

Algorithmik für Schwere Probleme

thgoebel@ethz.ch

ETH Zürich, FS 2021

This documents is a **short** summary for the course *Algorithmik für Schwere Probleme* at ETH Zurich. It is intended as a document for quick lookup, e.g. during revision, and as such does not replace attending the lecture, reading the slides or reading a proper book.

We do not guarantee correctness or completeness, nor is this document endorsed by the lecturers. Feel free to point out any errata, either by mail or on Github.

Contents

1	Einführung	3
1.1	Definitionen	4
2	Pseudopolynomielle Algorithmen	6
2.1	Rucksackproblem (Knapsack problem, KP)	7
3	Parametrisierte Algorithmen	8
3.1	Kernbildung	9
3.1.1	Kronenzerlegung	10
3.2	Tiefenbeschränkte Suchbäume	11
3.3	Iterative Kompression	11
3.4	Dynamische Programmierung für den Steinerbaum	12
3.5	Baumzerlegung	13
3.5.1	Beispiel: VCP parametrisiert mit Baumweite	14
3.6	Grenzen der Parametrisierbarkeit	16
4	Exakte Exponentialzeit Algorithmen	18
4.1	Branching-Algorithmen	18
4.1.1	Beispiel: 3-SAT	18
4.1.2	Beispiel: Max-IS	19
4.2	Dynamische Programmierung	20
4.3	Measure-and-Conquer	21
5	Approximations-Algorithmen	24
5.1	Nichtapproximierbarkeit	27
5.2	Approximationsschemata	27

1 Einführung

Konzepte

- NP-schwer vs. NP-vollständig
- Entscheidungsproblem
- Schwellwertsprache
- Diverse Notationen, Definitionen und Beispiele

Polynomzeit-Reduzierbarkeit Ein Entscheidungsproblem Π_1 ist “polynomzeit-reduzierbar” auf ein anderes Entscheidungsproblem Π_2 :

$$\begin{aligned} &\iff \exists \text{ Algo } \mathcal{A} \text{ s.t. } \text{time}_{\mathcal{A}} \in \text{poly} \wedge \Pi_1(x) = \Pi_2(\mathcal{A}(x)) \\ &\iff \Pi_2 \text{ mindestens so schwer wie } \Pi_1 \\ &\iff \Pi_1 \text{ höchstens so schwer wie } \Pi_2 \\ &\iff \Pi_1 \preceq_P \Pi_2 \end{aligned}$$

Klasse NP Nichtdeterministisch-Polynom-Zeit. Klasse der Probleme, die sich in Polynomzeit verifizieren lassen, und von einer nichtdeterministischen Turing-Maschine in Polynomzeit lösen lassen (z.B. durch Lösung raten und verifizieren).

NP-schwer (NP-hard) Ein Problem Π das “mindestens so schwer” ist wie alle Probleme in NP. D.h. alle Probleme in NP lassen sich auf Π reduzieren:

$$\forall \Pi' \in NP : \Pi' \preceq_P \Pi$$

Π muss nicht notwendigerweise in NP liegen (d.h. kann schwerer sein)! Beispiel: das Halteproblem (nicht entscheidbar, daher $\notin NP$).

NP-vollständig (NP-complete) Ein Problem Π , das in NP liegt und NP-schwer ist. “Repräsentativ” für die Menge NP, da sich alle Probleme aus NP darauf reduzieren lassen.
Beispiel: Satisfiability-Problem SAT (Satz von Cook).

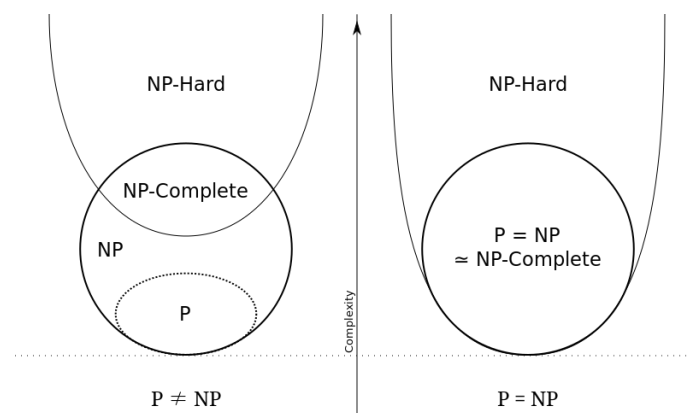


Figure 1: Mengendiagramm der Beziehungen (Quelle: Wikipedia)

“Schwere Probleme” meist NP-schwere Probleme, aber generell alle Probleme die sich nicht in Polynomzeit lösen lassen. **Sinnvollerweise gehen wir im Folgenden davon aus dass $P \neq NP$.**

Allgemein sind Instanzen unseres Problems nicht deterministisch in Polynomzeit lösbar¹. Mögliche Ansätze:

- a) nicht exakt sondern approximativ lösen (Approximationsalgorithmen)
- b) nicht deterministisch sondern nichtdeterministisch lösen (Randomisierte Algorithmen)
- c) nicht polynomiell sondern moderat exponentiell lösen²
- d) nicht alle sondern Instanzen mit einer bestimmten Struktur lösen (Parametrisierte Algorithmen)
- e) anderweitig zusätzliche Informationen über die Eingabe nutzen (Reoptimierung, Win-Win-Strategy)
- f) Heuristiken³

1.1 Definitionen

Entscheidungsproblem $P = (L, U, \Sigma)$ wobei

- Σ ein Alphabet
- $U \subseteq \Sigma^*$ die Menge der zulässigen Eingaben (als Wörter über dem Alphabet, als *Sprache*)
- $L \subseteq U$ die Menge der akzeptierten Eingaben (*JA-Instanzen*)

Ein Algorithmus \mathcal{A} *löst* P falls gilt:

$$\forall u \in U : A(x) = \begin{cases} 1 \text{ oder JA, if } x \in L \\ 0 \text{ oder NEIN, if } x \in U - L \end{cases}$$

Vertex Cover Problem VC “Der Hefepilz der parametrisierten Algorithmiker – ein Modellproblem.”

Eingabe U : ungerichteter Graph $G = (V, E)$ und $k \leq |V|, k \in \mathbb{N}$.

Ausgabe L : JA falls $\exists C \subseteq V$ s.t. $|C| \leq k$ mit $\forall \{u, v\} \in E : u \in C \vee v \in C$.

In Worten: Entscheide ob es ein Vertex Cover der Grösse $\leq k$ für einen gegebenen Graphen G gibt.

Satisfiability-Problem SAT

Eingabe: CNF-Formel $\Phi = C_1 \wedge \dots \wedge C_m$ mit Klauseln C_i über Variablen x_1, \dots, x_n .

Ausgabe: eine Variablen-Belegung die Φ erfüllt.

Bei l -SAT enthält jede Klausel maximal l Literale.

¹Es kann sein, dass einige instanzen in polynomzeit lösbar sind, siehe z.B. *Win-Win-Algorithmen*.

²D.h. die Basis der Exponentiation ist klein, z.B. 1.4^n statt 2^n .

³Nachteil: Im Gegensatz zu den anderen Ansätzen ist hier die Qualität (Laufzeit, ...) nicht beweisbar.

Optimierungsproblem $U = (L, M, cost, goal)$ wobei

- L die Sprache der zulässigen Eingaben⁴
- $M : L \mapsto \Sigma^*$ so dass $M(x)$ die Sprache der akzeptierten Lösungen für Eingabe x
- $cost: \forall x \in L \forall y \in M(x) : cost(y, x) = \text{Kosten der Lösung } y \text{ für Eingabe } x$
- $goal \in \{\min, \max\}$ das Optimierungsziel
- $Opt_U(x) = goal\{cost(y, x) | y \in M(x)\}$ die Kosten einer optimalen Lösung für Eingabe x

Minimum Vertex Cover Problem MIN-VC Wie VC, mit $cost(C, G) = |C| = \text{Grösse des Vertex Covers}$ und $goal = \min$.

MAX-SAT Wie SAT, mit $cost = \text{Anzahl belegte Variablen}$ und $goal = \max$.

Laufzeit eines Algorithmus' \mathcal{A} auf Eingabe x ist $\text{time}_{\mathcal{A}}(x)$ wobei $\text{time}_{\mathcal{A}} : \mathbb{N} \mapsto \mathbb{N}$. Die Laufzeit von \mathcal{A} in Abhängigkeit von der Grösse n der Eingabe ist: $\text{time}_{\mathcal{A}}(n) = \max\{\text{time}_{\mathcal{A}}(x) \mid |x| = n, x \in L\}$. Die Laufzeit wird in \mathcal{O} -Notation angegeben.

Schwellwertsprache (threshold language) definiert für ein Optimierungsproblem U als die Sprache die für jede legale Eingabe von U alle Tupel enthält mit der Eingabe selber und einer Zahl (in Binär) die grösser gleich der Kosten der optimalen Lösung ist:

$$\begin{aligned} \text{Lang}_U &= \{(x, a) \in L \times \{0, 1\}^* \mid \text{Opt}_U(x) \leq \text{Number}(a)\} \\ \text{Lang}_U &= \{(x, a) \in L \times \{0, 1\}^* \mid \text{Opt}_U(x) \geq \text{Number}(a)\} \end{aligned}$$

wo $\text{Number}(a)$ die Zahl mit der Binärdarstellung a (der *Schwellwert*) ist und jeweils $goal = \min / \max$.

Beispiel: $\text{Lang}_{\text{MIN-VC}}$ enthält für einen Eingabe x die optimal mit einem VC der grösser k alle Tupel (x, k') mit $k' \geq k$. Somit gilt z.B. auch, dass $\text{Lang}_{\text{MIN-VC}} = \text{VC}$, weil wir die Entscheidung des VC Problems herausfinden können, indem wir nach dem entsprechenden Tupel in $\text{Lang}_{\text{MIN-VC}}$ suchen. Anders herum können wir heraus finden ob ein Tupel $(x, a) \in \text{Lang}_{\text{MIN-VC}}$ indem wir VC mit $k = a$ aufrufen.

Achtung, diese Relation gilt nicht für alle Probleme, z.B. $\text{Lang}_{\text{MAX-SAT}} \neq \text{SAT}$ (da SAT leichter sein kann)!

U heisst "NP-schwer" falls Lang_U NP-schwer ist (warum?).⁵

Abschätzen von Anzahl Teilmengen Frage: wie viele Teilmengen $T \subseteq \{1, \dots, n\}$ mit $|T| \leq k$ gibt es? Grobe Abschätzung:

$$\sum_{0 \leq i \leq k} \binom{n}{i} \stackrel{\star}{\leq} \sum_{0 \leq i \leq k} n^i \stackrel{\star\star}{\leq} \frac{n^{k+1} - 1}{n - 1} \in \mathcal{O}(n^k)$$

Wobei \star für kleine k okay, aber grosse k ungenau ist, und $\star\star$ eine geometrische Summe.

Da die Teilmengen in lexikographischer Ordnung durchlaufen werden können, lassen sie sich in amortisiert $\mathcal{O}(n^k)$ enumerieren.

⁴Oben noch U !

⁵Wir erinnern uns, dass NP-Schwere für Entscheidungsprobleme definiert ist. Über die Schwellwertsprache erweitern wir dies für Optimierungsprobleme, indem wir aus dem Optimierungsproblem ein Entscheidungsproblem konstruieren.

2 Pseudopolynomielle Algorithmen

Konzepte

- Pseudopolynomialität
- Stark NP-schwer

Zahlproblem (integer value problem IVP) Probleme, dessen Eingaben vollständig durch Integer darstellbar sind.

Eingabe $x = x_1 \# \dots \# x_n$; $x_i \in \{0, 1\}^*$, interpretiert als Vektor $\text{int}(x) = (\text{number}(x_1), \dots, \text{number}(x_n))$.

Weiter sei

$$\text{max-int}(x) = \max_{i \in \{1, \dots, n\}} \text{number}(x_i)$$

die grösste vorkommende Zahl (im Wert, nicht in der Darstellung). Da die Eingabe binär definiert ist, kann $\text{Max} - \text{Int}(x)$ exponentiell in $|x|$ sein.

Beispiel: Travelling Salesperson Problem TSP (via Adjazenzmatrix des Graphen).

Pseudopolynomiell Sei U ein Zahlproblem und \mathcal{A} ein Algorithmus der U löst. \mathcal{A} heisst *pseudopolynomiell* falls für alle Eingaben x ein Polynom p existiert, so dass

$$\text{time}_{\mathcal{A}}(x) \in \mathcal{O}\left(p(|x|, \text{max-int}(x))\right)$$

D.h. auf Eingaben mit “kleinen Zahlen” ist \mathcal{A} polynomiell.

h-beschränktes Teilproblem Sei U ein Zahlproblem, $h : \mathbb{N} \mapsto \mathbb{N}$ monoton nicht-fallend. Das *h-beschränkte Teilproblem von U* (*h-value-bounded subproblem*) $\text{value}(h)$ - U ist das Teilproblem mit Eingaben I für die gilt: $\text{max-int}(I) \leq h(|I|)$.

Stark NP-schwer Ein Zahlproblem U heisst *stark NP-schwer* falls ein Polynom p existiert, so dass $\text{value}(p)$ - U NP-schwer ist.

In anderen Worten: U ist stark NP-schwer wenn es auch dann NP-schwer ist wenn alle darin vorkommenden Zahlen “klein” sind.

Beispiel: TSP. Generell jedes gewichtete Graph-Optimierungsproblem wenn das ungewichtete Pendant NP-schwer ist (hier: Hamiltonkreisproblem HCP).

Theorem Sei U stark NP-schwer. Dann existiert kein pseudopolynomieller Algorithmus für U .⁶

Beweis Ansatz: Wir nehmen an, dass ein pseudo poly. Algorithmus \mathcal{A} für U existiert. Über die Definition von Stark NP-schwer können wir dann zeigen, dass wir ein polynomiellen Algorithmus für $\text{value}(p)$ - U hätten, was ein Widerspruch ist.

⁶Wie immer unter der Annahme dass $P \neq NP$.

2.1 Rucksackproblem (Knapsack problem, KP)

Rucksackproblem Wie viel Diebesgut kann ich mitnehmen.

Eingabe I : Gewichte $w_i \in \mathbb{N}^+$, Kosten/Nutzen $c_i \in \mathbb{N}^+$, Limit/Kapazität $b \in \mathbb{N}^+$, wo $i \in \{1, \dots, n\}$.

Ausgabe: Indexmenge $T \subseteq \{1, \dots, n\}$ s.t. $\sum_{i \in T} w_i \leq b$

Kosten: $\text{cost}(T, I) = \sum_{i \in T} c_i$

Ziel: max

Lösung mit Dynamischer Programmierung (Dyn-Prog-KP) Wir iterieren über alle Teilprobleme I_i und speichern die Tripel $(k, W_{i,k}, T_{i,k}) = (\text{Nutzen}, \text{Gewicht}, \text{Indexmenge})$, also Mengen $T_{i,k}$ die exakt Nutzen k mit minimalen Gewicht $W_{i,k}$ erreichen. In jeder Iteration behalte für jeden vorhandenen Nutzen ein Tripel mit minimalen Gewicht. Lese am Ende den maximal erreichten Nutzen k^* (und sein T_{n,k^*}) aus.

Laufzeit

$$\mathcal{O}(|I|^2 \cdot \max\text{-int}(I))$$

da der Algorithmus $|I|$ Iterationen durchläuft und jeder Schritt in $\leq \sum_j^n c_j = |I| \cdot \max\text{-int}(I)$ läuft.

Verbesserung falls $b \ll \sum_{j=1}^n c_j$: Wir können den Algorithmus für Instanzen verbessern, bei denen die Gewichtsschranke kleiner ist als die Summe aller Nutzen.

Im oben stehenden Algorithmus speichern wir uns ein Tripel für jeden möglichen Nutzen (es gibt $\sum_{j=1}^n c_j$ solche Werte) das minimal Gewicht (und die jeweiligen Gegenstände), das diesen Nutzen erreicht. Da $b \ll \sum_{j=1}^n c_j$ können wir die Logik des Algorithmus “umkehren” und stattdessen für jedes mögliche Gewicht den maximal erreichbaren Nutzen speichern.

Die neue Laufzeit des neuen Algorithmus ist in $\mathcal{O}(n \cdot b)$, wobei $n \in \mathcal{O}(|I|)$ und $b \in \mathcal{O}(\max\text{-int}(I))$.

Weitere Verbesserung Wir können die zwei Ansätze kombinieren für Probleme bei denen $b \gg \sum_{j=1}^n c_j$, wobei allerdings nur wenig Objekte die Summe stark beeinflussen. Also wir nehmen an es gibt ein $i \ll n$, sodass $b \gg \sum_{j=1}^{n-i} c_j$ und $b \ll \sum_{j=i}^n c_j$.

Hierfür unterteilen wir das Problem in zwei Teilprobleme, eines, was nur die “leichten” Objekte enthält und eines welches nur die “schwere” Objekte beinhaltet. Wir rufen den ursprünglichen Algorithmus auf der “leichten Instanz”⁷ auf und die oben beschriebene Verbesserung auf die “schwere Instanz”. Wir können die beiden Lösungen nun kombinieren, indem wir für jede Lösung des einen Problems, die best mögliche Lösung aus dem anderen Problem suchen. Die Lösung ist die Kombination mit dem höchsten Gesamtnutzen.

Die Laufzeit des Algorithmus ist in

$$\mathcal{O}\left(n \cdot \sum_{j=1}^{n-i} c_j + i \cdot b\right).$$

⁷Leichte Instanz im Sinne von Gewicht, nicht Komplexität

3 Parametrisierte Algorithmen

Konzepte

- Parametrisierter Algorithmus, fpt
- Standard-Parametrisierung
- Kernbildung
- (Sichere) Datenreduktionsregel
- Kronenzerlegung
- Tiefenbeschränkte Suchbäume
- Iterative Kompression
- Baumzerlegung, Baumweite; bramble, bramble number
- Klasse $W[1]$
- Probleme: Edge Clique Cover, Cluster Editing, Steinerbaum, VCP via Baumweite

Idee Verallgemeinerung von pseudopolynomiellen Algorithmen: Laufzeit hängt von der Eingabegrösse nur polynomiell ab, aber darf extrem gross werden in der Grösse eines Parameters. Partitionierung in Problemklassen entlang des Parameters.

Parametrisierung Sei U ein Entscheidungsproblem, L die Sprache der Eingaben. $\text{par} : L \mapsto \mathbb{N}$ ist eine *Parametrisierung* von U falls gilt:

- (i) par ist in Polynomzeit berechenbar
- (ii) Für unendlich viele $k \in \mathbb{N}$ ist die Parameter- k -Menge für U

$$\text{Set}_U(k) := \{I \in L \mid \text{par}(I) = k\}$$

unendlich. Dies stellt sich dass par nichttrivial ist, z.B. nicht etwa $\text{par}(I) = |I|$. In anderen Worten, für jeden Parameterwert soll es beliebig viele Probleminstanzen geben.

Ein Algorithmus \mathcal{A} heisst *par-parametrisierter Polynomzeit-Algorithmus* für U falls gilt:

- (i) \mathcal{A} löst U
- (ii) \exists Polynom $p \exists$ (berechenbare) Funktion $f : \mathbb{N} \mapsto \mathbb{N}$ so dass $\forall I \in L$:

$$\text{time}_{\mathcal{A}}(I) \leq f(\text{par}(I)) \cdot p(|I|)$$

Dann heisst U *fixed-parameter-tractable bezüglich par*. \mathcal{A} heisst auch *fpt-Algorithmus für U bezüglich par* und läuft in fpt-Zeit.

Beispiel: Sei U ein Zahlproblem und sei $\text{Val}(I) := \max\{|x_i|\}$. Es gilt $\text{Max} - \text{Int}(I) \leq 2^{\text{Val}(I)}$ und ist Val eine Parametrisierung von U .

Ansätze Die *Standard-Parametrisierung* wählt als Parameter die Grösse der gewünschten Lösung (z.B. Grösse des VCs). Die *Strukturelle Parametrisierung* wählt eine bestimmte Eigenschaft der Eingabe (z.B. maximaler Knotengrad). Andere Ansätze betrachten wir im Folgenden.

3.1 Kernbildung

Idee Polynomielles Preprocessing, Datenreduktion, um die Instanz auf einen *Kern* zu verkleinern, der in seiner Grösse nur noch vom Parameter abhängt. Diesen Kern dann mit bekannten Algorithmen (oder brute force) lösen.

Kernbildung (kernelisation) Sei (U, par) ein parametrisiertes Entscheidungsproblem, L die Sprache der JA-Instanzen von U . Ein *Kernbildungs-Algorithmus* für (U, par) ist ein Polynomzeit-Algorithmus \mathcal{A} der jede Eingabe (I, k) in eine neue Eingabe (I', k') (den *Kern (kernel)*) transformiert so dass:

- (i) $I \in L \iff I' \in L$ (Korrektheit)
- (ii) $|I'| + k' \leq g(k)$ für $g : \mathbb{N} \mapsto \mathbb{N}$ (neue Grösse nur abhängig vom alten Parameterwert k)

Eine *Reduktionsregel* von \mathcal{A} heisst *sicher*, wenn sie (i) erfüllt.

Beispiel: VCP

Reduktionsregel: Sei $S \subseteq V$ ein VC von G so dass $|S| \leq k$. Dann enthält S alle Knoten mit $\deg_G(v) > k$ (warum?). Dann gilt für ein solches v :

$$(G, k) \in VCP \iff (G - \{v\}, k - 1) \in VCP$$

Zusätzlich entferne alle isolierten Knoten mit $\deg_G(v) = 0$ (sie decken keine Kanten ab).

Zu zeigen: wenn die Regel nicht mehr anwendbar ist, dann hat der verbleibende Graph G' entweder kein vertex cover, oder aber er ist “klein genug” (nur noch anhängig von k), also ein Kern.

Beobachtung: Sei G ein Graph ohne isolierten Knoten, mit einem vertex cover der Grösse m und mit $\max \deg_G(v) \leq k$. Dann gilt $|V| \leq m \cdot (k + 1)$ (warum?).⁸

Theorem: Diese Reduktionsregeln berechnen einen Kern der Grösse $\mathcal{O}(k^2)$ in Zeit⁹ $\mathcal{O}(k \cdot n)$.

Parametrisierter Algorithmus: Berechne einen Kern (G', k') . Wenn der Kern zu gross ist, geben NEIN aus. Andernfalls prüfe durch vollständige Suche ob ein VC mit $|S'| \leq k'$ existiert.

Theorem: Dies ist ein fpt-Algorithmus bzgl. der Standard-Parametrisierung. Die Laufzeit beträgt $\mathcal{O}(k \cdot n + k^{2k+2}) \subseteq \mathcal{O}(k^{2k+2} \cdot n)$.

Theorem Sei (U, par) ein parametrisiertes Entscheidungsproblem.

Ein fpt-Algorithmus für (U, par) existiert \iff Ein Kernbildungsalgorithmus für (U, par) existiert.

Edge Clique Cover Problem ECCP

Eingabe: ungerichteter Graph $G = (V, E)$, $k \leq \mathbb{N}$

Ausgabe: JA falls k Cliques¹⁰ C_1, \dots, C_k existieren, so dass $E = \bigcup_{i=1}^k E(C_i)$ (jede Kante ist in mind. einer Clique), sonst NEIN.

Reduktionsregeln:

- (i) $(G, k) \xrightarrow{\text{isolierter Knoten } u} (G - \{u\}, k)$

⁸Falls $|V| > k \cdot (k + 1)$, dann wir wissen dass $I \notin L$ und können NEIN ausgeben.

⁹Unter der Annahme dass die Knotengrade bereits gegeben sind, ansonsten $\mathcal{O}(k \cdot n + m)$.

¹⁰Clique: Knotenmenge so dass alle Knoten paarweise miteinander verbunden sind, d.h. einen vollständigen Teilgraphen bilden.

(ii) $(G, k) \xrightarrow{\text{isolierte Kante } e=\{u,v\}} (G - \{u, v\}, k - 1)$

(iii) $(G, k) \xrightarrow{\{u,v\} \in E \text{ s.t. } \{u,v\} \subsetneq N[u]=N[v]} (G - \{u\}, k)$

$N[u]$ ist die *geschlossene Nachbarschaft* von u , also alle benachbarten Knoten und u selbst.

Theorem: Das ECCP hat einen Kern mit $\leq 2^k$ Knoten.

3.1.1 Kronenzerlegung

Ziel: statt einem quadratischen Kern suchen wir einen linearen Kern für das VCP.

Bisher genutzte Strukturen zur Reduktion: hoher Knotengrad, gleiche geschlossene Nachbarschaft.

Kronenzerlegung (crown decomposition) Die *Kronenzerlegung* von $G = (V, E)$ ist eine Partitionierung $V = C \cup H \cup B$ so dass:

(i) $C \neq \emptyset$ ist ein independent set¹¹ in G

(ii) $N(C) = H$ (keine Kanten zwischen C und B)

(iii) Die Kanten zwischen C und H enthalten ein Matching M mit $|M| = |H|$ (M *saturiert* H).

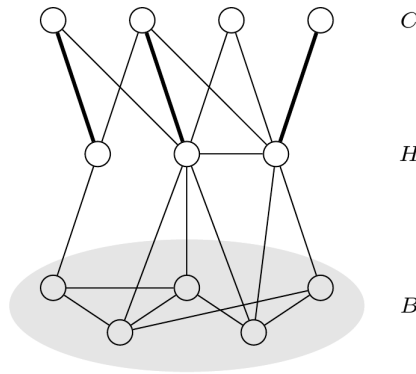


Figure 2: Kronenzerlegung: crown, head, body (Quelle: Vorlesung)

Satz von König Sei G ein ungerichteter bipartiter Graph, sei M ein Matching maximaler Kardinalität, sei S ein vertex cover minimaler Kardinalität. Dann gilt $|M| = |S|$.

Theorem: M und S können in Polynomzeit berechnet werden.

Lemma Sei G ungerichtet, ohne isolierte Knoten, mit $|V| \geq 3k + 1$. Dann existiert ein Polynomzeitalgorithmus der

(i) entweder eine Kronenzerlegung berechnet

(ii) oder ein Matching M mit $|M| \geq k + 1$ findet.

Beweis siehe Buch, Kapitel 6.2, Seite 145f.

¹¹Eine Menge von Knoten zwischen denen es keine Kante gibt.

Reduktion für VCP

Lemma: Sei G ein Graph mit Kronenzerlegung $V = C \cup H \cup B$ und sei $k \in \mathbb{N}$. Dann gilt:

$$(G, k) \in VCP \iff (G - (C \cup H), k - |H|) \in VCP$$

Theorem: Ein Kernbildungsalgorithmus mit obiger Reduktionsregel findet einen Kern mit $\leq 3k$ Knoten.

3.2 Tiefenbeschränkte Suchbäume

Idee Vollständige Suche über alle möglichen Lösungen, aber mit Suchbaumtiefe beschränkt im Parameter.

Beispiel: VCP Beobachtung: in jedem vertex cover S gilt $\forall e = \{v_1, v_2\} : v_1 \in S \vee v_2 \in S$. Daher verzweige von (G, k) nach $(G - \{v_1\}, k - 1)$ und $(G - \{v_2\}, k - 1)$. In den Blättern $(G_j, 1)$ ist das VCP trivial. Dieser Suchbaum hat Tiefe k und 2^{k-1} Blätter. Gesamtlaufzeit: $\mathcal{O}(2^k \cdot n)$.

Cluster Editing Problem CEP

Eingabe: Graph $G = (V, E)$, $k \in \mathbb{N}$

Ausgabe: JA falls durch Löschen/Einfügen von $\leq k$ Kanten G in eine Vereinigung (union) von disjunkten Cliques (= jeder connected component ist eine Clique) transformiert werden kann, sonst NEIN. Bekannterweise NP-schwer.

Theorem: Graph G besteht aus disjunkten Cliques

$\iff G$ enthält *keinen* Pfad aus drei Knoten¹²

\iff es gibt *keine* paarweise verschiedenen $u, v, w \in V : \{u, v\}, \{v, w\} \in E \wedge \{u, w\} \notin E$

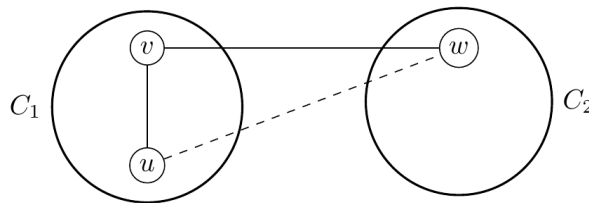


Figure 3: Cluster Editing Situation (Quelle: Vorlesung)

Algorithmus: Teste ob G bereits eine Vereinigung disjunkter Cliques ist, oder ob $k = 0$. Wenn nicht, finde $u, v, w \in V$ mit obiger Bedingung. Rekursiere auf $G_1 = (V, E - \{\{u, v\}\})$, $G_2 = (V, E - \{\{v, w\}\})$, $G_3 = (V, E \cup \{\{u, w\}\})$. Laufzeit¹³: $\mathcal{O}(n^3 \cdot 3^k)$.

3.3 Iterative Kompression

Idee Gegeben ein Kompressionsalgorithmus der aus einer (bzgl. dem Schwellwert) etwas zu grossen Lösung eine kleinere, gültige Lösung in fpt-Zeit berechnet. Iteriere: Instanz vergrössern, komprimieren, usw.:

¹²In eine Clique bilden drei Knoten einen Kreis, keinen Pfad.

¹³Die Rekursionsformel $T(k) = 3 \cdot T(k - 1)$ gibt $\mathcal{O}(3^k)$ rekursive Aufrufe.

$$\begin{array}{l}
(I_0, k_0) \text{ mit Lösung } S_0^*, |S_0^*| \leq k_0 \\
\begin{array}{c} \text{Instanz vergrößern} \\ \longrightarrow \end{array} (I_1, k_1) \text{ mit Lösung } S_1, |S_1| > k_1 \\
\begin{array}{c} \text{komprimieren} \\ \longrightarrow \end{array} (I_1, k_1) \text{ mit Lösung } S_1^*, |S_1^*| \leq k_1 \\
\vdots
\end{array}$$

Disjoint Vertex Cover Problem DVCP

Eingabe: $G = (V, E)$, vertex cover W , $k \in \mathbb{N}$

Ausgabe: JA falls es ein vertex cover S gibt mit $|S| \leq k \wedge S \cap W = \emptyset$ sonst NEIN.

Lemma: Jede DVCP-Instanz (G, W, k) kann in Zeit $\mathcal{O}(|V|^2)$ gelöst werden (warum?¹⁴).

Beispiel: VCP Idee: füge einen Knoten nach dem anderen hinzu. Starte mit $S_1^* = \emptyset$. In jeder Iteration setze $S_i \leftarrow S_{i-1}^* \cup \{v_i\}$ und benutze $\mathcal{A}_{compress}(G_i, S_i, k)$ um entweder S_i^* zu finden oder NEIN. Details siehe Buch, Algorithmen 6.6+6.7, Seite 151ff.

Kompression: Idee: probiere alle möglichen intersections X des gesuchten VCs mit S aus:

If $|S| \leq k$ return S . For all $X \subseteq S$ with $|X| \leq k$ do: solve DVCP on $(G - X, S - X, k - |X|)$ – if YES return $X \cup N(S - X)$.

Laufzeit: $\mathcal{O}(2^k \cdot n^2)$ also gesamt: $\mathcal{O}(2^k \cdot n^3)$.

3.4 Dynamische Programmierung für den Steinerbaum

Steinerbaum-Problem STP

Eingabe: $G = (V, E)$ mit Kantenkosten $c : E \mapsto \mathbb{N}$, *Terminalen* $S \subseteq V$, *Nicht-Terminalen/Steiner-Knoten* $N = V - S$

Lösungen: Teilbaum T von G (“Steinerbaum”) der alle Knoten aus S enthält (und einige aus N)

Kosten: $\text{cost}(T, (G, c, S)) = c(T) = \sum_{e \in E(T)} c(e) = \text{Summe der Kanten} = \text{“Grösse” des Baums}$

Ziel: min

DP-Ansatz Annahme: alle Blätter sind Terminale (warum?).

Beobachtung: falls $Y \subseteq N$ gegeben ist dann ist das STP einfach: berechne MST von $S \cup Y$.

\implies k-parametrisierter Algorithmus mit $k = |N|$.

Uninteressant, wir schauen im Folgenden ein DP mit dem Parameter $k = |S|$ an. Für alle $X \subseteq S$ und alle $v \in V - X$ berechne zwei Steinerbäume:

1. Auf Instanz (G, c, X) einen Steinerbaum mit minimalen Kosten $g(X)$.
2. Auf Instanz $(G, c, X \cup \{v\})$ einen Steinerbaum mit minimalen Kosten $g_{in}(X, v)$ so dass v ein innerer Knoten ist (d.h. $\deg(v) \geq 2$).

¹⁴Es reicht zu prüfen ob $N(W)$ ein vertex cover ist

Lemma Es gilt:

$$g_{in}(X, v) = \min_{\emptyset \neq X' \subset X} \{g(X' \cup \{v\}) + g((X - X') \cup \{v\})\} \quad (1)$$

$$g(X \cup \{v\}) = \min \left\{ \min_{w \in X} \{p(v, w) + g(X)\}, \min_{w \in V - X} \{p(v, w) + g_{in}(X, w)\} \right\} \quad (2)$$

wobei $p(v, w)$ die minimalen Kosten eines Pfades von v nach w in G sind. Beweis siehe Buch Kapitel 6.5, Seite 159ff.

Dreyfus-Wagner Algorithmus Eingabe (G, c, S) . Berechne $p(v, w)$ für alle $v, w \in V$. Initialisiere $g(\{x, y\}) := p(x, y)$ für alle $x, y \in S$. Für alle $X \subseteq S$ mit $|X| \in [2, |S| - 1]$ und alle $v \in V - X$ berechne $g_{in}(X, v)$ und $g(X \cup \{v\})$. Gebe $g(S)$ aus.

Laufzeit: $\mathcal{O}(n^2 \log n + n \cdot m + (k^2 +) 3^k \cdot n + 2^k \cdot n^2)$

3.5 Baumzerlegung

Motivation Viele schwere Graphprobleme sind auf Bäumen einfach. Z.B. Reduktionsregel für VCP: entferne Blätter und ihre Nachbarn, füge Nachbarn ins VC hinzu.

Ziel: “Baumähnlichkeit” von Graphen beschreiben und “Baum-Algorithmenvariante” anwenden.

Definition (Baumzerlegung) Eine *Baumzerlegung* (tree decomposition) von G ist ein Paar $D = (T, B)$ wobei $T = (V_T, E_T)$ ein Baum ist. Sei I eine Indexmenge, die die Knoten aus V_T aufzählt.¹⁵ Sei B eine Label-Funktion $B : I \mapsto 2^V$ die jedem Index i von V_T eine Knotenmenge $X_i \subseteq V$ (einen *bag*) zuordnet, so dass:

- (i) $\bigcup_{i \in I} X_i = V$ (alle Knoten kommen vor)
- (ii) $\forall \{u, v\} \in E \exists i \in I : u, v \in X_i$ (alle Kanten in einem bag)
- (iii) $\forall v \in V$ bilden die bags X_i mit $v \in X_i$ einen Teilbaum von T (Zusammenhang, Lokalität)

Die *Weite* von D ist $\max\{|X_i|\} - 1$.¹⁶

Die *Baumweite* $\text{tw}(G)$ von G ist die minimale Weite über alle möglichen Baumzerlegungen.

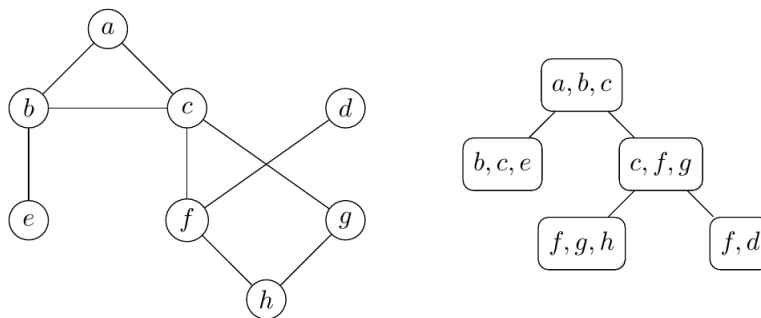


Figure 4: Graph G (links) und seine Baumzerlegung D (rechts) (Quelle: Vorlesung)

¹⁵Dies erlaubt es uns getrennt über die Knotenmenge V_T des Baums T und über die bags zu sprechen.

¹⁶Das -1 ist kosmetisch damit echte Bäume eine Baumweite von 1 haben.

Beobachtungen

- Sei G ein Baum. Dann ist $\text{tw}(G) = 1$.
- Sei G ein Kreis. Dann ist $\text{tw}(G) = 2$.
- Sei G ein Graph, $v \in V, e \in E$. Dann ist jede Baumzerlegung D für G auch eine Baumzerlegung für $G - e$. Auch kann D in eine Baumzerlegung D' für $G - v$ transformiert werden mit gleicher oder kleinerer Weite (z.B. durch Entfernen von v aus D).
 \implies Die Baumweite ist monoton: zusätzliche Knoten/Kanten verringern nicht die Baumweite.
- Sei G ein Graph, sei $C \subseteq V$ eine Clique der Grösse $|C| = k$. Dann gilt $\text{tw}(G) \geq k - 1$ (und es existiert ein bag, der alle Knoten der Clique enthält).
 \implies grosse Cliques \implies grosse Baumweite. ABER kleine Cliques \nRightarrow kleine Baumweite (z.B. $l \times l$ -Gitter-Graph).
 \implies Konzept der Clique kein genügendes Modell für den Zusammenhang (und damit der Baumähnlichkeit) eines Graphen. Verallgemeinerung: statt Knoten die sich gegenseitig berühren betrachten wir Mengen von Knoten die sich gegenseitig berühren.

Definition (bramble) Sei G ein Graph, sei $\mathcal{B} = \{B_1, \dots, B_k\}$ mit $B_i \subseteq V$. Wir sagen dass B_i, B_j sich *berühren* (*touch*), falls $B_i \cap B_j \neq \emptyset$ oder $\exists \{x_i, x_j\} \in E : x_i \in B_i \wedge x_j \in B_j$.

Wenn alle B_i sich paarweise berühren, dann heisst \mathcal{B} *bramble* ("Dornen-/Brombeergestrüpp") von G .

$C \subseteq V$ heisst *Überdeckung* (*cover*) von \mathcal{B} , falls $C \cap B_i \neq \emptyset$ für alle i .

Die *Ordnung* (*order*) von \mathcal{B} ist die Grösse einer minimalen Überdeckung von \mathcal{B} .

Die *bramble number* $\text{bn}(G)$ von G ist die maximale Ordnung eines bramble von G .

Beispiel: In einem $k \times k$ -Gitter-Graph sei ein "Kreuz" $C_{i,j}$ die Vereinigung von Reihe i und Spalte j . Dann ist $\mathcal{B} = \{C_{i,j} \mid 1 \leq i, j \leq k\}$ ein bramble von G der Ordnung k und jede Reihe und jede Spalte ist eine Überdeckung von \mathcal{B} .

Theoreme

- Sei G ein Graph. Dann gilt $\text{bn}(G) = \text{tw}(G) + 1$.
 \longrightarrow Die bramble number beschreibt die Grösse des grössten bags in einer optimalen Baumzerlegung. D.h. sie bietet eine untere Schranke für die Weite einer beliebigen Baumzerlegung.
- Die Konstruktion einer Baumzerlegung und Bestimmung der Baumweite sind NP-schwer.
 \longrightarrow Problem: das Parameter eines parametrisierten Algorithmus' sollte in Polynomzeit berechenbar sein!
- Sei G ein Graph mit $\text{tw}(G) = k$. Dann kann man eine Baumzerlegung D der Grösse/Weite $\leq 5k$ in Zeit $2^{\mathcal{O}(k)} \cdot n$ berechnen.

3.5.1 Beispiel: VCP parametrisiert mit Baumweite

Idee Löse VCP in den bags und setze die Gesamtlösung aus den Teillösungen anhand der Baumstruktur bottom-up zusammen.

Definition Eine Baumzerlegung heisst *einfach* (*simple*), falls $\forall i, j \in I : X_i \not\subseteq X_j$.

Lemma Jede Baumzerlegung lässt sich umwandeln in eine einfache Baumzerlegung.

Beweis: falls solche X_i, X_j existieren, dann müssen sie auf einem Pfad liegen. O.B.d.A. ist X_i Nachbar von X_j . Kontrahiere beide. Die Weite bleibt erhalten. Machbar in Zeit $\mathcal{O}(\text{tw}(G) \cdot n)$.¹⁷

Lemma Sei G ein Graph. Dann existiert eine Baumzerlegung $D = (T, B)$ von G mit (optimaler) Weite $\text{tw}(G)$ so dass $|V_T| \leq |V|$.

—→ Es gibt wenige bags, und diese sind klein.

Beweis: O.B.d.A. hat D Baumweite $\text{tw}(G)$ und ist einfach. Wähle ein $r \in T$ als Wurzel. $\forall v \in V$ sei $\text{high}(v)$ der höchste Knoten in V_T dessen bag v enthält. $\text{high}(v)$ ist eindeutig (warum?). Behauptung: $\forall X_i \exists v \in V : \text{high}(v) = X_i$. Fall nicht, gilt entweder $i = r$, aber dann ist $X_r = \emptyset$. Oder aber X_i hat einen parent X_j , und da es kein eindeutiges v gibt für X_i muss $X_i \subseteq X_j$ gelten. Beide Fälle sind ein Widerspruch zur Annahme dass D einfach ist. Also hat jeder bag mind. ein v an höchster Stelle, ergo muss $|V_T| \leq |V|$ gelten.

Notation Wurzel R , Orientierung von R zu den Blättern. Teilbaum T_X mit Wurzel X . Induzierter Teilgraph $G[T_X]$ von G mit all den Knoten die in einem bag aus T_X sind.

Definition Sei G ein Graph, sei $X \subseteq Y \subseteq V$, seien C_X, C_Y vertex cover für $G[X], G[Y]$. C_Y *erweitert* (*extends*) C_X , falls $C_X \subseteq C_Y$ und $C_Y \cap X = C_X$ (d.h. C_Y stimmt auf X mit C_X überein).

Theorem Sei G ein Graph und sei D eine einfache Baumzerlegung von G der Weite k .¹⁸ Dann lässt sich ein optimaler VC für G berechnen in Zeit $\mathcal{O}(2^k \cdot k \cdot n)$.

Beweis: Konstruktiv. Ziel: verwalte eine Tabelle, die für jeden bag X alle optimalen VCs für $G[T_X]$ enthält – diese lassen sich zu einem optimalen VC für G erweitern.

- 1) Initialisierung: Für alle bags X berechne alle zulässigen VCs für $G[X]$ und definiere $w_X : 2^X \mapsto \mathbb{N} \cup \{\infty\}$ für alle $C \subseteq X$ wie folgt:

$$w_X(C) = \begin{cases} |C| & \text{if } C \text{ is a VC for } G[X] \\ \infty & \text{otherwise} \end{cases}$$

D.h. w_X speichert die Grösse aller VC-Kandidaten für $G[X]$.

- 2) Bottom-up-Durchlauf von T : Für jeden Knoten Y merge die Information w_X aus jedem seiner Kinder X in w_Y hinein. Sobald alle Kinder abgearbeitet sind enthält w_Y die Grösse aller VC-Kandidaten für $G[T_Y]$.

Merge-Schritt: Für alle $C_Y \subseteq Y$, sei $Z = X \cap Y$ und sei $C_Z := (C_Y \cap Z) \subseteq Z$ (also C_Y erweitert C_Z). Dann berechne:

$$w_Y(C_Y) = w_Y(C_Y) + \min\{w_X(C_X) \mid C_X \subseteq X \text{ extends } C_Z\} - |C_Z|$$

Beispiel siehe Buch Kapitel 6.6, Seite 173.

Beobachtung: Jeder Knoten der in T_X aber nicht in X vorkommt, kommt nicht ausserhalb von T_X vor (wegen Eigenschaft 3 der Baumzerlegung). Dadurch bleibt die Tabelle klein, da wir für alle abgehandelten “unteren” Knoten den optimalen VC bereits kennen.

Laufzeit:

¹⁷Wenn die Ausgangszerlegung D wie im Theorem oben gebildet wurde, impliziert dies dass D $\mathcal{O}(n)$ viele bags hat.

¹⁸ $k = \text{tw}(G)$ muss nicht gelten!

- 1) Initialisierung: 2^{k+1} Teilmengen pro bag, jede in amortisiert¹⁹ $\mathcal{O}(k)$ testbar ob sie ein VC ist. $\leq n$ bags (da einfache Zerlegung). $\implies \mathcal{O}(2^k \cdot k \cdot n)$
- 2) Mergen von X in Vorgänger Y : Sortiere w_x -Tabelle bzgl. Ordnung auf $Z = X \cap Y$ in $\mathcal{O}(2^k \cdot \log 2^k)$. Finde Minimum für alle C_Z in $\mathcal{O}(2^k)$ (da Tabelle sortiert). Für jedes $C_Y \subseteq Y$ finde passendes C_Z in w_x -Tabelle in $\mathcal{O}(\log 2^k)$ dank binärer Suche. $\leq n$ merges. $\implies \mathcal{O}(2^k \cdot k \cdot n)$

Korollar Das VCP ist fpt bzgl. $\text{tw}(G)$.

[Ausblick] Theorem Sei (G, k) eine VCP-Instanz, so dass $G = (V, E)$ planar ist und $|V| = n$. Dann kann das VCP auf G in subexponentieller Zeit $2^{\mathcal{O}(\sqrt{k})} \cdot n$ gelöst werden.

Beweisidee: Berechne einen VC-Kern der Grösse $\mathcal{O}(k)$ (insbesondere bleibt der Graph planar). Ein planarer Graph kann mit Separatoren der Grösse $\mathcal{O}(\sqrt{k})$ in Teile der Grösse $\mathcal{O}(\sqrt{k})$ zerlegt werden. Dann gilt $\text{tw}(G) \in \mathcal{O}(\sqrt{k})$ und wir können eine Baumzerlegung mit Weite $\mathcal{O}(\sqrt{k})$ finden. Wende dann den $\text{tw}(G)$ -parametrisierten Algorithmus an.

3.6 Grenzen der Parametrisierbarkeit

Ziel Zeige, dass es Probleme gibt, die keine fpt-Algorithmen erlauben (unter komplexitätstheoretischen Annahmen).

Definition Das parametrisierte Problem (U_1, par_1) lässt sich auf (U_2, par_2) reduzieren mit *parametrisierter Standardreduktion*, falls es einen Algorithmus gibt, der in fpt-Zeit Instanzen (I_1, k) in (I_2, k') umwandelt, JA-Instanzen erhält, und k' nur von k abhängt (nicht von $|I_1|$).

Theorem Falls (U_2, par_2) fpt ist und eine parametrisierte Reduktion von (U_1, par_1) auf (U_2, par_2) existiert, dann ist auch (U_1, par_1) fpt.

Definitionen

- Das *gewichtete (weighted) 2-SAT Problem (W2SAT)* ist ein parametrisiertes Problem:
Eingabe: 2-KNF Formel F , $k \in \mathbb{N}$
Ausgabe: JA, falls es eine erfüllende Belegung für F mit Gewicht²⁰ genau k gibt.
Parametrisierung: Gewicht k
- Die Klasse $W[1]$ enthält alle parametrisierten Probleme, die sich auf W2SAT parametrisiert reduzieren lassen (c.f. Klasse NP).²¹
- Problem U ist *W[1]-schwer*, falls sich W2SAT auf U reduzieren lässt.
- Problem U ist *W[1]-vollständig*, falls es in $W[1]$ liegt und $W[1]$ -schwer ist.
- Die Klasse *FPT* enthält alle parametrisierten Probleme, für die ein fpt-Algorithmus existiert (c.f. Klasse P).

¹⁹Naiv $\mathcal{O}(k^2)$ da wir alle Kanten in $G[X]$ auf coverage testen müssen. Besser: iteriere über Teilmengen via Gray-Code, so dass sich immer nur ein Knoten ändert, d.h. wir nur $\leq k$ Kanten testen müssen.

²⁰Gewicht := Anzahl auf 1 gesetzter Variablen

²¹Der Name $W[1]$ ist historisch bedingt, die Klasse ist ursprünglich auch nicht via W2SAT definiert.

Theorem Falls $W[1] = FPT$, dann kann 3-SAT auf Eingabe F mit n Variablen in Zeit $2^{o(n)} \cdot |F|^{\mathcal{O}(1)}$ gelöst werden.

—→ Schwierigkeit von $W[1]$ steht in direkter Verbindung mit der subexponentieller Lösbarkeit von 3-SAT. Allerdings: die Bedingung dass ein solcher Algorithmus nicht existiert (*Exponential-Zeit-Hypothese ETH*) ist stärker als $P \neq NP$ (bzw. $3\text{-SAT} \notin P$). D.h. $W[1]$ -Vollständigkeit ist schwächer als NP-Vollständigkeit.

Theorem Das Independent-Set-Problem ist bzgl. Standardparameter $W[1]$ -vollständig.

4 Exakte Exponentialzeit Algorithmen

Konzepte

- \mathcal{O}^* -Notation
- Branching-Algorithmen: 3-SAT, Max-IS
- Dynamische Programmierung: TSP/Held-Karp, Graph Coloring
- Measure-and-Conquer

Motivation Entwerfe Algorithmen die zwar exponentiell, aber dennoch schneller als vollständige Suche sind. Insbesondere in Zeit c^n für $c < 2$ (z.B. $2^{\sqrt{n}} \approx 1.41^n$) anstelle von 2^n .

\mathcal{O}^* -Notation Idee: so wie wir in \mathcal{O} -Notation konstanten Faktoren weglassen, lassen wir in \mathcal{O}^* -Notation polynomielle Vorfaktoren weg (da der exponentielle Faktor dominiert).

Seien $f, g : \mathbb{N} \mapsto \mathbb{R}^+$. Dann gilt $f(n) \in \mathcal{O}^*(g(n))$, falls ein Polynom $p : \mathbb{N} \mapsto \mathbb{R}^+$ existiert, so dass $f(n) \in \mathcal{O}(p(n) \cdot g(n))$.

4.1 Branching-Algorithmen

Idee Instanz der Grösse n aufteilen in konstant viele kleinere Teilinstanzen der Grösse $n - d$, für eine Konstante d . Ähnlich divide-and-conquer.

4.1.1 Beispiel: 3-SAT

Annahme: O.B.d.A. hat jede Klausel Literale von paarweise verschiedenen Variablen und alle Klauseln sind paarweise verschieden, dank polynomielltem preprocessing. Dann gilt $|\Phi| \in \mathcal{O}(n^3)$.

Notation Sei Φ eine Formel in 3-KNF über Variablen X , sei $x \in X$, $\alpha \in \{0, 1\}$. $\Phi[x = \alpha]$ beschreibt die Formel in 3-KNF, die aus Φ entsteht indem man den Wert $x = \alpha$ einsetzt und vereinfacht.

Trivialer Ansatz Wähle eine Variable x , berechne $\Phi[x = 0]$ und $\Phi[x = 1]$, merge (Φ satisfiable $\iff \Phi[x = 0]$ satisfiable $\vee \Phi[x = 1]$ satisfiable). Zeit: $T(n) = 2 \cdot T(n-1) + \mathcal{O}(n) \implies T(n) \in \mathcal{O}^*(2^n) \implies$ keine Verbesserung, probiert nur alle Belegungen aus.

Branch-3SAT / Monien-Speckenmeyer Algorithmus

1. Falls $\Phi = \emptyset$ return JA
2. Wähle Klausel $F \in \Phi$ von minimaler Länge
3. Falls $F = \emptyset$ return NEIN
4. Falls $F = (l)$, rufe Branch-3SAT($\Phi[l = 1]$) auf
5. Falls $F = (l_1 \vee l_2)$, rufe Branch-3SAT($\Phi[l_1 = 1]$) und Branch-3SAT($\Phi[l_1 = 0, l_2 = 1]$) auf
6. Falls $F = (l_1 \vee l_2 \vee l_3)$, rufe Branch-3SAT($\Phi[l_1 = 1]$) und Branch-3SAT($\Phi[l_1 = 0, l_2 = 1]$) und Branch-3SAT($\Phi[l_1 = 0, l_2 = 0, l_3 = 1]$) auf

7. return JA falls einer der rekursiven Aufrufe JA returned, sonst NEIN

Theorem Branch-3SAT löst 3-SAT in Zeit $\mathcal{O}^*(1.8393^n)$ für alle Formeln über n boolesche Variablen.

Laufzeit:

$$T(n) \leq T(n-1) + T(n-2) + T(n-3) + \mathcal{O}(n)$$

$$T(0) = T(1) = T(2) = \mathcal{O}(n)$$

Nehme $T(n) = \alpha^n$ an, forme um nach $\alpha^3 - \alpha^2 - \alpha - 1 = 0$, finde reale Nullstelle mit $\alpha \in [1, 2)$.

4.1.2 Beispiel: Max-IS

Maximum Independent Set Problem Max-IS ²²

Eingabe: $G = (V, E)$

Lösungen: $\mathcal{M}(G) = \{S \subseteq \mathcal{M} \mid \{u, v\} \notin E \text{ for all } u, v \in S, u \neq v\}$

Kosten: $\text{cost}(S, G) = |S|$

Ziel: max

Algorithmus: Branch-Max-IS

1. if $V = \emptyset$: return \emptyset
2. let $v \leftarrow \min_{u \in V} \deg_G(u)$
3. let $N[v] = \{u_1, \dots, u_k\}$ be the *closed* neighbourhood²³ of v
4. return the largest set from the family $\{\{u_i\} \cup \text{Branch-Max-IS}(G - N[u_i]) \mid i \in \{1, \dots, k\}\}$

Theorem Branch-Max-IS löst Max-IS in Zeit $\mathcal{O}^*(3^{n/3}) \subseteq \mathcal{O}^*(1.4423^n)$.

Korrektheit: Per Konstruktion (wegen Beobachtung dass $\forall v \in V$ mindestens ein $u \in N[v]$ in der optimalen, maximum Lösung enthalten sein muss).

Laufzeit: Ziel: beschränke Anzahl Knoten des Rekursionsbaumes. Es gilt:

$$T(G) \leq 1 + T(G - N[v]) + \sum_{i=2}^k T(G - N[u_i])$$

$$T(n) = \max_{\substack{G=(V,E) \\ |V|=n}} T(G)$$

²²Recall: maximal == nicht erweiterbar. maximum == grösstes mögliches.

²³Oft nehmen wir o.B.d.A. an dass $u_1 := v$.

Ausserdem gilt Monotonie $T(n) \leq T(n+1)$ und $T(0) = 1$.

Sei $\delta_{\min}(G)$ der minimale Grad in G . Es gilt:

$$\begin{aligned}
 T(G) &\leq 1 + T(n - \delta_G(v) - 1) + \sum_{i=2}^k T(n - \delta_G(u_i) - 1) \\
 &\stackrel{(1)}{\leq} 1 + T(n - \delta_{\min}(G) - 1) + \delta_{\min}(G) \cdot T(n - \delta_{\min}(G) - 1) \\
 &= 1 + (1 + \delta_{\min}(G)) \cdot T(n - (\delta_{\min}(G) + 1)) \\
 &\leq 1 + \max_{1 \leq s \leq n} s \cdot T(n - s) \\
 T(n) &\stackrel{(2)}{\leq} 1 + \max_{1 \leq s \leq n} s \cdot T(n - s)
 \end{aligned}$$

wobei (1) aus der Monotonie folgt. (2) können wir schreiben, da die rechte Seite nicht mehr von einem bestimmten Graphen G abhängt, sondern nur noch von der Grösse der Eingabe.

Auflösen dieser letzten Ungleichung siehe Buch Kapitel 5.2, Seite 109f.

4.2 Dynamische Programmierung

Travelling Salesman Problem TSP

Eingabe: (O.B.d.A. vollständiger) Graph $G = (V, E)$, Kostenfunktion $c : E \mapsto \mathbb{N}$

Ausgabe: Hamiltonkreis H in G

Kosten: $\text{cost}(H, G) = \sum_{e \in H} c(e)$

Ziel: min

Problem: $\mathcal{O}(n!) = \mathcal{O}^*(2^{n \log n})$ viele mögliche Lösungen

Dyn-Prog-TSP / Held-Karp Algorithmus Für alle $S \subseteq \{v_2, \dots, v_n\}$ und jeden Knoten $x \in S$ berechne $l(S, x)$ als die minimalen Kosten eines Pfades der in v_1 startet, in $x \in S$ endet und genau alle Knoten aus S je einmal besucht.

1. Initialisierung: $l(\{v_i\}, v_i) \leftarrow c(\{v_1, v_i\})$ für alle $i \in \{2, \dots, n\}$
2. Für alle $j \in \{2, \dots, n-1\}$, für alle $S \subseteq \{v_2, \dots, v_n\}$ mit $|S| = j$, für alle $x \in S$ berechne:

$$l(S, x) \leftarrow \min_{y \in (S - \{x\})} \{l(S - \{x\}, y) + c(\{y, x\})\}$$

3. Finde Lösung:

$$l_{\min} \leftarrow \min_{v_i \in \{v_2, \dots, v_n\}} \{l(\{v_2, \dots, v_n\}, v_i) + c(\{v_i, v_1\})\}$$

Return l_{\min} .

Theorem Held-Karp löst das TSP in Zeit $\mathcal{O}^*(2^n)$.

Beweis: Korrektheit trivial. Schritte 1 und 3 in $\mathcal{O}(n)$. Schritt 2: $\mathcal{O}(2^n)$ viele Teilmengen S , für jede wählen wir $x, y \implies \mathcal{O}(n^2 \cdot 2^n)$.

Graph Coloring Problem (Graphfärbung) Sei G ein ungerichtet, ungewichteter Graph und $k \in \mathbb{N}$. Eine k -Färbung (k -coloring) von G ist eine Funktion $c: V \mapsto \{1, \dots, k\}$ so dass

$$\{x, y\} \in E \implies c(x) \neq c(y)$$

Die Menge $C_i := \{x \in V \mid c(x) = i\}$ heisst i -te *color class* von c . Die *chromatische Zahl* (*chromatic number*) $\chi(G)$ von G ist das kleinste k so dass G eine k -Färbung besitzt.

Eingabe: $G = (V, E)$

Ausgabe: $\chi(G)$

Bruteforce: $\mathcal{O}^*(n^n)$ da $\chi(G) \leq n$

Algorithmus: Dyn-Prog-Coloring Beobachtungen: 1) Jeder Farbklass ist ein independent set in G . 2) Wenn C_i kein maximales IS ist, dann existiert ein $x \in V - C_i$ dass zu i umgefärbt werden kann.

1. Initialisierung: $\chi(\emptyset) \leftarrow 0$

2. Für alle $i \in \{1, \dots, n\}$, für alle $X \subseteq V$ mit $|X| = i$ berechne:

$$\chi(X) \leftarrow 1 + \min\{\chi(X - I) \mid I \text{ ist ein maximales IS in } G[X]\}$$

Return $\chi(G) \leftarrow \chi(V)$

Theorem Dyn-Prog-Coloring löst das Graphfärbungsproblem in Zeit $\mathcal{O}^*(2.4423^n)$.

Beweis: Verwende Variante von Branch-Max-IS²⁴ um in $\mathcal{O}^*(3^{i/3})$ alle maximalen IS der Grösse i zu finden. Insgesamt:

$$\mathcal{O}^* \left(\sum_{i=1}^n \binom{n}{i} \cdot 3^{i/3} \cdot 1^{(n-i)/3} \right) = \mathcal{O}^* \left(\left(1 + 3^{1/3}\right)^n \right) \in \mathcal{O}^*(2.4423^n)$$

unter Anwendung des Binomischen Satzes.

4.3 Measure-and-Conquer

Idee Ähnlich zu Branching-Algorithmen: Teile das Problem in Teilinstanzen auf. Bei der Analyse betrachte aber nicht eine Rekursionsgleichung in der Eingabegrösse, sondern analysiere eine andere Grösse. D.h. entwerfe keine neuen Algorithmen, sondern verbessere die Analyse.

Definition Es gibt zwei Arten von Regeln, die wir auf die Eingabe anwenden:

1. (*data*) *reduction rules*, welche eine Instanz zu einer äquivalenten, kleineren Instanz reduzieren.
2. *branching rules*, welche eine Instanz in mehrere kleinere Teilinstanzen aufteilen.

Eine branching rule reduziert eine Instanz der Grösse n in l viele Teilinstanzen der Grössen $n - k_1, \dots, n - k_l$. $(k_1, \dots, k_l) \in \mathbb{Q}^{+l}$ heisst *branching vector* der branching rule.

²⁴Betrachtet geschlossene Nachbarschaft aller Knoten, nicht nur die von minimalem Grad, und gibt alle maximalen IS zurück (nicht nur das grösste).

Lemma (branching factor) Sei \mathcal{A} ein Algorithmus mit einer branching rule mit branching vector (k_1, \dots, k_l) . Nehme an, dass die Anwendung einer Regel (und das Testen ihrer Anwendbarkeit!) in Polynomzeit möglich ist. Beobachte ausserdem, dass Datenreduktionsregeln nur polynomiell oft angewendet werden können.

Dann ist die Laufzeit von \mathcal{A} auf eine Eingabe der Grösse n in Zeit $\mathcal{O}^*(\alpha^n)$; dabei ist α die eindeutige positive Nullstelle des Polynoms

$$x^n - x^{n-k_1} - x^{n-k_2} - \dots - x^{n-k_l}$$

α heisst *branching factor* der branching rule, auch notiert als $\alpha(k_1, \dots, k_l)$.

Falls \mathcal{A} mehrere branching rules verwendet, schätze jede Regel einzeln ab und nehme den worst case als obere Schranke.

Beobachtung

- (a) $\alpha(k_1, \dots, k_l) = \alpha(k_{i_1}, \dots, k_{i_l})$ für jede Permutation (i_1, \dots, i_l) von $(1, \dots, l)$.
- (b) $\alpha(k_1, k_2, \dots, k_l) > \alpha(k'_1, k_2, \dots, k_l)$ für jedes $k'_1 > k_1$
- (c) $\alpha(k_1, k_2) > \alpha(k_1 + s, k_2 - s)$

Intuitiv: Der branching factor wird kleiner wenn ein Element des branching vectors grösser wird, oder wenn der Vektor “balancierter” wird. Ein kleinerer branching factor ist besser.

Algorithmus: Max-Degree-Branch-Max-IS Ähnlich Branch-Max-IS, aber branched auf Knoten von maximalen (anstatt minimalem) Grad.

1. If $\exists v \in V : \deg_G(v) \leq 1$: return $1 + \text{MAX-DEGREE-BRANCH-MAX-IS}(G - N[v])$
2. If $\exists v \in V : \deg_G(v) \geq 3$: let v be a vertex of maximum degree. return $\max\{1 + \text{MAX-DEGREE-BRANCH-MAX-IS}(G - N[v]), \text{MAX-DEGREE-BRANCH-MAX-IS}(G - v)\}$
3. If $\exists v \in V : \deg_G(v) = 2$: compute a max IS in polynomial time on the remaining trees+cycles

Theorem Max-Degree-Branch-Max-IS löst Max-IS in Zeit $\mathcal{O}^*(1.3803^n)$

Beweis: Korrektheit: offensichtlich (Schritt 1: sichere Datenreduktionsregel).

Laufzeit: der branching vector in Schritt 2 ist $(1, 1 + \deg_G(v))$ bzw. im worst case $(1, 4)$ (siehe Beobachtung).

\implies branching factor $\alpha(1, 4) < 1.3803$.

Measure-and-Conquer Analyse Idee: Die branching-rule-Analyse betrachtet alle Knoten, aber ignoriert dass Knoten mit kleinem Grad “nicht so schlimm” sind für die Laufzeit. Seien $n_{\leq 1}, n_2, n_{\geq 3}$ die Anzahl Knoten von entsprechendem Grad. Definiere eine Gewichtsfunktion (arbiträr):

$$w(v) := \begin{cases} 0 & \text{if } \deg_G(v) \leq 1 \\ \frac{1}{2} & \text{if } \deg_G(v) = 2 \\ 1 & \text{if } \deg_G(v) \geq 3 \end{cases}$$

Das Gesamtgewicht des Graphen ist dann:

$$w(G) = \sum_{v \in V} w(v) = n_{\leq 1} \cdot 0 + n_2 \cdot \frac{1}{2} + n_{\geq 3} \cdot 1 = \frac{1}{2} \cdot n_2 + n_{\geq 3}$$

In den neuen Teilinstanzen betrachte nun die Gewichts-Verringerung (anstatt der Knotenanzahl-Verringerung). Sei $G_{in} = G - N[v]$ und $G_{out} = G - v$. Seien w_{in}, w_{out} die Werte, um die sich die Gewichte verringern:

$$w(G_{in}) \leq w(G) - w_{in} \quad w(G_{out}) \leq w(G) - w_{out}$$

Der branching vector ist also (w_{in}, w_{out}) .

Da in beiden Fällen mindestens v entfernt wird, wissen wir dass $w_{in}, w_{out} \geq 1$.

Jeder Knoten $u \in N(v)$ mit $\deg_G(u) = 2$ wird entweder entfernt (G_{in}) oder verliert seinen Nachbarn v (G_{out}). In beiden Fällen verringert sich das Gewicht um $\frac{1}{2}$, in der Summe also um 1.

Jeder Knoten $u \in N(v)$ mit $\deg_G(u) \geq 3$ wird in G_{in} entfernt, d.h. der Graph verliert Gewicht 1. Über G_{out} können wir in diesem Fall nichts sagen, das Gewicht von u bleibt schlimmstenfalls gleich.

Zusammengefasst: In beiden Fällen wird v entfernt (wobei $w(v) = 1$ sonst würden wir nicht auf v branchen). Ausserdem wird via die Nachbarschaft $N(v)$ über G_{in}, G_{out} hinweg $d := \deg_G(v)$ -mal das Knotengewicht 1 entfernt. Also gilt in der Summe:

$$w_{in} + w_{out} \geq 2 + d$$

Aus der vorherigen Beobachtung folgt, dass im schlimmsten Fall w_{in}, w_{out} unbalanciert sind, also der branching vector $(1, 1 + d)$ ist.

Fallunterscheidung:

$d = 3$ Der Graph hat nur Knoten von Grad 2 oder 3. Damit wissen wir, dass sich sowohl in G_{in} als auch in G_{out} das Gewicht aller Nachbarn von v um mindestens $\frac{1}{2}$ verringert:

$$w_{in}, w_{out} \geq 1 + 3 \cdot \frac{1}{2} = 2.5$$

$$\implies \alpha(2.5, 2.5) < 1.3196.$$

$$d \geq 4 \implies \alpha(1, 5) < 1.3248.$$

Wegen der Beobachtung wissen wir, dass $\alpha(1, 4) > \alpha(1, 5)$. Diese Fallunterscheidung erlaubt es uns den Fall $d = 3$ besser abzuschätzen als mit $\alpha(1, 4)$. Da für grössere d der branching factor α nur kleiner wird, können wir folgern:

Theorem Max-Degree-Branch-Max-IS löst Max-IS in Zeit $\mathcal{O}^*(1.3248^n)$.

5 Approximations-Algorithmen

Konzepte

- Relativer Fehler, Approximationsgüte
- Beispiele: 2-Approximation Min-VCP, Spannbaum-TSP, Christofides (metrisches TSP)
- Nichtapproximierbarkeit

Motivation Schwere Optimierungsprobleme: Statt Abstriche bei der Laufzeit zu machen (Parametrisierung, exponentiell aber mit kleiner Basis), machen wir Abstriche bei der Optimalität. Ein Approximationsalgorithmus berechnet eine zulässige Lösung (feasible solution), dessen Qualität nicht weit von der Qualität einer optimalen Lösung abweicht.

Definitionen Sei $U = (L, \mathcal{M}, cost, goal)$ ein Optimierungsproblem und sei \mathcal{A} ein Algorithmus für U .

- Für alle $I \in L$ ist der *relative Fehler von \mathcal{A} auf I*

$$\varepsilon_{\mathcal{A}}(I) := \frac{|cost(\mathcal{A}(I)) - Opt_U(I)|}{\min\{Opt_U(I), cost(\mathcal{A}(I))\}}$$

- Für alle $n \in \mathbb{N}$ ist der *relative Fehler von \mathcal{A}*

$$\varepsilon_{\mathcal{A}}(n) := \max_{I \in L} \{\varepsilon_{\mathcal{A}}(I) \mid |I| = n\}$$

- Für alle $I \in L$ ist die *Approximationsgüte (approximation ratio) von \mathcal{A} auf I*

$$R_{\mathcal{A}}(I) := \max \left\{ \frac{cost(\mathcal{A}(I))}{Opt_U(I)}, \frac{Opt_U(I)}{cost(\mathcal{A}(I))} \right\}$$

- Für alle $n \in \mathbb{N}$ ist die *Approximationsgüte (approximation ratio) von \mathcal{A}*

$$R_{\mathcal{A}}(n) := \max_{I \in L} \{R_{\mathcal{A}}(I) \mid |I| = n\}$$

- Für alle positiven reellen $\delta > 1$ heisst \mathcal{A} δ -Approximationsalgorithmus für U falls gilt

$$\forall I \in L . R_{\mathcal{A}}(I) \leq \delta$$

- Für jede Funktion $f : \mathbb{N} \mapsto \mathbb{R}^+$ heisst \mathcal{A} $f(n)$ -Approximationsalgorithmus für U falls gilt

$$\forall n \in \mathbb{N} . R_{\mathcal{A}}(I) \leq f(n)$$

Wobei die Definitionen sowohl $goal = \min$ als auch $goal = \max$ sinnvoll beschreiben.

Beispiel: 2-Approximationsalgorithmus für Min-VCP Idee: berechne ein maximales Matching M und ein approximiertes vertex cover C .

1. Initialisierung: Sei $C \leftarrow \emptyset, M \leftarrow \emptyset, E' \leftarrow E$.
2. while $E' \neq \emptyset$ do:
Wähle eine beliebige Kante $\{u, v\} \in E'$.
Update: $C \leftarrow C \cup \{u, v\}, M \leftarrow M \cup \{\{u, v\}\}, E' \leftarrow E' - \{\text{all edges incident to } u \text{ or } v\}$

Theorem Obiger Algorithmus ist ein 2-Approximationsalgorithmus für das Min-VCP.

Beweis: Korrektheit: nur Kanten die mindestens einen Nachbarn in C haben werden entfernt, daher ist C ein vertex cover.

Laufzeit: Linearzeit, $\mathcal{O}(|E|)$.

2-Approximation: $|C| = 2 \cdot |M|$. Um $|M|$ viele Kanten in M (und damit auch in G , da $M \subseteq E$) zu überdecken, werden $\geq |M|$ viele Knoten benötigt.

$\implies \text{Opt}_{\text{Min-VCP}}(G) \geq |M| \iff |C| \leq 2 \cdot \text{Opt}_{\text{Min-VCP}}(G)$.

Beispiel: Metrisches-TSP (Δ -TSP) Eingabe: vollständiger, gewichteter Graphen $G = (V, E)$ und Kostenfunktion $c : E \mapsto \mathbb{N}^+$. Metrisch == Dreiecksungleichung gilt:

$$\forall u, v, w \in V : c(\{u, v\}) \leq c(\{u, w\}) + c(\{w, v\})$$

Spannbaum-Algorithmus für Δ -TSP Idee: minimum Hamiltonkreise zu berechnen ist schwer, aber minimale Spann bäume sind einfach (Polynomzeit).

1. Berechne einen MST T von G
2. Wähle einen beliebigen $v \in V$. Berechne eine DFS in T ausgehend von v und ordne dabei die Knoten in Besuchsreihenfolge \overline{H} .
3. return Hamiltonkreis $H = \overline{H}, v$

Theorem Der Spannbaum-Algorithmus ist ein 2-Approximationsalgorithmus für das Δ -TSP.

Beweis: Korrektheit: offensichtlich.

Laufzeit: polynomiell, z.B. $\mathcal{O}(|E| \log |V|)$ mit Kruskal-MST.

2-Approximation: Sei H_{Opt} ein optimaler Hamiltonkreis, d.h. $\text{cost}(H_{\text{Opt}}) = \text{Opt}_{\Delta\text{-TSP}}(I)$. Dann gilt:

$$\text{cost}(T) = \sum_{e \in E(T)} c(e) \leq \text{cost}(H_{\text{Opt}})$$

da H_{Opt} durch Entfernen einer Kante zu einem Spannbaum wird und T ein minimaler Spannbaum ist.

Sei W die Eulertour die \mathcal{A} auf dem Multigraphen T_2 läuft (wobei T_2 aus T durch Verdoppeln aller Kanten gebildet wird). Die Kosten von ausgegebenen H ist eine E

$$\text{cost}(W) = 2 \cdot \text{cost}(T) \leq 2 \cdot \text{cost}(H_{\text{Opt}})$$

Wenn in H noch doppelt vorkommende Knoten abgekürzt werden, gilt dank der Dreiecksungleichung:

$$\text{cost}(H) \leq \text{cost}(W)$$

Insgesamt gilt also $\text{cost}(H) \leq 2 \cdot \text{cost}(H_{\text{Opt}})$, also ist \mathcal{A} eine 2-Approximation.

Lemma In jedem Graphen T ist die Anzahl Knoten mit ungeradem Grad gerade.

Beweis: Summe t aller Grade ist $2 \cdot |E|$ (gerade), Summe e aller geraden Grade ist gerade, also ist die Differenz $o = t - e$ auch gerade. Da alle Summanden in der Summe o aller ungeraden Grade ungerade sind muss die Anzahl Summanden (und damit die Anzahl Knoten mit ungeradem Grad) gerade sein.

Christofides-Algorithmus für Δ -TSP Idee: für eine Eulertour braucht jeder Knoten geraden Grad. Berechne ein perfektes Matching auf den Knoten mit ungeradem Grad und vereinige es mit dem MST.

1. Berechne eine MST T von G
2. Sei $S \leftarrow \{v \in V \mid \deg_T(v) \text{ ungerade}\}$
3. Berechne ein perfektes Matching M auf S in G mit minimalen Kosten
4. Berechne eine Eulertour ω auf dem Multigraphen $G' = (V, E(T) \cup M)$
5. Verkürze ω zu einem Hamiltonkreis H in G und gebe H aus

Theorem Der Christofides-Algorithmus ist ein 1.5-Approximationsalgorithmus für das Δ -TSP.

Beweis: Korrektheit: offensichtlich.

Laufzeit: nicht-trivial (insbesondere perfektes Matching), aber in Polynomzeit $\mathcal{O}(n^4)$.

1.5 Approximation: Es gilt (\star wegen der Dreiecksungleichung):

$$\text{cost}(H) \stackrel{\star}{\leq} \text{cost}(\omega) = \text{cost}(G') = \sum_{e \in E(T) \cup M} c(e) = \text{cost}(T) + \text{cost}(M)$$

Und wie beim Spannbaum-Algorithmus gilt $\text{cost}(T) \leq \text{cost}(H_{Opt})$.

Zeige nun dass $\text{cost}(M) \leq \frac{1}{2} \cdot \text{cost}(H_{Opt})$ gilt:

Sei $S = \{v_1, \dots, v_{2m}\}$ die Menge von Knoten mit ungeradem Grad in T , o.B.d.A. so dass sie in dieser Reihenfolge in H_{Opt} vorkommen:

$$H_{Opt} = v_1, \alpha_1, v_2, \alpha_2, \dots, \alpha_{2m-1}, v_{2m}, \alpha_{2m}, v_1$$

wobei α_i die (womöglich leeren) Pfade zwischen den v_i in H_{Opt} sind.

Betrachte die disjunkten Matching M_1, M_2 :

$$\begin{aligned} M_1 &:= \{\{v_1, v_2\}, \{v_3, v_4\}, \dots, \{v_{2m-1}, v_{2m}\}\} \\ M_2 &:= \{\{v_2, v_3\}, \{v_4, v_5\}, \dots, \{v_{2m-2}, v_{2m-1}\}, \{v_{2m}, v_1\}\} \end{aligned}$$

Dann gilt (\star wegen der Dreiecksungleichung und den übersprungenen α_i):

$$\text{cost}(M_1) + \text{cost}(M_2) = \sum_{i=1}^{2m} \text{cost}(v_i, v_{(i+1) \bmod 2m}) \stackrel{\star}{\leq} \text{cost}(H_{Opt})$$

Ausserdem gilt $\text{cost}(M) \leq \text{cost}(M_1), \text{cost}(M_2)$ da M_1, M_2 ebenfalls perfekte Matchings in S sind (aber M ein min-cost perfektes Matching ist). Es folgt:

$$\text{cost}(M) \leq \min\{\text{cost}(M_1), \text{cost}(M_2)\} \stackrel{\star}{\leq} \frac{1}{2} \cdot \text{cost}(H_{Opt})$$

wobei \star gilt weil nicht beide grösser als $\frac{1}{2} \cdot \text{cost}(H_{Opt})$ sein können.

5.1 Nichtapproximierbarkeit

Theorem Es existiert kein (polynomieller) 2^n -Approximationsalgorithmus für das allgemeine TSP, es sei denn $P=NP$.

Beweis: Reduktion von einem NP-schweren (Entscheidungs-)Problem (z.B. Hamiltonkreisproblem) auf das Problem eine approximierte TSP-Lösung zu finden.

Reduktionsidee: Füge fehlende Kanten hinzu, so dass G' vollständig ist. Definiere Kantengewichte:

$$c(e) = \begin{cases} 1 & \text{if } e \in E \\ n \cdot 2^n + 1 & \text{if } e \notin E \end{cases}$$

Sei \mathcal{A} ein Approximationsalgorithmus der auf Eingabe (G', c) eine 2^n -Approximation berechnet. Fallunterscheidung:

1. \exists Hamiltonkreis in $G \implies \exists$ Hamiltonkreis in G' mit Kosten n
2. \nexists Hamiltonkreis in $G \implies \forall$ Hamiltonkreise in G' haben Kosten $\geq (n-1) + (n \cdot 2^n + 1) > 2^n \cdot n$

Ist die Ausgabe von \mathcal{A} nun $> 2^n \cdot n$ dann weiss man, dass kein Hamiltonkreis der Kosten n in G existiert, da \mathcal{A} ein 2^n -Approximationsalgorithmus ist. Also kann \mathcal{A} genutzt werden um das Hamiltonkreisproblem zu lösen, was ein Widerspruch zu $P \neq NP$ ist.

5.2 Approximationsschemata

Definition: PTAS/FPTAS Sei $U = (L, \mathcal{M}, cost, goal)$ ein Optimierungsproblem und sei \mathcal{A} ein konsistenter Algorithmus für U .

\mathcal{A} heisst *polynomielles Approximationsschema* (*polynomial-time approximation scheme PTAS*) für U , falls gilt:

für alle $(I, \varepsilon) \in L \times \mathbb{R}^+$ berechnet \mathcal{A} eine zulässige Lösung $\mathcal{A}(I)$ mit $R_{\mathcal{A}}(I) \leq 1 + \varepsilon$ und die Laufzeit $\text{time}_{\mathcal{A}}(I, \varepsilon^{-1})$ ist polynomiell in $|I|$.²⁵

\mathcal{A} heisst *echtes polynomielles Approximationsschema* (*fully polynomial-time approximation scheme FPTAS*) für U , falls zusätzlich gilt:

die Laufzeit $\text{time}_{\mathcal{A}}(I, \varepsilon^{-1})$ ist polynomiell in $|I|$ und ε^{-1} .

Einfaches Rucksackproblem (Simple Knapsack Problem SKP)

Eingabe: $w_1, \dots, w_n, b \in \mathbb{N}^+$ (daher "einfach": Gewicht = Kosten)

Zulässige Lösungen: $T \subseteq \{1, \dots, n\}$ so dass $\sum_{i \in T} w_i \leq b$

Kosten: $cost(T) = \sum_{i \in T} w_i$

Ziel: max

Algorithmus: Greedy-SKP

1. Sortiere absteigend: $b \geq w_1 \geq \dots \geq w_n$
2. Initialisierung: $T \leftarrow \emptyset; cost(T) \leftarrow 0$
3. for all $i \in \{1, \dots, n\}$ do: if $cost(T) + w_i \leq b$ then $T \leftarrow T \cup \{i\}; cost(T) \leftarrow cost(T) + w_i$

²⁵Aber ohne Aussage über die Abhängigkeit von ε^{-1} , d.h. insbesondere darf $\text{time}_{\mathcal{A}}(I, \varepsilon^{-1})$ exponentiell in ε^{-1} sein.

Theorem Greedy-SKP ist eine 2-Approximation für das SKP.

Beweis: Laufzeit: $\mathcal{O}(n \log n)$

2-Approximation: Zeige: T ist optimal oder $\text{cost}(T) \geq \frac{b}{2}$. Falls $T = \{1, \dots, n\}$ dann ist T optimal. Andernfalls sei j der kleinste Index mit $j \notin T$ (da $b \geq w_1$ gilt $j \geq 2$). Dann gilt $\text{cost}(T) > b - w_j$ und $\text{cost}(T) \geq w_1 \geq w_j$. Addieren und durch 2 teilen ergibt: $\text{cost}(T) \geq (b - w_j + w_j)/2 = b/2$.

Algorithmus: PTAS-SKP Idee: Steuere durch den Parameter ε mit wie viel Aufwand eine optimale Lösung ausprobiert werden soll, und mache den Rest greedy.

Eingabe: $w_1, \dots, w_n, b \in \mathbb{N}^+, \varepsilon \in (0, 1)$ (für $\varepsilon \geq 1$ nehme direkt Greedy-SKP)

1. Sortiere absteigend: $b \geq w_1 \geq \dots \geq w_n$
2. $k \leftarrow \lceil \frac{1}{\varepsilon} \rceil$
3. Für alle $T \subseteq \{1, \dots, n\}$ mit $|T| \leq k$ und $\sum_{i \in T} w_i \leq b$: erweitere T zu T^* mit Greedy-SKP
4. return das T^* mit maximalem Gewicht

Achtung: Wir betrachten alle Teilmengen der Grösse k , nicht alle Teilmengen der ersten k Elemente!

Theorem PTAS-SKP ist ein PTAS für das SKP.

Beweis: Laufzeit: $\mathcal{O}(n \log n + 1 + n^k \cdot n) = \mathcal{O}(n^{\lceil \frac{1}{\varepsilon} \rceil + 1}) \rightarrow$ polynomial für fixes ε , aber exponentiell in ε

$(1 + \varepsilon)$ -Approximation:

Sei $T_{\text{Opt}} = \{i_1, \dots, i_p\}$ eine optimale Lösung für Eingabe I , d.h. $\text{cost}(T_{\text{Opt}}) = \text{Opt}_{\text{SKP}}(I)$. Fallunterscheidung:

$p \leq k$: PTAS-SKP betrachtet die Menge T_{Opt} , also ist T^* optimal und der relative Fehler ist 0.

$p > k$: PTAS-SKP betrachtet die Menge $P = \{i_1, \dots, i_k\}$, d.h. die k schwersten Elemente der optimalen Lösung. Sei P^* die berechnete greedy Erweiterung von P . Falls $P^* = T_{\text{Opt}}$ sind wir fertig. Sonst existiert ein $i_q \in T_{\text{Opt}} - P^*$ mit $i_q > i_k \geq k$. Es gilt:

$$\text{cost}(P^*) + w_{i_q} > b \geq \text{cost}(T_{\text{Opt}})$$

Dann gilt \star mit einem Durchschnitts-Argument (da w_{i_q} das kleinste Element der Summe ist):

$$w_{i_q} \stackrel{\star}{\leq} \frac{w_{i_1} + \dots + w_{i_k} + w_{i_q}}{k + 1} \leq \frac{\text{cost}(T_{\text{Opt}})}{k + 1}$$

Umformen ergibt:

$$\begin{aligned} R_{\text{PTAS-SKP}}(I)(I, \varepsilon) &\stackrel{\text{Def}}{=} \frac{\text{cost}(T_{\text{Opt}})}{\text{cost}(T^*)} \leq \frac{\text{cost}(T_{\text{Opt}})}{\text{cost}(P^*)} \\ &\leq \frac{\text{cost}(T_{\text{Opt}})}{\text{cost}(T_{\text{Opt}}) - w_{i_q}} \leq \frac{\text{cost}(T_{\text{Opt}})}{\text{cost}(T_{\text{Opt}}) - \frac{\text{cost}(T_{\text{Opt}})}{k+1}} \\ &= \frac{1}{1 - \frac{1}{k+1}} = \frac{k+1}{k} = 1 + \frac{1}{k} \leq 1 + \varepsilon \end{aligned}$$

Algorithmus: FPTAS-KP Idee: Ähnlich zum pseudopolynomiellen DP-Algorithmus (siehe Seite 7). Kostenwerte treiben Laufzeit nach oben, also runde sie.

Eingabe: $w_1, \dots, w_n, c_1, \dots, c_n, b \in \mathbb{N}^+, \varepsilon \in (0, 1)$

1. Initialisierung: $c_{max} \leftarrow \max\{c_1, \dots, c_n\}$; Rundungsfaktor $t \leftarrow \frac{\varepsilon \cdot c_{max}}{(1+\varepsilon) \cdot n}$
2. for all $i \in \{1, \dots, n\}$ do: $c'_i \leftarrow \lfloor \frac{c_i}{t} \rfloor$
3. Berechne eine optimale Lösung T' auf Eingabe $I' = w_1, \dots, w_n, c'_1, \dots, c'_n, b$ mit Dyn-Prog-KP
4. return T'

Theorem FPTAS-KP ist ein FPTAS für das KP.

Beweis: Korrektheit: Gewichte bleiben unverändert, daher ist T' eine zulässige Lösung für I .

Approximationsgüte: Voller Beweis siehe Buch Kapitel 7.5, Seite 217f. Umformen ergibt:

$$\begin{aligned} \text{cost}(T_{Opt}, I) &\geq \text{cost}(T', I) \geq \text{cost}(T_{Opt}, I) - n \cdot t \\ \implies 0 &\leq \text{cost}(T_{Opt}, I) - \text{cost}(T', I) \leq n \cdot t \stackrel{\text{Def}}{\leq} n \cdot \frac{\varepsilon \cdot c_{max}}{(1+\varepsilon) \cdot n} \leq \frac{\varepsilon}{(1+\varepsilon)} \cdot c_{max} \end{aligned}$$

D.h. die Differenz zwischen der optimalen und der berechneten Lösung ist beschränkt durch eine Funktion in ε und c_{max} .

Weiteres Umformen (siehe Buch) ergibt wie gewünscht: $R_{FPTAS-KP}(I, \varepsilon) \leq 1 + \varepsilon$.

Laufzeit: Es gilt:

$$\mathcal{O}(n + n + n \cdot \sum_{j=1}^n c'_j) = \mathcal{O}(\varepsilon^{-1} \cdot n^3)$$

dank folgender Abschätzung:

$$\sum_{j=1}^n c'_j = \sum_{j=1}^n \left\lfloor \frac{c_j}{t} \right\rfloor \leq \sum_{j=1}^n \left(c_j \cdot \frac{(1+\varepsilon) \cdot n}{\varepsilon \cdot c_{max}} \right) = \frac{1+\varepsilon}{\varepsilon} \cdot \frac{n}{c_{max}} \cdot \sum_{j=1}^n c_j \leq (1+\varepsilon) \cdot \varepsilon^{-1} \cdot n^2 \in \mathcal{O}(\varepsilon^{-1} \cdot n^2)$$

Die Laufzeit ist polynomiell in n und ε^{-1} , also ist der Algorithmus ein FPTAS für das KP.