EidA Zusammenfassung

Felix Leitl

12. Juli 2023

Inhaltsverzeichnis

Laufzeit **3**
- Θ-Notation . . . 3
- $\mathcal{O}$-Notation . . . 3
- Ω-Notation . . . 3
- Funktionsklassen . . . 3
- Rechenregeln . . . 3
  - Produktregel . . . 3
  - Summenregel . . . 3
  - Multiplikation mit einer Konstante . . . 3
- Funktionen Vergleichen . . . 4
  - Transitivität . . . 4
  - Reflexivität . . . 4
  - Symmetrie . . . 4
  - Transponierende Symmetrie . . . 4
- Rekursion, Rekurrenz und das Mastertheorem . . . 4
  - Rekursion . . . 4
  - Mastertheorem . . . 5

Sortieren **5**
- Definitionen . . . 5
- Vergleichsbasierte Algorithmen . . . 6
  - Insertionsort . . . 6
  - Bogosort . . . 6
  - Slowsort . . . 6
  - Bubblesort . . . 6
  - Mergesort . . . 6
  - Selectionsort . . . 6
  - Heapsort . . . 7
  - Quicksort . . . 7
- Nicht vergleichsbasierte Algorithmen . . . 7
  - Bucketsort . . . 7
  - Countingsort . . . 7
  - Radixsort . . . 8

Graphen **8**
Definitionen . . . 8
Graph . . . 8
Vollständiger Graph . . . 8
Grad . . . 8
Regulärer Grad . . . 8
Teilgraph . . . 8
Wege und Kreise . . . 8
Kürzeste Distanz . . . 9
Zusammenhängend . . . 9
Wald . . . 9
Baum . . . 9
Disjunktionen . . . 9
Bipartierter Graph . . . 9
Zusammenhängende Digraphen . . . 9
Netzwerke . . . 9
Fluss . . . 9
Größe des Flusses . . . 10
Algorithmen . . . 10
Breitensuche . . . 10
Tiefensuche . . . 10
Dijkstra . . . 10
Floyd-Warshall . . . 11
Page Rank . . . 11
Kruskal . . . 11
Prim . . . 11
Rückwärts Kruskal . . . 11

Laufzeit

Θ-Notation

Def. $\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 \geq 0, \text{ sodass } 0 \leq c_1 \cdot g(n) \leq c_2 \cdot g(n) \forall n \geq n_0\}$

Bedeutung:

1. kleine Werte von n sind nicht. wichtig
2. c_1 und c_2 begrenzen f nach oben und unten

$\mathcal{O}$-Notation

Def. $\mathcal{O}(g(n)) = \{f(n) : \exists c, n_0 \geq 0, \text{ sodass } 0 \leq f(n) \leq c \cdot g(n) \forall n \geq n_0\}$

1. $f = \Theta(g(n)) \Rightarrow f(n) = \mathcal{O}(g(n))$
2. $\mathcal{O}$-Notation gibt keine exakte obere Schranke an

Ω-Notation

Def. $\Omega(g(n)) = \{f(n) : \exists c, n_0 \geq 0, \text{ sodass } 0 \leq c \cdot g(n) \leq f(n) \forall n \geq n_0\}$

Theorem. Für zwei beliebige Funktionen $f(n)$ und $g(n)$ gilt: $f(n) = \Theta(g(n))$ genau dann wenn $f(n) = \mathcal{O}(g(n))$ und $f(n) = \Omega(g(n))$

Funktionsklassen

g(n)	**Wachstum**
1	konstant
$\log n$	logarithmisch
n	linear
$n \log n$	leicht überlinear
n^2	quadratisch
n^3	kubisch
n^k	polynomiell (k=Konstante)
2^n	exponentionell

Rechenregeln

Produktregel

$$f_1 = \mathcal{O}(g_1), f_2 = \mathcal{O}(g_2) \Rightarrow f_1 \cdot f_2 = \mathcal{O}(g_1 \cdot g_2)$$

Summenregel

$$f_1 = \mathcal{O}(g_1), f_2 = \mathcal{O}(g_2) \Rightarrow f_1 + f_2 = \mathcal{O}(g_1 + g_2)$$

Multiplikation mit einer Konstante

Die Konstante fällt weg

Funktionen Vergleichen

Transitivität

$$f(n) = \Theta(g(n)) \text{ und } g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$
$$f(n) = \mathcal{O}(g(n)) \text{ und } g(n) = \mathcal{O}(h(n)) \Rightarrow f(n) = \mathcal{O}(h(n))$$
$$f(n) = \Omega(g(n)) \text{ und } g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$

Reflexivität

$$f(n) = \Theta(f(n))$$
$$f(n) = \mathcal{O}(f(n))$$
$$f(n) = \Omega(f(n)$$

Symmetrie

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

Transponierende Symmetrie

$$f(n) = \mathcal{O}(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$

Rekursion, Rekurrenz und das Mastertheorem

Rekursion

Zusätzliche Kosten durch Divide and Conquer:

1. $T(n)$: Kosten für das Lösen der Instanz der Größe n
2. $f(n)$: Kosten für das Divide einer Instanz der Größe n
3. $g(n)$: Kosten für das Mergen von k Teillösungen

Daraus ergibt sich:

$$T(n) = k \cdot T(m) + f(n) + g(n)$$

Für $T(m)$ ergibt sich:

$$T(m) = k \cdot T(l) + f(m) + g(m)$$

Einsetzen von $T(m)$ in $T(n)$:

$$T(n) = k \cdot (k \cdot T(l) + f(m) + g(m)) + f(n) + g(n) = k^2 \cdot T(l) + k \cdot f(m) + k \cdot g(m) + f(n) + g(n)$$

Mastertheorem

Gegebne sei eine Rekurrenz der Form $T(n) = aT(\frac{n}{b} + f(n)$ mit Konstanten $a \geq 1$ und $b \geq 1$ sowie einer beliebigen Funktion $f(n)$. $T(n)$ kann asymptotisch wie folgt beschränkt werden:

1. Falls $f(n) = \mathcal{O}(n^{\log_b(a)-\epsilon})$, für eine Konstante ϵ, dann gilt $T(n) = \Theta(n^{log_b a})$
2. Falls $f(n) = \Theta(n^{\log_b(a)})$, dann gilt $T(n) = \Theta(n^{\log_b a} \log n)$
3. Falls $f(n) = \Omega(n^{\log_b(a)+\epsilon})$, für eine Konstante ϵ und falls $af(\frac{n}{b}) \leq c \cdot f(n)$ für eine Konstante $c < 1$ und ab einem $n_0 > 0$ für alle $n > n_0$, dann gilt $T(n) = \Theta(f(n)$

Sortieren

Definitionen

Damit die Elemente sortiert werden können muss eine Ordungsrelation vorhanden sein:

Es sei $R \subseteq AxA$ eine binäre Relation auf der Menge A.

- R heißt Quasiordnung auf A genau dann wenn R reflexiv und transitiv ist
- R heißt partielle Ordnung auf A genau dann, wenn R transitiv, reflexiv und antisymmetrisch ist
- R auf A heißt lineare Ordnung oder Totalordnung genau dann, wenn zusätzlich gilt: $\forall a, b \in A : R(a, b) \vee R(b, a)$

Ist R eine Relation des Typs 1-3, schreibt man meist $a \leq_R b$ oder $a \leq b$
Wir betrachten ebenfalls sogenannte strikte Ordnungen bei denen ein Element niemals mit sich selbst in Relation stehen darf.

Es sei $R \subseteq AxA$ eine binäre Relation auf der Menge A.

- R heißt partielle Ordnung auf A wenn R irreflexiv und transitiv ist
- R auf A heißt strikte Ordnung genau dann, wenn zusätzlich gilt: $\forall a \neq b \in A : R(a, b) \vee R(b, a)$

Für Sortierverfahren braucht es eine strenge schwache Ordnung:

Es sei $R \subseteq AxA$ eine binäre Relation auf der Menge A. R ist eine strenge schwache Ordnung wenn R eine strikte Ordnung und diese zusätzlich negativ transitiv ist.

Vergleichsbasierte Algorithmen

Brauchen im worst-case immer mindestens $\log(n!) = \Theta(n \log(n))$ Vergleiche

Insertionsort

Idee: Füge das aktuelle Element an die richtige Position im sortierten Teil ein

Best case: $\mathcal{O}(n)$
Worst case: $\mathcal{O}(n^2)$

Bogosort

Idee: wähle eine zufällige Permutation und überprüfe, ob diese sortiert ist

Best case: $\mathcal{O}(n)$
Average case: $\mathcal{O}(e-1)n!$
Worst case: $\mathcal{O}(n \cdot n!)$

Slowsort

Idee: Verzögere die Sortierung so lange es geht

Best case: $\mathcal{O}(n^{\frac{\log(n)}{2+e}})$
Average case: $\mathcal{O}(n^{\frac{\log(n)}{2+e}})$
Worst case:$\mathcal{O}(n^{\frac{\log(n)}{2+e}})$

Bubblesort

Idee: Das kleinste Element steigt wie eine Blase auf, dabei werden die Elemente paarweise verglichen

Best case: $\mathcal{O}(n)$
Worst case: $\mathcal{O}(n^2)$

Mergesort

Idee: Slowsort ist so langsam, weil es brauchbare Zwischenergebnisse ignoriert. Anstatt nur die beiden letzten Elemente der über Rekursion erhaltenen sortierten Teile zu vergleichen, fügt man die Teile zusammen, wobei man die Sortierung erhält

Rekurrenzgleichung: $T(n) = 2 \cdot T(\frac{n}{2} + \Theta(n))$
Daraus folgt mit dem Mastertheorem (2. Fall): $T(n) = \Theta(n \log n)$

Selectionsort

Idee: Selectionsort sucht für die aktuelle Position das geeignete Element, nämlich das kleinste (bzw. größte) im unsortierten Teil

Laufzeit: $\Theta(n^2)$

Heapsort

Idee: Nutzte max-heap.Eigenschaft eines Heaps, da Maximum an erster Stelle

Laufzeit: $\mathcal{O}(n \log n$

Quicksort

Idee:

- Divide: teile an Pivotelement in linke Hälfte, rechte Hälfte und Pivotelement, nach Partitionsschritt
- Conquer: rekursiv auf Hälften anwenden
- Combine: Zusammenfügen der einzelnen Elemente

Worst case: $\Theta(n^2)$
Best case: $\Theta(n \log n)$

Nicht vergleichsbasierte Algorithmen

Durch lösen von der Beschränkung auf Vergleiche schneller werden

Bucketsort

Idee:

1. Bucketsort nimmt an, dass die Elemente des zu sortierenden Arrays A im Intervall $[0, 1)$ gleich verteilt sind
2. Bucketsort teilt das Intervall $[0, 1)$ in n (Länge von A) gleich große Teilintervalle ("Buckets")
3. Anschließend werden die n Elemente von A in die Buckets verteilt
4. Nichtleere Buckets werden sortiert
5. Anschließend wird das Ergebnis aus den Buckets zusammengestzt

Best case: $\Theta(n)$
Average case: $\Theta(n)$
Worst case: $\Theta(n^2)$

Countingsort

Idee:

- Nimmt an, dass jedes der n Elemente des zu sortierenden Arrays A eine Ganzzahl zwischen 0 und k ist
- Für jedes Element x aus dem Array A wird zunächst die Anzahl an Elementen bestimmt, die $\leq x$ sind
- Diese Information wird dann genutzt um das Element x direkt an seine richtige Position im sortierten Array zu plazieren

Laufzeit: $\Theta(n + k)$

Radixsort

Idee: Sortiere die Elemente im sortierenden Array A Ziffer für Ziffer, beginne bei Least Significant Digit

Laufzeit: $\Theta(d \cdot (n + k))$ für n d-stellige Zahlen

Graphen

Definitionen

Graph

Sei $V = \{v_1, ..., v_n\}$ eine endliche Menge und $E \subseteq P_2(V) = \{\{u, v\} | u, v \in V, u \neq v\}$. Dann heißt das geordnete Paar $G = (V, E)$ (endlicher, schlichter, ungerichteter) Graph, wobei V die Knotenmenge und E die Kantenmenge von G ist. Ist $e = \{u, v\} \in E$ so sind u und v benachbart (adjazent). $e = \{u, v\}$ oder auch $u - v$ ist dabei eine Kante

Vollständiger Graph

Ein Graph heißt vollständig, wenn jede Ecke mit jeder anderen Ecke durch genau eine Kante verbunden ist. K_n bezeichnet den vollständigen Graphen mit n Ecken

Grad

Der Grad eines Knoten in $G = (V, E)$ ist die Anzahl an benachbarten Knoten. Der Grad eines Graphen entspricht dem maximalen Grad eines enthaltenen Knotens

Regulärer Grad

Ein Graph heißt regulär, wenn alle Knoten des Graphen den gleichen Grund haben

Teilgraph

Seien G und H Graphen. H heißt Teilgraph A von G, falls $V(H) \subseteq V(G)$ und $E(H) \subseteq E(G)$ gilt

Wege und Kreise

Ein Weg (auch Pfad genannt) ist eine Folge von Kanten. Ein einfacher Weg besucht keine Knoten doppelt. Ein Kreis ist ein einfacher Weg, wobei Anfangs- und Endknoten äquivalent sind. Ein einfacher Kreis besucht keine Knoten mehrfach

Kürzeste Distanz

Sei $G = (V, E)$ ein Graph und seien $u, v \in V$ Knoten. Die kürzeste Distanz $\delta(u, v)$ von Knoten u nach Knoten v ist die minimale Anzahl an Knoten in einem Pfad von u nach v. Wenn kein Pfad von u nach v existiert, so ist $\delta(u, v) = \infty$. Ein Pfad $\delta(u, v)$ vom u nach v wird als kürzester Pfad bezeichnet

Zusammenhängend

Graph $G = (V, E)$ heißt zusammenhängend, falls zwischen je zwei Knoten $u, v \in V$ ein Weg existiert

Wald

Graph $G = (V, E)$ ist ein Wald, falls G keine einfachen Kreise enthält

Baum

Graph $G = (V, E)$ ist ein Wald, falls G ein zusammenhängender Wald ist

Disjunktionen

Zwei Graphen sind disjunkt, wenn es keine Weg von dem einen in den anderen gibt. Zwei Wege sind disjunkt, wenn es keinen Knoten gibt, der in beiden enthalten ist

Bipartierter Graph

Ein einfacher Graph $G = (V, E)$ heißt bipartit, falls sich seine Knoten in zwei disjunkten Teilmengen A und B aufteilen lassen, sodass zwischen den Knoten innerhalb beider Teilmengen keine Kanten verlaufen

Zusammenhängende Digraphen

Digraph $G = (V, E)$ heißt stark zusammenhängend, falls für je zwei Knoten $u, v \in V$ gilt: Es gibt einen einfachen Weg von u nach v und von v nach u. Falls G ungerichtet zusammenhängend ist, ist G schwach zusammenhängend

Netzwerke

Ein gerichtetes Netzwerk $N = (V, E, c)$ besteht aus einem gerichtetem Graphen $G = (V, E)$ und einer Kapazitätsfunktion $c : E \to \mathbb{R}^+$ so wie zwei dedizierten Knoten $s, t \in V$, genannt die Quelle s und die Senke t

Fluss

Es sei $G = (V, E)$ ein gerichteter Graph, $c : E \to \mathbb{R}^+$ eine Kapazitätsfunktion, $s \in V$ die Quelle und $t \in V$ die Senke. Ein Fluss F in G ist eine Funktion $f : VxV \to \mathbb{R}$ die folgende Eigenschaften erfüllt:

1. Kapazitätsbegränzung

$$\forall u, v \in V \text{ gilt } 0 \leq f(u,v) \leq c(u,v)$$

   Fluss positiv und höchstens so groß wie die Kapazität

2. Flusserhaltung

$$\forall u \in V - \{s,t\}, \text{ gilt } \sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v)$$

   Der gesamte Fluss in einem Knoten (ohne s,t) muss genauso groß sein wie der gesamte Fluss aus diesem Knoten "Flow in = Flow out"

Größe des Flusses

Die Größe des ist definiert als die Summe aller Flüsse von der Quelle in den Knoten v minus die Summe aller Flüsse von dem Knoten v zur Quelle

Algorithmen

Breitensuche

Idee:

- Breitensuche arbeitet iterativ und läuft Level für Level durch den Graphen
- Level für Level bedeutet intuitiv, dass, gegeben ein Startknoten, sich der Algorithmus erst alle Nachbarn des Startknoten anschaut, bevor er die Nachbarn der Nachbarn anschaut, usw.

Laufzeit: $\mathcal{O}(|V| + |E|)$

Tiefensuche

Idee: Tiefensuche arbeitet im Gegensatz zur Breitensuche nicht Level für Level, sondern steigt immer so weit es geht in die Tiefe

Dijkstra

Idee:

- Generalisiert BFS für gewichtete Graphen
- Relaxierugn: Die Pfadlänge zu einem Knoten wird geschätzt und während des Durchlaufs aktualisiert
- Initial werden die Kosten der nicht zu erreichenden Knoten auf sehr hohe Werte gesetzt (für den Startknoten auf 0)
- Die Relaxoerung wird so lange. durchgeführt, bis keine Aktualisierung mehr möglich ist

Laufzeit: $\mathcal{O}(|V|^2 + |E|)$

Floyd-Warshall

Idee:

- Dynamische Programierung
- Wähle einen Knoten v aus und betrachte ob es über v einen kürzeren Weg zwischen allen andern Knotenpaaren gibt: Aktualisiere gegebenenfalls
- Wiederhole dies für alle Knoten

Laufzeit: $\Theta(|V|^3)$

Page Rank

Idee:

- Initial ist $r = [\frac{1}{n}, ..., \frac{1}{n}]^T$
- Wir multiplizieren r nun solange mit M^T bis wir das Ergebnis gut genug approximiert haben

$$r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1 - \beta)\frac{1}{n} + (\sum_{i \in D} \beta \frac{r_i}{n})$$

Wobei r der Pagrank, d der Ausgangsgrad, β die Sprungwahrscheinlichkeit, n die Anzahl der Knoten und D die Menge der dead ends ist

Kruskal

Idee: Wähle $n - 1$ Kanten mit minimalem Gewicht aus, sodass kein Kreis entsteht:

- Beginne mit leerer Kantenmenge
- Füge die günstigste Kante hinzu, die keinen Kreis verursacht

Prim

Idee: Wähle $n - 1$ Kanten mit minimalem Gewicht aus, sodass alle Knoten zusammenhängend sind:

- Wähle eine beliebigen Startknoten s
- Füge die günstigste Kante hinzu, die einen neuen Knoten zur Zusammenhangskomponente von s hinzu

Rückwärts Kruskal

Idee: Entferne $m - n + 1$ Kanten mit maximalem Gewicht, sodass alle Knoten zusammenhängen bleiben:

- Zu Beginn alle Kanten im Graph
- Entferne teuerste Kante, sodass alle Knoten zusammenhängend bleiben