Algorithmen für schwierige Probleme

February 3, 2025

Contents

1 3-SAT und Vertex Cover

1.1 Vertex Cover

Definition 1.1. Ein Vertex Cover für einen Graphen G ist eine Teilmenge $C \subseteq V$ der Knoten, sodass $\forall e = (u, v) \in E$ gilt $u \in C$ oder $v \in C$.

Definition 1.2. Für zwei Probleme A, B schreiben wir $A \preceq B$, wenn man A effizient auf B transformieren kann.

Satz 1.3. *Der Greedy Algorithmus, der immer den Knoten höchsten Grades wählt, kann beliebig schlechte Ergebnisse liefern.*

Satz 1.4. *Der folgende Algorithmus liefert eine 2-Faktor-Approximation an das optimale Vertex Cover.*

1. *setze* $C = \varnothing$
2. *WHILE* $E \neq \varnothing$
3. *wähle eine Kante* (u, v)
4. *setze* $V = V \setminus \{u, v\}$, $E = e = (w, x) \in E \mid w, x \notin \{u, v\}$ *und* $C = C \cup \{u, v\}$.

Satz 1.5. *Der folgenden Ansatz benutzt außerdem einen Parameter k als Eingabe und entscheidet, ob es ein Vertex Cover der Größe $\leq k$ gibt.*

1. *FUNCTION VCover*(G, C, l)
2. *if* $E = \varnothing$ *return True*
3. *if* $l = k$ *return False*
4. *Wähle eine Kante* (u, v) *in* G
5. *if VCover*$(G \setminus \{u\}, C \cup \{u\}, l + 1)$
6. *return True*
7. *else return VCover*$(G \setminus \{v\}, C \cup \{v\}, l + 1)$

Die Laufzeit ist offensichtlich $2^k\mathcal{O}(n)$.

1.2 SAT

Eine Formel F ist in KNF gegeben, also als Konjunktion von Disjunktionen von Literalen. Eine Belegung ist eine Funktion, die jeder Variable einen Wert zuweist. Eine partielle Belegung weist nur einer Teilmenge aller Variablen Werte zu und lässt die restlichen undefiniert.

Lemma 1.6. *Reduktion von 3-Färbbarkeit auf SAT.*
Für einen Graphen G wollen wir eine Formel F konstruieren. Zunächst definieren wir die Menge der Variablen von F als

$$VAR = \{x_v^1, x_v^2, x_v^3 : \ v \in V\}$$

für drei verschiedene Farben. In einer Lösung muss jeder Knoten eine Farbe bekommen, d.h wir brauchen die $|V|$ Klauseln

$$(x_v^1 \vee x_v^2 \vee x_v^3)$$

Außerdem dürfen zwei benachbarte Knoten nicht dieselbe Farbe bekommen, das heißt, es ergeben sich die $3\,|E|$ Klauseln

$$(\neg x_v^1 \vee \neg x_w^1) \wedge (\neg x_v^2 \vee \neg x_w^2) \wedge (\neg x_v^3 \vee \neg x_w^3)$$

Satz 1.7. *Der folgende Backtracking Algorithmus bietet einen guten Ansatz, um eine Formel F auf Erfüllbarkeit zu überprüfen.*

1. *FUNCTION search(F, α partielle Belegung)*
2. *if α belegt alle Variablen: return $\alpha(F)$*
3. *if α belegt eine Klausel mit 0: return False*
4. *if search($F, \alpha 0$) = True: return True*
5. *else: return search($F, \alpha 1$)*

Dieser Algorithmus kann sogar noch weiter verbessert werden, indem zuerst nach Klauseln mit nur einem Literal gesucht wird und dann diese belegt werden. Wenn diese nicht existieren, wird nach Klauseln mit zwei Literalen gesucht und diese werden belegt. Erst dann wird eine beliebige Klausel (bzw. ein beliebiges Literal) gewählt. Die Laufzeit ist beschränkt durch $\mathcal{O}(7^{\frac{n}{3}})$.

Das Erfüllbarkeitsproblem, in dem jede Klausel aus maximal zwei Variablen besteht, wird 2-SAT genannt. Dieses Problem kann in polynomieller Zeit gelöst werden.

Satz 1.8. *Im folgenden wird ein probabilistischer Algorithmus für 2-SAT eingeführt.*

1. $\alpha := (0, 0, ..., 0)$
2. *for* $i = 0$ *to* $f(F)$
3. *if* $\alpha(F) = 1$*: return True*
4. *wähle eine zufällige Klausel* C *mit* $\alpha(C) = 0$
5. *wähle ein zufälliges Literal* x *aus* C
6. *setze* $\alpha(x) = \overline{\alpha(x)}$

Die Funktion $f : F \mapsto n \in \mathbb{N}$ *wird später definiert.*
Ist F *unerfüllbar, so funktioniert der Algorithmus korrekt. Ist* F *erfüllbar, so findet der Algorithmus eine erfüllende Belegung mit Wahrscheinlichkeit* $p \geq \frac{1}{2}$.
Ist α^* *eine erfüllende Belegung und* α *eine beliebige andere Belegung, mit Hamming-Abstand* i *zu* α^**, so definieren wir uns* $T(i)$ *als die erwartete Anzahl an Bit-Flips, die nötig sind, um* α *in eine erfüllende Belegung zu transformieren. Man kann sehen, dass* $T(n) = n^2$*. Die erwartete Anzahl an Bitflips ist also* $\mathcal{O}(n^2)$*. Wählt man nun* $f(F) = 2n^2$*, so ist die Erfolgswahrscheinlichkeit von* $\frac{1}{2}$ *garantiert.*

2 Probabilistische Algorithmen

2.1 Min-Cut

Dieses Problem ist in P. Eine effiziente Lösung ist durch den Ford-Fulkerson Algorithmus mithilfe dem Max-Flow/Min-Cut Lemmas möglich. Im Folgenden wird ein effizienter probabilistischer Algorithmus besprochen.

1. WHILE $|V| > 2$ DO
2. wähle $(u, v) \in_R E$
3. kontrahiere (u, v)
4. ENDWHILE

Satz 2.1. *Der Algorithmus hat Laufzeit $\mathcal{O}(n^2(\log n)^2)$. Der Algorithmus findet immer einen Schnitt. Der Algorithmus benötigt für jede Instanz $n-1$ Schritte. Für einen gegebenen minimalen Schnitt C ist die Wahrscheinlichkeit, dass der Algorithmus C findet ist $\geq \frac{2}{n(n-1)}$. Es genügen also $\mathcal{O}(n^2)$ Wiederholungen für eine konstant kleine Fehlerwahrscheinlichkeit.*

In jedem Schritt des Algorithmus wird gehofft, dass keine Kante e aus dem minimal cut C gezogen wird. Im letzten Schritt beträgt die Fehlerwahrscheinlichkeit dafür aber $\frac{2}{3}$. Um dem entgegen zu wirken, ist die Idee, den Algorithmus ein wenig umzubauen. Nach etwa $n\left(\frac{\sqrt{2}-1}{\sqrt{2}}\right)$ Schritten wird der bisher erzeugte Teilgraph H von G bestimmt. Für diesen Graphen wird der Algorithmus zwei separate Male ausgeführt und der kleinere der beiden Cuts wird gewählt. Eine Kontraktion ist in $\mathcal{O}(n)$ mithilfe der Adjazenzmatrix möglich, also wird die Laufzeit

$$T(n) = \underbrace{\left(1-\frac{1}{\sqrt{2}}\right)n}_{\text{Anzahl Runden}} \cdot \underbrace{cn}_{\text{Laufzeit Kontraktion}} + \underbrace{2T\left(\frac{n}{\sqrt{2}}\right)}_{\text{Laufzeit Rekursion}}$$

Mittels Master-Theorem lässt sich die Laufzeit bestimmen als $T(n) = \mathcal{O}(n^2 \log n)$.
Die Wahrscheinlichkeit, dass C nun die ersten $n\left(1-\frac{1}{\sqrt{2}}\right)$ Schritte überlebt, ist nun $\approx \frac{1}{2}$. Mittels induktivem Einsetzen kann überprüft werden, dass die Erfolgswahrscheinlichkeit $p(n) \geq \frac{1}{\log n}$ ist. Es genügen daher $\log n$ Wiederholungen für eine konstante Fehlerwahrscheinlichkeit. Die gesamte Laufzeit ist daher $\mathcal{O}(n^2(\log n)^2)$.

2.2 All pairs shortest paths (APSP)

Wird jeder Pfad explizit ausgegeben, so kann die Ausgabe sehr groß sein. Man kann leicht ein Beispiel konstruieren, wo die gesamte Länge aller kürzesten Pfade $\Omega(n^3)$ ist. In diesem Fall wäre ein effizienter Algorithmus rein akademisch, da die Ausgabe ohnehin hohe Laufzeit benötigt.

Definition 2.2. Das Problem wird gelöst mithilfe der Shortest Path Matrix $S \in \mathbb{R}^{n\times n}$. Diese ist definiert als $S_{i,j} = k \in V$, wenn k Nachbar von i in einem kürzesten $i-j$ Pfad ist. Bemerke, dass S nicht eindeutig ist, da es mehrere kürzeste Wege geben kann. Mittels Dijkstra kann diese Matrix in $\mathcal{O}(n^3)$ berechnet werden.

Der folgende Algorithmus berechnet S indem Matrizenmultiplikation verwendet wird. Sei $MM(n)$ die beste bekannte Laufzeit für Matrixmultiplikation, dann ist die Laufzeit von dem Algorithmus $\mathcal{O}(MM(n)\log^2 n)$.

Definition 2.3. Wir definieren die Distanzmatrix D mittels $d_{i,j}$ = Länge eines kürzesten $i-j$-Pfades. Wir schreiben $\delta(G)$ für den Durchmesser eines Graphen.

Lemma 2.4. *Für die Adjazenzmatrix eines Graphen G, $u, v \in V$ und $k \in \mathbb{N}$ ist A^k die Matrix aller k-Pfade. In anderen Worten $a^k_{u,v}$ ist die Anzahl der $u-v$-Pfade der Länge k.*

Für $Z = A^2$ kann man einen Graphen G' mit Kanten E' definieren, sodass

$$E' = E \cup \{(i,j) \mid d_{i,j} = 2\}$$

Die Adjazenzmatrix ist dann A' gegeben durch

$$a'_{i,j} = \begin{cases} 0, & i = j \vee (z_{i,j} = 0 \wedge a_{i,j} = 0) \\ 1, & \text{sonst} \end{cases}$$

Es folgt, dass A' in $MM(n)$ berechnet werden kann.
Es ist nun klar, dass falls $\delta(G) \leq 2$, dann ist $D = 2A' - A$.

Lemma 2.5. *Für alle $i, j \in V$ gilt $d_{i,j}$ gerade $\Rightarrow d_{i,j} = 2d'_{i,j}$ und $d_{i,j}$ ungerade $\Rightarrow d_{i,j} = 2d'_{i,j} - 1$.*

Lemma 2.6. *Für alle $i, j \in V$ und $\forall k \in N(i)$ ist $d_{i,j} - 1 \leq d_{k,j} \leq d_{i,j} + 1$ und $\exists k \in N(i)$ sodass $d_{i,j} - 1 = d_{k,j}$.*

Lemma 2.7. *$\forall i, j \in V$ gilt $\forall k \in N(i)$, dass $d_{i,j}$ gerade $\Rightarrow d'_{k,j} \geq d'_{i,j}$ und*
$d_{i,j}$ ungerade $\Rightarrow d'_{k,j} \leq d'_{i,j}$ und $\exists k \in N(i)$ sodass $d'_{k,j} < d'_{i,j}$.
Insbesondere folgt $d_{i,j}$ gerade $\Rightarrow \sum_{k \in N(i)} d'_{k,j} \geq d(i) \cdot d'_{i,j}$ und
$d_{i,j}$ ungerade $\Rightarrow \sum_{k \in N(i)} d'_{k,j} < d(i) \cdot d'_{i,j}$.
Weiter kann man sehen, dass $\sum_{k \in N(i)} d'_{k,j} = (A \cdot D')_{i,j}$.

Der Algorithmus zur Berechnung der D Matrix (genannt APD, all pairs distance) ist nun
Funktion APD(A Adjazenzmatrix von G):

1. $Z = A^2$
2. Berechne A'
3. if $\forall i, j \in V,\ i \neq j$ gilt $a'_{i,j} = 1$ then return $D = 2A' - A$
4. $D' = APD(A')$

5. $F = A \cdot D'$
6. for each $(i,j) \in V \times V$:
7. if $f_{i,j} \geq d'_{i,j} - 1$ then $d_{i,j} = 2d'_{i,j}$
8. else $d_{i,j} = 2d'_{i,j} - 1$
9. return D

Bezeichne $T(n,k)$ die Laufzeit des Algorithmus für einen Graphen auf n Knoten mit Durchmesser k. Wir wissen, dass $T(n,2) = MM(n)$. Außerdem ist

$$T(n,k) = MM(n) + \mathcal{O}(n^2) + T\left(n, \left\lceil \frac{k}{2} \right\rceil\right) + MM(n) = \mathcal{O}(MM(n) \cdot \log(n)$$

2.3 Boolean Product Witness

Seien $A, B \in \{0,1\}^{n \times m}$.

Definition 2.8. Wir definieren $A \odot B = C$ durch

$$c_{i,j} = \vee_{k=1}^{n} a_{i,k} \wedge b_{k,j}$$

Die BPW Matrix ist definiert durch

$$w_{i,j} = \begin{cases} 0, & c_{i,j} = 0 \\ k, & c_{i,j} = 1 \text{ und } a_{i,k} \wedge b_{k,j} = 1 \end{cases}$$

Im zweiten Fall heißt $w_{i,j} = k$ ein Zeuge. Außerdem definiert man die Matrix $\hat{A}$ durch $\hat{a}_{i,k} = ka_{i,k}$. Diese kann in $\mathcal{O}(n^2)$ berechnet werden.

Gibt es für (i,j) genau einen Zeugen, so gilt

$$(\hat{A} \cdot B)_{i,j} = k$$

Sei nun $w > 0$ die Anzahl der Zeugen für eine Position (i,j). Sei außerdem $r = 2^s$ für ein $s \in \mathbb{N}$, sodass

$$\frac{n}{2} \leq r \cdot w \leq n$$

Unter Gleichverteilung werden nun zufällig r Elemente aus $\{1, ..., n\}$ gezogen und daraus die Menge R erstellt. Es gilt dann

$$\mathbb{P}[\text{in } R \text{ gibt es genau einen Zeugen für } (i,j)] = p = \frac{w\binom{n-w}{r-1}}{\binom{n}{r}}$$

Lemma 2.9. $p \geq \frac{1}{2^e}$

Definition 2.10. Wir definieren B^R als die Matrix, die aus B gewonnen wird, wenn alle Zeilen z mit $z \notin R$ auf 0 gesetzt werden.

Falls es in R genau einen Zeugen l für A, B gibt, so gilt $(\hat{A}B^R)_{i,j} = l$. Auf dieser Basis kann nun ein Algorithmus zur Lösung des BPW Problems konstruiert werden.

1. $W = -A \cdot B$
2. for $t = 0$ to $\lfloor \log n \rfloor$
3. $r = 2^t$
4. for $l = 1$ to $\lceil 2e \log n \rceil$
5. wähle $R \in_R \{1, ..., n\}$ mit $|R| = r$
6. Berechne $\hat{A}$, B^R und $Z = \hat{A}B^R$.
7. for (i, j)
8. if $w_{i,j} < 0$ and $z_{i,j}$ ist Zeuge then $w_{i,j} = z_{i,j}$
9. endfor
10. endfor
11. endfor
12. for (i, j) finde Zeuge $w_{i,j}$

Die Laufzeit des Algorithmus ist beschränkt auf $\mathcal{O}(MM(n) \cdot \log^2 n)$. Mit dieser Überlegung kann nun das APSP Problem gelöst werden. Dafür ist anzumerken, dass für $j \in N_G(i)$ gilt, dass die Distanz zu k $d_{i,k} = d_{j,k} + s$ für $s \in \{-1, 0, 1\}$ erfüllt.

1. $D = APD(A)$
2. for $s \in \{0, 1, 2\}$
3. Berechne $D^{(s)}$ via $d^{(s)}_{i,j} = 1 \Leftrightarrow d_{i,j} = s - 1 \mod 3$
4. $W^{(s)} = BPW(A, D^{(s)})$
5. endfor
6. Berechne S via $s_{i,j} = w_{i,j}^{d_{i,j} \mod 3}$

Die Gesamtkomplexität des Algorithmus ist $\mathcal{O}(MM(n) \log^2 n)$.

2.4 Perfektes Matching auf bipartiten Graphen

Das Problem ist effizient deterministisch zu lösen, aber Algorithmen sind üblicherweise sehr kompliziert. Wir stellen daher stattdessen einen einfachen probabilistischen Algorithmus vor.

Definition 2.11. Sie $G = (V_1 \cup V_2, E)$ ein biparttier Graph mit $|V_1| = |V_2| = n$. Man definiert die $n \times n$ Bipartitionsmatrx A_G durch

$$A_G = \begin{cases} x_{i,j}, & (i,j) \in E \\ 0, & \text{sonst} \end{cases}$$

Die Determinante D der Matrix wird wie üblich bestimmt als

$$\det(A) = \sum_{\pi \in S_n} sgn(\pi) \prod_{i=1}^{n} a_{i,\pi(i)}$$

Es ist offensichtlich, dass $\det(A_G)$ ein Polynom in den $x_{i,j}$ ist.

Satz 2.12. *Es gibt ein perfektes Matching in* $G \Leftrightarrow \det(A_G) \not\equiv 0$.

Zu bestimmen, ob die Determinante der Matrix das Null-Polynom ist, kann im Allgemeinen sehr aufwändig sein. Ein probabilistischer Ansatz ist es, zufällige Werte in das Polynom einzusetzen und zu überprüfen, ob die Funktion 0 ausgibt. Dabei hilft das folgende Lemma.

Lemma 2.13. *Sei g ein Polynom mit m Variablen und Grad $\leq d$. Dann hat $p \leq mdM^{m-1}$, wenn die Parameter $a_1, ..., a_m \in \{0, 1, ..., M-1\}$.*

Für das Polynom der Determinanten gilt $m \leq n^2, d = 1$. Der Algorithmus zum Überprüfen, ob das Polynom dem Nullpolynom entspricht, besteht nun daraus, zufällige Werte für die $x_{i,j}$ zu wählen und die Determinante der daraus entstandenen Matrix zu berechnen.

1. Berechne A_G
2. belege $x_{i,j} \in_R \{0, ..., M-1\}$ für alle i, j
3. setze diese Werte in A_G ein und bezeichne das Resultat als A'_G
4. if $\det(A'_G) \neq 0$: es gibt perfektes Matching
5. else: es gibt (wahrscheinlich) kein perfektes Matching

Der Algorithmus hat eine Fehlerwahrscheinlichkeit $\leq \frac{n^2}{M}$. Wählt man $M = 2n^2$, so ist die Fehlerwahrscheinlichkeit $\leq \frac{1}{2}$.

2.5 Perfektes Matching auf allgemeinen Graphen

Aus offensichtlichen Gründen gehen wir davon aus, dass n gerade ist.

Definition 2.14. Wir definieren wieder eine leicht veränderte Adjazenzmatrix

$$(A_G)_{i,j} = \begin{cases} 0, & (i,j) \notin E \\ x_{i,j}, & (i,j) \in E \wedge i < j \\ -x_{i,j}, & (i,j) \in E \wedge j < i \end{cases}$$

Ein Matching ist dann eine Permutation π mit $\pi^2 = id$ und $\pi(i) \neq i$. Die Idee ist nun, wieder über die Determinante der Matrix zu argumentieren. Es gilt

$$\det(A_G) = \sum_{\pi \in S_n} sgn(\pi) \prod_{i=1}^{n} a_{i,\pi(i)}$$

Es gilt wieder das selbe Resultat wie für bipartite Graphen.

Lemma 2.15. *G hat ein perfektes Matching* $\Leftrightarrow \det(A_G) \not\equiv 0$.

Es ist klar, dass der Algorithmus für bipartite Graphen dann auch für allgemeine Graphen funktioniert.

2.6 Perfektes Matching finden

Mit den vorherigen Teilen kann nun bestimmt werden, ob in einem Graphen ein Matching existiert. Ein solches zu finden, benötigt aber zunächst einige Überlegungen.

Definition 2.16. Sei $S = \{x_1, ..., x_n\}$ eine endliche Menge. $F = \{S_1, ..., S_k\}$ sodass $S_j \subseteq S$ heißt Mengensystem. Es ist klar, dass $|F| \leq 2^n$
Sei $w : S \to \{1, 2, ..., 2n\}$ eine Funktion die jedem Element aus S ein Gewicht zuordnet. $w(S_j)$ sei eine verkürzte Schreibweise für $\sum_{x \in S_j} w(x)$.

Lemma 2.17 (Isolierungslemma)**.** *Ist w zufällig und unter Gleichverteilung gewählt, so ist*

$$\mathbb{P}[\exists! S_i \in F : w(S_i) \ \textit{minimal}\ ] \geq \frac{1}{2}$$

Die Idee ist nun, mit diesem Lemma einen Algorithmus zur Bestimmung eines perfekten Matchings in bipartiten Graphen zu konstruieren. Auf Existenz eines perfekten Matchings können wir bereits überprüfen.
Sei nun $G = (U \cup V, E)$ ein bipartiter Graph mit perfektem Matching und $|U| = |V|$.

Wir statten die Kanten mit Gewichten w in $\{1, 2, ..., 2m\}$ aus. Im Rahmen des Lemmas sein $S = E$ und F die Menge der perfekten Matchings. Wir definieren $A \in \mathbb{R}^{n \times n}$ mit

$$a_{i,j} = \begin{cases} 1, & (u_i, v_j) \in E \\ 0, & \text{sonst} \end{cases}$$

Daraus definieren wir eine zweite Matrix B mit

$$b_{i,j} = \begin{cases} 2^{w_{i,j}}, & (u_i, v_j) \in E \\ 0, & \text{sonst} \end{cases}$$

Lemma 2.18. *Falls G genau ein perfektes Matching mit minimalem Gewicht w hat, so ist $\det(B) \neq 0$ und 2^w ist die größte Zweierpotenz die $\det(B)$ teilt.*

Lemma 2.19. *Gibt es in G genau ein perfektes Matching M minimalen Gewichts w, dann gilt*

$$(u_i, v_j) \in M \Leftrightarrow \frac{\det(B_{i,j})}{2^w} \mod 2 = 0$$

Dabei ist $B_{i,j}$ die Matrix B mit 0 an der Stelle (i, j).

Der Algorithmus ist nun klar. Man überprüft einfach für jede Kante, ob das obige Lemma erfüllt ist. Wenn ja, wird sie in das Matching aufgenommen.

2.7 Rot-Blau-perfektes Matching

Das folgende Problem kann mit einem probabilistischen Algorithmus effizient gelöst werden. Ein polynomieller deterministischer Algorithmus ist hingegen nicht bekannt.
Gegeben sei ein bipartiter Graph $G = (U \cup V, E)$ mit $|U| = |V| = n$ und mit einer Färbung c der Kanten $c : E \to \{r, b\}$, jede Kante ist also rot oder blau. Außerdem gibt es ein $k \leq n$. Die Frage ist nun, gibt es ein perfektes Matching mit genau k roten Kanten? Die Idee ist wieder, das Isolationslemma zu verwenden. Dafür wählen wir $S = E$ und F als die Menge der perfekten Matchings mit genau k roten Kanten. Weiter definieren wir uns $f : E \to \{p, 2p\}$ durch

$$f(e) = \begin{cases} 2p, & c(e) = r \\ p, & c(e) = b \end{cases}$$

wobei $p = 3n^3$. Daraus wird nun wieder eine Matrix gebaut, $B \in \mathbb{R}^{n \times n}$ mit

$$b_{i,j} = \begin{cases} 2^{w_{i,j} + f((i,j))}, & (i, j) \in E \\ 0, & \text{sonst} \end{cases}$$

Für ein perfektes Matching σ mit genau k roten Kanten gilt nun

$$\prod_{i=1}^{n} b_{i,\sigma(i)} = 2^{w(\sigma)+(n+k)p} \leq 2^{n\cdot 2m+(n+k)p} \leq 2^{2n^3+(n+k)p} \leq 2^{(n+k+1)p}$$

Es folgt, dass die Terme in $\det(B)$, die ein perfektes Matching mit mehr als k roten Kanten implizieren, diesen Term nicht annullieren können. Es gibt außerdem höchstens $n!$ viele perfekte Matchings mit weniger als k roten Kanten. Die Summe solcher Terme ist kleiner als $2^{n^2+(n+k-1)p+2n^3}$ was wiederum kleiner als $2^{w(\sigma)+(n+k)p}$ ist. Diese Summe kann den Term also auch nicht annullieren.
Der Algorithmus wählt die Gewichtsfunktion w zufällig und berechnet $\det(B)$. Das ist eine Summe von Zweierpotenzen. Der Algorithmus gibt aus, dass es ein perfektes Matching mit k roten Kanten gibt, genau dann, wenn in dieser Summe eine Zweierpotenz 2^r vorkommt, sodass $2^{(k+1)p} < 2^r < 2^{(k+n)p}$.
Interessant an diesem Problem ist, dass es ein NP Problem ist. Das heißt, man kann es probabilistisch zwar effizient lösen, ein deterministischer, polynomieller Algorithmus ist aber nicht bekannt.

2.8 Probabilistische Algorithmen in exponentieller Zeit

2.8.1 3-Färbbarkeit

Das Problem ist NP-vollständig. Wir werden zeigen, dass es einen probabilistischen Algorithmus mit Laufzeit $\mathcal{O}(p(n)c^n) := \mathcal{O}^*(c^n)$ gibt.
Die Idee des Algorithmus ist es, das Graphenproblem in ein SAT Problem umzuschreiben. Für jeden Knoten $v \in V$ und jede Farbe $j \in \{1, 2, 3\}$ wählen wir eine boole'sch Variable $x_{v,j}$, die 1 ist, genau dann, wenn v Farbe j bekommt. In der Formel F_G gibt es nun zwei Arten von Klauseln.

1. $\forall v \in V$: $(x_{v,1} \vee x_{v,2} \vee x_{v,3})$
2. $\forall (u, v) \in E$: $(\overline{x_{v,1}} \vee \overline{x_{u,1}}) \wedge (\overline{x_{v_2}} \vee \overline{x_{u,2}}) \wedge (\overline{x_{v,3}} \vee \overline{x_{u,3}})$

Eine erfüllende Belegung der Formel impliziert nun eine Färbung des Graphen.
Der Algorithmus wählt für jeden Knoten v eine Farbe j, die für v ausgeschlossen wird. D.h. v wird nicht mir j gefärbt und $x_{v,j} = 0$. Die Klauseln von Typ 1 werden dadurch reduziert auf zwei Literale und dadurch ist F_G reduziert auf eine Formel in der Familie 2-SAT. Diese lässt sich effizient lösen. Wir berechnen nun die Wahrscheinlichkeit, dass die reduzierte Formel F_G lösbar ist, wenn gegeben ist, dass G 3-färbbar ist. Sei f eine korrekte 3-Färbung für G. Die Wahrscheinlichkeit, dass der Algorithmus für einen beliebigen Knoten v genau die Farbe $f(v)$ ausschließt, ist $\frac{1}{3}$. Die Wahrscheinlichkeit, dass kein

Knoten seine Farbe verliert, ist also $\left(\frac{2}{3}\right)^n$. Um die Wahrscheinlichkeit für einen Fehler gering zu halten, wird der Algorithmus einfach l mal wiederholt. Bei $l = 100\left(\frac{3}{2}\right)^n$ ist der die Fehlerwahrscheinlichkeit $\leq 2^{-100}$. Die Laufzeit ist demnach $\mathcal{O}(p(n)\left(\frac{3}{2}\right)^n) = O^*(\left(\frac{3}{2}\right)^n)$.

2.8.2 Random Walk Algorithmus für 3-SAT

Der folgende Algorithmus versucht, 3-SAT zu lösen.

1. beginne mit einer zufälligen Belegung α
2. if $\alpha(F) \neq 1$, dann wähle eine Klausel C, sodass $\alpha(C) = 0$
3. wähle ein Literal u_l in C und verändere α, sodass u_l erfüllt (und damit C erfüllt)
4. wiederhole n mal
5. wenn α nicht zu einer erfüllenden Belegung umgewandelt werden konnte, wähle eine neue
6. wiederhole t mal

Sei α_0 eine erfüllende Belegung, die Hamming-Distanz p zur initialen Belegung α hat. Dann transformiert der random walk α in α_0 mit Wahrscheinlichkeit 3^{-p}. Wird die Laufzeit nun aber verlängert auf (zum Beispiel) $3p$ Schritte, dann dürfen p Schritte des random walks falsch sein, wenn dafür $2p$ richtig sind. Die Wahrscheinlichkeit, dass α_0 gefunden wird, ist damit $\binom{3p}{2p}\left(\frac{1}{3}\right)^{2p}\left(\frac{2}{3}\right)^p \approx 2^{-p}$. Man kann zeigen, dass die Annahme $p = \frac{n}{3}$ optimal ist. Deswegen wird im Algorithmus der random walk auf eine Länge von $n = 3p$ beschränkt.
Zur weiteren Untersuchung definieren wir

- E_1 ist das Zufallsereignis, dass $d(\alpha_0, \alpha) = \frac{n}{3}$
- E_2 ist das Zufallsereignis, dass der Algorithmus $\frac{2n}{3}$ Schritte in die richtige und $\frac{n}{3}$ Schritte in die falsche Richtung macht.

Es gilt dann, dass die Erfolgswahrscheinlichkeit $\geq \mathbb{P}[E_1 \wedge E_2]$ ist. Es gilt

$$\mathbb{P}[E_1] = \binom{n}{\frac{n}{3}} \cdot 2^{-n}$$

und

$$\mathbb{P}[E_2] = \binom{n}{\frac{n}{3}} \left(\frac{1}{3}\right)^{\frac{2n}{3}} \left(\frac{2}{3}\right)^{\frac{n}{3}}$$

Damit sieht man, dass die Erfolgswahrscheinlichkeit $\geq \frac{1}{q(n)} \left(\frac{3}{4}\right)^n$ ist. Wählt man $t = 20 \cdot \left(\frac{4}{3}\right)^n$, so ergibt sich insgesamt eine Fehlerwahrscheinlichkeit von $\leq e^{-20}$. Die Laufzeit ist $\mathcal{O}^*(\left(\frac{4}{3}\right))^n$.

3 Approximationsalgorithmen

Sei P ein Problem und X eine Menge von Instanzen. Für $x \in X$ nennen wir S_x die Menge der Lösungen für x. Auf der Menge $S = \bigcup_{x \in X} S_x$ definieren wir die Kostenfunktion m. Gesucht ist $opt(x)$ der minimale (bzw. maximale) Wert einer Lösung für alle Instanzen x.

Definition 3.1. A ist ein ε-Approximationsalgorithmus, wenn

$$m(A(x)) \geq (1 - \varepsilon) opt(x)$$

für Maximierungsprobleme und

$$m(A(x)) \leq \frac{1}{1 - \varepsilon} opt(x)$$

Definition 3.2 ((F)PTAS)**.** Ein Approximationsalgorithmus A für ein Problem P ist ein PTAS, wenn A mit Eingaben x und ε eine ε-approximative Lösung für x produziert und die Laufzeit polynomiell in $|x|$ ist. Hängt die Laufzeit zusätzlich von $\frac{1}{\varepsilon}$ ab, so ist A ein FPTAS.

3.1 Max-Cut

Gegeben $G = (V, E)$. Finde $C \subset V$, sodass

$$m(C) = |\{(u, v) \mid u \in C \; v \notin C\}|$$

maximal. Das Entscheidungsproblem ist NP-vollständig.

Definition 3.3. Sei $C \subseteq V$ und $v \in V$. Dann definieren wir

$$C \Delta \{v\} = \begin{cases} C \setminus \{v\}, & v \in C \\ C \cup \{v\}, & v \notin C \end{cases}$$

Damit können wir einen Algorithmus bauen.

1. $C = \varnothing$

2. while $\exists v \in V$ mit $m(C\Delta v) > m(C)$

3. $C = C\delta\{v\}$

4. end

Lemma 3.4. *Die Laufzeit ist $\mathcal{O}(n^4)$. Die gefundene Lösung C erfüllt $|C| \leq \frac{1}{2}opt$.*

3.2 Max-k-SAT

Sei $F = C_1 \wedge \cdots \wedge C_m$ eine Formel in der jede Klausel genau k Literale hat. Finde eine Belegung $\alpha : Var(F) \to \{0,1\}$, die

$$m(\alpha) = |\{C_i \mid \alpha(C_i) = 1\}|$$

maximiert. Wir geben einen Algorithmus mit Approximationsfaktor $\varepsilon = \frac{1}{2^k}$ an.
Sei dafür $a \in_R \{0,1\}^n$ eine zufällige Belegung. Wir definieren $f(F)$ als die erwartete Anzahl an Klauseln die eine zufällige Belegung erfüllt. Dafür definieren wir die Zufallsvariable

$$X_i = \begin{cases} 1, & \alpha \text{ erfüllt } C_i \\ 0, & \text{sonst} \end{cases}$$

Offensichtlich ist $\mathbb{P}[X_i] = \mathbb{P}[\alpha(C_i) = 1] = 1 - \frac{1}{2^{|C_i|}} = 1 - \frac{1}{2^k}$. Wir schließen

$$f(F) = m\left(1 - \frac{1}{2^k}\right)$$

Dieser Wert lässt sich aber auch anders bestimmen. Sei $\#(F, \alpha)$ die Anzahl der Klauseln in Fm die von α erfüllt werden. Sei $\overline{a} = (a_2, a_3, ..., a_n)$.

$$f(F) = \sum_{\alpha \in \{0,1\}^n} \mathbb{P}[a = \alpha] \cdot \#(F, \alpha) = \sum_{\alpha \in \{0,1\}^{n-1}} \mathbb{P}[a = 0\alpha] \cdot \#(F, 0\alpha) + \sum_{\alpha \in \{0,1\}^{n-1}} \mathbb{P}[a = 1\alpha] \cdot \#(F, 1\alpha)$$

$$= \frac{1}{2}\left(\sum_{\alpha \in \{0,1\}^{n-1}} \mathbb{P}[\overline{a} = \alpha] \cdot \#(F, 0\alpha) + \sum_{\alpha \in \{0,1\}^{n-1}} \mathbb{P}[\overline{a} = \alpha] \cdot \#(F, 1\alpha)\right)$$

$$= \frac{1}{2} \sum_{\alpha \in \{0,1\}^{n-1}} \mathbb{P}[\overline{a} = \alpha] \cdot (\#(F, 0\alpha) + \#(F, 1\alpha)) \leq \sum_{\alpha \in \{0,1\}^{n-1}} \mathbb{P}[\overline{a} = \alpha] \cdot \#(F|_{x_1=0}, \alpha) = f(F|_{x_1=0})$$

Das motiviert den Algorithmus

1. for $i = 1$ to n

2. $p_0 = f(F|_{x_i=0})$, $p_1 = f(F|_{x_i=1})$

3. if $p_0 \geq p_1$: $F = F|_{x_i=0}$

4. else: $F = F|_{x_i=1}$

5. endif

6. endfor

3.3 0-1 Rucksackproblem

Wir betrachten eine Menge an n Objekten i, die jeweils mit einem Gewicht g_i und einem Wert v_i ausgestattet sind. Wir bezeichnen mit G die Größe des Rucksacks. Für eine bestimmte Instanz x des Problems bezeichne S_x die Menge aller Lösungen. Wir beginnen mit einem Approximationsalgorithmus für $\varepsilon = \frac{1}{2}$.
Der Greedy-Algorithmus sortiert die Elemente nach ihrem Kosten-Nutzen-Verhältnis

$$\frac{v_1}{g_1} \geq \frac{v_2}{g_2} \geq \cdots \geq \frac{v_n}{g_n}$$

Ist i das erste Objekt, das nicht mehr zum Rucksack ergänzt werden kann, wenn die ersten $i-1$ Elemente gewählt wurden, dann wähle die bessere der beiden Lösungen $S_1 = \{1, 2, \ldots, i-1\}, S_2 = \{i\}$. Es ist nicht schwer zu sehen, dass der Algorithmus eine $\frac{1}{2}$-Faktor Approximation erreicht.
Für das Problem existiert außerdem ein FPTAS. Dieser basiert auf einem Ansatz der dynamischen Programmierung. Hierbei kann die Zielfunktion $w(i, h)$ rekursiv wie folgt bestimmt werden

$$w(i,h) = \begin{cases} 0, & h = 0 \\ w(i-1,h), & g_i > h \\ \max\{w(i-1,h), w(i-1,h-g_i) + v_i\} & g_i \leq h \end{cases}$$

Auf diesem Ansatz aufbauend, kann man sich eine Art duale Formulierung des Problems vorstellen. Dabei wird die Funktion

$$g(i,h) = \min_{S \subseteq \{1,2,\ldots,i\}} \{\sum_{i \in S} g_i \mid \sum_{i \in S} v_i \geq h\}$$

die sich wiederum rekursiv ermitteln lässt

$$g(i,h) = \begin{cases} 0, & h = 0 \\ \infty, & \sum_{j=1}^{i} v_i < h \\ \min\{g(i-1,h), g(i-1,h-v_i) - g_i\}, & \text{sonst} \end{cases}$$

Die Tabellengröße ist dabei $n^2 v_{\max}$.
Mit diesem Ansatz lässt sich nun ein FPTAS formulieren. Dafür werden zunächst die Werte v_i binär gerundet. Das heißt, wir definieren

$$v_i' = 2^b \lceil \frac{v_i}{2^b} \rceil$$

Hierbei ist b eine Konstante. Die Transformation wirkt sich auf die Binärdarstellung von v_i so aus, dass die letzten b Stellen auf 0 gesetzt werden. Führt man nun den obigen Algorithmus auf die Werte $(\frac{v_1'}{2^b}, \frac{v_2'}{2^b}, ..., \frac{v_n'}{2^b})$ aus, so ist die Laufzeit nun durch $n^2 \frac{v_{\max}}{2^b}$ beschränkt. Wir bezeichnen

$$m(A(x)) = \sum_{i \in S'} v_i$$

die vom Algorithmus gefundene Lösung und

$$opt(x) = \sum_{i \in S} v_i$$

die optimale Lösung. Es gilt dann

$$F = \frac{opt(x) - m(A(x))}{m(A(x))} \leq \frac{n \cdot 2^b}{v_{\max}} \leq \varepsilon$$

Für den FPTAS wird nun b so gewählt, dass

$$2^b \leq \frac{\varepsilon v_{\max}}{n} < 2^{b+1}$$

3.4 Approximierbarkeit

3.4.1 Travelling Salesman

Gegeben sei die Distanzmatrix D eines vollständigen Graphen G. Gesucht ist ein kürzester Hamilton Pfad. Wir haben bereits gesehen, dass dieser als eine Permutation π der Knoten geschrieben werden kann.

Satz 3.5. *Falls $P \neq NP$, so existiert kein ε-Approximationsalgorithmus für TSP in polynomieller Zeit.*

3.4.2 Clique

Das Cliquen-Problem in einem Graphen G besteht darin, eine maximale Clique C zu finden. Auch dieses Problem lässt sich nicht approximieren.

Satz 3.6. *Falls es ein ε mit einem ε-Approximationsalgorithmus für das Cliquen Problem gibt, so gibt es einen solchen Algorithmus für alle $\varepsilon \in (0, 1)$.*

3.5 Δ-TSP

Das Δ-TSP oder metrisches TSP Problem bezeichnet die Suche nach einem minimalen Hamilton-Pfads in einem Graphen G, der die Dreiecksungleichung erfüllt.

Definition 3.7. Eine quadratische Matrix D heißt Metrik, wenn alle ihre Einträge ≥ 0 sind und $\forall i, j, k$ gilt

$$d_{i,k} \leq d_{i,j} + d_{j,k}$$

Lemma 3.8. *$HK \leq \Delta$-TSP*

Wir beschreiben nun zwei Approximationsalgorithmen für das metrische TSP. Gegeben sei die Distanzmatrix D.

1. Finde einen minimalen Spannbaum T
2. Verdopple jede Kante in T, bezeichne diesen Graphen als T_E
3. Bestimme einen Euler-Kreis C in T_E
4. Generiere H aus C, indem alle Knoten c_i mit $\exists j < i$ sodass $c_j = c_i$ gelöscht werden.

Satz 3.9. *Es ist nicht schwer zu sehen, dass dieser Algorithmus einen Approximationsfaktor von $\frac{1}{2}$ erreicht.*

Der folgende Algorithmus nach Christofides erreicht sogar eine $\frac{1}{3}$-Faktor Approximation.

1. Finde einen minimalen Spannbaum T in G
2. Sei V' die Menge der Knoten mit ungeradem Grad in G
3. Finde kleinstes perfektes Matching M in V'
4. Sei $G_E = T \cup M$
5. Finde Euler-Kreis in G_E
6. Bestimme Hamilton-Kreis wie zuvor

Satz 3.10. *Der Algorithmus erreiche eine $\frac{1}{3}$-Faktor Approximation an Δ-TSP.*

3.6 PCP Theorem

PCP steht für Probabilistically Checkable Proofs. Dieses Resultat wird verwendet, um zu zeigen, dass manche Probleme nur bis zu einem bestimmten Faktor approximiert werden können. Die Klasse NP kann charakterisiert werden, als die Menge der Probleme, die polynomiell beweisbar sind.

Definition 3.11. Seien $r, q : \mathbb{N} \to \mathbb{N}$. Ein (r, q)-Beweis-Verfizierer für $L \subseteq \Sigma^*$ ist ein Polynomialzeit-Algorithmus A, der für Eingabe x und einen Beweis y (für x) $r(|x|)$ Zugallsbits generiert. $A(x, r)$ generiert eine Menge an Stellen $i_1, ..., i_{p(|x|)} \in \mathbb{N}$. $A(x, r, y(i_1), ...y(i_p)) \in \{0, 1\}$ bestimmt of $x \in L$ oder $x \notin L$.

Definition 3.12. Eine Sprache L ist in der Komplexitätsklasse $PCP(r, q)$, wenn es einen (r, q) Beweis-Verfizierer gibt mit den Eigenschaften

- $x \in L \Rightarrow \exists$ Beweis B, sodass $\mathbb{P}[A(x, r, B) = 1] = 1$
- $x \notin L \Rightarrow \forall$ Beweise B gilt $\mathbb{P}[A(x, r, B) = 1] \leq \frac{1}{2}$.

Satz 3.13.

- $NP = PCP(\mathcal{O}(\log n), \mathcal{O}(1))$
- $NP = PCP(\mathcal{O}(\log n), 3)$
- $P = PCP(\mathcal{O}(\log n), 2)$

Satz 3.14. *Falls $P \neq NP$, so existiert kein ε-Approximationsalgorithmus für Max-k-SAT mit $\varepsilon < \frac{1}{16}$.*

3.7 #DNF-SAT

Im folgenden betrachten wir Formeln nicht mehr wie bisher in KNF sondern in DNF. Überprüfen auf Erfüllbarkeit einer DNF ist einfach, da eine Formel F in DNF nicht erfüllbar ist, genau dann, wenn jede Klausel eine Variable und ihr Komplement enthält. Das Ziel des #DNF-SAT Problems ist es, zu ermitteln, wie viele erfüllende Belegungen existieren.

Definition 3.15. Sei $\mathcal{D}$ die Menge aller Formeln, die in DNF vorliegen. Wir definieren

$$f : \mathcal{D} \to \mathbb{N}$$

$$F \mapsto |\{\alpha : V(F) \to \{0, 1\}^n \mid \alpha(F) = 1\}|$$

die Funktion, die einer Formel F die Anzahl der erfüllenden Belegungen zuordnet.

Ziel dieser Sektion ist es, einen probabilistischen Algorithmus A zu konstruieren, der

$$\mathbb{P}[(1-\varepsilon)f(F) \leq A(F) \leq (1+\varepsilon)A(F)] \geq 1-\delta$$

erfüllt und eine Laufzeit hat, die polynomial in $|F|, \frac{1}{\varepsilon}$ und $\log(\frac{1}{\delta})$ ist.

Bemerkung 3.16. Ist F in KNF gegeben, so kann man $f(F)$ ermitteln mittels $f(F) = 2^n - f(\neg F)$. Es gilt $f \in SAT \Leftrightarrow f(\neg F) < 2^n$.

Bemerkung 3.17. Dieses Problem ist ein Spezialfall eines allgemeineren Problems. Sei U eine beliebige Menge und $|U|$ bekannt. Angenommen, es ist möglich schnell Elemente von U unter Gleichverteilung zu wählen und angenommen, es gibt eine Funktion $f : U \to \{0,1\}$. Wir wollen für

$$G = \{u \in U \mid f(u) = 1\}$$

die Kardinalität $|G|$ ermitteln.

Der folgende probabilistische Ansatz liefert eine erste Möglichkeit

1. $k = 0$
2. for $i = 1$ to N
3. wähle $u_i \in_R U$
4. if $f(u_i) = 1$ then $k = k + 1$
5. output $\frac{|U|}{N}k$

Satz 3.18. *Für* $0 < \varepsilon, \delta < 1$ *und* $N \geq \frac{4|U|}{\varepsilon^2|G|} \ln \frac{2}{\delta}$ *gilt die Abschätzung*

$$\mathbb{P}[(1-\varepsilon)\,|G| \leq A(U) \leq (1+\varepsilon)\,|G|] \geq 1-\delta$$

Bemerkung 3.19. Initial ist $|G|$ nicht bekannt, also kann man N nicht passend wählen. Außerdem ist es möglich, dass N nicht polynomial in $|F|$ wächst.

Wir verfolgen einen besseren Ansatz. Sei dafür eine Formel in DNF

$$F = I_! \vee I_2 \vee \cdots \vee I_m$$

mit den sogenannten Implikanten I_i. Definiere

- $U = \{(\nu.i) \mid \nu(I_i) = 1\}$ wobei ν erfüllende Belegungen für einen Implikanten sind
- $H_i = \{\nu \mid \nu(I_i) = 1\}$ und

- $G = \{\nu \mid \nu(F) = 1\}$

Es gilt $|G| \leq |U| \leq \sum_{i=1}^{m} |H_i|$.
Seien außerdem

$$f(\nu, j) = \begin{cases} 1, & i = \min\{j \mid \nu \in H_j\} \\ 0, & \text{sonst} \end{cases}$$

und $H = \{(\nu, j) \in U \mid f(\nu, j) = 1\}$. Es gilt $|H| = |G|$. Mit diesen Überlegungen kann nun ein Algorithmus formuliert werden.

1. $k = 0$
2. for $i = 1$ to N
3. wähle $(\nu, j) \in_R U$
4. if $f(\nu, j) = 1$ then $k = k + 1$
5. output $|U| \frac{k}{N}$

Satz 3.20. *Für $N \geq \frac{4}{\varepsilon^2} m \ln \frac{2}{\delta}$ ist die Ungleichung in Satz 3.18 erfüllt. Insbesondere ist die Größe von N unabhängig von $|G|$.*

Wieso können wir aber ein Stichprobe unter Gleichverteilung aus U wählen? Dafür schreiben wir $\forall j$

$$|H_j| = 2^{n - \#(\text{Literale in } H_j)}$$

Damit kann der Algorithmus ein wenig umformuliert werden zu

1. $k = 0$
2. for $i = 1$ to N
3. wähle $j \in \{1, 2, \ldots, m\}$ mit Wahrscheinlichkeit $\frac{|H_j|}{|U|}$
4. wähle $\nu \in_R H_j$
5. if $f(\nu, j) = 1$ then $k = k + 1$
6. output $|U| \frac{k}{N}$

Es folgt, dass $\mathbb{P}[(\nu, j)] = \frac{|H_j|}{|U|} |H_j| = \frac{1}{|U|}$.

Satz 3.21. *Sei U eine endliche Menge und $|U|$ bekannt. Seien $H_1, H_2, \dots H_m \subseteq U$ nicht notwendigerweise disjunkte Teilmengen und*

$$H = \bigcup_{i=1}^{m} H_i$$

Es gelte außerdem, dass

- *$|H_i|$ in polynomialer Zeit berechenbar ist*
- *Man kann aus H unter Gleichverteilung eine Stichprobe entnehmen*
- *$\nu \in H_i \ \forall i$ kann effizient entschieden werden*

Dann folgt: $|H|$ kann in polynomieller Zeit in $n, \frac{1}{\varepsilon}, \log \frac{1}{\delta}$ approximiert werden, wobei die Ungleichung aus Satz 3.18 erfüllt ist.

4 Parametrisierte Algorithmen

Definition 4.1. Ein parametrisiertes (Entscheidungs-) Problem ist ein Paar (L, Σ) wobei

1. $L \subseteq \Sigma^* \times \Sigma^*$ oder
2. $L \subseteq \Sigma^* \times \mathbb{N}$

Beispiel 4.2. *1.* $L_1 = \{(G, k) \mid G$ *hat ein Clique* $\geq k\}$

2. $L_2 = \{(G, k) \mid G$ *hat Maximalgrad* $\Delta(G) = k$ *und ist planar*$\}$

Für solche Probleme verwenden wir die Komplexitätsklasse FPT (Fixed Parameter Tractable). Solche Probleme sind algorithmisch lösbar in $\mathcal{O}(f(k) \cdot p(|x|))$ für ein Polynom p und eine beliebige Funktion f in k. p ist unabhängig von k und f ist unabhängig von n. Für L_1 in Beispiel 4.2 existiert ein Algorithmus mit Laufzeit $\mathcal{O}(n^k k^2)$, was die Definition für FPT nicht erfüllt. Tatsächlich ist unbekannt, ob $L_1 \in$ FPT.

Bemerkung 4.3. $P \subseteq FPT$.

4.1 Tiefenbeschränkte Suchbäume

Wir führen diese Strategie anhand des Vertex Cover Problems ein. Dieses Problem lässt sich schreiben als $\{(G, k) \mid G$ hat ein VC der Größe $\leq k\}$.
Wir benötigen zuerst einen Suchbaum. Dieser besteht aus Knoten (A, C) wobei A ein Teilgraph von G ist und $C \subseteq V$ mit $|C| \leq k$ eine Menge von Knoten ist. Die Strategie ist jetzt klar. Man berechnet alle möglichen Knotenmengen C mit $|C| \leq k$ und

dazugehörigen Teilgraphen $A = G - C$ und erhält ein Vertex Cover C^*, wenn ein Knoten $(A = (V_A, \varnothing), C^*)$ erreicht wird. Ansonsten existiert kein Vertex Cover der Größe $\leq k$. Der Suchbaum beginnt in der Wurzel $(G, \varnothing)$ und jeder Kindknoten entsteht aus seinem Elternknoten indem ein $v \in V_A$ entfernt und zu C_A hinzugefügt wird. Dieser Baum hat maximale Tiefe k und damit höchstens 2^k Knoten. Die Berechnung eines Knoten im Suchbaum benötigt Laufzeit n (wird ein Knoten entfernt, so auch alle inzidenten Kanten, das sind $\leq n - 1$ viele). Zum Überprüfen, ob eine Knotenmenge C_A ein Vertex Cover impliziert, benötigt Laufzeit m (es muss überprüft werden, ob in A jede Kante überdeckt wird). Die Laufzeit ist demnach $\mathcal{O}(2^k(n+m))$ und damit ist $VC \in FPT$.

Ein weiteres Problem, das sich mit dieser Technik lösen lässt, ist Hitting Set. Dieses Problem beruht auf einer endlichen Menge $S = \{a_1, a_2, ..., a_n\}$ und einem Mengensystem $\mathcal{P}(S) \supseteq F = \{X_1, X_2, ..., X_m\}$. Die Frage ist, ob es eine Menge $H \subseteq S$ mit $|H| \leq k$ gibt, sodass $H \cap X_i \neq \varnothing \ \forall i \in [m]$. Dafür nehmen wir außerdem an, dass $|X_i| \leq d$ für alle i. Wir konstruieren wieder den Suchbaum. Die Knoten sind in diesem Fall (F', H), wobei $F' = \{X'_1, X'_2, ..., X'_l\} \subseteq F$ und $H \subseteq S$. Die Wurzel ist wieder $(F, \varnothing)$ und aus einem Knoten (F', H) entstehen seine Kinder, indem ein Element $a' \in S \setminus H$ gewählt wird und $F'' = F' \setminus \{X_j \mid a' \in X_j\}$ und $H' = H \cup \{a'\}$ berechnet wird. Es existiert ein Hitting Set H^* mit $|H^*| \leq k$ genau dann, wenn es einen Knoten $(\varnothing, H^*)$ gibt. Der Suchbaum hat höchstens d^k Knoten und daher ein Laufzeit von $\mathcal{O}(d^k \cdot p(n, m))$.

Als letztes Beispiel für diese Technik betrachten wir Independent Sets auf planaren Graphen. Wir benötigen dafür zwei einfache Resultate.

Lemma 4.4. *In einem planaren Graphen G mit n Knoten, m Kanten und f Regionen gilt $n - m + f = 2$.*

Lemma 4.5. *In einem planaren Graphen G gibt es einen Knoten von Grad ≤ 5.*

Der Suchbaum besteht nun aus Knoten (H, S) mit planaren Teilgraphen H und $S \subseteq V$. Die Wurzel ist $(G, \varnothing)$. Für jeden inneren Knoten wählen wir ein $v \in V_H$ von Grad höchstens 5 und wir konstruieren die Kinder, indem entweder v oder einer seiner Nachbarn in das IS hinzugefügt wird und dieser Knoten mit allen seinen Nachbarn aus H entfernt wird. Es folgt daher, dass der Suchbaum Maximalgrad 6 hat. Gibt es ein IS der Größe mindestens k, so muss entweder v oder einer seiner Nachbarn darin enthalten sein und daher ist eine der 6 Optionen richtig. In so einem Fall gibt es also einen Knoten $(\varnothing, S^*)$ im Suchbaum und S^* ist in Independent Set der Größe k. Die Größe des Baumes ist beschränkt durch 6^k, also erreicht man eine Laufzeit von $\mathcal{O}(6^k p(n))$.

4.2 Reduktion auf den Problemkern

Die Idee ist es, eine Probleminstanz I der Größe $|I| = n$ auf eine kleinere Instanz I' der Größe $|I'| \leq f(k)$ zu reduzieren. Dieses Problem ist dann nur noch abhängig vom Parameter k. Wir demonstrieren dieses Verfahren anhand des Vertex Cover Problems. Gegeben sei $G = (V, E)$. Gesucht ist ein Vertex Cover H der Größe $\leq k$. Bemerke: gibt es $u \in V(G)$ mit $d(u) > k$, dann ist $u \in H$. Darauf basiert der folgende Algorithmus:

1. PHASE 1:
2. Sei $H = \{v \in V \mid d(v) > k\}$
3. if $|H| > k$: es gibt ein VC der Größe $\leq k$
4. else: $G' = G \setminus H$, $k' = k - |H|$.
5. PHASE 2:
6. if $|E(G')| > k \cdot k'$: es gibt kein VC der Größe $\leq k$
7. PHASE 3:
8. Suche ein VC der Größe $\leq k'$ in G'

Die Laufzeit dieses Algorithmus kann mittels Implementierung durch Adjazenzlisten auf $\mathcal{O}(kn + k^2 2^k)$ gehalten werden.

Als zweites Beispiel betrachten wir das 3-Hitting Set Problem. Hierbei handelt es sich um einen Spezialfall des Hitting Set Problems mit $|X_j| \leq 3 \; \forall X_j \in F$. Wir machen folgende Beobachtgungen: Ist H ein Hitting Set so gelten

- Falls es $> k$ Tupel X_j der Art $\{x, y, *\}$ gibt und $\exists H$ Hitting Set mit $|H| \leq k$ folge $x \in H$ oder $y \in H$. In F ersetzen wir daher alle Tupel $\{x, y, *\}$ durch $\{x, y\}$. (HS1)
- Falls x Element von $> k^2$ Tupeln X_j entfernen wir diese aus F und ergänzen x zu H. (HS2)

Nach diesen beiden Reduktionen gibt es für jedes $x \in S$ höchstens k^2 Tupel, die x enthalten. Ein Algorithmus könnte nun die Instanz durch die beiden obigen Regeln reduzieren und anschließend mittels Suchbaum eine Lösung finden. Das geschieht in Laufzeit $\mathcal{O}(p(m) + 3^k k^3)$.

Ein letztes Beispiel für die Reduktion ist das Feedback Vertex Set. Gegeben ein Multigraph G, $F \subseteq V$ ist ein Feedback Vertex Set, wenn $G - F$ ein Baum ist. Die Frage ist nun, ob ein FVS F der Größe $\leq k$ existiert. Wie im vorherigen Beispiel gehen wir dafür eine Reihe von Regeln durch. Sei initial $H = \varnothing$.

1. Falls es eine Schleife (d.h. $(v, v) \in E(G)$) gibt, setze $H = H \cup \{v\}, k = k-1$ (FVS1)
2. Hat eine Kante e Multiplizität $l > 2$, entferne $l - 2$ Kopien davon (FVS2)
3. Falls es einen Knoten v mit $d(v) \leq 1$ gibt, $G = G - v$ (FVS3)
4. Gibt es einen Knoten v mit $d(v) = 2$, kontrahiere eine der beiden zu v inzidenten Kanten (FVS4)
5. Ist $k < 0$, so existiert kein FVS der Größe $\leq k$.

Nach Anwendung aller 5 Regeln hat jeder Knoten in G' Grad ≥ 3. sei $V_{3k} = v_1, v_2, ..., v_{3k}$ die Menge der $3k$ Knoten mit höchstem Grad. Falls es in G' ein FVS F mit $|F| \leq k$ gibt, so gilt $V_{3k} \cap F \neq \varnothing$. An dieser Stelle kann wieder ein Suchbaum mit beschränkter Tiefe angewandt werden und man erhält einen Lösung für das FVS Problem in Laufzeit $\mathcal{O}(p(n)(3k)^k)$. Dafür benötigen wir aber noch zwei Lemmata.

Lemma 4.6. *Ist F ein FVS in $G = (V, E)$, so ist*

$$\sum_{v \in F} (d(v) - 1) \geq |E| - |V| + 1$$

Lemma 4.7. *Jedes FVS F in G' enthält mindestens einen Knoten in V_{3k}.*

4.3 Kronenzerlegung

Hierbei handelt es sich um eine andere Methode, Probleme auf den Problemkern zu reduzieren. Das geschieht mittels einer Graphenzerlegung.

Definition 4.8. Für einen ungerichteten Graphen G ist eine Kronenzerlegung eine Partition der Knotenmenge

$$V = K \dot{\cup} H \dot{\cup} H \dot{\cup} R$$

sodass

1. $K \neq \varnothing$ ist eine unabhängige Menge
2. es gibt ein Matching von H und K, das H komplett überdeckt
3. H trennt K und R

Satz 4.9. *In bipartiten Graphen G gilt*

$$\min\{|C| \mid C \text{ vertex cover}\} = \max\{|M| \mid M \text{ matching}\}$$

Lemma 4.10. *Es gibt einen Algorithmus für das folgende Problem auf Graphen mit mehr als $3k$ Knoten:*
Berechne Kronenzerlegung oder ein Matching der Größe mindestens $k+1$.

Wir zeigen nun, wie die Kronenzerlegung verwendet werden kann, um ein Vertex Cover zu bestimmen. Wir nehmen im Sinne des vorherigen Lemmas an, dass die Instanz G keine isolierten Knoten besitzt. Ist $|V| < 3k$, so ist G bereits auf den Problemkern reduziert. Wir nehmen daher an, dass $|V| \geq 3k$.
Wir verwenden den Algorithmus aus dem vorherigen Lemma und erhalten daher entweder ein Matching m mit $|M| \geq k$ oder eine Kronenzerlegung. Im ersten Fall gilt aufgrund des Satzes von König, dass kein k-Vertex Cover existiert. Im zweiten Fall, wenn es eine Kronenzerlegung $V = K \cup H \cup R$ gibt, so muss ein Vertex Cover C jede Kante vom Matching von H in K überdecken. Es ist immer günstiger, von solchen Kanten den Knoten aus H zu wählen. Daher können wir die Reduktion $(G', k') = (G \setminus H, k - |H|)$ anwenden.

Ein weiteres Beispiel für diese Reduktion ist die Duale Färbung. Bemerke, dass sich normale Knotenfärbungen nicht FPT-parametrisieren lassen, da bereits das 3-Färbbarkeitsproblem NP-vollständig ist. Bei Problem der Dualen Färbung, wollen wir einen Graphen in $n - k$ Farben färben, statt in k. Dieses Problem ist in FPT. Wir benötigen für die folgenden Überlegungen den Komplementärgraphen $\bar{G}$ von G. Wir können annehmen, dass G keine universellen Knoten hat, da diese eine eigene Farbe benötigen und wir diese daher entfernen können. Daher hat $\bar{G}$ keine isolierten Knoten. Falls G höchstens $3k$ Knoten hat, so ist das Problem bereits reduziert. Wir nehmen daher an $|V| \geq 3k$. Das selbe gilt natürlich für $\bar{G}$. Wir wenden wieder das Lemma an, diesmal auf $\bar{G}$. Angenommen, es gibt ein Matching $|M| \geq k + 1$. Fpr eine Matching Kante (u, v) in $\bar{G}$, gilt also, dass $(u, v) \notin E(G)$ und daher können die Knoten mit der selben Farbe gefärbt werden. Deshalb können wir $k + 1$ Farben für mindestens 2 Knoten verwende und der Graph ist $n-k$-färbbar. Im zweiten Fall erhalten wir ein Kronenzerlegung $V = K \cup H \cup R$ für $\bar{G}$. Wir reduzieren (G, k) wiederum auf (G', k') wobei $G' = G \setminus (K \cup H)$ und $k' = k - |H|$. Die Korrektheit dieser Regel folgt, weil K eine unabhängige Menge in $\bar{G}$ und daher eine Clique in G ist. Diese Knoten benötigen daher paarweise verschiedene Farben. Die Knoten in R müssen mit jeweils anderen Farben gefärbt werden, da K und R in $\bar{G}$ keine Kanten teilen und in G daher vollständing mit einander verbunden sind. Mit einer ähnlichen Argumentation können aber die Knoten in H mit den Farben von K gefärbt werden.

4.3.1 Graphenmodifikationsproblem

Beim $H_{i,j,k}$ Graphenmodifikationsproblem handelt es sich um eine weitere Anwendung des Reduktion auf den Problemkern.

Definition 4.11. Sei $U \subseteq V$ eine Teilmenge der Knoten. Wir bezeichnen mit $G[U]$ den von U induzierten Teilgraphen.

Definition 4.12. Eine Grapheneigenschaft Π ist eine Menge von Graphen. $G \in \Pi$ heißt Π-Graph. Π heißt vererbbar, falls jeder induzierte Teilgraph von G die Eigenschaft auch erfüllt. Ein Beispiel für so ein Π ist Bipartitheit.

Definition 4.13. Man kann Grapheneigenschaften auch über verbotene Teilgraphen definieren.

$$G \in \Pi \Leftrightarrow \ \forall H \in \mathcal{F} : \ H \not\subset G$$

wobei $\mathcal{F}$ eine Menge verbotener Teilgraphen ist.

Man kann zeigen

Lemma 4.14. Π *ist vererbbar genau dann, wenn* Π *mittels verbotener Teilgraphen charakterisiert werden kann.*

Definition 4.15. Eine Grapheneigenschaft Π heißt abgeschlossen unter Minorenbildung, wenn

$$G \in \Pi \wedge H < G \implies H \in \Pi$$

Satz 4.16. *Sei* Π *unter Minorenbildung abgeschlossen, dann existiert eine Menge* $O = \{H_1, ..., H_k\}$ *von Graphen, sodass*

$$G \in \Pi \Leftrightarrow H_i \not< G \ \forall i = 1, 2, ..., k$$

O heißt eine Obstruktionsmenge.

Wir wenden uns nun dem Graphenmodifikationsproblem zu. Gegeben eine Grapheneigenschaft Π und ein Graph G, lässt sich G mittels

- $\leq i$ Löschen von Knoten
- $\leq j$ Löschen von Kanten
- $\leq k$ Einfügen von Kanten

in einen Π-Graphen verwandeln?

Lemma 4.17. *Gegeben sei ein Graph G und eine vererbbare Grapheneigenschaft Π. Existiert ein Algorithmus A, der in Zeit $T(n)$ überprüfen kann, ob $G \in \Pi$, so gibt es einen Algorithmus, der in Zeit $\mathcal{O}(n \times T(n))$ einen minimalen induzierten verbotenen Teilgraphen H von G bestimmt.*

Satz 4.18. *Sei Π eine Grapheneigenschaft mit einer endlichen Menge verbotener Teilgraphen. Dann liegt das dazugehörige Modifiaktionsproblem in FPT mit Laufzeit $\mathcal{O}(f(i,j,k) \cdot n^c)$.*

Lemma 4.19. *Seien G und H zwei Graphen. Die Frage, ob $H < G$ kann in $\mathcal{O}(f(|H|) \cdot |G|^3)$ beantwortet werden.*

4.4 Farbkodierung und Hashing

Dies ist eine weitere Technik, um Probleme in Abhängigkeit eines Faktors k effizient zu lösen. Wir beginnen mit einer Einführung am Pfadproblem.
Gegeben sei ein Graph G mit n Knoten und eine Zahl $k \in \mathbb{N}$. Existiert ein Pfad der Länge k? Wir verwenden dafür das Konzept einer k-Färbung $c : V \to \{1, 2, ..., k\}$.

Definition 4.20. Ein Teilgraph H von G heißt voll bunt, wenn alle Knoten verschieden gefärbt sind.

Lemma 4.21. *Sei P ein k-Pfad in G und c eine zufällige Färbung der Knoten in G. Dann gilt*

$$\mathbb{P}[P \text{ ist voll bunt}] = \frac{k!}{k^k} > e^{-k}$$

Lemma 4.22. *Sei G ein Graph und c eine Färbung der Knoten mit k Farben. Existiert ein voll bunter k-Pfad, so kann dieser in $2^{\mathcal{O}(k)} |E|$ gefunden werden.*

Man sieht also, dass dieser Algorithmus mindestens $\mathcal{O}(e^k)$ oft ausgeführt werden müsste, um eine konstante Laufzeit zu erreichen. Das Problem ist, dass die Färbung zufällig gewählt wurde und es ist daher nicht sicher, ob ein voll bunter k-Pfad existiert. Wir definieren daher im folgenden eine Menge L von Färbungen, sodass gilt

$$V' \subseteq V, |V'| = k \implies \exists c \in L : \ |c(V')| = k \ \forall V' \subseteq V$$

Definition 4.23. Eine Familie F von Hashfunktionen $f : \{1, 2, ..., n\} \to \{1, 2, ..., k\}$ heißt perfekt, falls $\forall S \subseteq \{1, 2, ..., n\}$ mit $|S| = k$ gilt

$$\exists f \in F \ \forall a, b \in S : \ f(a) \neq f(b) \Leftrightarrow a \neq b$$

Satz 4.24. *Es ist möglich eine solche Familie F mit $|F| = 2^{\mathcal{O}(k)} \log^2 n$ zu konstruieren. Das geschieht in Laufzeit $2^{\mathcal{O}(k)} n \log^2 n$.*

Mit dem Resultat über einen voll bunten k-Pfad in zufälligen Färbungen ergibt sich nun, dass ein k-Pfad in FPT gefunden werden kann, falls einer existiert. Die genaue Laufzeit dafür ist $2^{\mathcal{O}(k)} \log^2 |V| \cdot |E|$.

Man kann auch das folgende Problem mittels Farbkodierung lösen:
Gegeben: Ein Graph G und ein Baum T mit $|T| = k$.
Gesucht: Ist T isomorph zu einem Teilgraphen H von T?
Wir beschreiben nun einen randomisierten Algorithmus, der einen Teilgraphen bestimmt (falls so einer existiert). Sei dafür $\varphi : T \to G$ eine Abbildung und $\varphi(T)$ der dazugehörige isomorphe Teilgraph von G. Wir beginnen mit einer zufälligen Färbung $f_{\{}1, 2, ..., n\} \to \{1, 2, ..., k\}$ der Knoten von G. Die Wahrscheinlichkeit, dass $\varphi(T)$ voll bunt ist, ist $< e^{-k}$.
Wir benötigen nun wieder eine Menge von Färbungen.
Wir fixieren einen Knoten $r \in V(T)$. Der Algorithmus berechnet für alle $w \in V(G)$ die Farbmenge aller bunten Kopien von $\varphi(T)$, in denen r auf w abgebildet werden. Die Menge solcher Farbklassen nennen wir $L_r^T(w)$. Ist $k = 1$, so ist $L_r^T(w) = \{f(w)\}$ und wir sind fertig. Sonst hat r einen Nachbarn r'. Wir spalten nun T in zwei Hälften T', T'', sodass $r \in T'$ und $r' \in T''$. Auf diesen beiden Teilgraphen wird das Verfahren rekursiv fortgesetzt. Wir erhalten also $L_r^{T'}(w)$ und $L_{r'}^{T''}(w)$ für alle w und fügen diese beiden Mengen zu $L_r^T(w)$ zusammen.
Dafür wird $\forall (u, v) \in E(G)$

- getestet, ob es $C \in F_r^{T'}(u)$, $C' \in F_{r'}^{T''}(v)$ gibt mit $C \cap C' = \varnothing$.
- Wenn schon, $L_r^T(w) = L_r^T(W) \cup \{C \cup C'\}$.

Die Laufzeit ist beschränkt durch $\mathcal{O}(4^k |E(G)|)$.

4.5 Baumzerlegungen

Definition 4.25. Sei G ein Graph. Eine Baumzerlegung von G ist ein Paar $(\{X_i \mid i \in V_T\}, T)$, sodass T ein Baum und $X_i \subseteq V(G)$ $\forall i$ eine Teilmenge der Knoten ist. Die Knoten des Baumes sind sogenannte Bags, die folgende Eigenschaften erfüllen

1. $\bigcup_{i \in V_T} X_i = V$
2. $\forall e = (u, v) \in V$ gibt es i sodass $u, v \in X_i$
3. $\forall i, j, k$ gilt, falls Bag j auf einem $i - k$ Pfad in T liegt, so ist $X_i \cap X_k \subseteq X_j$.

Definition 4.26. Die Weite einer Baumzerlegung ist definiert als

$$\max\{|X_i| - 1 \mid i \in V_T\}$$

Die Baumweite eines Graphen G ist definiert als

$$w(G) = \min\{k \mid \exists \text{ Baumzerlegung der Weite } k\}$$

Satz 4.27. *Sei G ein Graph mit Baumweite k. Dann gibt es einen Algorithmus, der eine optimale Baumzerlegung in $\mathcal{O}(f(k) \cdot n)$ findet, wobei $f(k) = 2^{35 \cdot k^2}$.*

Mithilfe einer Baumzerlegung und Prinzipien der dynamischen Programmierung können wir nun einige NP-Probleme effizienter lösen.

Gegeben einer Baumzerlegung der Weite k wollen wir zum Beispiel ein minimales Vertex Cover bestimmen. Für alle X_i können wir alle Vertex Cover für die Knoten in X_i in $f(k)$ berechnen. Die Idee ist nun, ausgehend von den Blättern der Bauzerlegung ein Vertex Cover iterativ zu bestimmen. Im folgenden sei G_i definiert als der induzierte Teilgraph von G bestehend aus allen Bags vor X_i in der Baumordnung (d.h. alle Bags im in X_i gewurzelten Teilbaum). Wir definieren $c : X_i \to \{0, 1\}$ mit $c(v) = 1 \Leftrightarrow v \in VC$. Wir halten alle solchen Färbungen von X_i in einer Tabelle fest. Dabei heißt c eine gültige Färbung, wenn sie auf X_i tatsächlich teil eines Vertex Cover ist.
Wir bewerten außerdem die Vertex Cover anhand ihrer Größe:

$$m(c) = \begin{cases} |c^{-1}(1)|, & \text{falls } c \text{ Vertex Cover ist} \\ \infty, & \text{sonst} \end{cases}$$

Wir können bisher nur einzelne Bags bezüglich eines Vertex Covers betrachten und bewerten. Angefangen bei den Blättern ist das ausreichend, wir wollen die Tabellen für die (induktiv) nächste Stufe in der Baumordnung erweitern.
Sei dazu $i, j \in V_T$ und X_j der Eltern-Bag von X_i. Für jede Färbung $c_\cap : X_i \cap X_j \to \{0, 1\}$ und jede Erweiterung $c_j : X_j \to \{0, 1\}$ von $c_\cap$ auf X_j ist

$$m(c_j) = m(c_j) + \min\{m(c_i) \mid c_i \text{ ist Erweiterung von } c_\cap\} - \left|c_\cap^{-1}(1)\right|$$

Es ist klar, dass c_j damit auf G_j erweitert wurde.
Wir wollen zum Schluss noch die Laufzeit des Algorithmus ermitteln. Die Tabelle einer Tasche ist $2^{|X_i|}$ Zeilen lang. Mithilfe einer geschickten Sortierung kann die Aktualisierung der Tabelle in $\mathcal{O}(2^{|X_i|} + 2^{|X_j|}) = \mathcal{O}(2^{w(G)} \cdot n)$ geschehen. Das wird zusammengefasst im

folgenden Satz:

Satz 4.28. *Sei G ein Graph und Z eine Baumzerlegung der Weite $w(Z)$. Dann kann ein minimales Vertex Cover in $\mathcal{O}(2^{w(Z)} \cdot n)$ bestimmt werden.*