kann und C' immer noch ein gültiges Vertex Cover ist.
- Die Größe der Nachbarschaft $|N(C)| = O(|V|)$ (proportional zur Anzahl der Knoten im Graphen).

Lokale Suche für Vertex Cover (Beispiel):

- **Starte mit einer initialen Vertex Cover C , z.B. $C = V$** (die Menge aller Knoten ist immer ein gültiges Vertex Cover, wenn auch selten minimal).
- **Verbesserungsschritt:** Wenn es einen Nachbarn $C' \in N(C)$ gibt, der ein *gültiges Vertex Cover* ist (d.h., C' ist immer noch ein VC, obwohl ein Knoten entfernt wurde), dann ist dieser Nachbar immer eine *bessere Lösung*, da $|C'| = |C| - 1$ (die Kardinalität, also die Anzahl der Knoten, ist um 1 geringer).
- **Iteration:** Ersetze die aktuelle Lösung C durch solches C' und wiederhole den Prozess.

Diese lokale Suche terminiert nach $O(|V|)$ Schritten, da in jedem Schritt die Größe des Vertex Covers um 1 reduziert wird und die minimale Größe 0 ist.

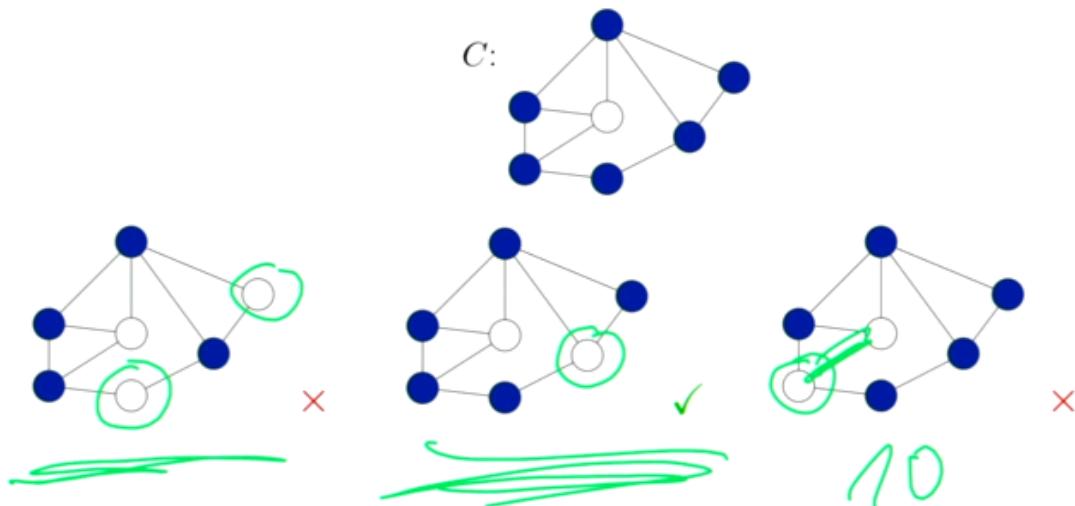
Die lokale Suche liefert nicht immer eine optimale Lösung.

Lokales Optimum:

- Ein *lokales Optimum* ist eine Lösung, bei der kein Nachbar (im Sinne der definierten Nachbarschaftsstruktur) strikt besser ist.
- Die lokale Suche kann die aktuelle Lösung daher nicht weiter verbessern, auch wenn es global bessere Lösungen geben könnte, die nicht direkt in der Nachbarschaft liegen.

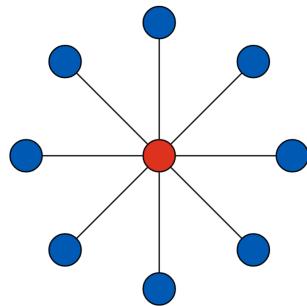
☰ Beispiel Slido

Frage 1: Welche blau markierten Mengen sind in der Nachbarschaft $N(C)$ enthalten?



Man darf in unserer Definition nur einen Knoten entfernen solange das Ergebnis stimmt. Beim 1. werden 2 entfernt und beim 3. Ist die eine Kante nicht mehr abgedeckt und dadurch kein Vertex Cover mehr. Daher stimmt nur 2..

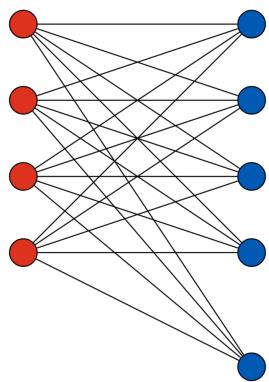
☰ Beispiel 1



(Globales) Optimum = roter Knoten in der Mitte
Lokales Optimum = alle blauen Knoten

Beispiel für eine Grenze von dieser Heuristik
Hier kann man keinen einzelnen Knoten entfernen.

☰ Beispiel 2



(Globales) Optimum = alle roten Knoten
Lokales Optimum = alle blauen Knoten

Hier kann man auch nichts entfernen ohne, dass es ungültig wird

☰ Beispiel 3

Optimum: Alle geraden Knoten



Lokales Optimum: Jeder dritte Knoten wird ausgelassen.



Hier auch nicht.

Die lokale Suche mit dieser Nachbarschaftsstruktur hat einfach ihre Grenzen

Lokale vs. globale Optima

Für die Maximierung einer Zielfunktion $f(x)$ gilt:

- Ein **lokales Maximum** in Bezug auf eine Nachbarschaftsstruktur \mathcal{N} ist eine Lösung x für die gilt: $f(x) \geq f(x')$ für alle $x' \in \mathcal{N}(x)$.
- **Intuitive Erklärung:** Ein lokales Maximum ist der höchste Punkt in einer bestimmten "Umgebung" (Nachbarschaft) der Lösung. Es ist nicht unbedingt der höchste Punkt der gesamten Funktion, sondern nur im unmittelbaren Umfeld.
- Die Nachbarschaftsstruktur bestimmt, welche Lösungen lokal optimal sind.

Verbesserungsmöglichkeiten (um bessere/globale Optima zu finden):

- Verwendung anderer/größerer Nachbarschaften.
- Iterierte lokale Suche: Wende lokale Suche wiederholt auf unterschiedliche Startlösungen an.
- Kombination unterschiedlicher lokaler Suchmethoden.

Lokale Suche: Vertex Cover

Alternative Nachbarschaft \mathcal{N}'

- Anstatt nur einen Knoten zu entfernen, können auch zwei Knoten entfernt und einer hinzugefügt werden.
- Formal beinhaltet $\mathcal{N}'(C)$ eine Knotenmengen $S \in \mathcal{N}(C)$ oder wenn S durch Hinzufügen eines Knotens von $V \setminus C$ und Entfernen von zwei Knoten aus C gebildet werden kann

und noch immer ein Vertex Cover ist.

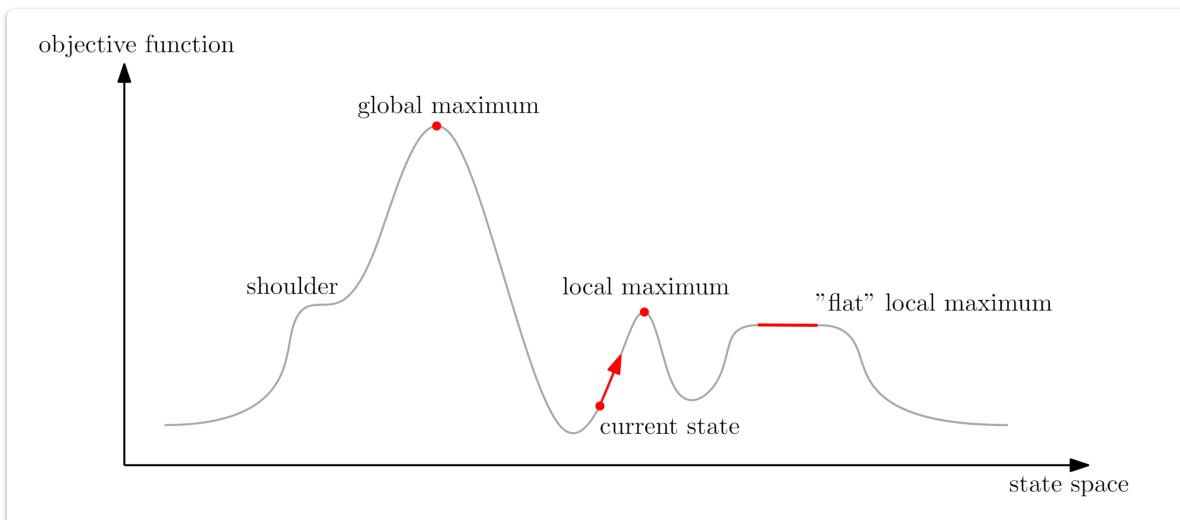
- $O(|\mathcal{V}|^3) \rightarrow$ lokale Suche wird aufwändiger.

- **Intuitive Erklärung:** Die Komplexität der lokalen Suche erhöht sich erheblich, wenn man eine komplexere Nachbarschaftsdefinition (z.B. Entfernen von zwei Knoten und Hinzufügen von einem) verwendet. Die Notation $O(|\mathcal{V}|^3)$ bedeutet, dass die Rechenzeit proportional zur dritten Potenz der Anzahl der Knoten im Graphen ($|\mathcal{V}|$) anwächst. Das ist deutlich mehr Aufwand als z.B. $O(|\mathcal{V}|)$, was für eine einfache Nachbarschaft der Fall wäre.

Lokale vs. globale Optima

Generelles Problem der lokalen Suche:

Es wird nur ein nächstes **lokales Optimum** gefunden, wir sind aber i.A. an einem globalen Optimum interessiert.



Lokale Suche: SAT

- **Gegeben:** Eine boolesche Formel in konjunktiver Normalform mit Variablen x_1, \dots, x_n .
 - **Beispiel:** $(\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$
- **Gesucht:** Variablenzuweisung, sodass alle Klauseln erfüllt werden.
- **Optimierungsvariante:** MAX-SAT - Maximiere die Anzahl der erfüllten Klauseln.

Hinweise:

- Repräsentation von Lösungen mit einem binären Vektor $x = \{0, 1\}^n$.
- **NP-vollständig** (Dies bedeutet, dass es wahrscheinlich keinen effizienten Algorithmus zur Lösung des Problems gibt, der in Polynomialzeit läuft).

k-flip Nachbarschaft für binäre Vektoren

- Nachbarlösungen haben eine *Hamming-Distanz* $\leq k$.
 - d.h., sie unterscheiden sich in bis zu k Bits (Positionen im binären Vektor).
- Größe der Nachbarschaft: $O(n^k)$ (Die Anzahl der Nachbarn wächst polynomiell mit n und exponentiell mit k).

Schrittfolge – Wahl von $x' \in N(x)$ in der lokalen Suche

Auswahlmöglichkeiten:

- **Best Improvement:** Durchsuche $N(x)$ *vollständig* und nimm eine beste Nachbarlösung.
- **First Improvement:** Durchsuche $N(x)$ in einer bestimmten Reihenfolge, nimm *erste* Lösung, die besser als x ist.
- **Random Neighbor:** Wähle eine *zufällige* Lösung aus $N(x)$.

Hinweise:

- Wahl kann starken Einfluss auf Performance haben.
- Allgemein ist kein Verfahren immer besser als ein anderes (die Effektivität hängt oft vom spezifischen Problem und den Daten ab).
- Beispielsweise ist ein Durchlauf von *Random Neighbor* meist schneller, dafür benötigt *Best Improvement* oft erheblich weniger Iterationen (da es in jedem Schritt die optimale Verbesserung wählt).

Abbruchkriterium

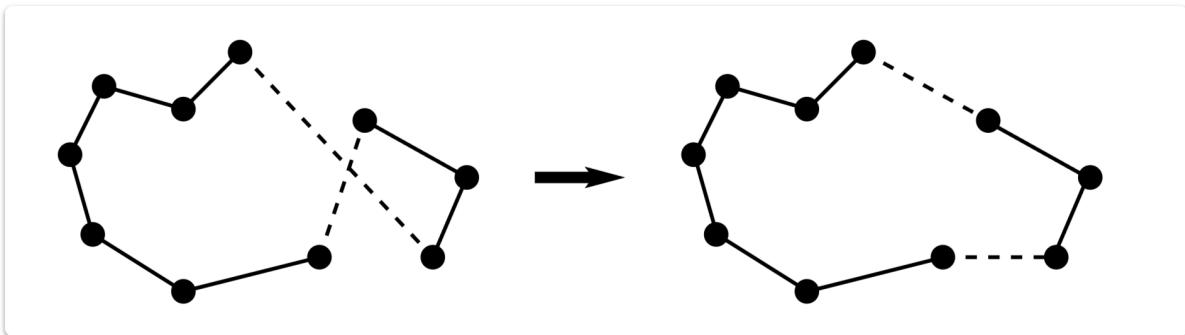
- Meist wird die lokale Suche beendet, wenn ein *lokales Optimum* erreicht wurde (ein Zustand, in dem keine der benachbarten Lösungen besser ist).
- Bei einer *Random Neighbor* Schrittfolge kann ein solches jedoch nicht direkt erkannt werden (da man nicht systematisch alle Nachbarn prüft).
- Manchmal ist eine vollständige lokale Suche auch zu *zeitaufwendig*.

Alternativen für Abbruch:

- nach einer bestimmten Iterationsanzahl oder Zeit.
- wenn eine ausreichend gute Lösung gefunden wurde.
- wenn keine weitere Verbesserung über eine bestimmte Anzahl letzter Iterationen erreicht wurde (Konvergenz).

Lokale Suche für das symmetrische TSP

Nachbarschaftsstruktur: Austausch zweier Kanten ("2-exchange" oder "2-opt").



Größe der Nachbarschaft: $O(n^2)$ (Die Anzahl der möglichen 2-opt Operationen wächst quadratisch mit der Anzahl der Knoten n).

Inkrementelle Evaluierung:

- Berechne den Wert einer Nachbarlösung *effizient* aus dem Wert der aktuellen Lösung unter Berücksichtigung der wegfallenden und hinzukommenden Kanten.
- Benötigt hier nur konstante Zeit im Vergleich zu $O(n)$ Zeit für eine vollständige unabhängige Berechnung des Zielfunktionswerts (was eine Neuberechnung der gesamten Tourlänge bedeuten würde).

2-opt lokale Suche für das symmetrische TSP

Mit *First-Improvement* Schrittfunktion (die erste verbesserte Nachbarlösung wird gewählt).

Vorgehen

- (1) Sei $E(T) = \{(i_1, i_2), (i_2, i_3), \dots, (i_n, i_1)\}$ die Menge der Kanten der aktuellen Tour T und sei $i_{p+1} = i_1$.
 - (Dies stellt die Menge der Kanten in einer zyklischen Tour dar, wobei i_n mit i_1 verbunden ist).
- (2) Sei $Z = \{(i_p, i_{p+1}), (i_q, i_{q+1}) \subset T | 1 \leq p, q < n \wedge p + 1 < q\}$ sei die Menge aller Paare nicht nebeneinanderliegender Kanten. (Dies sind die Kantenpaare, die für einen 2-opt Austausch in Frage kommen, d.h., sie dürfen nicht benachbart sein).
- (3) Für alle Kantenpaare $\{(i_p, i_{p+1}), (i_q, i_{q+1})\}$ aus Z :
 - Falls $c_{i_p i_{q+1}} + c_{i_q i_{p+1}} < c_{i_p i_{p+1}} + c_{i_q i_{q+1}}$ (Wenn die Summe der Längen der neuen Kanten kleiner ist als die Summe der Längen der alten Kanten):
 - $T \leftarrow (T \setminus \{(i_p, i_{p+1}), (i_q, i_{q+1})\}) \cup \{(i_p, i_q), (i_{p+1}, i_{q+1})\}$ (Ersetze die alten Kanten durch die neuen).
 - Gehe zu (4) (Eine Verbesserung wurde gefunden und angewendet).

- (4) retourniere T . (Die verbesserte Tour wird zurückgegeben).

Laufzeitkomplexität:

- Da $|N(x)| = O(n^2)$ (Größe der Nachbarschaft) und jede Nachbarlösung in konstanter Zeit inkrementell evaluiert wird, benötigt eine Iteration $O(n^2)$ Zeit.
- **Frage:** Wieviele Iterationen sind notwendig bis ein lokales Optimum erreicht ist?
 - **Worst-Case:** Bis zu $O(n!)$ Iterationen (ohne Beweis)! (Dies entspricht der Anzahl aller möglichen Permutationen der Knoten, was extrem hoch ist).
 - Die Worst-Case-Laufzeit dieser lokalen Suche ist daher *exponentiell*!
 - **Praxis:** Dennoch ist das Verfahren auch auf großen Instanzen meist *schnell*. Startet man mit einer sinnvollen Ausgangslösung (z.B. einer Heuristik), sind in der Regel nur wenige Iterationen erforderlich, um ein lokales Optimum zu erreichen.

r-opt Nachbarschaft für das Symmetrische TSP

- **Verallgemeinerung:** Die Idee der 2-opt Nachbarschaft kann verallgemeinert werden. Es werden $r \geq 2$ Kanten durch neue ersetzt.

Prinzip der r-opt lokalen Suche

- (1) Wähle eine beliebige Anfangstour $T = \{(i_1, i_2), (i_2, i_3), \dots, (i_n, i_1)\}$.
- (2) Sei Z die Menge aller r -elementigen Teilmengen von T . (Dies sind alle möglichen Gruppen von r Kanten, die aus der aktuellen Tour entfernt werden könnten).
- (3) Für alle $R \in Z$: Setze $S = T \setminus R$ und konstruiere alle Touren, die S enthalten. Ist ein S' besser als T , setze $T = S'$ und gehe zu (2). (Finde die beste Art und Weise, die verbleibenden Pfade in S mit r neuen Kanten zu einer vollständigen Tour zu verbinden, die eine Verbesserung darstellt).
- (4) T ist das Ergebnis.

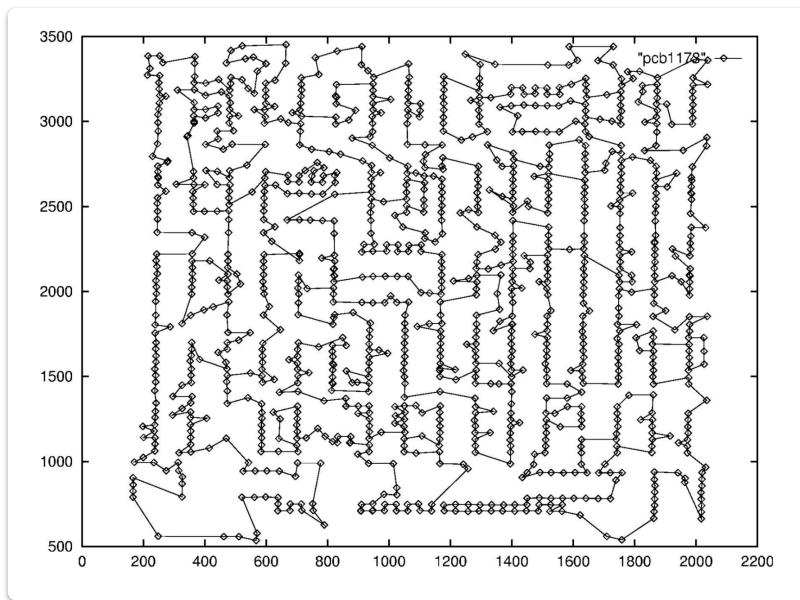
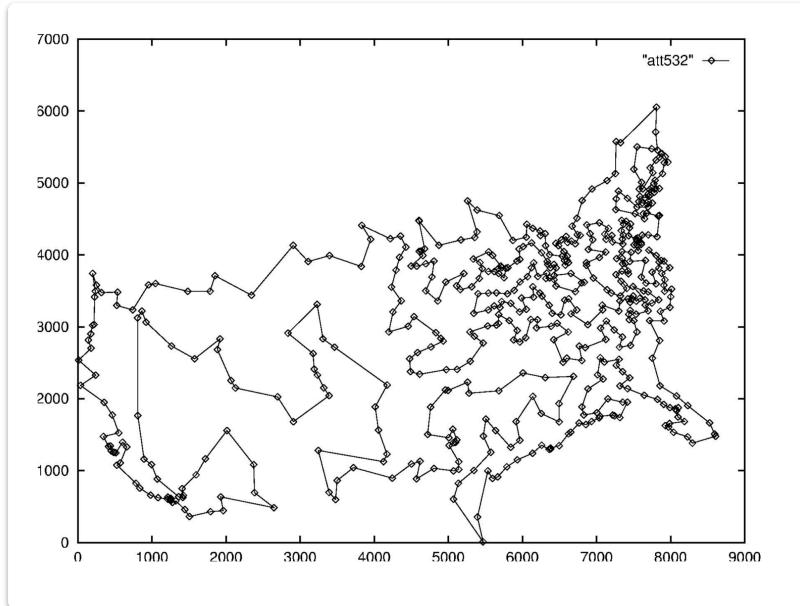
Laufzeitkomplexität der r-opt lokalen Suche

Größe einer r-opt Nachbarschaft:

- Es gibt $\binom{n}{r} = O(n^r)$ Möglichkeiten r unterschiedliche Kanten aus einer aktuellen Tour zu entfernen. (Dies ist die Anzahl der Kombinationen, n über r).
- Das Entfernen führt zu r nicht zusammenhängenden Pfaden.
- Diese können auf $O(r!)$ Möglichkeiten zu neuen Touren zusammengefügt werden.
- Somit ist $|N(x)| = O(n^r \cdot r!) = O(n^r)$. (Die $r!$ Faktoren sind dominant für kleine r , aber für festes r ist die Abhängigkeit von n entscheidend).

Hinweis: Wie bereits bei 2-opt ist auch hier im allgemeinen Fall die Worst-Case-Anzahl der möglichen Iterationen *nicht polynomiell* beschränkt.

2-opt Lösung für TSP



Hinweis: Die Lösung wurde durch eine lokale Suche mit Random-Neighbor Schrittfunktion gefunden. Es gibt 2 Kanten, die sich kreuzen, daher ist diese Lösung kein lokales Optimum

Zusammenfassung zur lokalen Suche für das TSP

Anwendung:

- **2-opt** wird sehr häufig eingesetzt.
 - Kommt meist auf ca. *6–8% an die optimale Lösung* heran.
- **3-opt** wird manchmal verwendet (deutlich zeitaufwändiger).
 - Kommt meist auf *3–4% an die optimale Lösung* heran.

- **4-opt** ist in der Praxis i.A. bereits zu *zeitaufwändig*.

Hinweis: Die Prozentangaben beziehen sich auf bestimmte Instanzen, die zum Testen der Algorithmen verwendet werden und sollen hier nur einen *groben Richtwert* vermitteln.

Weitere Nachbarschaftsstrukturen:

- **Verschieben eines Knotens an eine andere Position.**
 - Für das *asymmetrische TSP* (wo die Kosten von A nach B nicht unbedingt gleich den Kosten von B nach A sind) häufig besser geeignet.
- **Verschieben einer Teilsequenz an eine andere Position.**
- **Lin-Kernighan Heuristik (1973):**
 - Eine der *führenden, schnellen Heuristiken* für große TSPs.
 - Kommt meist auf *1–2% an das Optimum* heran.
 - *Variable Tiefensuche:* Die Anzahl der ausgetauschten Kanten ist nicht grundsätzlich beschränkt.
 - Es werden jedoch nur „vielversprechende“ Kantenaustausche durchprobiert (dies reduziert den Suchraum erheblich, indem nur vielversprechende Pfade verfolgt werden).

Maximaler Schnitt (Maximal Cut)

- **MAX-CUT:** Gegeben sei ein ungerichteter Graph $G = (V, E)$ mit positiven ganzzahligen Kantengewichten w_{uv} für alle Kanten $(u, v) \in E$. Finde eine Partition der Knoten (A, B) , sodass das Gesamtgewicht von Kanten, die Knoten in den unterschiedlichen Partitionen verbinden, maximiert wird.

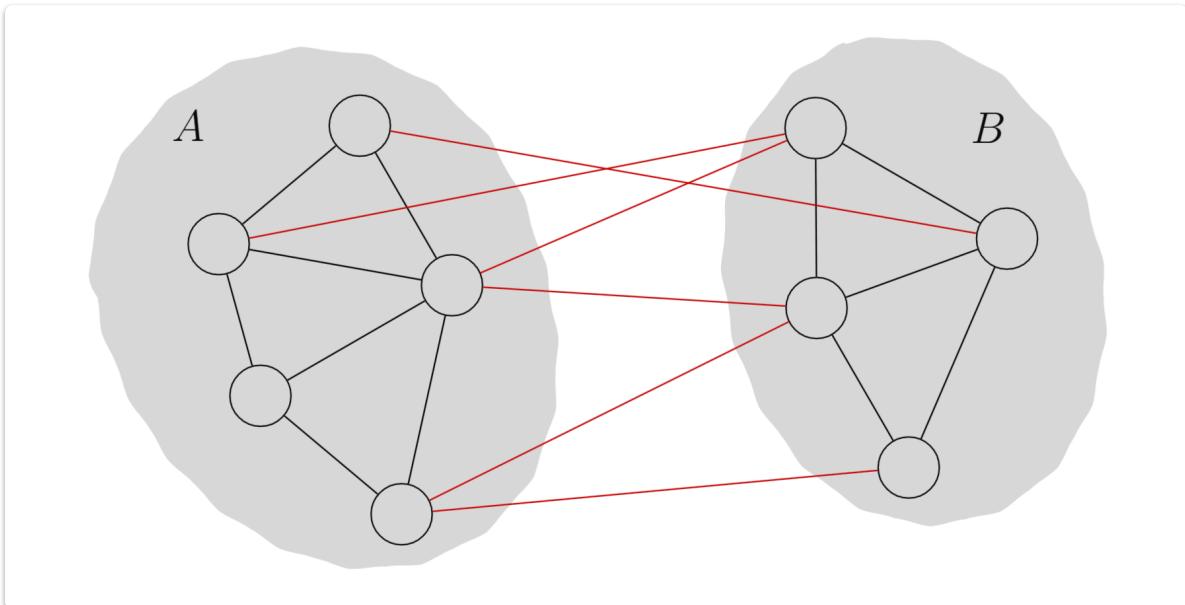
- Formel für das Gewicht der Kanten zwischen Partitionen A und B :

$$w(A, B) := \sum_{u \in A, v \in B} w_{uv}$$

Beispielanwendung:

- n Aktivitäten, m Personen.
- Jede Person möchte an zwei Aktivitäten teilnehmen.
- Plane die Aktivitäten am Morgen und am Nachmittag so, dass eine maximale Anzahl an Personen daran teilnehmen kann (dies entspricht der Maximierung der "Schnittkanten", d.h., der Personen, die an Aktivitäten in beiden Zeiträumen teilnehmen können).

Hinweis: MAX-CUT ist *NP-vollständig*.

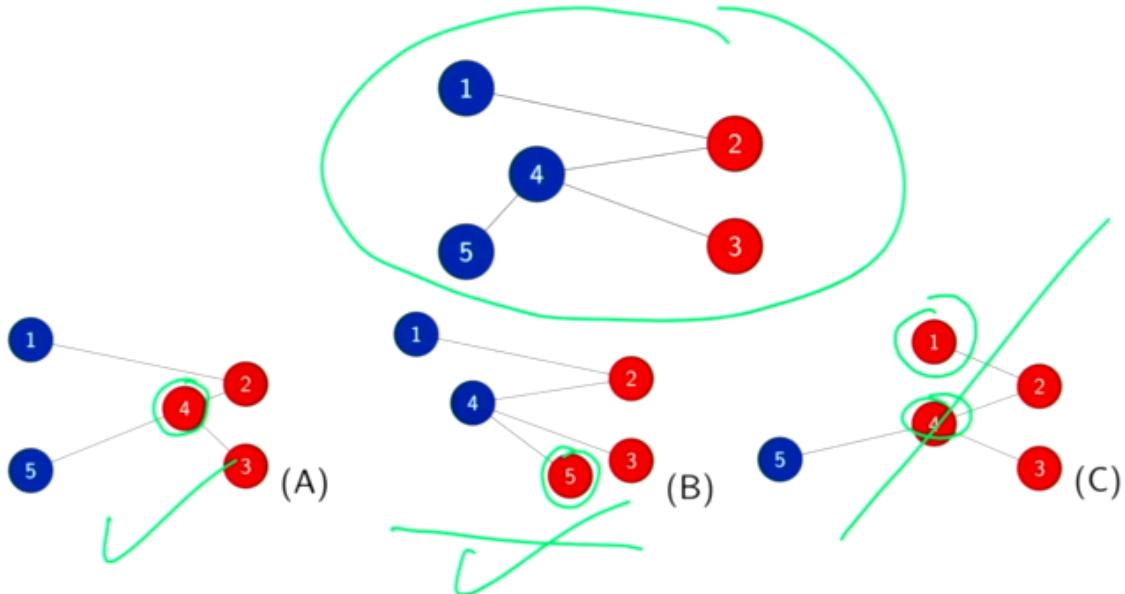


1-Flip-Nachbarschaft: Gegeben sei eine Partition (A, B) . Verschiebe einen Knoten von A nach B oder von B nach A . (Dies ist die einfachste Form der Nachbarschaft bei Partitions-Problemen, bei der nur ein Element seine Gruppenzugehörigkeit ändert).

Algorithmus: Ausgehend von einer gültigen Initiativlösung ist auf die 1-Flip-Nachbarschaft aufbauend unmittelbar eine einfache lokale Suche möglich. (Man beginnt mit einer beliebigen Aufteilung der Knoten und versucht dann, durch das Verschieben einzelner Knoten eine bessere Schnittgröße zu erreichen).

☰ Beispiel

Frage 3: Welche Lösungen sind Teil der Flip-Nachbarschaft?



Analyse der lokalen Suche

Wir können zeigen: Wird die lokale Suche ausgeführt bis eine lokal optimale Lösung erreicht wurde, so gilt eine *Approximationsgüte von 1/2*.

ⓘ Theorem

Theorem: Sei (A, B) eine lokal optimale Partition und sei (A^*, B^*) eine optimale Partition.

Dann ist

$$w(A, B) \geq \frac{1}{2} \sum_{e \in E} w_e \geq \frac{1}{2} w(A^*, B^*)$$

(Gewichte sind *nicht negativ*)

✓ Beweis

Beweis:

- Lokale Optimalität bedeutet, dass für alle

$$u \in A : \sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv}$$

Das Aufsummieren aller Ungleichungen ergibt:

$$2 \sum_{\{u,v\} \subseteq A} w_{uv} \leq \sum_{u \in A, v \in B} w_{uv} = w(A, B)$$

- Ähnlich ist $2 \sum_{\{u,v\} \subseteq B} w_{uv} \leq \sum_{u \in A, v \in B} w_{uv} = w(A, B)$
- Nun gilt,

$$\sum_{e \in E} w_e = \underbrace{\sum_{\{u,v\} \subseteq A} w_{uv}}_{\leq \frac{1}{2}w(A, B)} + \underbrace{\sum_{u \in A, v \in B} w_{uv}}_{w(A, B)} + \underbrace{\sum_{\{u,v\} \subseteq B} w_{uv}}_{\leq \frac{1}{2}w(A, B)} \leq 2w(A, B) \quad \square$$

- Jede Kante wird einmal gezählt.

Metaheuristiken

🔥 Definition

- **Metaheuristiken:** Sind problemunabhängig formulierte Algorithmen zur heuristischen Lösung schwieriger Optimierungsaufgaben.
 - **Anpassung:** Teile dieser Algorithmen müssen an das jeweilige Problem angepasst werden, wie beispielsweise die Nachbarschaftsstruktur auch in der lokalen Suche.

Betrachtete Metaheuristiken:

- Simulated Annealing
- Tabu-Suche
- Evolutionäre Algorithmen

Simulated Annealing (SA)

- Inspiriert durch den physikalischen Prozess der langsamen Abkühlung von Materialien zur Erreichung einer stabilen Kristallstruktur, z.B. nach dem Glühen eines Metalls.
- **Grundlegende Idee:** Auch *schlechtere Nachbarlösungen werden mit einer bestimmten Wahrscheinlichkeit akzeptiert*.
- **Schrittfunktion:** I.A. Random Neighbor (es wird ein zufälliger Nachbar als nächste Lösung gewählt).

Variablen:

- Z : (Pseudo-)Zufallszahl $\in [0, 1]$
- T : „Temperatur“

```

Simulated-Annealing():
t ← 0
T ← Tinit
x ← Ausgangslösung
while Abbruchkriterium nicht erfüllt
    Wähle  $x' \in N(x)$  zufällig
    if  $x'$  besser als  $x$ 
         $x \leftarrow x'$ 
    elseif  $Z < e^{-|f(x') - f(x)|/T}$ 
         $x \leftarrow x'$ 
     $T \leftarrow g(T, t)$ 
     $t \leftarrow t + 1$ 

```

Metropolis-Kriterium

(Das grün markierte ist das, was durch SA dazugekommen ist.) Mit der Wahrscheinlichkeit die hier ausgedrückt wird, akzeptieren wir auch schlechtere Versuche

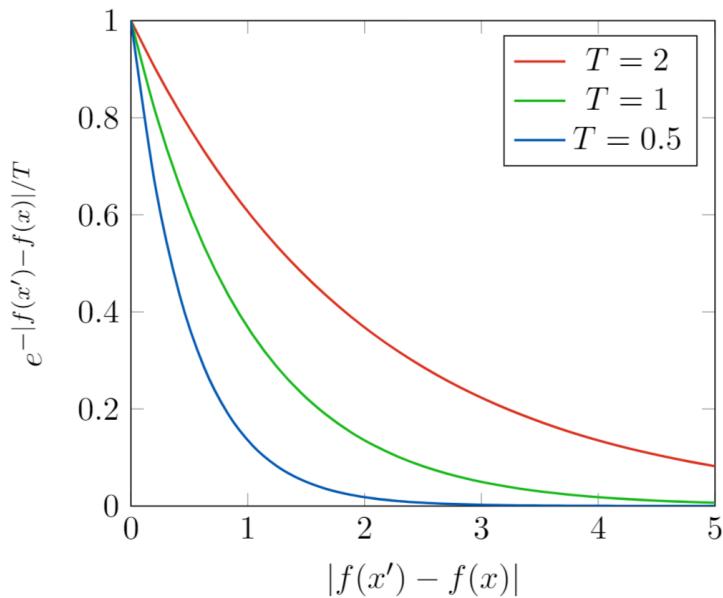
Metropolis Kriterium

Das Metropolis-Kriterium ist die Akzeptanzbedingung für schlechtere Lösungen in Simulated Annealing:

$$Z < e^{-|f(x') - f(x)|/T}$$

- Z : Zufallszahl $\in [0, 1]$
- $f(x')$: Zielfunktionswert der Nachbarlösung
- $f(x)$: Zielfunktionswert der aktuellen Lösung
- T : Aktuelle Temperatur

Metropolis Kriterium (Vergleich)



Das Diagramm zeigt, wie die Akzeptanzwahrscheinlichkeit (y-Achse) in Abhängigkeit von der Verschlechterung ($|f(x') - f(x)|$, x-Achse) für verschiedene Temperaturen (T) verläuft.

Eigenschaften des Metropolis-Kriteriums

- Nur geringfügig schlechtere Lösungen werden mit höherer Wahrscheinlichkeit akzeptiert als viel schlechtere.
 - Dies ist im Graphen ersichtlich: Für einen gegebenen T -Wert sinkt die Akzeptanzwahrscheinlichkeit drastisch, je größer die Verschlechterung $|f(x') - f(x)|$ wird.
- Anfangs, bei hoher Temperatur T , werden schlechtere Lösungen mit größerer Wahrscheinlichkeit akzeptiert als im späteren Verlauf bei niedrigerer Temperatur.
 - Im Graphen verschiebt sich die Kurve bei höherem T (z.B. $T = 2$) nach oben und ist flacher, was bedeutet, dass für dieselbe Verschlechterung eine höhere Akzeptanzwahrscheinlichkeit besteht als bei niedrigerem T (z.B. $T = 0.5$).
 - Dies ist ein Kernpunkt von Simulated Annealing: Am Anfang kann der Algorithmus "sprunghafter" sein und auch schlechtere Lösungen erkunden, um lokale Optima zu verlassen. Mit sinkender Temperatur wird der Algorithmus "konservativer" und konzentriert sich auf die Verfeinerung guter Lösungen.

Abkühlungsplan

Der Abkühlungsplan definiert, wie die Temperatur T im Laufe des Simulated Annealing Algorithmus reduziert wird. Dies ist entscheidend für das Konvergenzverhalten des Algorithmus.

Geometrisches Abkühlen

Dies ist eine gängige und einfache Methode zur Temperaturreduktion.

- **Faustregel für T_{init} (Anfangstemperatur):**
 - $T_{init} : f_{max} - f_{min}$, wobei f_{max} bzw. f_{min} eine obere bzw. untere Schranke oder Schätzung für den maximalen/minimalen Zielfunktionswert sind.
- **Abkühlfunktion $g(T, t)$:**
 - $g(T, t) = T \cdot \alpha$, mit $\alpha < 1$ (z.B. 0,999).
 - **Erklärung:** Die Temperatur wird in jeder Iteration um einen konstanten Faktor α multipliziert und sinkt somit exponentiell. Ein Wert von $\alpha = 0,999$ bedeutet eine sehr langsame Abkühlung, was mehr Zeit zum Erkunden des Lösungsraums lässt, aber auch länger dauert.
- **Häufige Praxis:**
 - Häufig wird die Temperatur auch über einige (z.B. $|N(x)|$) Iterationen *gleich gelassen* und dann jeweils *etwas stärker reduziert*.

Adaptives Abkühlen

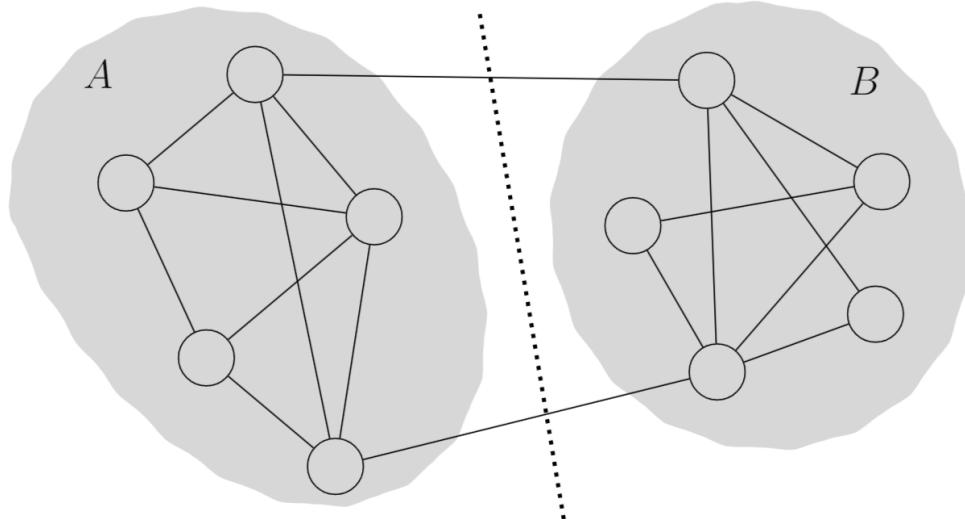
- Es wird der **Anteil der Verbesserungen an den letzten erzeugten Lösungen gemessen** und auf Grund dessen T stärker oder schwächer reduziert.

Beispiel: Simulated Annealing (SA) für Graph-Bipartitionierung

SA ist eine der ersten Anwendungen für die Graph-Bipartitionierung.

Definition für Graph-Bipartitionierung

- **Gegeben:** Ein Graph $G = (V, E)$, wobei V die Menge der Knoten und E die Menge der Kanten ist.
- **Gesucht:** Eine Partitionierung (Aufteilung) des Graphen G in zwei Knotenmengen A und B , sodass folgende Bedingungen erfüllt sind:
 - $|A| = |B|$: Die Anzahl der Knoten in A und B ist gleich.
 - $A \cap B = \emptyset$: Die Mengen A und B haben keine gemeinsamen Knoten (sie sind disjunkt).
 - $A \cup B = V$: Die Vereinigung von A und B ergibt alle Knoten des Graphen.
- **Minimiere:** Die Anzahl der Kanten zwischen A und B . Formal: $|\{(u, v) \in E \mid u \in A, v \in B\}|$. Das Ziel ist es, möglichst wenig Kanten zu haben, die die beiden Mengen verbinden.



Repräsentation der Lösung: Charakteristischer Vektor

Eine mögliche Lösung (also eine spezifische Aufteilung in A und B) wird durch einen charakteristischen Vektor $x = (x_1, \dots, x_n)$ repräsentiert, wobei $n = |V|$ die Gesamtzahl der Knoten ist.

- Jeder Eintrag x_i im Vektor entspricht einem Knoten i des Graphen.
- Wenn $x_i = 0$, wird Knoten i der Menge A zugewiesen.
- Wenn $x_i = 1$ (oder ein anderer Wert außer 0), wird Knoten i der Menge B zugewiesen.

Simulated Annealing (SA) für Graph-Bipartitionierung

Die Anwendung von Simulated Annealing auf dieses Problem beinhaltet folgende Schritte und Konzepte:

- **Nachbarschaft:** Um eine Nachbarlösung zu erzeugen (d.h. eine leicht veränderte Aufteilung), tauscht man jeweils einen Knoten von Menge A mit einem Knoten von Menge B aus. Dies erzeugt eine neue Partitionierung, die bewertet werden kann.
- **Zufällige Anfangslösung:** Der Algorithmus beginnt mit einer zufälligen Initialisierung, d.h., die Knoten werden zunächst zufällig auf die Mengen A und B aufgeteilt (wobei die Bedingung $|A| = |B|$ beachtet werden muss).
- **Geometrisches Abkühlen, $\alpha = 0.95$**
- **Iterationen auf jeder Temperaturstufe:** Auf jeder Temperaturstufe werden $n \cdot (n - 1)$ Iterationen durchgeführt.
- **Eine der ersten Anwendungen von SA:** Dies unterstreicht die historische Bedeutung der Graph-Bipartitionierung als frühes Anwendungsgebiet für Simulated Annealing.

Verbesserungen

Einschränkung der Nachbarschaft und Erlauben ungültiger Lösungen

- **Flip-Nachbarschaft (vgl. MAX-CUT):**

- Verschiebe einen einzelnen Knoten in andere Menge.
 - $|N(x)| = n$ anstatt $n^2/4$.
 - **Modifizierte Zielfunktion:**
 - $f(A, B) = |\{(u, v) \in E \mid u \in A, v \in B\}| + \gamma(|A| - |B|)^2$
 - γ : Faktor für das Ungleichgewicht (Strafterm für $|A| \neq |B|$).
-

Fazit zu Simulated Annealing

- Typischerweise einfach zu implementieren.
- Parametertuning notwendig, aber meist nicht so schwer.
- Für viele Probleme gute Resultate, aber häufig können ausgefeilte Methoden noch bessere Ergebnisse liefern.
- Viele Varianten/Erweiterungen:
 - Dynamische Strategien für das Abkühlen (Wiedererwärmen)
 - Parallelisierung
 - Kombination mit anderen **Methoden**

Tabu-Suche (TS)

⌚ Definition

- Basiert auf einem **Gedächtnis (History)** über den bisherigen Optimierungsverlauf.
 - Nutzt dieses Gedächtnis, um über lokale Optima hinwegzukommen.
- **Vermeidung von Zyklen** durch Verbieten des Wiederbesuchens früherer Lösungen.
- **I.A. (Im Allgemeinen) Best Improvement Schrittfunktion:**
 - In jedem Schritt wird die **beste erlaubte Nachbarlösung** angenommen.
 - Dies gilt auch, wenn diese schlechter ist als die aktuelle Lösung.

Variablen: Bisher beste gefundene Lösung x_{best} , Aktuelle Lösung x , Nachbarlösung x' , Tabu-Liste TL , Menge erlaubter Nachbarlösungen X' .

```
Tabu-Suche():
   $x_{\text{best}} \leftarrow x \leftarrow$  Ausgangslösung
   $TL \leftarrow \{x\}$ 
  while Abbruchkriterium nicht erfüllt
     $X' \leftarrow$  Teilmenge von  $N(x)$  unter Berücksichtigung von  $TL$ 
     $x' \leftarrow$  beste Lösung von  $X'$ 
    Füge  $x'$  zu  $TL$  hinzu
    Lösche Elemente aus  $TL$ , welche älter als  $t_L$  Iterationen sind
     $x \leftarrow x'$ 
    if  $x$  besser als  $x_{\text{best}}$ 
       $x_{\text{best}} \leftarrow x$ 
```

Das Gedächtnis: Tabuliste

📋 Eigenschaften

- **Explizites Speichern von vollständigen Lösungen:**
 - Nachteil: Speicher- und zeitaufwändig.
- **Meist bessere Alternative: Speichern von Tabuattributen.**
 - D.h., nur einzelne Aspekte von besuchten Lösungen werden gespeichert.
- Lösungen sind **tabu (verboten)**, falls sie Tabuattribute enthalten.
- Als Tabuattribute werden meist Variablenwerte benutzt, die von durchgeföhrten Zügen gesetzt wurden.
 - Die Umkehrung der Züge ist dann für t_L Iterationen verboten.
- **Wichtiger Parameter:** Tabulistenlänge t_L .

Parameter - Tabulistenlänge t_L

- Wahl von t_L ist häufig sehr kritisch!
 - Zu kurze Tabulisten können zu Zyklen führen (d.h., man kehrt zu schon besuchten Lösungen zurück).
 - Zu lange Tabulisten verbieten viele mögliche Lösungen und beschränken die Suche stark (dadurch wird der Suchraum zu stark eingeschränkt).
 - Geeignete Länge ist i.A. problemspezifisch.
 - Muss experimentell bestimmt werden, oder:
 - Immer zufällig neu wählen.
 - Adaptiv anpassen (\rightarrow Reactive Tabu Search).
-

Aspirationskriterien

- Manchmal wird eine Lösung verboten (d.h. ihre Attribute sind in der Tabuliste), obwohl sie sehr gut ist.
 - Aspirationskriterium: Überschreibt den Tabu-Status einer „interessanten“ Lösung.
 - D.h. die Lösung darf gewählt werden, obwohl sie eigentlich tabu wäre.
 - Beispiel eines oft benutzten Aspirationskriteriums:
 - Eine verbotene Lösung ist besser als die bisher beste gefundene Lösung.
-

☰ Beispiel: Tabu-Suche für das Graphenfärbeproblem

Graphenfärbeproblem:

- Gegeben: Graph $G = (V, E)$.
- Gesucht: Weise jedem Knoten $v \in V$ eine Farbe $x_v \in \{1, \dots, k\}$ zu.
 - Sodass für alle Kanten $(u, v) \in E$ gilt: $x_u \neq x_v$ (benachbarte Knoten dürfen nicht die gleiche Farbe haben).
- Hinweis: Ist ein NP-vollständiges Problem.
- Optimierungsvariante: Minimiere Anzahl der „verletzten“ Kanten (Kanten, bei denen benachbarte Knoten dieselbe Farbe haben).

Aspekte der Tabu-Suche für das Graphenfärbeproblem:

- Evaluierungskriterium: Minimiere die Anzahl der „verletzten“ Kanten (Kanten, deren Endknoten dieselbe Farbe haben).
- Nachbarschaft: Eine Nachbarlösung ist eine Färbung, die sich genau in der Farbe eines Knotens unterscheidet (d.h., nur ein Knoten ändert seine Farbe).
- Tabuattribute: Paare (v, j) mit $v \in V, j \in \{1, \dots, k\}$.

- Dies bedeutet: Bestimmte Farbzueweisungen (j) zu bestimmten Knoten (v).
 - **Tabukriterium:** Wird ein Zug $(v, j) \rightarrow (v, j')$ (Knoten v ändert Farbe von j nach j') durchgeführt, ist das Attribut (v, j) für t_L Iterationen verboten.
 - Das bedeutet, dass es für t_L Iterationen verboten ist, dem Knoten v wieder die Farbe j zuzuweisen, die er *vor* dem aktuellen Zug hatte.
 - **Aspirationskriterium:** Falls ein Zug zu einer besseren Lösung als der bisher gefundenen führt, ignoriere den Tabu-Status und akzeptiere diese Lösung.
 - **Einschränkung der Nachbarschaft:** Betrachte nur Zuweisungen für Knoten, die in eine Kantenverletzung involviert sind.
 - D.h., es werden nur Knoten umgefärbt, die aktuell eine "verletzte" Kante verursachen, um die Effizienz der Suche zu erhöhen.
-

Fazit zur Tabu-Suche

- Viele weitere unterschiedliche Strategien für:
 - Gedächtnis (z.B. unterschiedliche Arten von Tabulisten oder Gedächtnisstrukturen).
 - Diversifizierung der Suche (Strategien, um den Suchraum breiter zu erkunden und nicht in lokalen Optima stecken zu bleiben).
 - Intensivierung in der Nähe gefundener Elitelösungen (Strategien, um gute gefundene Lösungen genauer zu untersuchen und zu verbessern).
- Oft exzellente Ergebnisse und vergleichsweise schnell.
- Meist relativ aufwändiges Fine-Tuning (Feinabstimmung der Parameter) notwendig.

Evolutionäre Algorithmen

Idee: Grundprinzipien der natürlichen **Evolution** werden primitiv nachgeahmt, um schwierige Optimierungsaufgaben zu lösen.

- **Population:** Eine Menge von aktuellen Kandidatenlösungen wird verwendet.
- **Selektion:** Durch natürliche Auslese ("survival of the fittest") bleiben bessere Lösungen mit höherer Wahrscheinlichkeit erhalten und erzeugen neue Lösungen.
- **Rekombination:** Neue Lösungen werden durch zufallsgesteuerte Kreuzung oder Vererbung von Lösungsmerkmalen aus den "Eltern"-Lösungen abgeleitet.
- **Mutation:** Kleine zufällige Änderungen fügen neue Lösungsmerkmale hinzu, die nicht in den Elternlösungen vorkamen. Dies ermöglicht eine Variation der Elternlösungen.

Prinzip eines evolutionären Algorithmus:

- Selektierte Eltern Q_s
- Zwischenlösungen Q_r

```

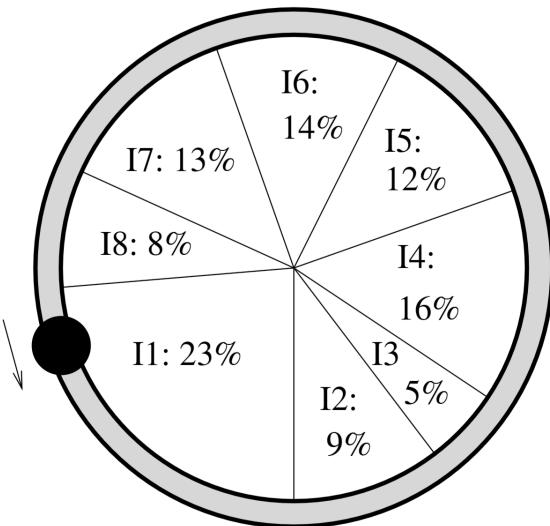
Evolutionär():
 $P \leftarrow$  Menge von Ausgangslösungen
Bewerte( $P$ )
while Abbruchkriterium nicht erfüllt
     $Q_s \leftarrow$  Selektion( $P$ )
     $Q_r \leftarrow$  Rekombination( $Q_s$ )
     $P \leftarrow$  Mutation( $Q_r$ )
    Bewerte( $P$ )

```

Fitness Proportional Selection

Roulette-Wheel Selection

- Sei $f(x_i) > 0$ der Zielfunktionswert (Fitness) jeder Lösung $x_i \in P$.
- $P = \{x_1, \dots, x_{|P|}\}$ repräsentiert die Population der Lösungen.
- Wir gehen von einem Maximierungsproblem aus.



Selektionswahrscheinlichkeit für Lösung x_i :

$$p_s(x_i) = \frac{f(x_i)}{\sum_{j=1}^n f(x_j)}$$

Selektionsdruck

Es ist wichtig, auf die Verhältnisse zwischen den Selektionswahrscheinlichkeiten der Lösungen in P zu achten.

Selektionsdruck:

Sei $p_s^{\max} = \max\{p_s(x_1), \dots, p_s(x_{|P|})\}$ die maximale Selektionswahrscheinlichkeit einer Lösung in der Population.

Sei $\bar{p}_s = 1/|P|$ die durchschnittliche Selektionswahrscheinlichkeit, wenn alle Lösungen gleich wahrscheinlich ausgewählt würden.

Dann ist der Selektionsdruck S definiert als:

$$S = p_s^{\max} / \bar{p}_s$$

- S zu niedrig: Ineffiziente Suche, da sie einer Zufallssuche ähnelt.
- S zu hoch: Rascher Verlust der Vielfalt in der Population, da einzelne (meist die besten) Lösungen zu häufig ausgewählt werden. Dies führt zu einer raschen Konvergenz zu einem lokalen Optimum.

Skalierung der Bewertungsfunktion

Skalierung: Um den Selektionsdruck S zu steuern, wird die Bewertungsfunktion meist skaliert, z.B. über eine lineare Funktion:

$$g(x_i) = a \cdot f(x_i) + b$$

mit geeigneten Werten a und b .

Hinweis: Skalierung ist auch notwendig für:

- Minimierungsprobleme (da die Roulette-Wheel Selection für Maximierungsprobleme ausgelegt ist und positive Fitnesswerte benötigt)
- Wenn $f(x_i) < 0$ (da die Fitness proportional Selection positive Fitnesswerte voraussetzt)

Alternative: Tournament Selektion

Alternative Vorgehensweise:

- (1) Wähle aus der Population k Lösungen gleichverteilt zufällig.
- (2) Die beste der k Lösungen ist die selektierte Lösung.

Eigenschaften:

- Relative Unterschiede in der Bewertung spielen keine Rolle. Es zählt nur, welche der k ausgewählten Lösungen die höchste Fitness hat.
 - Skalierung ist deshalb nicht erforderlich. (Im Gegensatz zur Fitness Proportional Selection)
 - Der Selektionsdruck wird über die Gruppengröße k gesteuert. Ein größeres k führt zu einem höheren Selektionsdruck, da die Wahrscheinlichkeit steigt, dass die "wirklich" beste Lösung der k zufällig ausgewählten auch tatsächlich die beste der gesamten Population ist und somit der Selektionsdruck steigt.
-

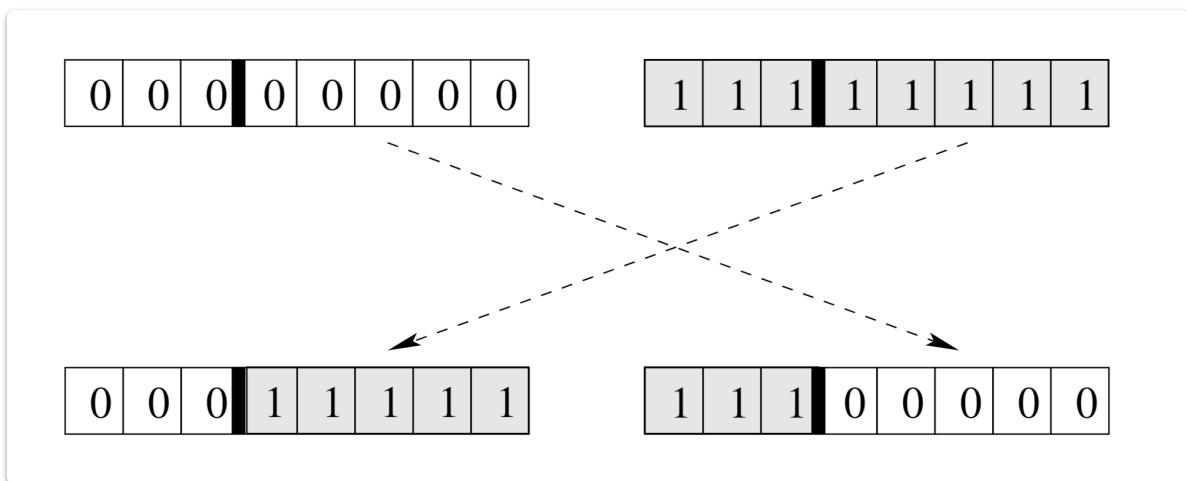
Rekombination

Rekombination: Aus zwei (oder mehreren) Elternlösungen wird eine neue Lösung abgeleitet.

Vererbung: Die neue Lösung sollte möglichst ausschließlich aus den Eigenschaften (Bestandteilen) der Eltern aufgebaut werden.

Die Rekombination ist meist eine zufallsbasierte, einfach gehaltene und schnelle Operation.

Beispiel für Bitstrings: 1-point crossover



Erklärung des 1-point crossover:

Beim 1-point crossover wird ein zufälliger "Schnittpunkt" innerhalb der Bitstrings der beiden Elternlösungen festgelegt. Alle Bits vor diesem Schnittpunkt stammen vom ersten Elternteil, und alle Bits nach dem Schnittpunkt stammen vom zweiten Elternteil, oder umgekehrt. Dies führt zur Erzeugung von zwei neuen Kindlösungen.

Mutation

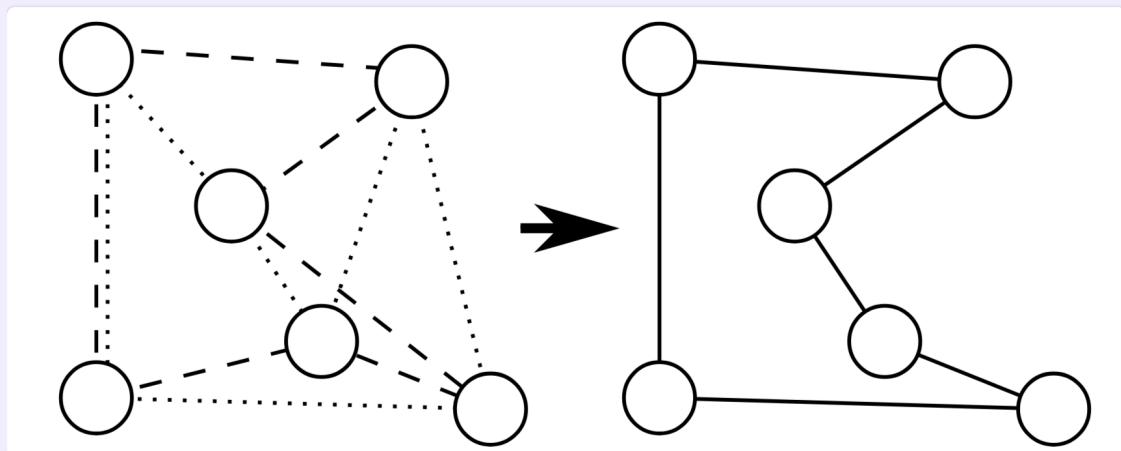
Möglichkeiten der Mutation:

- Für Bitstrings (eine Reihe von 0en und 1en) z.B. **Flip eines jeden Bits** mit kleiner Wahrscheinlichkeit. Das bedeutet, eine 0 wird zu einer 1 und eine 1 zu einer 0.
- Allgemein meist ein oder mehrere **zufällige Moves in einer sinnvollen Nachbarschaft**. Das bedeutet, dass eine kleine, zufällige Änderung an der Lösung vorgenommen wird, die aber immer noch im Bereich "sinnvoller" Lösungen liegt.

Hinweis: Vergleiche Random-Neighbor-Schrittfunktion der lokalen Suche. (Die Mutation ähnelt dem Prozess, bei dem in der lokalen Suche zufällig ein Nachbar einer aktuellen Lösung als nächste potenzielle Lösung ausgewählt wird.)

☰ Beispiel: Evolutionärer Algorithmus für TSP

Edge-Recombination: Eine neue TSP-Tour wird zufallsgesteuert möglichst nur aus Kanten aufgebaut, die bereits in zwei Elternlösungen vorkommen.



Mutation: Typischerweise ein zufälliger Move in einer klassischen Nachbarschaft wie z.B. 2-opt oder Verschieben eines Knotens an eine andere Position.

☰ Beispiel: Edge-Recombination für das TSP

Eingabe: Zwei gültige Touren T^1 und T^2 .

Ausgabe: Neue abgeleitete Tour T .

Variablen: Aktueller Knoten v , Nachfolgeknoten w , Kandidatenmenge für Nachfolgeknoten W .

Edge-Recombination(T^1, T^2):

Beginne bei einem beliebigen Startknoten $v \leftarrow v_0$, $T \leftarrow \{\}$

while es existieren noch unbesuchte Knoten

Sei W Menge noch unbesuchten Knoten, welche in

$T^1 \cup T^2$ adjazent zu v sind

if $W \neq \{\}$

Wähle einen Nachfolgeknoten $w \in W$ zufällig aus

else

Wähle einen zufälligen noch nicht besuchten Nachfolgeknoten w

$T \leftarrow T \cup \{(v, w)\}$, $v \leftarrow w$

Schließe die Tour: $T \leftarrow T \cup \{(v, v_0)\}$

Fazit zu evolutionären Algorithmen

- **Grundprinzip leicht umsetzbar.**
- **Häufig ist eine Kombination mit anderen Methoden sinnvoll:**
 - Ausgangslösungen mit Konstruktionsheuristiken erzeugen.
 - Problemspezifisches Wissen in Rekombination und Mutation ausnutzen.
 - Neue Kandidatenlösungen mit lokaler Suche etc. versuchen zu verbessern.
- Die **Lösungsgüte** und die **Laufzeit** hängen sehr stark von den konkreten Operatoren (z.B. der Art der Selektion, Rekombination und Mutation) ab.
- **Parallelisierung ist gut möglich**, da viele Operationen (z.B. Fitnessbewertung der Population, Generierung neuer Lösungen) unabhängig voneinander durchgeführt werden können.