Karlsruher Institut für Technologie

Tutorium Algorithmen 1

12 · Union-Find und dynamische Programmierung · 15.7.2024
Peter Bohner Tutorium 3

Wiederholung: MST

- nur für ungerichtete, gewichtete Graphen

- minimaler Spannbaum (MST):
  - Teilgraph mit gleicher Kantenmenge
  - verbindet alle Knoten des Original-Graphs
  - Gesamtkosten der Kanten so klein wie möglich

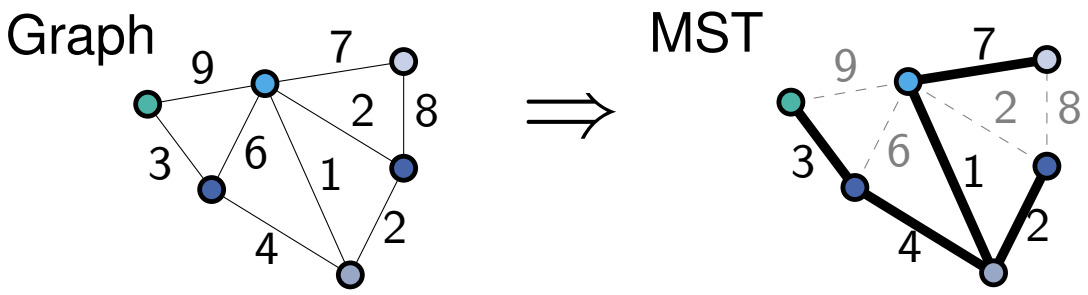
Wiederholung: MST

- nur für ungerichtete, gewichtete Graphen

- minimaler Spannbaum (MST):
  - Teilgraph mit gleicher Kantenmenge
  - verbindet alle Knoten des Original-Graphs
  - Gesamtkosten der Kanten so klein wie möglich
  - $\Rightarrow$ Kreisfrei (daher immer Baum)

Wiederholung: MST

- nur für ungerichtete, gewichtete Graphen
- minimaler Spannbaum (MST):
  - Teilgraph mit gleicher Kantenmenge
  - verbindet alle Knoten des Original-Graphs
  - Gesamtkosten der Kanten so klein wie möglich
  - $\Rightarrow$ Kreisfrei (daher immer Baum)



Wiederholung: Kruskal-Algorithmus

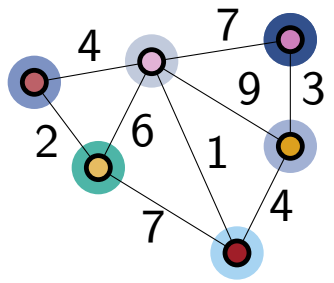
- findet MST
- Algorithmus:
  - Jeder Knoten bekommt seine eigene Menge
  - Betrachte billigste Kante zwischen zwei Mengen
  - verschmelze beide Mengen

Wiederholung: Kruskal-Algorithmus

- findet MST
- Algorithmus:
  - Jeder Knoten bekommt seine eigene Menge
  - Betrachte billigste Kante zwischen zwei Mengen
  - verschmelze beide Mengen
  - wiederhole letze beiden Schritte, bis nur eine Menge übrig ist

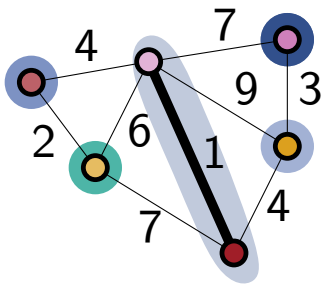
Wiederholung: Kruskal-Algorithmus

- findet MST
- Algorithmus:
  - Jeder Knoten bekommt seine eigene Menge
  - Betrachte billigste Kante zwischen zwei Mengen
  - verschmelze beide Mengen
  - wiederhole letze beiden Schritte, bis nur eine Menge übrig ist



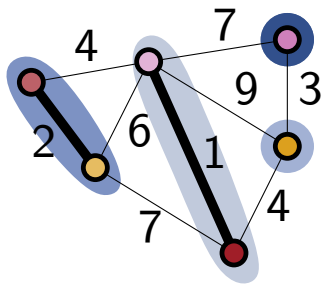
Wiederholung: Kruskal-Algorithmus

- findet MST
- Algorithmus:
  - Jeder Knoten bekommt seine eigene Menge
  - Betrachte billigste Kante zwischen zwei Mengen
  - verschmelze beide Mengen
  - wiederhole letze beiden Schritte, bis nur eine Menge übrig ist



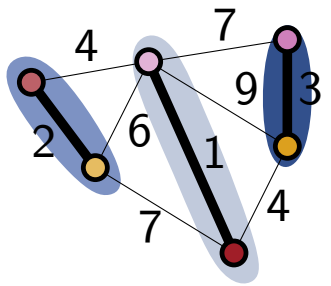
Wiederholung: Kruskal-Algorithmus

- findet MST
- Algorithmus:
  - Jeder Knoten bekommt seine eigene Menge
  - Betrachte billigste Kante zwischen zwei Mengen
  - verschmelze beide Mengen
  - wiederhole letze beiden Schritte, bis nur eine Menge übrig ist



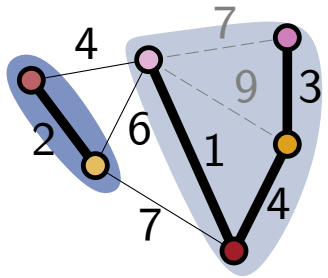
Wiederholung: Kruskal-Algorithmus

- findet MST
- Algorithmus:
  - Jeder Knoten bekommt seine eigene Menge
  - Betrachte billigste Kante zwischen zwei Mengen
  - verschmelze beide Mengen
  - wiederhole letze beiden Schritte, bis nur eine Menge übrig ist



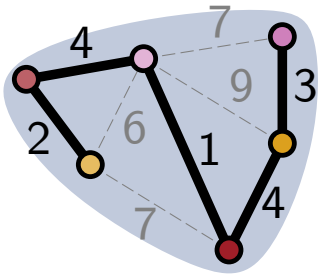
Wiederholung: Kruskal-Algorithmus

- findet MST
- Algorithmus:
  - Jeder Knoten bekommt seine eigene Menge
  - Betrachte billigste Kante zwischen zwei Mengen
  - verschmelze beide Mengen
  - wiederhole letze beiden Schritte, bis nur eine Menge übrig ist



Wiederholung: Kruskal-Algorithmus

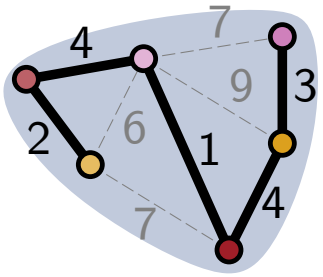
- findet MST
- Algorithmus:
  - Jeder Knoten bekommt seine eigene Menge
  - Betrachte billigste Kante zwischen zwei Mengen
  - verschmelze beide Mengen
  - wiederhole letze beiden Schritte, bis nur eine Menge übrig ist



Wiederholung: Kruskal-Algorithmus

- findet MST
- Algorithmus:
  - Jeder Knoten bekommt seine eigene Menge
  - Betrachte billigste Kante zwischen zwei Mengen
  - verschmelze beide Mengen
  - wiederhole letze beiden Schritte, bis nur eine Menge übrig ist

$\Rightarrow$ ausgewählte Kanten sind nie teurer als nötig
$\Rightarrow$ Kanten bilden immer MST



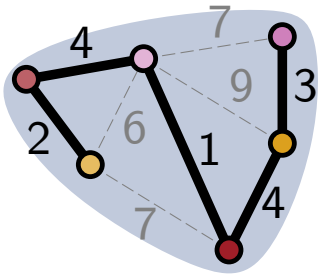
Wiederholung: Kruskal-Algorithmus

- findet MST
- Algorithmus:
  - Jeder Knoten bekommt seine eigene Menge
  - Betrachte billigste Kante zwischen zwei Mengen
  - verschmelze beide Mengen
  - wiederhole letze beiden Schritte, bis nur eine Menge übrig ist

$\Rightarrow$ ausgewählte Kanten sind nie teurer als nötig
$\Rightarrow$ Kanten bilden immer MST

Implementierungsdetails:

- betrachte alle Kanten aufsteigend nach Gewicht
- überspringe Kanten zwischen Knoten der gleichen Menge
- sonst verschmelze Mengen



Union-Find-Datenstruktur

- beschreibt paarweise disjunkte Mengen
- für uns sind Objekte Zahlen von 0 bis $n - 1$
  - für beliebige Objekte äquivalent
- billiges find (=„zu welcher Menge gehört Objekt“)
- billiges union (=„verschmelzen von Mengen“)

Union-Find-Datenstruktur

- beschreibt paarweise disjunkte Mengen
- für uns sind Objekte Zahlen von 0 bis $n-1$
  - für beliebige Objekte äquivalent
- billiges find (=„zu welcher Menge gehört Objekt“)
- billiges union (=„verschmelzen von Mengen“)

naive Implementierung

- speichere Array mit Länge n
- Arrayeintrag repräsentiert Menge (Zahl von 0 bis $n-1$)

Union-Find-Datenstruktur

- beschreibt paarweise disjunkte Mengen
- für uns sind Objekte Zahlen von 0 bis $n - 1$
  - für beliebige Objekte äquivalent
- billiges find (=„zu welcher Menge gehört Objekt“)
- billiges union (=„verschmelzen von Mengen“)

naive Implementierung

- speichere Array mit Länge n
- Arrayeintrag repräsentiert Menge (Zahl von 0 bis $n - 1$)
- Anfangszustand: Array speichert an Stelle i Zahl i
- find: Arrayeintrag zurückgeben
- union: Jeden Arrayeintrag betrachten und ggf. neue Menge eintragen

Union-Find-Datenstruktur

- beschreibt paarweise disjunkte Mengen
- für uns sind Objekte Zahlen von 0 bis $n-1$
  - für beliebige Objekte äquivalent
- billiges find (=„zu welcher Menge gehört Objekt“)
- billiges union (=„verschmelzen von Mengen“)

naive Implementierung

- speichere Array mit Länge n
- Arrayeintrag repräsentiert Menge (Zahl von 0 bis $n-1$)
- Anfangszustand: Array speichert an Stelle i Zahl i
- find: Arrayeintrag zurückgeben
- union: Jeden Arrayeintrag betrachten und ggf. neue Menge eintragen

$\Rightarrow$ union $\in \Theta(n)$

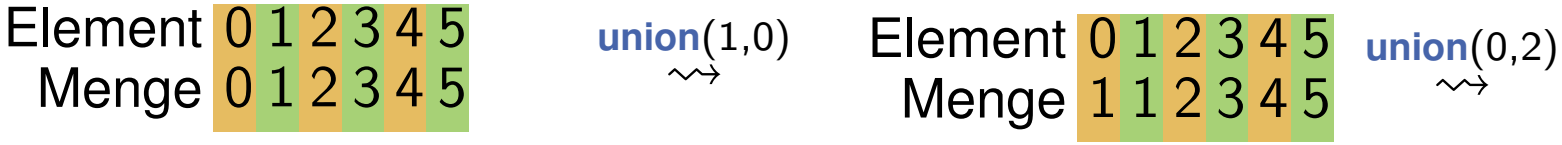
- sehr teuer

Beispiel

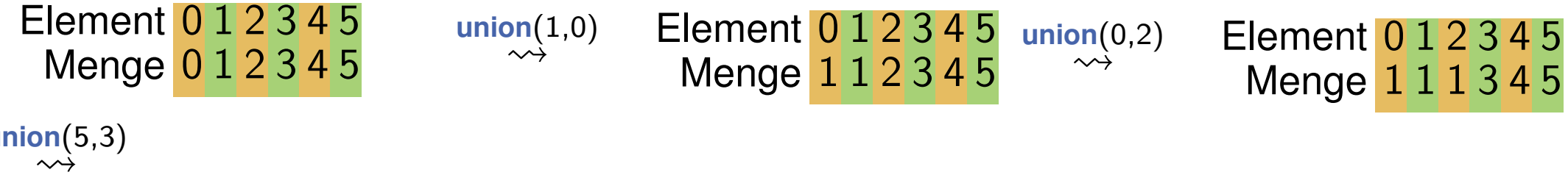
Element	0	1	2	3	4	5
Menge	0	1	2	3	4	5

union(1,0)
$\rightsquigarrow$

Beispiel


Element 0 1 2 3 4 5
Menge 0 1 2 3 4 5
union(1,0)
Element 0 1 2 3 4 5
Menge 1 1 2 3 4 5
union(0,2)

Beispiel


Element 0 1 2 3 4 5
Menge 0 1 2 3 4 5
union(1,0)
Element 0 1 2 3 4 5
Menge 1 1 2 3 4 5
union(0,2)
Element 0 1 2 3 4 5
Menge 1 1 1 3 4 5
union(5,3)

Beispiel


Element 0 1 2 3 4 5
Menge 0 1 2 3 4 5
union(1,0)
Element 0 1 2 3 4 5
Menge 1 1 2 3 4 5
union(0,2)
Element 0 1 2 3 4 5
Menge 1 1 1 3 4 5
union(5,3)
Element 0 1 2 3 4 5
Menge 1 1 1 5 4 5

Beispiel

Element	0	1	2	3	4	5
Menge	0	1	2	3	4	5

union(1,0) ⇝

Element	0	1	2	3	4	5
Menge	1	1	2	3	4	5

union(0,2) ⇝

Element	0	1	2	3	4	5
Menge	1	1	1	3	4	5

union(5,3) ⇝

Element	0	1	2	3	4	5
Menge	1	1	1	5	4	5

erweiterter Ansatz

- reverse lookup: Speichere pro Menge Liste aller Elemente

Menge	Elemente
0	$\langle\rangle$
1	$\langle 0, 1, 2\rangle$
2	$\langle\rangle$
3	$\langle\rangle$
4	$\langle 4\rangle$
5	$\langle 5, 5\rangle$

Beispiel

Element	0	1	2	3	4	5
Menge	0	1	2	3	4	5

union(1,0) ⇝

Element	0	1	2	3	4	5
Menge	1	1	2	3	4	5

union(0,2) ⇝

Element	0	1	2	3	4	5
Menge	1	1	1	3	4	5

union(5,3) ⇝

Element	0	1	2	3	4	5
Menge	1	1	1	5	4	5

erweiterter Ansatz

- reverse lookup: Speichere pro Menge Liste aller Elemente

Menge	Elemente
0	$\langle\rangle$
1	$\langle 0, 1, 2\rangle$
2	$\langle\rangle$
3	$\langle\rangle$
4	$\langle 4\rangle$
5	$\langle 5, 5\rangle$

- Frage: Wie kann man jetzt union beschleunigen?

Beispiel

Element	0	1	2	3	4	5
Menge	0	1	2	3	4	5

union(1,0) $\rightsquigarrow$

Element	0	1	2	3	4	5
Menge	1	1	2	3	4	5

union(0,2) $\rightsquigarrow$

Element	0	1	2	3	4	5
Menge	1	1	1	3	4	5

union(5,3) $\rightsquigarrow$

Element	0	1	2	3	4	5
Menge	1	1	1	5	4	5

erweiterter Ansatz

- reverse lookup: Speichere pro Menge Liste aller Elemente

Menge	Elemente
0	$\langle\rangle$
1	$\langle 0, 1, 2\rangle$
2	$\langle\rangle$
3	$\langle\rangle$
4	$\langle 4\rangle$
5	$\langle 5, 5\rangle$

- Frage: Wie kann man jetzt union beschleunigen?
- **union**: gehe nur über Elemente, die geändert werden
- **union** $\in \Theta($Größe der 2. Menge$)$
  $\Rightarrow$ ggf. wieder $\Theta(n)$

Beispiel

Element	0	1	2	3	4	5
Menge	0	1	2	3	4	5

union(1,0) ⇝

Element	0	1	2	3	4	5
Menge	1	1	2	3	4	5

union(0,2) ⇝

Element	0	1	2	3	4	5
Menge	1	1	1	3	4	5

union(5,3) ⇝

Element	0	1	2	3	4	5
Menge	1	1	1	5	4	5

erweiterter Ansatz

- reverse lookup: Speichere pro Menge Liste aller Elemente

Menge	Elemente
0	$\langle\rangle$
1	$\langle 0, 1, 2\rangle$
2	$\langle\rangle$
3	$\langle\rangle$
4	$\langle 4\rangle$
5	$\langle 5, 5\rangle$

- Frage: Wie kann man jetzt union beschleunigen?
- **union**: gehe nur über Elemente, die geändert werden
- **union** $\in \Theta($Größe der 2. Menge$)$
  $\Rightarrow$ ggf. wieder $\Theta(n)$
- Frage: Was ist der Speicherbedarf dieser Datenstruktur?

Beispiel

Element	0	1	2	3	4	5
Menge	0	1	2	3	4	5

union(1,0) ⇝

Element	0	1	2	3	4	5
Menge	1	1	2	3	4	5

union(0,2) ⇝

Element	0	1	2	3	4	5
Menge	1	1	1	3	4	5

union(5,3) ⇝

Element	0	1	2	3	4	5
Menge	1	1	1	5	4	5

erweiterter Ansatz

- reverse lookup: Speichere pro Menge Liste aller Elemente

Menge	Elemente
0	$\langle\rangle$
1	$\langle 0, 1, 2\rangle$
2	$\langle\rangle$
3	$\langle\rangle$
4	$\langle 4\rangle$
5	$\langle 5, 5\rangle$

- Frage: Wie kann man jetzt union beschleunigen?
- **union**: gehe nur über Elemente, die geändert werden
- **union** $\in \Theta($Größe der 2. Menge$)$
  $\Rightarrow$ ggf. wieder $\Theta(n)$
- Frage: Was ist der Speicherbedarf dieser Datenstruktur?
  - jeweils $O(n)$ für die Arrays und $O(n)$ für Listeneinträge

Weitere Optimierungen

- Laufzeit hängt stark von union-Muster ab
  - falls $|\text{2. Menge}| \in \Theta(1)$ ist Laufzeit optimal

Weitere Optimierungen

- Laufzeit hängt stark von union-Muster ab
  - falls $|\text{2. Menge}| \in \Theta(1)$ ist Laufzeit optimal

$\rightsquigarrow$ ggf. immer kleinere Menge auflösen

- weniger abhängig vom Muster der union-Aufrufe
- asymptotisch nicht signifikant besser

Weitere Optimierungen

- Laufzeit hängt stark von union-Muster ab
  - falls $|\text{2. Menge}| \in \Theta(1)$ ist Laufzeit optimal

$\rightsquigarrow$ ggf. immer kleinere Menge auflösen

- weniger abhängig vom Muster der union-Aufrufe
- asymptotisch nicht signifikant besser

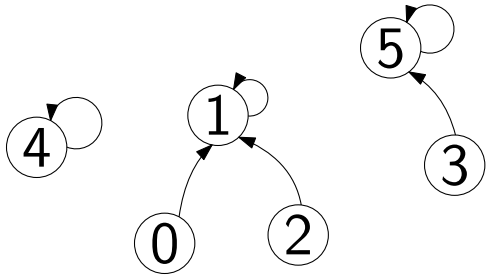
- beschränkt große Mengen sind eher Spezialfall
- Laufzeit sollte unabhängig von konkreten Daten sein

Tatsächliche Umsetzung

Jede Menge hat Repräsentant ($A[i] = i$)

Element	0	1	2	3	4	5
Menge	1	1	1	5	4	5

$\sim$



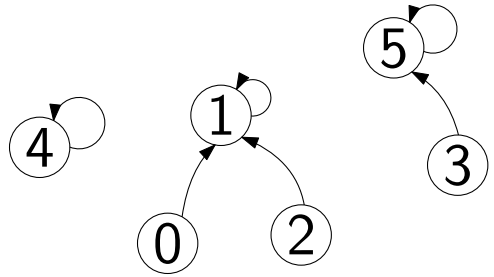
union(3,2)

Tatsächliche Umsetzung

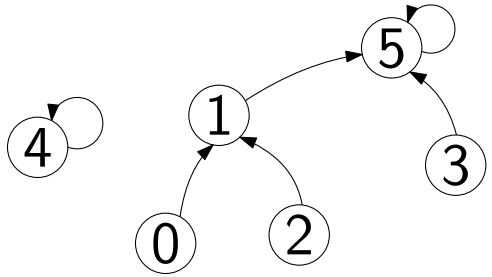
Jede Menge hat Repräsentant ($A[i] = i$)

Element	0	1	2	3	4	5
Menge	1	1	1	5	4	5

$\sim$



union(3,2) $\rightsquigarrow$

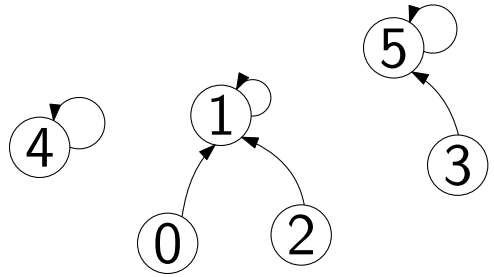


Tatsächliche Umsetzung

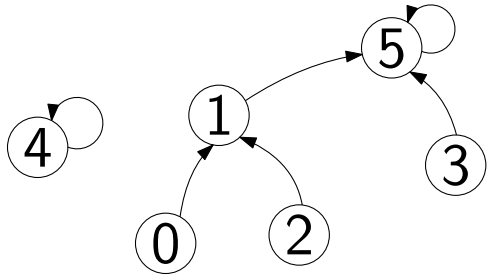
Jede Menge hat Repräsentant ($A[i] = i$)

Element	0	1	2	3	4	5
Menge	1	1	1	5	4	5

$\sim$



union(3,2) $\rightsquigarrow$



- **union**: Verschiebe Repräsentant einer Menge in andere Menge
- **find**: Setze $i \leftarrow A[i]$ bis i Repräsentant ist

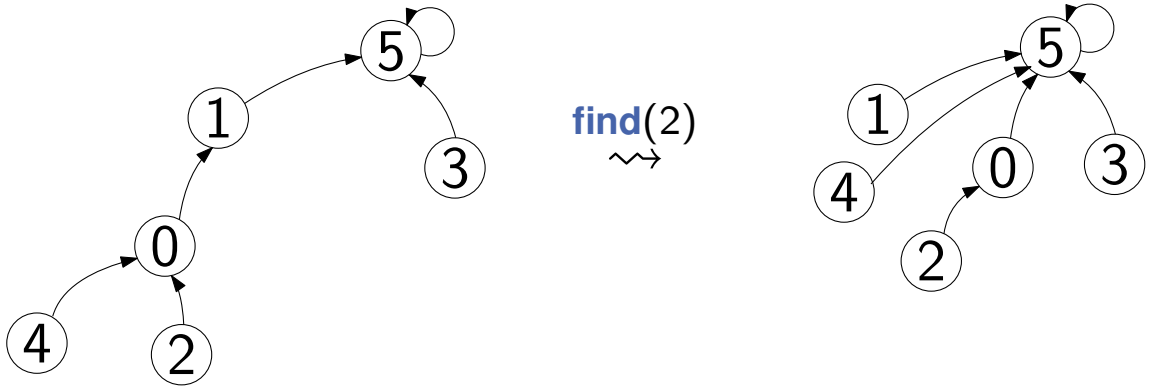
- worst-case: **union** verlängert Pfad um 1

$\Rightarrow$ **find** $\in \Theta(m)$, für m-union Aufrufe

$\Rightarrow$ Datenstruktur wird mit der Zeit unordentlich

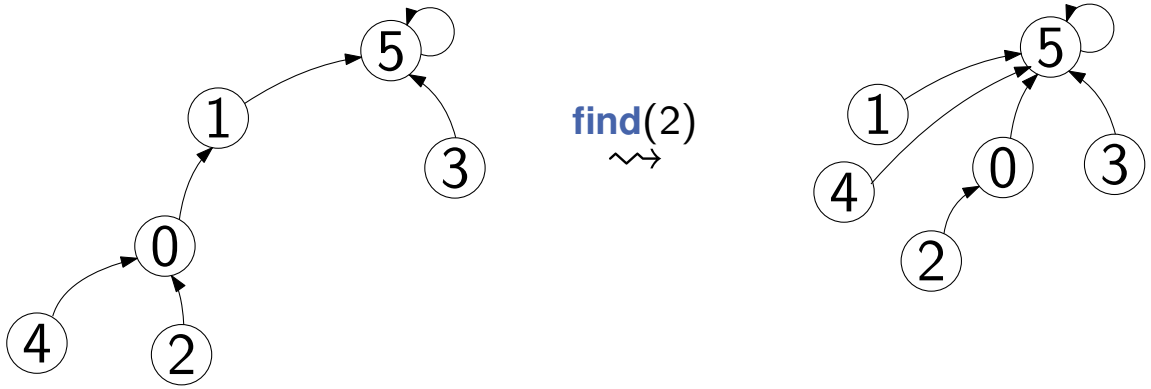
Pfad-Kompression

- verkürze Pfade falls möglich
  - häng Element direkt unter Repräsentant
  - insbesondere bei **find**



Pfad-Kompression

- verkürze Pfade falls möglich
  - häng Element direkt unter Repräsentant
  - insbesondere bei **find**

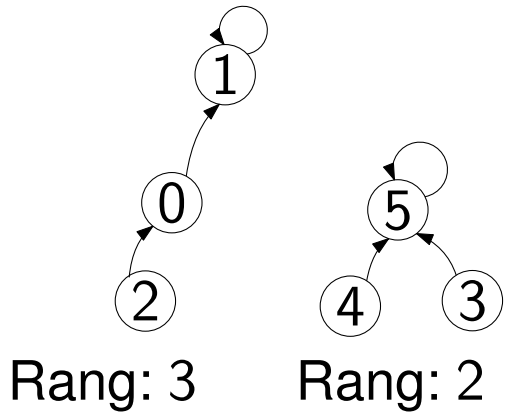


- $O(1)$ zusätzlicher Aufwand pro Element

$\Rightarrow$ nur tatsächlich genutzte Teilbäume werden aufgeräumt

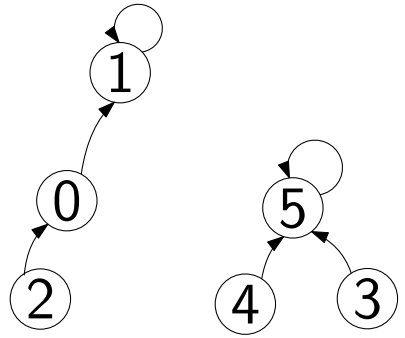
Union by Rank

- Rang einer Menge = „Länge des längsten Pfads“
- **find**$(u) \in \Theta($Rang von Menge von $u)$



Union by Rank

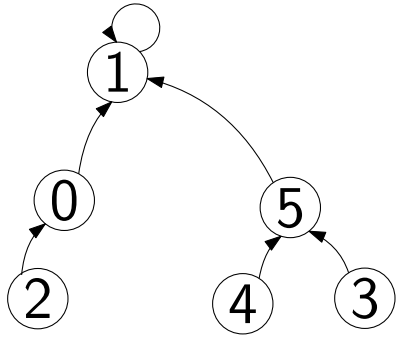
- Rang einer Menge = „Länge des längsten Pfads“
- **find**$(u) \in \Theta($Rang von Menge von $u)$



Rang: 3 Rang: 2

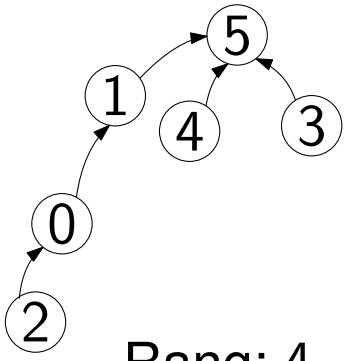
Beispiel: **union**$(2, 3)$

5 unter 1:



Rang: 3

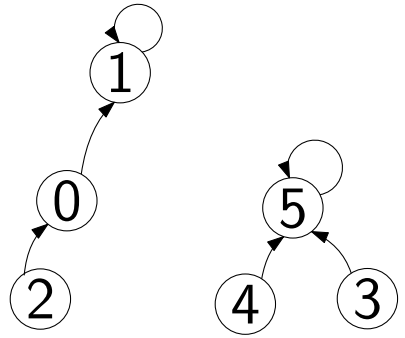
1 unter 5:



Rang: 4

Union by Rank

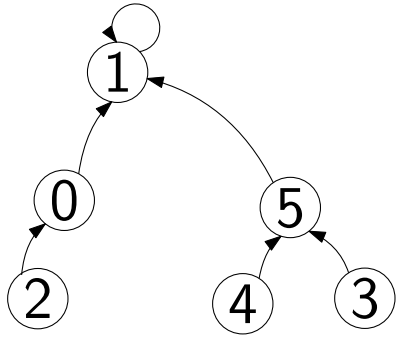
- Rang einer Menge = „Länge des längsten Pfads“
- **find**$(u) \in \Theta($Rang von Menge von $u)$



Rang: 3 Rang: 2

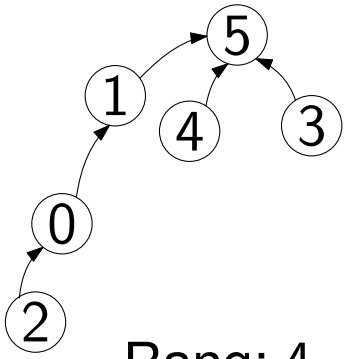
Beispiel: **union**$(2, 3)$

5 unter 1:



Rang: 3

1 unter 5:



Rang: 4

- Hänge immer flacheren Teilbaum unter tieferen

$\Rightarrow$ Rang wächst so möglichst langsam

Weiteres

- **union**, **find** $\in O(\log^* n) \sim O(1)$ (nur amortisiert)
  - nur für Pfadkompression/Union by Rank
  - Beweis in VL

Weiteres

- **union**, **find** $\in O(\log^* n) \sim O(1)$ (nur amortisiert)
  - nur für Pfadkompression/Union by Rank
  - Beweis in VL

Überprüfen Sie mithilfe einer Union-Find-Datenstruktur, ob ein ungerichteter Graph ein Baum ist.
Welche Laufzeit hat der Algorithmus?
Was sind die Vorteile/Nachteile gegenüber einer DFS?

Weiteres

- **union**, **find** $\in O(\log^* n) \sim O(1)$ (nur amortisiert)
  - nur für Pfadkompression/Union by Rank
  - Beweis in VL

Überprüfen Sie mithilfe einer Union-Find-Datenstruktur, ob ein ungerichteter Graph ein Baum ist.
Welche Laufzeit hat der Algorithmus?
Was sind die Vorteile/Nachteile gegenüber einer DFS?

- Erstelle eine Union-Find-Datenstruktur für alle Knoten
- durchlaufe alle Kanten (u, v)
  - Fall 1: **find**$(u) =$ **find**$(v) \Rightarrow$ Kante schließt Kreis $\Rightarrow$ kein Baum
  - Fall 2: **find**$(u) \neq$ **find**$(v) \Rightarrow$ **union**(u, v)
- Graph ist Baum $\Leftrightarrow$ Datenstruktur hat genau eine Menge

Weiteres

- **union**, **find** $\in O(\log^* n) \sim O(1)$ (nur amortisiert)
  - nur für Pfadkompression/Union by Rank
  - Beweis in VL

Überprüfen Sie mithilfe einer Union-Find-Datenstruktur, ob ein ungerichteter Graph ein Baum ist.
Welche Laufzeit hat der Algorithmus?
Was sind die Vorteile/Nachteile gegenüber einer DFS?

- Erstelle eine Union-Find-Datenstruktur für alle Knoten
- durchlaufe alle Kanten (u, v)
  - Fall 1: **find**$(u) =$ **find**$(v) \Rightarrow$ Kante schließt Kreis $\Rightarrow$ kein Baum
  - Fall 2: **find**$(u) \neq$ **find**$(v) \Rightarrow$ **union**(u, v)
- Graph ist Baum $\Leftrightarrow$ Datenstruktur hat genau eine Menge
- Laufzeit: $\Theta(m \log^* n + n)$

Vergleich

- mit DFS
  - asymptotisch schneller (max. Faktor $\log^* n$)
  - Speicherbedarf: $O(n + m)$
- mit union-find
  - Kantenliste reicht als Eingabe
  - Speicherbedarf: $O(n)$

Dynamische Programmierung

- Rekurrenzgleichung beschreibt Algorithmus

Beispiel: Floyd-Warshall-Algorithmus

$$sp(u, v, 0) = \begin{cases} c(u, v), u \neq v \wedge (u, v) \in E \\ 0, u = v \\ \infty, \text{sonst} \end{cases}$$

$$sp(u, v, k) = \min(sp(u, v, k-1), sp(u, k, k-1) + sp(k, v, k-1))$$

Dynamische Programmierung

- Rekurrenzgleichung beschreibt Algorithmus
- Unterschied zu Teile und Herrsche:
  - Zwischenergebnisse werden oft wiederverwendet

Beispiel: Floyd-Warshall-Algorithmus

$$sp(u, v, 0) = \begin{cases} c(u, v), u \neq v \wedge (u, v) \in E \\ 0, u = v \\ \infty, \text{sonst} \end{cases}$$

$$sp(u, v, k) = \min(sp(u, v, k-1), sp(u, k, k-1) + sp(k, v, k-1))$$

Dynamische Programmierung

- Rekurrenzgleichung beschreibt Algorithmus
- Unterschied zu Teile und Herrsche:
  - Zwischenergebnisse werden oft wiederverwendet
- Berechne alle Zwischenergebnisse & speichere sie
  - Reihenfolge hängt von Abhängigkeiten ab

$\Rightarrow$ Abwägung zwischen Laufzeit & Speicherbedarf

Beispiel: Floyd-Warshall-Algorithmus

$$sp(u, v, 0) = \begin{cases} c(u, v), u \neq v \wedge (u, v) \in E \\ 0, u = v \\ \infty, \text{sonst} \end{cases}$$

$$sp(u, v, k) = \min(sp(u, v, k-1), sp(u, k, k-1) + sp(k, v, k-1))$$

Beispiel: Longest common suffix

Gegeben: $w_1, w_2 \in \Sigma^*$
Gesucht: Zerlegung $p_1, p_2, s \in \Sigma^*$ mit $w_1 = p_1 s$, $w_2 = p_2 s$ mit $|s|$ maximal

Beispiel: Longest common suffix

Gegeben: $w_1, w_2 \in \Sigma^*$
Gesucht: Zerlegung $p_1, p_2, s \in \Sigma^*$ mit $w_1 = p_1 s$, $w_2 = p_2 s$ mit $|s|$ maximal

- Idee: DP über Länge des Suffix
- $S(k) =$„Längster Suffix mit Länge $\leq k$“
- $S(0) = 0$ (ε ist immer ein Suffix)

Beispiel: Longest common suffix

Gegeben: $w_1, w_2 \in \Sigma^*$
Gesucht: Zerlegung $p_1, p_2, s \in \Sigma^*$ mit $w_1 = p_1 s, w_2 = p_2 s$ mit $|s|$ maximal

- Idee: DP über Länge des Suffix
- $S(k) =$„Längster Suffix mit Länge $\leq k$“
- $S(0) = 0$ (ε ist immer ein Suffix)
- $S(k) = \begin{cases} k, S(k-1) = k-1 \wedge w_1(|w_1| - k) = w_2(|w_2| - k) \\ S(k-1), \text{sonst} \end{cases}$
- Gesucht: $S(\min(|w_1|, |w_2|))$

Beispiel: Longest common suffix

Gegeben: $w_1, w_2 \in \Sigma^*$
Gesucht: Zerlegung $p_1, p_2, s \in \Sigma^*$ mit $w_1 = p_1 s, w_2 = p_2 s$ mit $|s|$ maximal

- Idee: DP über Länge des Suffix
- $S(k) =$„Längster Suffix mit Länge $\leq k$“
- $S(0) = 0$ (ε ist immer ein Suffix)
- $S(k) = \begin{cases} k, S(k-1) = k-1 \wedge w_1(|w_1| - k) = w_2(|w_2| - k) \\ S(k-1), \text{sonst} \end{cases}$
- Gesucht: $S(\min(|w_1|, |w_2|))$

Fall 1: $|s| \geq k-1$ und $k$tes Zeichen gleich $\Rightarrow |s| \geq k$
Fall 2: Längerer Suffix als $|s| = S(k-1)$ nicht möglich

Beispiel: Longest common suffix

Gegeben: $w_1, w_2 \in \Sigma^*$
Gesucht: Zerlegung $p_1, p_2, s \in \Sigma^*$ mit $w_1 = p_1 s, w_2 = p_2 s$ mit $|s|$ maximal

- Idee: DP über Länge des Suffix
- $S(k) =$„Längster Suffix mit Länge $\leq k$“
- $S(0) = 0$ (ε ist immer ein Suffix)
- $S(k) = \begin{cases} k, S(k-1) = k-1 \wedge w_1(|w_1| - k) = w_2(|w_2| - k) \\ S(k-1), \text{sonst} \end{cases}$
- Gesucht: $S(\min(|w_1|, |w_2|))$

Fall 1: $|s| \geq k-1$ und $k$tes Zeichen gleich $\Rightarrow |s| \geq k$
Fall 2: Längerer Suffix als $|s| = S(k-1)$ nicht möglich

- keine gemeinsamen Basisfälle
- iterative & rekursive Lösung nahezu gleich

Longest common substring

Gegeben: $w_1, w_2 \in \Sigma^*$
Gesucht: Zerlegung $p_1, p_2, s_1, s_2, c \in \Sigma^*$ mit $w_1 = p_1 c s_1 \wedge w_2 = p_2 c s_2$ mit $|c|$ maximal
Beispiel: $w_1 =$ Algorithmen, $w_2 =$ Logarithmus $\Rightarrow c =$ rithm

Longest common substring

Gegeben: $w_1, w_2 \in \Sigma^*$
Gesucht: Zerlegung $p_1, p_2, s_1, s_2, c \in \Sigma^*$ mit $w_1 = p_1 c s_1 \wedge w_2 = p_2 c s_2$ mit $|c|$ maximal
Beispiel: $w_1 =$ Algorithmen, $w_2 =$ Logarithmus $\Rightarrow c =$ rithm

- Idee: Jeder Substring hört an einem Index auf

Definiere **LCS**$(i, j) =$„Längster Suffix von $w_1(0) \ldots w_1(i)$ und $w_2(0) \ldots w_2(j)$“

Longest common substring

Gegeben: $w_1, w_2 \in \Sigma^*$
Gesucht: Zerlegung $p_1, p_2, s_1, s_2, c \in \Sigma^*$ mit $w_1 = p_1 c s_1 \wedge w_2 = p_2 c s_2$ mit $|c|$ maximal
Beispiel: $w_1 =$ Algorithmen, $w_2 =$ Logarithmus $\Rightarrow c =$ rithm

- Idee: Jeder Substring hört an einem Index auf

Definiere **LCS**$(i, j) =$„Längster Suffix von $w_1(0) \dots w_1(i)$ und $w_2(0) \dots w_2(j)$“

- **LCS**$(0, j) = 0$
- **LCS**$(i, 0) = 0$
- **LCS**$(i, j) = \begin{cases} LCS(i-1, j-1) + 1, w_1(i) = w_2(j) \\ 0, \text{sonst} \end{cases}$

Gesucht: $\max_{0 \le i < |w_1|, 0 \le j < |w_2|}$ **LCS**(i, j)

Longest common substring

Gegeben: $w_1, w_2 \in \Sigma^*$
Gesucht: Zerlegung $p_1, p_2, s_1, s_2, c \in \Sigma^*$ mit $w_1 = p_1 c s_1 \wedge w_2 = p_2 c s_2$ mit $|c|$ maximal
Beispiel: $w_1 =$ Algorithmen, $w_2 =$ Logarithmus $\Rightarrow c =$ rithm

- Idee: Jeder Substring hört an einem Index auf

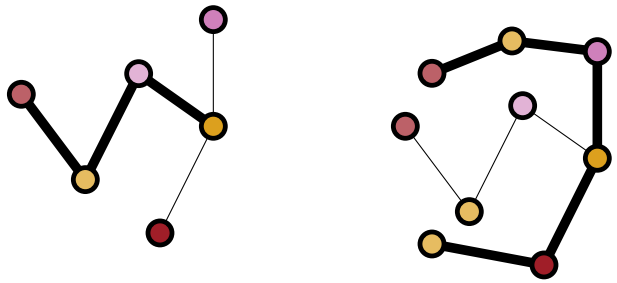
Definiere **LCS**$(i, j) =$„Längster Suffix von $w_1(0) \dots w_1(i)$ und $w_2(0) \dots w_2(j)$“

- **LCS**$(0, j) = 0$
- **LCS**$(i, 0) = 0$
- **LCS**$(i, j) = \begin{cases} LCS(i-1, j-1) + 1, w_1(i) = w_2(j) \\ 0, \text{sonst} \end{cases}$

Gesucht: $\max_{0 \leq i < |w_1|, 0 \leq j < |w_2|}$ **LCS**(i, j)

- wenige gemeinsame Zwischenergebnisse

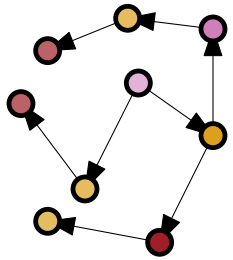
Längste Pfade in ungerichteten Bäumen

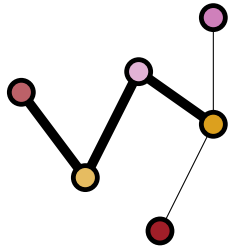


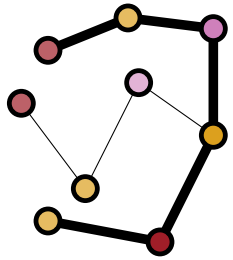
Längste Pfade in ungerichteten Bäumen

Beispielalgorithmus:

- Wähle einen Knoten als Wurzel
- Definiere DP über Baumstruktur:







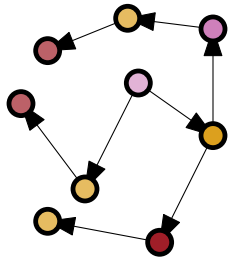
Längste Pfade in ungerichteten Bäumen

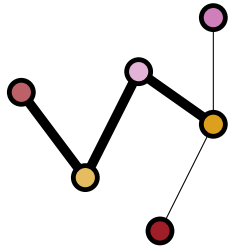
Beispielalgorithmus:

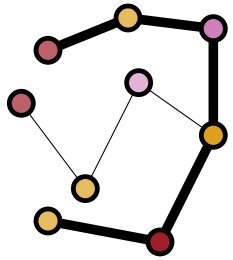
- Wähle einen Knoten als Wurzel
- Definiere DP über Baumstruktur:

max. Pfad in Teilbaum unter v = Maximum aus:

- max. Pfad zu Kindknoten $u + 1$
- max. Pfad in Teilbaum von Kindknoten u
- max. Pfad zu Kind u + max. Pfad zu Kind $v + 2$



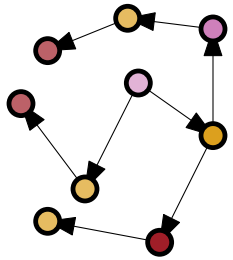


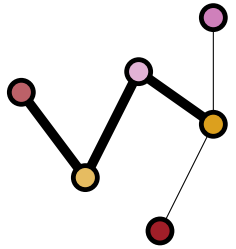


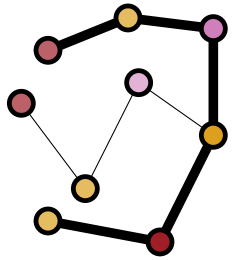
Längste Pfade in ungerichteten Bäumen

Beispielalgorithmus:

- Wähle einen Knoten als Wurzel
- Definiere DP über Baumstruktur:







max. Pfad in Teilbaum unter v = Maximum aus:

- max. Pfad zu Kindknoten $u + 1$
- max. Pfad in Teilbaum von Kindknoten u
- max. Pfad zu Kind u + max. Pfad zu Kind $v + 2$

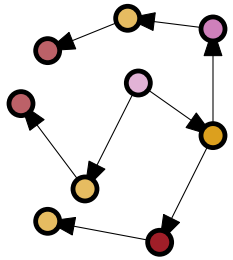
- max. Pfad zu Knoten u = Höhe des Teilbaums unter u

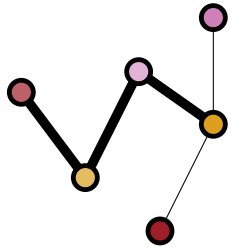
$\Rightarrow$ Teilproblem ist ebenfalls DP

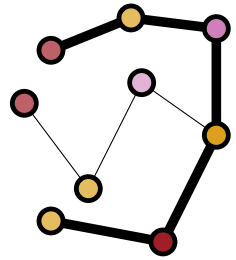
Längste Pfade in ungerichteten Bäumen

Beispielalgorithmus:

- Wähle einen Knoten als Wurzel
- Definiere DP über Baumstruktur:







max. Pfad in Teilbaum unter v = Maximum aus:

- max. Pfad zu Kindknoten $u + 1$
- max. Pfad in Teilbaum von Kindknoten u
- max. Pfad zu Kind u + max. Pfad zu Kind $v + 2$

- max. Pfad zu Knoten u = Höhe des Teilbaums unter u

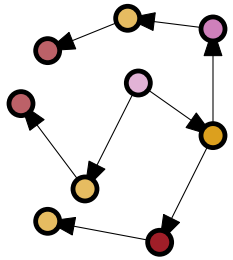
$\Rightarrow$ Teilproblem ist ebenfalls DP

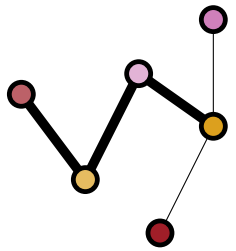
- Frage: Können hier Zwischenergebnisse wiederverwendet werden?

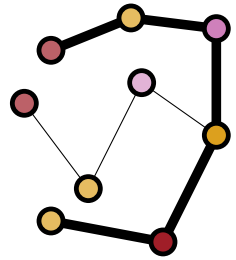
Längste Pfade in ungerichteten Bäumen

Beispielalgorithmus:

- Wähle einen Knoten als Wurzel
- Definiere DP über Baumstruktur:







max. Pfad in Teilbaum unter v = Maximum aus:

- max. Pfad zu Kindknoten $u + 1$
- max. Pfad in Teilbaum von Kindknoten u
- max. Pfad zu Kind u + max. Pfad zu Kind $v + 2$

- max. Pfad zu Knoten u = Höhe des Teilbaums unter u

$\Rightarrow$ Teilproblem ist ebenfalls DP

- Frage: Können hier Zwischenergebnisse wiederverwendet werden?
- Jeder Knoten wird nur einmal betrachtet

$\Rightarrow$ rekursive Lösung auch schnell (vgl. Tiefensuche)

Aufgabe

Gegeben Folge von Matrizen A_i mit $A_i \in \mathbb{R}^{n_i \times n_{i+1}}, 0 \leq i < k$
Gesucht ist die Reihenfolge in der $A_1 \cdot A_2 \ldots A_k$ berechnet werden muss, sodass der Gesamtaufwand minimal ist.
Beschreiben Sie das Problem als DP

Der Aufwand für eine Matrixmultiplikation ist $n \cdot m \cdot r$ für AB mit $A \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^{m \times r}$
Beispiel: $(AB)C$ kostet $nmr + nrk = nr(m + k)$ und $A(BC)$ kostet $mrk + nmk = mk(r + n)$
$(A \in \mathbb{R}^{n \times m}, B \in \mathbb{R}^{m \times r}, C \in \mathbb{R}^{r \times k})$

Aufgabe

Gegeben Folge von Matrizen A_i mit $A_i \in \mathbb{R}^{n_i \times n_{i+1}}, 0 \leq i < k$
Gesucht ist die Reihenfolge in der $A_1 \cdot A_2 \ldots A_k$ berechnet werden muss, sodass der Gesamtaufwand minimal ist.
Beschreiben Sie das Problem als DP

Der Aufwand für eine Matrixmultiplikation ist $n \cdot m \cdot r$ für AB mit $A \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^{m \times r}$
Beispiel: $(AB)C$ kostet $nmr + nrk = nr(m + k)$ und $A(BC)$ kostet $mrk + nmk = mk(r + n)$
$(A \in \mathbb{R}^{n \times m}, B \in \mathbb{R}^{m \times r}, C \in \mathbb{R}^{r \times k})$

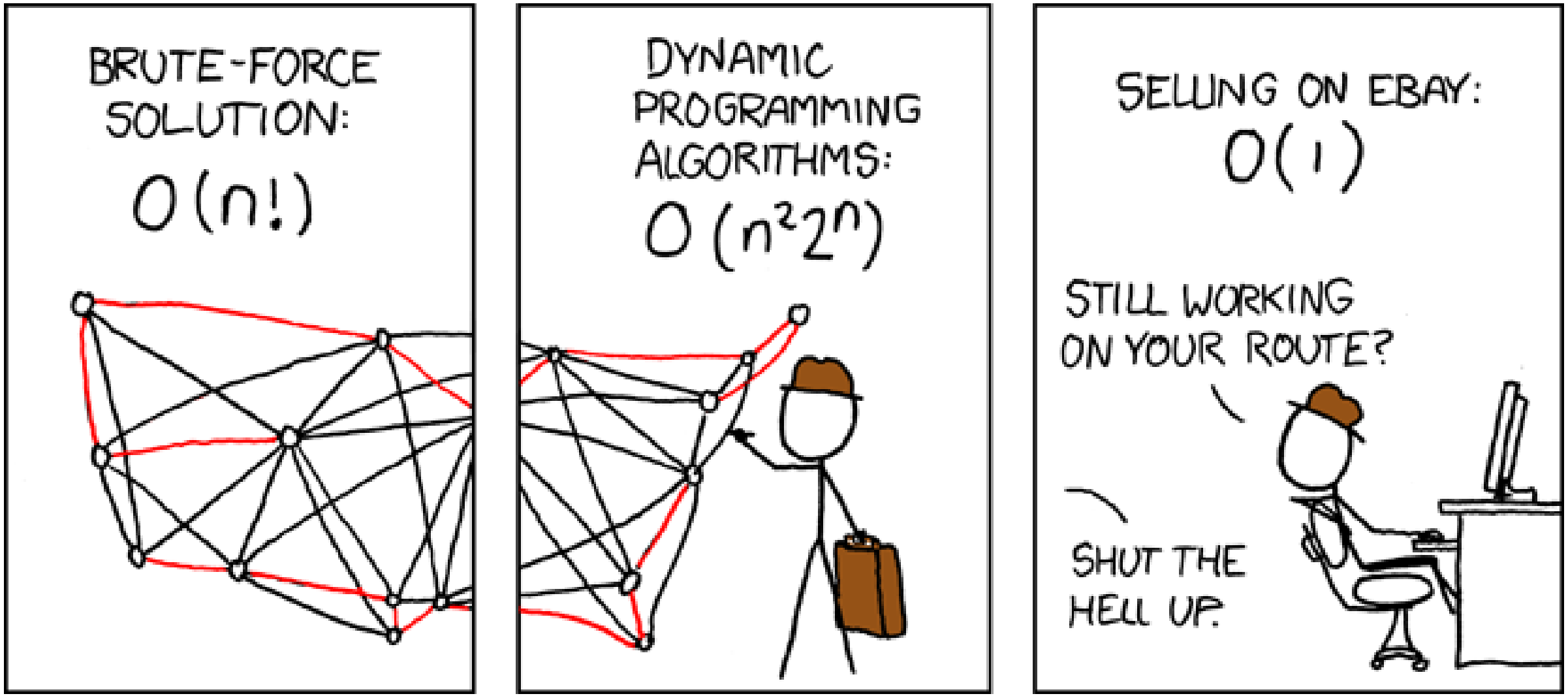
- Setze: $M(i, j, r) =$„Werte Matrizen i bis j aus und klammere bei r“
- $R(i, j) = \min_{i \leq k < j} M(i, j, k)$
- $M(i, j, r) = n_r n_{r+1} n_{r+2} + R(i, r - 1) + R(r + 1, j)$
- $M(i, i + 1, i) = n_i n_{i+1} n_{i+2} (0 \leq i < k)$

Fragen?

Karlsruher Institut für Technologie

Fragen!

Ende



https://xkcd.com/399/