Algorithmen und Datenstrukturen (für ET/IT)

Wintersemester 2012/13

Dr. Tobias Lasser

Computer Aided Medical Procedures
Technische Universität München

Notizen

Programm heute

Notizen

Algorithmus: Dijkstra

Input: Graph $G = (V, E)$, $w : E \to \mathbb{R}^+$, Startknoten $s \in V$
Output: Vorgänger-Liste pred, Distanz-Markierung d
Dijkstra(G, w, s):

```
for each (Knoten v ∈ V) { // Initialisierung
  pred[v] = NULL; d[v] = ∞;
}
d[s] = 0;
Q = Priority Queue mit Elementen V, Schlüsseln d;
while ( !Q.isEmpty() ) { // Hauptschleife
  u = Q.extractMin();
  for each (v ∈ adj[u] mit v ∈ Q) {
    if (d[u] + w(u,v) < d[v]) {
      pred[v] = u; d[v] = d[u] + w(u,v);
      Q.decreaseKey(v, d[v]);
    }
  }
}
```

Notizen

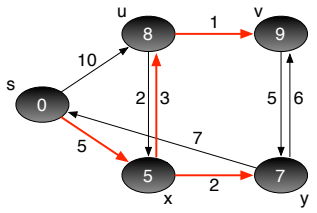
Dijkstra-Algorithmus

Nach Ausführung von **Dijkstra**(G, w, s) gilt für $v \in V$:

- d[v] = Gewicht $w_{min}(v, s)$ des kürzesten Pfades von v zu s
- pred[v] = Vorgängerknoten
- Kürzester Pfad von v zu s:

  pred[v], pred[pred[v]], ..., s

Beispiel:



Notizen

Dijkstra: Laufzeit

Laufzeit:

- Annahme: Q implementiert als binärer Min-Heap
- Zeile 1-3: $O(|V|)$
- Zeile 5: entspricht buildMinHeap, also $O(|V|)$
- Zeile 6-15: Ausführung $|V|$ mal
  - Zeile 7: $O(\log |V|)$
  - Zeile 8-14: Ausführung inkl. äusserer while-Schleife: insgesamt $|E|$ mal (siehe DFS/BFS)
  - Zeile 12: $O(\log |V|)$

Gesamt: $O((|V| + |E|) \log |V|)$

einfacher: $O(|E| \log |V|)$

```
Dijkstra(G, w, s):
1    for each (Knoten v ∈ V) {
2      pred[v] = NULL; d[v] = ∞;
3    }
4    d[s] = 0;
5    Q = Priority Queue(V, d);
6    while ( !Q.isEmpty() ) {
7      u = Q.extractMin();
8      for each (v ∈ adj[u] mit v ∈ Q) {
9        if (d[u] + w(u, v) < d[v]) {
10         pred[v] = u;
11         d[v] = d[u] + w(u, v);
12         Q.decreaseKey(v, d[v]);
13       }
14     }
15   }
```


Notizen

Dijkstra: Komplexität

- Komplexität des Dijkstra Algorithmus hängt entscheidend von der Implementierung der Priority Queue ab!
- Varianten:
  - als verkettete Liste: $O(|V|^2)$
  - als binärer Heap: $O(|E| \log |V|)$
  - als Fibonacci Heap: $O(|E| + |V| \log |V|)$

Notizen

Dijkstra: Korrektheit

(nur Beweisidee, dies ist kein formaler Beweis!)

Annahme: bisherige Iterationen waren korrekt, bisher bearbeitete Knoten: $X \subset V$

- nächster Iterationsschritt nimmt kürzeste direkte Verbindung von Knoten $x \in X$ zu noch nicht bearbeitetem Knoten $y \in V \setminus X$ hinzu
- d[y] ist nun d[x] + $w(x, y)$
- jeder andere Pfad zu y hat entweder
  - eine Kante, die aus X heraus geht und ist damit nicht kürzer als (x, y)
  - mehrere Kanten, und ist damit nicht kürzer als d[y], da die Kanten positives Gewicht haben

Entscheidende Annahme: Kanten haben positives Gewicht!

Notizen

Dijkstra: Anwendungen

Dijkstra ist einer der am häufigsten verwendete Graph-Algorithmen

Beispiele:

- Routenplanung in GIS (Geographic Information System)
  - Navigationssystem im Auto
  - Maps Applikation (Google, Bing, Apple etc.)
  - Routen mit Flugzeugen, Bahn usw.
- Routing Protokolle für IP Netzwerke
  - z.B. Open Shortest Path First
- Pfadplanung von Robotern, UAV/Dronen, etc.
- Segmentierung von medizinischen Bilddaten

Notizen

Programm heute

Notizen

Minimaler Spannbaum

Sei $G = (V, E)$ zusammenhängender Graph mit Gewichtsfunktion $w : E \to \mathbb{R}$.

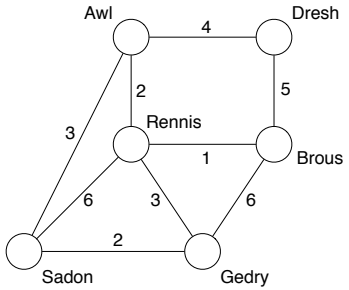
- Spannbaum: Teilgraph $G' = (V, E')$, der ein Baum ist und alle Knoten von G enthält.
- minimaler Spannbaum: Spannbaum G' mit minimalem Gewicht

$$w_{G'} = \sum_{(x,y) \in E'} w(x, y)$$

Notizen

Minimaler Spannbaum: Beispiel

Beispiel: 6 (virtuelle) Städte und Kosten für Strassenbau dazwischen (in Million Euro):



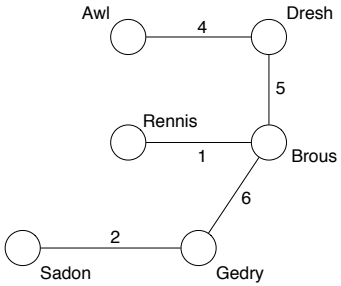
Problem: Strassenbau mit minimalen Kosten, so daß alle Städte verbunden sind (direkt oder über andere Städte)

Lösung: minimaler Spannbaum

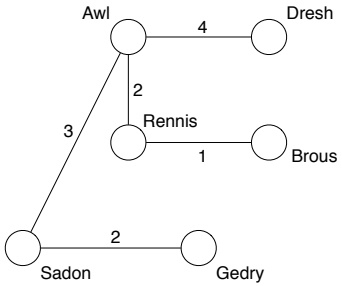
Notizen

Minimaler Spannbaum: Beispiel 2

Mögliche Lösungen:



Gewicht: 18



Gewicht: 12

Notizen

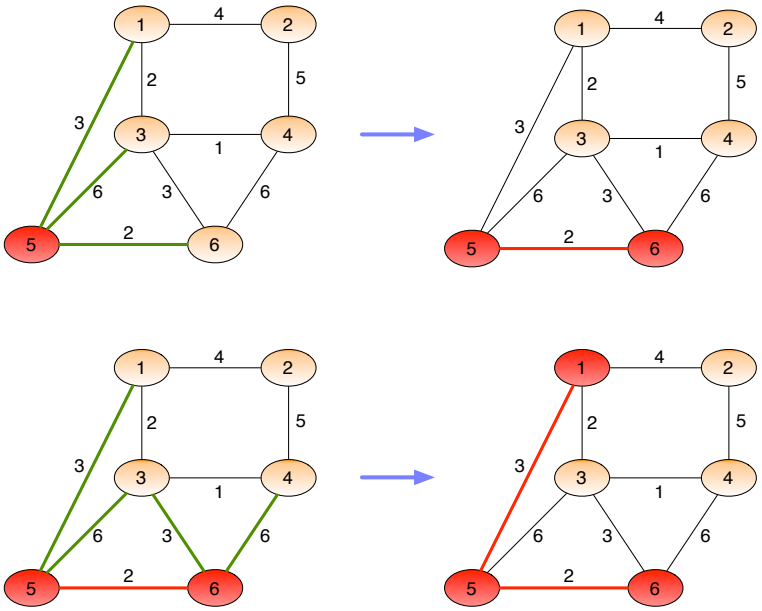
Minimaler Spannbaum: Algorithmen

Sei $G = (V, E)$ zusammenhängender Graph mit Gewichtsfunktion $w : E \to \mathbb{R}$.

- Minimaler Spannbaum von G:
  - Algorithmus von Kruskal: Greedy-Algorithmus
    Komplexität: $O(|E| \log |V|)$
  - Algorithmus von Prim: Greedy-Algorithmus
    Komplexität: $O(|E| \log |V|)$
  - viele Varianten davon als parallele Algorithmen

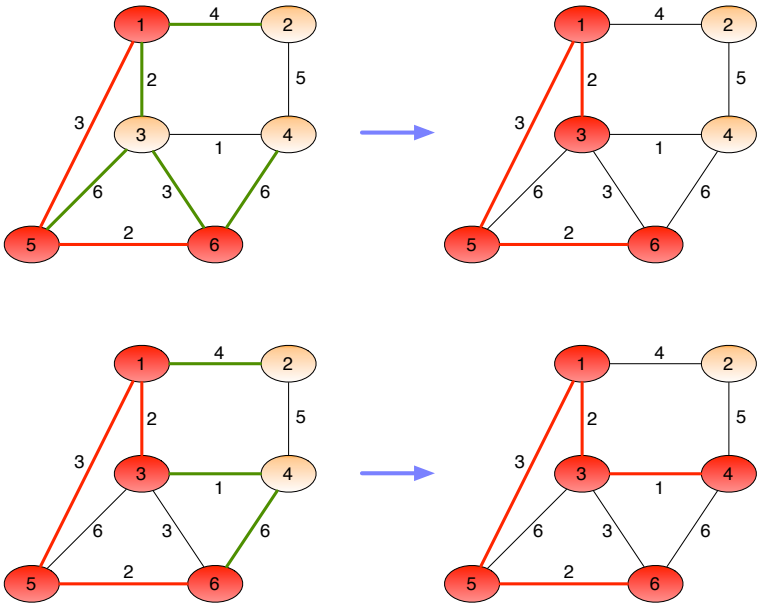
Notizen

Beispiel-Ablauf: Prim Algorithmus 1



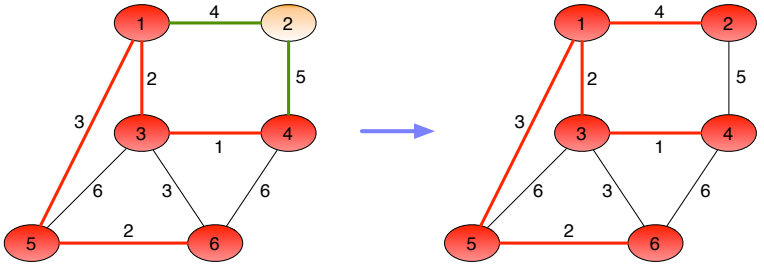
Notizen

Beispiel-Ablauf: Prim Algorithmus 2



Notizen

Beispiel-Ablauf: Prim Algorithmus 3



Notizen

Beobachtungen:

- Zwischenlösungen von Prim Algorithmus sind Bäume
- es werden öfters mehrere Kanten zum selben Knoten betrachtet (s. oben)
  - Vereinfachung: betrachte nur Kante mit minimalem Gewicht

Prim Algorithmus

Sei $G = (V, E)$ zusammenhängender Graph mit Gewichtsfunktion $w : E \to \mathbb{R}$.

- Startknoten $s \in V$ für minimalen Spannbaum
- Graph repräsentiert als Adjazenzliste adj
- jeder Knoten (ausser s) hat Vorgänger im Spannbaum pred
- jeder Knoten hat Markierung g
  - kleinstes Gewicht um Knoten mit aktuellem Spannbaum zu verbinden
- Hilfsmittel: Priority Queue Q

Notizen

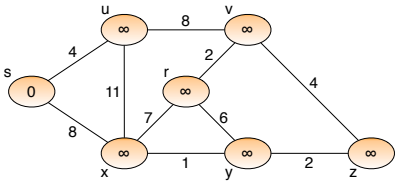
Algorithmus: Prim

Input: Graph $G = (V, E)$, $w : E \to \mathbb{R}$, Startknoten $s \in V$
Output: Vorgänger-Liste pred
Prim(G, w, s):

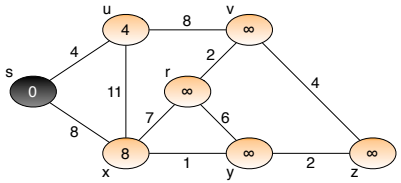
```
for each (Knoten v ∈ V) { // Initialisierung
  pred[v] = NULL; g[v] = ∞;
}
g[s] = 0;
Q = Priority Queue mit Elementen V, Schlüsseln g;
while ( !Q.isEmpty() ) { // Hauptschleife
  u = Q.extractMin();
  for each (v ∈ adj[u] mit v ∈ Q) {
    if (w(u, v) < g[v]) {
      pred[v] = u; g[v] = w(u, v);
      Q.decreaseKey(v, g[v]);
    }
  }
}
```

Notizen

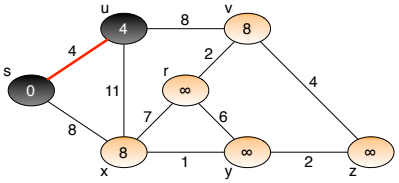
Beispiel: Ablauf Prim Algorithmus 1



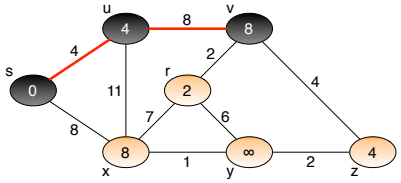
Q: (s,0) | (u, ∞) | (v, ∞) | (r, ∞) | (x, ∞) | (y, ∞) | (z, ∞)



Q: (u, 4) | (x, 8) | (r, ∞) | (v, ∞) | (y, ∞) | (z, ∞)



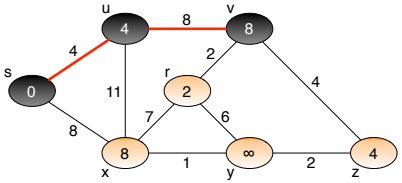
Q: (v, 8) | (x, 8) | (r, ∞) | (y, ∞) | (z, ∞)



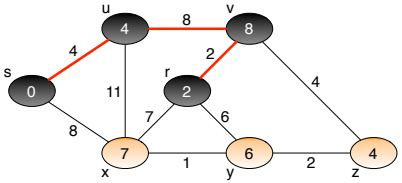
Q: (r, 2) | (z, 4) | (x, 8) | (y, ∞)

Notizen

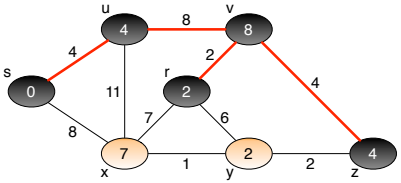
Beispiel: Ablauf Prim Algorithmus 2



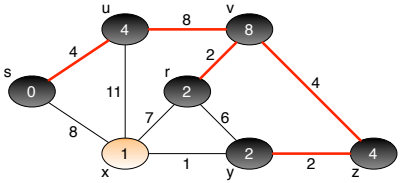
Q: (r, 2) | (z, 4) | (x, 8) | (y, ∞)



Q: (z, 4) | (y, 6) | (x, 7)



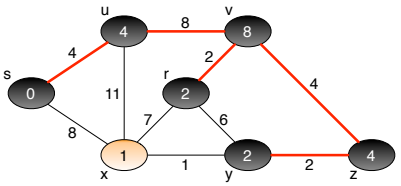
Q: (y, 2) | (x, 7)



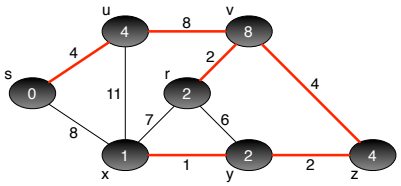
Q: (x, 1)

Notizen

Beispiel: Ablauf Prim Algorithmus 3



Q: (x, 1)



Q:

Nach Ausführung von **Prim**(G, w, s) gilt für $v \in V$:

- pred[v] = Vorgängerknoten im Spannbaum
- Pfad in Spannbaum von v zu Wurzel s:

  pred[v], pred[pred[v]], ..., s

Notizen

Prim: Laufzeit

Laufzeit:

- Annahme: Q implementiert als binärer Min-Heap
- Zeile 1-3: $O(|V|)$
- Zeile 5: entspricht buildMinHeap, also $O(|V|)$
- Zeile 6-14: Ausführung $|V|$ mal
  - Zeile 7: $O(\log |V|)$
  - Zeile 8-13: Ausführung inkl. äusserer while-Schleife: insgesamt $|E|$ mal (siehe DFS/BFS)
  - Zeile 11: $O(\log |V|)$

Gesamt: $O((|V| + |E|) \log |V|)$

einfacher: $O(|E| \log |V|)$

Prim(G, w, s):

```
1    for each (Knoten v ∈ V) {
2      pred[v] = NULL; g[v] = ∞;
3    }
4    g[s] = 0;
5    Q = Priority Queue(V, g);
6    while ( !Q.isEmpty() ) {
7      u = Q.extractMin();
8      for each (v ∈ adj[u] mit v ∈ Q) {
9        if (w(u, v) < g[v]) {
10         pred[v] = u; g[v] = w(u, v);
11         Q.decreaseKey(v, g[v]);
12       }
13     }
14   }
```


Notizen

Prim: Komplexität

- Komplexitätsanalyse von Prim fast identisch mit Dijkstra!
- Komplexität des Algorithmus von Prim hängt entscheidend von der Implementierung der Priority Queue ab!
- Varianten:
  - als verkettete Liste: $O(|V|^2)$
  - als binärer Heap: $O(|E| \log |V|)$
  - als Fibonacci Heap: $O(|E| + |V| \log |V|)$

Notizen

Prim: Anwendungen

- Planung von Netzwerken
  - Strassennetz
  - Kommunikations-Netzwerk
  - elektronische Schaltungen
- Clustering von Daten
  - Daten als Knoten, "Nähe" als Kanten, entferne "lange" Kanten aus minimalem Spannbaum $\rightarrow$ Clustering
- Extrahieren/Tracking von Objekten aus Bildern in Computer Vision

Notizen

Ausblick: Graphen-Algorithmen

- Fluss in Graphen: statt Kantengewichten gibt es Kapazitäten, betrachtet wird Fluss von Quelle zu Senke
  - Problem: finde maximalen Fluss
  - Anwendung: z.B. Fluss in Kommunikations-Netzwerken
- Einfärben von Graphen: Färbe Knoten von Graph so ein, dass keine benachbarten Knoten diesselbe Farbe haben
  - Anwendungen: z.B. Scheduling, Sudoku
- Planare Graphen: lässt sich Graph ohne Kanten-Überschneidung zeichnen?
  - Anwendung: z.B. Chip- bzw. Platinen-Design
- Klassifikation von medizinischen Daten: über analytische Operationen auf Adjazenzmatrix, z.B. Laplace-Operator
  - Anwendungen: z.B. Identifikation von Melanomen, Tracking von Endoskopen

Notizen

Programm heute

Notizen

What is the matrix?

Was ist eine Matrix?

- Anordnung von Zahlen $(a_{ji}) \subset \mathbb{R}$ in einem $m \times n$ Muster:

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} =: A$$

- Element des Vektorraumes $\mathbb{R}^{m \times n}$

$$A \in \mathbb{R}^{m \times n}$$

- Lineare Abbildung $f : \mathbb{R}^n \to \mathbb{R}^m$ mit

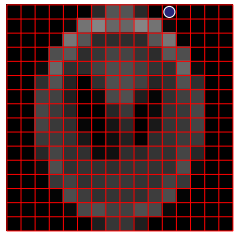
$$f(x) = Ax$$

wobei A $m \times n$ Matrix.

Notizen

Beispiel: Anwendung von Matrizen

- Adjazenzmatrix von Graphen
  - effizienter als Adjazenzlisten für dichte Graphen (viele Kanten)
  - erlaubt analytische Operationen wie Laplace-Operator/Eigenwerte
- Bilder im Computer: gespeichert als Matrix



Notizen

Speicherung von Matrizen

Speicherung als sequentielle Liste / Array:

- row-major: Zeilen werden zuerst durchlaufen

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \quad \Rightarrow \quad [a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33}]$$

- column-major: Spalten werden zuerst durchlaufen

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \quad \Rightarrow \quad [a_{11}, a_{21}, a_{31}, a_{12}, a_{22}, a_{32}, a_{13}, a_{23}, a_{33}]$$

Notizen

Matrix-Operationen

Seien $A, B \in \mathbb{R}^{m \times n}$ mit $A = (a_{ji})$, $B = (b_{ji})$ und $\lambda \in \mathbb{R}$.

- Addition:

$$A + B = \begin{pmatrix} a_{11} + b_{11} & \cdots & a_{1n} + b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & \cdots & a_{mn} + b_{mn} \end{pmatrix}$$

- Skalarmultiplikation:

$$\lambda A = \begin{pmatrix} \lambda a_{11} & \cdots & \lambda a_{1n} \\ \vdots & \ddots & \vdots \\ \lambda a_{m1} & \cdots & \lambda a_{mn} \end{pmatrix}$$

Notizen

Matrix-Operationen (Fortsetzung)

Seien $A = (a_{ji}) \in \mathbb{R}^{m \times n}$, $x = (x_i) \in \mathbb{R}^n$ und $B = (b_{ji}) \in \mathbb{R}^{n \times r}$.

- Matrix-Vektor-Multiplikation:

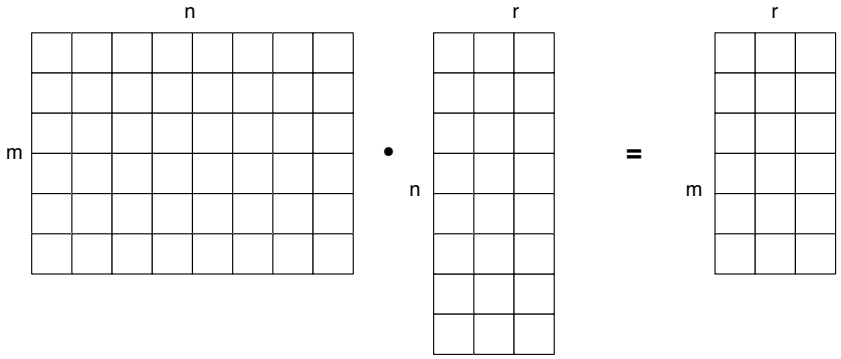
$$A \cdot x = \begin{pmatrix} a_{11}x_1 + \ldots + a_{1n}x_n \\ \vdots \\ a_{m1}x_1 + \ldots + a_{mn}x_n \end{pmatrix}$$

- Matrix-Matrix-Multiplikation:

$$A \cdot B = \begin{pmatrix} a_{11}b_{11} + \ldots + a_{1n}b_{n1} & \cdots & a_{11}b_{1r} + \ldots + a_{1n}b_{nr} \\ \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \ldots + a_{mn}b_{n1} & \cdots & a_{m1}b_{1r} + \ldots + a_{mn}b_{nr} \end{pmatrix}$$

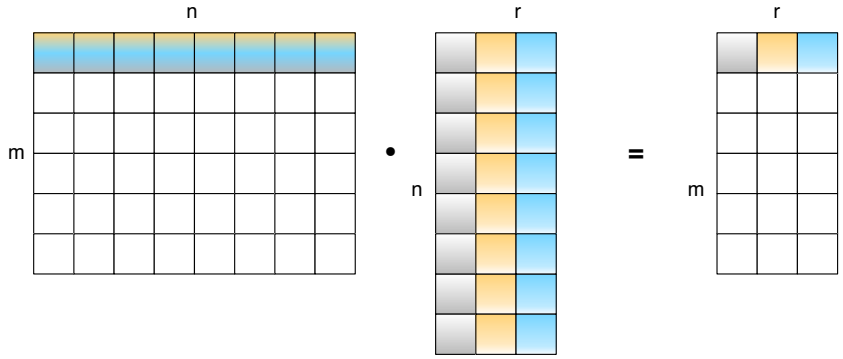
Notizen

Matrix-Multiplikation



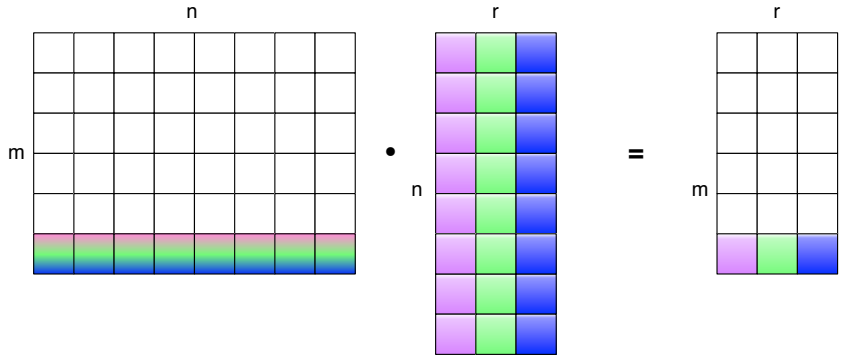
Notizen

Matrix-Multiplikation 2



Notizen

Matrix-Multiplikation 3



Notizen

Matrix-Multiplikation: Komplexität

Seien $A = (a_{ji}) \in \mathbb{R}^{n \times n}$ und $B = (b_{ji}) \in \mathbb{R}^{n \times n}$ (quadratisch).

$$A \cdot B = \begin{pmatrix} a_{11}b_{11} + \ldots + a_{1n}b_{n1} & \cdots & a_{11}b_{1n} + \ldots + a_{1n}b_{nn} \\ \vdots & \ddots & \vdots \\ a_{n1}b_{11} + \ldots + a_{nn}b_{n1} & \cdots & a_{n1}b_{1n} + \ldots + a_{nn}b_{nn} \end{pmatrix}$$

Komplexität:

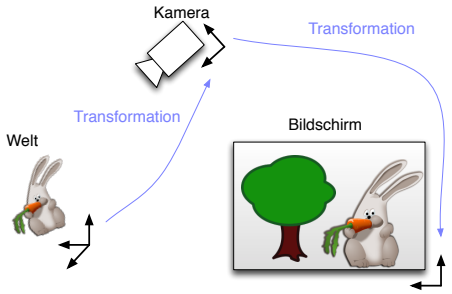
- pro Eintrag: n Additionen, n Multiplikationen
- insgesamt n^2 Einträge
- $A \cdot B$ also n^3 Additionen und n^3 Multiplikationen
- Komplexität: $\Theta(n^3)$ arithmetische Operationen

Notizen

Beispiel: Anwendung von Matrix-Multiplikation

- Wechsel von Koordinaten-Systemen können als Matrix-Vektor-Multiplikation dargestellt werden
  - Matrix heisst hier auch Transformation
- mehrere Wechsel hintereinander können mittels Matrix-Matrix-Multiplikation zu einer Transformation zusammengefasst werden

Beispiel: Augmented Reality



Notizen

Demo: Augmented Reality

☞ Augmented Reality Demo

Notizen

Matrix-Multiplikation: Strassen-Algorithmus

Seien $A, B \in \mathbb{R}^{n \times n}$ mit n 2er-Potenz ($n = 2^k$), $n > 1$.

- Divide & Conquer Ansatz zur Matrizen-Multiplikation
- A, B aufteilen in vier $n/2 \times n/2$ Matrizen:

$$A = \begin{pmatrix} \mathbf{a}_{11} & \mathbf{a}_{12} \\ \mathbf{a}_{21} & \mathbf{a}_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} \mathbf{b}_{11} & \mathbf{b}_{12} \\ \mathbf{b}_{21} & \mathbf{b}_{22} \end{pmatrix}$$

- Produkt $A \cdot B$ berechnen als:

$$A \cdot B = \begin{pmatrix} \mathbf{a}_{11}\mathbf{b}_{11} + \mathbf{a}_{12}\mathbf{b}_{21} & \mathbf{a}_{11}\mathbf{b}_{12} + \mathbf{a}_{12}\mathbf{b}_{22} \\ \mathbf{a}_{21}\mathbf{b}_{11} + \mathbf{a}_{22}\mathbf{b}_{21} & \mathbf{a}_{21}\mathbf{b}_{12} + \mathbf{a}_{22}\mathbf{b}_{22} \end{pmatrix}$$

- $\mathbf{a}_{ik}\mathbf{b}_{kj}$ ist selbst Matrix-Matrix-Produkt
- rekursiv aufteilen bis 1×1 Produkt
- Komplexität: immer noch $\Theta(n^3)$

Notizen

Strassen-Algorithmus

- Berechne:

$$\begin{aligned}
\mathbf{q}_1 &= (\mathbf{a}_{11} + \mathbf{a}_{22}) \cdot (\mathbf{b}_{11} + \mathbf{b}_{22}) \\
\mathbf{q}_2 &= (\mathbf{a}_{21} + \mathbf{a}_{22}) \cdot \mathbf{b}_{11} \\
\mathbf{q}_3 &= \mathbf{a}_{11} \cdot (\mathbf{b}_{12} - \mathbf{b}_{22}) \\
\mathbf{q}_4 &= \mathbf{a}_{22} \cdot (\mathbf{b}_{21} - \mathbf{b}_{11}) \\
\mathbf{q}_5 &= (\mathbf{a}_{11} + \mathbf{a}_{12}) \cdot \mathbf{b}_{22} \\
\mathbf{q}_6 &= (\mathbf{a}_{21} - \mathbf{a}_{11}) \cdot (\mathbf{b}_{11} + \mathbf{b}_{12}) \\
\mathbf{q}_7 &= (\mathbf{a}_{12} - \mathbf{a}_{22}) \cdot (\mathbf{b}_{21} + \mathbf{b}_{22})
\end{aligned}$$

- Dann ist:

$$A \cdot B = \begin{pmatrix} \mathbf{q}_1 + \mathbf{q}_4 - \mathbf{q}_5 + \mathbf{q}_7 & \mathbf{q}_3 + \mathbf{q}_5 \\ \mathbf{q}_2 + \mathbf{q}_4 & \mathbf{q}_1 + \mathbf{q}_3 - \mathbf{q}_2 + \mathbf{q}_6 \end{pmatrix}$$

- Komplexität: $\Theta(n^{\lg 7}) = \Theta(n^{2.807})$

Notizen

Matrix-Matrix-Multiplikation

Seien $A, B \in \mathbb{R}^{n \times n}$.

- naiver Algorithmus: $\Theta(n^3)$
- Strassen-Algorithmus (1969): $\Theta(n^{2.807})$
  - weniger numerisch stabil als naiver Algorithmus
  - n muss 2er-Potenz sein
  - benötigt deutlich mehr Speicher als naiver Algorithmus
- Coppersmith-Winograd Algorithmus (1987): $O(n^{2.376})$
  - erst praktikabel für Grössen, die mit heutigen Computern nicht bearbeitet werden können
  - es existieren verbesserte Varianten (2011) mit $O(n^{2.3727})$

Notizen

Zusammenfassung

Notizen

Notizen