Analyse von Algorithmen im erwarteten Fall

Lehrvortrag

22.10.2021

Malte Schilling

Lernziel der Vorlesung

Die Vorlesung ist als Teil der Vorlesung *Algorithmen und Datenstrukturen* konzipiert.

Voraussetzungen

- Informatik I – *Grundlagen der Programmierung* (grundlegende Programmierparadigma sind bekannt; Programmierung in funktionalen und objektorientierten Programmiersprachen).
- Aus *Algorithmen und Datenstrukturen* wird die Einführung von grundlegenden Datenstrukturen (Listen, Arrays, Bäume, …)
- und von Asymptotischeen Komplexitätsklassen vorausgesetzt.
- Dazu werden Grundlagen Wahrscheinlichkeitsrechnung erwartet.

Ziel und Fokus

Lernziel (aus der Modulbeschreibung): *“Kosten von Berechnungen mathematisch zu modellieren und auszuwerten”*.

Konkret:

- Kennen von Kriterien für die Analyse von Algorithmen.

- Bestimmen der Laufzeit für Best-, Worst- und Average-Case.

Algorithmen

Recap: Algorithmen

Eigenschaften von Algorithmen

- *Inputs* (null oder mehr)
- *Outputs* (einer oder mehr): stehen in einem bestimmten Verhältnis zu den Inputs
- *Begrenztheit*: Ein Algorithmus muss nach einer endlichen Anzahl von Schritten enden.
- *Eindeutigkeit*: Jeder Schritt eines Algorithmus muss genau definiert sein; die auszuführenden Aktionen müssen für jeden Fall streng und eindeutig festgelegt sein.
- *Effektivität*: alle seine Operationen sind so einfach, dass sie genau und in einer endlichen Zeitspanne von jemandem mit Bleistift und Papier ausgeführt werden können.

Definition Algorithmus

Ein Algorithmus ist ein wohldefinierter Rechenvorgang, der einen Wert oder eine Reihe von Werten als Eingabe annimmt und einen Wert oder eine Reihe von Werten als Ausgabe erzeugt. (Cormen et al. 2009)

(Knuth 1997)

Beispiel: Sortieren

Problemstellung

Eingabe: Sequenz von n natürlichen Zahlen $[a_1, \ldots, a_n], a_i \in \mathbb{N}$, z.B.

$$[8, 5, 2, 7, 1, 4, 3, 6]$$

Ausgabe: Geordnete Ausgabesequenz (Umordnung), so dass $a_{\pi_1} \leq a_{\pi_1} \ldots \leq a_{\pi_n}$,

$$[1, 2, 3, 4, 5, 6, 7, 8]$$

Algorithmen

Quicksort, Insertionsort

Vergleich von Algorithmen

Zwei Algorithmen ALG_A und ALG_B sind für ein Problem gegeben. Wie vergleichen wir diese? Was sind die Kriterien für eine Analyse?

- Korrektheit: Prüfen, ob ein Algorithmus immer die korrekte Ausgabe für jede Eingabe liefert.
- Verständlichkeit
- Effizienz: Wieviel Zeit und Speicherplatz wird benötigt?
  - Laufzeit der Algorithmen
  - Speicher-Aufwand

Vorannahme

Wir nehmen Korrektheit als Voraussetzung für die weitere Analyse an.
Unser Fokus in der heutigen Vorlesung liegt auf der Laufzeit-Analyse von Algorithmen.

Beispiel: Lineare Suche

Eingabe: Das zu suchende Element K und ein Input-Array E mit $n > 0$ Einträgen.

Ausgabe: Antwort, ob oder wo K in E enthalten ist.

Beispiel: Lineare Suche

Eingabe: Das zu suchende Element K und ein Input-Array E mit $n > 0$ Einträgen.

$$K = 4, E = [8, 5, 2, 7, 1, 4, 3, 6]$$

Ausgabe: Antwort, ob K in E enthalten ist.

True

Pseudocode

```python
for (int index = 0; index < n; index++):
    if (E[index] == K):
        return true; // Element wurde gefunden
return false; // Fall: wurde nicht gefunden
```

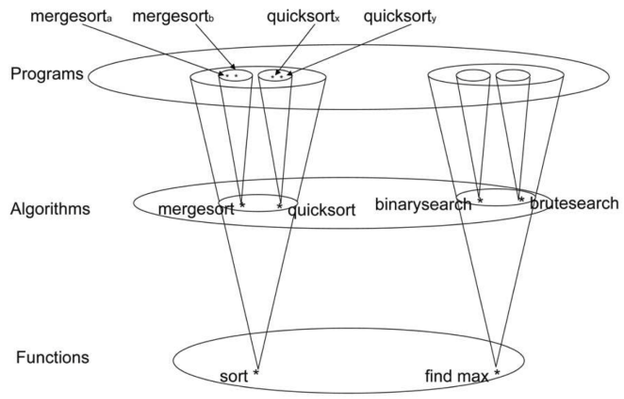
Wir wollen nun die Laufzeit bestimmen, als die Zahl notwendiger Rechenschritte.

Die Beurteilung der Effizienz eines Algorithmus soll dabei unabhängig sein von

- verwendeter Hardware, Computer,
- Programmiersprache.

Abstraktes Modell: *Random Access Machine*

- Ein **Prozessor** arbeitet ein Programm sequentiell ab.
- Alle Daten sind **direkt zugreifbar** im Speicher – und jeder Speicherzugriff dauert gleich lang.
- Jedes Datum paßt in eine Speichereinheit.
- Primitive Operationen benötigen konstante Zeit:
  - Zuweisung
  - arithmetische und logische Operationen
  - Vergleichsoperationen
  - Ablaufsteuerung



(Yanofsky 2010)

Elementare Operationen

Wir sind in der Analyse nicht interessiert an der exakten Zahl von Rechenschritten oder Operationen.

Für die Analyse legen wir elementare Operationen fest, z.B.

- Vergleich von zwei Zahlen bei der linearen Suche oder beim Sortieren
- Anzahl von Multiplikationen bei Matrixmultiplikation

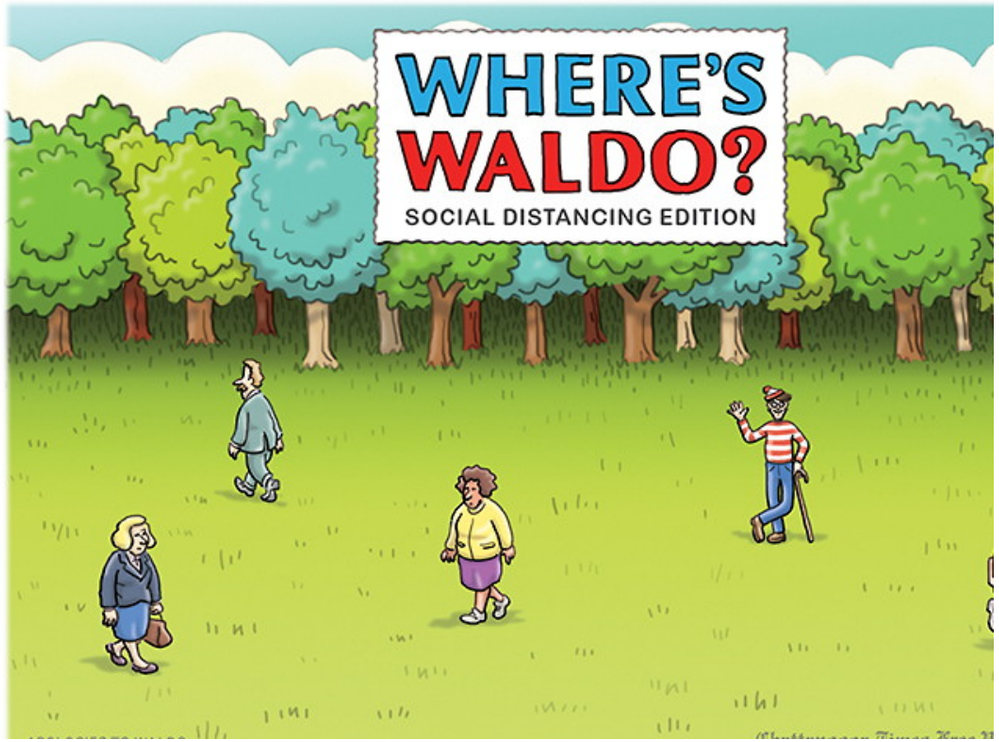
Elementare Operationen

- Anzahl elementare Operationen soll gut die Gesamtzahl Operationen abschätzen.
- Anzahl der elementaren Operationen wird als Basis für die Bestimmung des Wachstums der Zeitkomplexität genommen.

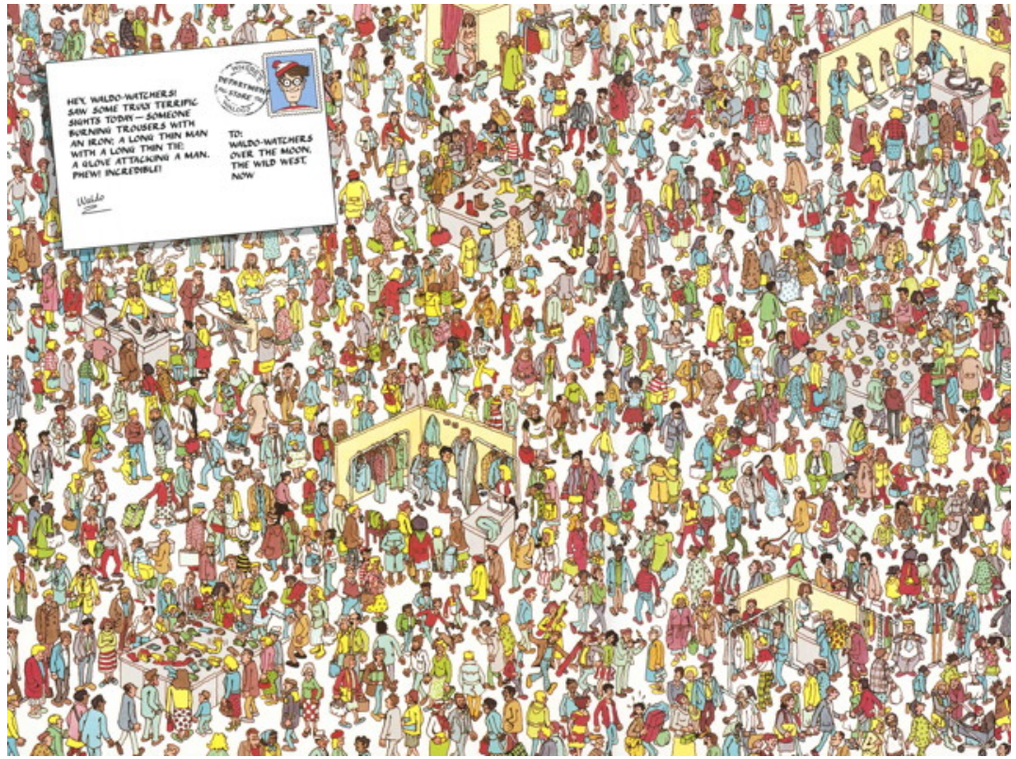
Abhängigkeit der Laufzeit von Eingabegröße

Ferner hängt die Laufzeit auch von der Eingabegröße ab.

Kleiner Suchraum



Großer Suchraum



https://mymodernmet.com/wheres-waldo-social-distancing-cartoon/

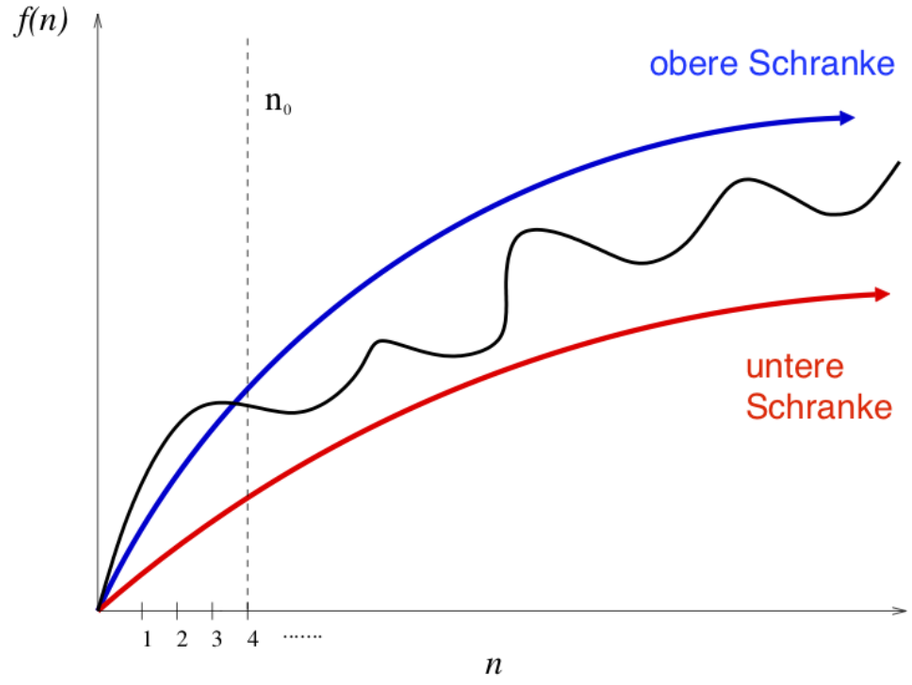
Effizienz von Algorithmen

Begriff der Laufzeit

Laufzeit

Wir betrachten die Laufzeit

- als Funktion über die Eingabegröße, Eingabelänge
- bezogen auf serielle Maschinen
- und damit abstrahierend über konkreter Zeitdauer und stattdessen bezogen auf eine festgelegte elementare Operation.



(Katoen 2020)

Analyse der Laufzeit von Algorithmen

$D_n =$ Menge aller Eingaben der Länge n

$t(I) =$ für Eingabe I benötigte Anzahl elementarer Operationen

Beispiel lineare Suche

Elementare Operation: Vergleich einer Zahl K mit dem Element $E[index]$

Inputs:

$K = 4$

$D_8 = \{[8,5,2,7,1,4,3,6],$
$[1,2,3,4,5,6,7,8], \dots\}$

Output: $True$

Algorithmus

```Python
for (int index = 0; index < n; index++):
    if (E[index] == K):
        return true; // Element wurde gefunden
return false; // Fall: wurde nicht gefunden
```

Laufzeit

Für jeden Input individuell nun bestimmbar.

Wählen von bestimmten Instanzen: Best-Case Fall

Best-Case Szenario

Gegeben durch die minimal benötigte Anzahl von Operationen für eine Eingabe der Länge n:

$$B(n) = \mathbf{min}(t(I)|I \in D_n)$$

Beispiel

Wir suchen ein Element von D_n, für das $\mathbf{min}(t(I)|I \in D_n)$.

Input: $K = 4, D_8 = \{[8,5,2,7,1,4,3,6], [1,2,3,4,5,6,7,8], \dots\}$

```python
for (int index = 0; index < n; index++):
    if (E[index] == K):
        return true; // Element wurde gefunden
return false; // Fall: wurde nicht gefunden
```

Für $I = [4,1,2,3,5,6,7,8] \in D_8$ benötigen wir $t(I) = 1$ Vergleich.

Allgemein: Mit $I = [K, \dots]$ bleibt $t(I) = 1$ konstant. Und damit ist $B(n) = 1$.

Wählen von bestimmten Instanzen: Worst-Case Fall

Worst-Case Szenario

Gegeben durch die maximal benötigte Anzahl von Operationen für eine Eingabe der Länge n.

$$W(n) = \mathbf{max}(t(I)|I \in D_n)$$

Beispiel

Wir suchen ein Element von D_n, für das $\mathbf{min}(t(I)|I \in D_n)$.

Input: $K = 4, D_8 = \{[8,5,2,7,1,4,3,6], [1,2,3,4,5,6,7,8], \dots\}$

```python
for (int index = 0; index < n; index++):
    if (E[index] == K):
        return true; // Element wurde gefunden
return false; // Fall: wurde nicht gefunden
```

Für $I = [1,2,3,5,6,7,8,4] \in D_8$ benötigen wir $t(I) = 8$ Vergleich.

Allgemein: Mit $I = [\dots, K]$ benötigen wir $t(I) = n$ Vergleiche. Und damit ist $W(n) = n$.

Einschub: Erwartungswert

Erwartungswert

$$E(X) = x_1 * P(X = x_1) + x_2 * P(X = x_2) + \ldots + x_N * P(X = x_N) = \sum_{i=1}^{n} x_i P(X = x_i)$$

mit X Zufallsvariable, x_i sind die Ausprägungen der Variable und $P(X = x_i)$ jeweils die dazugehörende Wahrscheinlichkeit.

Beispiel: 2-facher Münzwurf

Bei einem Glücksspiel wird eine Münze zweimal hintereinander geworfen. Wenn einmal *Kopf* erscheint, gewinnen wir 1 Euro, wenn beidemale *Kopf* erscheint, gewinnen wir 5 Euro. Wie hoch ist unser erwarteter Gewinn, wenn wir einmal spielen?

Wir haben vier mögliche Ausgänge:

	x_1	x_2	x_3	x_4
Abfolge	$[K, K]$	$[K, Z]$	$[Z, K]$	$[Z, Z]$
WK $P(x_i)$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
Gewinn	5	1	1	0

Erwartungswert

$$E(X) = \sum_{i=1}^{4} x_i P(X = x_i) = \frac{1}{4} \cdot 5 + \frac{1}{4} \cdot 1 + \frac{1}{4} \cdot 1 + \frac{1}{4} \cdot 0 = \frac{7}{4}$$

$$E(X) = \sum_{i=1} x_i P(X = x_i) = \frac{}{4} * 5 + \frac{}{4} * 1 + \frac{}{4} * 1 + \frac{}{4} * 0 = \frac{}{4}$$

Analyse für den Erwarteten Fall

Average-Case Szenario

Die Average-Case Laufzeit ist die von dem Algorithmus durchschnittlich benötigte Anzahl elementarer Operationen auf einer beliebigen Eingabe der Länge n.

$$A(n) = \sum_{I \in D_N} P(I) * t(I)$$

$P(I) =$ Wahrscheinlichkeit, dass Eingabe I auftritt

(Ottmann and Widmayer 2012)

Average-Case für lineare Suche

Erwartungswert: $\frac{n}{2}$ erscheint intuitiv, ist aber abhängig davon, ob das Element enthalten ist.

Unterscheidung zwei Fälle

- K kommt in E vor oder
- K kommt nicht in E vor.

```Python
for (int index = 0; index < n; index++):
    if (E[index] == K):
        return true; // Oder: index ausgeben
return false; // Fall nicht gefunden
```

- Annahme: alle Elemente in E sind unterschiedlich und gleich wahrscheinlich verteilt.
- Dann ist die Wahrscheinlichkeit für $K == E[i]$: $P(K == E[i]) = \frac{1}{n}$
- Die Anzahl benötigter Vergleiche im Fall $K == E[i]$ ist $t(E) = i + 1$ für $i \geq 0$.

Damit ergibt sich als average-case:

$$
\begin{aligned}
A_{K \in E}(n) &= \sum_{i=0}^{n-1} P(K == E[i] | \text{K in E}) * t(K == E[i]) \\
&= \sum_{i=0}^{n-1} \left(\frac{1}{n}\right) * (i+1) \\
&= \left(\frac{1}{n}\right) \sum_{i=0}^{n-1} (i+1) \\
&= \left(\frac{1}{n}\right) * \frac{n(n+1)}{2} \\
A_{K \in E}(n) &= \frac{n+1}{2}
\end{aligned}
$$

Average-Case für lineare Suche

Insgesamt ergibt sich damit:

$$A(n) = P(\text{K in E}) * A_{K \in E}(n) + P(\text{K nicht in E}) * A_{K \notin E}(n)$$

Average-Case für lineare Suche

Insgesamt ergibt sich damit:

$$
\begin{aligned}
A(n) &= P(\text{K in E}) * A_{K \in E}(n) + P(\text{K nicht in E}) * A_{K \notin E}(n) \\
&= P(\text{K in E}) \frac{n+1}{2} + P(\text{K nicht in E}) * A_{K \notin E}(n) \\
&= P(\text{K in E}) \frac{n+1}{2} + (1 - P(\text{K in E})) * A_{K \notin E}(n) \\
&= P(\text{K in E}) \frac{n+1}{2} + (1 - P(\text{K in E})) * n \\
&= P(\text{K in E}) \frac{-n+1}{2} + n
\end{aligned}
$$

Beispiele

Für $P(\text{K in E})$

- $= 1 : A(n) = \frac{n+1}{2}$, wie vorher hergeleitet.

- $= 0 : A(n) = n = W(n)$, da E komplett überprüft werden muss.
- $= \frac{1}{2} : A(n) = \frac{3n+1}{4}$, ungefähr 75% aller Elemente von E werden verglichen.

Überblick: Laufzeit-Analyse für die Lineare Suche

Laufzeit-Analyse:

- Die Beurteilung der Effizienz eines Algorithmus soll unabhängig sein von verwendeter Hardware, Computer, Programmiersprache.
- Wir sind in der Analyse nicht interessiert an der exakten Zahl von Rechenschritten oder Operationen. Für die Analyse legen wir elementare Operationen fest
- Wir betrachten die Laufzeit eines Algorithmus nicht als eine Zahl, sondern als eine Funktion abhängig von der Eingabelänge.
- Wir sind dabei nicht an der exakt benötigten Zeit interessiert, sondern der asymptotischen Performanz – dem Wachstum der Laufzeit abhängig von der Eingabe.

Elementare Operation

Vergleich einer Zahl K mit dem Element $E[index]$ aus z.B. $E = [8, 5, 2, 7, 1, 4, 3, 6]$.

Laufzeit:

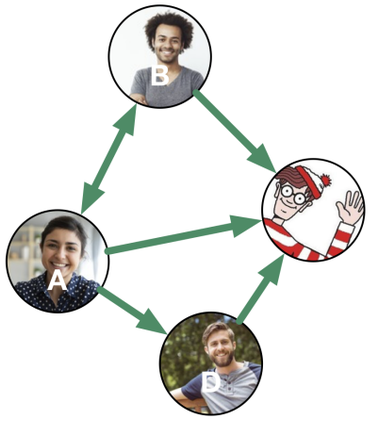
$W(n) = n \in \mathcal{O}(n)$; $B(n) = 1 \in \mathcal{O}(1)$; $A(n) = P(\text{K in E})\frac{-n+1}{2} + n \in \mathcal{O}(n)$

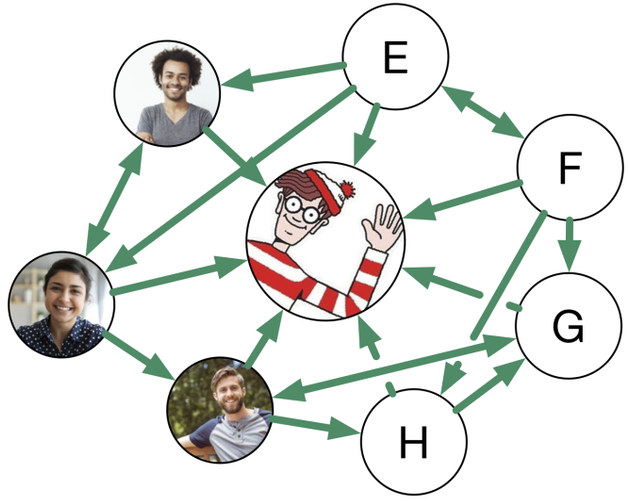
Beispiel Analyse: Where is Waldo?

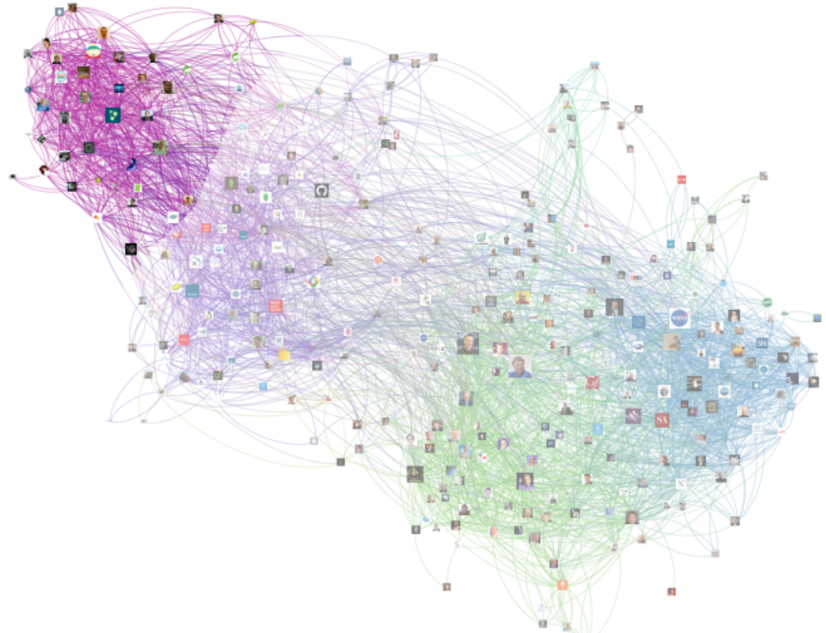
Beispiel: Waldo auf twitter

Als Beispiel: Wir suchen in twitter Waldo.

Unsere Annahme: Waldo ist prominent – er ist jemand dem alle folgen und von dem alle immer News bekommen möchten.





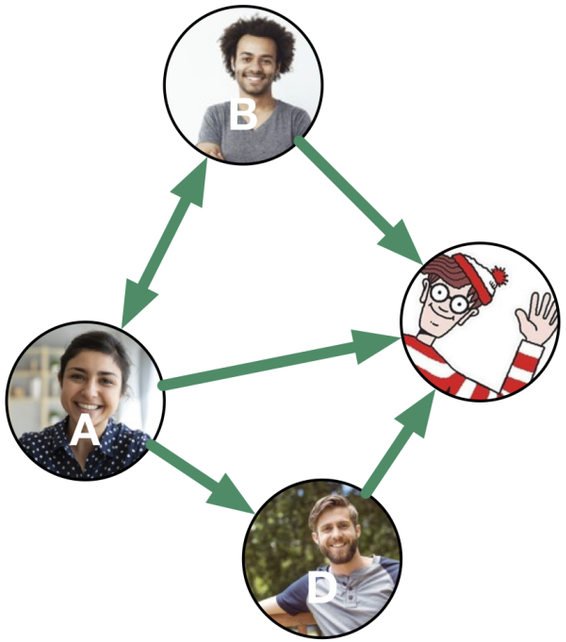


http://allthingsgraphed.com/2014/11/02/twitter-friends-network/

Prominenten Beispiel

Ziel: Finde die prominente Person

Einer prominenten Person (celebrity) folgen alle auf twitter.



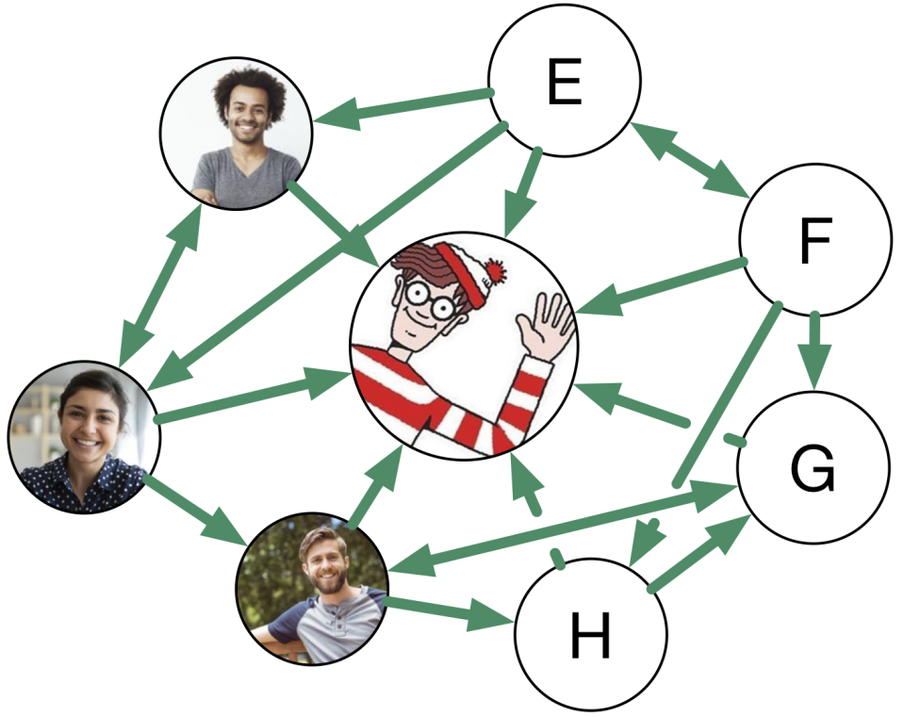
Matrix-Darstellung

$$K_{ij} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Eingabe:

- $n \times n$ Verbindungsmatrix, bei $n \in \mathbb{N}$ Personen

Prominenten Beispiel



Matrix-Darstellung

$$K_{ij} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Naiver Ansatz für einen Algorithmus

Für jeden Account einzeln:

- Testen, ob diesem alle folgen.

Formalisierung Prominentenbeispiel

Eingabe

- $n \in \mathbb{N}$ Personen,
- von diesen ist genau eine Person ein Prominenter,
- $n \times n$ Verbindungsmatrix, so dass für $0 \leq i, j \leq n$:

$$K[i,j] = \begin{cases} 1 & \text{wenn Person } i \text{ Person } j \text{ folgt} \\ 0 & \text{ansonsten} \end{cases}$$

Ausgabe

Person k wird als prominente Person zurück gegeben, genau dann wenn

$$0 \leq i < n, i \neq k \Rightarrow K[i,k] = 1$$

Komplexität des Prominentenproblems

Für den naiven Ansatz wird für jede der n Personen einzeln getestet, ob ihm alle folgen ($n-1$ Vergleiche mit allen anderen Personen).

- Wie ist die Laufzeit für den Worst-Case für diesen Algorithmus?
- Wie ist die Laufzeit für den Average-Case für diesen Algorithmus?
- Können wir einen besseren Algorithmus finden?

Antworten (interaktiv in zoom anotieren)

	$\in \mathcal{O}(1)$	$\in \mathcal{O}(\log n)$	$\in \mathcal{O}(n)$	$\in \mathcal{O}(n^2)$	$\in \mathcal{O}(2^n)$
Worst-Case					
Best-Case					
Average-Case					

Programmieraufgabe: Naiver Ansatz Prominentenproblem

Initialisierung Python Umgebung.

```
import numpy as np
import matplotlib.pyplot as plt

K_list = []
for i in range(2,20):
    K = np.random.randint(2, size=(i,i))
    for j in range(0,i):
        K[j,j] = 1
    set_promi = np.random.randint(i)
    K[set_promi,:] = 0
    K[:,set_promi] = 1
    K_list.append(K)

comp_list = []
for K in K_list:
    promi = -1
    comparison_count = 0
    for i in range(0, K.shape[0]):
        promi_found = True
        for j in range(0, K.shape[0]):
            if K[j,i]==0:
```

Verbessern Ansatz

Für den naiven Ansatz wird für jede der n Personen einzeln getestet, ob ihm alle folgen ($n-1$ Vergleiche mit allen anderen Personen).

Damit ergibt sich für den Worst-Case: $W(n) \in \mathcal{O}(n^2)$.

Average-Case

Für die gegebenen geschachtelten Schleifen ist $A(n) = W(n)$.

- Wie kann die Average-Case Laufzeit einfach verbessert werden?
- Was ergibt sich dann für eine Average-Case Laufzeit nach $A(n) = \sum_{I \in D_N} P(I) * t(I)$?
- Und was für eine Best- und Worst-Case Laufzeit?

Verbesserter Algorithmus

- Durchlaufe jede Spalte einzeln (also n Spalten – äußere for-Schleife),
- von oben nach unten (für $j = 0$ bis maximal $j = n - 1$).

```python
for K in K_list:
  promi = -1
  comparison_count = 0
  for i in range(0, K.shape[0]):
    promi_found = True
    for j in range(0, K.shape[0]):
      if K[j,i]==0:
        promi_found = False
        break
      comparison_count += 1
    if promi_found:
      promi = i
```

Laufzeiten: $W(n)$? $A(n)$? $B(n)$?

$A(n) = \sum_{I \in D_N} P(I) * t(I)$

Summary und Einordnung

Zusammenfassung

Kriterien für die Analyse von Algorithmen

- Korrektheit: Prüfen, ob ein Algorithmus immer die korrekte Ausgabe für jede Eingabe liefert.
- Wieviel Zeit und Speicherplatz wird benötigt?
  - Zeitkomplexität: benötigte Zeit
  - Platzkomplexität: benötigte Platz
- Verständlichkeit

Laufzeitanalyse

- Die Komplexität wird immer betrachtet im Verhältnis zu einer bestimmten Eingabe.
- Wir sind dabei nicht an der exakt benötigten Zeit interessiert, sondern der asymptotischen Performanz – dem Wachstum der Laufzeit abhängig von der Eingabe.
- Wir unterscheiden Worst-, Best- und Average-Case Szenarien.

- Betrachten von *Komplexität eines Problems* an sich: Wie gut kann ein Algorithmus sein? Was ist seine optimale Laufzeit?
  - für lineare Suche $\in O(n)$,
  - aber unter zusätzlichen Bedingungen kann dies verbessert werden, z.B. in geordneten Listen ist die Suche $\in O(\log n)$.
- Für sehr teure Rechenprobleme müssen wir die Korrektheit der Lösung evtl. aufgeben und betrachten dann zusätzlich ein *Mass für die Qualität der Ausgabe*.
- Vertiefen der Bedeutung des Average-Case Szenarios: Die Szenarien unterscheiden sich in der Auswahl der Instanzen, die herangezogen werden für die Laufzeitbestimmung. In komplexeren Problemen kann so die Worst-Case Sicht zu sehr auf einzelne (unwahrscheinliche) Fälle fokussieren. Vertiefen anhand des Sortierproblems
  - Insertion Sort als einführendes Beispiel
  - QuickSort als Beispiel für eine verbesserte Average-Case Laufzeit

Algorithm 1 Quicksort

```
1: procedure QUICKSORT(A, p, r)
2:     if p < r then
3:         q = PARTITION(A, p, r)
4:         QUICKSORT(A, p, q − 1)
5:         QUICKSORT(A, q + 1, r)
6:     end if
7: end procedure
8: procedure PARTITION(A, p, r)
9:     x = A[r]
10:    i = p − 1
11:    for j = p to r − 1 do
12:        if A[j] < x then
13:            i = i + 1
14:            exchange A[i] with A[j]
15:        end if
16:        exchange A[i] with A[r]
17:    end for
18: end procedure
```

References

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition*. MIT Press. http://mitpress.mit.edu/books/introduction-algorithms.

Katoen, Joost-Pieter. 2020. “Datenstrukturen Und Algorithmen.” Lecture Notes, RWTH Aachen.

Knuth, Donald E. 1997. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Third. Reading, Mass.: Addison-Wesley.

Nebel, Markus. 2012. *Entwurf Und Analyse von Algorithmen: Markus Nebel*. Studienbücher Informatik. Wiesbaden: Vieweg & Teubner. http://d-nb.info/1018322566/04.

Ottmann, Thomas, and Peter Widmayer. 2012. *Algorithmen Und Datenstrukturen, 5. Auflage.* Spektrum Akademischer Verlag.

Yanofsky, Noson S. 2010. “Towards a Definition of an Algorithm.” *Journal of Logic and Computation* 21 (2): 253–86. doi:10.1093/logcom/exq016.