Algorithmen für schwierige Probleme

November 21, 2024

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 $2^{(k+1)p} < 2^r < 2^{(k+n)p}$.