

Algorithmen und Wahrscheinlichkeiten

1. Graphentheorie
2. Minimal-Spanning-Tree
3. Algorithmus von Prim
4. Algorithmus von Kruskal
5. Zusammenhang
6. Artikulationsknoten Graph
7. Brücke
8. Implementierung Artikulationsknoten
9. Eulerkreis / Eulertour
10. Eulerweg / Eulerpfad
11. Hamiltonkreis
12. Travelling Salesmen
13. Matchings
14. Satz von Hall / Heiratsatz
15. Approximation Hamiltonkreis
16. Färbungen
17. Wahrscheinlichkeitstheorie
18. Laplace-Raum
19. Bedingte Wahrscheinlichkeit
20. Abhängige/unabhängige Ereignisse
21. Zufallsvariablen
22. Erwartungswert
23. Dichte Verteilung
24. Varianz/Normalverteilung
25. Bernoulli-Verteilung
26. Binomial-Verteilung
27. Geometrische-Verteilung
 - 27.1. Warten auf k. ten Erfolg
 - 27.2. Coupon Collector
28. Poisson-Verteilung
29. Mehrere Zufallsvariablen
30. Waldsche Identität
31. Ungleichung von Markov
32. Chebyshev-Ungleichung
33. Chernoff-Ungleichung
34. Monte-Carlo-Algorithmus
35. Las-Vegas-Algorithmus
36. Reduktion Fehlerwahrscheinlichkeit
37. Sortieren und Selektieren
38. Primzahntest
39. Target-Shooting
40. Entscheiden, welche Antwort ich nehme
41. Hashing
42. Bloomfilter
43. Lange Pfade
44. Bunte Pfade
45. Flüsse in Netzwerken
46. S-T-Schnitte
47. Augmentierende Pfade
48. Restnetzwerk
49. Ford-Fulkerson-Algorithmus
50. Bipartites Matching als Fluss
51. Konvexe Mengen der Flüsse
52. Minimale Schnitte in Multigraphen
53. Kleinstter umschließender Kreis
54. Zusammenfassung Laufzeiten

Graphentheorie

$$n = |V| \quad m = |E|$$

- Weg: Folge von Knoten (inkl. Mehrfachknoten)
- Pfad: Weg, welcher 1 Knoten x, y verbindet (alle Knoten versch.)
- Kreis: Start- Endpunkt ist gleich
- zusammenhängend: Wenn ich von jedem zu jedem Knoten komme
- Baum: zusammenhängender, kreisfreier Graph
- Wenn zwei Knoten verbunden sind \rightarrow adjazent
- Für jeden normalen Graphen gilt: $\sum_{v \in V} \deg(v) = 2|E|$

Minimal-Spanning-Tree

- Zu jedem zusammenhängenden Graph gibt es einen Spannbaum, der die gleiche Knotenmenge besitzt
 - Wir wollen minimalen Spannbaum finden, bei dem die Kosten minimal sind.
 - Die kleinste Kante ist immer im MST vorhanden
 - Blau Rege)
 - Wir haben Schnitt und dort haben wir keine Verbindung, deshalb färben wir minimale Kante des Schnitts blau ein
 - Rote Regeln)
 - Wenn C ein Kreis im Graph ist, und keine rote Kante besitzt, färbe die maximale Kante des Kreises rot
- \hookrightarrow Wenn wir beide Regeln nicht mehr anwenden können \rightarrow MST
- \hookrightarrow Blaue Kanten bilden MST
- \hookrightarrow Rote Kanten können auch sofort gelöscht werden

Algorithmus von Prim (Man lässt Baum durch Graphen wachsen)

- Der Algorithmus wird $|V|-1$ -Mal ausgeführt und nur die blaue Regel angewandt

- Somit haben wir keinen Kreis (ohne rote Regel)

1. Wähle Startknoten aus \rightarrow Untermenge S

2. While $S \neq$ komplette Knotenmenge

 2.1. Wähle kleinste Schnittkante aus

 2.2. Färbe Kante blau und füge Endknoten zu S

- Laufzeit: $O(\underbrace{n-1}_{|V|} \times \underbrace{\text{while-Schleife}}_{|E|})$ Durchlauf Adjazenzliste $\xrightarrow{|E|}$ Kanten

$$O(|V| \cdot |E|)$$

- Um Knoten, der neu dazukommt, effizient finden zu können: $\delta^-[x] :=$ Leichteste Kante von X in Menge S

$\text{pred}[x] :=$ gibt den Knoten in Teilmenge aus,

der über leichteste Kante verbunden ist

1. Wähle Startknoten v

2. Für alle Nachbarn von v , setze $\delta^-(x) = c(v, x)$ / $\text{pred}(x) = v$

3. Für alle nicht-Nachbarn $\delta^-(x) = \infty$ / $\text{pred}(x) = \text{undefined}$

4. Füge alle Schlüssel $\delta^-(x)$ in Priority Queue ein

5. While $S \neq$ komplette Knotenmenge

 5.1. $x \leftarrow \text{ExtractMin}(\text{Priority Queue})$

 5.2. Finde Anfangsknoten $\text{pred}(x)$

 5.3. Für alle Nachbarn vom neuen Knoten x

 5.4. Wenn $c(x, N(x)) < \delta^-(N(x))$

 5.4.1. $\delta^-(x) = c(x, N(x))$

 5.4.2. $\text{pred}(N(x)) = x$

- Laufzeit: Schleife + Priority Queue (mittels MinHeap)
 $O(|V|) \cdot O(\log |E|)$

Algorithmus von Kruskal (viele Bäume verwackeln zu grossen Bäumen)

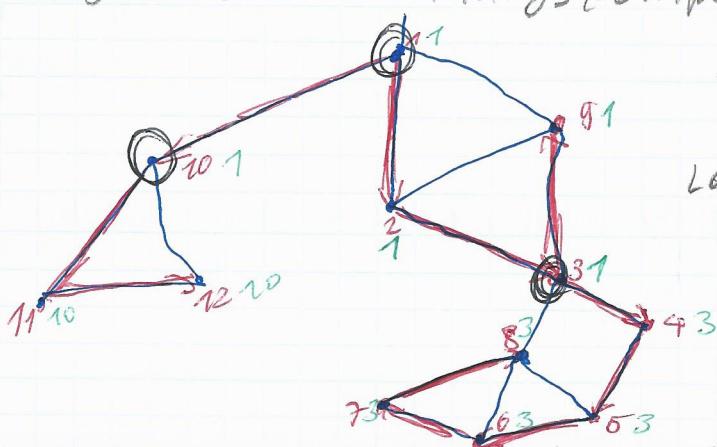
1. Sortiere Kanten aufsteigend
 2. Initialisiere UnionFind (jeder Knoten eigene Partition)
 3. Für alle Kanten
 - 3.1. $e_i = \{x, y\}$
 - 3.2. Wenn $\text{find}(x) = \text{find}(y) \rightarrow$ Farbe e_i rot
 - 3.3. ansonsten färbe e_i blau und $\text{union}(x, y)$
- Laufzeit: $O(\underbrace{|E|}_{\text{Schleife umhängen}} \cdot \underbrace{\log |V|}_{\text{umhängen}})$

Zusammenhang

- Wie viele Knoten/Kanten muss man entfernen, damit Graph nicht mehr zusammenhängend ist
- Wenn Graph k -zusammenhängend ist, dann kann ich mir zwei Knoten aussuchen, und finde dann k verschiedene Pfade von $u \rightarrow v$

Artikulationsknoten

- ii. Ein Knoten in einem zusammenhängenden Graph heißt Artikulationsknoten, wenn durch sein entfernen, die Zahl der Zusammenhangskomponenten sich erhöht"



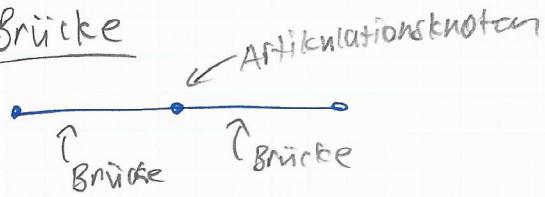
$\text{DFS}[v] :=$ Nummer
Tiefensuche

$\text{LOW}[v] :=$ für jeden Knoten
welche kleinste
DFS-Nummer
kann ich erreichen
1. Rückwärtsknoten
 ∞ Vorwärtsknoten

- Artikulationsknoten: $\text{low}(\text{kind } v') \geq \text{dfs}(v)$

• Wurzelknoten $\deg > 1$

Brücke



- Entweder $\deg(x) = 1$

- oder x ist Artikulationsknoten

- Graph mit Brücke ist 1-zusammenhängend

Implementierung Artikulationsknoten / Brücken

1. Wir erzeugen DFS-Baum

2. Low-Value bestimmen

$$\cdot \text{low}(v) = \min \{ \text{low}(\text{Rückwärtskante}), \text{low}(\text{kinder}) \}$$

Eulertour / Eulerkreis

- Weg im Graph, der alle Kanten ein Mal enthält

- Graph enthält Eulertour, wenn alle Gerade gerade sind

1. Wähle Startknoten

2. Wähle Nachbar beliebig

3. Setze Kante als besucht

4. Nimm Nachbarknoten als Anfangsknoten und wiederhole

dies so lange, bis wir ganz am Start sind

5. Führe Algorithmus solange aus, bis alle Kanten drin sind

- Laufzeit: $O(|E|)$

Eulerpfad / Eulerweg

- Wenn Start-/Endknoten nicht identisch sein müssen

- Alle Gerade gerade, außer zwei

↗
Anfangs-/Endpunkt

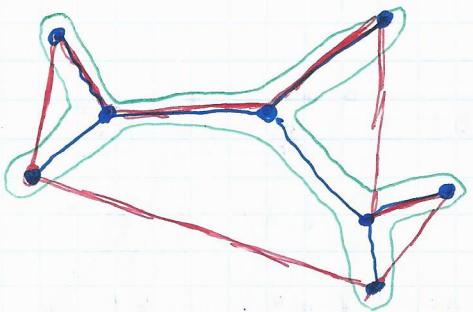
Hamiltonkreis

- Kreis, der alle Knoten genau ein Mal enthält
- Jeder Graph mit $|V| \geq 3$ und Minimalgrad $\geq \frac{|V|}{2}$
- Hamiltonkreis-Check (Dynamische Programmierung)
 - Wenn es Hamiltonkreis gibt, dann muss Knoten ① auch drin sein. Dann gibt es auch Knoten x , mit einem Pfad von $1 \rightarrow x$, der alle Knoten enthält
 - $P_{S,x} = \begin{cases} 1 & \text{es gibt einen } 1-x \text{ Pfad, der genau alle Knoten } \\ & \text{aus Teilmenge } S \text{ enthält} \\ 0 & \text{sonst} \end{cases}$
 - Für $|S|=2 \Rightarrow$ ziemlich einfach zu berechnen (=Nachbar)
 1. $|S|=2 / P_{S,x} = 1$, wenn x ist Nachbar von ①
 2. for all $s=3 \rightarrow n$
 - 2.1. for all $S \subseteq V$ mit $① \in S$ und $|S|=s$
 - 2.1.1. for all $x \in S$ mit $x=①$
 - 2.1.1.1. $P_{S,x} = P_{S \setminus \{x\}, y} : y \in N(x), y \neq 1$
 - enthält Graph einen Hamiltonkreis ist NP-vollständig
 \hookrightarrow Laufzeit: $n^2 \cdot 2^n$

Traveling Salesmen

- Kürzeste Rundreise
- Metrischer Graph falls Δ -Ungleichung gilt $d(a,c) \leq d(a,b) + d(b,c)$

 Minimal spanning Tree

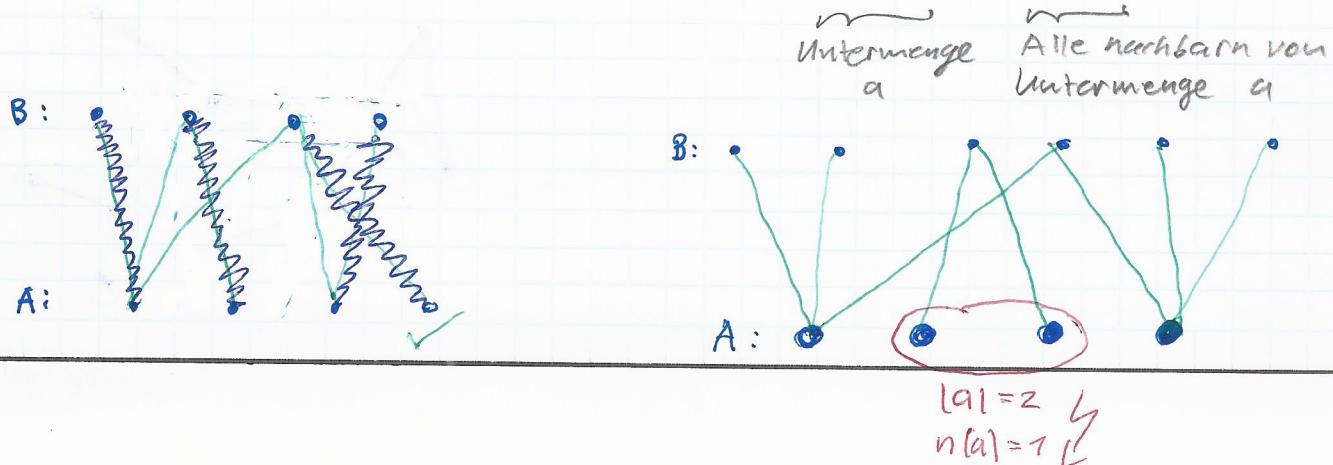


Matchings

- Teilmenge der Kantenmenge eines Graphs
 - Perfectes Matching: Wenn alle Grade = 1 (Gibt es nicht immer)
 - Kardinalitätsmaximales Matching: Größtmögliches Matching
↳ Kann auch perfekt sein
 - Inklusionsmaximales Matching: Nicht erweiterbar
↳ Nicht bestes Matching
 - Greedy-Inklusions-Matching
1. $M \leftarrow \emptyset$
2. while es gibt noch edges im Graph
2. 1. wähle eine Kante zufällig
 2. 2. gib Kante in M
 2. 3. Lösche Kante von Graph und alle benachbarten Kanten
- ↳ Erzeugt inklusions-Matching
3. Kardinalitätsmaximales-Matching
1. Greedy
 2. Finde ungerade Pfade:
 3. Dadurch haben wir Kardinalitäts-Matching
- Bei geraden Pfaden ist kardinalitätsmaximales = inklusionsmaximales

Satz von Hall (=Heiratssatz)

- Bipartiter Graph enthält Matching mit Kanten $|A|$, genau dann, wenn $\forall a \subseteq A: |a| \leq N(a)$



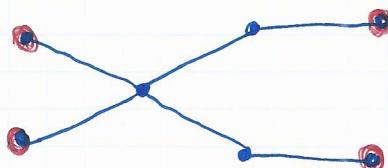
1/2 Approximation Hamiltonkreis

- Aus MST & Matching kann man Hamiltonkreis erstellen, der max. $1.5 \times$ länger ist, als minimaler Hamiltonkreis

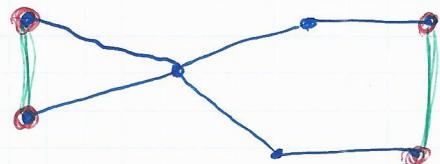
1. Berechnung MST für Graph G



2. Wir markieren Knoten mit ungeraden Grad

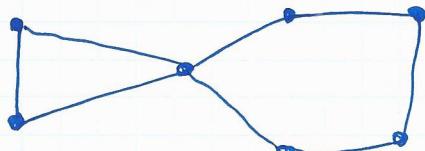


3. Berechne minimales Matching der ungeraden Knoten

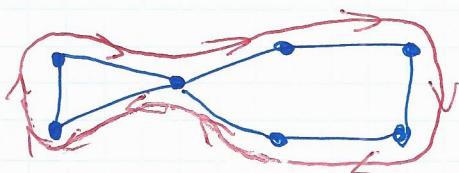


4. Vereinige Kanten zu einem Graph

Alle Knoten haben geraden Grad



5. Berechne Eulertour



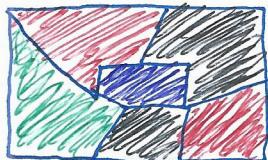
6. Konstruiere durch Abkürzungen Hamiltonkreis

↳ Dreiecksungleichung

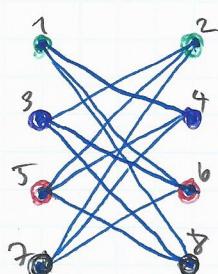
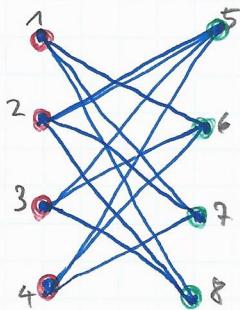


Färbungen

- Jeder Knoten hat eine Farbe, wobei zwei Knoten, die über eine Kante verbunden sind, nicht die gleiche Farbe haben dürfen
- Chromatische Zahl: Minimale Anzahl an Farben für Graph, damit dieser bunt ist: $\chi(G)$
- 4-Farben-Theorem: Man benötigt für planaren Graph nur maximal 4 Farben (z.B. Landkarte)



- Partite Graphen:
 - Bipartit: 2 färbbar
 - k -partit: k färbbar
- Greedy-Algorithmus:
 - Liefert zulässige Färbung mit $[\max_{v \in V} \deg(v)] + 1$ Farben
 - Das macht er in $O(|E|)$
 - 1. Wähle Reihenfolge der Knoten beliebig
 - 2. $c[V_1] = 1$ // Gebe Knoten „Farbe“ 1
 - 3. for $i=2$ bis n do
 - 3.1. $c[V_i] = \min \{ k \in N \mid k \neq c[N(v_i)] \}$
 - Auf die Reihenfolge kommt es an!



Wahrscheinlichkeitstheorie

Elementarereignisse

- Ergebnismenge: $\Omega = \{w_1, w_2, w_3, \dots, w_k\}$
- Summe aller Elementarereignisse $\sum_{i=1}^k w_i = 1 / 100\%$
- Eine Menge $E \subseteq \Omega$ heißt Ereignis
 - $\Pr[\emptyset] = 0$
 - $\Pr[\Omega] = 1$
 - $0 \leq \Pr[E] \leq 1$

- Gegenereignis von A: $\bar{A} = \Omega \setminus A$ (Alles andere, außer A)

Addition von Teilmengen

Überschneidungen

$$\Pr[A_1 \cup A_2] = \Pr[A_1] + \Pr[A_2] - \Pr[A_1 \cap A_2]$$

• Für mehr als zwei Vereinigungen: Siebformel

• Wenn Siebformel zu kompliziert: Boolesche Ungleichung
Union Bound

• Union Bound / Boolesche Ungleichung

$$\hookrightarrow \Pr\left[\bigcup_{i=1}^n A_i\right] \leq \sum_{i=1}^n \Pr[A_i]$$

Laplace-Raum

- Endlicher Wahrscheinlichkeitsraum

- Alle Elementarereignisse haben gleiche Wahrscheinlichkeit

$$\hookrightarrow \Pr[E] = \frac{|E|}{|\Omega|}$$

uniform verteilt

Bedingte Wahrscheinlichkeiten

- Wir schränken Elementarereignis mit zusätzlichen Bedingungen ein

\hookrightarrow wenn sich deshalb Wahrscheinlichkeit verändert \rightarrow bedingt wahrsch.

$$\Pr[A|B] = \frac{\Pr[A \cap B]}{\Pr[B]} \quad \begin{array}{l} \text{"A bedingt B"} \\ \text{"Wir bedingen B und wollen wahrsch. A"} \end{array}$$

Abhängige / unabhängige Ereignisse

- Unabhängig, g.d.w.:

$$\cdot \Pr(A \cap B) = \Pr(A) \cdot \Pr(B)$$

$$\cdot \Pr(A|B) = \frac{\Pr(A \cap B)}{\Pr(B)} = \frac{\Pr(A) \cdot \Pr(B)}{\Pr(B)} = \Pr(A)$$

- Um bei unabhängigen Ereignissen auszurchnen, mit welcher Wahrscheinlichkeit A & B zugleich eintreten, muss man diese einfach miteinander multiplizieren

$$\hookrightarrow \Pr[A \cap B] = \Pr[A] \cdot \Pr[B]$$

- Bei mehr als zwei Ereignissen gilt Multiplikationsatz

$$\hookrightarrow \Pr[A_1 \cap \dots \cap A_n] = \Pr[A_1] \cdot \Pr[A_2|A_1] \cdot \Pr[A_3|A_1 \cap A_2] \dots \text{abhängig}$$

$$\hookrightarrow \Pr[A_1 \cap \dots \cap A_n] = \Pr[A_1] \cdot \Pr[A_2] \cdot \Pr[A_3] \dots \cdot \Pr[A_n] \text{ unabhängig}$$

↳ Dadurch kann geprüft werden, ob mehrere Mengen unabhängig sind

- Physikalisch unabhängig \Rightarrow stochastisch unabhängig

- Satz der totalen Wahrscheinlichkeit

• Wir wollen B wissen, obwohl $B \subseteq A_1 \cup \dots \cup A_n$

$$\cdot \Pr[B] = \sum_{i=1}^n \underbrace{\Pr[B|A_i] \cdot \Pr[A_i]}_{\Pr[B \cap A_i]}$$



Umkehrung der Frage

- Satz von Bayes

$$\cdot \Pr[A_i|B] = \frac{\Pr[B|A_i] \cdot \Pr[A_i]}{\Pr[B]}$$

↳ Damit kann man Reihenfolge der Bedingung umdrehen

Zufallsvariablen

- Indikatorvariable, wenn entweder 1/0 $\rightarrow \mathbb{E}[X] = p_1[X]$
- Zufall gibt Variable einen Wert
- $X = X_1 + X_2 + X_3$ (Große Zufallsvariable kann mehrere Kleine haben)

Erwartungswert

- Welcher Wert durchschnittlich bei einer Zufallsvariable zu erwarten ist

$$\mathbb{E}[X] = 0 \cdot 0,3 + 0,10 \cdot 0,4 + 0,23 \cdot 0,2 + 1,0 \cdot 0,1 = 0,19$$

$$\mathbb{E}[X] = \sum_{x \in \omega_X} x_i \cdot \Pr[X=x_i]$$

$$\text{- Erwartungswert ist linear: } \mathbb{E}[X] = \mathbb{E}[X_1] + \mathbb{E}[X_2] + \mathbb{E}[X_3]$$

Betrag	Pr
0	0,3
0,10	0,4
0,23	0,12
1,0	0,1

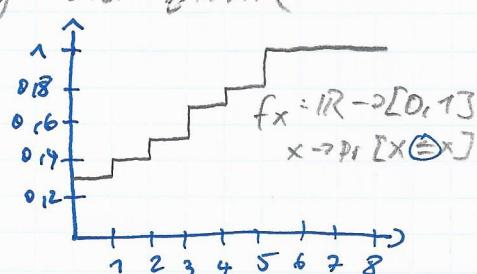
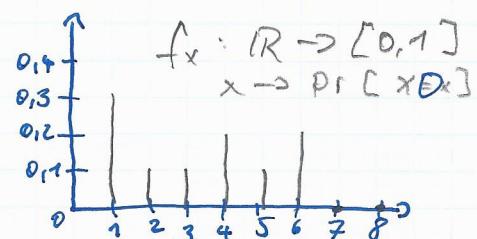
Dichte- / Verteilfunktion von Zufallsvariable

- Dichte sagt aus, bei welchem Wert ich welche Wahrscheinlichkeit habe

- Alle Werte zusammen ergeben 1

- Jeder Balken ist zwischen 0-1

- Die Verteilung ist die Aufsummierung der Dichte



Varianz

→ Streuung des Erwartungswertes

- Wie viel weicht die Zufallsvariable von ihrem Erwartungswert ab?

$$\text{Var}[X] := \sum_{x \in \omega_X} (x - \mathbb{E}[X])^2 \cdot \Pr[X=x]$$

- Standardabweichung $\sigma := \sqrt{\text{Var}[x]}$

- Um Varianz besser ausrechnen zu können

$$\hookrightarrow \text{Var}[x] = \mathbb{E}[x^2] - (\mathbb{E}[x])^2$$

Bernoulli-Verteilung

Experiment mit nur zwei Resultaten

$$- f_x(x) = \begin{cases} p & \text{für } x=1 \\ 1-p & \text{für } x=0 \end{cases} \quad p = \text{Erfolgswahrscheinlichkeit}$$

- Benutzt man bei Indikatorvariablen

$$\hookrightarrow x \sim \text{Bernoulli}(p)$$

Wir werfen Münze 1x: Indikatorvariable für Kopf

Binomialverteilung

p = Erfolgswahrscheinlichkeit

- Wenn wir Münze mehr als ein Mal werfen

$$- f_x(x) = \begin{cases} \binom{n}{x} \cdot p^x \cdot (1-p)^{n-x} & x \in \{0, 1, 2, 3, \dots, n\} \\ 0 & \text{sonst} \end{cases}$$

- Ich werfe Münze n-Mal und will, dass Kopf x Mal vorkommt

$$\hookrightarrow x \sim \text{Bin}(n, p)$$

Geometrische Verteilung

p = Erfolgswahrscheinlichkeit

- Eine Aktion wird so lange ausgeführt, bis man das erste Mal erfolgreich ist

$$- f_x(i) = \begin{cases} p(1-p)^{i-1} & \text{für } i \in \mathbb{N} \\ 0 & \text{sonst} \end{cases}$$

- $\mathbb{E}[x] = \frac{1}{p}$ (\Rightarrow Wenn ich Kopf schon will, sagt mir Erwartungswert, dass ich 2x werfen muss, wenn $p=0,5$)

$$\hookrightarrow x \sim \text{Geom}(p)$$

Gedächtnislosigkeit

• Wahrscheinlichkeit, dass ich beim 1. Wurf Kopf sehe ist gleich gross, wie beim 1000. Wurf

Warten auf k-ten Erfolg

- $\gamma := \# \text{Würfe bis zum } k\text{-ten Mal Kopf}$ ($\underbrace{\text{Kopf}, \text{Kopf}, \dots, \text{Kopf}}_{k\text{-mal hintereinander}}$)

$$1. \text{ Wurf } k: \frac{1}{p} \quad \text{Muss addiert werden, da geometrische Verteilung geschichtungslos ist.}$$

$$2. \text{ Wurf } k: \frac{1}{p} + \frac{1}{p} = \frac{2}{p}$$

$\overbrace{1. \text{ Wurf} \quad 2. \text{ Wurf}}$

$$k. \text{ Wurf } k: \frac{1}{p} + \frac{1}{p} + \dots + \frac{1}{p} = \frac{k}{p}$$

$\overbrace{\quad \quad \quad \quad \quad}_{k \text{ mal}}$

Coupon collector

- n verschiedene Coupons, alle gleich wahrscheinlich

- $X := \text{"Anzahl Käufe, bis ich alle Karten habe"}$

$$- X = x_1 + x_2 + x_3 + \dots + x_n$$

- $x_1 = 1$ (1 Bild, welches ich kaufe)

$$- x_2 \sim \text{geom}\left(\frac{n-1}{n}\right)$$

$$- x_i \sim \text{geom}\left(\frac{n-(i-1)}{n}\right)$$

$$\hookrightarrow \mathbb{E}[X] = \sum_{i=1}^n \mathbb{E}[x_i] = \sum_{i=1}^n \frac{n}{n-i+1} = n \sum_{i=1}^n \frac{1}{i} = n \cdot \overbrace{\log(n)}^{\text{Overhead, Bilder die ich schon habe}}$$

Poisson-Verteilung

→ Modelliert seltene Ereignisse

- Ereignis, denen zwar mit sehr geringer Wahrscheinlichkeit auftritt. (Beispielsweise #Lente Herzinfarkt)

$$- f_x(i) = \begin{cases} \frac{e^{-\lambda} \cdot \lambda^i}{i!} & \text{für } i \in \mathbb{N}_0 \\ 0 & \text{sonst} \end{cases}$$

$$\hookrightarrow X \sim \text{Po}(\lambda)$$

Poisson-Verteilung als Grenzwert der Binomialverteilung

$$- \text{Bin}(n, \frac{\lambda}{n}) \xrightarrow{n \rightarrow \infty} \text{Po}(\lambda)$$

↪ gilt für Binomialverteilung, dass Anzahl Versuche sehr gross ist, so kann man Verteilung durch Poisson approximieren

Mehrere Zufallsvariablen

- $\Pr[X=x \cap Y=y] = \Pr[X=x, Y=y]$
- Gemeinsame Dichte: $f_{X,Y}(x,y) := \Pr[X=x, Y=y]$
- Randdichte: $f_X(x) := \sum_{y \in W_Y} f_{X,Y}(x,y)$
Dichten der einzelnen Zufallsvariablen
- $E[X+Y] = E[X] + E[Y]$ (= Linearität des Erwartungswertes)
- $E[X \cdot Y] = E[X] \cdot E[Y]$
- $\text{Var}[X+Y] = \text{Var}[X] + \text{Var}[Y]$
- $\text{Var}[X \cdot Y] \neq \text{Var}[X] \cdot \text{Var}[Y]$
- Ist X/Y unabhängig - Check: $\Pr[X_1=a, X_2=b] = \Pr[X_1=a] \cdot \Pr[X_2=b]$

Waldsche Identität

- Wenn wir zwei Zufallsvariablen N/X haben, welche unabhängig sind

- N sei eine Zahl und X wird so oft gemacht, wie die Zufallsvariable N

- Somit ist $Z := \sum_{i=1}^N X_i$

$$E[Z] = \underbrace{E[N] \cdot E[X]}$$

Dann darf man N und X als Phasen anschauen und diese multiplizieren.

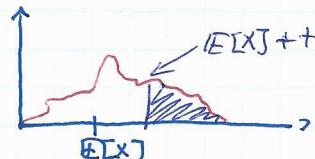
Ungleichung von Markov

- Nur für nicht-negative Zufallsvariablen

- Gibt nur eine obere Schranke an

$$\Pr[X \geq t] \leq \frac{\mathbb{E}[X]}{t}$$

↳ Die Wahrscheinlichkeit, dass die Zufallsvariable $> t$ ist, ist kleiner gleich dem Erwartungswert durch t .



- Wahrscheinlichkeit, dass Zufallsvariable $\geq k \cdot \mathbb{E}[X]$ Erwartungswert ist

$$\Pr[X \geq k \cdot \mathbb{E}[X]] = \frac{1}{k}$$

Chebyshev-Ungleichung

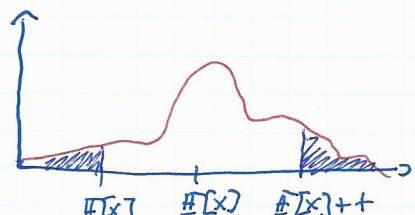
- Gibt uns untere sowie obere Schranke an, da Wahrscheinlichkeit in Beitragsstrichen steht

- Präziser als Markov-Ungleichung

$$\Pr[|X - \mathbb{E}[X]| \geq t] \leq \frac{\text{Var}[X]}{t^2}$$

$$\Pr[|X - \mathbb{E}[X]| \geq t \cdot \mathbb{E}[X]] \leq \frac{\text{Var}[X]^2}{(\mathbb{E}[X])^2}$$

↳ Zufallsvariable weicht um $\pm t$ vom $\mathbb{E}[X]$ um $\% \%$ ab



Ungleichung von Chernoff

- Zufallsvariable muss Summe von unabhängigen, gleichverteilten Bernoulli Variablen sein

$$\Pr[X \geq (1+\alpha) \cdot \mathbb{E}[X]] \leq e^{-\frac{1}{3} \cdot \alpha^2 \cdot \mathbb{E}[X]}$$

(Abweichung)

- Beispiel $\alpha = 0,1$: Zufallsvariable weicht von Erwartungswert um mindestens 10% ab

Monte-Carlo-Algorithmus (= Manchmal (rap))

- Algorithmus terminiert immer
- Gibt manchmal falsches Ergebnis aus \rightarrow Mehrfach laufen lassen
- z.B. randomisierte Primzahltest

Las-Vegas-Algorithmus (= Laufzeit verschissen)

- Algorithmus muss nicht immer terminieren \rightarrow Endlos
- Gibt immer richtiges Ergebnis aus
- z.B. randomisiertes Quicksort

Reduktion Fehlerwahrscheinlichkeit

- Führen wir einen randomisierten Algorithmus mehrfach aus, so reduziert sich die Fehlerwahrscheinlichkeit
- $\Pr[\text{"Algorithmus korrekt"}] \geq \xi$
 ξ Korrektheitswahrscheinlichkeit (gegeben)
- $\Pr[\text{"Algorithmus korrekt"}] \geq 1 - \sigma^m$
 σ Fehlerwahrscheinlichkeit (gewünscht)
- Ich kann den Fehler durch konstant ofte Wiederholungen beliebig klein machen
- $(1 - \xi)^N \leq \sigma$ (N = Anzahl Aufrufe)

Sortieren und Selektieren

- Quicksort (=Las Vegas)

- Wählt Pivot zufällig

- Laufzeit hängt von Glück mit Pivotwahl ab

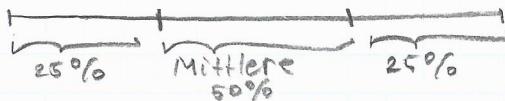
• Durchschnittliche Laufzeit: $n \cdot \log(n)$

- QuickSelect

- Wir wollen k grösstes Element im Array

- Vorgehen: Quicksort, aber immer nur diejenige Partition nehmen, in der sich Arrayindex k befindet

• Veranschaulichung:



- x_i := Indikator, ob Pivot aus mittleren 50% kommt

↳ $x_i \sim \text{Bernoulli}(0.5)$

↳ Mindestens 25% fallen weg,

noch zu betrachtende Elemente: $3/4$

- Ich wähle Pivot so lange, bis es in den mittleren 50% vorkommt: $Y_k = \# \text{Aufrufe in der } k\text{-ten Phase}$

$Y_k \sim \text{geom}(1/2)$

↳ Jede Phase muss ich $E[Y_k] = 2x$ ausführen,

damit ich in mittleren 50% lande

$$\Rightarrow E[\text{"Anzahl rekursive Aufrufe"}] \leq \sum_{k=1}^{\infty} \underbrace{E[Y_k]}_{=2} \cdot \underbrace{\left(\frac{3}{4}\right)^{k-1}}_{\substack{\text{Anzahl} \\ \text{Elemente} \\ \text{pro Aufruf}}} \cdot n \quad \substack{< N \\ \text{Schlüsselvergl.}}$$

↳ wird immer wichtiger

$$\leq \frac{8n}{3}$$

↳ $O(n)$

Primzahltest (= Monte Carlo)

- Wenn n prim ist, dann gilt: $a^{n-1} \bmod(n) = 1$
- Wenn dies nicht gilt, ist Zahl sicher nicht prim
- Laufzeit: $O(\log(n))$
- Falls n prim ist \rightarrow Immer prim
- Falls n nicht prim ist $\rightarrow \frac{1}{3}$ Prim

Target-Shooting (= Monte Carlo)

- Approximation von Mengen

$$S \subseteq U$$

1. Generiere ein zufälliges Element in U , countGlobal++
2. Prüfe, ob Element auch in S liegt, wenn ja count++
3. Berechne $\frac{\text{count}}{\text{countGlobal}}$ = Approximation

Entscheidung, welche Antwort ich nehme

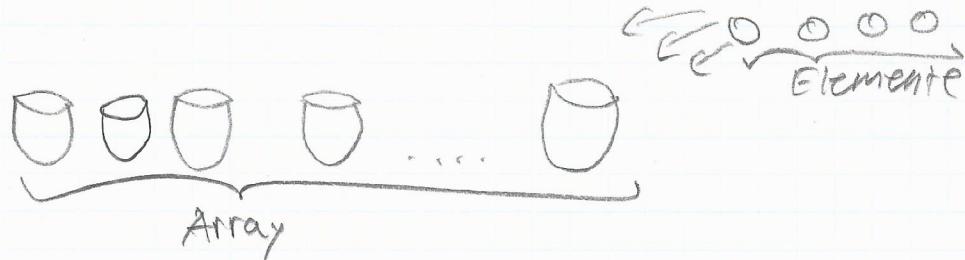
- Das, was mehr vorkommen ist
- $X_i = \begin{cases} 1 & \text{iter Aufruf ist richtig} \\ 0 & \text{sonst} \end{cases}$
- $X_i \sim \text{Bernoulli} (\equiv \frac{1}{2} + \varepsilon)$
- $X = \sum_{i=1}^n X_i$
- $\Pr[\text{"Abg gibt nach } n \text{ Wiederholungen Antwort 'Nein'}]$

$$\Leftrightarrow \Pr\left[X < \frac{n}{2}\right] = \Pr\left[X \geq (1-\delta) \cdot \mathbb{E}[X]\right] \leq e^{-\frac{1}{2} \cdot \delta \cdot \mathbb{E}[X]}$$

Ja kommt
weniger als
die Hälfte
vor

Hashing

- Alle Hashwerte sollten gleich oft vorkommen
- Wenig Kollisionen
- Effizient berechenbar
- f^{-1} schwierig zu berechnen
- Abstrakt: Werfe Bälle in Bins (Bins and Balls)



- Power of two choices: Wähle zwei Körbe und nehme denjenigen mit weniger Bällen
- Universelles Hashing: $\Pr[\text{Kollision}] = \frac{1}{n}$

Bloomfilter (Anstatt alle Daten zu speichern, speichere ich Bloom-Filter)

- Datensatz von Duplikaten bereinigen \hookrightarrow Liste mit bösen Websites
- Verwende k-Hashfunktionen
- Speichere m Bits (initialisiert mit 0)
- Wenn ich Element einfüge:
 1. Durch alle Hashfunktionen
 2. An jeder Stelle, die jede Hashfunktion ergibt, Zahl auf 1 setzen
- Wenn ich Element überprüfe:
 1. Durch alle Hashfunktionen
 2. Wenn an allen Orten = 1, dann wahrscheinlich Duplikat
- false-positive
 - Wenn Algo ja ausgibt, es aber auch false sein kann
 - Je mehr Hashfunktionen, desto größer muss m sein
 - Optimal: 2-3 Hashfunktionen

Lange Pfade

- Wir wollen Pfad im Graph mit Länge $B = \log(n) \downarrow$ finden
- 1. Wir färben die Knoten zufällig mit $k = b+1$ versch. Farben ein
- 2. Dann prüfen wir, ob es einen bunten Pfad der Länge $k-1$

↳ Wenn wir einen Graph mit Pfad der Länge $k-1$ haben, dann ist die Wahrscheinlichkeit, dass unser Algorithmus diesen auch findet $\geq e^{-k}$

- Nun können wir Monte-Carlo-Algorithmus daraus bauen.

↳ Wir lassen Algorithmus λ -Mal laufen

- Gibt er 1x Ja aus, hören wir auf \rightarrow Ja
- Gibt er immer Nein aus \rightarrow Nein

Bunte Pfade

- Um bunte Pfade der Länge $k-1$ zu finden,
- Auf dem Pfad müssen alle k -Färbungen genau ein Mal vorkommen
- Dynamische Programmierung
 - $P_0(v)$ = die Farbe von v selbst
 - $P_1(v)$ = Pfad zu allen Nachbarn, die andere Farbe haben, als v selbst
 - $P_i(v)$ = Ich schaue mir alle Nachbarn von $P_{i-1}(v)$ an, und erweitere mit diesen die möglichen Pfade, falls deren Farben noch nicht vorgekommen sind

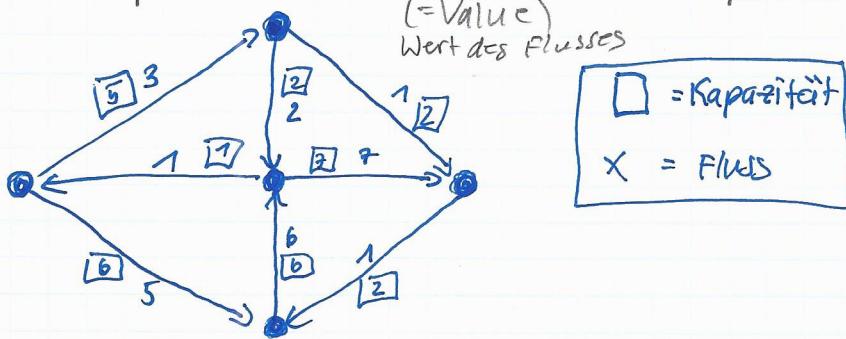
Flüsse in Netzwerken

- $s \in V$ ist Quellknoten (=source)
- $t \in V$ ist Senkknoten (=Target)
- Jede Kante hat eine Kapazität c
- Fluss darf nicht negativ sein
- Fluss darf nur die maximale Kapazität sein
- Flusserhaltung

• Das System ist abgesehen von S/T abgeschlossen

$$\underbrace{\text{netoutflow}(s)}_{\text{Input}} = \underbrace{\text{val}(f)}_{\text{im System}} = \underbrace{\text{netinflow}(t)}_{\text{Output}}$$

(=Value)
Wert des Flusses



$$\underbrace{\text{netoutflow}(s) = 7}_{3 - 1 + 5 = 7} = \underbrace{\text{netinflow}(t) = 7}_{1 + 7 - 1 = 7}$$

$S-T$ -Schnitte

- Für jeden $S-T$ -Schnitt gilt: $\text{val}(f) \leq \text{cap}(S, T)$

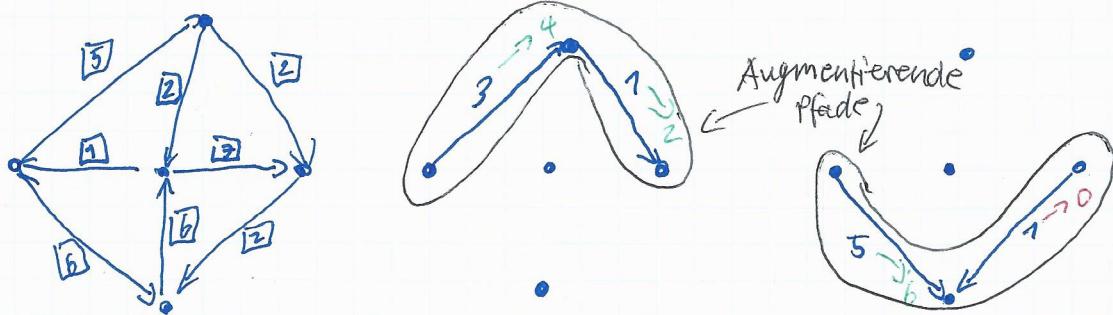
Maxflow-Mincut-Theorem

$$\max \text{val}(f) = \min \text{cap}(S, T)$$

↳ Der grösstmöglicher Fluss von $S \rightarrow T$ ist gleich dem kleinstmöglichen Schnitt, da dieser das Maximum vorgibt

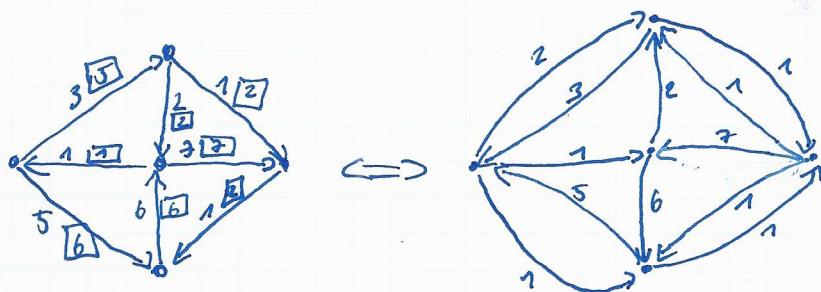
Augmentierende Pfade

- Beliebiger Pfad von $S \rightarrow \dots \rightarrow T$
- Wir nehmen Pfad und schauen, ob der Fluss von diesem Pfad erhöht werden kann
- Die Richtung der Pfeile ist dabei egal



Restnetzwerke

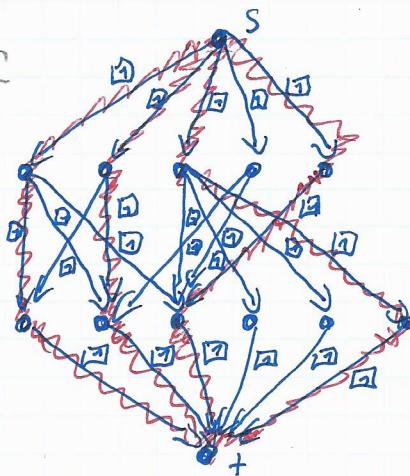
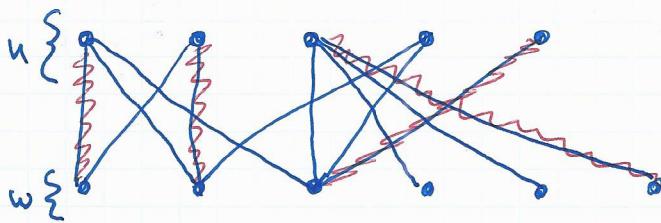
- Netzwerk von den noch freien Kapazitäten
- Wenn es im Restnetzwerk keinen Fluss mehr von $S \rightarrow T$ gibt, dann ist Fluss maximiert



Ford-Fulkerson-Algorithmus

- Suche mit Hilfe des Restnetzwerkes augmentierende Pfade, solange es welche hat, und erhöhe damit den Fluss
 - Laufzeit: $O(m \cdot n \cdot u)$ ($u = \text{maximale Kapazität}$)
1. Setze Fluss zu 0 (überall alle Flüsse 0 setzen)
 2. Solange es $S-T$ -Pfad im Restnetzwerk gibt (BFS/DFS)
 - 2.1. Erhöhe Fluss durch den Pfad
 3. Return maximaler Fluss

Bipartites Matching als Flussproblem

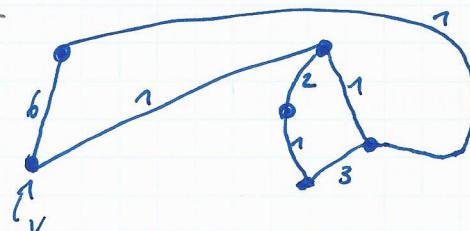
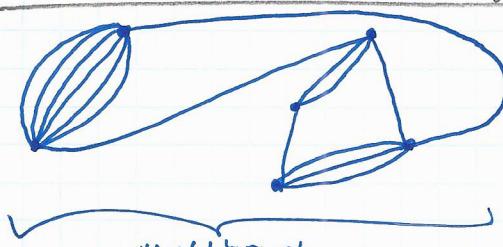


- Größtes Matching $\hat{=}$ maximaler Fluss im Netzwerk

Konvexe Mengen der Flüsse

- Entweder hat ein Netzwerk genau einen Fluss (o-Fluss)
 - Oder das Netzwerk hat unendlich Flüsse
 - ↳ Jede Kante ist ein Vektor
 - ↳ Somit stellt ein Netzwerk eine konvexe Hülle dar
 - ↳ m Kanten \rightarrow Raum \mathbb{R}^m

Minimale Schnitte in Multigraphen



- bis mitten minimalen Schnitt bestimmen

↳ Wie viele Kanten muss ich durchschneiden, damit Graph nicht mehr zusammenhängend ist

1. Variante: Berechnung mittels Ford-Fulkerson-Algorithmus

1. Wir wandeln Multigraph zu Netzwerk um
 2. setzen Quelle beliebig
 3. Mache daraus gerichtetes Netzwerk 
 4. Für jeden Knoten setzen wir als Senke und berechnen minCut
 5. Globaler minCut ist Ergebnis
 $\hookrightarrow O(n^4 \cdot \log(n))$

- 2. Variante: Zufällige Kantenkontraktion



- Zwei Knoten werden vereinigt
- $\text{cut}()$: Nimm zufällig zwei Kanten, kontrahiere sie, und mache das so lange, bis es nur noch eine Kante gibt. Gebe dann ihren Wert zurück
- \hookrightarrow Läuft in $O(n^2)$ ← Eine Kantenkontraktion kann in $O(n)$ durchgeführt werden
- gleichverteilte Kante kann in $O(n)$ gezogen werden

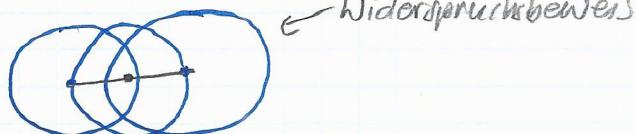
- $\Pr[\text{"Minimaler Schnitt bleibt gleich bei Kontraktion"}] = 1 - \frac{2}{n}$
- $\Pr[\text{"}\text{cut}(\text{)}\text{ gibt richtigen minimalen Schnitt aus"}] = \frac{1}{\binom{n}{2}}$
- (\hookrightarrow) Um richtiges Ergebnis zu erhalten:
 1. Wir lassen Algo $\lambda \cdot \binom{n}{2}$ mal laufen
 2. Kleinster errechneter Wert geben wir aus
 $\hookrightarrow O(n^4 \cdot \log(n))$

- 3. Variante Bootstrapping

- Wir machen zufälligen Algo bis t -Knoten
- Ab t -Knoten nehmen wir Ford-Fulkerson-Algorithmus
 $\hookrightarrow O(n^3 \cdot \text{poly})$

Kleinstter umschliessender Kreis

- Menge P von n Punkten in der Ebene
- Wir wollen kleinsten Kreis $C(P)$ bestimmen
- Punkte dürfen am Rand des Kreises liegen
- Der kleinste umschliessende Kreis ist eindeutig



- 3 Punkte definieren den kleinsten Kreis
- $$\hookrightarrow Q \subseteq P / |Q|=3 / C(Q) = C(P)$$

1. Repeat bis gefunden

1.1. wähle $Q \subseteq P$ mit $|Q|=12$ zufällig

1.2. Bestimme $C(Q)$

1.3. Wenn $P \subseteq C(Q) \rightarrow \text{return}$

1.4. Verdopple alle Punkte außerhalb von $C(Q)$

Array
 [1 1 1 1 ... 1]

↓
 $\frac{1}{2}$

$\Rightarrow O(n \cdot \log(n))$

LaufzeitenPriority Queue

- Prim: $O(|E| + |V| \cdot \log(|V|))$ / Insert: $O(n)$ / Extract-Min: $O(n)$ / Decrease-Key: $O(n)$
- Kruskal: $O(|E| \cdot \log(|V|))$ / Insert: $O(n)$ / Find: $O(m)$ / Union: $O(n)$
- EulerTour: $O(|E|)$ sortieren Union find
- Hamiltonkreischeck: $O(2^n)$
- CouponCollector: $O(n \cdot \log(n))$
- Quicksort: $O(n \cdot \log(n))$
- QuickSelect: $O(n)$
- Lange Pfade: $O(\text{poly})$ wenn Länge $\log(n)$ ist
- Ford-Fulkerson: $O(m \cdot n \cdot u)$ u = höchste Kapazität
- Minimale Schnitte in Multigraphen
 - Ford-Fulkerson: $O(n^9 \cdot \log(n))$
 - Kontraktion: $O(n^2)$ ohne Mehrfachausführung
 - Kontraktion: $O(n^9 \cdot \log(n))$ mit Mehrfachausführung
 - Bootstrapping: $O(n^3 \cdot \text{poly})$
- Kleinster Kreis mit Punktverdopplung: $O(n \cdot \log(n))$