Karlsruhe Institute of Technology

Karlsruher Institut für Technologie

5. Zusatztutorium
Tiefensuche, Dynamische Programmierung

Algorithmen I, 5. Zusatztutorium

Henriette Färber | August 28, 2023

Themen für heute

Wo sind wir?

Motivation

Wir wollen einen Graphen erkunden!

Motivation

Wir wollen einen Graphen erkunden!

Aber haben wir dazu nicht schon die Breitensuche (BFS)?

Motivation

Wir wollen einen Graphen erkunden!

Aber haben wir dazu nicht schon die Breitensuche (BFS)?

- Startet bei beliebigen Startknoten s
- Erkundet Graphen schichtweise
- Auf ungewichteten Graphen: kürzeste Wege von s aus
- Allgemein: Knoten geordnet nach Distanz / Anzahl Kanten zwischen ihnen und s

Motivation

Wir wollen einen Graphen erkunden!

Aber haben wir dazu nicht schon die Breitensuche (BFS)?

- Startet bei beliebigen Startknoten s
- Erkundet Graphen schichtweise
- Auf ungewichteten Graphen: kürzeste Wege von s aus
- Allgemein: Knoten geordnet nach Distanz / Anzahl Kanten zwischen ihnen und s

Was wollen wir?

Eine Knotenordnung, die nicht (nur) vom Abstand zu einem Startknoten abhängt und Erreichbarkeit zwischen Knoten abbildet. Auf gerichteten Graphen: eine topologische Sortierung (wenn möglich).

Tiefensuche (depth-first search)

Idee: stets so weit absteigen wie möglich

- Oder anders: setze Weg entlang unentdeckter Knoten möglichst lang fort
- Beginnend bei Startknoten s
- Wähle stets (beliebigen) noch nicht besuchten Nachbarn
- Falls alle Nachbarn besucht: eine Ebene nach oben

Tiefensuche (depth-first search)

Idee: stets so weit absteigen wie möglich

- Oder anders: setze Weg entlang unentdeckter Knoten möglichst lang fort
- Beginnend bei Startknoten s
- Wähle stets (beliebigen) noch nicht besuchten Nachbarn
- Falls alle Nachbarn besucht: eine Ebene nach oben

In Pseudocode gegossen:

```
1: DFS(G = (V, E) : Graph, v : Node)
2:     COLOR(v)
3:     for u ∈ N(v) do
4:         if ¬COLORED(u) then
5:             DFS(G, u)
```

Tiefensuche (depth-first search)

```
1: DFS(G = (V, E) : Graph, v : Node)
2:     COLOR(v)
3:     for u ∈ N(v) do
4:         if ¬COLORED(u) then
5:             DFS(G, u)
```

Das an sich ist noch wenig spannend. Wir wollen uns ja eine Knotenordnung berechnen! Was brauchen wir dafür?

Tiefensuche (depth-first search)

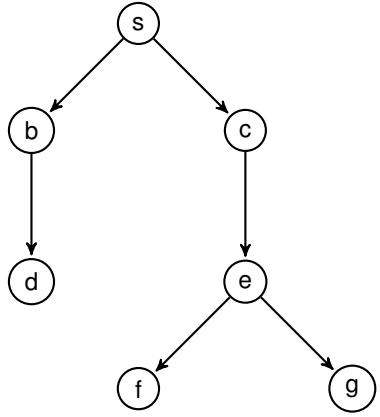
```
1: DFS(G = (V, E) : Graph, v : Node)
2:     COLOR(v)
3:     for u ∈ N(v) do
4:         if ¬COLORED(u) then
5:             parent[u] = v
6:             DFS(G, u)
7:     FIN[v] := curr
8:     curr := curr + 1
```

Für jeden Knoten v merken wir uns:

- Seinen Elter-Knoten `parent`(v)
- Wann er vollständig abgearbeitet wurde, dazu `FIN`(v) (gleich mehr dazu)

Kantentypen

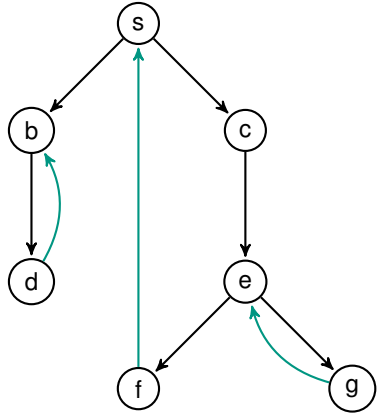
- Baumkante



Tiefensuchreihenfolge: s, b, d, c, e, f, g

Kantentypen

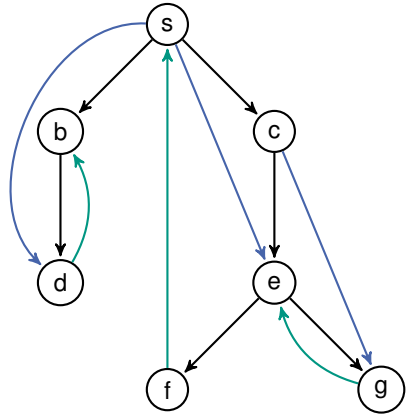
- Baumkante
- Rückkante zu einem Vorgänger



Tiefensuchreihenfolge: s, b, d, c, e, f, g

Kantentypen

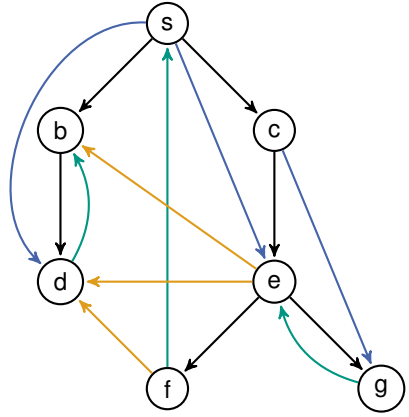
- Baumkante
- Rückkante zu einem Vorgänger
- Vorkante zu einem Nachfolger im selben Teilbaum



Tiefensuchreihenfolge: s, b, d, c, e, f, g

Kantentypen

- Baumkante
- Rückkante zu einem Vorgänger
- Vorkante zu einem Nachfolger im selben Teilbaum
- Querkante in einen bereits abgearbeiteten Teilbaum hinein



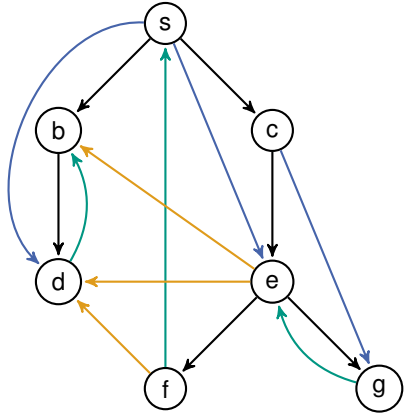
Tiefensuchreihenfolge: s, b, d, c, e, f, g

Kantentypen

- Baumkante
- Rückkante zu einem Vorgänger
- Vorkante zu einem Nachfolger im selben Teilbaum
- Querkante in einen bereits abgearbeiteten Teilbaum hinein

Querkanten können bei DFS auf ungerichteten Graphen nicht auftreten.

Warum?



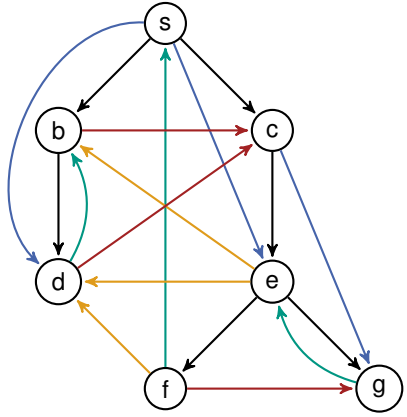
Tiefensuchreihenfolge: s, b, d, c, e, f, g

Kantentypen

- Baumkante
- Rückkante zu einem Vorgänger
- Vorkante zu einem Nachfolger im selben Teilbaum
- Querkante in einen bereits abgearbeiteten Teilbaum hinein

Querkanten können bei DFS auf ungerichteten Graphen nicht auftreten.

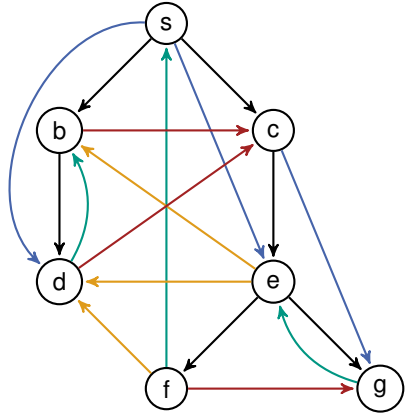
Eine Querkante (u, v) zeigt an, dass v von u aus nicht erreichbar war, als u abgearbeitet wurde.



Tiefensuchreihenfolge: s, b, d, c, e, f, g

Kantentypen

- Baumkante
- Rückkante zu einem Vorgänger
- Vorkante zu einem Nachfolger im selben Teilbaum
- Querkante in einen bereits abgearbeiteten Teilbaum hinein
- Illegale Kanten können nicht auftreten



Tiefensuchreihenfolge: s, b, d, c, e, f, g

Was wollen wir über Knoten wissen?

Für BFS berechnen wir für nur Knoten ihren Elter und ihre Entfernung zum Startknoten. Für DFS:

- Elter-Knoten (und darüber die Lage im DFS-Baum)

Was wollen wir über Knoten wissen?

Für BFS berechnen wir für nur Knoten ihren Elter und ihre Entfernung zum Startknoten. Für DFS:

- Elter-Knoten (und darüber die Lage im DFS-Baum)
- **DFS-Nummer**, wird bei Entdeckung vergeben
- **FIN-Nummer**, wird vergeben, wenn vollständig abgearbeitet
- **low-Wert**, die kleinste aus unterliegendem Teilbaum erreichbare DFS-Nummer

Im Folgenden: Aller Werte inklusive globalen Zählern mit 0 initialisiert

Was wollen wir über Knoten wissen?

Für BFS berechnen wir für nur Knoten ihren Elter und ihre Entfernung zum Startknoten. Für DFS:

- Elter-Knoten (und darüber die Lage im DFS-Baum)
- **DFS-Nummer**, wird bei Entdeckung vergeben
- **FIN-Nummer**, wird vergeben, wenn vollständig abgearbeitet
- **low-Wert**, die kleinste aus unterliegendem Teilbaum erreichbare DFS-Nummer

Erinnerung: unser rekursiver Ansatz für DFS

```
1: DFS(G = (V, E) : Graph, v : Node)
2:     COLOR(v)
3:     for u ∈ N(v) do
4:         if ¬COLORED(u) then
5:             DFS(G, u)
```

Wann berechnen wir diese Werte in unserem Algorithmus?

DFS-Nummer

- Beschreibt den Zeitpunkt, zu dem ein Knoten das erste Mal besucht wird
- Deswegen auch: Nachfolger im DFS-Baum haben ausschließlich höhere DFS-Nummern
- Berechnung über globalen Zähler
- im Pseudocode: `dfsNum`

DFS-Nummer

- Beschreibt den Zeitpunkt, zu dem ein Knoten das erste Mal besucht wird
- Deswegen auch: Nachfolger im DFS-Baum haben ausschließlich höhere DFS-Nummern
- Berechnung über globalen Zähler
- im Pseudocode: `dfsNum`

```
1: DFS(G = (V, E) : Graph, v : Node)
2:     COLOR(v)
3:     dfsNum[v] := dfsCounter
4:     dfsCounter := dfsCounter + 1
5:     for u ∈ N(v) do
6:         if ¬COLORED(u) then
7:             DFS(G, u)
```

FIN-Nummer

- Beschreibt den Zeitpunkt, zu dem ein Knoten vollständig abgearbeitet ist
- Deswegen auch: Nachfolger im DFS-Baum haben ausschließlich niedrigere FIN-Nummern
- Berechnung über globalen Zähler
- im Pseudocode: `FIN`

FIN-Nummer

- Beschreibt den Zeitpunkt, zu dem ein Knoten vollständig abgearbeitet ist
- Deswegen auch: Nachfolger im DFS-Baum haben ausschließlich niedrigere FIN-Nummern
- Berechnung über globalen Zähler
- im Pseudocode: `FIN`

```
1: DFS(G = (V, E) : Graph, v : Node)
2:     COLOR(v)
3:     for u ∈ N(v) do
4:         if ¬COLORED(u) then
5:             DFS(G, u)
6:     FIN[v] := finCounter
7:     finCounter := finCounter + 1
```

FIN-Nummer

- Beschreibt den Zeitpunkt, zu dem ein Knoten vollständig abgearbeitet ist
- Deswegen auch: Nachfolger im DFS-Baum haben ausschließlich niedrigere FIN-Nummern
- Berechnung über globalen Zähler
- im Pseudocode: `FIN`

Gegeben sei ein Knoten $v \in V$. Wie finden wir nur anhand von DFS- und FIN-Nummer heraus, welche Knoten die Kinder von v im DFS-Baum sind?

FIN-Nummer

- Beschreibt den Zeitpunkt, zu dem ein Knoten vollständig abgearbeitet ist
- Deswegen auch: Nachfolger im DFS-Baum haben ausschließlich niedrigere FIN-Nummern
- Berechnung über globalen Zähler
- im Pseudocode: `FIN`

Gegeben sei ein Knoten $v \in V$. Wie finden wir nur anhand von DFS- und FIN-Nummer heraus, welche Knoten die Kinder von v im DFS-Baum sind?

Es gilt für Knoten $u, v \in V$:

$$\texttt{dfsNum}[v] < \texttt{dfsNum}[u] \wedge \mathit{FIN}[v] > \mathit{FIN}[u] \Rightarrow u \text{ ist Kind von } v$$

low-Wert

- `low`(v) ist Minimum von `dfsNum`(v) und **kleinster DFS-Nummer, die aus dem Teilbaum unter v über eine Rückkante erreichbar ist**
- Berechnung über die DFS-Nummern (Beachte: alle relevanten Nummern bereits korrekt gesetzt)
- im Pseudocode: `low`

low-Wert

- `low`(v) ist Minimum von `dfsNum`(v) und **kleinster DFS-Nummer, die aus dem Teilbaum unter v über eine Rückkante erreichbar ist**
- Berechnung über die DFS-Nummern (Beachte: alle relevanten Nummern bereits korrekt gesetzt)
- im Pseudocode: `low`

```
1: DFS(G = (V, E) : Graph, v : Node)
2:     COLOR(v)
3:     low[v] := dfsNum[v]
4:     for u ∈ N(v) do
5:         if ¬COLORED(u) then
6:             DFS(G, u)
7:             low[v] := min{low[v], low[u]          // Baumkante
8:         else
9:             low[v] := min{low[v], low[u]          // Nichtbaumkante
```

Und nun alles zusammen:

```
1:  DFS(G = (V, E) : Graph, v : Node)
2:      COLOR(v)
3:      dfsNum[v] := dfsCounter
4:      dfsCounter := dfsCounter + 1
5:      low[v] := dfsNum[v]
6:      for u ∈ N(v) do
7:          if ¬COLORED(u) then
8:              DFS(G, u)
9:              low[v] := min{low[v], low[u]}          // Baumkante
10:         else
11:             low[v] := min{low[v], low[u]}          // Nichtbaumkante
12:     FIN[v] := finCounter
13:     finCounter := finCounter + 1
```

Beispiel

Knoten:

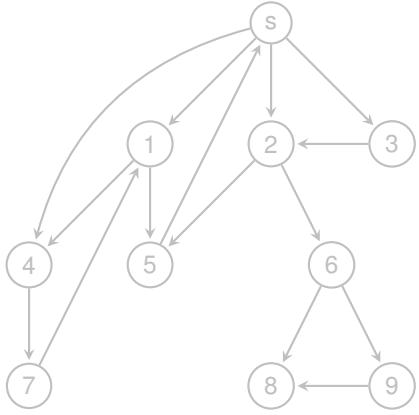
`dfsNum`

`FIN`

`low`

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Beispiel

Knoten:

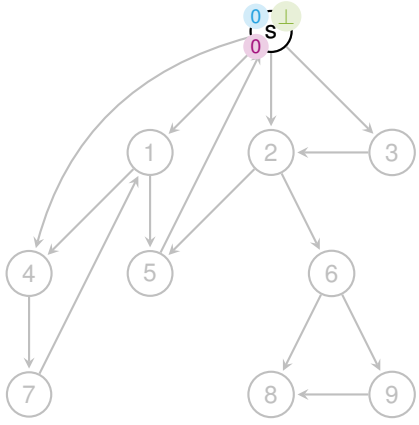
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Starte bei Startknoten *s*

Überspringen

Beispiel

Knoten:

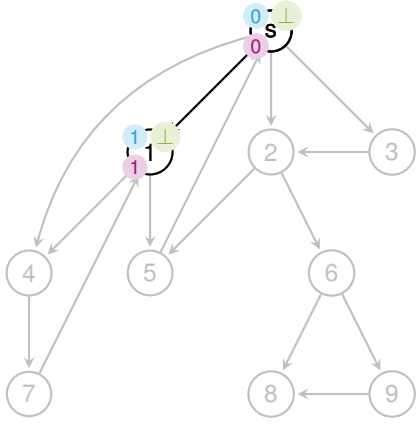
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Verfolge Baumkante $(s, 1)$

Beispiel

Knoten:

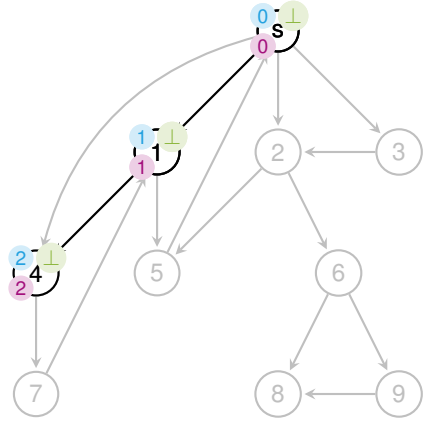
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Verfolge Baumkante $(1, 4)$

Beispiel

Knoten:

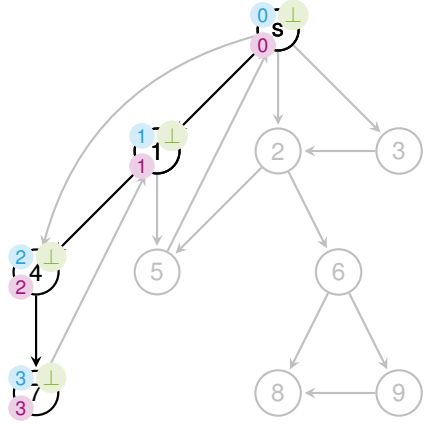
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Verfolge Baumkante $(4, 7)$

Überspringen

Beispiel

Knoten:

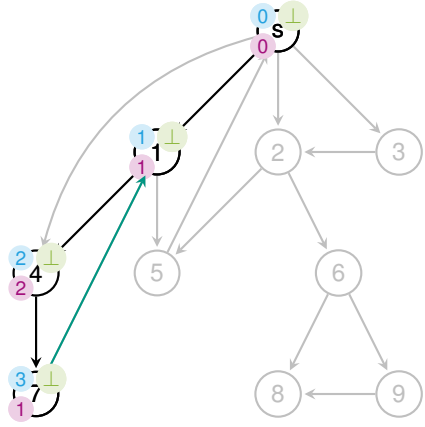
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Entdecke Rückkante $(4, 1)$, aktualisiere $\texttt{low}[4] := 1$

Beispiel

Knoten:

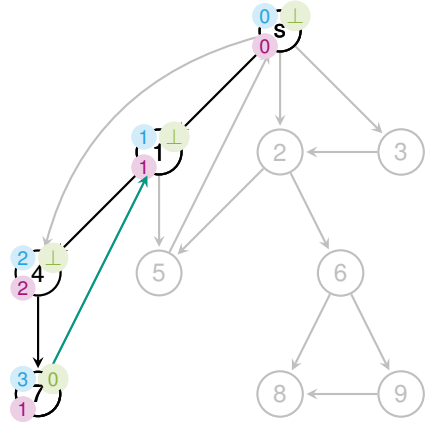
`dfsNum`

`FIN`

`low`

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Finalisiere 7 und aktualisiere `FIN`[7] := 0, weiter mit Elter 4

Beispiel

Knoten:

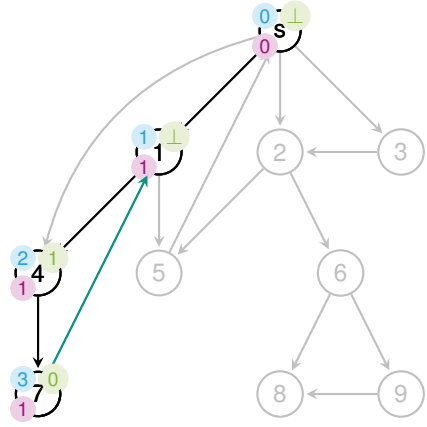
`dfsNum`

`FIN`

`low`

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Finalisiere 4 und aktualisiere `FIN`[4] := 1, weiter mit Elter 1

Beispiel

Knoten:

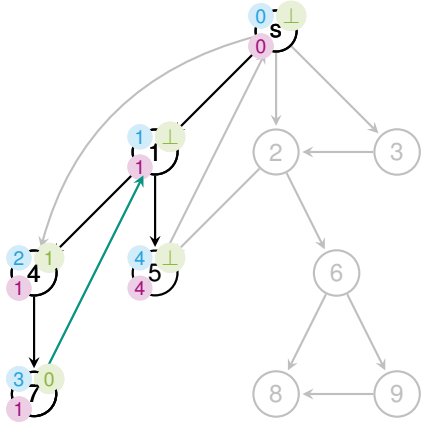
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Verfolge Baumkante $(1, 5)$

Überspringen

Beispiel

Knoten:

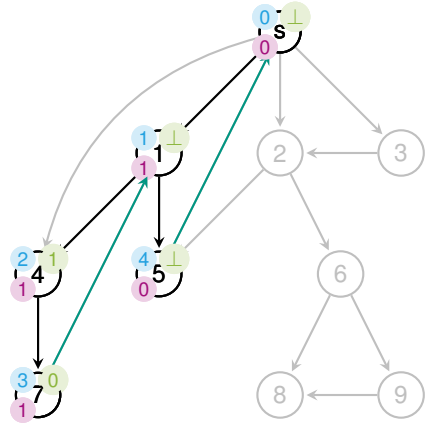
`dfsNum`

`FIN`

`low`

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Entdecke Rückkante $(5, s)$, aktualisiere $\texttt{low}[5] := 0$

Beispiel

Knoten:

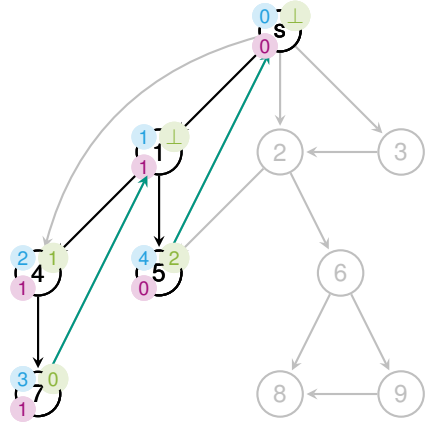
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Finalisiere 5 und aktualisiere FIN[5] := 2, weiter mit Elter 1

Überspringen

Beispiel

Knoten:

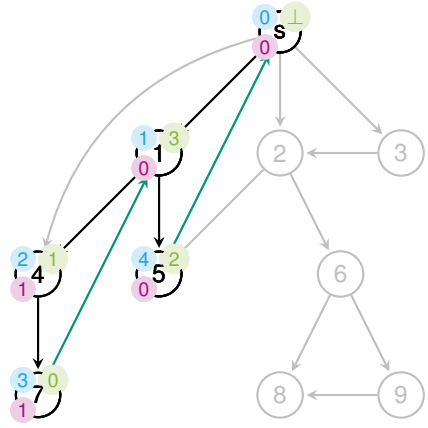
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Finalisiere 1 und aktualisiere FIN[1] := 3, weiter mit Elter *s*

Beispiel

Knoten:

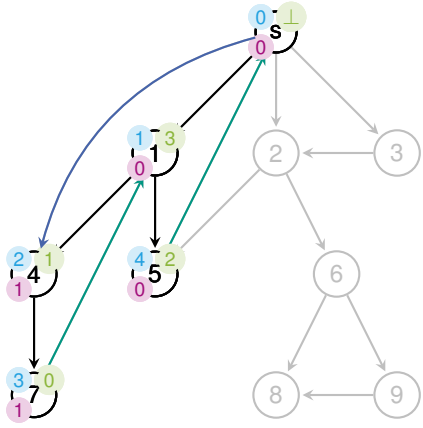
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Entdecke Vorkante $(s, 4)$

Beispiel

Knoten:

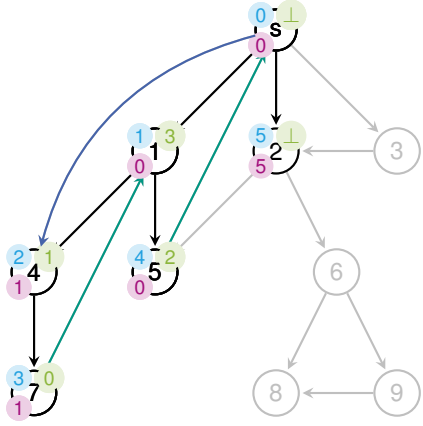
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Verfolge Baumkante $(s, 2)$

Überspringen

Beispiel

Knoten:

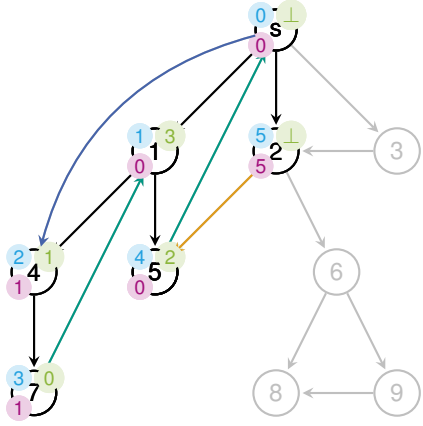
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Entdecke Querkante $(2, 5)$

Beispiel

Knoten:

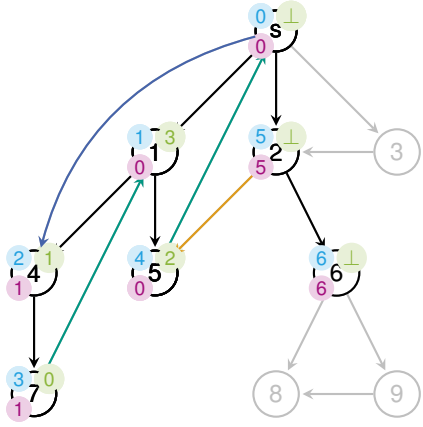
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Verfolge Baumkante $(2, 6)$

Beispiel

Knoten:

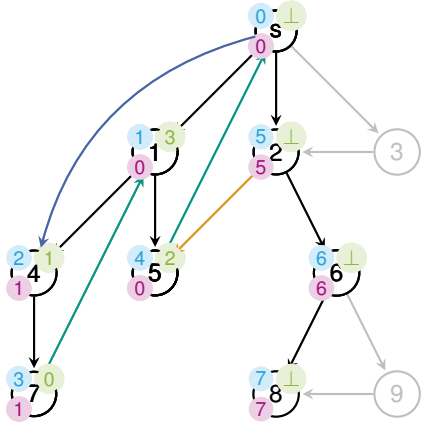
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Verfolge Baumkante $(6, 8)$

Überspringen

Beispiel

Knoten:

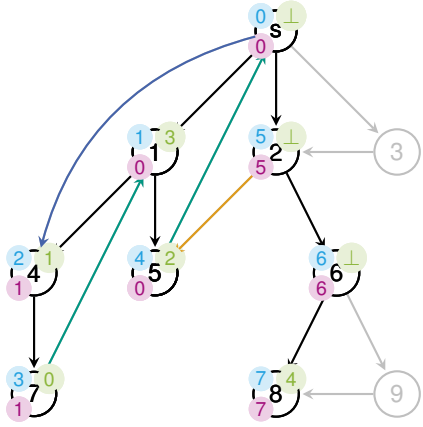
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Finalisiere 8 und aktualisiere FIN[8] := 4, weiter mit Elter 6

Beispiel

Knoten:

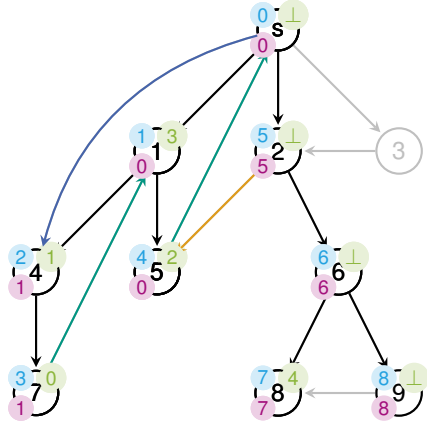
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Verfolge Baumkante $(6, 9)$

Beispiel

Knoten:

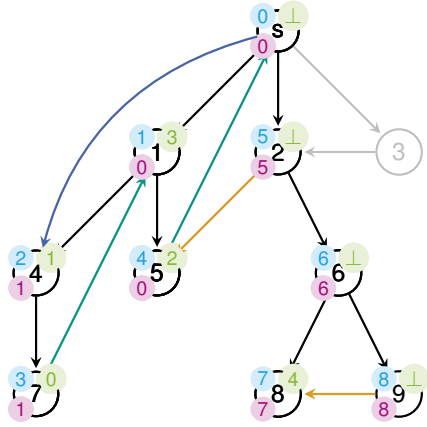
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Entdecke Querkante $(9, 8)$

Beispiel

Knoten:

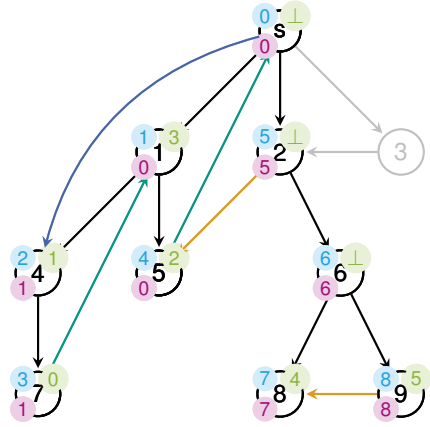
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Finalisiere 9 und aktualisiere FIN[9] := 5, weiter mit Elter 6

Beispiel

Knoten:

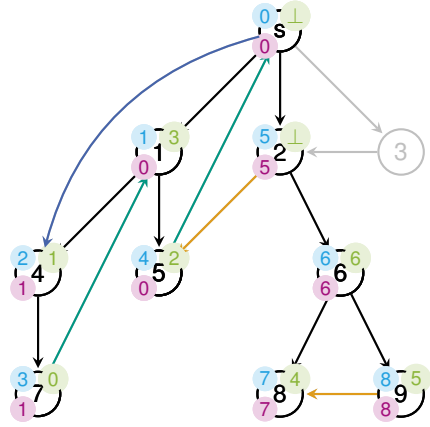
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Finalisiere 6 und aktualisiere FIN[6] ≔ 6, weiter mit Elter 2

Beispiel

Knoten:

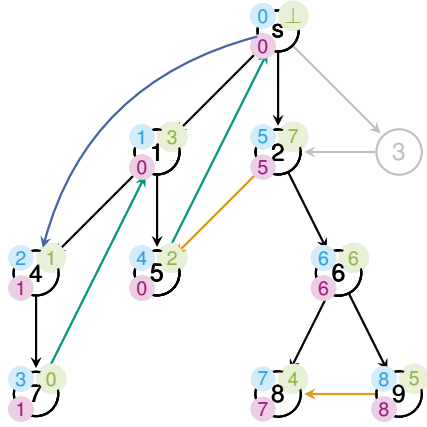
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Finalisiere 2 und aktualisiere FIN[2] := 7, weiter mit Elter *s*

Beispiel

Knoten:

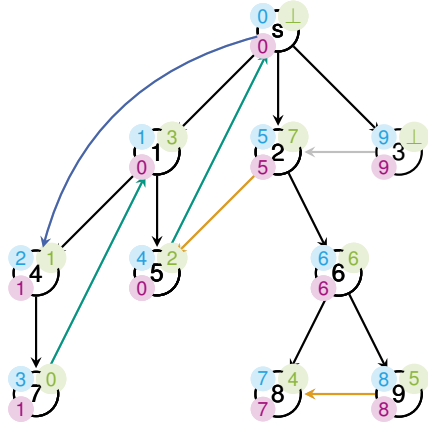
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Verfolge Baumkante $(s, 3)$

Überspringen

Beispiel

Knoten:

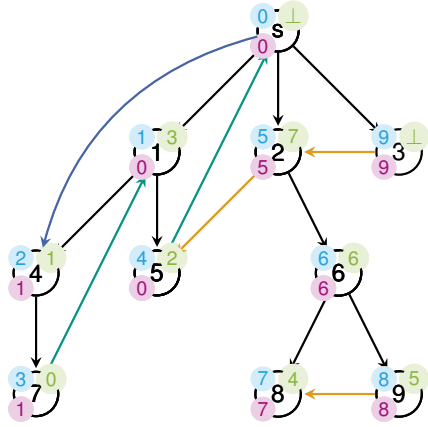
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Entdecke Querkante $(3, 2)$

Überspringen

Beispiel

Knoten:

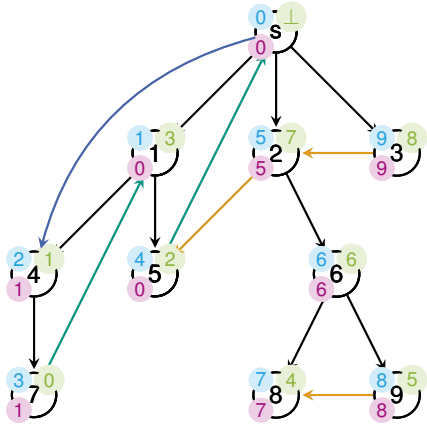
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Finalisiere 3 und aktualisiere FIN[3] := 8, weiter mit Elter *s*

Beispiel

Knoten:

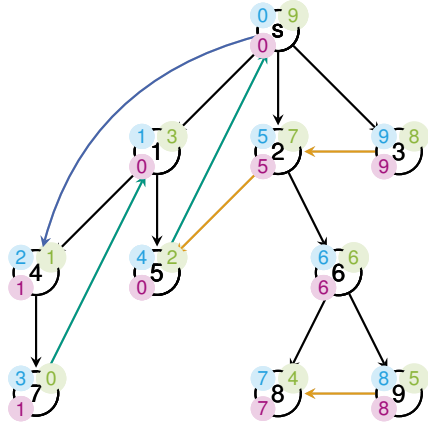
dfsNum

FIN

low

Kanten:

- unbekannt
- Baumkante
- Rückkante
- Vorkante
- Querkante



Finalisiere *s*

Kantentypen bestimmen

Gegeben eine Kante $(u, v) \in E$, wie können wir ihren Kantentyp bestimmen?

Kantentypen bestimmen

Gegeben eine Kante $(u, v) \in E$, wie können wir ihren Kantentyp bestimmen?

Kantentyp	DFS-Nummer	FIN-Nummer	Anmerkung
Baumkante	klein $\rightarrow$ groß	groß $\rightarrow$ klein	$\texttt{parent}[u] = v$
Vorkante	klein $\rightarrow$ groß	groß $\rightarrow$ klein	$\texttt{parent}[u] \neq v$
Rückkante	groß $\rightarrow$ klein	klein $\rightarrow$ groß	—
Querkante	groß $\rightarrow$ klein	groß $\rightarrow$ klein	—

Kantentypen bestimmen

Gegeben eine Kante $(u, v) \in E$, wie können wir ihren Kantentyp bestimmen?

Kantentyp	DFS-Nummer	FIN-Nummer	Anmerkung
Baumkante	klein $\rightarrow$ groß	groß $\rightarrow$ klein	$\texttt{parent}[u] = v$
Vorkante	klein $\rightarrow$ groß	groß $\rightarrow$ klein	$\texttt{parent}[u] \neq v$
Rückkante	groß $\rightarrow$ klein	klein $\rightarrow$ groß	—
Querkante	groß $\rightarrow$ klein	groß $\rightarrow$ klein	—

Wichtige Erkenntnis

Die Informationen, sprich: die Nummern, zu den Knoten dienen nicht nur der Bestimmung von Kantentypen. Wir können mit ihnen auch Beziehungen zwischen Knoten erkennen.

Beziehungen zwischen Knoten

Sei $G = (V, E)$ ein Graph und seien $u, v \in V$ gegeben. Dann gilt für das Ergebnis der DFS auf G:

Beobachtung	Schlussfolgerung
$\texttt{dfsNum}[u] < \texttt{dfsNum}[v]$	u wurde vor v entdeckt
$\texttt{FIN}[u] < \texttt{FIN}[v]$	u wurde vor v finalisiert
$\texttt{FIN}[u] < \texttt{dfsNum}[v]$	es gibt keinen Pfad von u nach v
$\texttt{dfsNum}[v] < \texttt{dfsNum}[u] \wedge \texttt{FIN}[u] < \texttt{FIN}[v]$	u ist Kind von v
$\texttt{lowNum}[u] = \texttt{dfsNum}[v] \wedge u$ ist Kind von v	u liegt auf einem Kreis

Iterativ oder rekursiv?

Wir kennen beide Varianten der Tiefensuche. Der rekursive Ansatz hat sowohl Vor- als auch Nachteile:

Iterativ oder rekursiv?

Wir kennen beide Varianten der Tiefensuche. Der rekursive Ansatz hat sowohl Vor- als auch Nachteile:

Vorteile

- Übersichtlicher
- Ggf. einfacher zu verstehen
- Ggf. einfacher zu implementieren

Nachteile

- Rekursionstack hat auf realen PCs endliche Größe
- Funktionsaufrufe brauchen Zeit (und Ressourcen)
- Globale Variablen nötig (problematisch insbesondere bei mehreren parallelen Aufrufen)

Es geht auch iterativ!

Iterative DFS

```
1: ITERDFS(G = (V, E) : Graph, s : Node)
2: |   parent : [Node; n] = ⟨⊥, ..., ⊥⟩
3: |   todo, done : Stack⟨Node⟩ = ⟨⟩
4: |   todo.push(s)
```

Iterative DFS

```
ITERDFS(G = (V, E) : Graph, s : Node)
    parent : [Node; n] = ⟨⊥, . . . , ⊥⟩
    todo, done : Stack⟨Node⟩ = ⟨⟩
    todo.push(s)
    while ¬todo.empty() do
        v : Node = todo.pop()
        if ¬COLORED(v) then
            . . .
```

Woran erinnert uns das bisher? An unsere Implementierung der Breitensuche!

Iterative DFS

```
1:  ITERDFS(G = (V, E) : Graph, s : Node)
2:      parent : [Node; n] = ⟨⊥, ..., ⊥⟩
3:      todo, done : Stack⟨Node⟩ = ⟨⟩
4:      todo.push(s)
5:      while ¬todo.empty() do
6:          v : Node = todo.pop()
7:          if ¬COLORED(v) then
8:              COLOR(v)
9:              while parent[v] ≠ done.top() do done.pop()
10:             done.push(v)
```

Iterative DFS

```
1:  ITERDFS(G = (V, E) : Graph, s : Node)
2:      parent : [Node; n] = ⟨⊥, ..., ⊥⟩
3:      todo, done : Stack⟨Node⟩ = ⟨⟩
4:      todo.push(s)
5:      while ¬todo.empty() do
6:          v : Node = todo.pop()
7:          if ¬COLORED(v) then
8:              COLOR(v)
9:              while parent[v] ≠ done.top() do done.pop()
10:             done.push(v)
11:             for u ∈ N(v) do
12:                 if ¬COLORED(u) then
13:                     todo.push(u)
14:                     parent[u] = v
15:     while ¬done.empty() do done.pop()
```

Wo sind wir?

Topologische Sortierung

Topologische Sortierung

(Total-)Ordnung auf Objekten gemäß einer vorgegebenen Abhängigkeits-Beziehung

Topologische Sortierung

Topologische Sortierung

(Total-)Ordnung auf Objekten gemäß einer vorgegebenen Abhängigkeits-Beziehung

Topologische Sortierung auf Graphen

Sei $G = (V, E)$ ein gerichteter Graph. Eine topologische Sortierung auf G ist eine Ordnung der Knoten V, sodass jede Kante von einem kleineren zu einem größeren Knoten zeigt.

Topologische Sortierung

Topologische Sortierung

(Total-)Ordnung auf Objekten gemäß einer vorgegebenen Abhängigkeits-Beziehung

Topologische Sortierung auf Graphen

Sei $G = (V, E)$ ein gerichteter Graph. Eine topologische Sortierung auf G ist eine Ordnung der Knoten V, sodass jede Kante von einem kleineren zu einem größeren Knoten zeigt.

Was müssen wir für die Existenz einer topologischen Sortierung voraussetzen?

Topologische Sortierung

Topologische Sortierung

(Total-)Ordnung auf Objekten gemäß einer vorgegebenen Abhängigkeits-Beziehung

Topologische Sortierung auf Graphen

Sei $G = (V, E)$ ein gerichteter Graph. Eine topologische Sortierung auf G ist eine Ordnung der Knoten V, sodass jede Kante von einem kleineren zu einem größeren Knoten zeigt.

Eine topologische Sortierung auf G existiert genau dann, wenn G azyklisch ist (also ein DAG).

TopoSort

```
1: TOPOSORT(G = (V, E): DAG) : [Node; n]
2:     topo: [ℕ; n] = ⟨0, ..., 0⟩
3:     for v ∈ V do
4:         if not marked v then
5:             DFS(G, v)
6:     return topo
```

TopoSort

```
1:  TOPOSORT(G = (V, E): DAG) : [Node; n]
2:      topo: [ℕ; n] = ⟨0, ..., 0⟩
3:      for v ∈ V do
4:          if not marked v then
5:              DFS(G, v)
6:      return topo
7:  DFS(G = (V, E): Graph, v : Node)
8:      mark v
9:      for u ∈ N(v) do
10:         if not marked u then
11:             DFS(G, u)
12:     finNum[v] := finCounter
13:     topo[finCounter] := v
14:     finCounter := finCounter + 1
```

Topologische Sortierung auf DAGs

Wir können uns zur Berechnung einer topologischen Sortierung die DFS zu Nutze machen!

- Idee: FIN-Nummern nutzen
- Vorgehen: Führe DFS aus, bis alle Knoten entdeckt, sortiere Knoten entsprechend ihrer FIN-Nummern

Topologische Sortierung auf DAGs

Knoten:

`dfsNum`

`FIN`

`low`

Kanten:

unbekannt

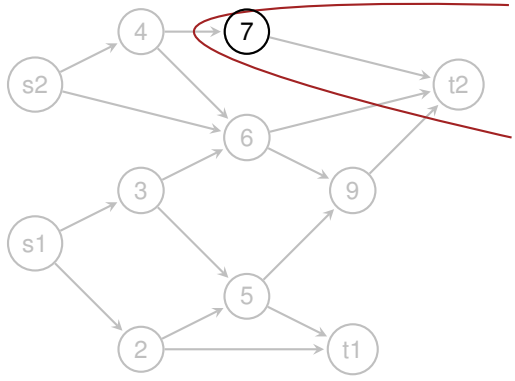
Baumkante

Rückkante

Vorkante

Querkante

Topologische Sortierung:



Topologische Sortierung auf DAGs

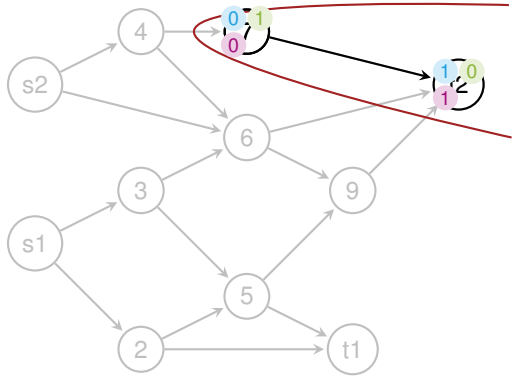
Knoten:

`dfsNum`
`FIN`
`low`

Kanten:

unbekannt
Baumkante
Rückkante
Vorkante
Querkante

Topologische Sortierung:
t2, 7



Topologische Sortierung auf DAGs

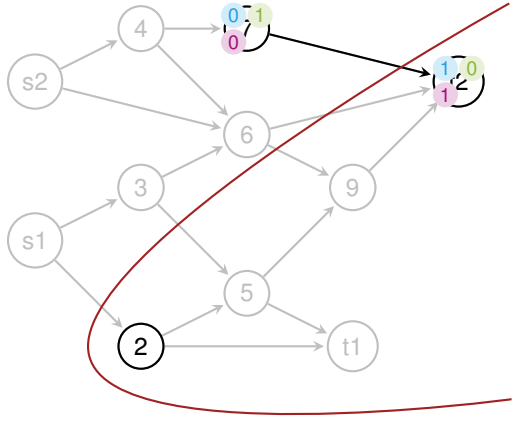
Knoten:

`dfsNum`
`FIN`
`low`

Kanten:

unbekannt
Baumkante
Rückkante
Vorkante
Querkante

Topologische Sortierung:
t2, 7



Topologische Sortierung auf DAGs

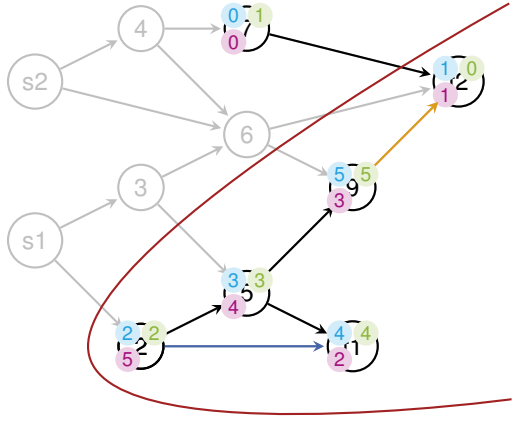
Knoten:

`dfsNum`
`FIN`
`low`

Kanten:

unbekannt
Baumkante
Rückkante
Vorkante
Querkante

Topologische Sortierung:
t2, 7, t1, 9, 5, 2



Topologische Sortierung auf DAGs

Knoten:

`dfsNum`

`FIN`

`low`

Kanten:

unbekannt

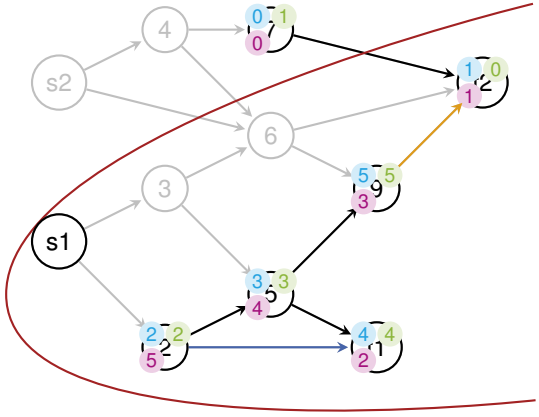
Baumkante

Rückkante

Vorkante

Querkante

Topologische Sortierung:
t2, 7, t1, 9, 5, 2



Topologische Sortierung auf DAGs

Knoten:

`dfsNum`

`FIN`

`low`

Kanten:

unbekannt

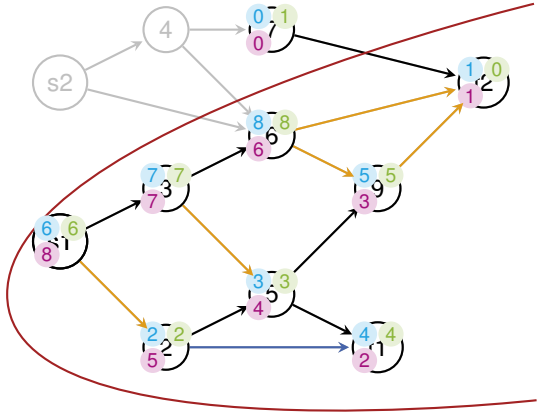
Baumkante

Rückkante

Vorkante

Querkante

Topologische Sortierung:
t2, 7, t1, 9, 5, 2, 6, 3, s1



Topologische Sortierung auf DAGs

Knoten:

dfsNum

FIN

low

Kanten:

unbekannt

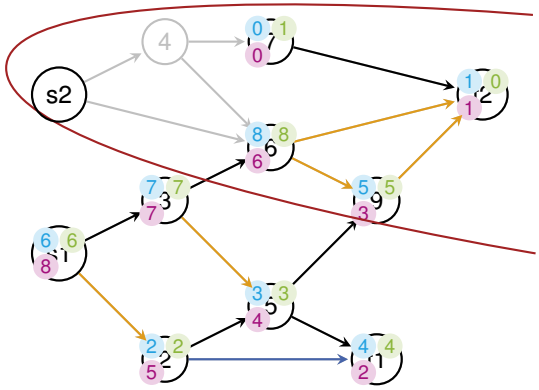
Baumkante

Rückkante

Vorkante

Querkante

Topologische Sortierung:
t2, 7, t1, 9, 5, 2, 6, 3, s1



Topologische Sortierung auf DAGs

Knoten:

`dfsNum`

`FIN`

`low`

Kanten:

unbekannt

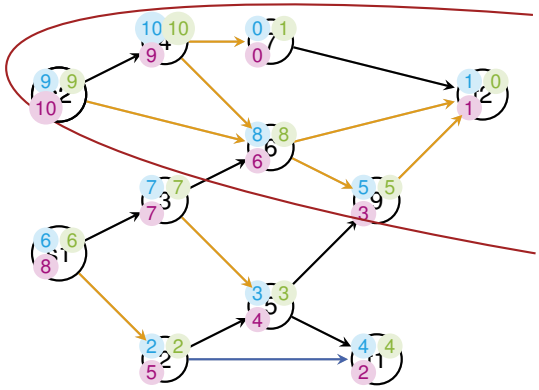
Baumkante

Rückkante

Vorkante

Querkante

Topologische Sortierung:
t2, 7, t1, 9, 5, 2, 6, 3, s1, 4, s2



Wo sind wir?

Dynamische Programmierung

Dynamische Programmierung

Ansatz zur Lösung von Problemen, deren **optimale Lösung** sich **aus optimalen Lösungen von Teilproblemen** zusammensetzt.

- Idee: Berechne Lösungen der Teilprobleme, berechne darauf aufbauend die Lösung des Gesamtproblems
- Basis: **Rekurrenz**, anhand derer aus Teillösungen eine neue Lösung berechnet wird (muss gezeigt werden!)
- Wichtig: Optimale Lösungen von Teilproblemen ergeben neue optimale Lösung
- Sinnvolle Formulierung von Teilproblemen kann knifflig sein und erfordert etwas Kreativität

DP - MST

Bekanntes Beispiel

Wir kennen bereits einen Algorithmus, der dynamische Programmierung verwendet: Der Kruskal-Algorithmus.

- Teilprobleme: Teilgraphen, Teillösugen: Teil-MSTs
- Basis-Instanz: Teilgraph mit nur einem Knoten, sind ihr eigener MST
- Aus der VL wissen wir: MST setzt sich aus MSTs zusammen
- Also: Kruskal berechnet optimale Lösung des Gesamtproblems aus optimalen Lösungen von Teilproblemen

Einschub: Greedy-Algorithmus

Def.: Greedy-Algorithmus

Ein **Greedy-Algorithmus** ist ein Algorithmus, der **lokal optimale Schritte** macht. Es wird schrittweise der Folgezustand ausgewählt, der zum Zeitpunkt der Wahl das beste Ergebnis verspricht (Bewertung der möglichen Teillösugen durch eine Bewertungsfunktion).

Greedy-Algorithmen sind i.d.R. schnell, lösen Probleme jedoch nicht optimal.

Einschub: Greedy-Algorithmus

Def.: Greedy-Algorithmus

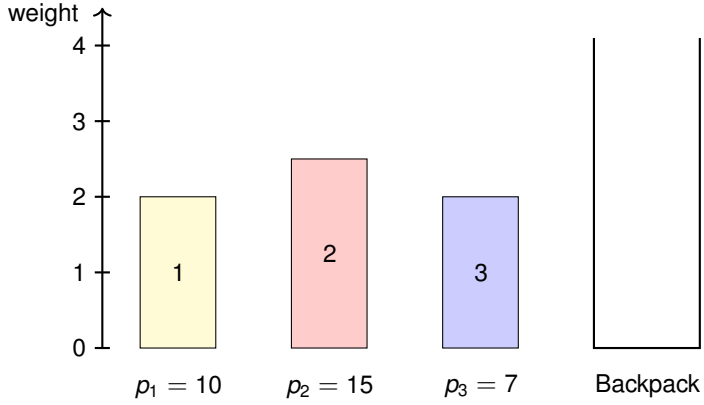
Ein **Greedy-Algorithmus** ist ein Algorithmus, der **lokal optimale Schritte** macht. Es wird schrittweise der Folgezustand ausgewählt, der zum Zeitpunkt der Wahl das beste Ergebnis verspricht (Bewertung der möglichen Teillösugen durch eine Bewertungsfunktion).

Greedy-Algorithmen sind i.d.R. schnell, lösen Probleme jedoch nicht optimal.

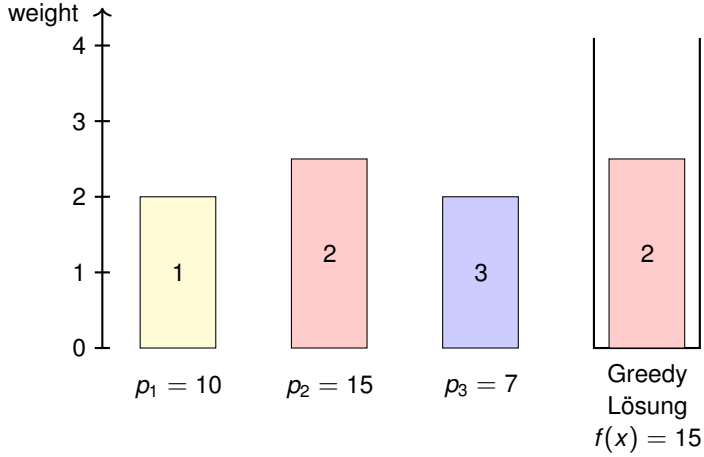
Deshalb: Beweis bei DP notwendig

Um zu zeigen, dass unser DP-Algorithmus eine optimale Lösung liefert, müssen wir beweisen, dass das Zusammensetzten optimaler Lösungen von uns formulierter Teilprobleme immer optimale Lösungen liefert.

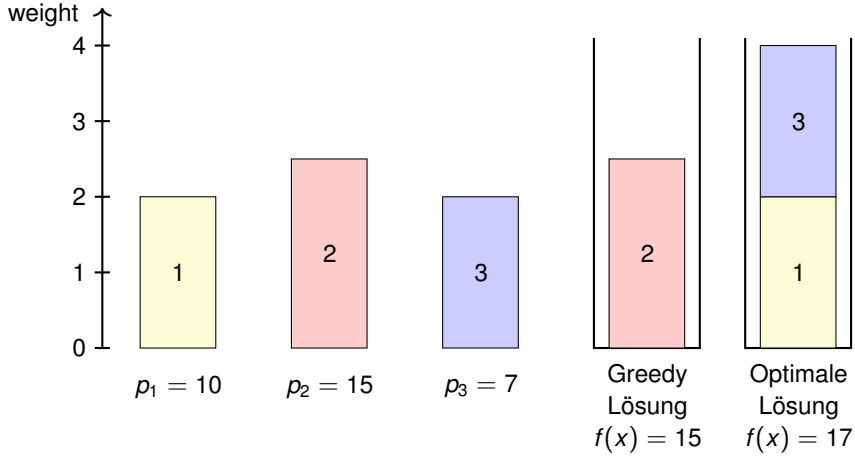
Rucksackproblem



Rucksackproblem



Rucksackproblem



DP - Rucksackproblem

Gegeben:

- feste Rucksack-Kapazität M
- n Elemente, jeweils mit einem Gewicht w_i und einem Profit p_i

Gesucht: Auawahl an Elementen so, dass...

- Profit maximiert wird
- die Summer der Gewichte der ausgewählten Elemente M nicht übersteigt

Wie sehen sinnvolle Teilprobleme einer Rucksackprobleminstanz aus?

DP - Rucksackproblem

Gegeben:

- feste Rucksack-Kapazität M
- n Elemente, jeweils mit einem Gewicht w_i und einem Profit p_i

Gesucht: Auawahl an Elementen so, dass...

- Profit maximiert wird
- die Summer der Gewichte der ausgewählten Elemente M nicht übersteigt

Teilprobleme: Einschränkung auf eine Teilkapazität $C \leq M$ und eine Teilmenge der ersten $i \leq$ Elemente.

Lemma zu Teilproblemen des Rucksackproblems

Für den optimalen Profit $P(i, C)$ eines Rucksacks der Kapazität C mit den Elementen $\{1..i\}$ gilt:

$$P(i, C) = \max(\underbrace{P(i-1, C)}_{\text{wenn } i \text{ nicht gewählt wird}}, \underbrace{P(i-1, C-w_i) + p_i}_{\text{wenn } i \text{ gewählt wird}})$$

DP - Rucksackproblem

- Wir setzen also als Rekursionsanfang/kleinstes Teilproblem $P(0, C) = 0$.
- Basisinstanzen:

DP - Rucksackproblem

- Wir setzen also als Rekursionsanfang/kleinstes Teilproblem $P(0, C) = 0$.
- Basisinstanzen:
  - $(0, C)$ mit optimaler Lösung $P(0, C) = 0$
  - $(i, 0)$ für $i \in \{1, \ldots, n\}$ mit optimaler Lösung $P(i, 0) = 0$
- Davon ausgehend: schrittweise Kapazität erhöhen / Elementmenge erweitern und Lemma von eben nutzen
- Teillösungen werden festgehalten in Tabelle mit $n \times M$ Einträgen

DP -Rucksackproblem Algo

```
1:  DYNAMICKNAPSACK(p: [ℕ; n], w: [ℕ; n], M: ℕ) : ℕ
2:      P : [[ℕ_0;M + 1];n + 1] = ⟨⟨0, ..., 0⟩, ..., ⟨0, ..., 0⟩⟩
3:      decision : [[Bool;M + 1];n + 1]= ⟨⟨f, ..., f⟩, ..., ⟨f, ..., f⟩⟩
4:      for i ∈ {1, ..., n − 1} do
5:          for C ∈ {0, ..., M} do
6:              if C ≥ w_i and P[i − 1][C − w_i] + p_i > P[i − 1][C] then
7:                  P[i][C] := P[i − 1][C − w_i] + p_i
8:                  decision[i, C] := w                          // gültige Lösung
9:              else
10:                 P[i][C] := P[i − 1][C]
11:                 decision[i, C] := f
12: return P[n][M]
```

Laufzeit:

DP -Rucksackproblem Algo

```
1:  DYNAMICKNAPSACK(p: [ℕ; n], w: [ℕ; n], M: ℕ) : ℕ
2:      P : [[ℕ_0;M + 1];n + 1] = ⟨⟨0, ..., 0⟩, ..., ⟨0, ..., 0⟩⟩
3:      decision : [[Bool;M + 1];n + 1]= ⟨⟨f, ..., f⟩, ..., ⟨f, ..., f⟩⟩
4:      for i ∈ {1, ..., n − 1} do
5:          for C ∈ {0, ..., M} do
6:              if C ≥ w_i and P[i − 1][C − w_i] + p_i > P[i − 1][C] then
7:                  P[i][C] := P[i − 1][C − w_i] + p_i
8:                  decision[i, C] := w                          // gültige Lösung
9:              else
10:                 P[i][C] := P[i − 1][C]
11:                 decision[i, C] := f
12: return P[n][M]
```

Laufzeit: $\mathcal{O}(nM) \Rightarrow$ *pseudopolynomiell*

DP - Rucksackproblem Beispiel

Es sei ein Rucksack der Kapazität $M = 7$, der Gewichtsvektor $w = (1, 3, 5, 4)$ und der Profitvektor $p = (10, 15, 30, 20)$ gegeben.

Wir haben eine Lösungstabelle inder der Algorithmus $P = (i, C)$ und in Klammern $decision[i][C]$ in jeder Zelle einträgt.

i \ C	0	1	2	3	4	5	6	7
0	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)
1	0 (**f**)							
2	0 (**f**)							
3	0 (**f**)							
4	0 (**f**)							

DP - Rucksackproblem Lösung

Lösung

i \ C	0	1	2	3	4	5	6	7
0	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)	0 (**f**)
1	0 (**f**)	10 (**w**)	10 (**w**)	10 (**w**)	10 (**w**)	10 (**w**)	10 (**w**)	10 (**w**)
2	0 (**f**)	10 (**f**)	10 (**f**)	15 (**w**)	25 (**w**)	25 (**w**)	25 (**w**)	25 (**w**)
3	0 (**f**)	10 (**f**)	10 (**f**)	15 (**f**)	25 (**f**)	30 (**w**)	40 (**w**)	40 (**w**)
4	0 (**f**)	10 (**f**)	10 (**f**)	15 (**f**)	25 (**f**)	30 (**f**)	40 (**f**)	40 (**f**)

```
1: GETSOLUTION: {0,1}^n                                   // Rekonstruiere Lösung
2:     C := M
3:     x : [Bool;n+1] = ⟨f, ..., f⟩                       // Index 0 in x wird ignoriert
4:     for i ∈ {n, ..., 1} do
5:         x[i] := decision[i, C]
6:         if decision[i, C][i] == w then C := C − w_i
7:     return x
```

Lösen von Problemen mit DP

Schritt 1 - Kreativität:

Wie kann das Problem in Teilprobleme zerlegt werden, sodass alle Lösungen der Teilprobleme eine Gesamtlösung ergeben?

Lösen von Problemen mit DP

Schritt 1 - Kreativität:

Wie kann das Problem in Teilprobleme zerlegt werden, sodass alle Lösungen der Teilprobleme eine Gesamtlösung ergeben?

Schritt 2 - Rekurrenz:

Wie können wir rekursiv aus den Teilprobleme eine Gesamtlösung berechnen?

Ist die neue Lösung korrekt und (falls notwendig) optimal? (Beweisen!)

Lösen von Problemen mit DP

Schritt 1 - Kreativität:

Wie kann das Problem in Teilprobleme zerlegt werden, sodass alle Lösungen der Teilprobleme eine Gesamtlösung ergeben?

Schritt 2 - Rekurrenz:

Wie können wir rekursiv aus den Teilproblemen eine Gesamtlösung berechnen?

Ist die neue Lösung korrekt und (falls notwendig) optimal? (Beweisen!)

Schritt 3 - Algorithmuskonstruktion:

Wie kann die rekursive Formel iterativ berechnet werden?

In welcher Reihenfolge werden die Lösungen berechnet um keine Abhängigkeitsprobleme zu bekommen?

Schritt 1 - Kreativität

Zentrale Frage: Wie kann das Problem zerlegt werden?

- $n \rightsquigarrow n - 1$
- $n \rightsquigarrow n/2$
- Baum $\rightsquigarrow$ Teilbäume
- String $\rightsquigarrow$ Substring(s)
- Array $\rightsquigarrow$ TeilArray(s)
- Mengen von Elemente $\rightsquigarrow$ Teilemenge(n) der Elemente
- ...

Schritt 1 - Kreativität

Zentrale Frage: Wie kann das Problem zerlegt werden?

- $n \rightsquigarrow n - 1$
- $n \rightsquigarrow n/2$
- Baum $\rightsquigarrow$ Teilbäume
- String $\rightsquigarrow$ Substring(s)
- Array $\rightsquigarrow$ TeilArray(s)
- Mengen von Elemente $\rightsquigarrow$ Teilemenge(n) der Elemente
- ...

Wichtig dabei; Teillösungen müssen zu Gesamtlösung zusammengebaut werden können.

Schritt 2 - Rekurrenz

Nachdem wir eine Idee haben wie wir das Problem zerlegen können, geht es darum eine rekursive Formel zu finden, in der alle Fälle inbegriffen sind, die auftreten können um eine optimale Lösung aus Teillösungen zu erzeugen.

Für das Rucksackproblem gilt:

$$P(i, C) = \max(\underbrace{P(i-1, C)}_{\substack{\text{wenn } i \text{ nicht} \\ \text{gewählt wird}}}, \underbrace{P(i-1, C-w_i) + p_i}_{\text{wenn } i \text{ gewählt wird}})$$

In diesem Fall haben wir 2 Möglichkeiten: wir nehmen das aktuelle Element i oder wir nehmen das aktuelle Element i nicht.

Schritt 2 - Rekurenz 2

In diesem Beispiel ist relativ schnell ersichtlich, warum wir aus den Teilproblemen eine optimale Lösung erzeugen:
Wir betrachten die Elemente nacheinander für jede Kapazität. Dadurch entscheiden wir jeweils lokal, ob es für eine bestimmte Kapazität und eine ausgewählten Teilmenge an Elementen sinnvoll ist, das aktuell betrachtete Element mitzunehmen.
Konkret:
Für Menge $M = \{e_1, \ldots, e_n\}$ haben wir im Schritt i, e_i aus Menge $M_i = \{e_1, \ldots, e_i\}$ betrachtet und das für alle validen Kapazitäten. Somit haben wir alle möglichen Kombinationen mindestens einmal durchprobiert.

Schritt 3 - Algorithmuskonstruktion

Im Schritt 3 ist es das Ziel einen iterativen Algorithmus zu schreiben der die Rekursionsformel umsetzt.

Bei vielen DP Problemen muss oft das gleiche Ergebnis mehrfach berechnet werden. Durch iterativen Algorithmenentwurf können Zwischenergebnisse gespeichert werden, dadurch müssen Zwischenergebnisse nur ein mal berechnet werden. Die Laufzeit ist polynomiell in der Eingabegröße.

Simples Beispiel dafür, dass nur das Implementieren der Rekurrenzformel für exponentielle Laufzeit sorgt, sind die Fibonacci Zahlen (siehe VL).

Levenshtein Distanz

Levenshtein Distanz

Gegeben zwei Strings S_1, S_2 betrachte folgende Operationen auf S_1

- insert(i, c) Füge in S_1 an Stelle i Zeichen c ein, die nachfolgenden Zeichen rutschen nach hinten.
- delete(i) Entferne Zeichen i aus S_1, die nachfolgenden Zeichen rutschen nach.
- replace(i, c) Ersetze in S_1 Zeichen i durch Zeichen c, nachfolgende Zeichen bleiben unverändert.

Die Levenshtein Distanz gibt die minimale Anzahl an Operationen an, um S_1 in S_2 zu transformieren.

Beispiel Levenshtein Distanz

S_1 `levenshtein`

S_2 `meilenstein`

Transformation

- `levenshtein`

Beispiel Levenshtein Distanz

S_1 `levenshtein`

S_2 `meilenstein`

Transformation

- `levenshtein`
- `mevenshtein` replace(0, m)

Beispiel Levenshtein Distanz

S_1 `levenshtein`

S_2 `meilenstein`

Transformation

- `levenshtein`
- `mevenshtein` replace(0, m)
- `meivenshtein` insert(2, i)

Beispiel Levenshtein Distanz

S_1 `levenshtein`

S_2 `meilenstein`

Transformation

- `levenshtein`
- `mevenshtein` replace(0, m)
- `meivenshtein` insert(2, i)
- `meilenshtein` replace(3, l)

Beispiel Levenshtein Distanz

S_1 `levenshtein`

S_2 `meilenstein`

Transformation

- `levenshtein`
- `mevenshtein` replace(0, m)
- `meivenshtein` insert(2, i)
- `meilenshtein` replace(3, l)
- `meilenstein` delete(7)

Beispiel Levenshtein Distanz

S_1 `levenshtein`

S_2 `meilenstein`

Transformation

- `levenshtein`
- `mevenshtein` replace(0, m)
- `meivenshtein` insert(2, i)
- `meilenshtein` replace(3, l)
- `meilenstein` delete(7)

$\rightarrow$ Levenshtein Distanz: 4

Schritt 1 - Levenshtein Distanz - DP

Idee: auf kürzeren Wörtern ist die Levenshtein Distanz leichter zu berechnen(weniger Kombinationen), d.h. wir unterteilen unsere Wörter. Aber wie?

Schritt 1 - Levenshtein Distanz - DP

Idee: auf kürzeren Wörtern ist die Levenshtein Distanz leichter zu berechnen(weniger Kombinationen), d.h. wir unterteilen unsere Wörter. Aber wie?

halbieren Problem: Wo unterteilen wir? Wir müssten S_1 und S_2 so teilen, dass die Schnitte in den Zeichenketten nach den Operationen gleich sind. Dementsprechend müssten wir wissen wie viele Einfüge- und Löschoperationen wir auf einem der beiden Teilstrings machen.

Schritt 1 - Levenshtein Distanz - DP

Idee: auf kürzeren Wörtern ist die Levenshtein Distanz leichter zu berechnen(weniger Kombinationen), d.h. wir unterteilen unsere Wörter. Aber wie?

halbieren Problem: Wo unterteilen wir? Wir müssten S_1 und S_2 so teilen, dass die Schnitte in den Zeichenketten nach den Operationen gleich sind. Dementsprechend müssten wir wissen wie viele Einfüge- und Löschoperationen wir auf einem der beiden Teilstrings machen.

verkürzen Wenn wir die Strings entsprechend der Operation um 1 verkürzen, fangen wir bei einem Zeichen an und erweitern so unsere Lösung bis wir beide Wörter vollständig gleichgemacht haben.

Schritt 2 - Levenshtein Distanz - DP

Notation: $S_k = c_k^0 c_k^1 \dots c_k^{n_k - 1}$ es sei $S_k^l = c_k^0 c_k^1 \dots c_k^l$ der Teilstring von S_k mit den ersten l Zeichen.

$$lev(S_1^{m_1}, S_1^{m_2}) = \begin{cases} m_1 & m_2 = 0 \\ m_2 & m_1 = 0 \\ & \\ & \\ & \end{cases}$$

Schritt 2 - Levenshtein Distanz - DP

Notation: $S_k = c_k^0 c_k^1 \dots c_k^{n_k-1}$ es sei $S_k^l = c_k^0 c_k^1 \dots c_k^l$ der Teilstring von S_k mit den ersten l Zeichen.

$$lev(S_1^{m_1}, S_1^{m_2}) = \begin{cases} m_1 & m_2 = 0 \\ m_2 & m_1 = 0 \\ lev(S_1^{m_1-1}, S_2^{m_2-1}) & c_1^{m_1} = c_2^{m_2} \\ \\ \\ \end{cases}$$

Schritt 2 - Levenshtein Distanz - DP

Notation: $S_k = c_k^0 c_k^1 \dots c_k^{n_k-1}$ es sei $S_k^l = c_k^0 c_k^1 \dots c_k^l$ der Teilstring von S_k mit den ersten l Zeichen.

$$lev(S_1^{m_1}, S_1^{m_2}) = \begin{cases} m_1 & m_2 = 0 \\ m_2 & m_1 = 0 \\ lev(S_1^{m_1-1}, S_2^{m_2-1}) & c_1^{m_1} = c_2^{m_2} \\ 1 + \min \begin{cases} lev(S_1^{m_1}, S_2^{m_2-1}) & \text{insert} \\ \\ \\ \end{cases} \end{cases}$$

Schritt 2 - Levenshtein Distanz - DP

Notation: $S_k = c_k^0 c_k^1 \dots c_k^{n_k-1}$ es sei $S_k^l = c_k^0 c_k^1 \dots c_k^l$ der Teilstring von S_k mit den ersten l Zeichen.

$$lev(S_1^{m_1}, S_1^{m_2}) = \begin{cases} m_1 & m_2 = 0 \\ m_2 & m_1 = 0 \\ lev(S_1^{m_1-1}, S_2^{m_2-1}) & c_1^{m_1} = c_2^{m_2} \\ 1 + \min \begin{cases} lev(S_1^{m_1}, S_2^{m_2-1}) & \text{insert} \\ lev(S_1^{m_1-1}, S_2^{m_2}) & \text{delete} \end{cases} & \end{cases}$$

Schritt 2 - Levenshtein Distanz - DP

Notation: $S_k = c_k^0 c_k^1 \dots c_k^{n_k - 1}$ es sei $S_k^l = c_k^0 c_k^1 \dots c_k^l$ der Teilstring von S_k mit den ersten l Zeichen.

$$lev(S_1^{m_1}, S_1^{m_2}) = \begin{cases} m_1 & & & m_2 = 0 \\ m_2 & & & m_1 = 0 \\ lev(S_1^{m_1-1}, S_2^{m_2-1}) & & & c_1^{m_1} = c_2^{m_2} \\ 1 + \min \begin{cases} lev(S_1^{m_1}, S_2^{m_2-1}) & \text{insert} \\ lev(S_1^{m_1-1}, S_2^{m_2}) & \text{delete} \\ lev(S_1^{m_1-1}, S_2^{m_2-1}) & \text{replace} \end{cases} & & \text{otherwise.} \end{cases}$$

Schritt 2 - Levenshtein Distanz - DP

Warum ist die Rekurrenz korrekt?

Fall 1: m_1, when $m_2 = 0$
Wir brauchen m_1 delete-Operationen um den Unterschied auszugleichen.

Fall 2: m_1, when $m_2 = 0$
Wir brauchen m_1 insert-Operationen um den Unterschied auszugleichen.

Fall 3: $lev(S_1^{m_1-1}, S_2^{m_2-1})$, when $c_1^{m_1} = c_2^{m_2}$
Wenn die letzten beiden Buchstaben($c_1^{m_1}$, $c_2^{m_2}$) gleich sind, dann ändert sich die Distanz nicht.

Schritt 2 - Levenshtein Distanz - DP

Warum ist die Rekurrenz korrekt?

Fall 4:
Tritt keiner der vorherigen Fälle ein müssen wir der String verändern und somit wird die Distanz um 1 größer. Die Frage ist aber wie sich die zwei Teilsrings $S_1^{m_1}$ und $S_2^{m_2}$ verändern.

insert Wenn eine Insert-Operation gemacht wird, ist die Distanz die Gleiche wie für ein Zeichen von S_2 weniger. Dies liegt daran, dass das letzte Zeichen($c_2^{m_2}$) von S_2^{m-1} an $S_1^{m_1}$ gehängt wird.

delete Wenn eine Delete-Operation gemacht wird ist das äquivalent zu insert, wir machen aber $S_1^{m_1}$ kürzer.

replace Da der Wert $c_1^{m_1}$ in S_1 durch $c_2^{m_2}$ ersetzt wird, schauen wir uns die Levenshtein Distanz der kürzeren Strings an, da der aktuelle Wert abgeglichen wurde.

Schritt 3 - Levenshtein Distanz - DP

In der Rekursion von $lev(S_1^{m_1}, S_1^{m_2})$ haben wir folgende Zwischenergebnisse:

- $lev(S_1^{m_1}, S_2^{m_2-1})$
- $lev(S_1^{m_1-1}, S_2^{m_2})$
- $lev(S_1^{m_1-1}, S_2^{m_2-1})$

Diese müssen also vor $lev(S_1^{m_1}, S_2^{m_2})$ berechnet werden.

Zwei Teilstrings von (S_1, S_2) $\rightsquigarrow$ 2-dimensionalem Array.

Schritt 3 - Levenshtein Distanz - DP

Beispiel:

S_1 `verstecken`

S_2 `erschrecken`

S_1 \ S_2	-	e	r	s	c	h	r	e	c	k	e	n
-	0	1	2	3	4	5	6	7	8	9	10	11
v	1											
e	2											
r	3											
s	4											
t	5											
e	6											
c	7											
k	8											
e	9											
n	10											

Schritt 3 - Levenshtein Distanz - DP

LEV($S_1 : [char; m_1], S_2 : [char; m_1]) : \mathbb{N}$
  $lev : [[char; m_2]m_1] = \langle\langle 0, \ldots, 0\rangle, \ldots, \langle 0, \ldots, 0\rangle\rangle$
  for $k = 0$ to m_1 **do**
    $lev[k][0] = k$
  for $l = 0$ to m_2 **do**
    $lev[0][l] = l$

Schritt 3 - Levenshtein Distanz - DP

$\text{LEV}(S_1 : [char; m_1], S_2 : [char; m_1]) : \mathbb{N}$
- $lev : [[char; m_2]m_1] = \langle\langle 0, \ldots, 0\rangle, \ldots, \langle 0, \ldots, 0\rangle\rangle$
- **for** $k = 0$ to m_1 **do**
  - $lev[k][0] = k$
- **for** $l = 0$ to m_2 **do**
  - $lev[0][l] = l$
- **for** $k = 1$ to m_1 **do**
  - **for** $l = 1$ to m_2 **do**
    - **if** $S_1^{k-1} = S_2^{l-1}$ **then**
      - $lev[k][l] = lev[k-1][l-1]$

Schritt 3 - Levenshtein Distanz - DP

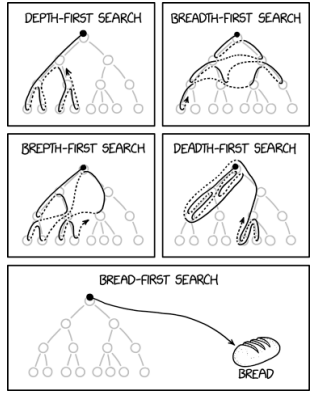
```
LEV(S_1 : [char; m_1], S_2 : [char; m_1]) : ℕ
    lev : [[char; m_2]m_1] = ⟨⟨0, ..., 0⟩, ..., ⟨0, ..., 0⟩⟩
    for k = 0 to m_1 do
        lev[k][0] = k
    for l = 0 to m_2 do
        lev[0][l] = l
    for k = 1 to m_1 do
        for l = 1 to m_2 do
            if S_1^{k-1} = S_2^{l-1} then
                lev[k][l] = lev[k − 1][l − 1]
            else
                lev[k][l] = 1+
                min(lev[k − 1][l − 1], lev[1][l − 1], lev[k − 1][l])
    return lev[m_1][m_2]
```

Schritt 3 - Levenshtein Distanz - DP

S_1 \ S_2	-	e	r	s	c	h	r	e	c	k	e	n
-	0	1	2	3	4	5	6	7	8	9	10	11
v	1	0	1	2	3	4	5	6	7	8	9	10
e	2	1	1	2	3	4	5	5	6	7	8	9
r	3	2	1	2	3	4	4	5	6	7	8	9
s	4	3	2	1	2	3	4	5	6	7	8	9
t	5	4	3	2	2	3	4	5	6	7	8	9
e	6	5	4	3	3	3	4	4	5	6	7	8
c	7	6	5	4	3	4	4	5	4	5	6	7
k	8	7	6	5	4	4	5	5	5	4	5	6
e	9	8	7	6	5	5	5	5	6	5	4	5
n	10	9	8	7	6	6	6	6	6	6	5	4

Das war es für heute!

Gibt es Fragen?



Quelle: https://xkcd.com/2407/