Algorithmen I – Zusammenfassung

Inhalt

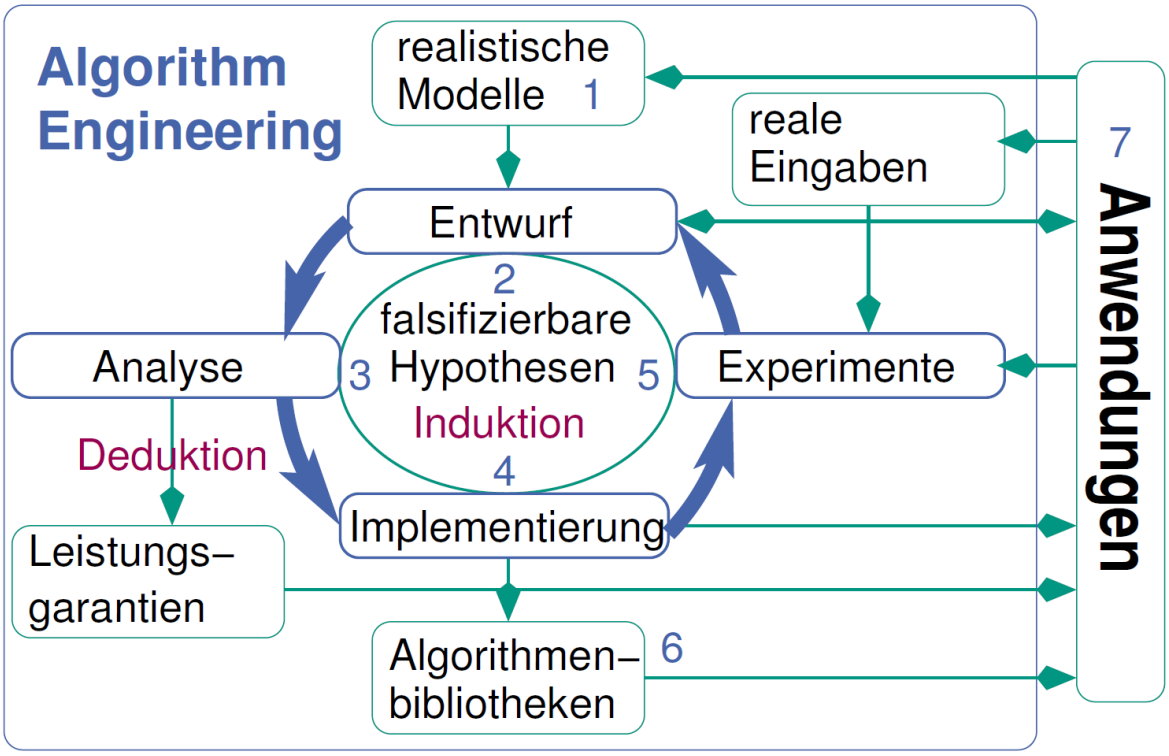
Analyseverfahren 4
Algorithmik als „Algorithm Engineering“ 4
Schleifeninvariante 4
Asymptotisches Verhalten 4
$o(n)$ 4
$\omega(n)$ 4
$\Theta(n)$ 4
$O(n)$ 4
$\Omega(n)$ 4
Master-Theorem 5
Amortisierte Laufzeitanalyse 5
Karazuba-Ofman-Multiplikation 5
Algorithmus 5
$recMult(a, b)$ 5
Laufzeit 5
Datenstrukturen 6
Verkettete Listen 6
Opterationen 6
Dummy header 6
Sentinel (Wächter) 6
Felder (Arrays) 6
Unbound Array 7
Cyclic Array 7
Stack and Queue 7
Listen vs Felder 7
Hotlist 8
$Lookup(Key\ k): Data$ 8
$Insert(Key\ k, Data\ d)$ 8
$Delete(Key\ k)$ 9
Prioritätslisten 9
Binäre Heaps 9
Suchbäume 9
(a,b)-Bäume 9

Union-Find Datenstruktur 9
Operationen 9
Hashing 9
Verketten 9
Lineare Suche 10
Sortiere 10
Insersort 10
Algorithmus 10
Mergesort 11
Aufschrieb 11
Algorithmus 11
Quicksort 12
Aufschrieb 12
Algorithmus 12
Quickselect 13
Bucketsort 13
Algorithmus 13
LSD-Radixsort 13
Algorithmus 13
Heapsort 14
Algorithmus 14
Sortieralgorithmen-Übersicht 14
Graphen 14
Eigenschaften 14
Direkter Azyklischer Graph (DAG) 14
Bipartitheit 14
Eulerscher Kreis 14
Hamiltonscher Kreis 14
Graphenrepräsentation 15
Kantenfolge 15
Adjazenzmatrizen 15
Adjazensfelder 15
Adjazenzlisten 16
Graphentraversierung 16
Breitensuche 16
Tiefensuche 16

Kürzeste Wege ... 17
Dijkstra -Algorithmus ... 17
Bellman-Ford-Algorithmus ... 17
Von überall nach überall ... 17
Minimale Spannbäume ... 17
Definition ... 17
Jarník-Prim-Algorithmus ... 18
Kruskal Algorithmus ... 18
Jarník-Prim vs. Kruskal ... 18
Vertex-Cover ... 18
Optimierung ... 18
Linear Programming ... 18
Dynamische Programmierung ... 19
Voraussetzung ... 19
Rucksackproblem ... 19
Systematische Suche ... 19
Algorithmus ... 19
Beispiel ... 20

Analyseverfahren

Algorithmik als „Algorithm Engineering“



Schleifeninvariante

Eine Bedingung die Vor- und nach jedem Schleifendurchlauf gilt.

Asymptotisches Verhalten

$o(n)$

$$f(n) \in o(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$\omega(n)$

$$f(n) \in \omega(g(n)) \Leftrightarrow \limsup_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

$\Theta(n)$

$$f(n) \in \Theta(g(n)) \Leftrightarrow 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$$

$O(n)$

$$f(n) \in O(g(n)) \Leftrightarrow 0 \leq \limsup_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$$

$\Omega(n)$

$$f(n) \in \Omega(g(n)) \Leftrightarrow 0 < \liminf_{n\to\infty} \frac{f(n)}{g(n)} \leq \infty$$

Master-Theorem

Ist T(n) definiert als:

$$T(n) = \begin{cases} d * T\left(\left\lceil \frac{n}{b} \right\rceil\right) + c * n & \text{für } n = 1 \\ sonst \end{cases}$$

So gilt:

$$T(n) \in \begin{cases} \Theta(n) & d < b \\ \Theta(n \log n) & d = b \\ \Theta(n^{\log_b(a)}) & d > b \end{cases}$$

Amortisierte Laufzeitanalyse

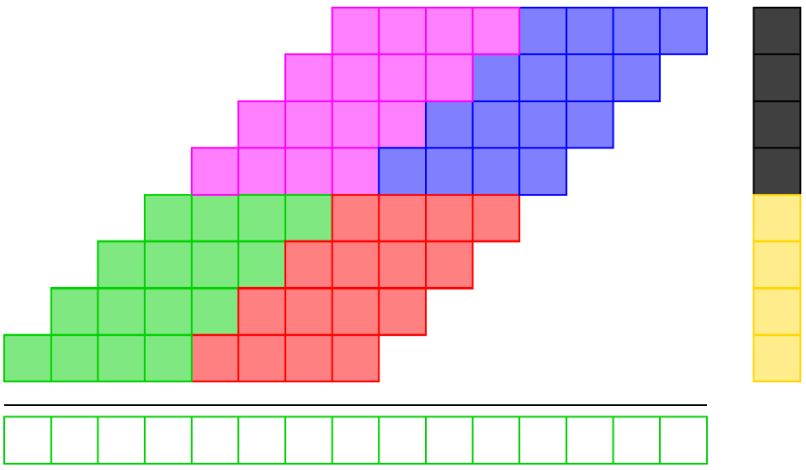
Methode um die Worstcase-Laufzeit besser abzuschätzen (Kosten von selten auftretenden teuren Operationen werden auf häufige billige Operationen verteilt).

Karazuba-Ofman-Multiplikation

Algorithmus

$recMult(a, b)$

1. Wenn n=1
   a. Gib a*b zurück
2. Sonst
   a. $a = a_1 * B^k + a_0$
   b. $b = b_1 * B^k + b_0$
   c. Gib zurück
   $recMult(a_1, b_1) * B^{2k} + \left(recMult(a_0, b_1) + recMult(a_1, b_0)\right) * B^k + recMult(a_0, b_0)$



Laufzeit

$$O(n^{\log_2 3})$$

Datenstrukturen

Verkettete Listen

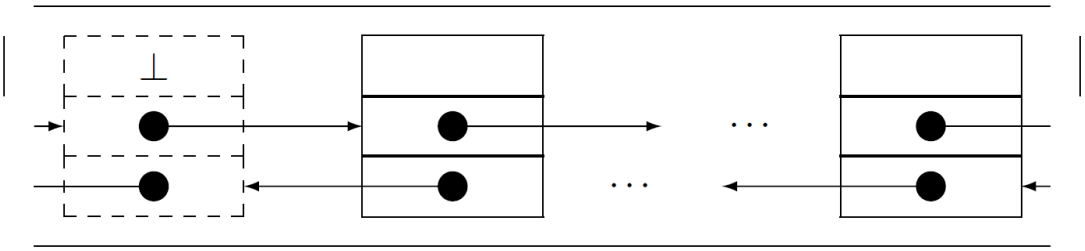
Alle Listenoperationen zur Listenmanipulation können mit der Benutzung von splice durchgeführt werden.

Opterationen

get	Gibt Element mit dem Index i zurück
size	Gibt die Anzahl der gespeicherten Elemente zurück
splice	Löst eine Teilfolge aus einer Datenstruktur und hängt sie in eine andere des gleichen Typs
popFront	Löscht das vorderste Element der Datenstruktur und gibt dieses zurück
popBack	Löscht das hinterste Element der Datenstruktur und gibt dieses zurück
remove	Entfernt ein Element mit dem Index i
insertBefore	Hängt ein Element vor das Element mit dem Index i
insertAfter	Hängt ein Element hinter das Element mit dem Index i
pushFront	Fügt ein Elemet am Anfang der Datenstruktur ein
pushBack	Fügt ein Element am Ende der Datenstruktur ein
concat	Hängt zwei Datenstrukturen des Gleichen Typs aneinander an
makeEmpty	Leert die Datenstruktur
findNext	Finde das nächste Element e

Dummy header

Listenelement, dass keine Informationen enthält aber Zeiger. Schließt den Kreislauf einer Liste und vermeidet das Behandeln von Null-Zeigern.



Sentinel (Wächter)

Bei findNext Operation wird in das Datenfeld des „dummy headers“ das gesuchte Datum geschrieben somit Terminiert die Ausführung automatisch bei Erreichen des „dummy headers“ Es muss nun nur einmal nachträglich entschieden werden, ob es sich um den „dummy header“ handelt.

Felder (Arrays)

Ein Bereich im Speicher (fester Größe) der von dem Programm reserviert wurde.

Unbound Array

Ist ein unbeschränktes Array. Das heißt, sobald das Array voll ist, müssen die Daten in ein neues größeres Array kopiert werden (i.d.R. doppelt so großes). Wird das Array zu leer so werden die Daten in ein kleiner kopiert (meist wird die Größe halbiert und das geschieht wenn das Ursprungsarray nur noch zu einem Viertel voll ist).

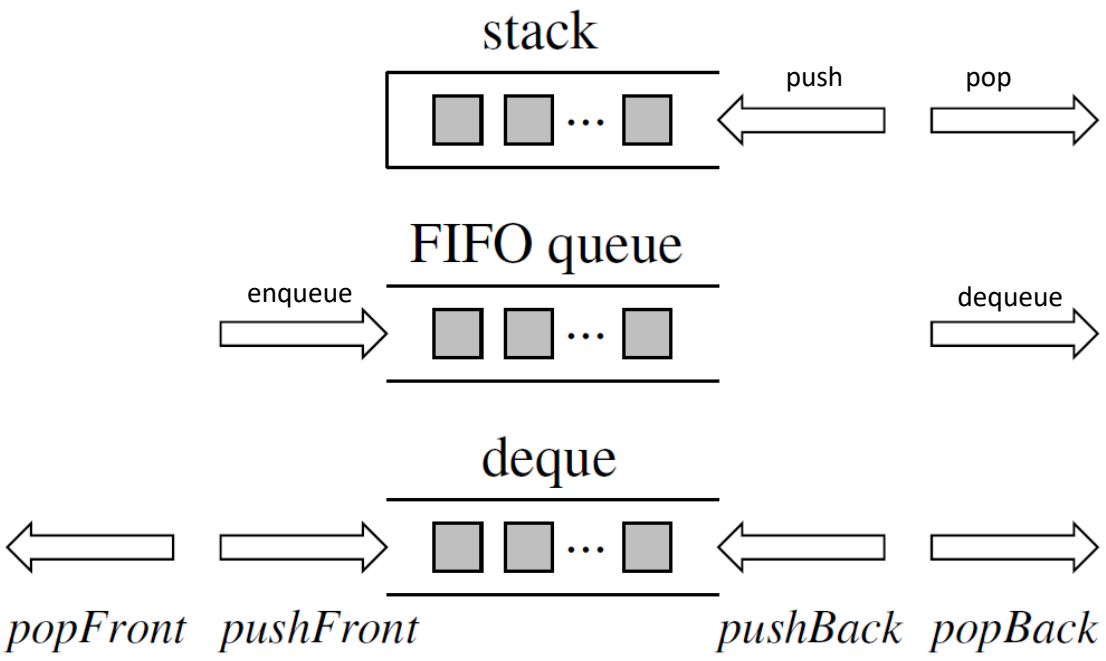
Amortisiert sind die Vergrößerung- und Verkleinerungsoperationen von ihrer Laufzeit Konstant.

Cyclic Array

Ein gebundenes Array, welches durch rechentricks wie ein Kreis erscheint, es kann also auch am Anfang etwas eingefügt und gelöscht werden, ohne Verschiebe- oder Kopieroperationen auszulösen.

Stack and Queue

Nur eine Schnittstelle die gewissen Operationen erlaubt => weniger Fehleranfällig



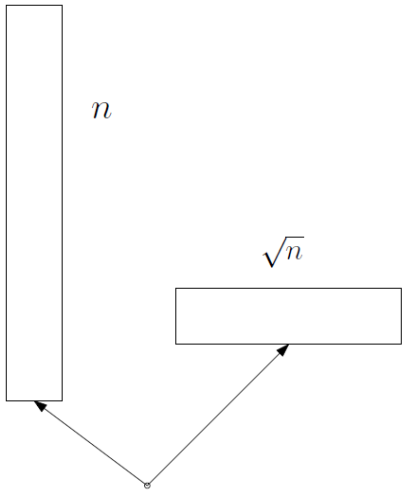
Listen vs Felder

Listen	Felder
Flexibel	Beliebiger Zugriff
Remove, splice, …	Einfach
Kein Verschnitt	Kein Zeigeroverhead
	Cache-effizient

Operation	List	SList	UArray	CArray	explanation '*'
$[\cdot]$	n	n	1	1	
$\lvert\cdot\rvert$	1^*	1^*	1	1	not with inter-list splice
first	1	1	1	1	
last	1	1	1	1	
insert	1	1^*	n	n	insertAfter only
remove	1	1^*	n	n	removeAfter only
pushBack	1	1	1^*	1^*	amortized
pushFront	1	1	n	1^*	amortized
popBack	1	n	1^*	1^*	amortized
popFront	1	1	n	1^*	amortized
concat	1	1	n	n	
splice	1	1	n	n	
findNext,...	n	n	n^*	n^*	cache-efficient

Hotlist

Ist ein Array bei dem der vordere Teil ein sortiertes Array ist (der große n). Der hintere Teil ist ein Unsortiertes Array (Hotlist) der Größe $\sqrt{n}$.



$Lookup(Key\ k): Data$

1. Durchsuche geordnetes Array mit binärer Suche
2. Durchsuche Hotlist komplett

Laufzeit

$$O(\log n + \sqrt{n}) = O(\sqrt{n})$$

$Insert(Key\ k, Data\ d)$

1. Ist in der Hotlist freier Platz so wird an der nächsten freien Position eingefügt.
2. Ist in der Hotlist kein Platz dann sortiere die Hotlist und merge (wie bei Mergesort) Hotlist und sortierte Liste. k ist das erste Element in der neuen Hotlist.

Laufzeit

$$O\left(\sqrt{n}^2 + n\right) = O(n)$$

$Delete(Key\ k)$

1. Wenn weniger als $O(\sqrt{n})$ Löschoperationen
    a. Suche k und setze „valid“-Bit auf 0
    b. $Laufzeit$: $O(\sqrt{n} + \log n) = O(n)$
2. Bei mehr als $O(\sqrt{n})$ Löschoperationen
    a. Reorganisation da es Zusammenführung gab
    b. $Laufzeit$: $O(n)$

Laufzeit

Prioritätslisten

Binäre Heaps

Ein Baum mit der Eigenschaft: Für alle Knoten gilt $parent(v) <= v$

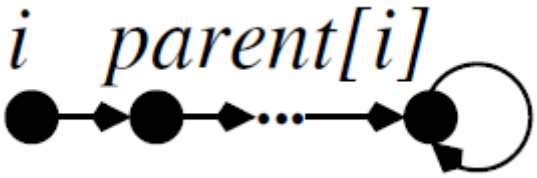
Suchbäume

(a,b)-Bäume

Der Knotengrad ist zwischen a und b groß. Höhe ist somit $log_a n$

Union-Find Datenstruktur

Array, bei dem jeder Knoten auf einen anderen Knoten in der Datenstruktur zeigt. Der Repräsentant eines Waldes zeigt immer auf sich selbst und hängt am Ende dieser Kette.



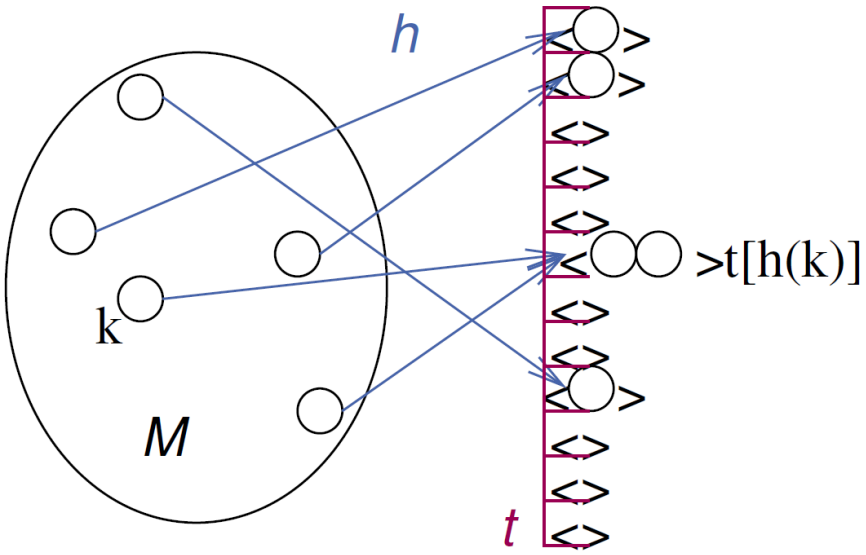
Operationen

- find() – findet Repräsentanten der Partition
- union() – verbindet zwei Partitionen

Hashing

Verketten

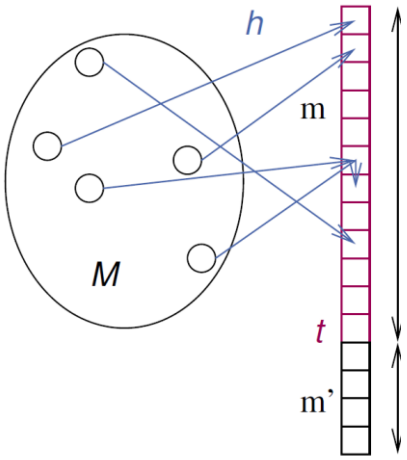
Ein Array aus Listen dient zur Speicherung. Bei Kollisionen wächst somit einfach die Liste und die ***referentielle Integrität*** wird gewahrt.



Lineare Suche

Die Elemente werden direkt in einem Array gespeichert.

Tritt eine Kollision auf, so wird das Array bis zum nächsten Freien Platz durchlaufen (Keine ***referentielle Integrität*** und keine ***Leistungsgarantien***)



Sortiere

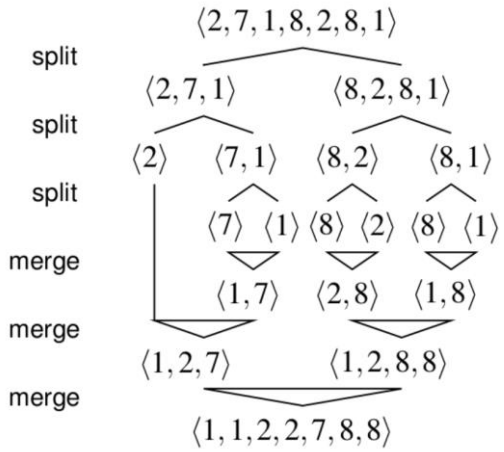
Insersort

Algorithmus

1. Nehme ein Element und füge es an der richtigen Stelle in den sortierten Teil des Arrays ein (Elemente müssen natürlich aufgerückt werden).
2. Wiederhole schritt 1 bis der unsortierte Teil des Arrays leer ist.

Mergesort

Aufschrieb



Algorithmus

Split (list : $< e_1, \cdots, e_n >$)

1. Ist $|list| = 1$
    a. Dann gib $< e_1 >$ zurück
    b. Sonst gib $Merge(Split\left(< e_1, \cdots, e_{\frac{n}{2}\downarrow} > \right), Split\left(< e_{\frac{n}{2}\downarrow+1}, \cdots, e_n > \right))$ zurück

Merge ($firstList$: $< e_1, \cdots, e_n >$, $secondList$: $< e_1, \cdots, e_n >$)

1. Solange in firstList und $second$List nicht leer
    a. Vergleiche Vorderstes Element aus firstList und $second$List
    b. Lösche niedrigeres von beiden aus der jeweiligen Liste und füge es hinten an die sortierte Liste
2. Füge die nicht-leere Liste an die sortierte Liste hinten an.
3. Gib die sortierte Liste zurück

Quicksort

Aufschrieb

Partitionierung

p, $\overline{i}$, $\underline{j}$

3	6	8	1	0	7	2	4	5	9
$\underline{9}$	6	8	1	0	7	2	4	5	3
$\overline{9}$	$\underline{6}$	8	1	0	7	2	4	5	3
$\overline{9}$	6	$\underline{8}$	1	0	7	2	4	5	3
$\overline{9}$	6	8	$\underline{1}$	0	7	2	4	5	3
1	$\overline{6}$	8	9	$\underline{0}$	7	2	4	5	3
1	0	$\overline{8}$	9	6	$\underline{7}$	2	4	5	3
1	0	$\overline{8}$	9	6	7	$\underline{2}$	4	5	3
1	0	2	$\overline{9}$	6	7	8	$\underline{4}$	5	3
1	0	2	$\overline{9}$	6	7	8	4	$\underline{5}$	3
1	0	2	$\overline{9}$	6	7	8	4	5	3
1	0	2	$\overline{3}$	6	7	8	4	5	9

Gesamt

3 6 8 1 0 7 2 4 5 9

1 0 2 | 3 | 6 7 8 4 5 9

0 | 1 | 2 | | 4 5 | 6 | 9 7 8

4 | 5 | | 8 7 | 9 |

7 | 8 |

Algorithmus

qSort $(list : < e_1, \cdots, e_n >, start : int, end : int)$

1. Abbruchbedingung: $start >= ende$
2. Suche Pivot-Element

3. $m = partition(list, start, end, pivotPos)$
4. Führe qSort aus für $< e_{start}, \cdots, e_{m-1} >$ und $< e_{m+1}, \cdots, e_{end} >$

Partition $(list : < e_1, \cdots, e_n >, start : int, end : int,\ pivotPos : int)$

1. Tausche das Pivot Element ans Ende des Arrays
2. Sei $i := start$ (i markiert den Start vom Bereich größer als Pivot)
3. Für jedes Element in der Teilliste von $start$ bis $ende - 1$
    a. Wenn aktuelles Element j kleiner als Element i
        i. dann tausche i und j
        ii. $i++$
4. Tausche Pivot vom Ende zurück in die Mitte (tausche i und $ende$) und gibt die Position vom Pivot zurück

Quickselect

Algorithmus um ein Element nahe des Medians in einer Liste zu finden. Gib das n-kleinste Element zurück.

s	k	p	a	b	c
$\langle 3,1,4,5,9,\mathbf{2},6,5,3,5,8\rangle$	6	2	$\langle 1\rangle$	$\langle 2\rangle$	$\langle 3,4,5,9,6,5,3,5,8\rangle$
$\langle 3,4,5,9,\mathbf{6},5,3,5,8\rangle$	4	6	$\langle 3,4,5,5,3,5\rangle$	$\langle 6\rangle$	$\langle 9,8\rangle$
$\langle 3,4,\mathbf{5},5,3,5\rangle$	4	5	$\langle 3,4,3\rangle$	$\langle 5,5,5\rangle$	$\langle\rangle$

qSelect$(list : < e_1, \cdots, e_n >, start : int, end : int, nSmallest : int)$

1. Abbruchbedingung: $end == start$ (Rückgabe vom Pivot)
2. Wähle zufälligen $pivotIndex \in [start, end]$
3. $pivotIndex = Partition(list, start, end, pivotIndex)$
4. Wenn $nSmallest == pivotIndex$
    a. Gib $list[pivotIndex]$ zurück
5. Sonst, wenn $nSmallest < pivotIndex$
    a. Gib zurück $qSelect(list, start, pivotIndex - 1, nSmallest)$
6. Sonst
    a. Gib zurück $qSelect(list, pivotIndex + 1, end, nSmallest)$

Bucketsort

Algorithmus

1. Initialisiere ein K großes Array
2. Für jedes Element hänge beim jeweiligen Bucket das Element an
3. Füge alle Buckets zusammen (bei mehreren Elementen in einem Bucket nutze Selcetionsort)

LSD-Radixsort

Algorithmus

1. Jeweils nach den Stellen, beginnend von hinten immer in die jeweiligen Fächer von $0, \cdots, 9$ einsortieren.
2. Wiederhole Schritt 1 solange, bis die Anzahl der Stellen der größten Zahlen abgearbeitet wurden

3. Füge alle Fächer zusammen

Heapsort

Algorithmus

1. Baue Heap auf mit Eingabearray
2. Solange Heap nicht leer, rufe $DeleteMin()$ und füge es dem Ergebnisarray hinzu.

Sortieralgorithmen-Übersicht

	best	average	worst	Speicher	inplace	stabil
Insertsort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Ja	Ja
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Nein	Ja
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	Nein	Nein
Bucketsort	$O(n+K)$	$O(n+K)$	$O(n+K)$	$O(K)$	Nein	Ja
Radixsort	$O(d(n+K))$	$O(d(n+K))$	$O(d(n+K))$	$O(n)$	(Nein)	(Ja)
Heapsort	$O(n\ log\ n)$	$O(n\ log\ n)$	$O(n\ log\ n)$	$O(1)$	Ja	Nein

Graphen

Eigenschaften

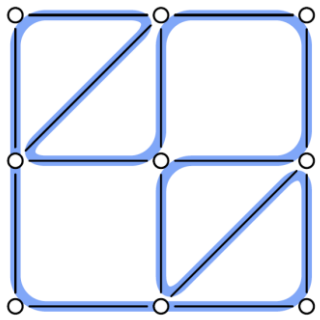
Direkter Azyklischer Graph (DAG)

Bipartitheit

Ein bipartiter Graph ist ein Graph, in dem zwei Gruppen von Knoten existieren, innerhalb derer keine Knoten miteinander verbunden sind.

Eulerscher Kreis

Ein Zyklus, der alle Kanten eines Graphen genau einmal enthält.

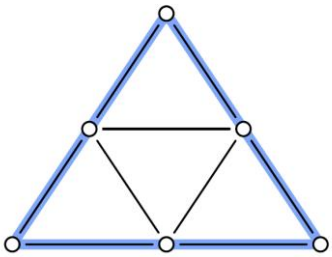


Voraussetzung

Der Graph G ist eulersch, wenn G zusammenhängend ist und alle Knoten geraden Knotengrad haben (und die Kantenmenge nicht leer ist).

Hamiltonscher Kreis

Ein Zyklus, der alle Knoten eines Graphen genau einmal enthält (Beginn und Ende wird nur einmal gezählt).



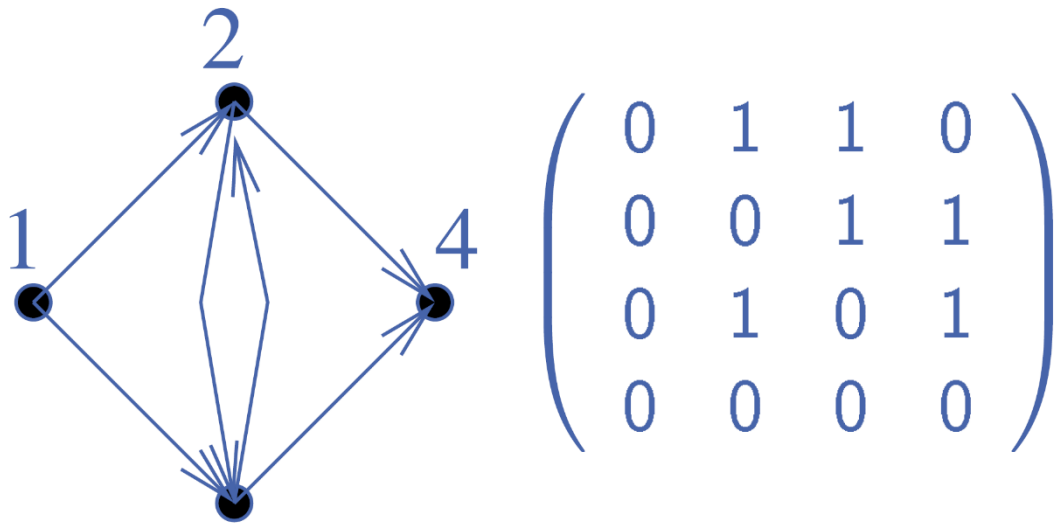
Graphenrepräsentation

Kantenfolge

Alle Kanten werden in einer Liste oder einem Feld gespeichert.

Adjazenzmatrizen

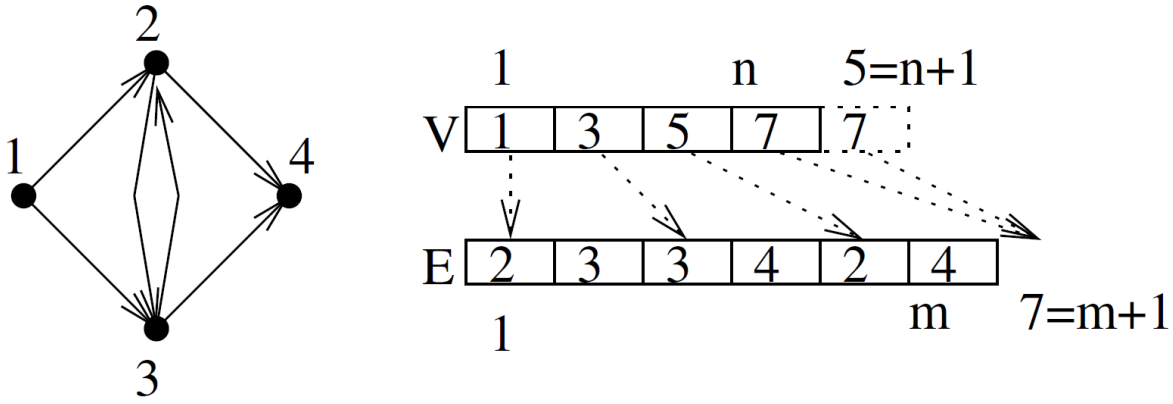
Eine Matrix A bei der gefragt wird, ob eine Kante von einem zu einem anderen Knoten besteht.



$$A_{ij}^{k} = Anzehl\ der\ k - Kanten - Pfade\ von\ i\ nach\ j$$

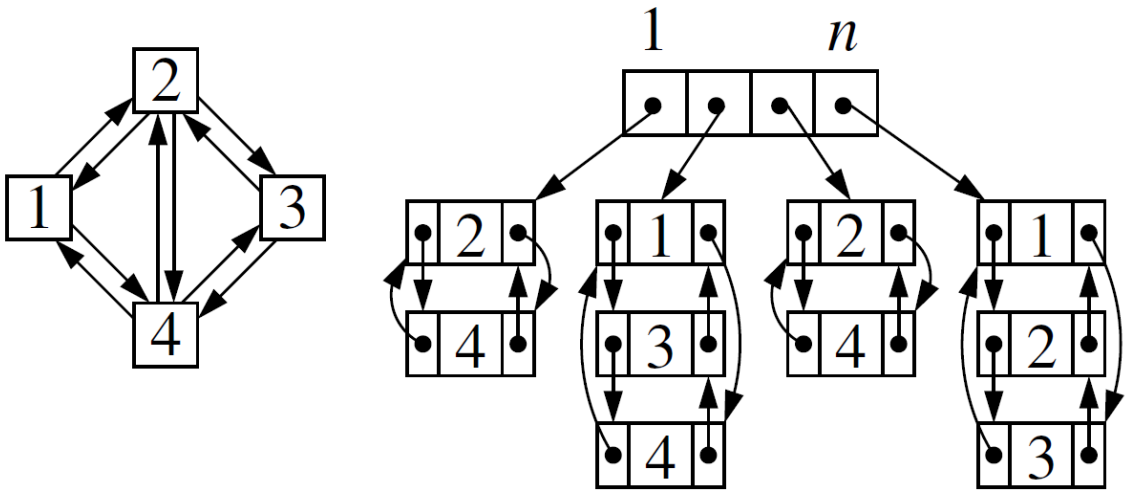
Adjazensfelder

Besteht aus zwei Feldern. Das Feld E speichert den Endknoten der jeweiligen Kante. Der Index eines Elements in V gibt den Startknoten der Kante an. Der Wert des Elements ist der Startindex der Endknoten in E.



Adjazenzlisten

Ein Feld das auf Listen Zeigt, die jeweils alle Nachfolger enthält.



Graphentraversierung

Breitensuche

Laufzeit

$$O(|V| + |E|)$$

Algorithmus

1. Bestimme den Knoten, an dem die Suche beginnen soll, markiere ihn als besucht und speichere ihn in einer Warteschlange ab.
2. Entnimm einen Knoten vom Beginn der Warteschlange.
    a. Falls das gesuchte Element gefunden wurde, brich die Suche ab und liefere „gefunden“ zurück.
    b. Anderenfalls hänge alle bisher unmarkierten Nachfolger dieses Knotens ans Ende der Warteschlange an und markiere sie als besucht.
3. Wenn die Warteschlange leer ist, dann wurde jeder Knoten bereits untersucht. Beende die Suche und liefere „nicht gefunden“ zurück.
4. Wiederhole ab Schritt 2.

Tiefensuche

Laufzeit

$$O(|V| + |E|)$$

Algorithmus

1. Bestimme den Knoten, an dem die Suche beginnen soll
2. Expandiere den Knoten und speichere der Reihenfolge nach den noch nicht erschlossenen Nachfolger in einem Stack
3. Rufe rekursiv für jeden der Knoten in dem Stack DFS (depth first search oder Tiefensuche) auf

a. Falls das gesuchte Element gefunden worden sein sollte, brich die Suche ab und liefere ein Ergebnis
b. Falls es keine nicht erschlossenen Nachfolger mehr gibt, lösche den obersten Knoten aus dem Stack und rufe für den jetzt oberen Knoten im Stack DFS (depth first search oder Tiefensuche) auf

Kürzeste Wege

Dijkstra -Algorithmus

Vorbedingung

- Graph ist zusammenhängend
- Graph hat keine *negativen* Kreise

Laufzeit

$$O(|V| * \log(|V|) + |E|)$$

Algorithmus

1. Weise allen Knoten die beiden Eigenschaften „Distanz“ und „Vorgänger“ zu. Initialisiere die Distanz im Startknoten mit 0 und in allen anderen Knoten mit ∞.
2. Solange es noch unbesuchte Knoten gibt, wähle darunter denjenigen mit minimaler Distanz aus und
    a. speichere, dass dieser Knoten schon besucht wurde.
    b. Berechne für alle noch unbesuchten Nachbarknoten die Summe des jeweiligen Kantengewichtes und der Distanz im aktuellen Knoten.
    c. Ist dieser Wert für einen Knoten kleiner als die dort gespeicherte Distanz, aktualisiere sie und setze den aktuellen Knoten als Vorgänger (auch als relaxieren bezeichnet).

Bellman-Ford-Algorithmus

Laufzeit

$$O(|V| * |E|)$$

Algorithmus

1. Weise allen Knoten die beiden Eigenschaften „Distanz“ und „Vorgänger“ zu. Initialisiere die Distanz im Startknoten mit 0 und in allen anderen Knoten mit ∞.
2. Alle Kanten relaxieren ((n-1)-mal)
3. Wenn sich bei erneutem relaxieren nochmal eine Änderung gibt so gibt es einen negativen Kreis

Von überall nach überall

Nichtnegative Kanten

|V|-mal Dijkstra

Beliebige Kantengewichte (nicht optimal -> nur Algo I)

|V|-mal Bellman-Ford

Minimale Spannbäume

Definition

Ein Teilgraph eines ungerichteten Graphen, der ein Baum ist und alle Knoten dieses Graphen enthält (mit minimalem Gesamtgewicht der Kanten).

Jarník-Prim-Algorithmus

Laufzeit

$$O(|V| * \log(|V|) + |E|)$$

Algorithmus

1. Wähle einen beliebigen Knoten als Startgraph T.
2. Solange T noch nicht alle Knoten enthält:
    a. Wähle eine Kante e mit minimalem Gewicht aus, die einen noch nicht in T enthaltenen Knoten v mit T verbindet.
    b. Füge e und v dem Graphen T hinzu.

Kruskal Algorithmus

Laufzeit

$$O(|E| * \log(|E|))$$

Algorithmus

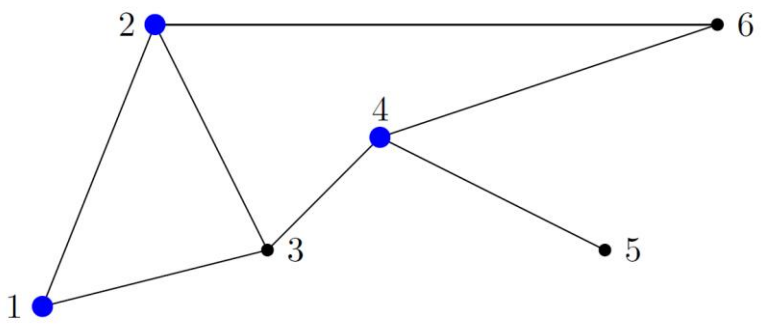
1. Sei T = {}
2. Für jede Kante (u,v) (nach aufsteigender Reihenfolge) in E
    a. Wenn u und v in unterschiedlichen Teilbäumen von (V,T) sind dann füge u und v zu T hinzu.

Jarník-Prim vs. Kruskal

- Pro Jarník-Prim:
    - Asymptotisch gut für alle |E|, |V|
    - Sehr schnell für |E| >> |V|
- Pro Kruskal:
    - Gut bei $|E| = O(|V|)$
    - Kantenliste reicht aus
    - Profitiert von schnellen Sortieren

Vertex-Cover

Eine Teilmenge der Knoten eines Graphen, sodass jede Kante mindestens einen Knoten dieser enthält.



Optimierung

Linear Programming

- Kostenfunktion $f(x) = c * x$, wobei c der Kostenvektor ist.

- $m\ Constraints$ der Form $a_i * x \sim_i b_i$
  $wobei \sim_i \epsilon \{\leq, \geq, =\}$

Lösbar in polynomieller Zeit.

Dynamische Programmierung

Voraussetzung

Die Optimale Lösung setzt sich zusammen aus optimalen Lösungen von Teilproblemen. Gibt es mehrere Lösungen für ein Teilproblem so muss es egal welche Lösung für dieses benutzt wird.

Rucksackproblem

$Sei\ w_i\ das\ Gewicht\ und\ p_i\ der\ Gewinn\ des\ Elements\ i; C\ die\ Kapazität\ des\ Rucksacks$

In folgender Tabelle gilt somit die Formel:

$$P(i, C) = \max(P(i-1, C), P(i-1, C-p_i) + p_i)$$

Maximiere $(10, 20, 15, 20) \cdot \mathbf{x}$,
so dass $(1, 3, 2, 4) \cdot \mathbf{x} \leq 5$

$P(i, C), (\text{decision}[i, C])$

$i \setminus C$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0, (0)	10, (1)	10, (1)	10, (1)	10, (1)	10, (1)
2						
3						
4						

Algorithmus

1. Beginne mit i=1
2. Fülle jede Zeile der Tabelle mit $P(i, C) = \max(P(i-1, C), P(i-1, C-p_i) + p_i)$ aus.
   a. Wenn $P(i-1, C-p_i) + p_i > P(i-1, C)$ dann wird das aktuelle Element nicht genommen ($decision[i, C] = 0$)

Lösung Zusammensetzen

1. Initialisiere $c := Kapazität\ des\ Rucksacks$
2. Durchlaufe alle Elemente (beginnend von hinten -> größtes i)
3. Ist $decision[i, c] == 1$
   a. Reduziere c um das Gewicht vom Element i
   b. Das Element ist in der Lösung enthalten

Systematische Suche

Algorithmus

1. Rekursiver Algorithmus.

2. Schätze obere Schranke ab.
3. Ist obere Schranke größer als bisher Erreichtes dann
    a. Ist noch Platz für das Element i
        i. Steige weiter nach links ab (aktuelles Element wird eingepackt)
    b. Ist die obere Schranke noch immer größer als bisher Erreichtes
        i. Steige nach rechts ab (aktuelles Element wird nicht eingepackt)

Beispiel

Gewicht	1	3	2	4
Gewinn	10	24	14	20