Theoretische Informatik 1
Algorithmen und Datenstrukturen

Klausurvorbereitung

Marvin Borner

Wintersemester 2022/23

Dies ist meine WIP Zusammenfassung, welche hauptsächlich mir dienen soll. Ich schreibe außerdem ein inoffizielles Skript, welches auf https://marvinborner.de/algo.pdf zu finden ist.

Inhalt

1 Tricks

- $\log_a(x) = \frac{\log_b(x)}{\log_b(a)}$

2 Big-O-Notation

- $f \in \mathcal{O}(g)$: $f \leq g$
- $f \in \Omega(g)$: $f \geq g$
- $f \in o(g)$: $f < g$
- $f \in \omega(g)$: $f > g$
- $f \in \Theta(g)$: $f = g$

3 Master Theorem

Mit $T(n) = aT(\lceil n/b \rceil) + \mathcal{O}(n^d)$ für $a > 0$, $b > 1$ und $d \geq 0$, ist

$$T(n) = \begin{cases} \mathcal{O}(n^d) & d > \log_b a \\ \mathcal{O}(n^d \log_n) & d = log_b a \\ \mathcal{O}(n^{log_b a}) & d < \log_b a \end{cases}$$

4 Bäume

- Binary Höhe: $\log n$; Binary Anzahl: $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$
- Heap:
  - `Heapify`: rekursiv swap bis Bedingung nicht verletzt: $\mathcal{O}(\log n)$
  - `Decrease`: erniedrigen, dann heapify auf node: $\mathcal{O}(\log n)$
  - `Increase`: erhöhen, dann rekursiv mit parent swappen bis bedingung nicht verletzt: $\mathcal{O}(\log n)$
  - `ExtractMax`: Wurzel mit letzter leaf ersetzen, `heapify(root)`: $\mathcal{O}(\log n)$
  - `Insert`: Einfügen an nächster Stelle mit $-\infty$, `Increase(x)` : $\mathcal{O}(\log n)$
  - `Build`: Array irgendwie als Baum schreiben, `heapify` auf jedem Knoten von unten nach oben: $\mathcal{O}(n)$
- Priority queue: Als heap analog

5 Hashing

- irreversible Reduzierung des Universums

6 Graphen

- Adjazenzmatrix: $a_{ij} = 1$ bzw Gewicht wenn Kante von i nach j
  - für dense
- Adjazenzliste: array von Knoten mit Listen von ausgehenden Kanten
  - für sparse
- Sources: längste finishing time von DFS
- Sinks: alle Kanten umdrehen, dann sources finden
- SCC: DFS auf umgedrehtem Graph, startend bei ursprünglicher source ($\rightarrow$SCC). Weiter bei restlichen Knoten (nach absteigenden finishing times): $\mathcal{O}(V + E)$
- Cycles: durch DFS wenn visited nochmal visited wird
- Topological sort: no cycles; DFS und Knoten nach absteigenden finishing times sortieren

6.1 DFS

- $\mathcal{O}(V+E)$ bzw. $\mathcal{O}(V^2)$ für Matrix

TODO: Anleitung schriftlich

6.2 BFS

- gleich wie DFS nur queue statt stack
- $\mathcal{O}(V+E)$ bzw. $\mathcal{O}(V^2)$ für Matrix
- gut für shortest path, einfach jedes Mal distance updaten

6.3 Relaxation

- anfangs alle ∞ außer Start

```
function Relax(u,v)
    if v.dist > u.dist + w(u, v)
      v.dist = u.dist + w(u,v)
      v.π = u
```

6.4 Bellman-Ford

- $(|V|-1)$-mal alle Kanten relaxen, dann noch einmal alle relaxen (wenn sich was ändert, dann cycle): $\mathcal{O}(V \cdot E)$
- in undirected muss in beide Richtungen relaxed werden
- blöd für negative Gewichte
- geht auch dezentral/asynchron, damit einzelne Knoten sich updaten können

6.5 Dijkstra

- gierig: nimmt einfach immer die nächstbeste Kante
- $\mathcal{O}((V+E)\log V)$ mit min-priority Queues
- für PPSP einfach stoppen wenn Knoten erreicht wurde (oder auch bidirektional, dann besser)

6.6 Floyd-Warshall

- besser für APSP (da Bellmand/Dijkstra je für jeden Vertex ausführen müssten: $\mathcal{O}(V^2 \cdot E ...)$)
- dreimal `for` für Matrix

6.7 A*

- wie Dijkstra, nur mit nächstem $d(s,u)+w(u,v)+\pi(v)$ statt $d(s,u)+w(u,v)$ (Heuristik)

6.8 Kruskal

- für MST
- Kanten aufsteigend sortieren; je nächste Kante zu leerem Graph hinzufügen solange kein Zykel entsteht
- oder voll toll mit union-find: $\mathcal{O}(E \log V)$
  - weil dann Zykelerkennung ez geht und so

6.9 Prim

- für MST
- bei random starten, dann immer nächstbeste Kante hinzufügen (Kreis erweitert sich)
- mit priority queue $\mathcal{O}(E \log V)$

7 Sortierung

- **stabil**: wenn gleicher Wert am Ende gleichen Index
  - selection: stabil
  - insertion: stabil
  - heap: instabil
  - merge: stabil
  - quick: stabil (out-of-place)
  - counting: stabil (mit Listen als buckets)
- Annahme: konstante Vergleiche

7.1 Selection

- je kleinstes Element entfernen und in Ausgabe pushen: $\mathcal{O}(n^2)$

7.2 Insertion

- je Element nach links bewegen bis es kleiner-gleich ist. Dann für nächstes wiederholen: $\mathcal{O}(n^2)$ oder best-case $\mathcal{O}(n)$

7.3 Bubble

- Paare von hinten durchgehen, je swappen wenn größer: $\mathcal{O}(n^2)$ oder best-case $\mathcal{O}(n)$

7.4 Merge

- List halbieren, rekursiv beide Listenhälften merge-sorten und mergen: $\mathcal{O}(n \log n)$ weil Master-Theorem
- merge sollte nicht kopieren (also doof in funktionalen Sprachen)

7.5 Heap

- Heap aus Zahlen erstellen, je `ExtractMax` anwenden: $\mathcal{O}(n \log n)$ worst und best

7.6 Quick

- je Pivotelement (e.g. Median) wählen, dann kleiner/größer Pivot-Hälfte rekursiv zusammenfügen: $\mathcal{O}(n^2)$ oder best-case $\mathcal{O}(n)$
- durch random Pivotelement ist worst-case $\mathcal{O}(n \log n)$ zu erwarten

7.7 Counting

- array mit buckets, sortieren nach Schlüsselwert; am Ende concat ausgeben: $\mathcal{O}(K + n)$ - besser als vergleichsbasierte Algorithmen

7.8 Radix

- Zahlen werden als Strings vom Ende aus durch Counting-Sort sortiert: $\mathcal{O}(d(n+K))$ mit d als Anzahl der Ziffern
- block-based: je nochmal x buckets

7.9 Bucket

- Werte aus Intervall $[i/n, (i+1)/n[$ werden bucket i zugeordnet; dann jeden Bucket mit Insertion sortieren: $\mathcal{O}(n)$ oder worst-case $\mathcal{O}(n \log n)$

8 Suchen

8.1 Binary

- (Array sortiert); `A[n/2]` betrachten, entsprechend in linker/rechter Hälfte weitersuchen: $\mathcal{O}(\log n)$

8.2 Exponential

- Zweierpotenz-Indizes durchgehen, bei $q < 2^i$ binary auf 2^{i-1} und 2^i

8.3 Binary tree

- links $\leq$ parent, rechts $\geq$ parent: $\mathcal{O}(\log n)$
- rekursiver tree-walk für sort: $\mathcal{O}(n)$
- insert wird obv sehr unbalanced: $\mathcal{O}(n)$
- delete einfach nächstbestes passendes Element als Substitution suchen: $\mathcal{O}(n)$

8.3.1 AVL

- LeftRotation/RightRotation für balancing von Subbäumen (TODO?)
- insert/delete: $\mathcal{O}(\log n)$ da balancing $\mathcal{O}(1)$

9 Gier

- einfach immer das nächstbeste nehmen (Heuristiken)
- häufig nicht optimal, dafür effizient
  - bei Knapsack bspw. doof
  - für Dijkstra, Kruskal, Prim aber auch global optimal

9.1 Lokale Suche

- bei zufälligem Wert starten, lokale Nachbarschaft betrachten. Aufhören, sobald keine bessere Lösung in lokaler Nachbarschaft ist
- mit unterschiedlichen Startwerten wiederholen
- bei TSP durch swappen eigentlich not too bad (also für TSP obv)

9.1.1 Simulated annealing

- am Anfang zu höherer Wahrscheinlichkeit auch schlechtere Lösungen in lokaler Nachbarschaft betrachten (in Abhängigkeit der Differenz)
- Wahrscheinlichkeit ist je $e^{-\Delta/T}$
- eigentlich ganz cool, aber T muss halt passend gewählt werden

10 Dynamik

- Lösen von Problemen durch Kombination der Teilproblemlösungen (bottom-up vs. top-down)

10.1 Knapsack

- Dieb*in will möglichst viel stehlen hehe
- Fälle: Objekt j wird nicht eingepackt: $K(V, j) = K(V, j-1)$
- Fälle: Objekt j wird eingepackt: $K(V, j) = K(V - vol_j, j-1) + val_j$
- bottom-up Array konstruieren mit beiden Fällen und je Maximum nehmen
- top-down bspw. durch rekursive Memoization
- damit $\mathcal{O}(n \cdot V_{\text{total}})$ (abhängig von $V \implies$ Skalierung siehe Approximationsalgorithmen)

11 Backtracking

- bspw. toll bei Queen's problem
- einfach backtracken im Lösungsbaum wenn Teillösung ungültig

12 Branch and Bound

- bound: untere Schranke bestimmen und mit aktueller Lösung vergleichen
- bei TSP ist die untere Schranke bspw. das Gewicht des MST
- verbessert nur durchschittliche Laufzeit

13 Approximationsalgorithmen

- Algorithmus finden, der der tatsächlichen Lösung möglichst nahe kommt
- Güte von Approximation A mit Instanz I entsprechend $\alpha_A = \max_I \frac{\mathrm{A}(I)}{\mathrm{OPT}(I)}$
- Beispiele
  - Set/Vertex cover
  - TSP: Zusätzliche Annahme: Dreiecksungleichung. Dann ist lower bound MST und upper bound zweifacher Graph Durchlauf unter Verwendung der Dreiecksungleichung
    - damit $\alpha_A \leq 2$
  - Knapsack: Volumina skalieren und inten - Rundungsfehler ergibt entsprechend α_A

14 Line

- $\sigma = \sigma_1, ..., \sigma_m$ Sequenz unbekannter σ
- `opt(`σ`)` sind Kosten des besten offline Algorithmus
- `A(`σ`)` sind Kosten des online Algorithmus für σ
- **c-kompetitiv**, wenn für σ gilt $A(\sigma) \leq c\text{opt}(\sigma)$

14.1 Buy-or-rent

- Wie lange Skifahren? Leihen 50, kaufen 500 Euro.
- $\sigma_i = 1 \implies$ skifahren, $\sigma_i = 0 \implies$ nicht skifahren
- $\frac{A(\sigma)}{\text{copt}(\sigma)} = \frac{50(t-1)+500}{50t}$ minimal für $t = 10$
  - also Ski kaufen nach 10 Tagen

14.2 List-update

- move-to-front ist am besten

14.3 Dating

- wie viele Dates bevor Entscheidung?
- Kalibrierungsphase mit t Personen; in Phase 2 entscheiden sobald jemand besser als alle aus Phase 1
- blabla $t = \frac{n}{e}$ also mit ca. 1/3 kalibrieren

15 Closest pair of points

- brute-force ($\mathcal{O}(n^2)$) und sortieren ($\mathcal{O}(n \log n)$) beides nicht optimal
- divide and conquer: Raum rekursiv aufteilen, Abstand closest pair je Seite: $\mathcal{O}(n \log n)$
  - problem in der Mitte lässt sich 2d elegant lösen
- randomisierte Lößung: $\sqrt{n}$ zufällige Punkte und paarweise Abstände
  - kleinster Abstand ist δ
  - Hashing, Würfel, blabla effizient (TODO?)

16 KD-Bäume

TODO.