Effiziente Algorithmen

Übersicht

E = Knoten
k = Anzahl N = Daten V = Kanten

Wiederholung

(eig. Klausur)

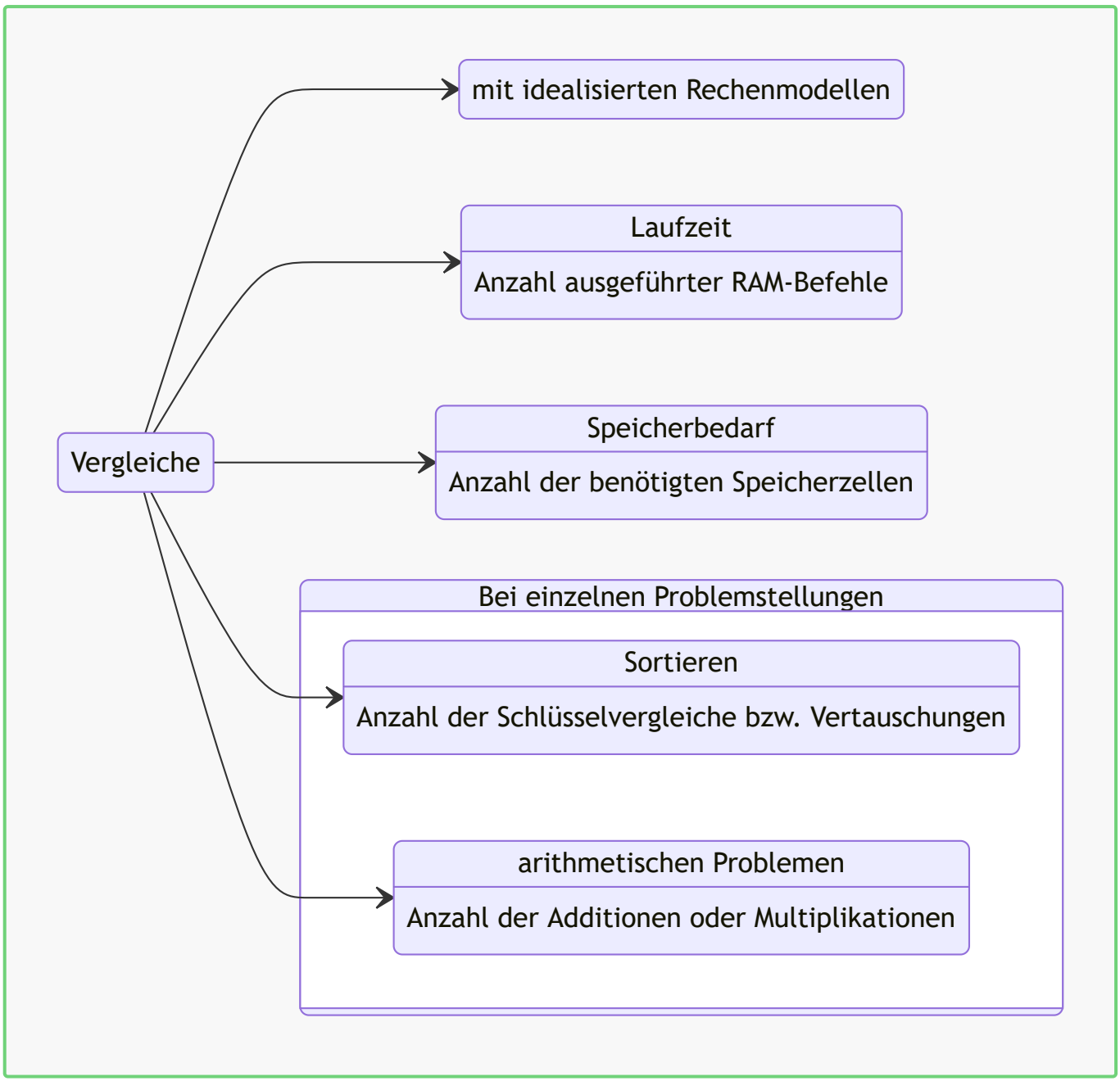
Allgemein Algorithmus

Eigenschaften

- Korrektheit: genau zeigen was das problem ist um diese zu zeigen
- Laufzeit
- Speicherplatz
- Kommunikationszeit
- Güte: wie gut ist die Lösung gegenüber der besten Lösung

Vergleich von Algorithmen

Wir verwenden idealisierte Rechenmodelle wie die Registermaschine (Random Access Machine, RAM): festgelegter Befehlssatz (Assembler-ähnlich), abzählbar unendlich viele Speicherzellen. Die Laufzeit ist dann die Anzahl ausgeführter RAM-Befehle, der Speicherbedarf ist die ¨ Anzahl der benötigten Speicherzellen. Bei den einzelnen Problemstellungen ermitteln wir charakteristische Parameter. Beim Sortieren sind dies die Anzahl der Schlüsselvergleiche bzw. Vertauschungen, bei arithmetischen Problemen bspw. die Anzahl der Additionen oder Multiplikationen.



Komplexität

Ist wichtig da bei verteilten bzw. parallelen Rechner keine signifikante Laufzeitverbesserung bringen. Bei exponentieller Laufzeit wie 2n kann ein doppelt so schneller Rechner nur eine um eins größere Eingabe in der gleichen Zeit bearbeiten, da $2 \cdot 2^n = 2n + 1$ ist.

Laufzeiten - O-notation

O(n²)

$n = 2^1 0 => 3s$
$m = 2^1 1 => 12s$
weil: $m^2 = (2 * n)^2$
=> $n^3 = Faktor 8$

> Faktor 4 durch Sekunden Anzahl, und * 2 weil m = 2 * n (?)

log(a*b) = log(a) + log(b)

O zählt Operationen als obere Schranke für worst-, average- oder best-case Abschätzungen angegeben werden. Groß-O heißt nicht automatisch worst-case. (unterschiedlich welche z.B. Arithmetische) O(g) = {f | ∃c ∈ N ∃n0 ∈ N ∀n ≥ n0 : 0 ≤ f(n) ≤ c · g(n)} Dabei betrachten wir nur Funktionen mit natürlichen Funktionswerten, also $f, g : N \to N$, weil wir die Anzahl von Schritten oder benutzten Speicherzellen angeben

worst Case

Die Menge der zulässigen Eingaben der Länge n bezeichnen wir mit W_n, die Anzahl der Schritte von Algorithmus A für Eingabe w mit A(w). Dann ist die Worst-Case-Komplexität definiert als $T_A(n) = supA(w)|w \in W_n$ und ist eine obere Schranke für die maximale Anzahl der Schritte, die Algorithmus A benötigt, um Eingaben der Größe n zu bearbeiten

$$O(g) = \{f|\exists c \in \mathbb{N}\, \exists n_0 \in \mathbb{N}\, \forall n \geq n_0 : 0 \leq f(n) \geq c * g(n)\}$$

∃ = existiert
∀ = für alle
: = gilt
n_0 => Schranken Funktion ab n_0 gültig
c => Konstante um Schranken die niedriger sind anzuzeigen (darum auch:)

immer das höchste n = (die kleinste) oberste Schranke

Ω = untere Schranke: best case Größe wird gesucht f unf g am ende tauschen

Θ = exakte Schranke

log^k(n): bei k = 2: log(n)*log(n) log basis ist egal weil log_b(n) = log_a(n)/log_a(b)

asymptotische Aufwandsabschätzung: **Grenzwert Betrachtung**
Funktion auf knoten UND kanten
n0 und Funktionen basieren auf 2
Durchschnittlichte O Betrachtung

Berechenbarkeit:
Abzählbare Sprache: der Länge nach alle Wörter **Sprache** -> **eine endliche, nichtleere Menge von Zeichen**. Zeichen werden auch Symbole oder Buchstaben genannt. Wörter Endliche Folgen $(x_1, \ldots, x_k) mit x_i \in A$ der Länge k über dem Alphabet A

Beweis nicht berechenbar

Reduktion. Eine Sprache L heißt reduzierbar auf L′, falls es eine totale berechenbare Funktion $f : \Sigma * - \to \Sigma *$ gibt mit x ∈ L ⟺ f(x) ∈ L′.
wir schreiben dann: L ≤ L′(mittels f)
Idee: L ist nicht schwerer zu lösen als L′
Falls L nicht entscheidbar ist und L ≤ L′ (mittels f) gilt, dann ist auch L′ nicht entscheidbar. Denn wäre L′ entscheidbar, dann wäre auch L entscheidbar:

Berechne für eine beliebige Eingabe x für L den Wert f(x) und entscheide, ob f(x) ∈ L′ gilt.
Damit wäre auch entschieden, ob $x \in L$ ist.

RAM: Random Access Machine/Registermaschinen, wie pc mit Miniassembler

uniform: **wie oft addiert**
logarithmisch: **uniform + wie lang waren die zahlen**

Ca: der Berechenbarkeit-Begriff:
Ein Programm p möglich, was zur jeder Eingabe ne Ausgabe macht = Berechenbarkeit

> Ein Funktion ist berechenbar wenn man diese als Programm schreiben kann

Bsp. NICHT BERECHENBAR

Diagonalsprache

DIAG = {⟨M⟩ | M ist dTM, die ⟨M⟩ nicht akzeptiert} EingabeXFunktion zeigt Ausgabe: Funktion welche die diagonalen Ausgaben hat negiert, ist nicht in der Liste

Halteproblem

Stoppt ein Programm zu einer bestimmten Eingabe H = {⟨M⟩x | M ist dTM, die gestartet mit Eingabe x hält} Schwerer oder gleich nicht DIAG

Halteproblem bei leerem Band

H_0 = {⟨M⟩ | M ist dTM, die gestartet mit Input ϵ hält} Schwerer oder gleich H

Totalitätsproblem

{⟨M⟩ | M hält fur jeden Input }

Endlichkeitsproblem

{⟨M⟩ | M hält fur endlich viele Inputs}

Äquivalenzproblem

{⟨M⟩, ⟨M'⟩ | M und M′ akzeptieren die gleiche Sprache}

Turingmaschine

Eine dTM besteht aus einem unendlich langen Band, einem Schreib-/Lesekopf und einer Zustandssteuerung:

- Das potentiell unendliche Band ist in Felder unterteilt, wobei jedes Feld ein einzelnes Zeichen des Arbeitsalphabets der Maschine enthalten kann.
- Auf dem Band kann sich der Schreib-Lesekopf bewegen, wobei nur das Zeichen, auf dem sich dieser Kopf gerade befindet, im momentanen Rechenschritt verändert werden kann.
- Der Kopf kann in einem Rechenschritt nur um maximal eine Position nach links oder rechts bewegt werden.

- Noch nicht besuchte Felder enthalten das Blank-Symbol B.

Momentaufnahme $\alpha z \beta$
α = **Zeichen vor Zeiger** z = **zustand** β = **Zeichen nach Zeiger (Zeiger auf erstes Zeichen)**

Unterprogramme:
(by me) Zustandsgruppen + Sprung in ein andere Unterprogramm Oder: **Zusatz band => nur dort ausführen**

für jedes Zeichen kann ein Zustand erstellt werden zum speichern => **Weil Eingabe endlich, kann man mit endlichen zuständen merken**

Spuren möglich, weil die **Spuren zam gefasst dann das Zeichen** sind

Mehrere (k-)Bänder möglich, wobei **jeder kopf unabhängig** ist => **1. band Arbeitsband** => **alle anderen blank** wird **durch 2k Spuren simuliert mit pointer**

Aber jede t(n)-zeit- und s(n)-platzbeschränkte k-Band TM kann durch eine O(t(n)· s(n))-zeit- und O(s(n))-platzbeschränkte TM simuliert werden.

(endlich viele) variablen = neues band

Abfragen durch unterprogramm => je nach Lösung ins nächste springen

dTM und nTM ändern nichts an der berechenbarkeit

UNTERSCHIED dTM und nTM nTMs besitzen anstelle der Übergangsfunktion eine Übergangsrelation, d.h. jede Konfiguration hat mehrere mögliche Nachfolgekonfigurationen. Der Akzeptanzbegriff ist anders definiert: **Eine nTM akzeptiert eine Eingabe genau dann wenn eine Berechnung von der Startkonfiguration in eine akzeptierende Endkonfiguration existiert.**

nTM lässt sich mit exponentieller zeit mit dTM simulieren (alle pfade ausprobieren)

C Programme abzählbar
Abbildungsmenge überabzählbar
=> **nicht alle funktionen durch c programm realisierbar**

input x funktion => diagonale nehmen und negieren => neue funktion die nirgends steht

Gödelnummer: **wird iwie für DIAG verwendet**

> eine erste, nicht berechenbare Funktion (DIAG) **Programm von TM erstellt in andere TM laden und ausführen**
> **dafür unär die Tabelle codieren** (1 trennt eintrage, zwei eines neue zeile, drei Einsen zum von EIngabe trennen)
> => Werte und Programme gleich behandeln

Zeichne und L,R,N in zustand merken weil beschränkt
Zustand nicht, weil nicht beschränkt, in der zu lesenden TM maschine (aber endlich)
DIAG => ok wenn nicht ok, nicht ok wenn ok => wie mit den funktionen, die diagonale gedreht

> Entscheidbar = berechenbar

Eine Funktion f : N k → N ist berechenbar, falls es ein RAM-Programm P gibt (Achtung: die Existenz reicht aus!), sodass gilt: P berechnet m zur Eingabe n1, . . . , nk genau dann wenn f(n1, . . . , nk) = m gilt

Entscheidbare Sprachen

Eine entscheidbaren (rekursiven) Sprachen ist abgelschossen gegen Komplementbildung, da aus akzeptierenden Endzuständen von M werden nicht-akzeptierende Endzustände von M' und **man führt einen neuen akzeptierenden Endzustand ein, der immer dann erreicht wird, wenn die ürsprungliche Maschine M in einem nicht-akzeptierenden Endzustand hält**. **abgelschossen gegen Komplementbildung**: **L und Not(L) sind rekusiv aufzählbar. rekusiv aufzählbar**: **wenn eine TM L akzeptiert**

rekusive Sprache muss beim nicht akzepieren anhalten, eine rekusiv aufzählbare nicht.

!DIAG ist nicht entscheidbar, weil entscheidbare sprachen abgeschlossen gegenüber Komplementbildung sind

$x \in L \iff f(x) \in L'$
Sprache $L \leq L'$ bedeutet L nicht schwerer zu lösen als L'
$ALT \leq NEU$ mittels f
wenn neu lösbar, dann alt auch

Für das Halteproblem H Funktion f(x) bilden:
$x \notin(DIAG) \iff f(x) \in H$ setzen
zudem ist $H \leq H_0$

Rekusiv aufzählbar ist NICHT gegen Komplementbildung abgeschlossen, sonst könnten beide verwendet werden zum entscheidungstreffen

Sprachen gleichhalt und Halten für Inputs kann man nicht entscheiden mit einer TM

Rekusion

optimaltitätsprinzip muss erfüllt sein: aus optimalen teillösungen bildet sich die optimale Gesamtlösung
rek(prams) = {list von reks(new params) [+ endbedinguns addition]}

zeigen in 3 Möglichkeiten:

- immer wieder einsetzen
- supstition
- raten und zeigen, dass es richtig ist

Komplexitätstheorie

P (**polynomielle zeitbetränkung**) vs. NP (**NICHT DETERMINISTISCH**) => **darum nicht klar ob gleich oder nd**
L (**logaritmisch platzbeschränkt, nur schreib beschränkt**) vs. NL
PSPACE vs. NPSPACE => gleich!

> wir betrachten immer die Entscheidungsvariante (es gibt eine Lösung) und nicht die Optimierungsversion (die optimaler Lösung) der Probleme, wenn es um NP-Vollständigkeit geht, da es bei der Optimierungsvariante reicht es nicht, die "geratene" Lösung zu verifizieren. Es müsste sonst auch sichergestellt werden, dass es keine bessere Lösung gibt. Sonst auch NP-Vollständig und nicht effizient.

polynomiell reduzierbar

Eine Sprache L ist polynomiell reduzierbar auf eine Sprache L', wenn eine Funktion f existiert, mit x ∈ L ⟺ f(x) ∈ L' und f ist in polynomieller Zeit berechenbar

> Eine Sprache L ist polynomiell reduzierbar, wenn es eine, in polynomieller Zeit berechenbar Funktion gibt, welche die Sprache L auf eine Sprache L' reduzieren kann

Graphen

- **Clique**: **alle knoten durch kanten zu allen anderen knoten der clique verbunden**
  - **Entscheidung**: **Einschränkung k gibt min Knotenzahl an** (**für festes k P**, wenn Eingabe (like n/2) möglich ist, dann np)
  - **Optimierung**: Min suchen ist zu schwer => suche immer in den Hälften
  - **suche**: id suche dann auch lösbar (faktor n)
- **Independent Set: Gruppe welche durch keine kante verbunden ist** (wieder k möglich)

Suchen sollen gelöst werde, aber auch Entscheidungs beschränkbar, dies am besten polynomiell

Entscheidungsvariante: gibt es eine min/max k **Optimierungsvariante**: gib min/max größe **Suchvariante**: gib min/max knoten

NP vollständig: lässt sich auf L reduzieren, wobei $L \in NP$

SATisfiability

kann eine Formel F erfüllt werden

1. **erster Durchlauf variablen suchen**
2. **nicht deterministisch ausprobieren**

a -> b: wenn a dann b $a -> b \equiv \not{a} \wedge b$

Wir müssen eine Boolesche Formel F angeben, so dass gilt: x ∈ L ⟺ F ist erfüllbar
Die gesuchte Formel F enthält folgende boolesche Variablen:

- $zust_{t,z}$, t = 0, . . . , p(n) und z ∈ Z, $zust_{t,z}$ = 1 ⟺ nach t Schritten befindet sich M im Zustand z
- $pos_{t,i}$, t = 0, . . . , p(n) und i = −p(n), . . . , p(n), $pos_{t,i}$ = 1 ⟺ der Schreib-Lesekopf von M befindet sich nach t Schritten auf Position i
- $band_{t,i,a}$, t = 0, . . . , p(n) und i = −p(n), . . . , p(n) und a ∈ Γ, $band_{t,i,a}$ = 1 ⟺ nach t Schritten befindet sich auf Bandposition i das Zeichen a

Die Formel F ist aus mehreren Teilformeln aufgebaut, die unterschiedliche Dinge repräsentieren:

- **Randbedingungen**: **Die TM M ist zu jedem Zeitpunkt t in genau einem Zustand**; der Kopf von M befindet sich zu jedem Zeitpunkt t an genau einer Bandposition; zu jedem Zeitpunkt t und an jeder Bandposition i steht genau ein Zeichen.
- **Anfangsbedingung**: **Die TM M ist im Anfangszustand z0**; der Schreib-/Lesekopf befindet sich auf dem ersten Zeichen; die ersten n Bandpositionen enthalten die Eingabe x1, . . . , xn; an allen anderen Bandpositionen steht ein Blank.
- Beim Übergang vom Zeitpunkt t nach t + 1 darf sich an der Stelle, wo sich der SchreibLesekopf befindet, der Bandinhalt ändern; an allen anderen Stellen darf sich der Bandinhalt nicht ändern.
- **Endbedingungen**: **Ist die Maschine M in einem akzeptierenden Zustand**?

> 3KNF-SAT ist NP-vollständig durch die vorrangeheende Umformung

BSP Aufgabe zur Laufzeit 3Sat

Betrachten Sie den folgenden Algorithmus. Sei Φ eine Formel in konjunktiver Normalform, bei der jede Klausel höchstens 3 Literale enthält.

```
3SAT(Φ):
  if 3SAT(Φ|x)
    return true
  return 3SAT(Φ|¬x)
```

Dabei bedeutet Φ|x, dass die Variable x der Formel Φ mit 1 belegt ist. Schätzen Sie die Laufzeit ab (mit Herleitung) und begründen Sie, warum der Algorithmus korrekt ist:

Variable x muss entweder mit 0 oder 1 belegt werden. Da alle 2n mögliche Belegungen getestet werden, ist der Algorithmus korrekt. T(n) = 2 · T(n − 1) + poly(n) ∈ O∗ (2^n) $b^n \approx 2 \cdot b^{n-1} + n^k \ | : b^{n-1} \iff$ b ≈ $2 + \frac{n^k}{b^{n-1}} \mid lim_{n\to\infty} \frac{n^k}{b^{n-1}} = 0 \iff$ b ≈ 2

wenn spezial fall schwer, dann das hauptprob auch, da es den spezialfall ja enthält

> KNF-SAT ist NP-vollständig

> 2KNF-SAT liegt in P, kreise prüfen: wenn $a -> \not{a}$, dann nicht erfüllbar

HORN-SAT: alle die 1 werden müssen auf 1 setzen, rest auf null wenn 1 -> 0 ist, dann falsch sonst richtig

DNF-SAT ist in P, nur klammern suchen in dennen $a und \not{a}$, dann geht nicht, sonst schon

> achtung wegen umformkosten!

Clique

3KNF-SAT $\leq_p$ CLIQUE

jedes Literale ist ein Knoten => pro klausel nicht verbinden, sonst wenn nicht negiert

wenn logik erfolgreich ist, verfolgen = clique#

wenn clique, dass in logik eintragen

HCP

Jeden Knoten einmal besuchen als Kreis

> Bei Euler jede Kante + Kreis und beim Travelling Salesmen Problem soll eine optimale Route durch alle Knoten gesucht

Gerichteter Hamilton-Kreis

variablen alle durch laufen, durch die klauseln immer da rein wo es in der kausel steht und der ausgang gleich negatives und positives vorkommen durch eigene lane

Ungerichteter Hamilton-Kreis

wie dis erste nur jede var hat 3 knoten 1. alle und zu 2. zu 3. zu allen outs

wenn 2 drin vorkommen soll muss es von 1 nach 3 oder 3 nach 1

Exakte Algorithmen

O* bzw O => **nur noch das exponentielle anschauen**

3-Sat

mehrfach Setzung, am start

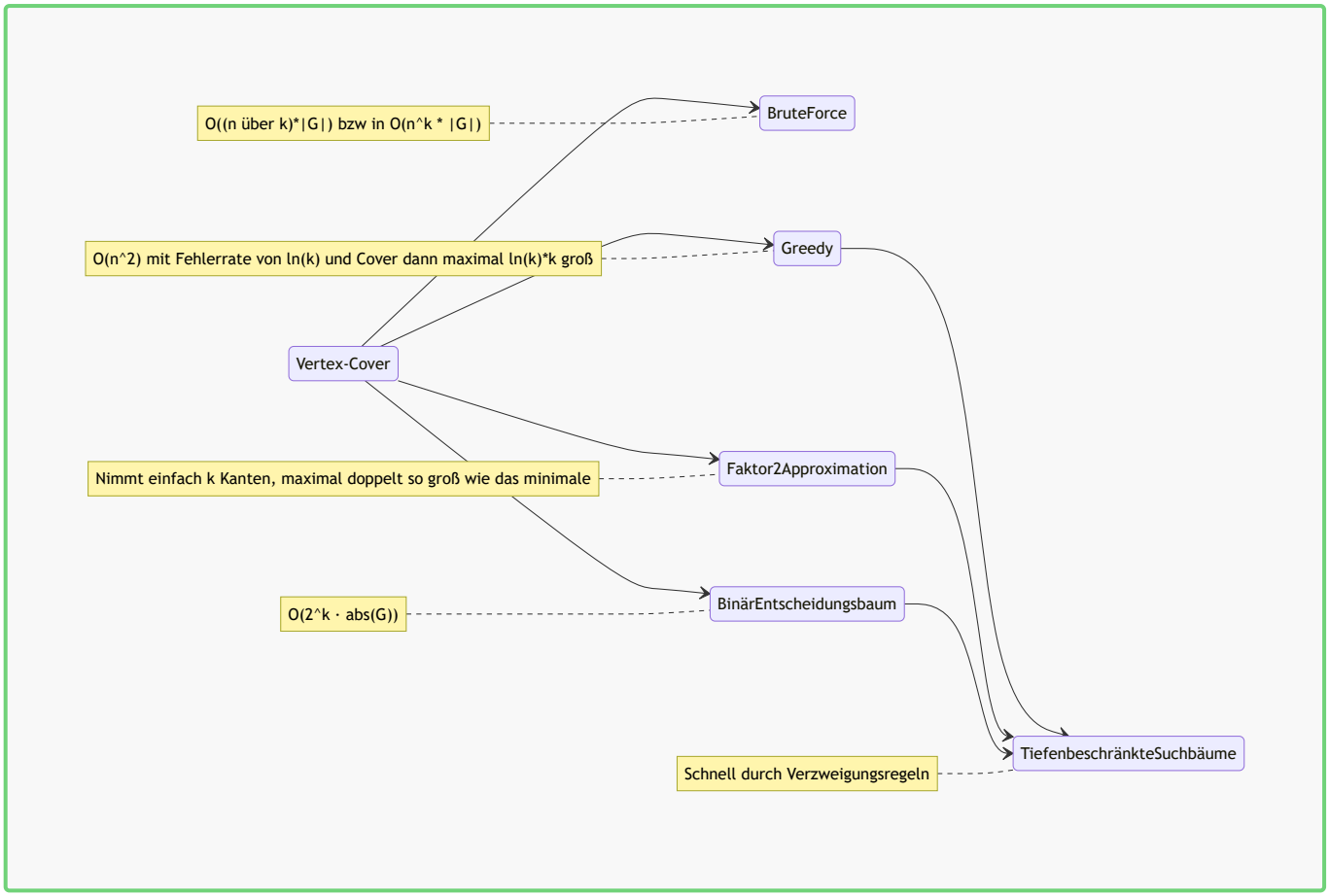
vertex-Cover

Finde zu einem Graphen G = (V, E) und einer Zahl k ∈ N eine Knotenmenge
V′ ⊆ V mit |V′| ≤ k,
sodass jede Kante e ∈ E inzident zu einem Knoten in V′ ist.

> inzident ist eine Kannte zu einem Knoten, an dem sie hängt.

Also wird eine Knotenmenge <= k gesucht, welche mit allen Kanten verbunden ist.

Brute-Force-Ansatz Suche ein Vertex Cover, indem alle (n über k) viele Teilmengen der Größe k betrachtet werden.
Laufzeit O((n über k)|G|) ∈ *O($n^k*|G|$)*



greedy

- C := ∅
- while es gibt noch Kanten in G do
  - sei v ein Knoten mit größter Anzahl Nachbarn
  - nimm v in C auf
  - entferne v und alle inzidenten Kanten aus G

> Einfach nach möglichst großer Grad (Nachbar) Zahl wählen

$O(n^2)$ => aber nur kann Fehler der Größe ln(k) haben Cover kann dann Maximal ln(k)*k groß sein

Faktor-2-Approximation

- C := ∅
- while es gibt noch Kanten in G do
  - nimm irgendeine Kante {u, v} von G
  - nimm u und v beide in C auf
  - entferne u und v und alle dazu inzidenten Kanten aus G

Sei F die Menge der ausgewählten Kanten, sei C das berechnete Vertex-Cover und sei {u, v} ∈ F. **Jedes Vertex-Cover C' muss u oder v enthalten, da sonst die Kante {u, v} nicht abgedeckt würde.**

> Also gilt: $|C'| \geq 1/2|C| \iff |C| \leq 2|C'|$

binärer Entscheidungsbaum für Vertex-Cover-Problem

von einer kannte rekursiv beide - möglichkeiten versuchen binärer Entscheidungsbaum mit beschränkter Tiefe
k: function VC(G, k)
if k = 0 und es gibt noch Kanten in G then return false
if es gibt keine Kanten in G then return true
nimm irgendeine Kante {u, v} von G
x := VC(G − {u}, k − 1)
y := VC(G − {v}, k − 1)
return x ∨ y
$Laufzeit : O(2^k \cdot abs(G))$

Tiefenbeschränkte Suchbäume

Verktor (a,b,..,n) beschreibt um wie viel sich das Problem verkleinert je nach dem welcher Zweig betrachtet wird. Bei Entscheidungsbäumen wird dann auch nur in einen Zweig gewechselt, je nach Bedingung.

FPT (Fixed Parameter Tractable): Eine Algo-Klasse mit einer Laufzeit von $O(f(k(I)) \cdot |I|^c)$, welche ein parametrisiertes Entscheidungsproblem beschreibt. wobei c ∈ N eine Konstante, I eine Eingabe und k der Parameter ist. Trennt die exponentielle Größe von der Eingabegröße.

Verzweigungsregeln

- **wenn grad > k => muss es drin sein, sonst alle Nachbarn (mehr als k) benötigt**
  - dannach nur noch k^2 Kanten abdeckbar
- löschen dauert n*k, weil jeder Eintrag nun max k verweise hat
  - => alles rausnehmen was einfach ist => rest ist: Problemkern
- **Knoten mit grad 1 nicht nehmen => immer den nachbarn**
- **wenn Knoten nicht enthalten, dann safe alle nachbarn**
- wenn dann v auf a und b und a auf b ist, wähle a und b (weil regeln davor beachtet wurden)
- wenn a und b nicht verbunden aber a und b zeigen auf w, dann v und w
- wenn a und b nichts gemeinsam => a und b oder nicht a und nicht b aber v . . .

Regeln wurden nur bist Grad 4 definiert

> Ernorm viele Verzweigungen scheinen wirklich performant zu sein

independent set

rekusiv:

- v ohne alle nachbarn
- sonst ohne v lösen
- dann max returnen

wenn nachbarschaft leer => v rein und nicht rekusiv

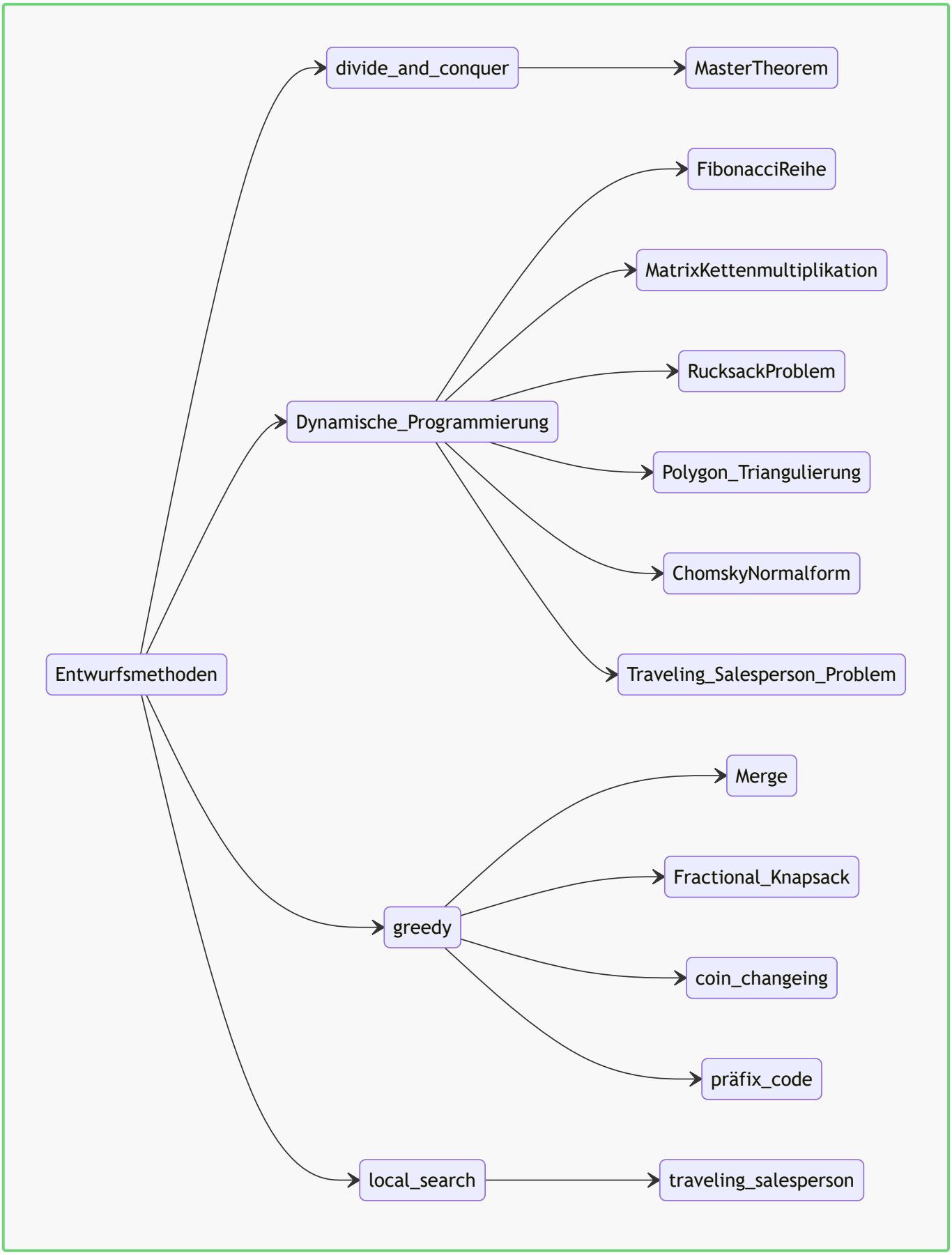
wenn v nur einen nachbar hat => v rein (weil egal ob v oder nachbar)

regulärer graph

k regulärer graph kopieren und verbinden = k+1 regulären graph

Entwurfsmethoden

Methoden zum Entwerfen von Algos.



divide and conquer

- **teilen, aufteilen (rekusive lösen), zamfassen**
- $x^n = x^{n/2} + n^{n/2}$

- beide $n^{n/2}$ sind gleich, also nur einmal berechnen
- eine **gesamte Formel durch immer wieder einsetzen** möglich,
    - **dann schauen und verallgemeinern**
- Ende muss dann noch definiert werden
- oder Substition aka. zamfassen in variablen

> Matrix ist auch gut aufteilbar

Master-Theorem

Ein Master-Theorem ist ein Hauptsatz einer Laufzeitfunktion.

Im Falle von divide and conquer:

Vergleich Aufteilung hoch Länge und Faktoranzahl leitet die Laufzeit her
divide and conquer Aufteilungen können hier eingesetzt werden

> Je nach Fall haben wir verscheidene Laufzeiten als gekürzte Formel

Dynamische Programmierung

- **brauchen öfter das selbe berechnet**
- **Abspeichern als tabelle und immer wieder einsetzen**
- von oben nach unten muss geprüft werden ob die Werte schon da sind
- **Macht es auf kosten des Speichers liniar**
- Bei fibonacci von klein nach groß -statt groß nach klein, dann **Dynamische Programmierung**

> Rekusive Probleme werden mit Zwischenspeicher itterativ gelöst

Ablauf:

1. **rekusiver ansatz**
2. durch überlegen rekusiv **ende definiern**
3. **doppelte rechnung durch speicherung vermeiden**

Matrix-Kettenmultiplikation

durch bessere klammerung können SEHR viele multiplikationen gespart werden

> Herleitung/Plan-Erstellugn in welcher Reihenfolge Multipliziert werden soll.

Rucksack-Problem

Rekusiv: eins nehmen oder nicht => O(2^n)

Dynamisch: i: 1-n, h: 0-G, G max Kapazität; bei h 0 bis kleinstes g wert 0, dann immer eintragen für i n, dann i-- immer zusammen fassen und berechnen
(lange gewichte sind in der TM binär kodiert, welche für eine längere laufzeit sorgen "psydoexpolaufzeit")
nur wen G klein ist, ist die tabelle bis G schnell (oder teilen und nachkommastellen weg lassen, dann aber nur eine Annäherung)

Polygon Triangulierung

Polygon in dreiecke mit möglichst kleinen umfang aufteilen, wo bei sich die neuen kanten nicht kreuzen dürfen **Kosten: summe über umfang aller 3ecke**

meine ideen

- anfang von jedem punkt einmal
- rekusiv teilen zu allen nicht nachbar punkt
- wenn fertig (ende bei 3 knoten), dann umfang returnen
- beide returns der teilgraphen vergleich

lösung

Dreieck raus nehmen, bei i bis j: zwischen i und j muss eine linie sein, k wird ausprobiert: (i,k) und (k,j) wird aufgerufen => dann enden i,k,j umfang dazu berechnen

dynamisch wie die Matrix-Kettenmultiplikation

Chomsky-Normalform

wird ein Wort von einer Gramma in der Normalform akzeptiert

1. speicher array mit Position, Länge, Variable und setzte alles false
2. init jedes Zeichen mit einer variable
3. nachbarn zusammen fassen durch wortlänge (z.B. wortlänge 2+1 = 3)

> "rückwärts" kombinieren und schauen ob man auf S folgen kann

Traveling Salesperson Problem

Kreissuche optimieren in dem 1 gelöscht wird und dann der optimale weg gesucht wird, am ende muss dann 1 noch besucht werden

von Startknoten + Menge den Wert speichern

> Wenn x zu z über y der kürzeste weg ist, ist x zu y auch der kürzeste weg gewählt (optimalitätsprinzip)
> Dadruch dann dynamische programmierung möglich.

beste O ist $O(n^2 * 2^n)$ für eine exate Lösung

Greedy

- **meiste gute laufzeit**
- Müssen **immer neu bewiesen** werden
- **einfache algos, komplexere beweise**
- greedy **funkt nicht immer** => siehe coin changeing
    - immer hinterfragen!!

Merge

- **am besten immer die kleinsten als erstes mergen**
- **Beweis durch induktion:** greedy-Baum immer am besten durch gewichtete externe pathlänge

$$\sum_{i=1}^{n} d_i * q_i$$

- vorstellen es gibt ein anderne optimalen
  - knoten mit max tiefe: nach folger nehemen und tauschen mit den ersten vom greedy (wenn schon gleich dann schon fertig)
- max vs greedy vergleichen => können nicht schlechter sein mit greedy
  - dann eine datei weniger => dadruch n+1

Fractional Knapsack

- Rucksack **mit teil objekten** möglich (dadruch teil profit)
- **besten profit pro gewichtseinheit suchen**

sortieren nach wert/gewicht sollte es einen anderen optimalen geben, dann würde im ersten bereich einer fehlen und im hinteren teil was hinzu kommen
Gewichtseinheit*Wert immer noch kleiner gleich des alten, weil: sortiert

coin changeing

- **sowenig münzen wie möglich für einen betrag**
  - **die größte so oft es geht, dann die nächste größe so oft es geht**
  - (geht **NUR im US geldsytem**, weil zwischen größen übersprungen werden könnten durch einen zwischen wert)

!! gegenbeispiel

nur wenns alle münzwerte gibt

=> es bleibt nur dynamische programmierung, da das optimalitätsprinziep gilt: wert von max coin kann immer durch den max coin gewählt werden
Preise zu best münzset speichern

bei 25c,10c,5c,1c max 4 * 1c weil ab 5c, lieber 1* 5c ETC (nur **in US möglich weil die sich bilden aus den anderen**) => größte münze kann nicht ersetzt werden

präfix code

jedes zeichen unterschiedliche länge 0,01,011,0111 etc

greedy: wahrscheinlichkeit für zeichen bekommen kleinste werte

1. immer kleine wahrscheinlichkeit zusammenfassen
2. teilbäume werden von der wahrschenlichkeit zusammen gefasst (geht wiel unabhängig)

local search

- **immer wenn einem nichts mehr einfällt**
- **zufrieden mir jeglicher art der Annäherung**

1. **beliebige lösung** berechnen (sofern es geht sonst auch das vergessen)
2. **mutieren durch geringfügige lokale veränderungen**
3. **wenn besser dann übernehmen**
4. **ggf nur lokales optimum, darum 1. öfter mit anderer Lösung**

> **Zufalls reihenfolge erstellen**: array immer kleiner machen (letzten wert in lücke) und Randomzahl Modulo mit neuer länger
> **alle reihenfolgen**: rekusiv - alle testen

traveling salesperson

wenn vollvermascht: irgend eine reihenfolge wählen und berechnen **immer 2 knoten tauschen** wenn kanten richtung wichtig ist, muss deutlich mehr berechnet werden, sonst nur wenig kanten

auch mit mehr knoten für tauschen möglich

sortieren

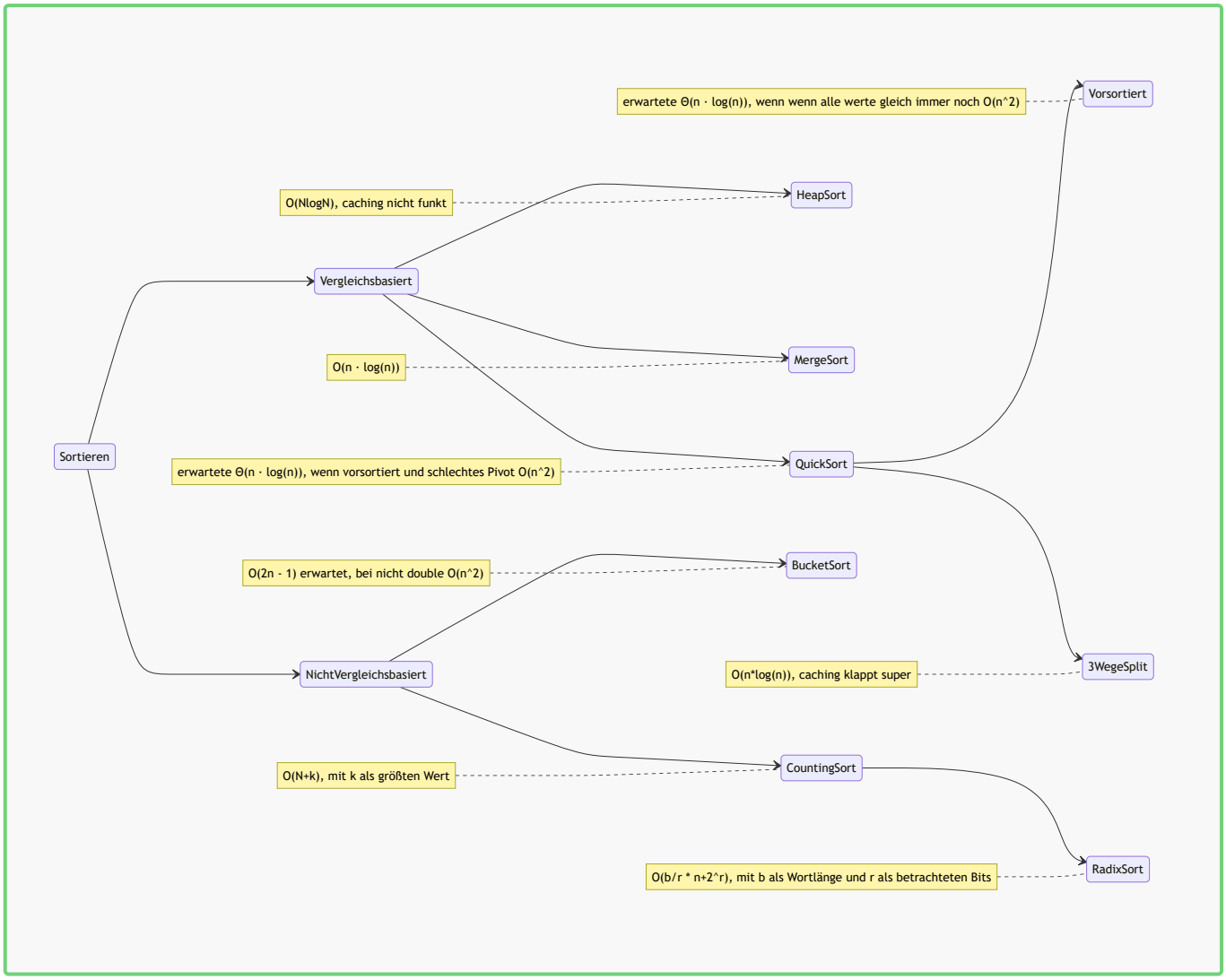
O(N log N) ist bei schlüssel vergleichen immer der worstcase, -> Mergesort und Heapsort

immer O(N log N) da:

- **höhe von Entscheidungsbäumen**
- Worst-Case
  - n! verschiedene Permutationen über n Zahlen
  - **Entscheidungsbaum hat mindestens n! Blätter**
  - **Binärbaum der Höhe h hat maximal $2^h - 1$ Blätter**
  - **also 2h ≥ n!**

$$h \geq log(n!) \geq log((n/e)^n) = log(n^n) - log(e^n) = n \cdot log(n) - n \cdot log(e) \ \in \Omega(n \cdot log(n))$$

(**Beweis durch baum, welcher immer weiter mit einer information geteilt werden kann**)



quicksort

worst-case Laufzeit ist $O(n^2)$ stark vorsortierte Folgen oder solche Folgen, die viele gleiche Elemente enthalten

Verbesserung für vorsortiert

Zufallsstrategie: wähle als **Pivot-Element ein zufälliges Element** aus A[l...r] und vertausche es mit A[l]. ⇒ Laufzeit ist unabhängig von der zu sortierenden Folge ⇒ mittlere/erwartete Laufzeit: Θ(n · log(n))

wenn links als p gewählt und es stark sortiert ist, fällt immer nur p raus $O(N^2)$
=> einfach zufalls wert nutzen wenn alle werte gleich immer noch banane, weil wenn alle gleich sind, das gleiche problem wie davor

3-wege-split-quicksort

=> **3. kategorie einbauen und die mitte (mit gleichen werten) nicht mehr anfassen**
=> **links und rechts durch 2 indexe gleich werte wie pivot sammeln**

- sonst aber auch bei nicht soguten aufteilung O(n*log(n))

- **um die rekusivtiefe zu reduzieren, wird das größere problem iterativ gelöst durch eine schleife und das kleinere problem rekusiv**
- dann bester quicksort
- **caching klappt super!**

heap-sort

max-heap (muss erstellt werden): erster ist letzter sort wert (niedrige zahl ist am anfang), dann **versickert zum je höchsten wert** - bis nach unten O(NlogN)
erstellt wird der heap durch werte über den blättern versickern (blätter sind schon für sich ein heap) (**ist linar, weil tiefe lansamer wächst als die anzahl zu versickernde** reduziert wird) O(N)
superlangsam, weil caching nicht funkt

mergesort

- **Teile die Folge in zwei etwa gleich große Teilfolgen**.
- **Sortiere die Teilfolgen rekursiv**.
- **Mische die sortierten Teilfolgen zu einer Folge zusammen**.
- $T(n) = 2 \cdot T(n/2) + c \cdot n \in O(n \cdot \log n)$

countingsort

werte von null bis k in einem k langen array zählen
dann ab schreiben
zurück schreiben von rechts (falls es z.b. klassen sind) => **dadurch ein stabiles verfahren (rechtes "Gleiches" objekt bleibt recht)**

O(n+k)

darum nur **sinnvoll bei kleinen ks**

RadixSort

nach stelle sortieren (im byte oder so) VON HINTEN nach vorne (geht nur wenn es stabil angewendet wird)

generell nur sinnvoll für ints!
für stringwerte z.b. problematisch, da nicht nur die ersten werte angeschaut werden
(by me: vllt von vorne und dann in gruppen?)

O(b/r * n+2^r), mit b als Wortlänge und r als betrachteten Bits

bucket sort

wie couting mit gruppen

einfügen in listen, an richtiger stelle (kann optimiert werden durch skipper und co)
(by me: das noch mal durch fächer erweitern? => z.b. via hashs)

> Zahlen zwischen [0 und 1) nach erster Nachkommerziffer gruppieren, dann sortieren

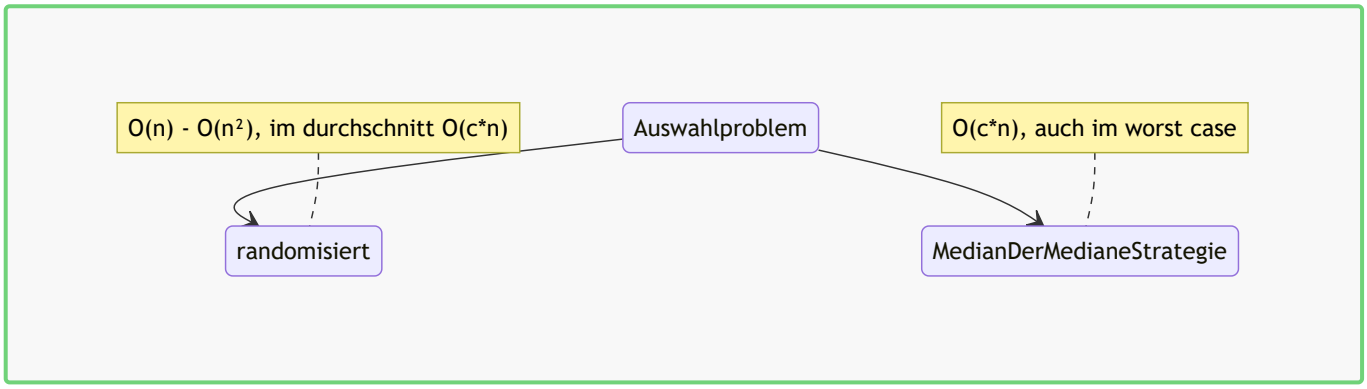
nur sinnvoll für double werte mit O(2n-1) erwartete, sonst schlechter O(N^2)

Auswahlproblem (Median-Berechnung)

- **das i-kleinste element einer unsortierten Liste finden**
- **wenn sortiert super izi**

> min max schneller finden: paare sortieren dann über die halben listen für min und max 3*n/2 statt 2n
> also min auf grade und max auf ungerade

i-kleinste, 4x das kleinste element finden: sortieren, dann zugreifen



randomisiert

(in dem fall immer das richtige Ergebnis (nicht immer die beste Zeit))

(**like quicksort aber nur immer eine seite weiter machen**)

random zwischen L und R pivot wählen und partitionieren

bei kleiner, R weiter machen bei gleich rückgabe bei größer,L weiter machen

O(n) - O(n^2)

im durchschnitt c*n

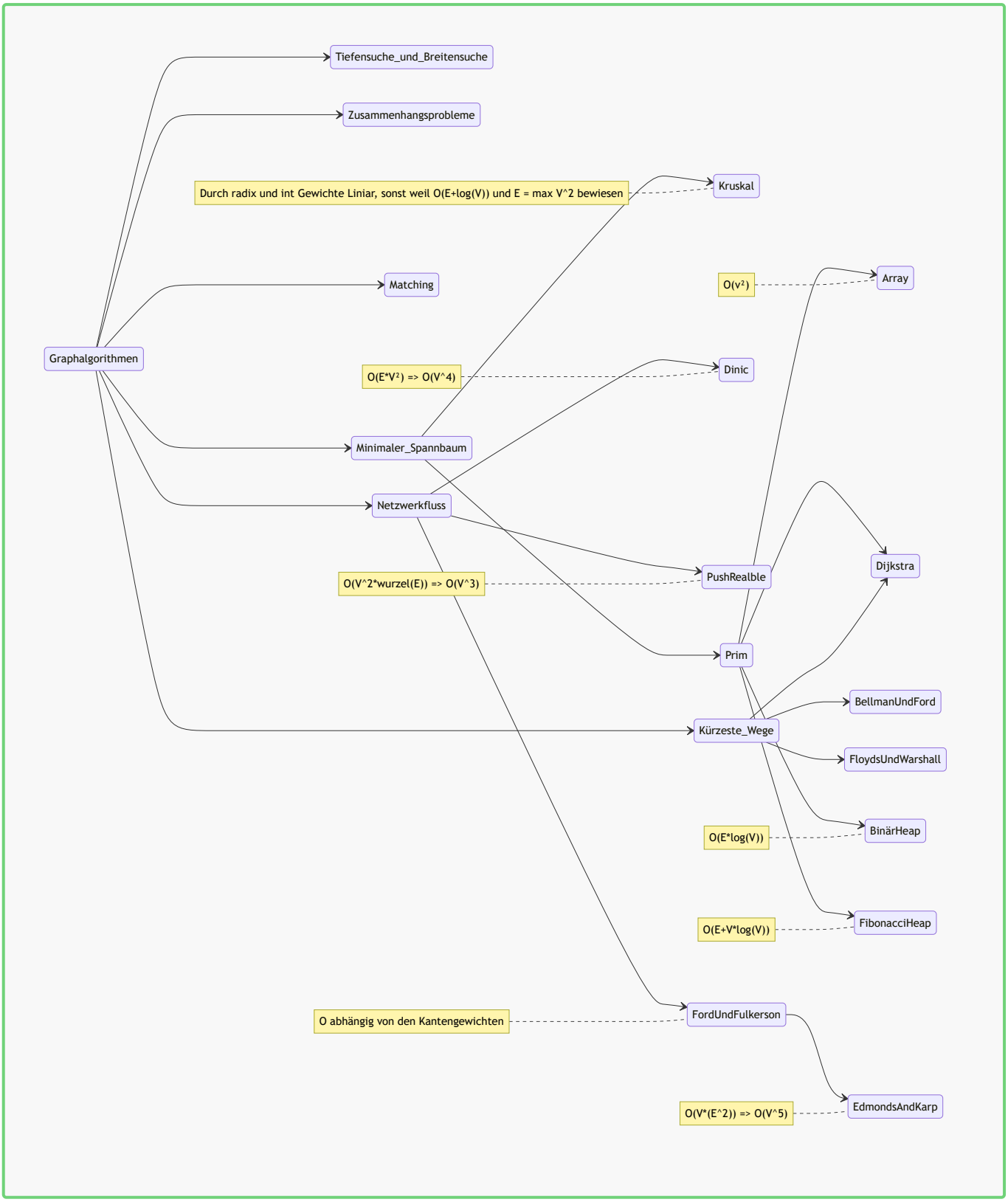
Median-der-Mediane-Strategie für worst-case

wieder prtitionieren, aber esfällt immer einen festen teil weg, durch die pivot wahl:

- **5er gruppen aufteilen**
  - **da drin median bilden** (konstant weil 5 (constant viele) elemente)
- darüber den median bilden
- dadruch in den 5er gruppen **immer min 2 kleiner/größer** (der 5)
- davon den median = ca die hälfte wieder kleiner/größer
  - **=> CA!! 1/4 der elemente muss snoch betrachtet werden**
- NICHT unendlich genauer medaian, weil z.b. nd druch 5 teilbar
  - => ist aba ja nur die pivot wahl!
- selbst im worst-case liniar!

> wieso nicht auch so im Quicksort? median-median-strategie geht **nur wenn alle elemente UNGLEICH** sind. 3-wege-split-quicksort weiterhin benötigt!

Graphalgorithmen



- **verbundene Knoten (Kanten) als 2-Tupel definieren**
  - N über 2 dadruch als max Kantenanzahl
  - mit gewicht, als 3-Tupel definieren
  - > Eine geordnete Zusammenfassung von Objekten heißt Tupel
- **keine Schleifen** (weil **Paare**)

- **keine parallelen** Kanten (weil **Menge keine doppelten Paare** zu lässt)
  - => dafür Multimengen benötigt
- sehr sehr mächiges modelierungs Werkzeug

Begriffe

unterschied gerichtete und ungerichtete Graphen

Ein gerichteter Graph G = (V, E) besteht aus

- einer endlichen Menge von Knoten V = {v1, . . . , vn} und
- einer Menge von gerichteten Kanten E ⊆ V × V .

Bei einem ungerichteten Graphen G = (V, E) sind die Kanten ungeordnete Paare: E ⊆ {{u, v} | u, v ∈ V, u != v}

Weg in einem Graphen

Sei G=(V,E) und u,v ∈ V p = $(v_0, v_1, \ldots, v_k)$ ist ein Weg wenn $v_0 = u, v_k = v$ und $v_{i-1}, v_i \in E für 1 \leq i \leq k$

induzierten Teilgraph

Teilgraph bei dem ein oder mehrere Knoten und alle von denen inzidenten Kanten entfernt werden

speicherung

Art der Speicherung hat Auswirkung auf Laufzeit der Algos und sind unterscheidlich groß

Adjazenz-Matrix

- in 2d array speichern => wenn verbunden 1 (oder gewicht) sonst 0
- quadratischer speicher
- ist x mit y verbunden, sehr schnell herrauszufinden**
- nachbarknoten in n

liste der vebundenen knoten

- wenig speicherplatz
- ist knoten x mit y verbunden im worst case n
- alle nachbar knoten ist einfach

adjazenz-array

- wie eine mischung
- eine liste welche mit index auf start und ende der liste verweist
- ansprechbar
- guten speicherbedarf
- und als list verwendbar
- DYNAMISCH NICHT SINNVOLL, wenn einer gelöscht wird ist ende

Tiefen- und Breitensuche

jeden knoten einmal **besuchen und nachbarn in die liste packen WENN unbesucht**

stack = tiefensuche (last in, first out) => alle daruf packen und checken **queue = breitensuche** (first in, first out) => alle rein packen und druch gehen

rekusiv ist Tiefensuche, weil geht erst ganz tief:

- gibt **nur all erreichbaren knoten**
- **globaler end und start zähler**
  - immer **schreiben bei Besuch**
- **baumkanten** erstellen, **wenn unbesucht**
- **wenn punkt auf stack**: **rückwärtskante**
- **wenn nicht auf stack aber besucht**
  - **beginn index größer**: **querkante** (bei **ungerichtet: baumkanten**)
  - **beginn index ist kleiner**: **vorwärtskante** .. HÄTTE eine Baum kante werden können (Bei **ungerichtet: rückwärtskanten**)

eigendliche Tiefensuche: solange noch ein unbesuchter punkt exestiert: tiefensuche von dort aus starten

kreise finden durch rückwärtskante finden!

topologische Sortiereung

tiefensuche kreisfrei + jeder knoten bekommt seine endnummer (a > b & a > c => a zeigt auf b und a auf c, anders rum nicht)

Gegeben: Ein gerichteter Graph G = (V, E). Gesucht: Eine Nummerierung $\pi(v_1), \ldots, \pi(v_n)$ der Knoten, sodass gilt:_ (u, v) ∈ E ⇒ π(u) > π(v)

Algorithmus: Modifizierte Tiefensuche G ist kreisfrei ⇒ dfe-Nummern sind topologische Sortierung

zusamenhangsprobleme

gerichtet

jeder knoten erreichbar: starkzusammenhängend (**schwach wäre bei richtung egal**) **stark zusammenhangskomponente: maximaler starkzusammenhängender induzierter teilgraph**

1. tiefen suche (höchste endindex, wird nächster start)
2. alle kanten umdrehen**
3. tiefen suche vom neuen start
4. DANN baumkanten sind starke zusammenhangskomponente

Erklärung

Anders erklärt:

G ist **stark zusammenhängend**, wenn es zwischen **jedem Knotenpaar** einen Weg in G gibt. **starke Zusammenhangskomponente** von G ist ein bzgl. der Knotenmenge **maximaler, stark zusammenhängender, induzierter Teilgraph** von G.

stark zusammenhangskomponente A und B

A zeigt auf B, dann kann nicht von B auf A (sonst nur eine)

1. Tiefensuche
    1. A start: kommt nach B und abarbeitet sie
    2. B start: macht B fertig und fängt dann A an => A hat hösten wert
2. Tiefensuche:
    1. wir können nicht aus A raus, makieren alle As
    2. kommen deswgeen nicht mehr aus B raus

ungerichtet

- **es heißt nur noch zusammenhängend**
    - **oder k-fach zusammenhängend**
- **nur Baum- und Rückwärtskanten**
- **Schnittpunkt: ein knoten welcher beim rausnehmen die zusammenhängenden komponenten erhöht**
    - k-fach zusammenhängend ist die Anzahl der punkte dir man rausnehmen muss, damit die zusammenhängenden mehr werden (?) - bzw wieviel wege es von einem knoten zum anderen gibt

bei splitung

wenn alle teilbäume einen weg nach oben haben dann ist es KEIN schnittpunkt
=> ergeben zsmhandgskomp

tiefensuche erweiteren

low wert: min (id, rückwärtskanten zeigt auf id, baum kanten (nach unten) low wert)
wenn er nicht kleiner als id ist SAFE ein schnittpunkt

vorgänger merken, da man sonst nicht zwischen rückwärtskante und baumkante (woher wir kommen) unterscheidbar ist

Minimale Spannbäume

ungeeignet für kürzeste Wege.

definition

- baum => kein kreis
- zusammenhängend (ig? by me)
- **teilgraph eines graphen**
- **gleiche knoten menge** (also die selben)

- **V-1 kanten => damit kein kreis** (V = anzahl der knoten | E = kanten)
- gesucht: **min costen**

idee

1. Knoten **partitionieren**
2. **billigste kante wählen, welche die partitionen verbindet**

Kruskal

1. **jeder knoten in seine Menge**
2. kanten **sortierne nach nicht fallenden gewicht** (>=)
3. **kanten durch laufen**
    1. schauen in welcher Menge u und v ist und wenn unterscheidlich (und kein kreis)
    2. kante **hinzu nehmen**
    3. **Mengen von u und v vereinigen**

Speicherung: **blätter** erstellen, wenn 2 zam gehören eine wurzel erstellen
kleinerer baum an WURZEL des größeren anhängen (damits flach bleibt)
=> sozusagen ein umgedehter baum von blätter zur wurzel

speicher rang und vorgänger (wurzel = 0)

union

1. wurzel finden
2. ränge vergleichen
3. wenn gleich sind muss der rang + 1 - sonst an den größeren hängen

$O(e log(v)), weil max v^2 sein kann und durch log(v^2) => 2 log(v)$ gezeigt, alles so lang

init O(V)

optimerung

pfad komprimieren und direkt die Wurzel für alle Spreichern wenn durch einen 2. durch lauf bei finden

mit radix durch int gewichten + die optimerung: liniar!

Prim

1. benötigt **Prio-Liste**
2. für **jeden knoten, Wert auf unendlich**
3. **startknoten wählen und auf 0 setzen**
4. **preiswerteste aus liste nehmen** und entfernen aus der liste
    1. **alle nachbarknoten anschauen**
        1. **wenn nachbar noch in liste und das gewicht kleiner ist**, **schlüssel von nachbar aufs gewicht verkleinern**
        2. und **als vorgänger speichern**

Listen speicher

Array: O(v^2) binär-Heap: O(e*log(V)) (wurzel raunehmen, kleineres versickern, und bei decrease nach oben vertauschen) fibonacci-Heap: O(*E*+*V*log(V)) (wird noch gezeigt - aber soll krass sein)

E = V^2 dann gehts nd besser, sonst ist fibo besser

Kürzeste Wege

üblich:

- egal ob **ungereichtet oder gerichtet**
- **zusammenhängend**
- **positive gewichte**
  - weil A sonst in nem **loop der kleiner 0 ist immer wieder durchlaufe**
  - und B man **nicht einfach auf 0 bis unendlich "normalisieren"** kann
    - da sonst die normalisierung unterschiede aufweist zwischen wegen über einer vs über mehrere kannten
  - und C kurze werte fallen dann zu spät auf und es müsste ALLES was folgt neu berechnet werden was die laufzeit hochtriebt

> Längerster Weg ist NP vollständig weil reduktion von Hamilton Path (negative kreise funken nicht)

weil minimal gleich ist wie maximal (in minus) => daher auch NP vollständig (dennoch polynominell lösbar???)

VORSICHT: ANDERS ALS MIN SPANNBAUM

Optimalitätsprinzip: weg von V0 bis Vk am kürzesten, dann auch alle teil wege kürzeste wege
Dreiecksgleichung: von a nach c max a nach b + b nach x

Dijkstra

genau **wie Prim (auch laufzeit)**
knoten schlüssel + der weg bis her

nicht für Routenplanung

weil **restriktions**: **form-, via-, to-rules**, beim z.b.: nicht links abbiegen (also verbote)

algo umwandeln oder graph modifizieren (mega knoten von 8 und intern dann mappen wohin)

Beweis

- **durch wiederspruch**
  - **annehmen, dass es einen kürzeren gibt**
  - **Optimalitätsprinzip wurde verletzt**
- dann iwie dis die distanz nicht kleiner sein kann .. wow.
  - werte können nur kleiner werden
- vorgänger muss kleiner sein

- sollte u nicht richtig sein muss der vorgänger gleich sein
- das geht nd

Breiten suche

wenn alle Kanten gleiches gewicht haben, dann super schnell

Bellman / Ford

- **Berechnet: Kürzesten weg von einem zu allen Knoten**
- **Werte in einem Array**, ggf zusätzlich weg speichern
- **kann auch negative werte**
- **alle kanten neu berechnen v-1 mal** (max länge - ausser negativer kreis)
  - dann noch mal - **bei erneuter Änderung ein negativer kreis**
  - => **ungültig**
- v0 bis v1 iwann in durchlauf 1 und vk-1 bis vk iwann in druchlauf k
- Ablauf:
  - Die äußere Schleife wird (V − 1)-mal durchlaufen.
    - Die innere Schleife wird E-mal durchlaufen.
      - Alle Operationen der inneren Schleife kosten Zeit O(1).
- → Gesamte Laufzeit in Θ(V · E).
- **Korrektheit: kürzester Weg ohne negativen Kreise besucht keinen Knoten zweimal daher höchstens V-1 Knoten**
- **optimierung:**keine kreise: topologische sortierung mit triefen suche - so durch laufen und abbrechen bei keiner Änderung
- **erweiterung**:ggf müssen knoten doch öfter besucht werden als v-1 (like e-roller routen mit aufladen)
  - (neuge algos werden entwickelt)
  - da auch negarive werte
  - dort gibt es aber keine negativen kreise, weil sonst energie problem gelöst

Floyds

- **Achtung: Negative Kantengewichte sind erlaubt!**
- **kürzeste wege zwischen JEDEM knotenpaar**
  - Also Matrix

> zeigt alle Verbindungen an von jedem zu jedem knoten

- **nehmen jeden knoten einmal raus und testen ob es weniger wird, dann speichern**
- **3 for schleifen**
- $O(N^3)$: Jeden Start und jeden End knoten mit jedem Zwischen knoten überprüfen
- **kreise werden iwie erkannt**
- 2d array reichet, weil alles von der letzte runde verworfen wird
- Erstellen einen neuen Graphen mit Kanten für alle Wege (verbindungen über Knoten und Kanten von einem Knoten zum anderen)
  - **Transitiver abschluss**
  - In G* existiert eine Kante (vi, vj) genau dann, wenn in G ein Weg zwischen vi und vj existiert. Sei auch hier V = {1, 2, . . . , n}.

Netzwerkfluss

- **Graphen mit kapazitäten (c) als Gewicht an den Kanten**
- mit **quelle (s)** und **senke (t)**
- **gesucht wird der maximale fluss**
- fluss ist **mindestens 0 (bei uns) und max c**
- was **in** den knoten reingeht muss **auch wieder raus (ausser s und t)**
    - (ggf kann man auch eine quelle und senke machen die in und out hat)
- **aktive knoten: knoten mit überschuß**

Präfluss

keine negativen überschüße
aber Überschuss und iwas mit kapazitäten

Schnitt

Menge S: menge mit quelle Menge **not(S)**: ohne quelle **mit senke**

kapazität des Schnittes H(S) ist c(S): alle **kapazität der kanten c(e)** die **von S nach not(S) aka. E(S,Not(S))** laufen **addieren**

der **fluss F(f) ist an jedem schnitt ablesbar** durch alle f(e) von S nach Not(S) - alle f(e) von not(S) nach S (weil **Änderungen = 0** sind **wenn man einen knoten verschiebt**)

f(x) <= c(x), wobei der genaue wert von gesammt fluss abnhängt

> der **fluss** ich **also MAX**: min c(S) **kapazität des min schnitts**

restgraph

Restgraph G_R **enthält alle vergrößerbaren pfade** also die wo noch "rest" kapazität ist (f(e) < c(e)) und **vergibt die gewichtung c(e) - f(e) als Rest**

schichtgrpah

Teilgraph von G_R, der nur Kanten (u, v) ∈ E_R enthält, für die δ(s, v) = δ(s, u) + 1 gilt.

> also nur die, die eins weiter rechts sind außer es gibt keins weiter rechts,
> wobei die quelle s ganz links ist

Der Restgraph nur mit Kanten mit wachsender Distanz zu s

Tiefensuche auf Schichtgraph ist ein kürzester Weg also auch okey hier

Aufbau mit Breitensuche:

- wenn volle kanten da waren, sind sie weg
- vorwärs, was noch könnte
- rückwärts was wenn der fluss ernidrigt werden kann

Rückwärts Kanten müssen auch betrachtet werden, falls die aktuelle funktion "falsch" verteilt
Fluss kann also vergrößert werden über eine rückwärtskante

Blockierender Fluss

Jeder Weg von s nach t im Schichtengraphen enthält eine gesättigte Kante. = keine zunehmenden Pfade mehr

Zum erreichen mit tiefensuche (Dinic):

1. **bilde Restgraph**
2. **bilde schichtengraph O(E)**
3. **Suche einen Weg von s nach t**
4. **kleinsten Wert des Weges** (im schichtengraphen) von allen Kanten des Weges **abziehen**
5. Kanten mit **Null-Werten werden rausgeworfen**
6. **springe zu 1 bis keine Ändernung mehr**
    - max O(E) mal (und dauert O(V))
    - => O(E*V), weil wegen O(V)
    - => O(E*V²) => $O(V^4)$

erreichen mit Ford und Fulkerson

1. **jeden flusswert 0**
2. verscheidenen regeln wann mehr oder weniger werden könnte und **kleinsten pfad > 0 (von s nach t) speichern**
    - Regeln: mal oben mal unten rum
3. **erhöhen des flusses um delta-Werten (kleinster wert) auf dem pfad**
4. **2 und 3 immer wiederholen solange wir einen finden**

> Laut internet wie Dinic ohne schichtengraph - aber mit dem problem, dass auch kanten zurück führend es sehr komplex machen

- **Für ganzzahlige Kapazitäten ist die Anzahl der Flussvergrößerungen durch F_max beschränkt**
    - Daher: Die **Anzahl der Flussvergrößerungen ist abhängig von den Kantengewichten**, und **nicht allein abhängig von der Größe des Graphen**!
- **Problem**:
    - terminiert nicht immer für irrationale Kapazitäten.
    - konvergiert für irrationale Kapazitäten nicht einmal unbedingt gegen F_max

Edmonds and Karp

Anzahl Iterationen: O(V · E) Breitensuche zum Finden kürzester Pfade: O(E)
Wähle immer einen zunehmenden Pfad mit der minimalen Anzahl Kanten (durch breitensuche) O(V* (E^2)) wenn E = V² => $O(V^5)$

> gibt an WIE Pfade bei Ford und Fulkerson gefunden werden

Push-Realble-Algorhitmen

- ermöglicht **überfluss/überschuß WÄHREND des Algos (aktive knoten)**
    - => **weniger abtragen als reinkommen**

- arbeitet wieder **mit restgraph**!
- **start knoten sendet alles was geht**
- **startknoten höhe = V (damit erst mal alles erreicht ist)**
  - => **dadruch höhe auch begrenzt**, weil zurück muss **der weitentfernteste nur (fast) doppelt so hoch** sein
- **push**: **überschuß abtragen zum nächsten knoten**
  - von-knoten brauch größere höhe (eig genau +1) als der zu-knoten
  - **bei rückwärtskante wird es abgezogen**
- **sättigend**: kante verschwindet, weil wird voll genutzt
  - max (E*V) oft sättingde
  - und max $O(V^2*E)$ oft nicht sättingede
- **relable**: **verändert höhe**, auf **kleinster nachbarknoten +1**
  - max $O(V^2)$ mal
- **solange push bis kein übscherschuß mehr**, wen kein push (nirgends mehr) möglich:
  - **dann immer relable!**
  - **derweil** immer ein **präfluss**
- wenn gut fest gelegt wird, zu welchen wir pushen dann:
  - Max $O(V^3)$ bzw $O(V^2*\text{wurzel}(E))$
  - (wähle den am höchsten liegenden)
- **Terminiert weil**:
  - **Ein Überschuss an einem beliebigen Knoten kann immer zur Quelle hin wieder abgebaut werden.**
  - Die **Knoten können nicht beliebig angehoben werden**

Matching-Problem

Um eine bestmögliche Zuordnung zu finden (um möglichst viel zu verwenden)

- **Teilmenge von kanten, welche keinen gemeinsamen knoten** haben
- **Maximales Matching: nicht zu vergrößeren (im aktuellen Zustand)**
- **Maximum Matchting: größtest matching** (max JEDEN knoten 1x) => V/2 groß
- stundenplan, gleiche prob mit 3 dims
  - auch disjunktiv wie studenten auf plätze (bipartites Matching)
- **bipär = kein kreis ungerader länge** (aufteilbar und es zeigt nur eine menge aud die andere)

mit fluss algo

- **quelle und senke hinzufügen**
- jeden knoten von **v1 mit quelle verbinden**
- kanten werden von **v1 nach v2 gerichtet** (c = 1)
- **v2** wird **mit senke** verbunden
- **Flüsse auf kanten (0 oder 1) geben** an welche kante eine verbindung ist und welche nicht

mit idee des fluss algos

(weil **weg suchen und erhöhren**)

alternierender weg suchen welcher abwechselnd matcheing und nicht matching kanten
dann tauschen, sofern der 1. und der letzte knoten NICHT an einer matching kante hängen (zunhemend alternierend)
=> eine matschingkante mehr
geht also mit knoten die GARNICHT verbunden sind (länge 1) und sonst auch nur bei ungraden weg längen
muss glaube ch generell überprüft werden ob man den knoten sonst noch iwo drin hat und obs am ende dann noch wiirklich mehr ist

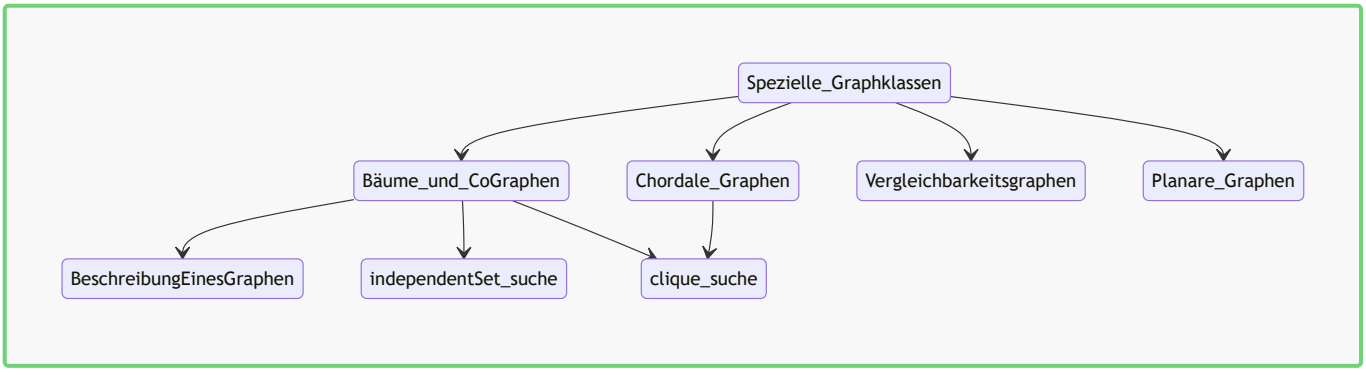
also:
teifen-/breitensuche (starten bei einem knoten ohne matching kante)
max v/2 runden
O(V*(V+E)) => max V^3

matching auf allgemeinen graphen

- **Kreise ungrader länge machen dann probleme**
- **Kreis ungerader Länge heißt Blüte**
- entdeckte **mit einem megaknoten (Blüte) ersetzen**
  - (durch ein 2. mal besuchen)
- **in der blüte dann wen von zu bis vertasuchen** (wenn nicht nur berührt sondern druchläuft)
  - drin **immer den geraden weg** nehemn
- O(wurzel(v)*E)

Spezielle Graphklassen

- Eine **Grapheigenschaft heißt monoton, falls** für jeden Graphen mit dieser Eigenschaft gilt, dass **auch jeder seiner Subgraphen diese Eigenschaft hat**.
- Eine **Grapheigenschaft heißt hereditär, falls** für jeden Graphen mit dieser Eigenschaft gilt, dass **auch jeder seiner induzierten Subgraphen diese Eigenschaft hat**.
  - => **wenn monoton dann auch hereditär**
- **k-färbung**: **unabhängig patritionierne** und **jede bekommt ihre eigene farbe** - **min anzahl chromatische zahl oder so**
  - patritionierne: 2 Knoten einer Partition dürfen keine Kante haben
- **max Clique ist das min der k-farben**
- **Independent Set Anzahl ist min von Anzahl der cliquen**
- **zweigraphen sind isomorph zu einander, wenn man einen graph anders durch nummeriert und den anderen erhalten kann**
- **Graphen werden durch eigenschaften in klassen unterteilt**



Bäume und co-graphen

- **Wald = menge von bäumen bzw. ein kreisfreier graph**
- die **einzelnen bäume werden zu mehren bei trennungen weder monoton noch hereditär**
  - **darum wald monoton und hereditär**
- **bipartit (keine ungraden kreise)** sind **monoton und hereditär**
  - **da der kreis schon vorher drin sein müsste**
- beides wichtig für rekusive algos
- **Co-Baum beschreibt die verbindungen und vereinigte mengen**
  - in länge der Tiefen/breiten O(V+E) suche Co-Graph zu erkennen
  - **besteht nur aus disjunkten summen und disjunkte vereinigung**
    - Summe (x): Verbinden
    - Vereinigung (U): totalvermascht
  - Graph mit nur einem Knoten ist auch n Co-Graph

> Co-Graph beschreibt die Verbindung/den Zusammenhang der einzelnen Knoten eines Graphens

suche independent set:

- **blatt = 1**
- **vereinigung**: **addieren der max independent sets**
- **summe (alle): nur max independet set** der beiden
- Für jedes Blatt v in T setze α(G[v]) = 1.
- Für jeden mit ∪ gekennzeichneten Knoten v mit den Kindern v1 und v2 in T setze α(G[v]) = α(G[v1]) + α(G[v2])
- Für jeden mit × gekennzeichneten Knoten v mit den Kindern v1 und v2 in T setze α(G[v]) = max{α(G[v1]), α(G[v2])}

suche Clique:

- **blatt = 1**
- **vereinigung: max wert**
- **summe: addieren der max cliquen**

suche färbung like clique

partition in cliquen like independent set

chordale Graphen

- **= wenn ein kreis länge >= 4 existiert muss es eine abkürzung geben**
  - VORSICHT: **nicht nur dreiecke nutzten -reicht nicht!**
    - damit lassen sich ja auch Kreise bilden!
  - ???sozu sagen skippen in den kreisen für jeden graden oder jeden ungraden knoten???
- **nur hereditär**

clique

- bedingung:
  - in h1 (links) und h2 (rechts) gibt keine abkürzung da min weg

    - in s (mitte) muss dann eine clique sein damit jedes knoten paar eine sehne erhält
- finden (geht nur wenn chordal):
    - **perfektes knoten-Eliminationsschema PES:** so aufzählbar das alle verbundenen knoten die noch folgen,
        - **dann noch verbunden zum EINEM vorherigen Knoten: clique bilden**
    - simplizialknoten s finden und aufschrieben um PES zu bilden, denn dan rausnehmen und den nächsten s wählen
- labeln:
    - i = anfagnegen bei count knoten
    - von a alle nachbarn i
    - i--
- loop bis end:
    - größte lable als nachbar suchen
    - alle nachbarn i
    - i--
- umgedreht (von klein nach groß)= PES
- max clique und indeoendent set auch iwie durch chordale Graphen leicht berechenbar

> das was der erkennungs algorhytmus liefert kann genutzt werden um andere probleme zu lösen

Vergleichsgraphe

- **Ein ungerichteter Graph heißt Vergleichbarkeitsgraph, wenn eine Orientierung der Kanten existiert, so dass der orientierte Graph transitiv ist**.
    - Alle zeigen auf einen Punkt (über andere punkte)
- **jede ungerichtete kante benötigt eine richtung**
- wenn die bedingung gilt **wenn a => b und b => c dann auch a => c (immer) = Vergleichbarkeitsgraph**
- **Gammarelation**: **wenn a nach b dann auch a' nach b, wenn KEINE aa' kante existiert**
    - dadurch:
        - keine schleifen oder paralelle kanten
        - wenn ba in E dann ab in E^-1
- **Implikationsklassen sind** die, welche durch die bedingungen von einander abhängig sind
    - **Gruppen die von einem Punkt ausgehen (direkt)**
- **berechnet wird ein dekompositionsschema**, eine **menge an kanten welche in einer knotenmenge liegen die eine Implikationsklasse liegen**
- Implikationsklassen auslesen/hübsche pfeile machen:
    - Mit Gammarelation (nicht die andere da) eine Implikationsklasse brechnen und samt ingegengerichtete kanten raus nehmen
    - wenn noch weitere kannten dann noch mal machen

- $O(E^2)$

- **daraus folgen kanten in jedem schritt** (die gewählte)

- und **eine transitiveorienterung des graphen** => **kein kreis** => **topologische sortierung (rückwärts - Reihenfolge wie aus stack genommen) möglich**

  - => **max clique via. längsten Weg berechnung (entlang der sortierung)**
  - => **max indepententset durch bipatite verbindung (via flussalgo) also ein max matching (berechnet min vertex cover (= independetset))**

> durch eine Erkennung folgen also schnelle lösungen für schwere probleme

Planare Graphen

- **graph den man kreuzungsfrei in die Ebene zeichnen KANN**
  - **Planare Einbettung, wenn man das auch macht**
  - Alle teilgraphen sind es dann auch (**monoton**)
- Vertretter
  - 4er cluqie ja
  - 5er nein (K5)
  - 3 und 3 welche vollvermachst sind auch nein (K3,3)
- **eulers polyeder formel (für planare Grpahen)**:
  - **Anzahl Knoten - Anzahl Kanten + Anzahl Flächen = 2**
    - **Anzahlflächen wird auch die äußere gewertet**
    - Kein Kreis = eine fläche
  - knoten - kante beim baum (keine kreise) = 1
    - => + Fläche = 2

> immer wenn man eine kante rausnimmt verliert man eine fläche, wenn man also das = 2 hat kann man immer zu einem baum reduzieren

wichtig:

- **max E <= 2*V - f**
- **Eulers Polyeder-Formel gilt V − E + f = 2**

weil:

- **jede kante hat max 2 flächen** und auch nur kreiskanten
- **jede fläche brauch min 3 kanten** (weil keine schleifen und parallele kreise)
- dann die eulers polyeder formel einsetzen

bei **bipartiten graphen**:

- **safe keine ungraden kreise**

  - => **jede fläche min 4 kanten**
  - => E <= 4*V -8 => /2 => E <= 2*V-4

- dadruch k5 nicht planar, weil max 9 kanten hat aber 10

    - bei k3,3 nicht planar, weil max 8 aber 9
- **maximal kontengrad 5** bei planaren graphen
- **Graph-Minor** ist ein **teilgraph oder eine kante berachtet und durch EINEN knoten ersetzt**
    - wenn ein **graph kein K5 oder K3,3 als minor** ernthält **dann Planar**
- **Unterteilungs Graph wenn eine kante durch nur neue knoten und kanten ersetzt**
    - wenn ein **graph weder K5 noch k3,3 als unterteilungsgraph** enthält **dann Planar**
- **"Grad 2 kanten" zamfassen dann alle 5 und 6 elementigenteilmengen prüfen** ist dann **eine erkennung für planare** graphen
- geht aber **schneller mit magic like PQ-bäume**
- **jeder planare graph kann mit 5 farben gefärbt werden**
    - Es gibt **einen Knoten v mit Knotengrad höchstens 5**
        - Hätte jeder Knoten einen Knotengrad von mindestens 6, dann gäbe es mindestens $\frac{6 \cdot V}{2} = 3 \cdot V$ viele Kanten in dem Graphen, was aber nicht sein kann, da jeder planare Graph höchstens E ≤ 3 · V − 6 viele Kanten hat.
    - Wir berechnen eine **Färbung für G − {v} mit 6 Farben**.
    - **Anschließend fügen wir v mit den dazu gehörenden Kanten wieder ein.**
        - Da der Knoten v in G höchstens 5 Nachbarn hat, ist noch eine Farbe für v frei.
    - mit 4 ist nicht bewiesen nur probiert mit PC
    - 3 ist NP-vollständig (auf 3 KNJ SAT abbildbar)

max 4er clique

- **farbe 1 mit 0 kanten, farben2 mit tiefensuche möglich sonst normaler weise k sonst NP**

Vorrangwarteschlangen

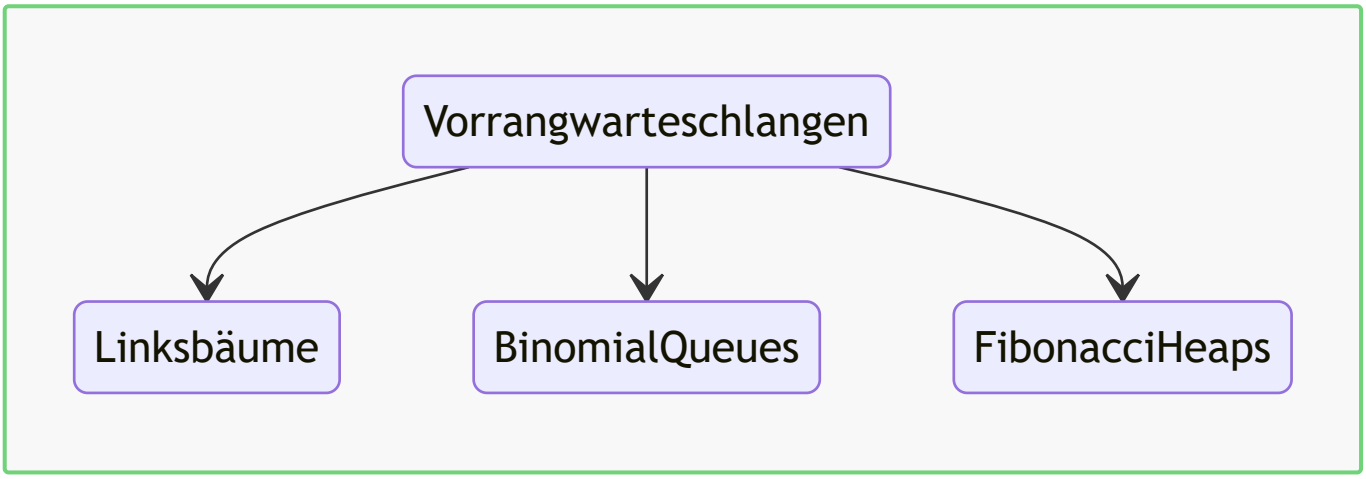
Linked List und Binary Heap so kombinieren, dass die vorteile beider genutzt werden

brauchen wir für: kürzeste Wege nach Dijkstra, Minimaler Spannbaum nach Prim. Bei Kruskals Algorithmus zur Berechnung minimaler Spannbäume kann anstelle des Sortierens der Kanten zu Beginn auch eine Priority-Queue verwendet werden. (In WSY haben wir den A ⋆ -Algorithmus kennengelernt.)

Soll folgende Operationen unterschützen:

- **MakeHeap**() erzeugt neuen Heap ohne Elemente
- **Insert**(H, x) fügt Knoten x in Heap H ein
- **Minimum**(H) liefert Zeiger auf Knoten mit minimalem Element im Heap H
- **ExtractMin**(H) entfernt minimales Element aus Heap H und liefert Zeiger auf den Knoten
- **Union**(H1, H2) erzeugt neuen Heap, der die Elemente aus H1 und H2 enthält
- **DecreaseKey**(H, x, k) weist Knoten x im Heap H neuen, kleineren Wert k zu (Zeiger auf x muss bekannt sein)
- **Delete**(H, x) entfernt Knoten x aus Heap H (Zeiger auf x muss bekannt sein)

	Linked List	Binary Heap
MakeHeap	Θ(1)	Θ(1)
Insert	Θ(1)	Θ(log(n))
Minimum	Θ(n)	Θ(1)
ExtractMin	Θ(n)	Θ(log(n))
Union	Θ(1)	Θ(n)
DecreaseKey	Θ(1)	Θ(log(n))
Delete	Θ(1)	Θ(log n)



Linksbäume

- **(min-)heap geordnet (kinder sind >= Knoten)**
- **als binär bäume aufgebaut**
- **Innere Knoten**: **Distanzwert des rechten Kindes plus 1**.
- **diztanz des linken kind muss >= des rechten kind**
  - => **links tiefer als rechts**
  - => **ziel rechtslastig arbeiten**
- **max 2^Distanz als blätter ins gesamt**
- erstellen und min in O(1)
- **rest durch Union** - Rekusiv aufrufen:

1. der **teilbaum mit der größeren wurzel kommt rechts unter den anderen**
2. **Dann könnte rechts tiefer sein**, **wenn ja links und rechts tauschen**
3. **Rekusions ende wenn ein baum als blatt eingefügt wird**

- **rechts ist log(n) tief** dardruch **alles andere als make(x) und min() (O(1)) in O(log(n))**

- **beim delete entstehten unten 2 links bäume** und der alte hauptbaum muss ggf noch getauscht werden nach dem aktuellisieren der distanzwerte

  - **dann die beiden linksbäume wieder unionieren**
  - **internet sagt O(N)** - **aber es ist O(log(N))**

        - **weil wenn wir links alles tauschen müssen, dann nur wenn es in einem balanzierten baum steht**

- Insert(D, x): **Erzeuge einen Linksbaum E, der nur einen einzigen inneren Knoten x mit Distanz 1 hat.**
    - Führe d**ann Union**(D, E) aus. → Laufzeit: O(log(n))
- ExtractMin(D): **Entferne die Wurzel und verschmelze die beiden Teilbäume der Wurzel.** → Laufzeit: O(log(n))
- Delete(D, x):
    - der **Linksbaum zerfällt beim Entfernen von x in den oberhalb von x liegenden Teilbaum in den linken und rechten Teilbaum**
    - **ersetze den Knoten x durch ein Blatt b**
    - **tausche ggf.** b **mit seinem Geschwister**, um links den Teilbaum mit größerer Distanz anzuordnen
    - **adjustiere die Distanzwerte von b bis zur Wurzel**
    - **verschmelze die drei Teilbäume miteinander** → Laufzeit: O(log(n))
- DecreaseKey(D, x, k): **Lösche Knoten x** aus D mittels Delete(D, x). Andere den Schlüssel von x auf k und **füge ihn mittels Insert(D, x) ein**. → Laufzeit: O(log(n))

Binominal-HEAP/-Queues

- **Mehrere heap geordnete Binomnalbäume (wurzeln sind verkettet)**
- **jeder nächste baum hat doppelt so viele wie der vorgänger baum**
    - eine kopie die unter die wurzel des anderen gehangen wird
    - anzahl der nachfolger steigt dadruch
- **wenn man ein min entfernt bleiben kleinere binominal heaps stehen**
- B_k hat 2^k knoten und die höhe k
- in jeder ebende sind k über i (tiefe) knoten
- **Binär-Darstellung von N gibt an welche und wieviele bäume man braucht** von B_k k ist die wertigkeit des bits - jede 1 ist ein baum und 0 wird ignoriert
    - wurzel der binären zahl hat max log(N) bits
- make wieder O(1), **min aber O(Log(N))** - da man jeden baum durch gehen muss
- **Union** wird immer der gleiche typ vereinigt .. **wie binäre addition** O(log(N))
- **insert neuer baum wert k 1, dann union**
- **extract min raus nehemen (log(n)) und union => O(log(N))**
- **DecreaseKey**(H, x, k): **Setze den Schlüssel von Knoten x auf den Wert k und führe dann UpHeap(H, x) aus**
    - Laufe im Baum hoch und vertausche den Knoten mit seinem Vorgänger, falls der Vorgänger einen größeren Wert speichert. → Laufzeit: Θ(log(n))
- **delete knotne nach ganz oben und dann rauswerfen**
- nur wegen O(log(N)) **schlechter als Linksbäume**

Fibonacci-Heaps

ränge und makierungen + Binominal-Queues-Idee

- **Bäume einfach nicht vereinigen beim lösche (doch via ExtractMin) und einfügen**
- Erst **bei extractmin das vereinigen nachholen**
- **zusatz zeiger auf min element** (damit man es nicht mehr suchen muss)

keine feste Struktur

- **Sammlung min-heap-geordneter Bäume**
- Struktur implizit durch erklärte Operationen definiert

> jede mit den bereitgestellten Operationen aufbaubare Struktur ist ein Fibonacci-Heap

Implementierung:

- die **Wurzeln der Bäume sind doppelt zyklisch verkettet**
- **Zeiger auf das kleinste Element** der Wurzelliste
- **Kinder der Knoten ebenfalls doppelt zyklisch verkettet**
- **jeder Knoten hat einen Rang und ein Markierungsfeld**
- **Rang: entspricht der Anzahl der Kinder**; es werden **nur Bäume gleichen Rangs verschmolzen**, damit "breite" Bäume entstehen (siehe ExtractMin)
- **Markierungsfeld**: es sollen **nicht zuviele Knoten eines Teilbaums gelöscht werden**, damit die Bäume nicht zu schmal werden (siehe DecreaseKey)

union ist nun einfach O(1): ketten werden vereinigt

Insert genau so + min überprüfen => **ganzviele insetr = liniare liste**

extract min:

- **rauswerfen und kinder mit rein nehmen in wuzel liste**
- **bäume vom selben rang** (anzahl der kinder) **werden verschmelzt**
  - **array merkt sich für jeden möglichen rang einen zeiger**
  - **currentzeiger zeigt auf min**
  - **durch laufen** (max O(N) (nur am anfang) also **amortisiert nur O(log(N))** (also durchschnitt))
    - **zeiger vom entsprechneden array eintrag auf knoten**
    - **wenn es schon vorkommt VERSCHMELZEN** (große wurzel unter die kleine)
      - **Rang wird erhöht**!!
      - **wenn besetzt weiter verschmelzen**

> nicht geegnet für echtzeit system wegen langen ausreißern (gesamt bearbeitung ist aber kurz)

DecreaseKey:

- **verkleinert in die wurzel liste packen** und min ggf neu setzten
- **könnte zu dünn werden**, darum **markieren**
  - **eltern teil markieren**, als **nicht mehr trennen**
    - wenn **schon makiert elternteil auch trennen**: **in wurzel** wird **markierung entfernen**
      - und sein **vorgänger makieren worstcase O(Log(N)) amortisiert O(1)** also konstant

Union(H1, H2): **Hänge die Wurzellisten** von **H1 und H2 aneinander** und setze den **neuen Minimalzeiger auf** das **Minimum** der bisherigen Minimalelemente von H1 und H2. → Laufzeit: Θ(1) **Insert(H, x)**: **Erzeuge Fibonacci-Heap** H', der **nur x enthält**. Der **Rang von x ist 0** und das Element ist **nicht markiert**. **Vereinige H und H' mittels Union**. → Laufzeit: Θ(1) **Delete(H, x)**: Setze **x auf einen sehr kleinen Schlüssel (-unendlich)** und führe dann **ExtractMin(H)** aus. → Laufzeit: O(n) im worst-case, O(log(n)) amortisiert

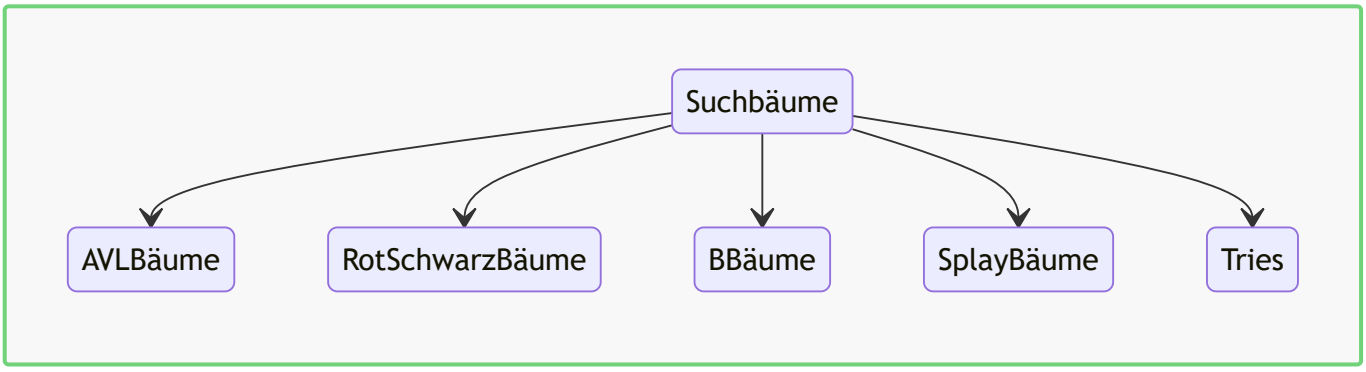
c++ bib für prioqueue hat kein DecreaseKey oder Delete, weil verringert einfügen statt decrease für delete makeriung erstellen (wobei delete nicht benötgt wird eig)
?hätte gedacht folge stuff muss vorher iwie hoch genommen werden oder so

routen planen z.b. fast planare graphen also max 3x jeden knoten einfügen

gibt noch n paar verbesseurngen

Suchbäume

links kleienr rechts größer



basic

Normal im **worstcase eine liste Ohne Balancierung** würde durch das Einfügen von aufsteigend sortierten Werten ein **zu einer Liste** degenerierter Baum entstehen. **Beim Einfügen** eines Wertes muss also immer **bis ans Ende der Liste gelaufen werden**. Als Laufzeit zum Einfügen von n Elementen in einen solchen Baum erhalten wir daher: $T(n) = \sum_{i=1}^{n} i \in \Theta(n^2)$

Einfügen zufälliger Werte: Beim **erfolgreichen Suchen** eines Wertes werden gerade **soviele Vergleiche** benötigt, wie der **gesuchte Knoten von der Wurzel entfernt** ist

walks

- **inorder: links ausgabe rechts**
- **pre order: ausgabe links rechts**
- **post order: links rechts ausgabe**

vorgänger

- **nach oben, wenn es geht nach links + max** suche
- sonst oben (bis man einen vorgänger hat (links hat ja nur höhere sonst))
- wenn das auch nd geht gibts keinen

löschen

- bei **blatt löschen** und knoten sagen dis er weg ist
- **ein kind, dann preknoten auf kind zeigen lassen**
- bei **zwei kindern**, dann ist **max left or min right die neue wurzel**

Blattsuchbäume

werte liegen in den blättern und sind doppelt verkettet

AVL-Baum

- **Balance wert wird mit geführt umzu rotieren**
- wert **darf nur -1, 0 oder 1 sein (recht - links)**
- Diese **können bei änderungen berechnet werden** und müssen **nicht als komplette höhe verwaltet** werden
- (neue b werte könnenaus alten b werten ermitteltwerden)
- beim abbau der rekusion

dadurch Höhe O(log(N))

wirkt aufwänfig ist aber relativ einfach durch 4 zeiger neu setzen möglich
ggf. Log(N) mal

Rot-Schwarz-Baum

Balancierte Binärbäume, ein **zusätzliches Bit** pro Knoten **als Zusatzinformation**

im **schlimmsten fall** doppelt so **weit wie AVL** - dafür keine balance werte aktuallisieren

- Die **Wurzel und die Blätter (NIL's) sind schwarz**.
- knoten sind gefärbt
- **wenn knoten rot** -> **parent ist schwarz**
- **zu jedem blatt** kommen wir mit der **gleichen anzahl schwarzer knoten** (blackhight wird gespeichert)

2Log(N) zeig bar in dem die roten zu den schwarzen gezogen werden (2,3,4 baum) -,original höchstens doppelt so tief

1. **einfügen**
2. **rot färben**
3. **bedingung prüfen** bis **alles okey** (rekusiv)
    1. **wenn vorgänger und geschwister rot sind, dann umfärben**
    2. **wenn unterschiedlich rotation**
        1. bei der letzen wird die **reihe daruntrer rot und wurzel schwarz**
        2. allgemein sonst halt die gleiche farbe bzw die **nicht wurzel/parentknoten farbe**

einbisschen weniger speicher als AVL und weil einfach zu implementieren in cpp stdlib

B-Baum

- **keine Binärbäume!**
- hat eine **Ordnung m**, welche angibt, **wieviel nachfolger ein konten haben kann**
- alle **blätter** haben die **gleiche tiefe**
- **min m/2 viele kinder**
- **k Kinder** speichern **brauchen k-1 schlüsselwerte** im parent
- **zeiger zwischen 2 werten hat nur werte zwische den werten** - am anfang und am ende halt nur < bzw >
- Alle **Schlusselwerte** eines Knotens sind **aufsteigend sortiert**

wenn ein **blatt max wert erreicht** hat (m-1) dann **aufsplitten mit neuem zeiger** der durch mittleren wert geteilt wird.. **ist neues einfügen also rekusion** - ggf bekommt man eine neue Wurzel

B^+ **verkettet noch einmal die blätter** und da nur noch schlüssel speichern (z.b. vom kleineren baum der größte) und **keine werte** - sonst fehlen die ja => hoher log = wenig tiefen- darum immer ganze speicher seite als nnoten gerne auhc im wam halten

Splay-Baum

- laufzeit erhöhen durch oft zugreifen nach oben
- **nur im durchschnitt sehr schnell**
- **speicher effiziernter** weil **keine zusatz infos**
- immer anpassen bei zugriff
- wenn schoch bekannt direkt optimalen baum erstellen
- immer **wenn wir auf ein element zugreifen scheiben wir es richtung wurzel**
- **wenn das element nd existiert** dann der **vorgänger** oder nachfolger nehmen
- l,r,ll,rr,lr oder rl operationen bringen den knoten richtung wurzel - solange bis es die wurzel wurde!!!
- werte müssen sich halt weiterhin an suchbaum regeln halten

durch das alles ist einfügen, zugreifne und co izi in der wurzel möglich

worstcase O(m*log(N)) nach m operationen

Tries

(Aussprachen wie Trys um nicht mit trees zu verwechseln)

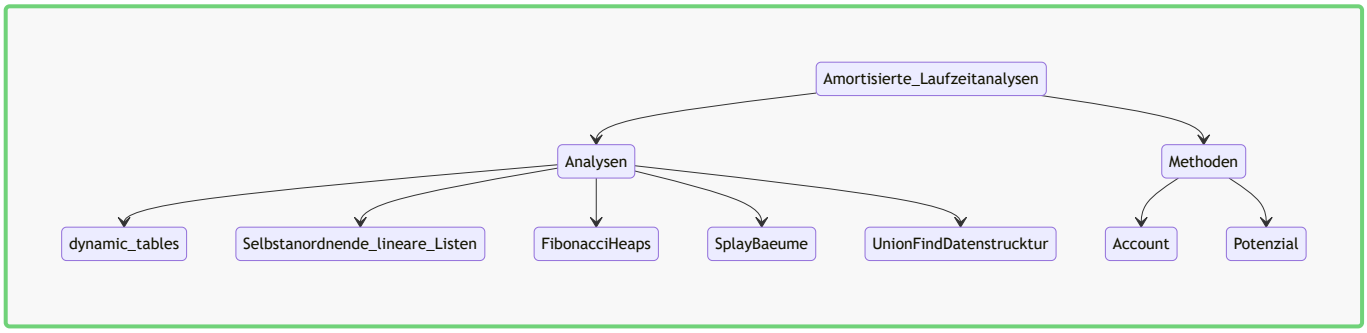
- Ein **Schlüssel**, der **aus einzelnen Zeichen** eines Alphabets besteht, wird in Teile zerlegt.
- Der **Baum wird anhand der Schlüsselteile aufgebaut**: Bits, Ziffern oder Zeichen eines Alphabets oder Zusammenfassung der Grundelemente wie Silben der Länge k.
- Eine **Suche im Baum nutzt nicht die totale Ordnung der Schlüssel**, sondern erfolgt durch Vergleich von Schlüsselteilen.
- **Jede unterschiedliche Folge** von **Teilschlusseln ergibt einen eigenen Suchweg** im Baum.
- **Alle Schlüssel** mit dem **gleichen Präfix** haben in der **Länge des Präfixes den gleichen Suchweg**. Digitale Suchbäume werden daher auch **Präfix-Bäume** genannt.
- Die **Höhe eines Tries** wird durch den **längsten abgespeicherten Schlüssel** bestimmt.

- Die **Gestalt des Baumes** hängt von der **Verteilung der Schlüssel ab**, nicht von der Reihenfolge ihrer Abspeicherung.
- Knoten, die nur **NULL-Zeiger** besitzen, **werden nicht angelegt**.
- **Schlechte Speicherplatzausnutzung** aufgrund **dünn besetzter Knoten** und vieler **Einweg-Verzweigungen** in der Nähe der Blätter.

um strings abspeichern zu können
Verzweigen anhand er buchstaben

Realisierung als **binärer Baum**: **Speichere in jedem Knoten nur Verweise auf zwei Knoten**: Einen zum **ersten Kind** und einen **zum Geschwister**. ABER **Zugriff auf Kind** mit bestimmten Namen **nicht mehr in konstanter** Zeit möglich. Daher wird zusätzlich Hashing angewendet:

Amortisierte Laufzeitanalyse



Account-Methode/Bankkonto-Paradigma

Kosten der i-ten Operation mit t_i und die **amortisierten Kosten der i-ten Operation** mit a_i bezeichnet. **Falls** $t_i < a_i$ **ist**, schreibe den **Uberschuss auf dem Konto gut**, falls $t_i > a_i$ ist, zahle die **Differenz vom Konto aus**. **Wichtig: Der Kontostand darf niemals negativ sein!**

Potenzial-Methode

Anstelle von **Kontoständen** weise der Datenstruktur eine **potenzielle** Energie zu

Φ_i **das Potenzial** nach Ausführung der **i-ten Operation**

Betrachte eine **Folge** von **n Operationen**. Ordne **jedem Bearbeitungszustand ein nicht-negatives Potenzial** zu, und **ordne jeder Operation amortisierte Kosten** zu.

Die **amortisierten Kosten** a_i der **i-ten Operation sind die tatsächlichen Kosten** t_i **plus der Differenz der Potenziale**

Stack mittels Array implementation

push worst case O(n) es **musste Speicherbereich für 2n Elemente allokiert** werden, dann mussten **n Elemente in den größeren Speicherbereich verschoben** werden, und anschließend wurde das **neue Element in den Stack** eingetragen

dynamic tables

wenn **array vollen - in ein doppelt so großes array** laden und dieses weiter verwenden

push kann allso im **woarcase O(N)** sein
bei N mal ausführen aber nicht $O(N^2)$!!
nur bei allen **2er potenzen sehr teuer sonst super billig** (O(1)) - genaue berechnung => 3*N = O(N)

Amotisiert (via arregat-methode): O(N)/N (durchschnitt)=> ein aufruf hat O(N) - **eine kann mal teuer sein aber meist nicht (nicht für echtzeitsysteme!)** nur laufzeit im ganzen (für alle operationen die selbe)

für unterschiedliche laufzeit möglichkeiten

Account-methode:

- jede **op kostet - "geraten"**
- alles was **gespart wird, wird gespeichert**
- wenns noch **länger dauert, dann zahle diff vom speicher**
- kontostand darf **niemals negativ** werden (auch nicht nur kurz? - ja auch nd nur kurz)
  eher so als beweis ig

potenzial-methode:

- **Energie einer datenstrucktur zuweisen**
- jeder **Zustand bekommt potenzial (nicht negativ)**
- potentzial nach op wird geraten / die funktion
- **Amortisierte kosten** für **jede OP werden berechnet**
  damit es eine obereschranke ist muss **p von N - p von 0 positiv sein**
- mit der funktion können die amortisierten konsten für jeden schritt berehcnet werden
- (ist das O()) worst case wird da betrachted

pop immer **halbieren wenn kleiner als 1/4**, dann kann man die **kosten fürs nächste verdoppeln wieder reinholen**
die p-funktion gibt an wie weit man von 1/2 weg ist

Selbstanordnende lineare Listen

was **oft gebraucht** wird, soll **nach vorne**

> 80% der zugriffe ist auf 20% der daten

möglichkeiten:

1. **nach ganzvorne bringen (MF)**
2. **ein nach vorne bringen (T)**
3. **häufigkeiten mitführen (FC)**

unterscheidliche zugriffs reihenfolge ist nun auch wichtig

Algo A: sucht eine element und kann es verschieben kosten: **pos k suche kostet k**, richtung anfang an die liste ist kostenfrei, nach hinten kostet es sie richtung

letzteres nicht vorhanden in den beispielen
anzahl kostenfreier vertauschen k-1 und der kosten tauschs N-k

Potenzialfunktion - inversionen zwischen 2 listen werden gezählt

> Invesion wenn sich die reihenfolge eines paares ändert

=> MF MAX 2x so schlecht wie das perfekte A
bei mf werden alle inversionen und nicht inversionen (von der anzahl) vertauscht

> also kann man seinen algo mit einem nicht entdeckten optimalen Algo vrrgleichtn mit laufzeitanalysen

Fibonacci-Heaps

armotisierte kosten durch **potenzialn rechnung**
t_i + p_i - P_{i-1}

bei M operationen davon N vielen Inserts:
N +(M-N)*Log(N), weil N konstant => O(M*Log(N))

- Höhe der Bäume in den Fibonacci-Heaps
- sei: ein Knoten x mit deg(x) = k ist die Wurzel eines Baums mit mindestens F_k Knoten
- Seien $y_1, \ldots, y_k$ die Kinder von x in der zeitlichen Reihenfolge, in der sie an x angehängt wurden. Dann gilt: deg(y_1) ≥ 0 und deg(y_i) ≥ i − 2 für i = 2, 3, . . . , k

Induktionsanfang: deg(y_1) ≥ 0 ist klar. **Induktionsschluss für i ≥ 2**:

- Als y_i **an x angehängt** wurde, waren $y_1, \ldots, y_{i-1}$ Kinder von x, also gilt **deg(x) ≥ i − 1**.
- Da y_i an x angehängt wird, gilt: **deg(x) = deg(y_i),** also gilt **deg(y_i) ≥ i − 1**.
- Seither hat Knoten y_i **höchstens ein Kind verloren**, sonst hätten wir y_i von x abgetrennt, also gilt: **deg(y_i) ≥ i − 2**

Die Potenzial-Funktion zu einem Fibonacci-Heap H ist definiert als **Φ(H) = b(H) + 2 · m(H)**

Dabei bezeichnet **b(H) die Anzahl der Bäume in der Wurzelliste und m(H) die Anzahl der markierten Knoten**

Splay-Bäume

kürzt sich gut weg

z.b. auch, dass nur 2 ränge gändert werden und vorher 1 gleich nacher 2 ist
und sich die ränge der unterne bäume nicht ändern

r(x) ist der rang von x (Log(N)) T ist der gesamte baum und x der betrachtete **doppelt rotation 3(r(T)-r(x))** **einfache rotation 3(r(T)-r(x)+1)** <- also das ist das max => bei M operationen davon N vielen Inserts: O(m*log(N))

Union-Find-Datenstrucktur

- Pfad **komprimierung** um laufzeit zusparen

- dazu **knoten direkt unter die wuzel** hängen um von x direkt zur wurzel zu kommen (rekusiv)
  knoten in disjunkte Mengen
  wie ein baum **aber von blättern zur wuzel** (repräsentant(leader) der menge)
  vorgänger und rand (zum verschmelzen) in jeder node
  beim verschmelzen: **den flacheren baum unter den größeren** hängen (wenn gleich hoch wächst der baum um eine ebende)
- **ränge** können **nur vom repräsentanten geändert** werden - alle anderen werden irgendwann falsch sein => gibt kein delete
- log*-funktion wird hier genommen statt der akaman aus den büchern
- gibt den wert an wie oft mal log nutzen muss bis er < 2 ist *log(2^2^2^2^2^2) = log*(2^65536) = 5**
- bei 2^400 atomen im all, wird das woohl nie errecht
- also niemals mehr erreichbar .. in etwa konstant..
- invert wäre es nun die **tower-funktion (wächst EXTREM stell)**
- zur einteilung in ebende mit **rang in die funktion tower schmeissen**
- auf geteilt wird dann in *log(N)-Blöcke** .. also max 5 mit den werten wird amotisiert (mit geldwerten aber der arregat nicht bankkonto funkiton) **log^{(i)}(n) := n falls i = 0 log(log(i−1)(n)) für i ≥ 1**

Leader, kind(des leaders), Path und block(wechsel)
2m+m*log*(N) (<- gemeint log*()) => O(m*log*(N)) und log*(N) max 5 => O(M)

> Akaman wächst extem schnell, die inverse extrem langsam - wird sonst genommen anstat log*()

- Laufzeit-Abschätzung für die union- und die find-Operation bei der Union-Find-Datenstruktur
- Zur Laufzeitabschätzung ordnen wir den Knoten Blöcke zu. Ein Knoten x wird dem Block b zugeordnet, wenn gilt: **tower(b − 1) < rank(x) ≤ tower(b)**
- Dabei definieren wir **tower** als **Umkehrfunktion** der obigen *log-Funktion.**
- **Find: Es wird ein Weg x1, x2, x3, . . . , xl** durchlaufen, **wobei x1 der Parameter der find-Operation** ist, und **xl der Repräsentant** des Baums bzw. der Menge.
- **Kosten von find(x_1) entsprechen der Länge l** des Weges.
- Jeder Knoten auf dem Weg zahlt jeweils 1$ in eines von mehreren Konten.
  - **leader account: darin zahlen Repräsentanten ein**
  - **child account**: darin **zahlen Kinder der Repräsentanten ein**
  - **block accounts**: darin zahlen **Knoten** x_k ein, **die in einem anderen Block sind als deren Vorgänger x_{k+1}**
  - **path account**: darin zahlen **alle** anderen Knoten ein
- Der Gesamtbetrag aus allen Konten gibt dann die Summe der Laufzeiten aller Operationen an

Jede find-Operation zahlt

- **1$ in das Repräsentanten-Konto (leader account)**,
- **höchstens 1$ in das Kinder-Konto (child account)**, und
- *höchstens 1$ in jedes Block-Konto (block account), wovon es nur log(n) viele gibt.**

Algos für aktuelle Hardware

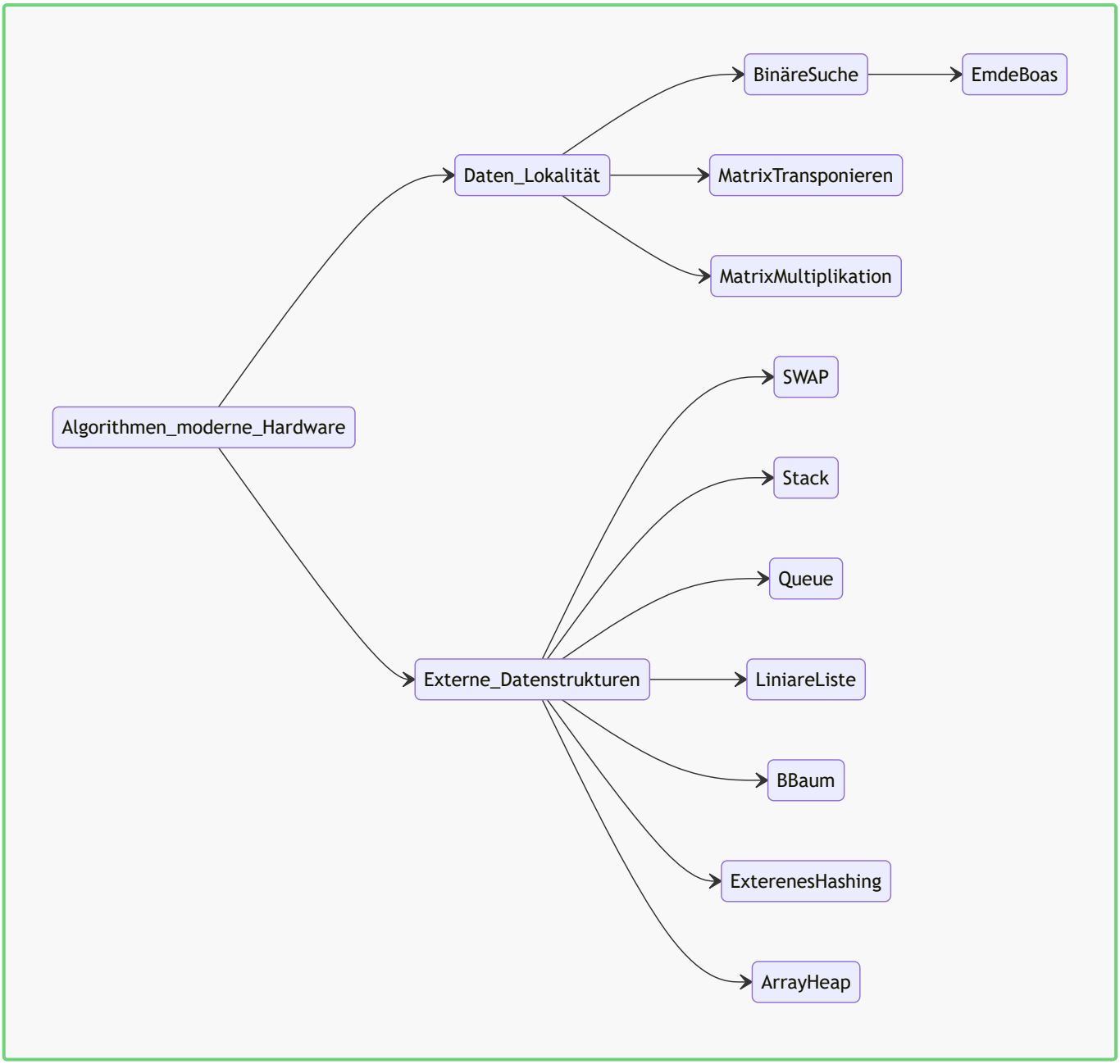
Speichergeschwindigkeit wächst langsamer als die Prozessorgeschwindigkeit (durch parallele Kerne)
caching und prefetching können daher wichtig um die CPUs ausnutzen zu können

liniares lesen ist deutlich schneller als random

Model wäre bei den ganzen arten an CPUs zu komplex
darum wird mit einem externen speichermodel optimiert, welches nur 2 ebenen hat:

- **CPU mit Internenspeicher M**
- **Größe B für IO-Operation auf**
- externer Speicher
- ggf. erweiterbar mit mehreren festplatten kann auch als Cache + RAM verstanden werden

Algos können nur um einen Konstantenfaktor verbessert werden



binäre suche

es wird **viel im array gesprungen**, daher im array so speichern, dass nach zugriffen sortiert wird
also die **mitte nach vorne**, dann **2i+1 kleiner als 2i+3 das größere**
noch **besser nach van Emde Boas**:

- **suchbaum erstellen**
- **in häfte teilen**
- **erst die erste hälfte**, dann alle **teil bäume die entstanden in der unteren hälfte**
- das wird **rekusiv gemacht für alle bäume**, **bis jeder block in den cache** passt (nicht sehr oft da immer Wurzel(n)-reduzierung) => von Log_2(N) auf Log_B(N)

Transponieren einer Matrix

Zeile kann immer in Blöcken gelesen werden, Spalten nicht
darum $O(N^2)$

darum **BxB große blöcke machen** (sofern 2 davon in den cache passen) und **diese verschieben** $O(N^2/B)$ - eig x2 also B/2 beschleunigung

Matrix Multiplikation

ijk - $O(N^3)$ ikj - $O(N^3/B)$ - eig x2 also B/2 beschleunigung vorteiele: wir gehen in den **inneren nur rows** durch
zudem kann noch der wert der **2. schleife in einen Register gepackt** - dadruch muss nicht mal im cache geschaut werden dafür

Noch besser:
wieder in blöcken arbeiten (da eine zeile größer sein kann als eine Cache-Zeile)
dabei muss ggf beim ende **auf gepasst** werden, **falls der block zu groß ist** zudem wieder in ikj und register
hier fällt dann noch die 2 weg und es ist wirklich $O(N^3/B)$

exterene Datenstruckturen

Daten im Hauptspeicher, welche es ergmöglichen daten auf der festplatte zu finden

- **swap**
  - nicht für ein bestimmtes problem - sehr allgemein
- **stack**
  - **intern 2 blöcke**
    - es werden ja nur **die oberen genutzt**
    - zudem kann **beim anfangen des 2. blockes der nächste nachgeladen werden** am besten aber erst bei nur noch 1/4 speichern/laden
  - **extern den rest**
- **queue**
  - **intern 2 blöcke - für rausnehmen und einfügen je einen**
  - **rest extern**
  - ggf wenn leer schwierig - fallunterscheidungen
- **Liniare-liste**
  - **aufeinanderfolgende** knoten in **einen block**
    - naiv nur sinnvoll beim suchen!
    - **ein einfügen würde jeden block verschieben**
    - immer schauen, das immer nur b/2 gefüllt werden .. wenn voll .. halbieren und spliten
    - NOCH BESSER: **jeder block muss 2/3B** viele elemente haben
      - wenn **voll** einfach **in die nachbar** blöcke schieben
      - wenn **nachbarn auch voll** sind, dann **spliten**
      - **beim löschen zusammen fassen wenn sie zusammen < 2/3 sind**
      - ”seltsame“ Regel da wenn Nachbarblöcken zu viele Elemente enthalten wieder splitten etc.
      - O(1+n/B)
      - mit paaren werden die cache misses im worstcase reduziert
- **B-baum**
  - **jeder knoten ein block (Zeiger+schlüsselwerte zum trennen)**
  - **sehr wenig IO operartionen** um eine sehr große menge an Daten ansprechbar

    - O(log_b(N))
- **exterenes Hashing**
    - **direkten zugriff** durch **berechnung eines arrayindexes**
    - O(1)
    - **großer Wertebereich mit wenig elementen** wäre sehr **verschwenderisch**
    - **viel viel kleineres array nehmen** - ca so groß wie elemente die wir speichern wollen
    - funktion bildet **von universum auf arraygroße** ab
        - **kollisionen** durch synonyme **möglich**
            - **wuzel(pi*n/2) ist die anzahl der kollisionen**
            - es gibt immer eine menge K die viele kollisionen hat
            - **keine vermeidung, daher umgang**
                - im **array ist ein pointer** auf eine **kette/liste** (index natürlich mitschrieben)
                    - mit **belegungsfakotr n/m**: in der regel ist **N element von O(M) daher O(1)** => laufzeit 1+1 oder 1+1/2
                    - ggf **O(N) wenn array voll ist** muss neu gehasht werden, nach einer erweiterung
                    - perfekt für externe datenstruckturen (siehe liniare ketten)
                        - andere struckturen auch okey - hashbäume - bäume etc
                - **oder nächster freier platz** bzw ausweichposition
                    - **berechnen durch permutation** des arrays MAX m versuche
                        - z.b. **liniar hash+i mod m**
                            - **primäre häufung** (nachbarschlüsselwerte)
                            - **um sovoller um so länger dauerts** => kette ab halbvoll besser (daher muss man dann schon vergrößern)
                        - **oder quadratisch hash** + -1^i * up(i/2)^2 mod m
                            - **sekundäre häufung** (gleicher schlüsselwert)
                        - **der schlüssel muss auch in jeden Versuch einfließen**
                            - **beste** wäre für **jeden schlüssel eine zufalles permutation** haben um zu testen wie gut es werden kann
                                - **natürlich nicht sinnvoll** aber zum überprüfen
                                - **1/1-belegungsfaktor bzw 1/belegungfaktor ln (1/1-belegungsfaktor)**
                            - **double hashing** (beschte)
                                - **ausweich position durch hashfunktion + einer andere hashfunktion * i .. mod M**
                    - **such index muss gespeichert** werden
                    - **gelöschte werte** dürfen nur **als gelöscht makiert** werden
                        - beim **reinschreiben darf es überschrieben werden**
            - Eignet sich **garnicht** für externe speicherung **weil man die ganz zeit rumspringt**
        - beispiele
            - k%m (am besten primzahl)
            - (k*a mod 2^w) / 2^w-r
- Array-Heap / Priowarteschlangen
    - zu komplex ;D
    - heaps mit k nachfolgern
    - **wohl extrem langsam z.b. geohashing kann bei dijkstra viel besser sein**

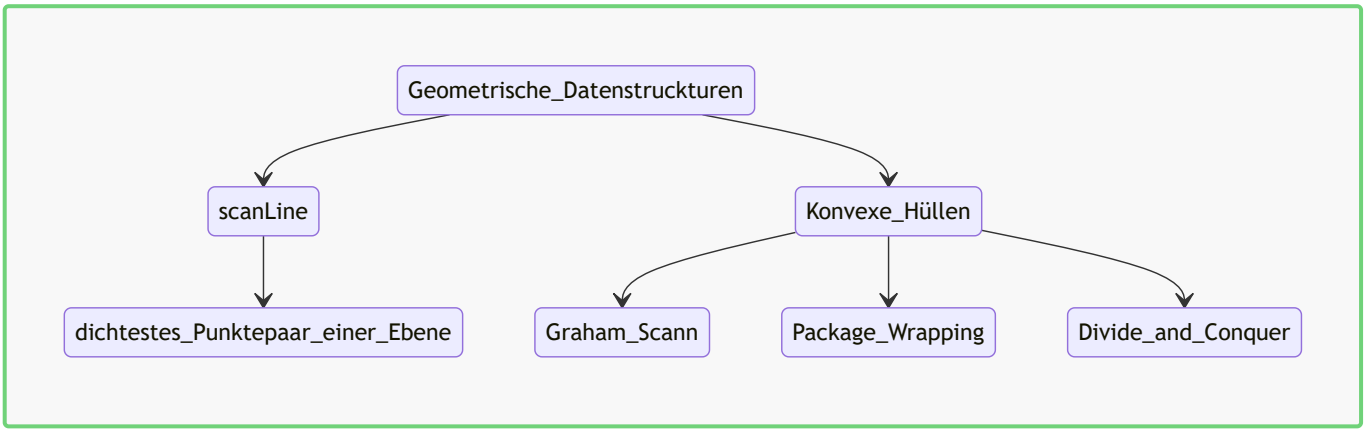
> sowas ist in der STxxL implementert
> Geohashing ähnliche Werte nah abbilden (hashen ohne diffusion?)

Geometrische Datenstruckturen

um bereichsanfrgaen auf koordinaten zu machen

konvex wenn zwei punkte IN einer Menge verbindbar sind

polygon darfauf jeden punkt nur kanten zeigen (Polygon zug)



scan-line

linie über ein koordinaten system (szene)

> Ziel: mehr dim prob in ein dim umwandeln

- unterteilt in **tote, aktive und inaktive objekte**
- **bewegung** über **diskret**e schritte
- **reduziert um eine diminsion**
    - (ich denke rekusiv auf rufbar)
- **speichert anfangs und endpunkte einer achse**
- arbeit mit **suchbaum**
- durch **endpunkte links eines elemnts**: **sortiert in eine suchbaum** packen (aktiv)
- **oberer und unternachbar können gesehen werden** (**links und rechts können sich nichtsehen weil disjunkte y koordinate**, sonst halt nacheinander)
- **endpunkt rechts**: **aus suchbaum raus nehmen (tot)** dann **oben und unterernachbar auch sichbar** O(N*log(N))
- **paare könntne doppelt vorkommen** (ggf speichern)
- **gegenseitig sichbar ist ein planaer graph darum nur 3n-6 sichtbare elemente**

iso-orientiert:

- **ähnlich** kann man bei horizontale und vertikalten linien eine linien art durch gehen und **alle schnitt punkte (dies mal VON BIS) durch gehen**
- natürlich werden **die vertikalen linien durch gegangen** durch den scan-line - **horizonte werden als blattsuchbaum (balanciert) gespecihert**
- O(N*Log(N) + k) mit k schnittpunkten also bis zu $O(N^2)$
- platzbedarf O(N)

allgemein:

- **linien beliebig im raum**
- **datenstrucktur swapt bei schnittpunkten**
- **nachbarn werden auf schnittpunkte geprüft**
  - sollte es einen geben, **dann beim erreich:ausgabe+umsortierung und nachbarn neu prüfen**
  - (vorraussetung: in jedem punkt max 2 linien)
- O((n+k)*log(n))
- kann besser werden zu O(N*log(N)+k), k kann halt n^2 werden: immer nur links den 1. schnittpunkt aufnehmen und beim wechsel den nächsten
- mehrere linien an einem punkt auch möglichj - nacheiander an einer x koordinate

dichtestes Punktepaar einer Ebene

ähnlich zu dichtestes paar einer Zahlenfolge
=> **sortieren und nachbarn betrachten** => wie geometrisch?

Wie Scan-Line, aber mit mehreren linien

- **punkte nach x aufteigend sortieren**
- **durchgehen und bei jedem punkt halten**
- **aktive=alle im bereich**
- der **bereich ist so groß wie der aktuelle MIN wert**
- **beim ändern natürlich auch aktuellisieren**, nicht nur bei der eigendlichen scan-line
- **wenn zwei punkte aufeinanter liegen, kann direkt terminiert werden**
- **x + min <= x2 zum prüfen ob der nächste punkt noch drin ist** oder nicht
- **min kann dann auch auf der Y achse betrachtet werden** (eig sogar halbkreis - aber zu schwer)
  - => **dazu werden die Aktiven knoten, nach Y- werten sortiert via balancierter SUchbaum** (genial)
- **Platz O(N)**
- **Zeit O(N*Log(N) + sum bis n von k) => k sind bereichts anfragen**
  - => **max $O(N^2)$** wenn es immer i elemente sind
- **weil in einem rechtreck MIN*2MIN max 10 punkte drin sind => nur (O(N)*Log(N))**
  - => **max m/2 distanz sonst schon ein gefunden** => max 10 pkt passen dadurch
  - => **eig sogar max 6**

Konvexe Hüllen

kleinste konvexe hülle einer Menge
like gummiband um nägel

> zeit ersparniss für PC: wenn nicht sichbar oder keine kollisionen dann muss es auch nicht detaliert berechnet werden

Graham Scann

linksdrehungg gegen uhrzeigersinn durch 3 punkte und **rechtsdrehung im Uhrzeigersinn**:
berechnung durch steigungen
oder detiminante > 0 => linksdrehung
punkte nummieren von punkt 1 **sortiert nach winkel von rechts nach links** immer 3 punkte betrachten von **kleinsert Y an fangen** bei **rechts drehung den mittleren Punkt raus werfen** sonst durchlaufen bis urspungspunkt
SONDERFALL bei graden von 3 punkten (eig aber egal)
O(2N), weil wir immer einen punkt raus werfen
verkettet liste als speicher
dann durch sortieren am anfang O(N*log(N))
platz O(N)

Package Wrapping

kleinster y, bei **mehreren kleiner x && y** dann immer den **kleinsten wunkel (rechtsrum)**
beim absteigen dann (**nach dem höchsten punkt** erreicht wurde) dann **inventierter winkel**
eig muss man die iwie nd wiirklich berechnen Laufzeit O(|CH|*N) => wenn alle punkte auf der konvexen hülle liegen, dann $O(N^2)$ Platz O(N)

verbesserung druch interiiot point elimination:

> Vor bereitung für vorbereitung

aus **jeder ecke den ersten punkt** wählen => für alle 4 punkte gibt es ein möglichkeit den max oder min zu bekommen durch addition oder subtraktion von x und y zu bekommen
alle die im 4eck zwischen denen sind können weg
=> erkennung via links drehungen durch FÜR EINMAL RUM in O(N) verarbeitet vorteil von **NICHTS bis GANZ VIEL**
im AVG: Reduktion von sqrt(N) (für berechnung gleichverteilung annehmen können)

Divide and Conquer

aufteil in 2 hüllen (rekusiv)
dann **zusammen fassen**

- **random punkt von links** wählen
  - **PUNKT IM POLIGON**: **strahl in eine richtung**, **wenn ungrade schnitpunkte dann DRIN**
    - einfach **mit allen polygonkanten testen**
    - uff dachte halt winkel kleiner als next für jeden punkt
    - - **sonderfällt** durch **ecke** oder **auf linie**
  - wenn in beiden teilen:
    - beide **listen mit merge sort verbinden**
    - **angefangen vom winkel des punktes**

    - **wenn nur in links**, dann **einen "keil" bilden**
        - die **weg lassen sonst mergen**
        - **damit die liste sortiert bleibt für Graham scan** uff
            - **brauch** einen **punkt AUF oder IN** der **hülle**
        - **angefangen vom winkel des punkte**s
- dann **konvexe hülle auf der neuen liste**

T(N) = 2T(n/2)+O(N) => **sortieren wird nicht benötigt | liniar** O(N*log(N))$$