Datenstrukturen und Algorithmen

Zusammenfassung der Vorlesungen 13-19

Prof. Dr. Erika Ábrahám

Theorie Hybrider Systeme
Informatik 2

http://ths.rwth-aachen.de/teaching/ss-14/datenstrukturen-und-algorithmen/

Diese Präsentation verwendet in Teilen Folien von Joost-Pieter Katoen.

17. Juni 2014

RWTH AACHEN UNIVERSITY

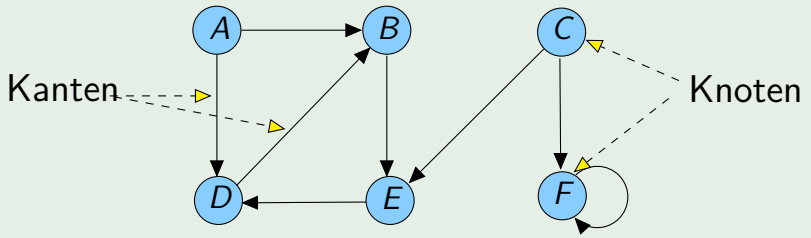
Übersicht

Übersicht

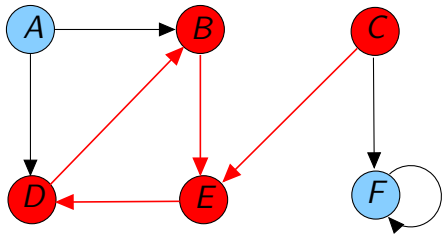
Gerichteter Graph

Beispiel

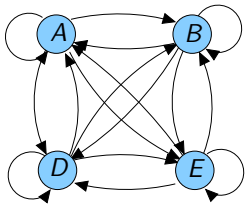
- $V = \{A, \ldots, F\}$
- $E = \{(A,B), (A,D), (B,E), (C,E), (C,F), (D,B), (E,D), (F,F)\}$



Terminologie bei Graphen

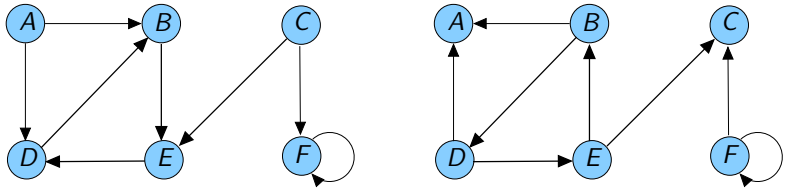


Beispiel: Rote Knoten und Kanten bilden einen Teilgraphen



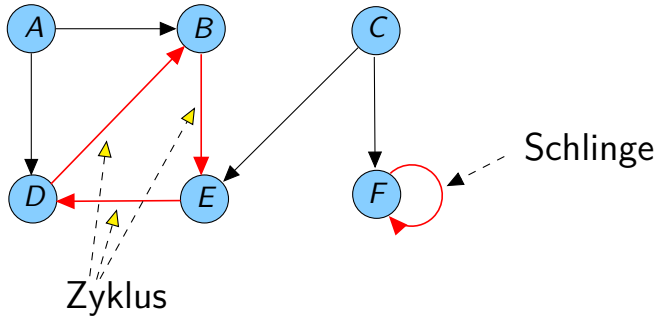
Beispiel: Vollständiger symmetrischer Digraph

Terminologie bei Graphen



Beispiel: Ein Graph und sein transponierter Graph

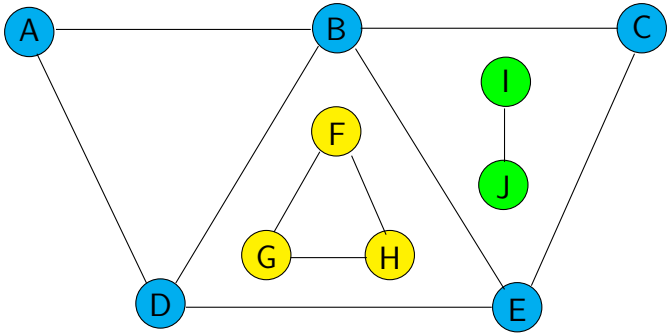
Pfade und Zyklen



$A\,B\,E\,D\,B$ und $C\,F\,F$ sind Pfade.

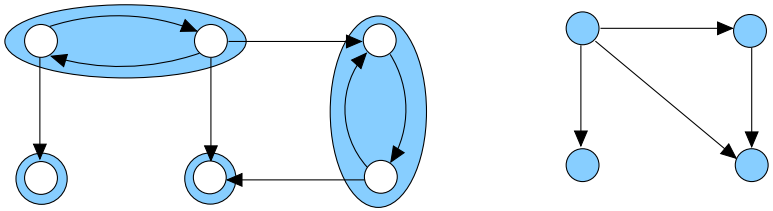
$E\,D\,B$ und $C\,F$ sind einfache Pfade.

Ungerichtete, zusammenhängende Graphen



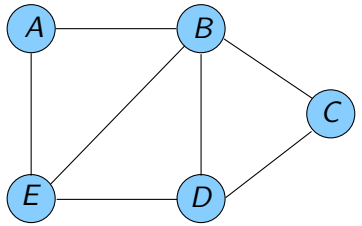
Beispiel: Die Zusammenhangskomponenten eines ungerichteten Graphen.

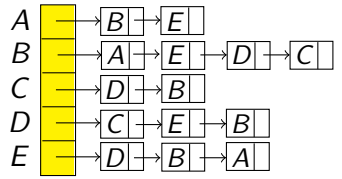
Starke Zusammenhangskomponenten, Kondensationsgraph



Ein Digraph, seine SCCs und sein Kondensationsgraph.

Darstellung eines ungerichteten Graphen



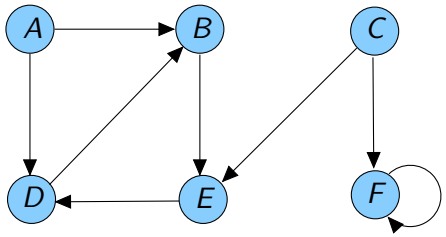


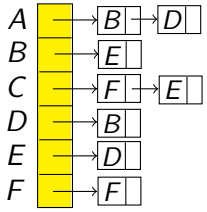
Adjazenzliste

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Adjazenzmatrix

Darstellung eines gerichteten Graphen





Adjazenzliste

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Adjazenzmatrix

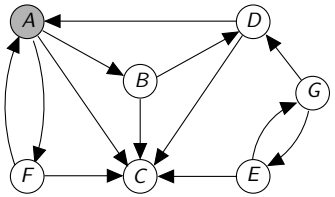
Graphendurchlauf

Graphendurchlaufstrategien

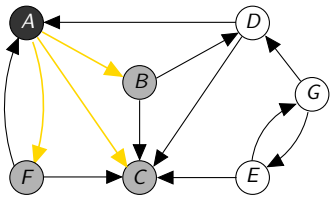
- Tiefensuche
- Breitensuche

In Adjazenzlisten-Darstellung:
Zeitkomplexität $\mathcal{O}(|V| + |E|)$, Platzbedarf $\Theta(|V|)$.

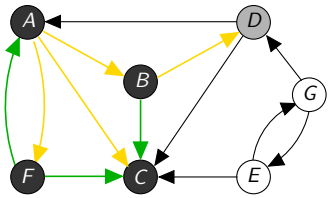
Breitensuche: Beispiel



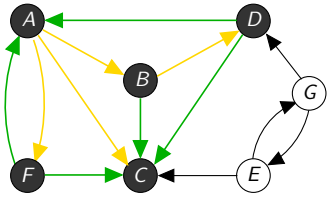
Beginn der Breitensuche



Erforsche alle folgenden nicht-gefundenen Knoten

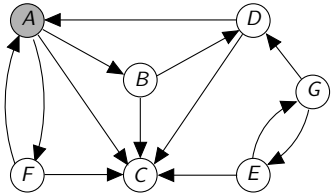


Erforsche alle folgenden nicht-gefundenen Knoten

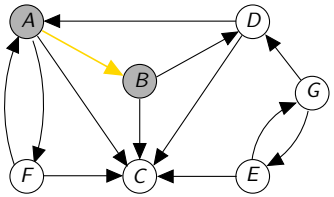


Fertig!

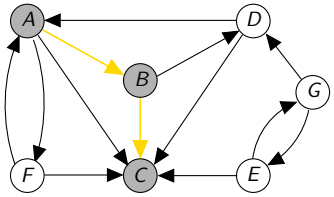
Tiefensuche: Beispiel



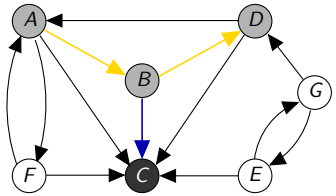
Beginn der Tiefensuche



Erforsche einen Knoten



Erforsche einen Knoten



Sackgasse!
Backtracke und erforsche den nächsten Knoten...

Anwendungen der Tiefensuche

- Erreichbarkeitsanalyse
- CCs in ungerichteten Graphen
- SCCs in gerichteten Graphen
- Topologische Sortierung in gerichteten azyklischen Graphen
- Kritische-Pfad-Analyse in gewichteten gerichteten azyklischen Graphen

CCs in ungerichteten Graphen

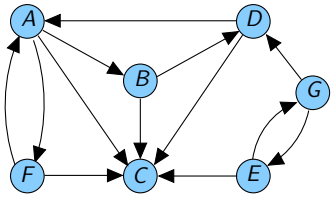
Problem

Finde die Zusammenhangskomponenten (CCs) eines ungerichteten Graphen G.

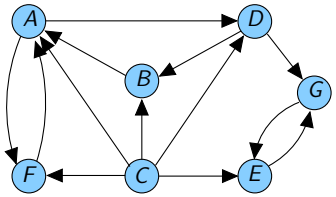
Lösung

- *Finden des CCs eines Knotens v:*
  Verwende Tiefen- oder Breitensuche, um alle aus v erreichbaren Knoten zu bestimmen.
- *Die Zeitkomplexität ist $\mathcal{O}(|V| + |E|)$.*

SCCs in gerichteten Graphen: Sharir's Algorithmus



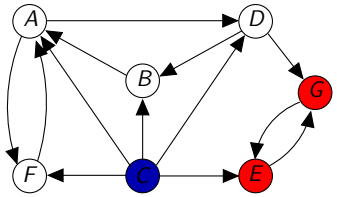
Ursprünglicher Digraph



Transponierter Graph

E
G
A
F
B
D
C

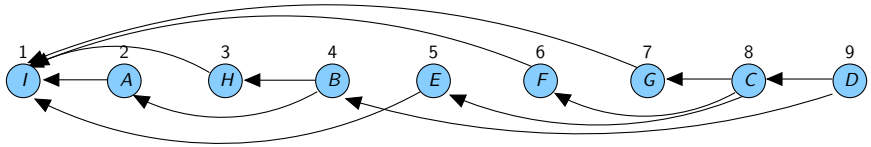
Stack am Ende der Phase 1



In Phase 2 gefundene starke Komponenten

Zeitkomplexität: $\mathcal{O}(|V| + |E|)$, Speicherkomplexität: $\mathcal{O}(|V|)$.

Topologische Sortierung

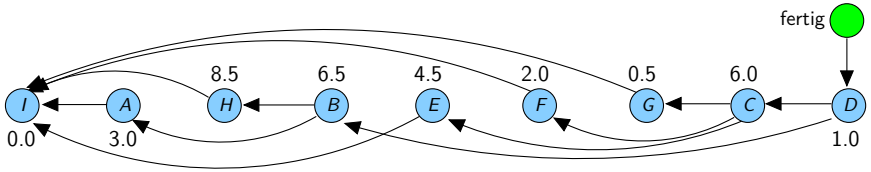


I	A	H	B	E	F	G	C	D
wake up	choose clothes	shower	dress	make coffee	make toast	pour juice	eat breakf	leave

Abhängigkeitsgraph, der topologischen Ordnung entsprechend gezeichnet.

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

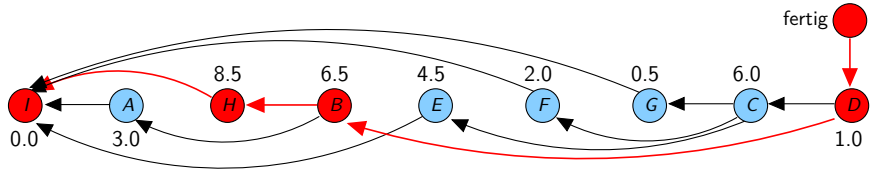
Kritische-Pfad-Analyse



I	A	H	B	E	F	G	C	D
wake up	choose clothes	shower	dress	make coffee	make toast	pour juice	eat breakf	leave
0.0	3.0	8.5	6.5	4.5	2.0	0.5	6.0	1.0

Dauer

Kritische-Pfad-Analyse



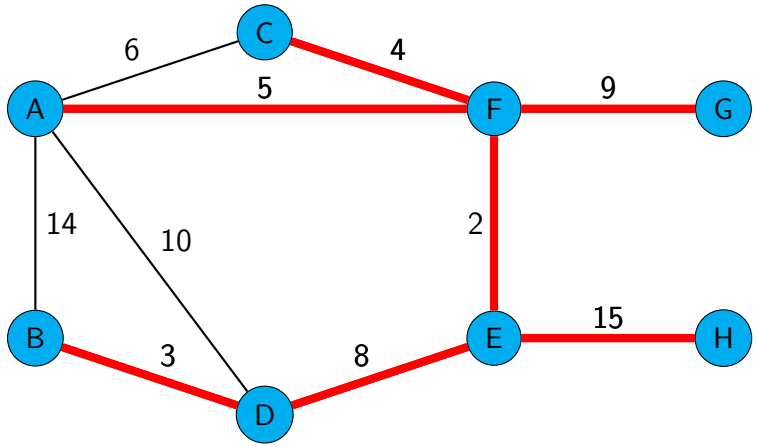
I	A	H	B	E	F	G	C	D
wake up	choose clothes	shower	dress	make coffee	make toast	pour juice	eat breakf	leave
0.0	3.0	8.5	6.5	4.5	2.0	0.5	6.0	1.0

Dauer

- $eft = 1 + 6.5 + 8.5 + 0 = 16$.

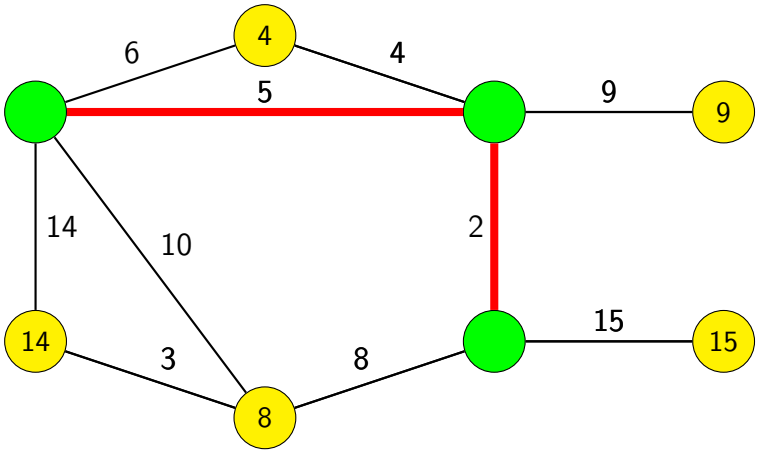
Übersicht

Minimaler Spannbaum



Das ist *ein* minimaler Spannbaum (mit Gesamtgewicht 46).
In diesem Fall ist es auch der einzige.

Prim's Algorithmus



Drei Prioritätswarteschlangenimplementierungen

$T(n,m) \in \mathcal{O}\left(n \cdot T(\texttt{insert}) + n \cdot T(\texttt{getMin}) + n \cdot T(\texttt{delMin}) + m \cdot T(\texttt{decrKey})\right)$

	Implementierung		
Operation	unsortiertes Array	sortiertes Array	Heap
`isEmpty()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`insert(e,k)`	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
`getMin()`	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
`delMin()`	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$
`decrKey(e,k)`	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
Prim	$\mathcal{O}(n^2 + m))$	$\mathcal{O}(n^2 + m \cdot n)$	$\mathcal{O}(n \log n + m \log n)$

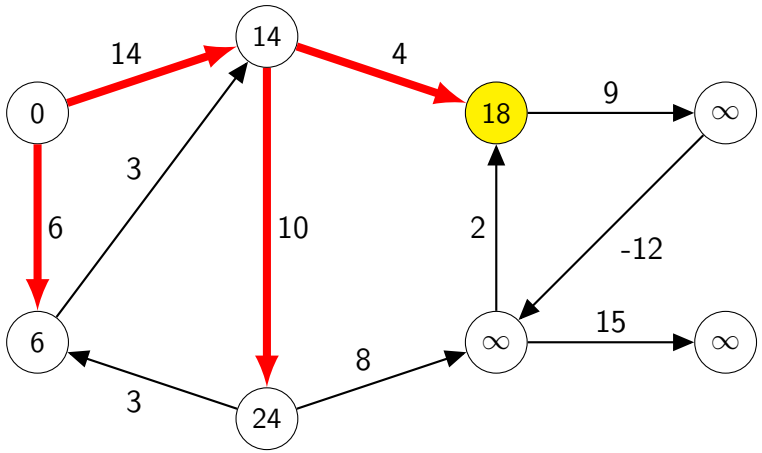
Übersicht

Kürzeste Pfade

Kürzeste-Pfade-Probleme:

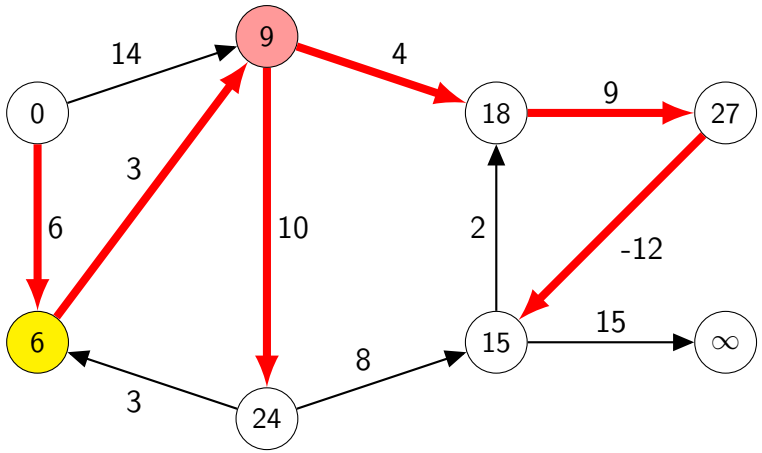
- Kürzeste Pfade von einem Startknoten s zu allen anderen Knoten: Single-Source Shortest Paths (SSSP).
- Kürzeste Pfade von allen Knoten zu einem Zielknoten t.
  Lässt sich auf SSSP zurückführen.
- Kürzeste Pfade für *ein* festes Knotenpaar u, v.
  Es ist kein Algorithmus bekannt, der asymptotisch schneller als der beste SSSP-Algorithmus ist.
- Kürzeste Pfade für *alle* Knotenpaare.
  All-Pairs Shortest Paths.

Single-Source Shortest Paths: Bellman-Ford



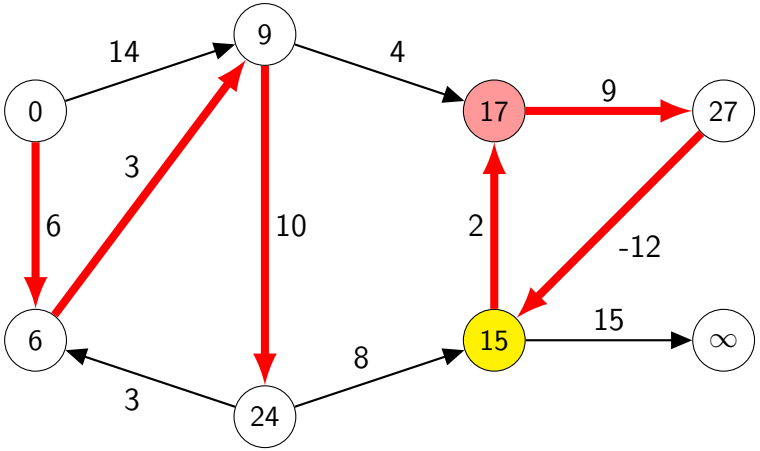
Zeitkomplexität: $\mathcal{O}(|V| \cdot |E|)$.

Single-Source Shortest Paths: Bellman-Ford



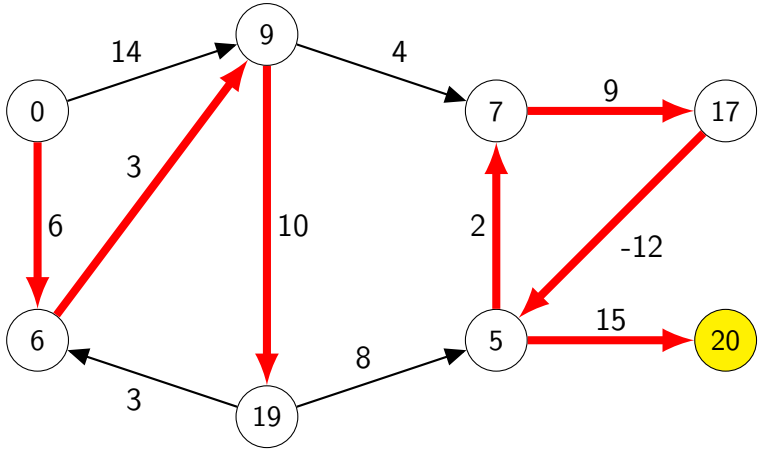
Zeitkomplexität: $\mathcal{O}(|V| \cdot |E|)$.

Single-Source Shortest Paths: Bellman-Ford



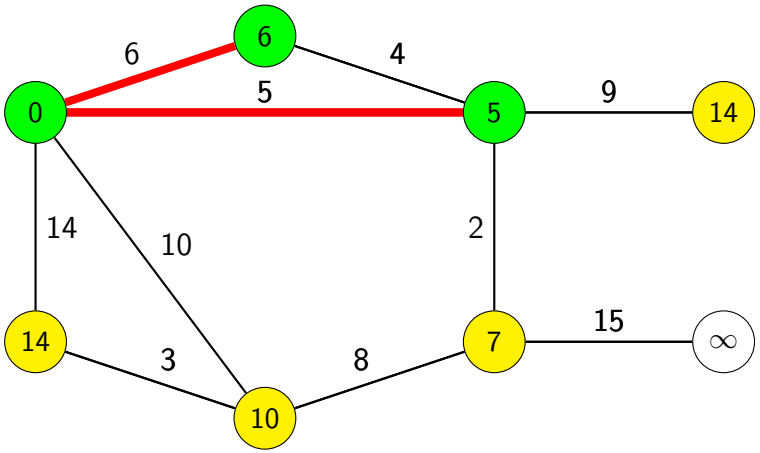
Zeitkomplexität: $\mathcal{O}(|V| \cdot |E|)$.

Single-Source Shortest Paths: Bellman-Ford



Zeitkomplexität: $\mathcal{O}(|V| \cdot |E|)$.

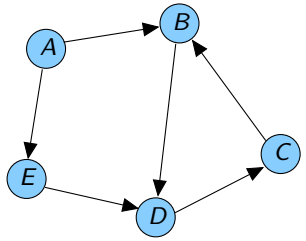
Single-Source Shortest Paths: Dijkstra



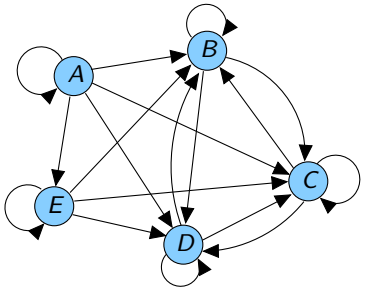
Zeitkomplexität: $\mathcal{O}(|V|^2)$, Platzkomplexität: $\mathcal{O}(|V|)$.

Transitive Hülle: Warshall

$$R = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \text{ und die transitive Hülle } R^* = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

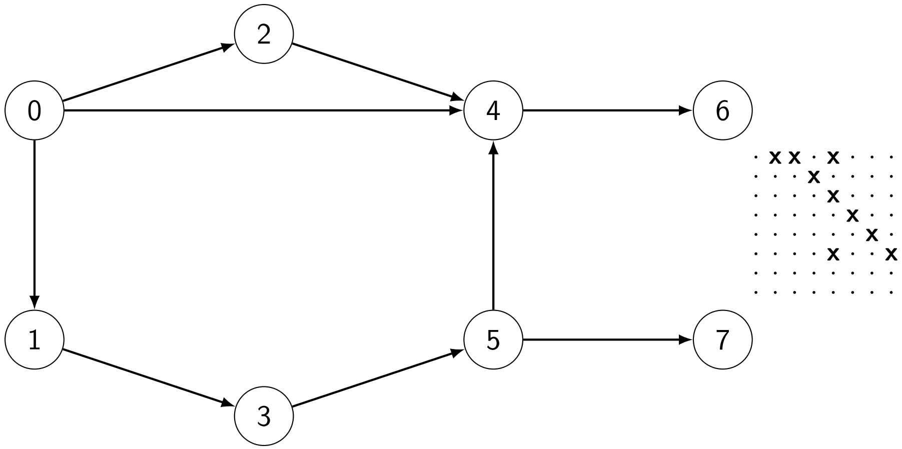


Binäre Relation R

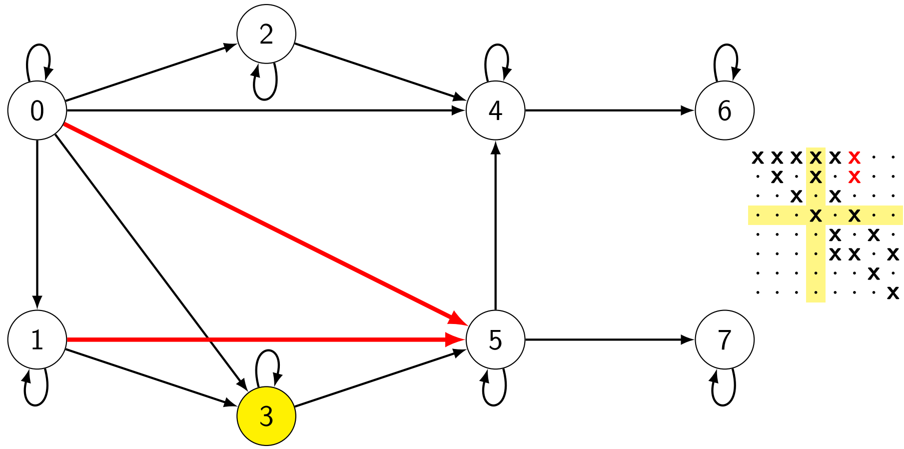


Transitive Hülle R^*

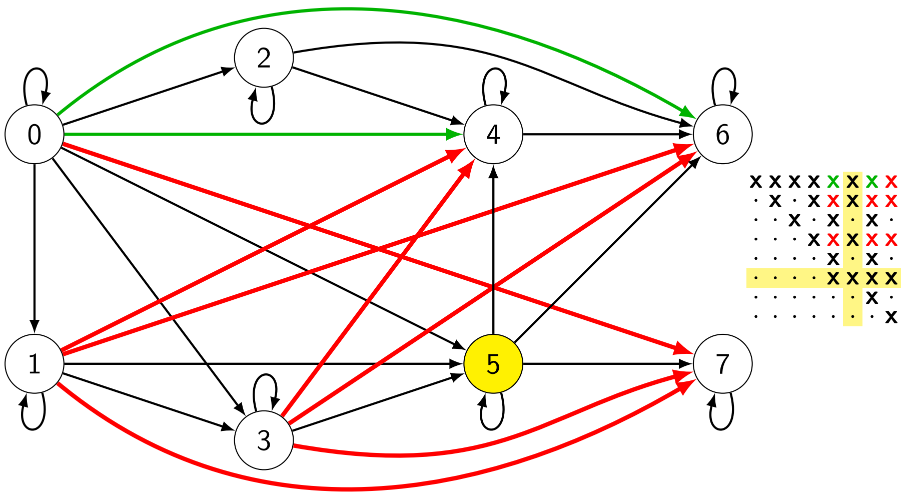
Transitive Hülle: Warshall



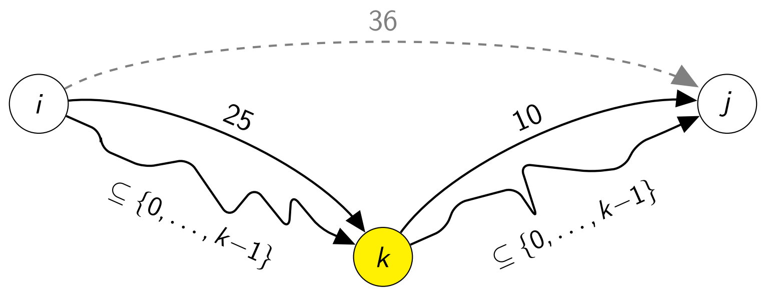
Transitive Hülle: Warshall



Transitive Hülle: Warshall



All-Pairs Shortest Paths: Floyd



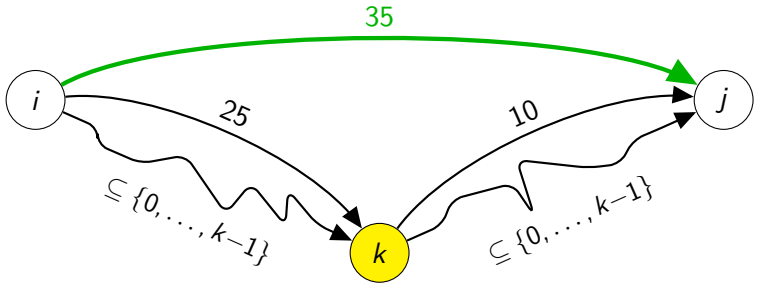
- Vorgehen wie bei Warshall, jedoch mit folgender Rekursionsgleichung:

$$d_{ij}^{(k)} = \begin{cases} W(i,j) & \text{für } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{für } k > 0 \end{cases}$$

$$(\text{statt: } t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right))$$

- Auch hier arbeiten wir direkt im Ausgabearray: `D[i,j]` = $d_{ij}^{(\cdot)}$.

All-Pairs Shortest Paths: Floyd



- Vorgehen wie bei Warshall, jedoch mit folgender Rekursionsgleichung:

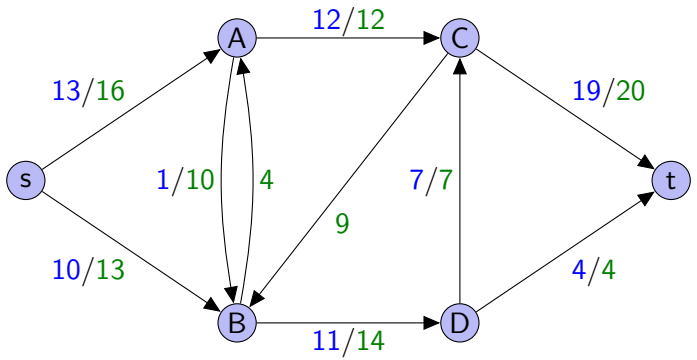
$$d_{ij}^{(k)} = \begin{cases} W(i,j) & \text{für } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{für } k > 0 \end{cases}$$

(statt: $t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$)

- Auch hier arbeiten wir direkt im Ausgabearray: `D[i,j]` = $d_{ij}^{(\cdot)}$.

Übersicht

Maximaler Fluss: Beispiel

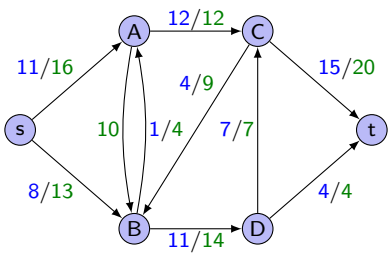


- Ein maximaler Fluss in diesem Beispiel hat den Wert $|f| = 23$.
- Es kann mehrere maximale Flüsse geben.

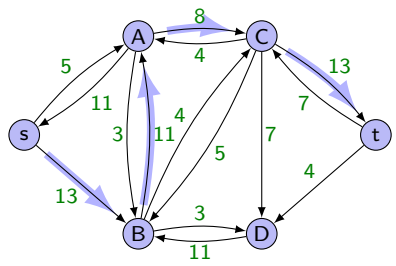
Maximaler Fluss: Ford-Fulkerson

Algorithmus

Initialisiere Fluss f zu 0
while es gibt einen augmentierenden Pfad p
  do augmentiere f entlang p // $f := f + f_p$
return f



Flussnetzwerk G



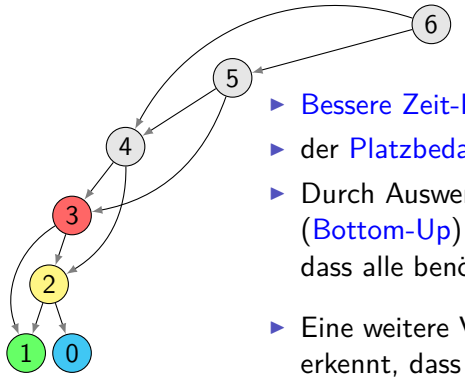
Restnetzwerk G_f

Laufzeit: $\mathcal{O}(E \cdot |f^*|)$.

Übersicht

Memoization: Dynamische Programmierung

- Memoization hilft, wenn die Teilprobleme überlappen.

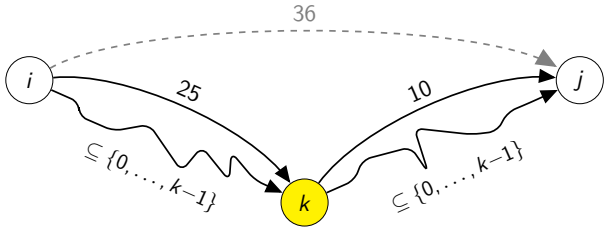


Abhängigkeitsgraph

- Bessere Zeit-Komplexität: $\Theta(n)$ statt $\Theta(2^n)$, aber
- der Platzbedarf wächst dabei auf $\Theta(n)$.
- Durch Auswertung von unten-nach-oben (Bottom-Up) kann sogar sichergestellt werden, dass alle benötigten Werte bereits berechnet sind.
- Eine weitere Verbesserung ergibt sich, indem man erkennt, dass hier jeweils nur die zwei letzten Werte benötigt werden (in-place).

$\Rightarrow$ Auf diesen Grundideen basiert die Dynamische Programmierung.

Rekursionsgleichungen: Floyd-Warshall

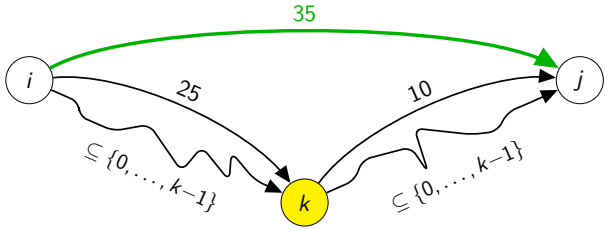


Sei $d_{ij}^{(k)}$ die Länge eines kürzesten Pfades von i nach j über Knoten in der Menge $\{0,1,\ldots,k\}$.
Rekursionsgleichung die dem Algorithmus zur Grunde liegt:

$$d_{ij}^{(k)} = \begin{cases} W(i,j) & \text{für } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{für } k > 0 \end{cases}$$

- Top-down Abhängigkeit: $d_{ij}^{(k)}$ hängt von $d_{ij}^{(k-1)}$ ab.
- Bottom-up Berechnung: $d_{ij}^{(0)}, d_{ij}^{(1)}, d_{ij}^{(2)}, \ldots$ usw.

Rekursionsgleichungen: Floyd-Warshall



Sei $d_{ij}^{(k)}$ die Länge eines kürzesten Pfades von i nach j über Knoten in der Menge $\{0,1,\ldots,k\}$.
Rekursionsgleichung die dem Algorithmus zur Grunde liegt:

$$d_{ij}^{(k)} = \begin{cases} W(i,j) & \text{für } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{für } k > 0 \end{cases}$$

- Top-down Abhängigkeit: $d_{ij}^{(k)}$ hängt von $d_{ij}^{(k-1)}$ ab.
- Bottom-up Berechnung: $d_{ij}^{(0)}, d_{ij}^{(1)}, d_{ij}^{(2)}, \ldots$ usw.

Dynamische Programmierung

Dynamischer Programmierung kann man i. d. R. in vier Teile gliedern:

1. Charakterisiere die Struktur einer optimalen Lösung, und stelle fest ob diese DP ermöglicht.
   - Teilprobleme sind (teilweise) überlappend
   - Rekursive Abhängigkeit zwischen den Teilproblemen.
2. Stelle die Rekursionsgleichung (top-down) für den Wert der Lösung auf.
3. Löse Rekursionsgleichung bottom-up.
4. Bestimme aus dem Wert der Lösung die Argumente der Lösung. Rekonstruiere die Lösung.

Anwendungen

- Longest Common Subsequence (LCS)
- Ketten von Matrixmultiplikationen
- Das Rucksackproblem

Anwendung: Longest Common Subsequence

Sei $A_i = (a_1, \ldots, a_i)$ der i-te Präfix von $A = (a_1, \ldots, a_m)$ für $0 \leqslant i \leqslant m$.

Lemma (Optimale Teilstruktur)

1. *Enden zwei Sequenzen mit dem selben Zeichen $a_m = b_n$, dann ist dieses Zeichen auch Teil der LCS:*
$$LCS(A_m, B_n) = (LCS(A_{m-1}, B_{n-1}), a_m).$$
2. *Andernfalls gilt entweder*
$$LCS(A_m, B_n) = LCS(A_m, B_{n-1}) \quad \textit{oder}$$
$$LCS(A_m, B_n) = LCS(A_{m-1}, B_n).$$
Insbesondere ist
$$|LCS(A_m, B_n)| = \max(|LCS(A_m, B_{n-1})|, |LCS(A_{m-1}, B_n)|).$$

Anwendung: Longest Common Subsequence

Wir können wieder die Rekursiongleichung für den *Wert*, also die Länge der LCS aufstellen: $L[i,j] = |LCS(A_i, B_j)|$

$$L[i,j] = \begin{cases} 0 & \text{für } i = 0 \text{ oder } j = 0 \\ L[i-1, j-1] + 1 & \text{falls } a_i = b_j,\ i,j > 0 \\ \max(L[i, j-1], L[i-1, j]) & \text{falls } a_i \neq b_j,\ i,j > 0 \end{cases}$$

- Das lässt sich direkt als Algorithmus umsetzen.
- Dessen Laufzeit ist $O(|A| \cdot |B|)$, ebenso seine Platzkomplexität.

Anwendung: Matrixmultiplikation

$$\underbrace{(A_1 \cdot \ldots \cdot A_k)}_{k \text{ Matrizen}} \cdot \underbrace{(A_{k+1} \cdot \ldots \cdot A_n)}_{n-k \text{ Matrizen}}$$

$m[i,j]$ sei die minimale Anzahl Multiplikationen für die Teilkette $A_i \cdot \ldots \cdot A_j$.

- Offenbar ist $m[i,i] = 0$ für alle $0 < i \leqslant n$.
- Die Dimension einer Teilkette ist $d_{i-1} \times d_j$.
- Teilen bei Position k ergibt: $m[i,j] = m[i,k] + d_{i-1} \cdot d_k \cdot d_j + m[k+1,j]$.
- Wir suchen dabei das optimale k, also:

$$m[i,j] = \begin{cases} 0 & \text{für } i = j, \\ \min_{i \leqslant k < j}(m[i,k] + m[k+1,j] + d_{i-1} \cdot d_k \cdot d_j) & \text{für } i < j. \end{cases}$$

Anwendung: Matrixmultiplikation

Beispiel

Sei $A_0 \in \mathbb{R}^{30\times 1}$, $A_1 \in \mathbb{R}^{1\times 40}$, $A_2 \in \mathbb{R}^{40\times 10}$, $A_3 \in \mathbb{R}^{10\times 25}$.

```
m[i,j] = min(m[i,k] + m[k+1,j] + dim[i]*dim[k+1]*dim[j+1]);
dim[] = {30,   1,   40,   10,  25};
```

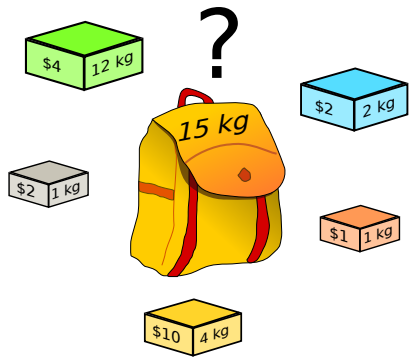
$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 1200 & 700 & 1400 \\ & 0 & 400 & 650 \\ & & 0 & 10\,000 \\ & & & 0 \end{bmatrix} \end{array} \quad \begin{bmatrix} 0 & 0 & 0 \\ & 1 & 2 \\ & & 2 \\ & & \end{bmatrix}$$

- $((A_0A_1)A_2)A_3$ benötigt 20 700 Multiplikationen.
- Rekonstruktion: $A_0 \cdot ((A_1 \cdot A_2) \cdot A_3)$ ist optimal – 1400 Multiplikationen.

Anwendung: Das Rucksackproblem

Das Rucksackproblem (0-1 Knapsack)

Gegeben sei ein Rucksack, mit maximaler Tragkraft M, sowie n Gegenstände, die sowohl ein Gewicht als auch einen Wert haben.
Nehme möglichst viel Wert mit, ohne den Rucksack zu überladen.



Anwendung: Das Rucksackproblem

Wir wollen das Problem mittels dynamischer Programmierung lösen.

- Wir bestimmen zunächst $c_{\max}$:
- Angenommen wir kennen den maximalen Wert $c'_{\max}$ des Rucksacks mit Tragkraft M', bei dem nur die ersten $n-1$ Gegenstände berücksichtigt werden.
- Für $c_{\max}$ stellt sich also die Frage, ob der n-te Gegenstand (Gewicht: w_{n-1}, Wert: c_{n-1}) mitgenommen wird.

Betrachten wir beide Fälle:

Ohne: $c_{\max}$ wäre dann gleich $c'_{\max}$ für $M' = M$.

Mit: $c_{\max}$ wäre dann gleich $c'_{\max} + c_{n-1}$ für $M' = M - w_{n-1}$.
Falls $M' < 0$, dann setzen wir $c'_{\max} = -\infty$ („geht nicht“).

$\Rightarrow$ Wähle den Fall mit dem größeren Wert.

Anwendung: Das Rucksackproblem

Sei also $C[i,j]$ der maximale Wert des Rucksacks mit Tragkraft j, wenn man nur die Gegenstände $\{0,\ldots,i-1\}$ berücksichtigt.

Es ergibt sich folgende Rekursionsgleichung:

$$C[i,j] = \begin{cases} \max(C[i-1,j], c_{i-1} + C[i-1, j-w_{i-1}]) & \\ -\infty & \text{für } j < 0 \\ 0 & \text{für } i = 0,\ j \geqslant 0 \end{cases}$$

- Dann ist $c_{\max} = C[n, M]$.
- Diese Rekursionsgleichung lösen wir nun bottom-up, indem wir die Rucksäcke mit allen möglichen Gewichten $\{0,\ldots,M\}$ berechnen, wenn wir jeweils einen weiteren Gegenstand hinzunehmen.

Anwendung: Das Rucksackproblem

Offen ist noch die Frage, *welche* Gegenstände ($S \subseteq G$) nun eigentlich mitgenommen werden müssen, um $c_{\max}$ zu erreichen.

- Falls $C[i,j] = C[i-1,j]$ ist, dann wurde der Gegenstand nicht mitgenommen (auch bei $c_i = 0$).
- Ausgehend von $C[n,M]$ kann man somit (mit Hilfe der w_i) die Menge S rekonstruieren (in $\Theta(n)$).

Beispiel

`w[] = { 2, 12, 1, 1, 4 }, c[] = { 2, 4, 2, 1, 10 }, M = 15`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2	0	0	2	2	2	2	2	2	2	2	2	2	4	4	6	6
3	0	2	2	4	4	4	4	4	4	4	4	4	4	6	6	8
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

Übersicht

Mathematische Hilfsmittel

Vektor, Skalarprodukt, Betrag, Determinante

- Vektor (im $\mathbb{R}^n$, insbesondere $n = 2$):

$$\vec{a} = \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \qquad \vec{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}.$$

- Skalarprodukt (dot product):

$$\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2.$$

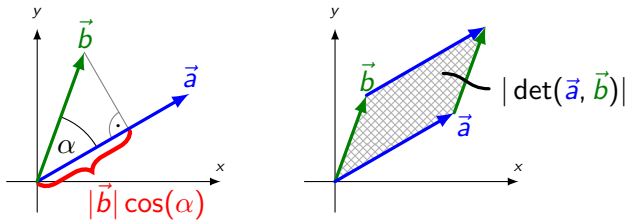
- Betrag (Länge):

$$|\vec{a}| = \sqrt{\vec{a} \cdot \vec{a}} = \sqrt{a_1^2 + a_2^2}.$$

- Determinante für $A = [\vec{a}, \vec{b}] = \begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix}$:

$$\det A = \det(\vec{a}, \vec{b}) = a_1 b_2 - a_2 b_1.$$

Geometrische Interpretation

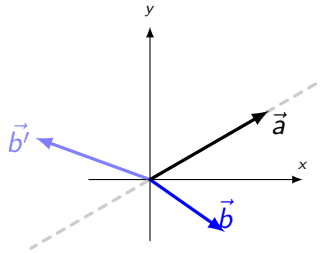


- Es gilt: $\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 = |\vec{a}||\vec{b}|\cos(\alpha)$. („Länge der Projektion“).
- Die Fläche (allgemein: Volumen) des durch $\vec{a}$ und $\vec{b}$ aufgespannten Parallelogramms ist gerade der Absolutwert der Determinanten.
  Oder: Die Determinante liefert eine vorzeichenbehaftete Fläche.

Winkelbestimmung

Problem

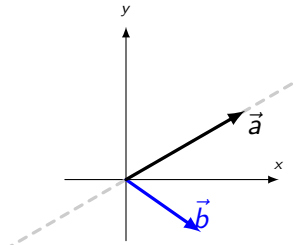
Liegt ein Vektor $\vec{b}$ links oder rechts von einem gegeben Vektor $\vec{a}$?



Winkelbestimmung

Problem

Liegt ein Vektor $\vec{b}$ links oder rechts von einem gegeben Vektor $\vec{a}$?

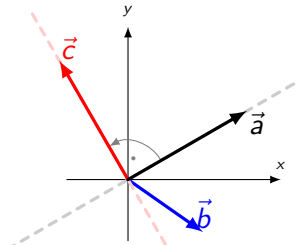


- Wir betrachten zunächst $\vec{b}$.

Winkelbestimmung

Problem

Liegt ein Vektor $\vec{b}$ links oder rechts von einem gegeben Vektor $\vec{a}$?

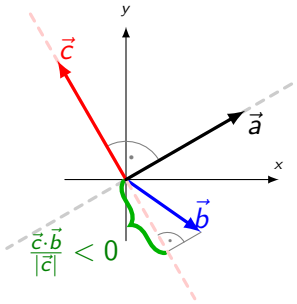


- Wir betrachten zunächst $\vec{b}$.
- Konstruiere $\vec{c}$, den zu $\vec{a}$ im mathematisch positiven Sinn (Gegenuhrzeigersinn) um 90 Grad gedrehten Vektor.

Winkelbestimmung

Problem

Liegt ein Vektor $\vec{b}$ links oder rechts von einem gegeben Vektor $\vec{a}$?

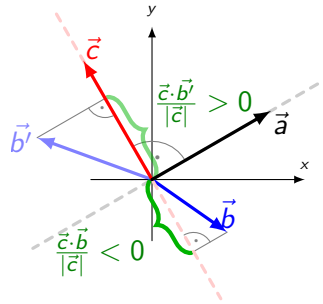


- Wir betrachten zunächst $\vec{b}$.
- Konstruiere $\vec{c}$, den zu $\vec{a}$ im mathematisch positiven Sinn (Gegenuhrzeigersinn) um 90 Grad gedrehten Vektor.
- Projeziere $\vec{b}$ auf $\vec{c}$. Da $\vec{b}$ rechts von $\vec{a}$ liegt und damit von $\vec{c}$ wegzeigt, ist $\vec{c}\cdot\vec{b}$ negativ.

Winkelbestimmung

Problem

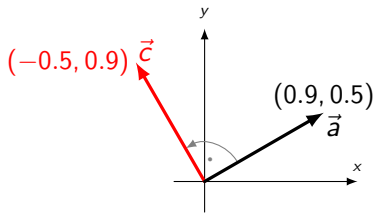
Liegt ein Vektor $\vec{b}$ links oder rechts von einem gegeben Vektor $\vec{a}$?



- Wir betrachten zunächst $\vec{b}$.
- Konstruiere $\vec{c}$, den zu $\vec{a}$ im mathematisch positiven Sinn (Gegenuhrzeigersinn) um 90 Grad gedrehten Vektor.
- Projeziere $\vec{b}$ auf $\vec{c}$. Da $\vec{b}$ rechts von $\vec{a}$ liegt und damit von $\vec{c}$ wegzeigt, ist $\vec{c}\cdot\vec{b}$ negativ.
- $\vec{b}'$ dagegen liegt links von $\vec{a}$, daher ist $\vec{c}\cdot\vec{b}'$ positiv.

Winkelbestimmung

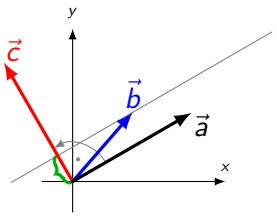
Wie berechnet sich aber $\vec{c}$ aus $\vec{a}$?



- Für $\vec{a} = (a_1, a_2)$ ist $\vec{c} = (c_1, c_2)$ gerade $(-a_2, a_1)$.

Winkelbestimmung

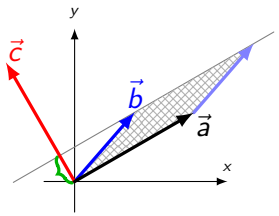
Wie berechnet sich aber $\vec{c}$ aus $\vec{a}$?



- Für $\vec{a} = (a_1, a_2)$ ist $\vec{c} = (c_1, c_2)$ gerade $(-a_2, a_1)$.
- Insgesamt ist damit $\vec{c} \cdot \vec{b} = c_1 b_1 + c_2 b_2 = -a_2 b_1 + a_1 b_2$

Winkelbestimmung

Wie berechnet sich aber $\vec{c}$ aus $\vec{a}$?

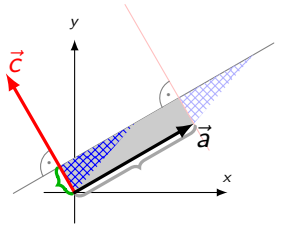


- Für $\vec{a} = (a_1, a_2)$ ist $\vec{c} = (c_1, c_2)$ gerade $(-a_2, a_1)$.
- Insgesamt ist damit $\vec{c} \cdot \vec{b} = c_1 b_1 + c_2 b_2 = -a_2 b_1 + a_1 b_2 = \det(\vec{a}, \vec{b})$.

Zusammenfassung der Vorlesungen 13-19 Algorithmische Geometrie

Winkelbestimmung

Wie berechnet sich aber $\vec{c}$ aus $\vec{a}$?

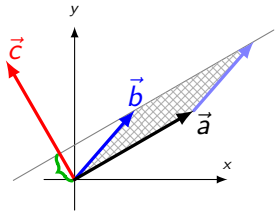


- Für $\vec{a} = (a_1, a_2)$ ist $\vec{c} = (c_1, c_2)$ gerade $(-a_2, a_1)$.
- Insgesamt ist damit $\vec{c} \cdot \vec{b} = c_1 b_1 + c_2 b_2 = -a_2 b_1 + a_1 b_2 = \det(\vec{a}, \vec{b})$.

Zusammenfassung der Vorlesungen 13-19 Algorithmische Geometrie

Winkelbestimmung

Wie berechnet sich aber $\vec{c}$ aus $\vec{a}$?



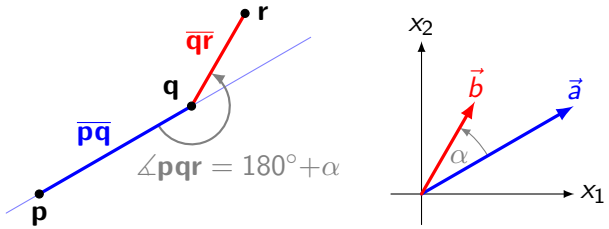
- Für $\vec{a} = (a_1, a_2)$ ist $\vec{c} = (c_1, c_2)$ gerade $(-a_2, a_1)$.
- Insgesamt ist damit $\vec{c} \cdot \vec{b} = c_1 b_1 + c_2 b_2 = -a_2 b_1 + a_1 b_2 = \det(\vec{a}, \vec{b})$.
- Ist $\det(\vec{a}, \vec{b}) = 0$, dann sind $\vec{a}$ und $\vec{b}$ parallel (bzw. antiparallel).
- Ist $\det(\vec{a}, \vec{b}) > 0$, dann liegt $\vec{b}$ links von $\vec{a}$.
- Ist $\det(\vec{a}, \vec{b}) < 0$, dann liegt $\vec{b}$ rechts von $\vec{a}$.

Zusammenfassung der Vorlesungen 13-19 Algorithmische Geometrie

Winkelbestimmung

Problem

Gegeben der Streckenzug $(\mathbf{p}, \mathbf{q}, \mathbf{r})$. *Wird bei* $\mathbf{q}$ *nach links oder rechts abgebogen? Oder: Ist der Winkel* $\measuredangle\mathbf{pqr} > 180°$ *oder* $< 180°$*?*

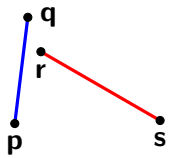


- Wir verwenden wieder die Determinante.
- Dazu berechnen wir $\vec{a} = \vec{d}_{pq} = \mathbf{q} - \mathbf{p}$ und $\vec{b} = \vec{d}_{qr} = \mathbf{r} - \mathbf{q}$.
- $\det(\vec{a}, \vec{b}) > 0$, falls der Knick nach links geht ($\measuredangle\mathbf{pqr} > 180°$, $\alpha > 0$).
- Wenn $\mathbf{r}$ auf (der Verlängerung von) $\overline{\mathbf{pq}}$ liegt, dann ist $\det(\vec{a}, \vec{b}) = 0$ ($\measuredangle\mathbf{pqr} = 0°$ oder $= 180°$).

Schnitt zweier Strecken

Problem

Gegeben zwei Strecken $\overline{\mathbf{pq}}$ und $\overline{\mathbf{rs}}$. Schneiden sich diese?



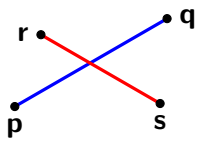
- Wir sind nicht an der Position des Schnittpunktes interessiert.
- Idee: Wir testen, ob die Endpunkte **r** und **s** auf verschiedenen Seiten von $\overline{\mathbf{pq}}$ liegen. Ebenso für **p** und **q** bezüglich $\overline{\mathbf{rs}}$.

Zusammenfassung der Vorlesungen 13-19 Algorithmische Geometrie

Schnitt zweier Strecken

Problem

Gegeben zwei Strecken $\overline{\mathbf{pq}}$ und $\overline{\mathbf{rs}}$. Schneiden sich diese?



- Wir sind nicht an der Position des Schnittpunktes interessiert.
- Idee: Wir testen, ob die Endpunkte **r** und **s** auf verschiedenen Seiten von $\overline{\mathbf{pq}}$ liegen. Ebenso für **p** und **q** bezüglich $\overline{\mathbf{rs}}$.

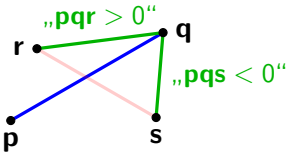
Prof. Dr. Erika Ábrahám Datenstrukturen und Algorithmen 53/57

Zusammenfassung der Vorlesungen 13-19 Algorithmische Geometrie

Schnitt zweier Strecken

Problem

Gegeben zwei Strecken $\overline{\mathbf{pq}}$ und $\overline{\mathbf{rs}}$. Schneiden sich diese?



- Wir sind nicht an der Position des Schnittpunktes interessiert.
- Idee: Wir testen, ob die Endpunkte **r** und **s** auf verschiedenen Seiten von $\overline{\mathbf{pq}}$ liegen. Ebenso für **p** und **q** bezüglich $\overline{\mathbf{rs}}$.

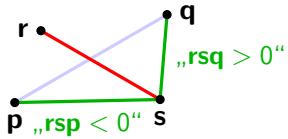
Prof. Dr. Erika Ábrahám Datenstrukturen und Algorithmen 53/57

Zusammenfassung der Vorlesungen 13-19 Algorithmische Geometrie

Schnitt zweier Strecken

Problem

Gegeben zwei Strecken $\overline{\mathbf{pq}}$ und $\overline{\mathbf{rs}}$. Schneiden sich diese?



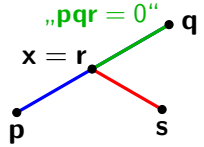
- Wir sind nicht an der Position des Schnittpunktes interessiert.
- Idee: Wir testen, ob die Endpunkte **r** und **s** auf verschiedenen Seiten von $\overline{\mathbf{pq}}$ liegen. Ebenso für **p** und **q** bezüglich $\overline{\mathbf{rs}}$.

Prof. Dr. Erika Ábrahám Datenstrukturen und Algorithmen 53/57

Schnitt zweier Strecken

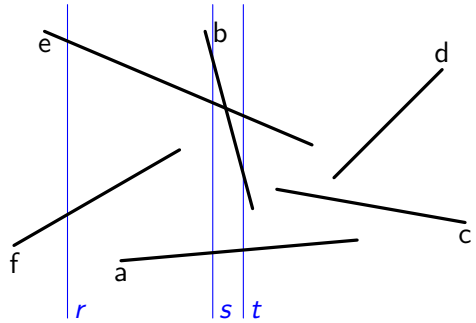
Problem

Gegeben zwei Strecken $\overline{\mathbf{pq}}$ und $\overline{\mathbf{rs}}$. Schneiden sich diese?



- Wir sind nicht an der Position des Schnittpunktes interessiert.
- Idee: Wir testen, ob die Endpunkte **r** und **s** auf verschiedenen Seiten von $\overline{\mathbf{pq}}$ liegen. Ebenso für **p** und **q** bezüglich $\overline{\mathbf{rs}}$.
- Sonderfall: det = 0. Der Endpunkt, etwa **x**, liegt also auf der Verlängerung von $\overline{\mathbf{pq}}$ (bzw. $\overline{\mathbf{rs}}$).
  Es bleibt zu prüfen, ob **x** auch zwischen **p** und **q** liegt.

Schnitt eines beliebigen Streckenpaares



Beispiel

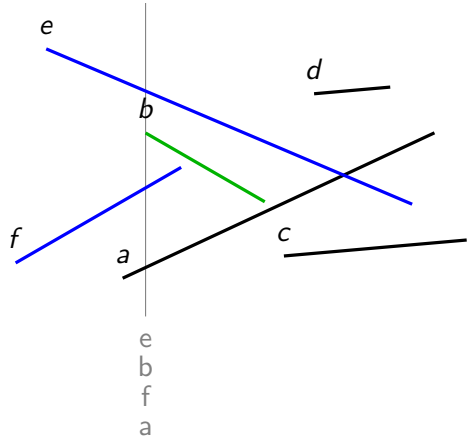
- e $>_r$ f; sowie f mit a nicht vergleichbar bei r (usw.).
- b $>_s$ a, e $>_s$ a, b $>_s$ e.
- b $>_t$ a, e $>_t$ a, e $>_t$ b. (Der Schnitt vertauscht die Reihenfolge von e und b).

Vergleichbarkeit

Zwei Strecken s_1 und s_2 heißen vergleichbar an der Stelle r, wenn beide die vertikale Linie bei r (in der ersten Dimension) schneiden.

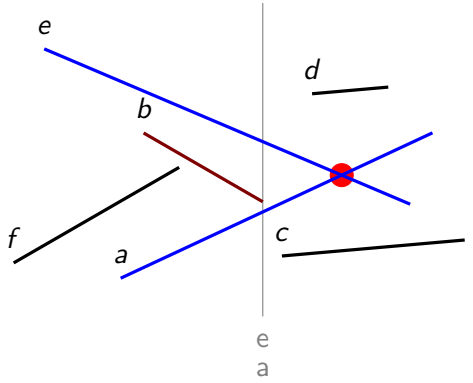
- Wenn s_1 an der Stelle r über s_2 liegt schreiben wir $s_1 >_r s_2$, sonst $s_2 >_r s_1$, bzw. $s_1 =_r s_2$.

Schnitt eines beliebigen Streckenpaares: Sweepline



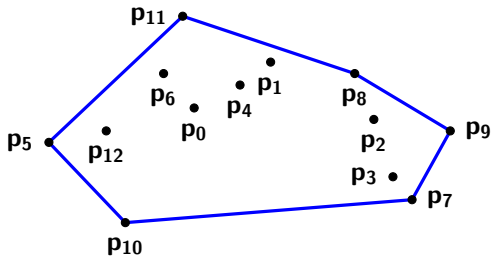
- Hier wurde der erste Schnitt gefunden (Algorithmus terminiert).
- Zeitkomplexität: $O(n \log n)$

Schnitt eines beliebigen Streckenpaares: Sweepline



- Hier wurde der erste Schnitt gefunden (Algorithmus terminiert).
- Zeitkomplexität: $O(n \log n)$

Konvexe Hülle

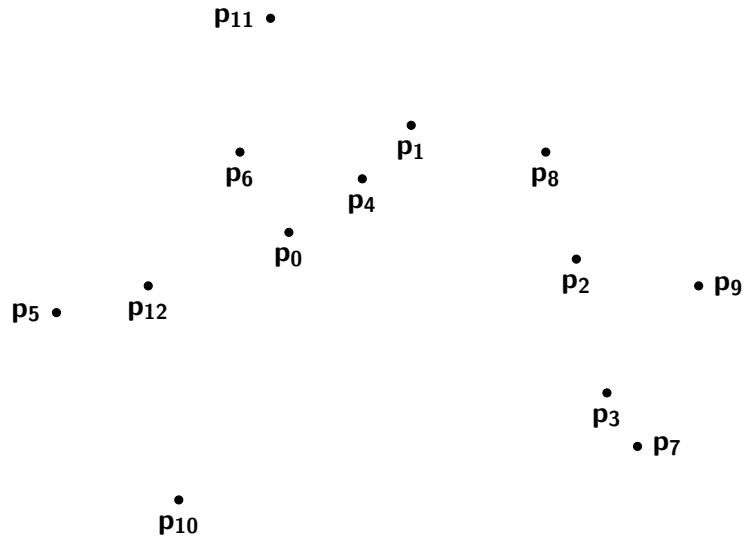


Konvexe Hülle

Die konvexe Hülle einer Menge Q von Punkten ist das kleinste konvexe Polygon P, für das sich jeder Punkt in Q entweder auf dem Rand von P oder in seinem Inneren befindet.

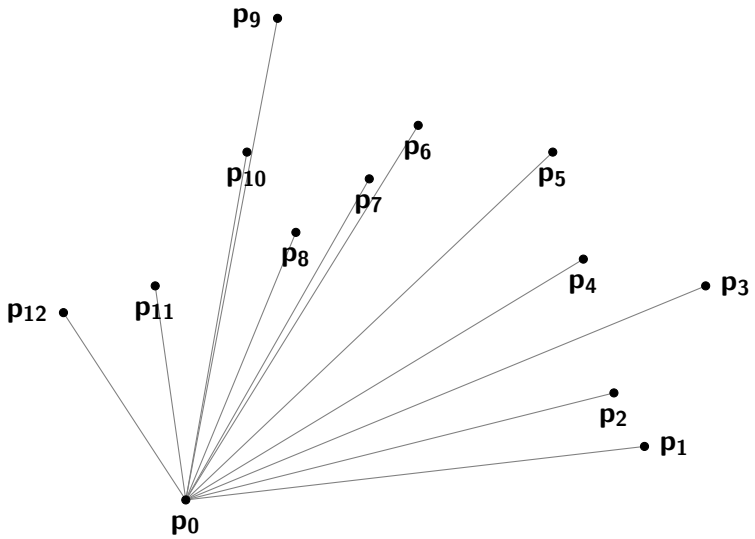
- Betrachte jeden Punkt als Nagel, der aus einem Brett herausragt.
- Die konvexe Hülle hat dann die Form, die durch ein straffes Gummiband gebildet wird, das alle Nägel umschließt.

Konvexe Hülle: Graham's Scan



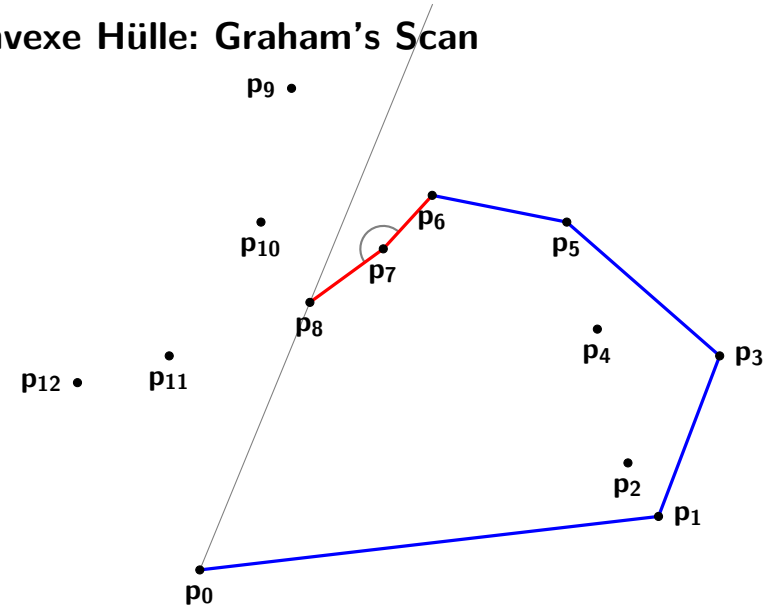
Zeitkomplexität: $\mathcal{O}(n \log n)$.

Konvexe Hülle: Graham's Scan



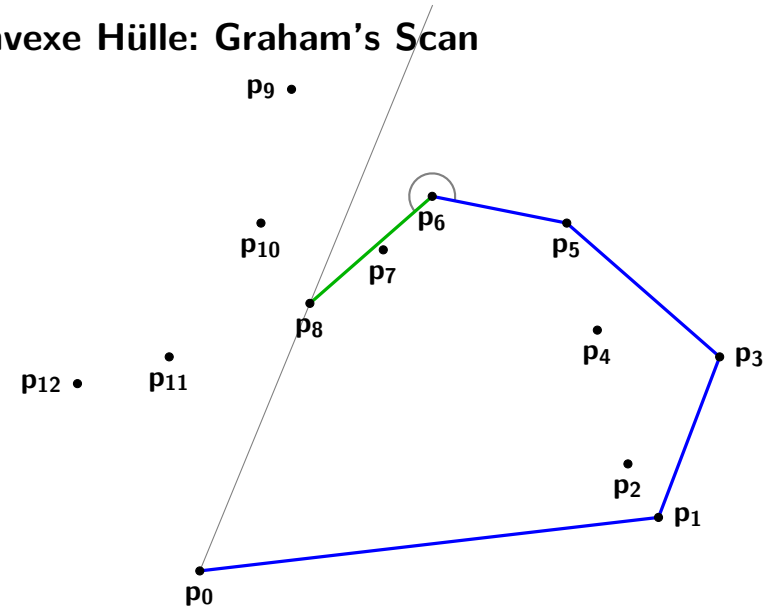
Zeitkomplexität: $\mathcal{O}(n \log n)$.

Konvexe Hülle: Graham's Scan



Zeitkomplexität: $\mathcal{O}(n \log n)$.

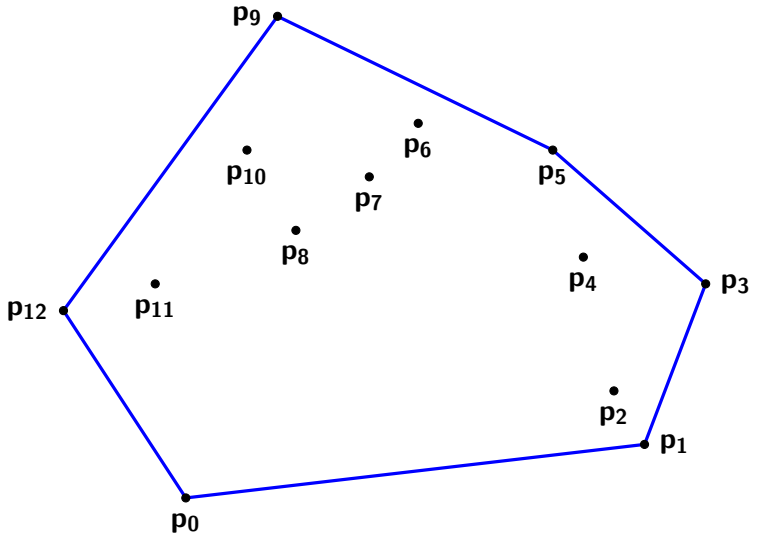
Konvexe Hülle: Graham's Scan



Zeitkomplexität: $\mathcal{O}(n \log n)$.

Zusammenfassung der Vorlesungen 13-19 Algorithmische Geometrie

Konvexe Hülle: Graham's Scan



Zeitkomplexität: $\mathcal{O}(n \log n)$.

Prof. Dr. Erika Ábrahám Datenstrukturen und Algorithmen 57/57