# Dynamische Programmierung

## Studiengangbezogene Lehrveranstaltung im Rahmen der Vorlesung Datenstrukturen und Algorithmen im Bachelor-Studiengang Informatik

Dr. Ralf Schlüter

Lehrstuhl Informatik 6
Human Language Technology and Pattern Recognition
Computer Science Department, RWTH Aachen University
D-52056 Aachen, Germany

7. Mai 2019

# Dynamische Programmierung: Einordnung

- Konzept zur Algorithmenentwicklung
- Effiziente Lösung bestimmter Klassen von Optimierungsproblemen
- Effiziente Strukturierung $\rightarrow$ geeignete Datenstrukturen

# Dynamische Programmierung: Konzept

Generelles Konzept der dynamischen Programmierung:

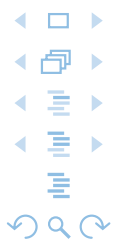
- teile Problem in Teilprobleme auf,
- löse Teilprobleme und speichere die Ergebnisse,
- kombiniere Ergebnisse der Teilprobleme zur Lösung des Problems

Richard Bellman 1957:

- dynamisch: sequentiell
- Programmierung: Optimierungsproblem mit Einschränkungen

# Dynamische Programmierung: Grundlagen

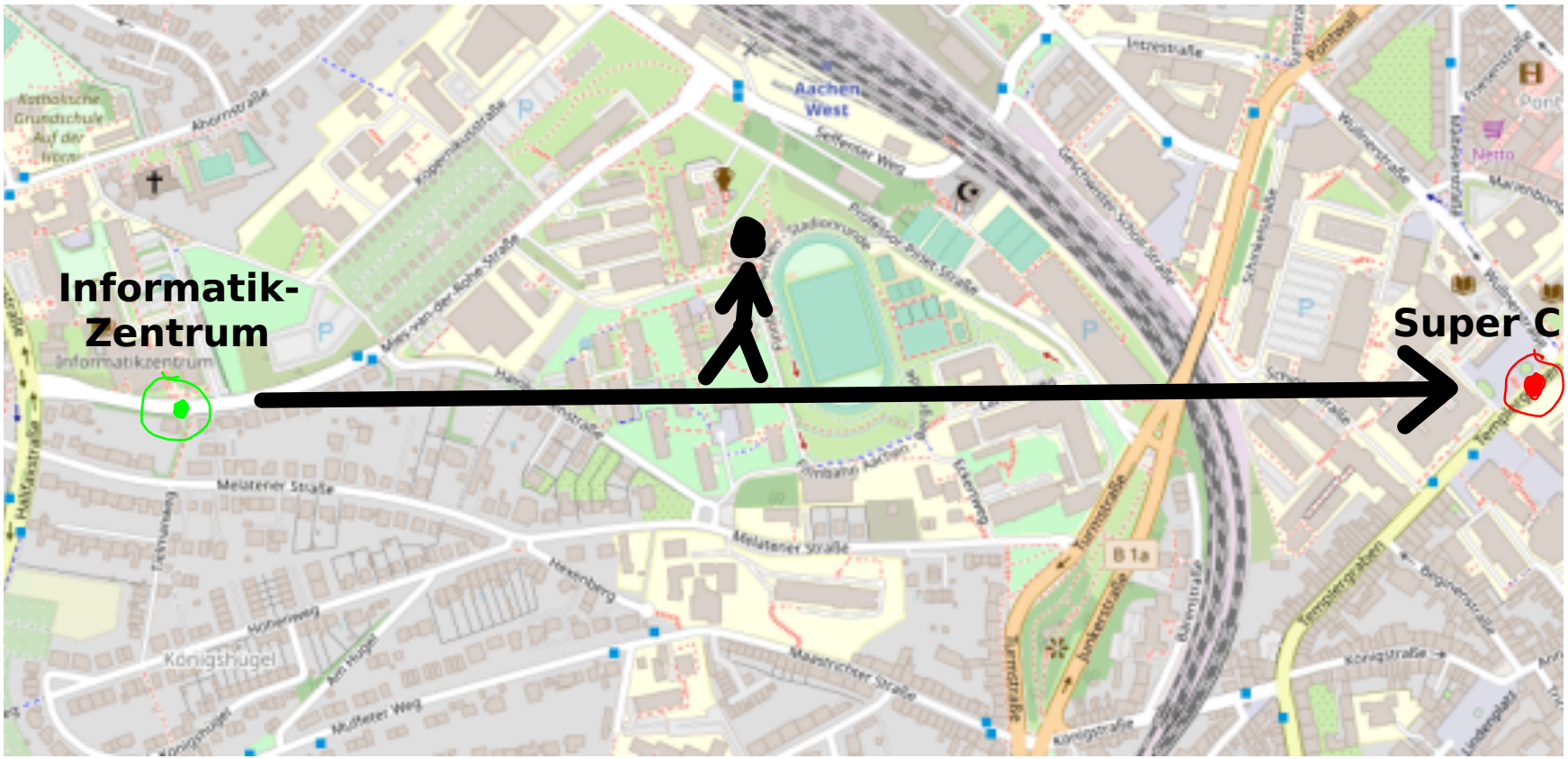
- Voraussetzung zur Anwendung der dynamischen Programmierung:
  - Aufteilbarkeit eines Problems der Größe $N$ in Teilprobleme kleinerer Größen $1, \ldots, N-1$
    $\rightarrow$ *Bellmansches Optimalitätsprinzip*
- Lösung eines Problems der Größe $N$:
  - setzt Lösungen der Teilprobleme der Größen $1, \ldots, N-1$ voraus,
  - Lösungen liegen in einer Tabelle vor
  - Teillösungen können unterschiedlich zusammengesetzt werden:
  - (mindestens) eine dieser Möglichkeiten stellt die optimale Lösung dar
  - nach dieser optimalen Lösung wird systematisch gesucht
- Effiziente Vermeidung von Redundanz:
  - Dynamische Programmierung ist dann vorteilhaft, wenn Teillösungen "überlappen", d.h. Teillösungen für die Lösung mehrerer größerer Teilprobleme benötigt werden
  - Speicherung in Tabelle vermeidet redundante Neuberechnungen



# Dynamische Programmierung: Algorithmusentwicklung

- Entwicklung eines Algorithmus auf Basis dynamischer Programmierung:
  1. Charakterisierung des Lösungsraums und der Struktur der angestrebten optimalen Lösung
  2. Rekursive Zerlegung der optimalen Lösung in Teillösungen (und deren zugeordnete Werte)
  3. Konzeption des Algorithmus:
     - Vorgehensweise *Bottom-up*, so, dass
     - Teillösungen nach aufsteigender Größe $n = 1, \ldots, N$ erstellt/gespeichert werden,
     - und bei der Bestimmung einer höheren Teillösung $n$ die jeweils notwendigen Teillösungen $1, \ldots, n-1$ bereits vorliegen

# Beispiel: kürzester Pfad



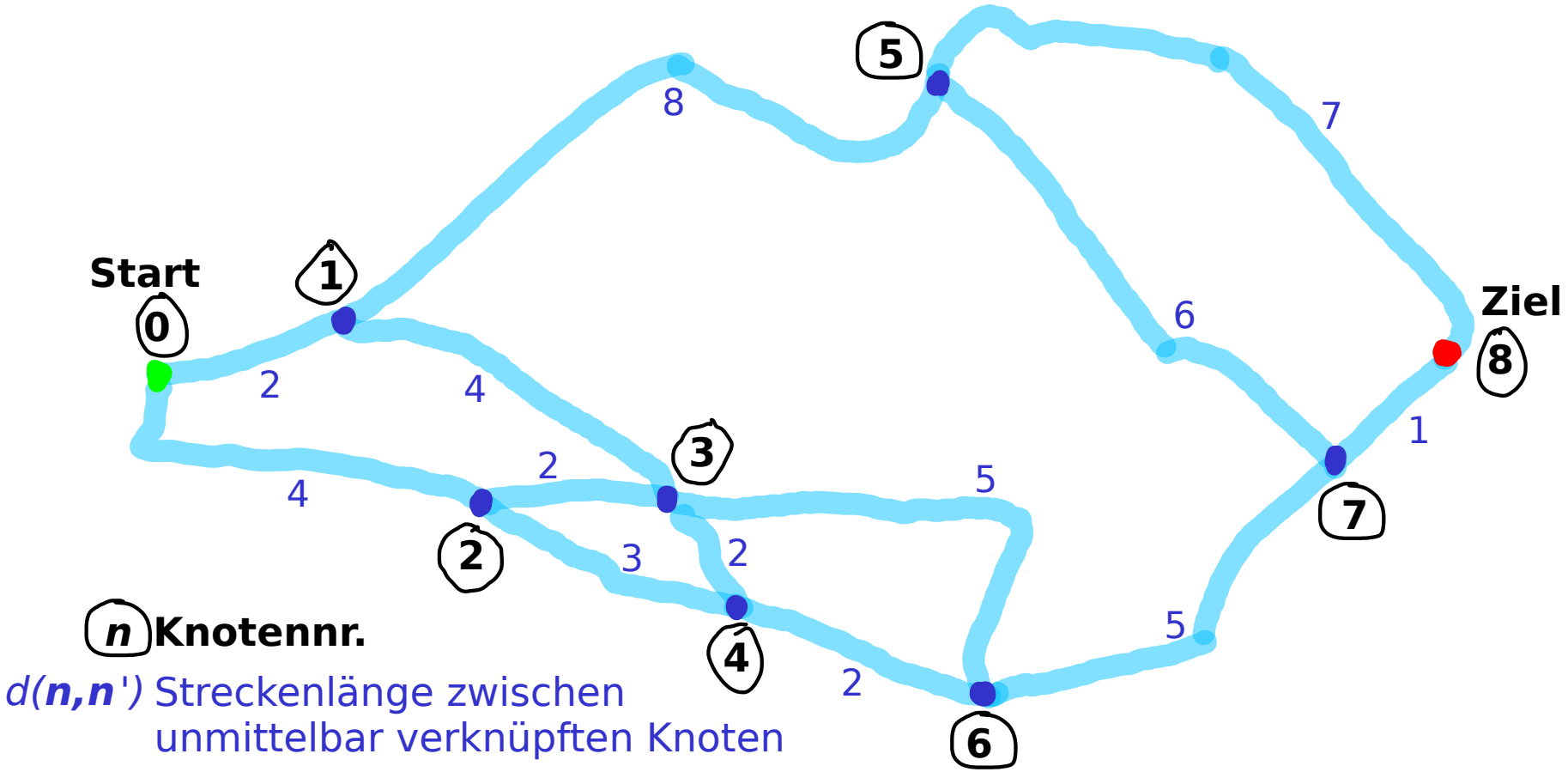

(Daten von OpenStreetMap. Veröffentlicht unter ODbL)

- Problem: Kürzester Fußweg vom Informatikzentrum zum SuperC

# Beispiel: kürzester Pfad

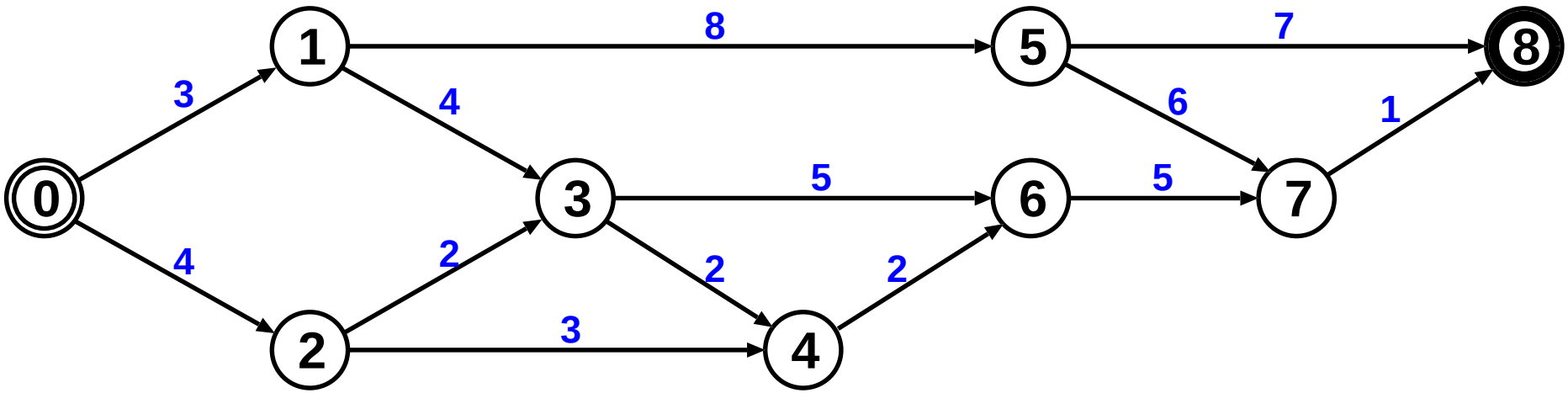

- Konstruktion des Beispiels:
  - Bestimmte Menge an Teilstrecken
  - Teilstrecken werden nur von links nach rechts/in östlicher Richtung verfolgt → gerichteter Graph ohne Zykel

# Beispiel: kürzester Pfad

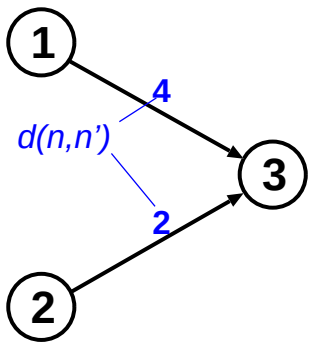



- Teillösungen: optimale Teilstrecken vom Start bis zum Knoten $n$
- Hilfsfunktion: optimale Teilstreckenlänge $D(n)$ zum Knoten $n$

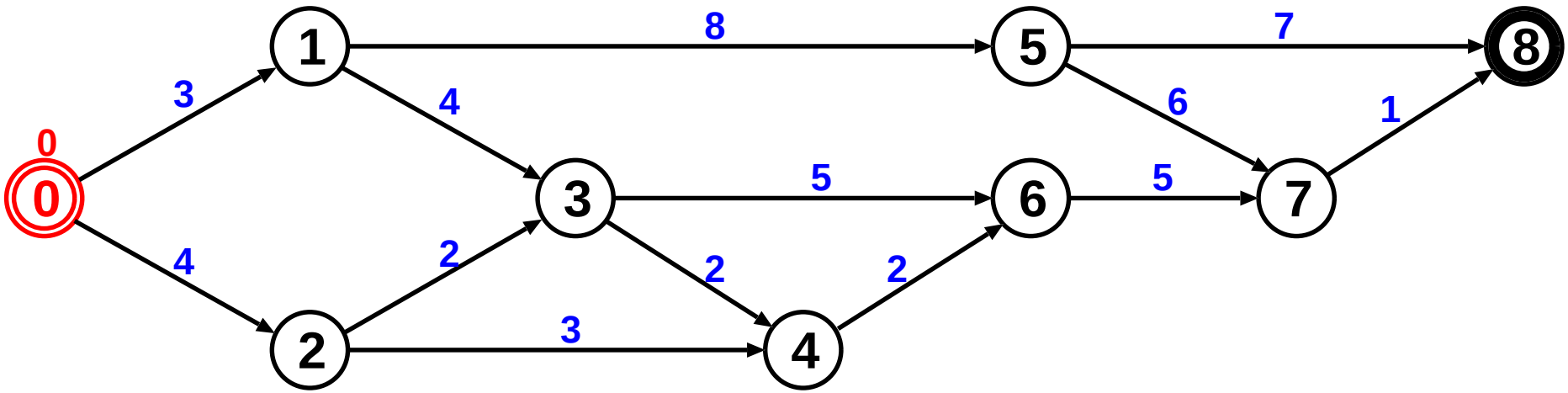
# Beispiel: Formalisiert




- Darstellung des Problems als gerichteten, azyklischen Graphen
- Strecke von Knoten $n'$ zu Knoten $n$: $d(n, n')$



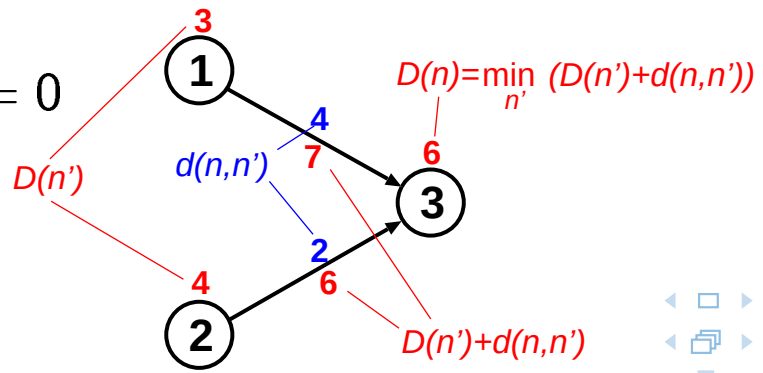
# Beispiel: Dynamische Programmierung



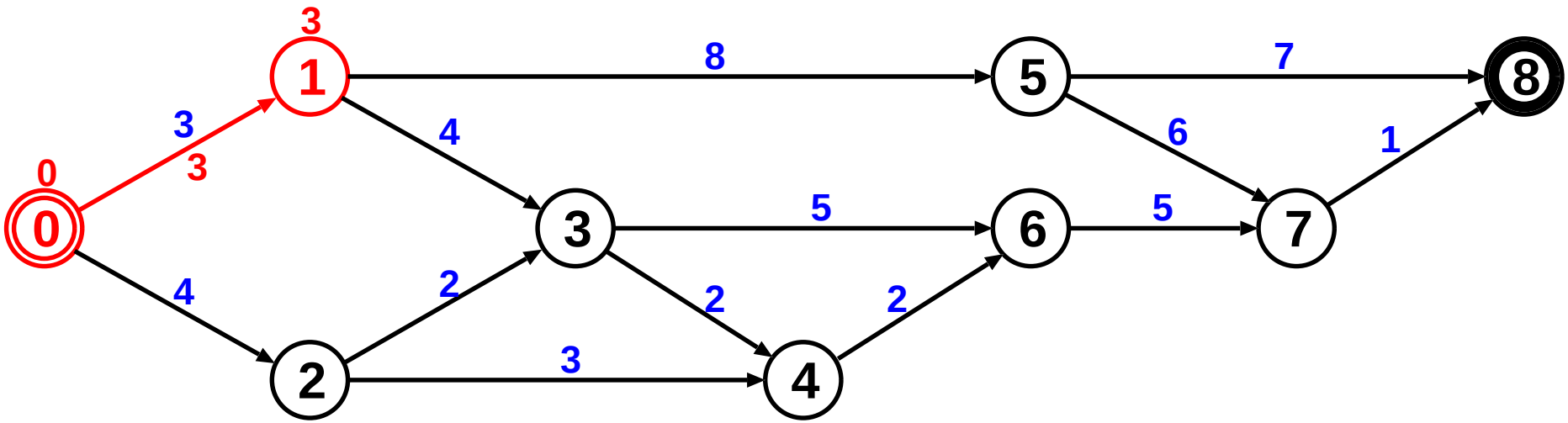

- Durchführung der dynamischen Programmierung
- Bestimmung optimaler Teilstrecken in topologisch aufsteigender Reihenfolge:
  Knoten $n = 0$: Randbedingung: $D(0) = 0$
  - Rekursive Berechnung:

$$D(n) = \min_{n'} \left( D(n') + d(n, n') \right)$$

Minimierung über Vorgängerknoten $n'$ von Knoten $n$



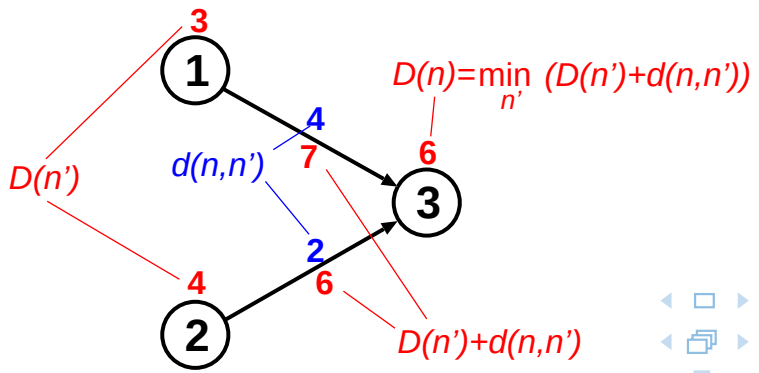
# Beispiel: Dynamische Programmierung



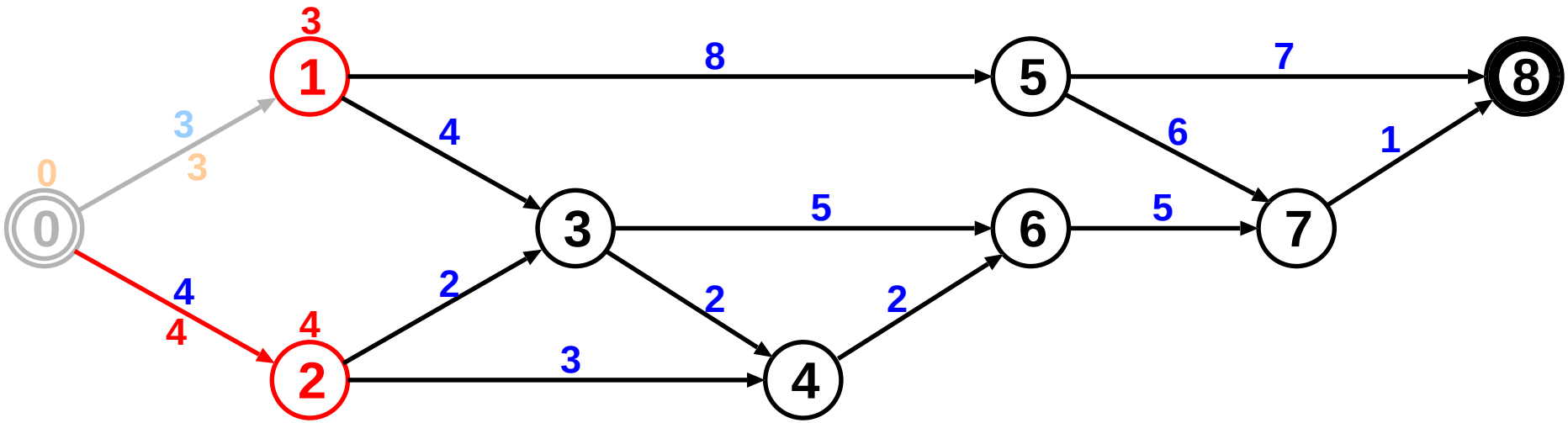

- Durchführung der dynamischen Programmierung
- Bestimmung optimaler Teilstrecken in topologisch aufsteigender Reihenfolge: Knoten $n = 1$
  - Rekursive Berechnung:

$$D(n) = \min_{n'} \left( D(n') + d(n, n') \right)$$

Minimierung über Vorgängerknoten $n'$ von Knoten $n$



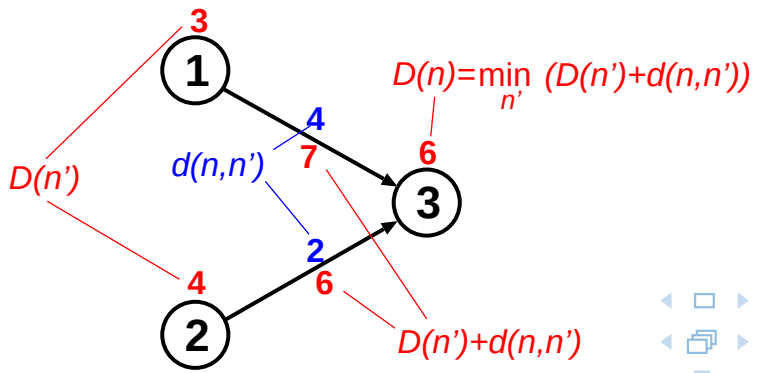
# Beispiel: Dynamische Programmierung



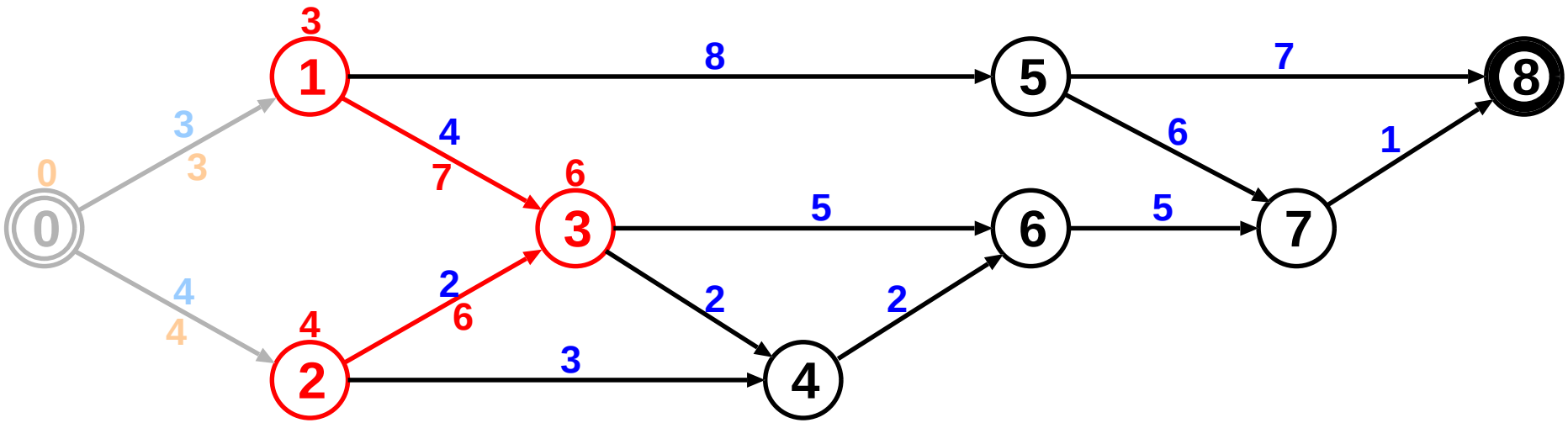

- Durchführung der dynamischen Programmierung
- Bestimmung optimaler Teilstrecken in topologisch aufsteigender Reihenfolge: Knoten $n = 2$
  - Rekursive Berechnung:

$$D(n) = \min_{n'} \left( D(n') + d(n, n') \right)$$

Minimierung über Vorgängerknoten $n'$ von Knoten $n$



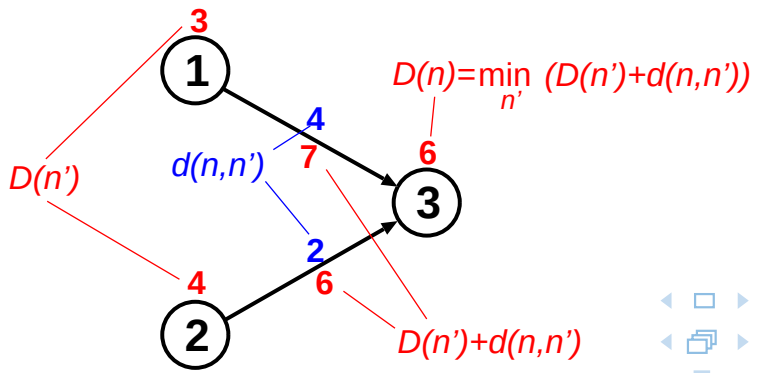
# Beispiel: Dynamische Programmierung



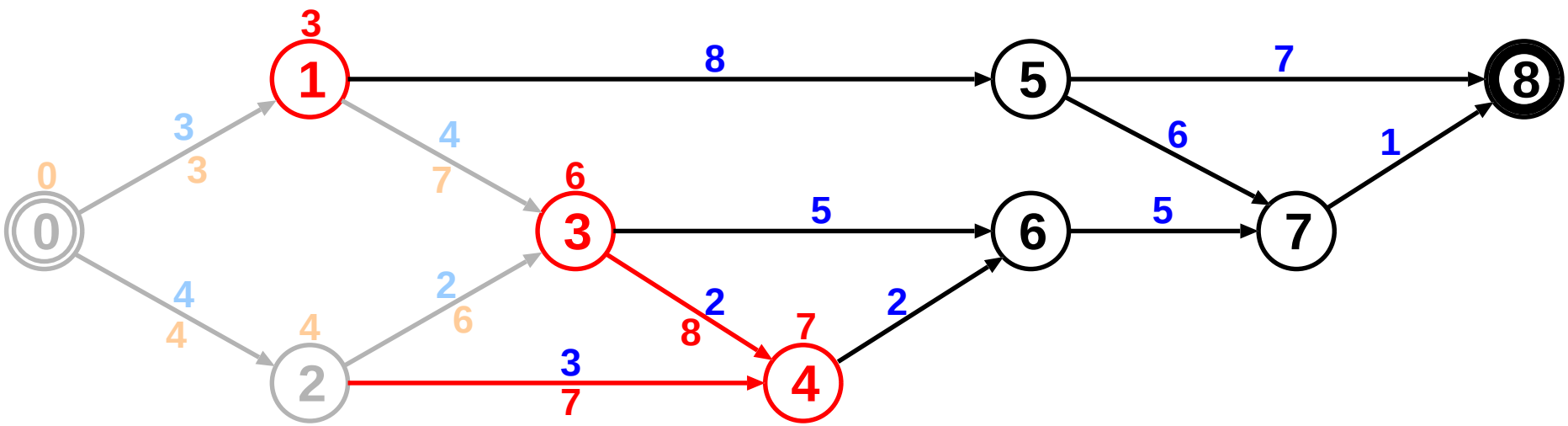

- Durchführung der dynamischen Programmierung
- Bestimmung optimaler Teilstrecken in topologisch aufsteigender Reihenfolge: Knoten $n = 3$
  - Rekursive Berechnung:

$$D(n) = \min_{n'} \left( D(n') + d(n, n') \right)$$

Minimierung über Vorgängerknoten $n'$ von Knoten $n$



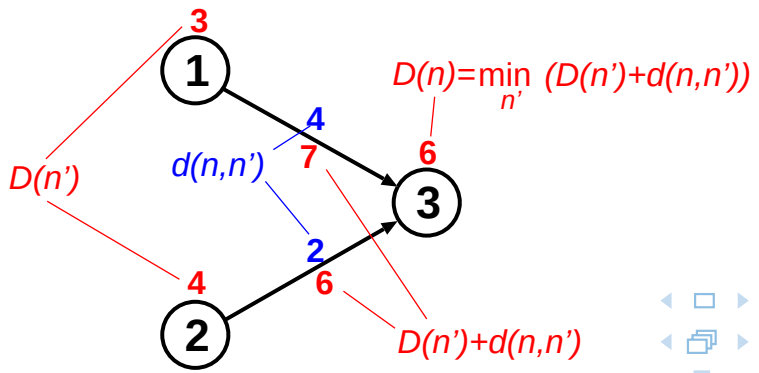
# Beispiel: Dynamische Programmierung



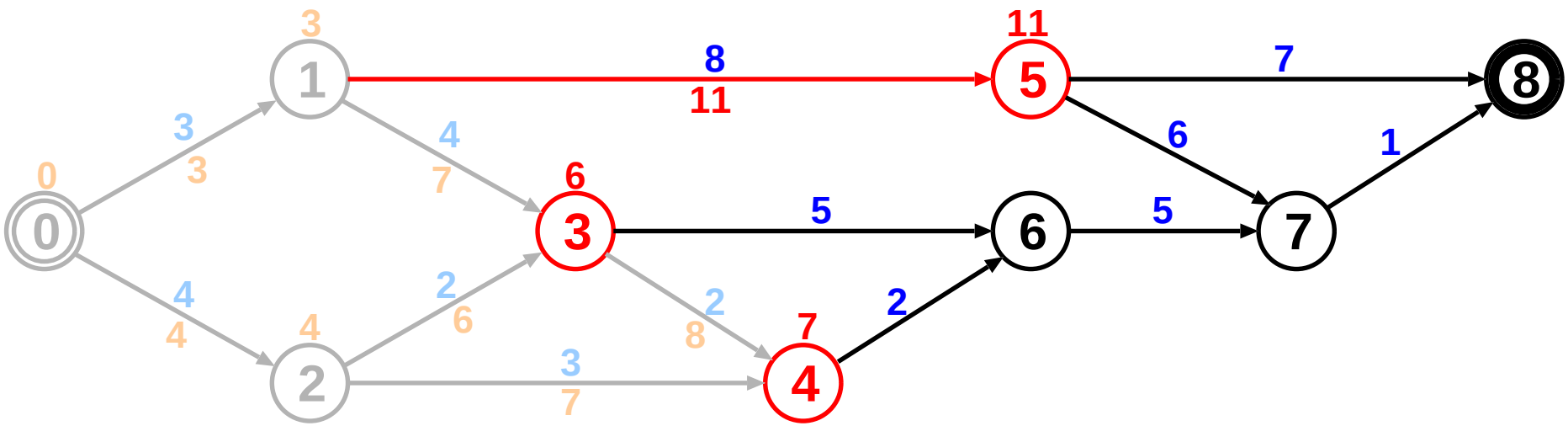

- Durchführung der dynamischen Programmierung
- Bestimmung optimaler Teilstrecken in topologisch aufsteigender Reihenfolge: Knoten $n = 4$
  - Rekursive Berechnung:

$$D(n) = \min_{n'} \left( D(n') + d(n, n') \right)$$

Minimierung über Vorgängerknoten $n'$ von Knoten $n$



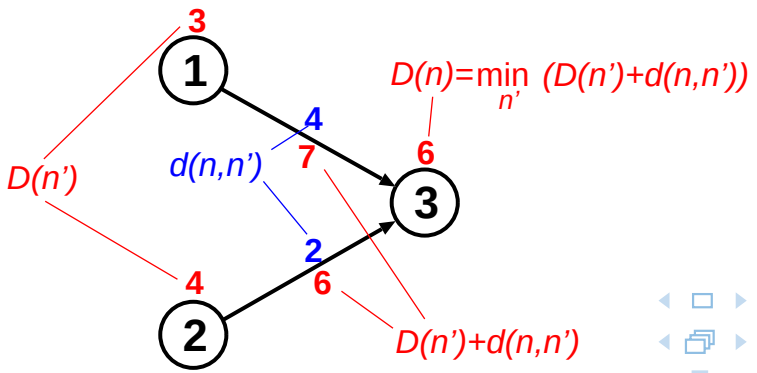
# Beispiel: Dynamische Programmierung



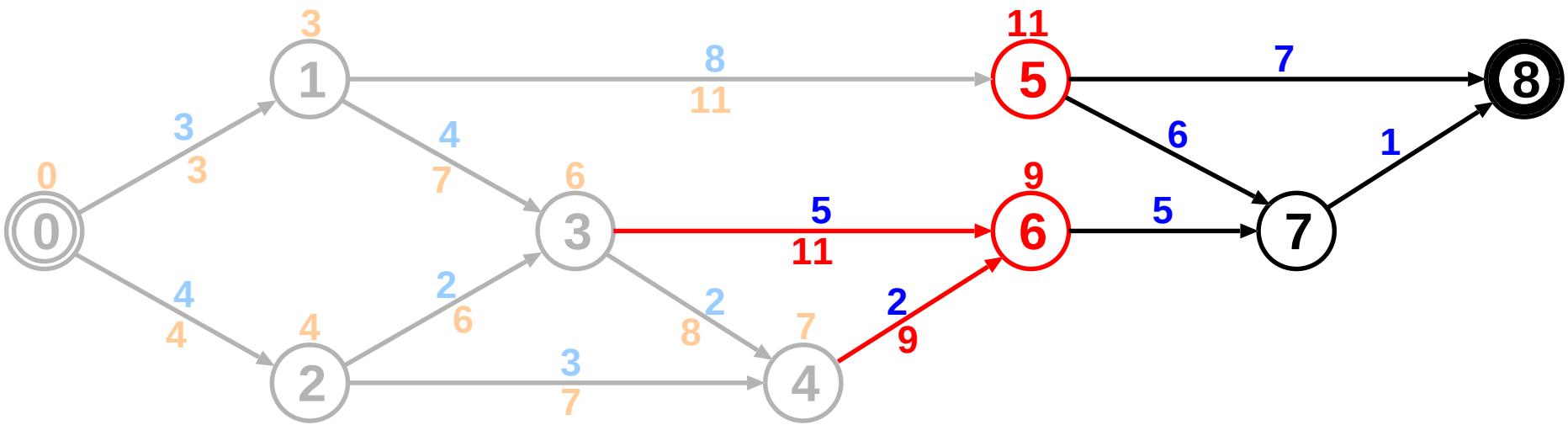

- Durchführung der dynamischen Programmierung
- Bestimmung optimaler Teilstrecken in topologisch aufsteigender Reihenfolge: Knoten $n = 5$
  - Rekursive Berechnung:

$$D(n) = \min_{n'} \left( D(n') + d(n, n') \right)$$

Minimierung über Vorgängerknoten $n'$ von Knoten $n$



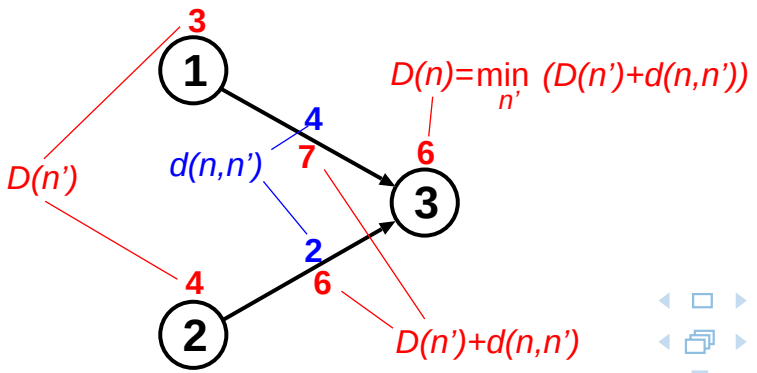
# Beispiel: Dynamische Programmierung



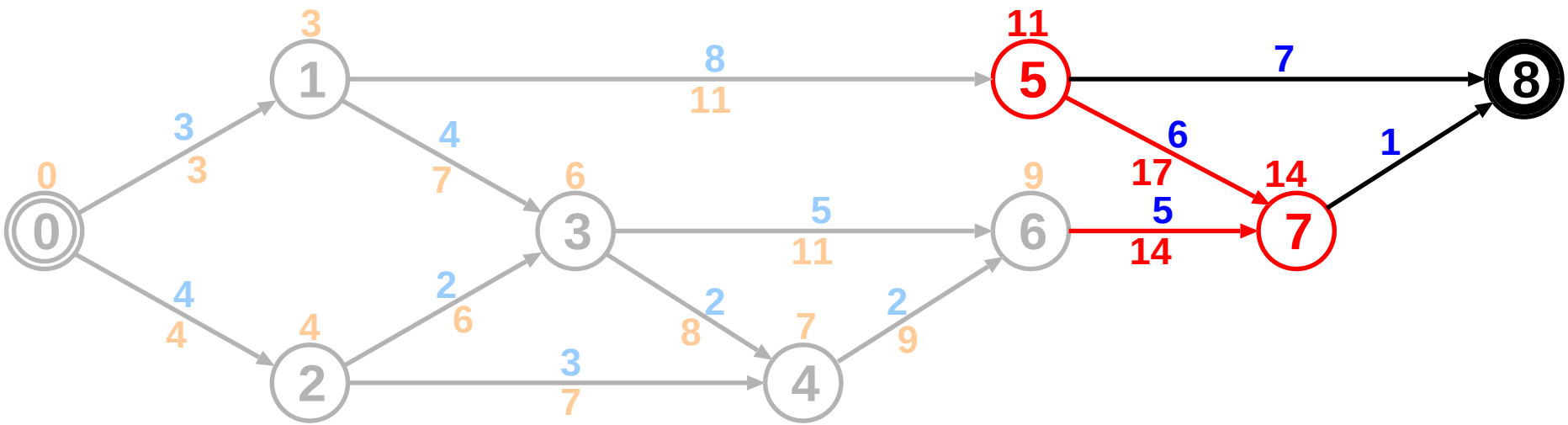

- Durchführung der dynamischen Programmierung
- Bestimmung optimaler Teilstrecken in topologisch aufsteigender Reihenfolge: Knoten $n = 6$
  - Rekursive Berechnung:

$$D(n) = \min_{n'} \left( D(n') + d(n, n') \right)$$

Minimierung über Vorgängerknoten $n'$ von Knoten $n$



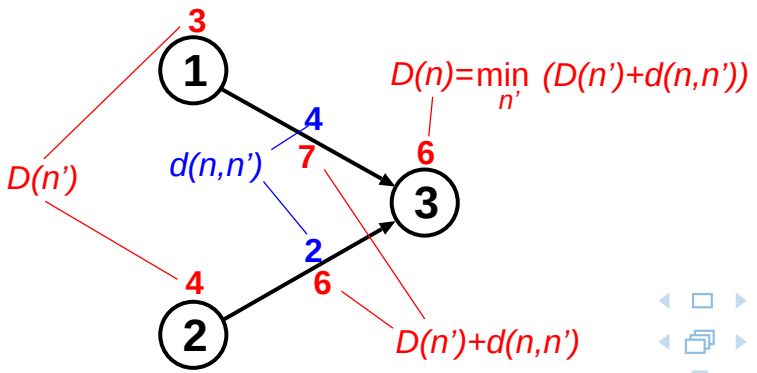
# Beispiel: Dynamische Programmierung



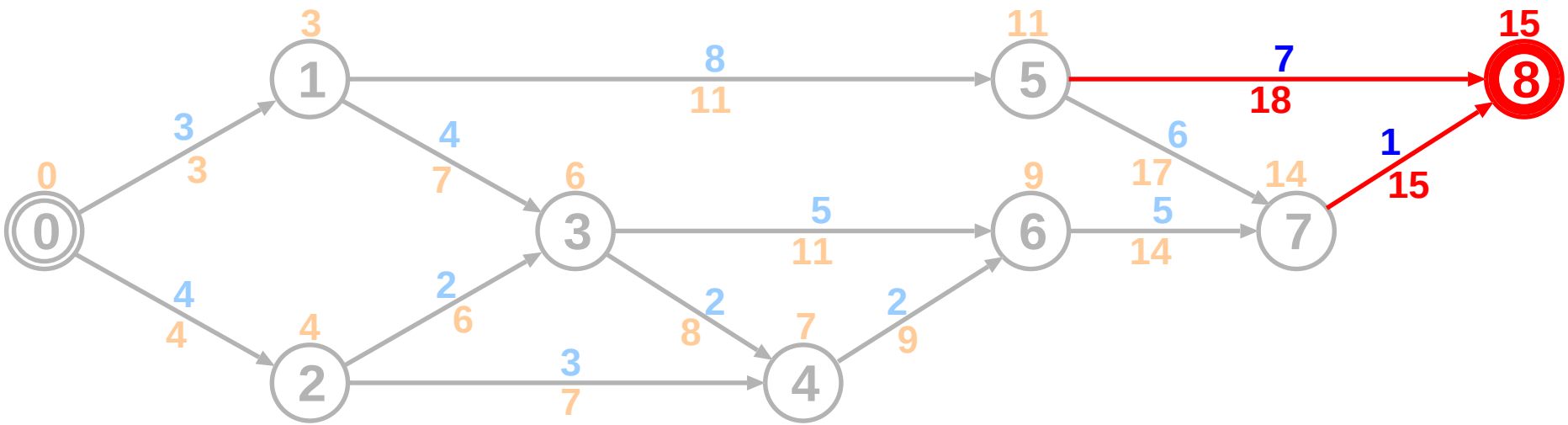

- Durchführung der dynamischen Programmierung
- Bestimmung optimaler Teilstrecken in topologisch aufsteigender Reihenfolge: Knoten $n = 7$
  - Rekursive Berechnung:

$$D(n) = \min_{n'} \left( D(n') + d(n, n') \right)$$

Minimierung über Vorgängerknoten $n'$ von Knoten $n$



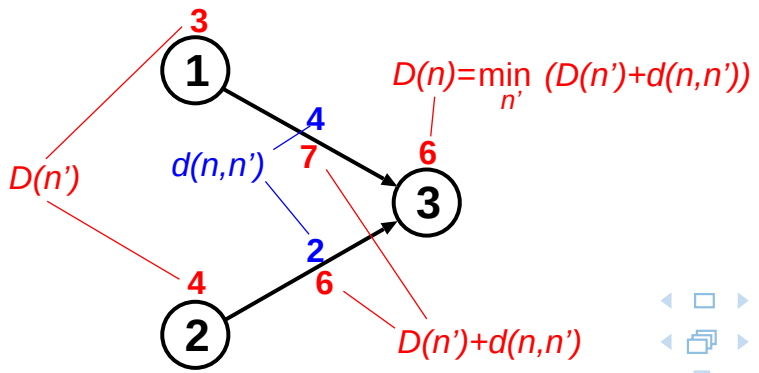
# Beispiel: Dynamische Programmierung



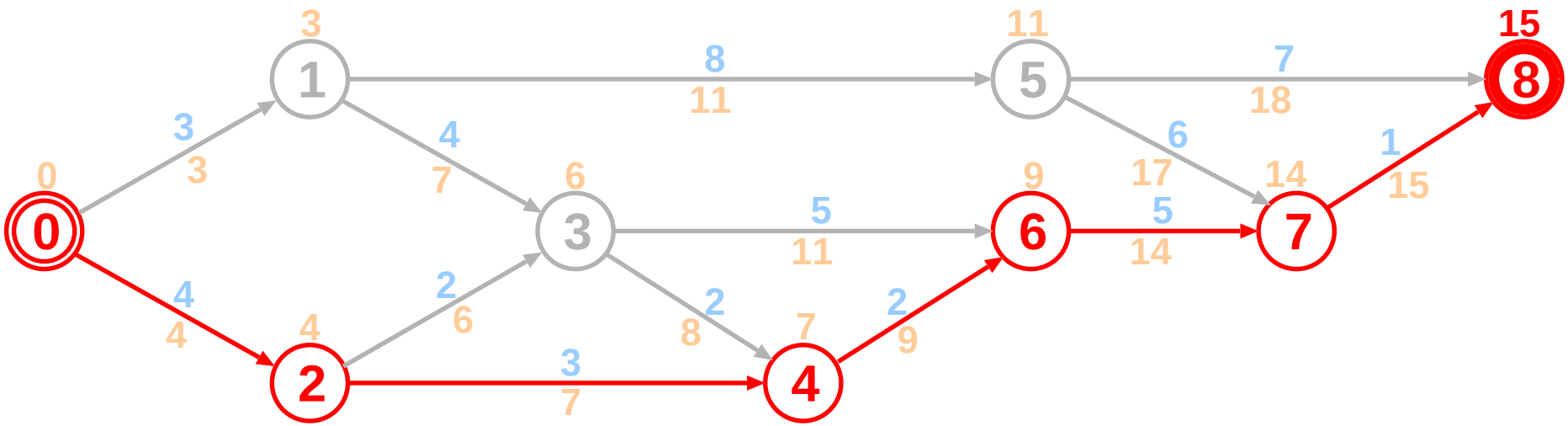

- Durchführung der dynamischen Programmierung
- Bestimmung optimaler Teilstrecken in topologisch aufsteigender Reihenfolge: Knoten $n = 8$
  - Rekursive Berechnung:

$$D(n) = \min_{n'} \left( D(n') + d(n, n') \right)$$

Minimierung über Vorgängerknoten $n'$ von Knoten $n$

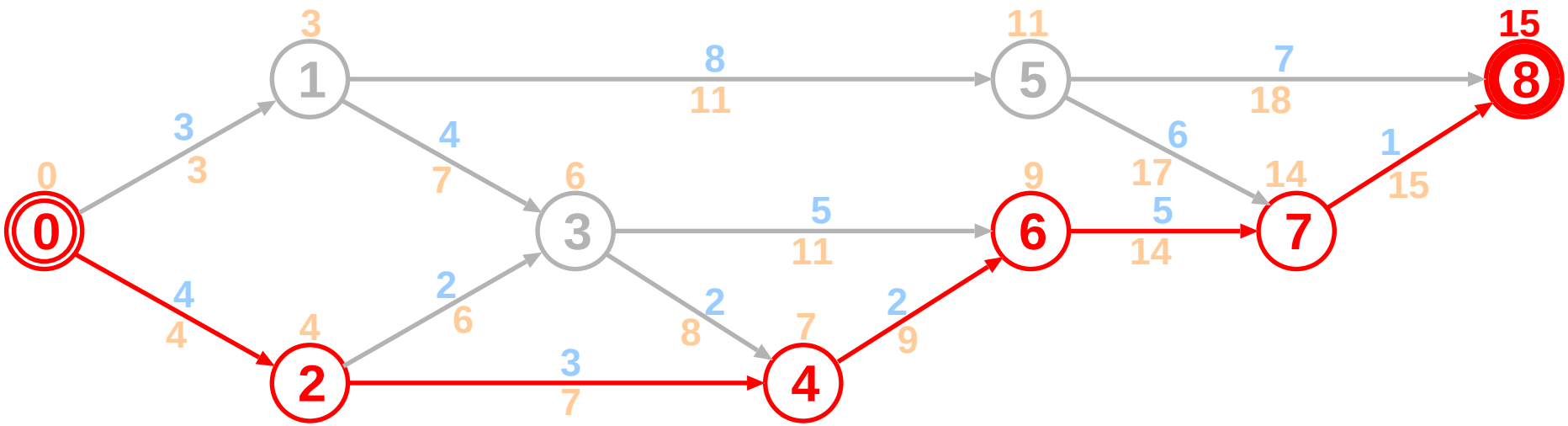


# Beispiel: Dynamische Programmierung




- Abschluss: optimaler Pfad.
- Komplexität: $\sim$ # Knoten & mittl. # einlaufender Kanten
  - Zeit: 13 Schritte
  - Speicher: 8 Knoten (nur Pfadlängenberechnung: max. 3 zugleich)

# Beispiel: Dynamische Programmierung




- Beispiel: vereinfachte Form des *Single Source Shortest Path* Problems
  - Wie z.T. in der Literatur diskutiert, wird dieses Problem zwar mit einem *Greedy*-Algorithmus optimal gelöst, dieser ist in diesem Fall jedoch nicht effizient [Schöning 2001].
  - Eine effiziente Lösung liefert der *Dijkstra*-Algorithmus, der als eine Umsetzung des Prinzips der dynamischen Programmierung angesehen werden kann [Schöning 2001].

# Dynamische Programmierung gegen Teilen & Herrschen

Grundidee: Sukzessive Aufteilung eines Problems in Teilprobleme

*Dynamische Programmierung*: *Bottom-Up*-Methode

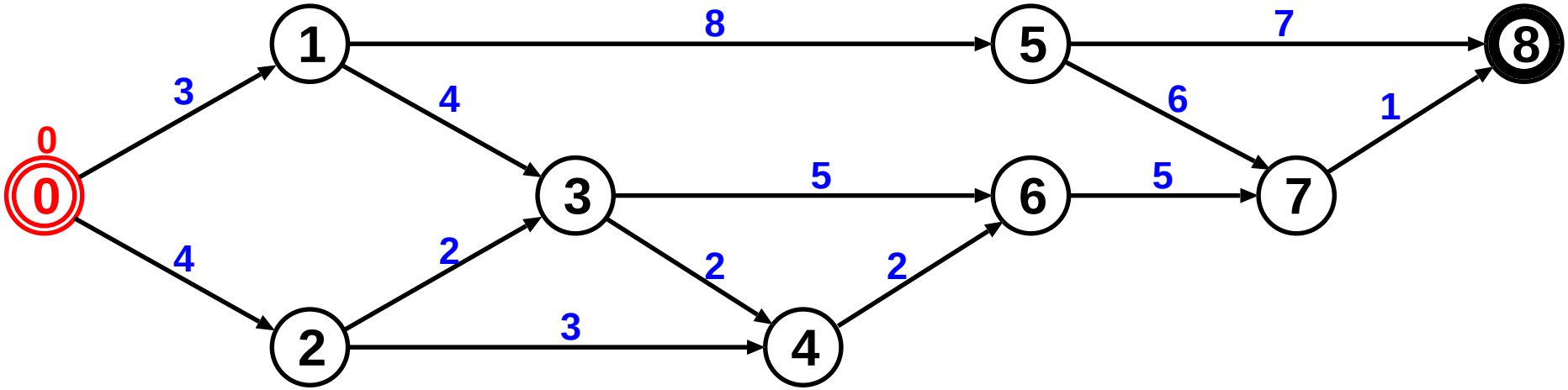
- Konzertierte Lösung der Teilprobleme
- Berücksichtigung von Redundanzen:
  Teilprobleme werden jeweils einmalig gelöst und tabelliert
- Ablauf der Berechnung als Graph darstellbar:
  unterschiedliche höhere Teilprobleme greifen
  auf gleiche Teillösungen zu

*Teilen und Herrschen*: *Top-Down*-Methode

- Separate Lösung der Teilprobleme (rekursiv)
- Keine Berücksichtigung von Redundanzen:
  Teilprobleme können auf gemeinsamen Teilproblemen beruhen
  deren ggf. schon erfolgte Lösung nicht ausgenutzt wird
- Ablauf der Berechnung als Baumstruktur darstellbar:
  Teilprobleme werden unabhängig voneinander gelöst

# Beispiel: Teilen und Herrschen - *Top-Down*...

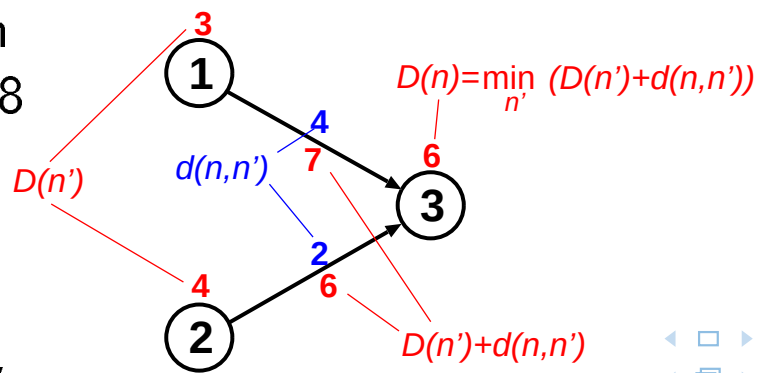
...oder wie man DP nicht implementieren sollte




- Notation/Darstellung wie gehabt: Teilstrecken $D(n)$
- Randbedingung am Startknoten: $D(0) = 0$
- Bestimmung optimaler Teilstrecken nun rekursiv beginnend bei Endknoten $n = 8$

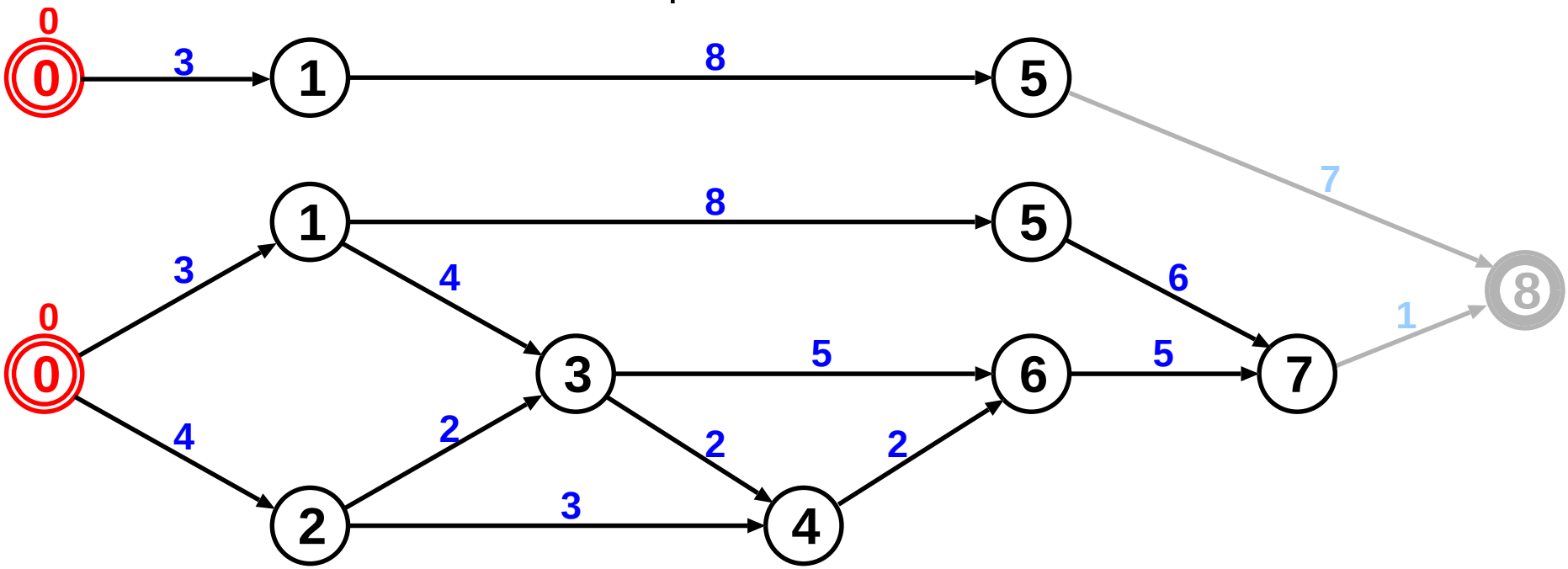
$$D(n) = \min_{n'} \left( D(n') + d(n, n') \right)$$

Minimierung am Knoten $n$ ruft rekursiv Lösungen $D(n')$ an Vorgängerknoten auf



# Beispiel: Teilen und Herrschen - *Top-Down*...

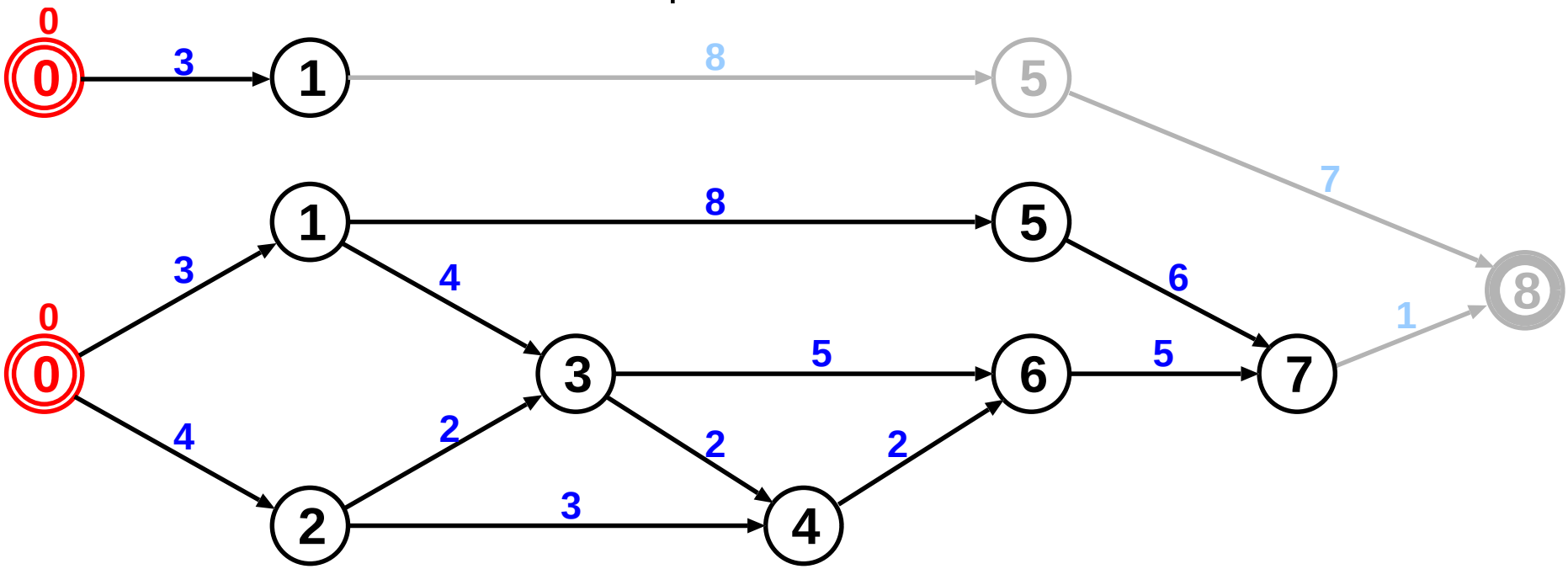
...oder wie man DP nicht implementieren sollte




- Beginnend mit Endknoten $n = 8$: rekursiver Abruf von Lösungen für Teilprobleme

# Beispiel: Teilen und Herrschen - *Top-Down*...

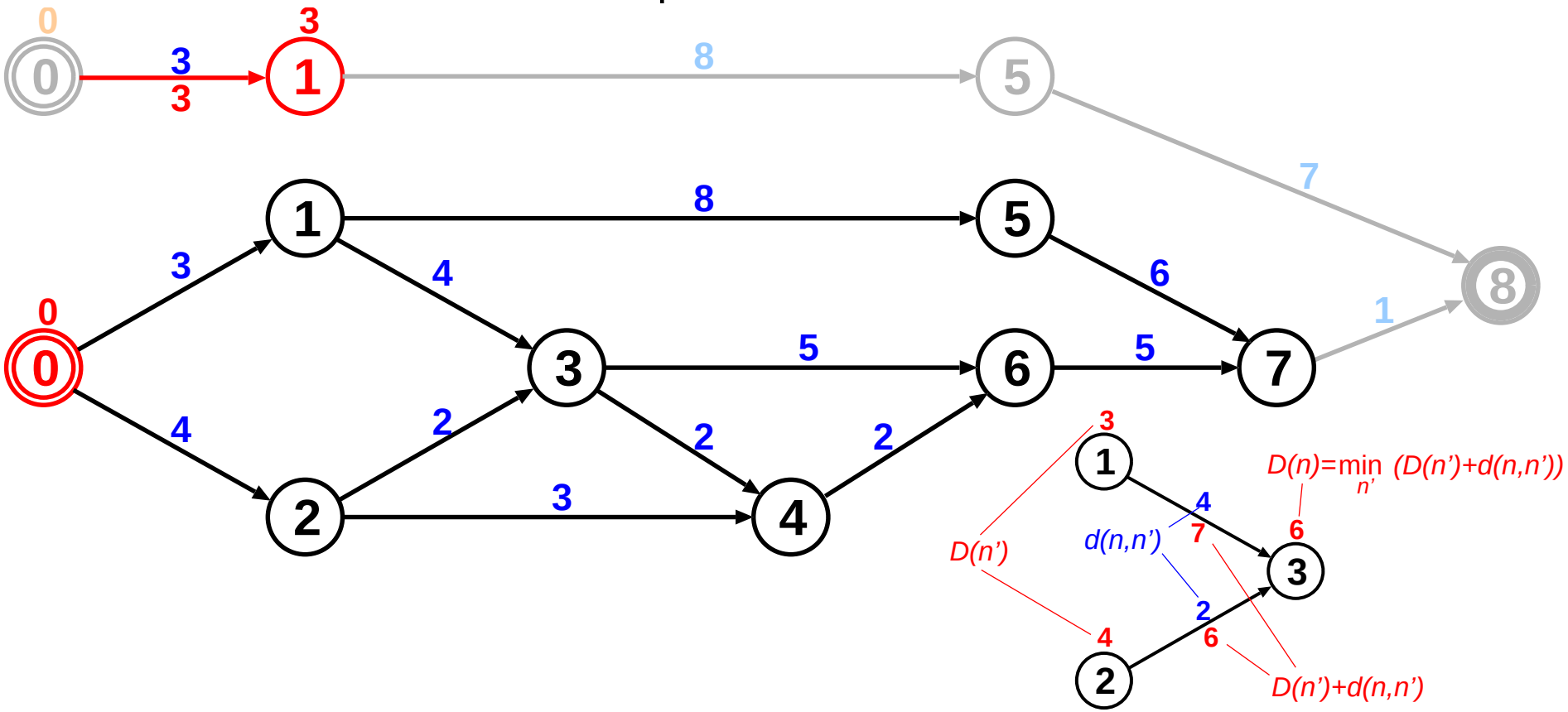
...oder wie man DP nicht implementieren sollte




- Beginnend mit Endknoten $n = 8$: rekursiver Abruf von Lösungen für Teilprobleme

# Beispiel: Teilen und Herrschen - *Top-Down*...

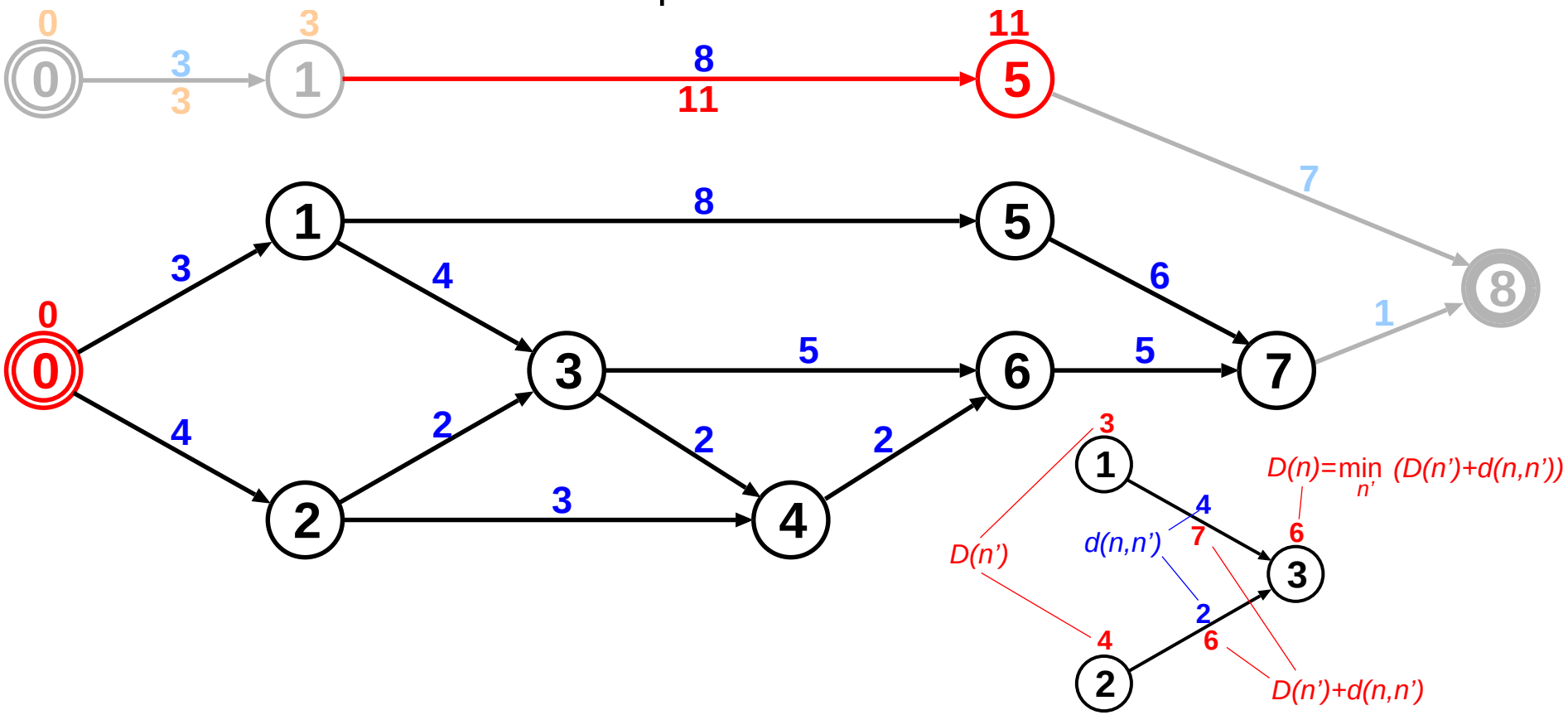
...oder wie man DP nicht implementieren sollte




- Beginnend mit Endknoten $n = 8$: rekursiver Abruf von Lösungen für Teilprobleme:
- Erreicht Ablauf Startknoten, kann Berechnung beginnen,

# Beispiel: Teilen und Herrschen - *Top-Down*...

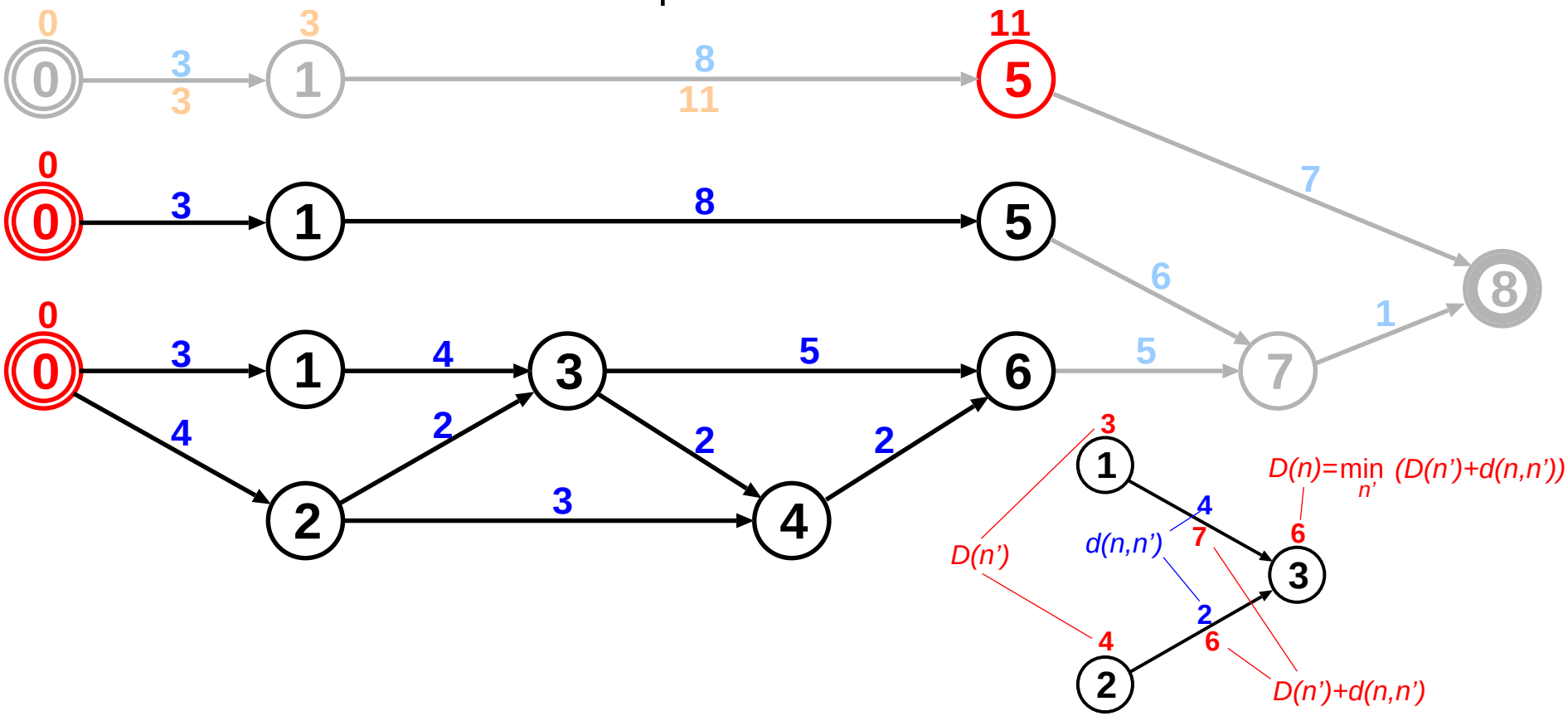
...oder wie man DP nicht implementieren sollte




- Beginnend mit Endknoten $n = 8$: rekursiver Abruf von Lösungen für Teilprobleme:
- Erreicht Ablauf Startknoten, kann Berechnung beginnen,
- und Ergebnisse zu höheren Teilproblemen weitergereicht

# Beispiel: Teilen und Herrschen - *Top-Down*...

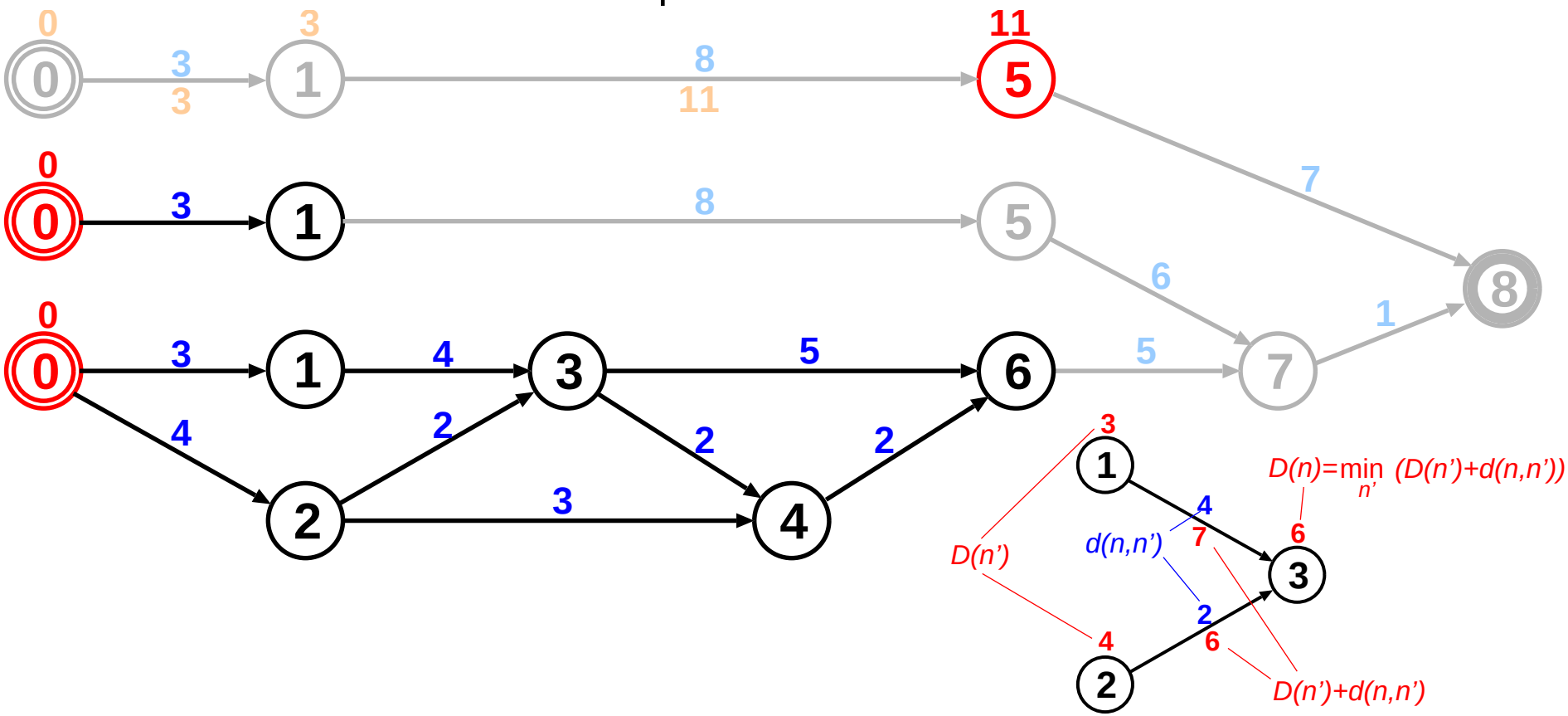
...oder wie man DP nicht implementieren sollte




- Beginnend mit Endknoten $n = 8$: rekursiver Abruf von Lösungen für Teilprobleme:
- Erreicht Ablauf Startknoten, kann Berechnung beginnen,
- und Ergebnisse zu höheren Teilproblemen weitergereicht

# Beispiel: Teilen und Herrschen - *Top-Down*...

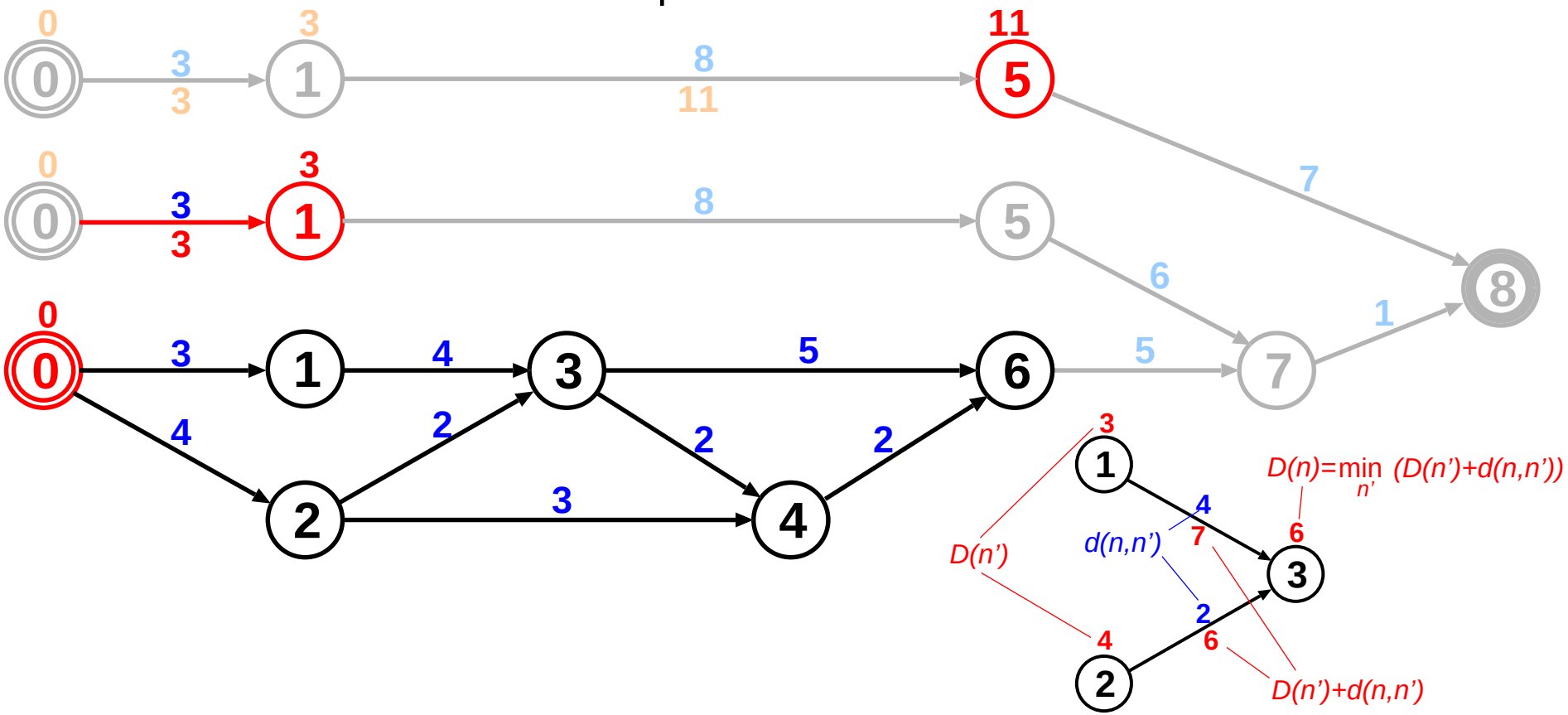
...oder wie man DP nicht implementieren sollte




- Beginnend mit Endknoten $n = 8$: rekursiver Abruf von Lösungen für Teilprobleme:
- Erreicht Ablauf Startknoten, kann Berechnung beginnen,
- und Ergebnisse zu höheren Teilproblemen weitergereicht

# Beispiel: Teilen und Herrschen - *Top-Down*...

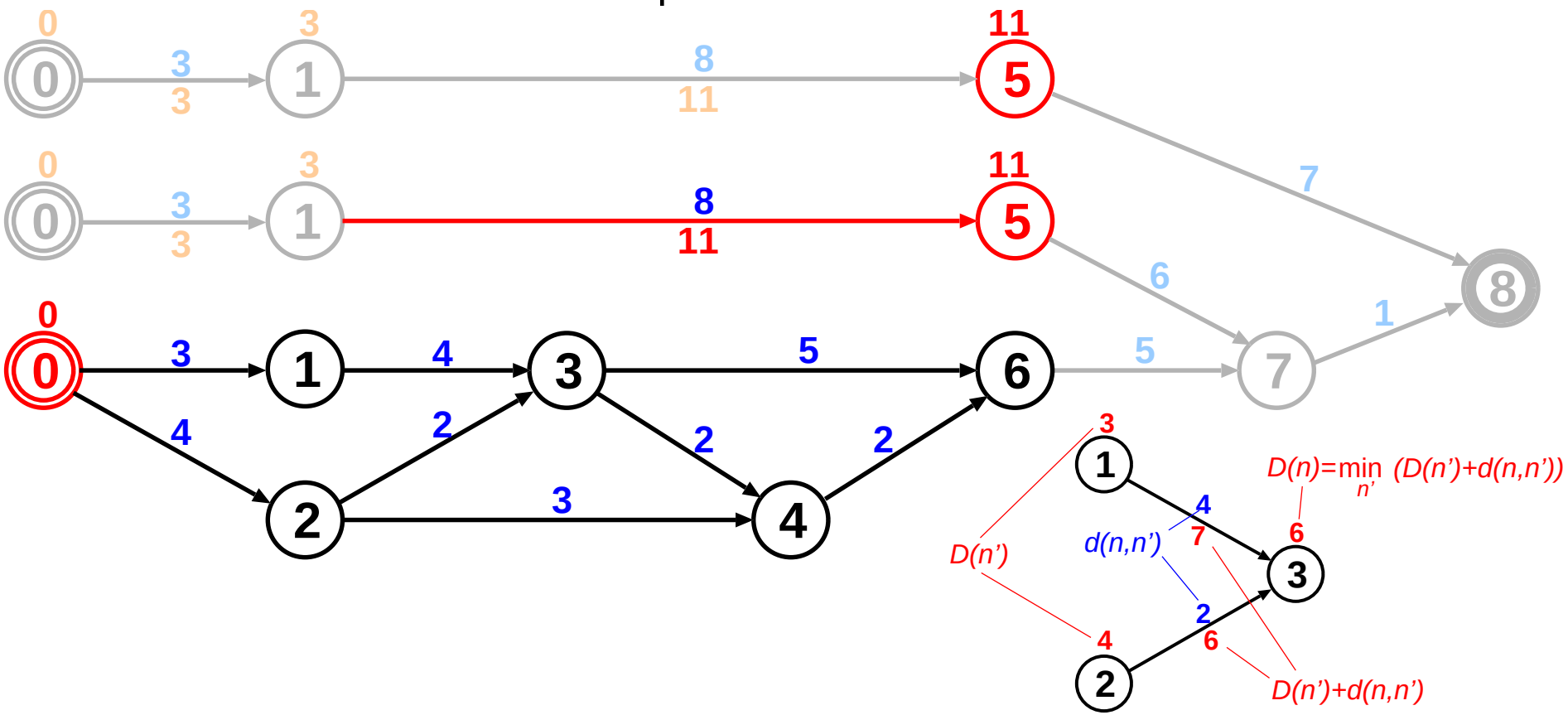
...oder wie man DP nicht implementieren sollte




- Beginnend mit Endknoten $n = 8$: rekursiver Abruf von Lösungen für Teilprobleme:
- Erreicht Ablauf Startknoten, kann Berechnung beginnen,
- und Ergebnisse zu höheren Teilproblemen weitergereicht

# Beispiel: Teilen und Herrschen - *Top-Down*...

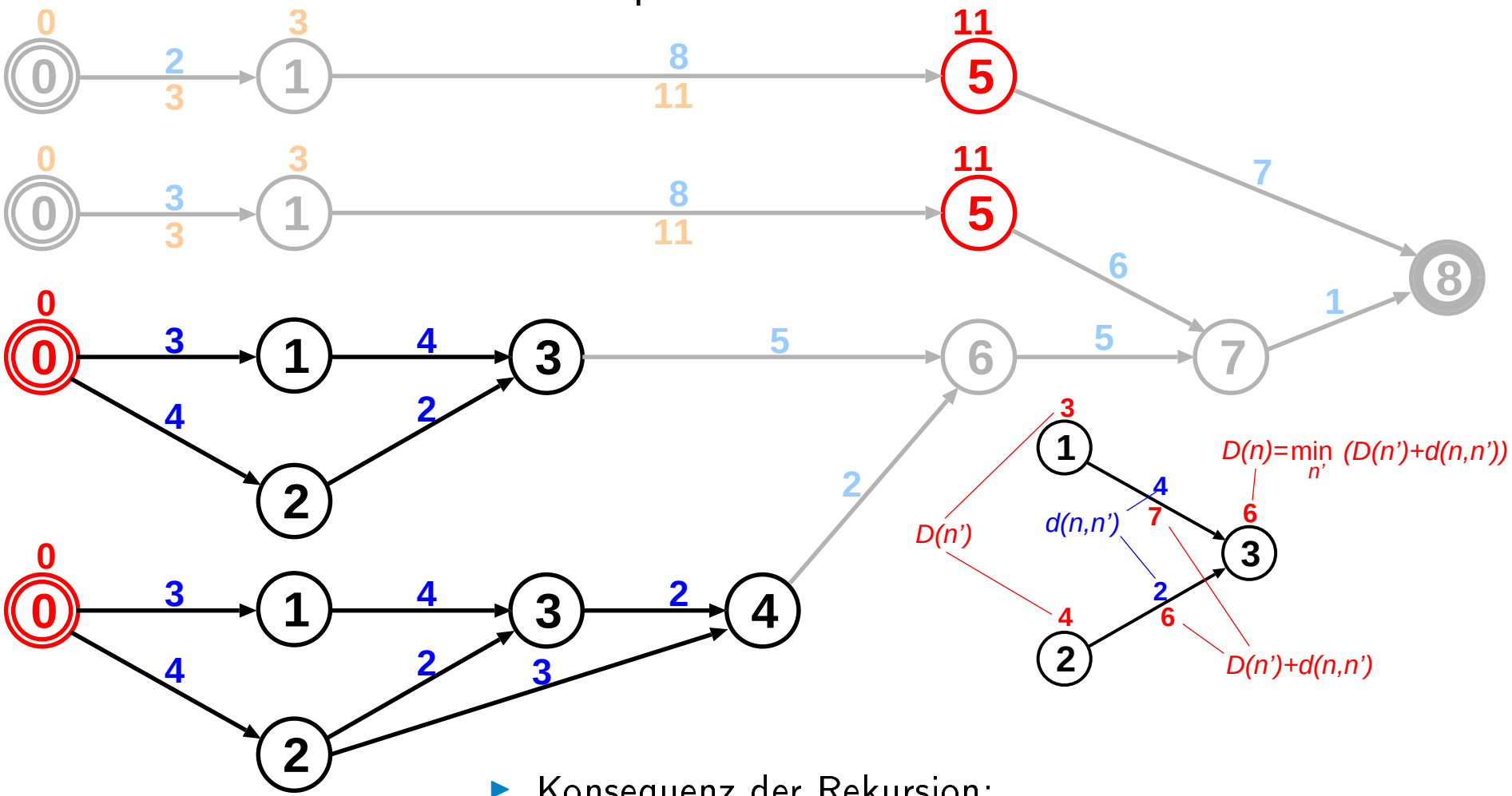
...oder wie man DP nicht implementieren sollte




- Beginnend mit Endknoten $n = 8$: rekursiver Abruf von Lösungen für Teilprobleme:
- Erreicht Ablauf Startknoten, kann Berechnung beginnen,
- und Ergebnisse zu höheren Teilproblemen weitergereicht

# Beispiel: Teilen und Herrschen - *Top-Down*...

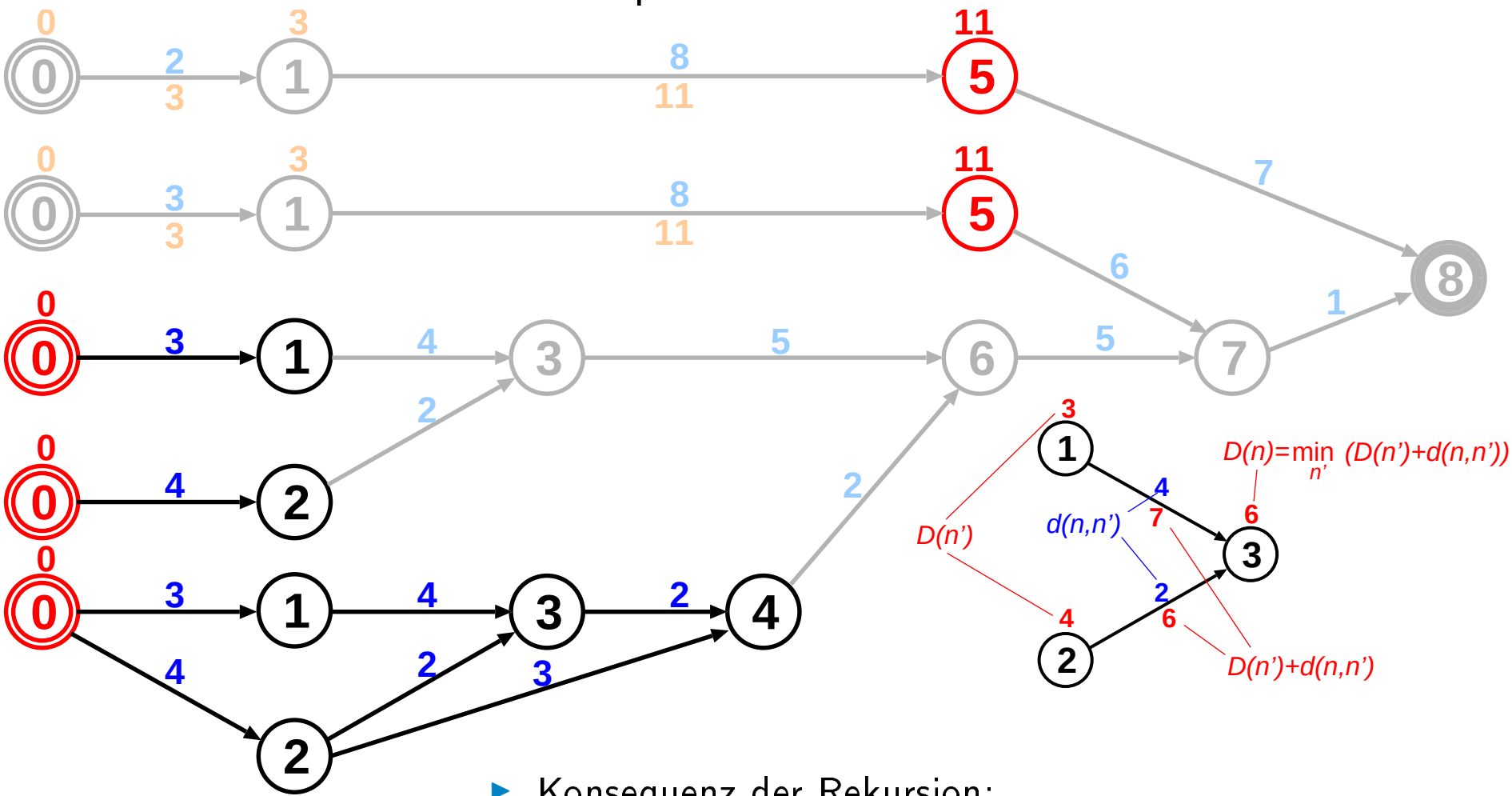
...oder wie man DP nicht implementieren sollte




- Konsequenz der Rekursion:
  sukzessive Aufspaltung des Graphen

# Beispiel: Teilen und Herrschen - *Top-Down*...

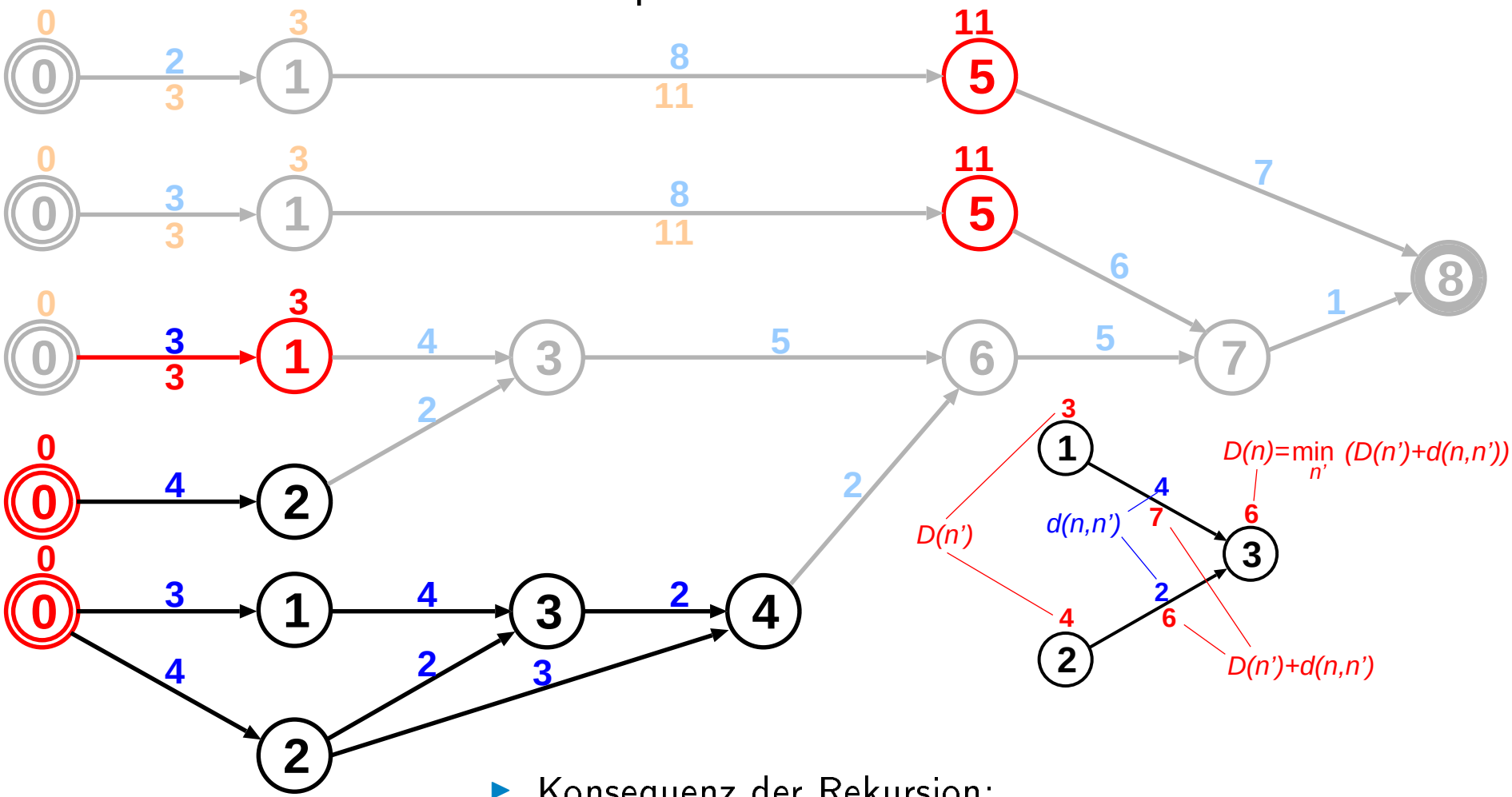
...oder wie man DP nicht implementieren sollte




- Konsequenz der Rekursion:
  sukzessive Aufspaltung des Graphen

# Beispiel: Teilen und Herrschen - *Top-Down*...

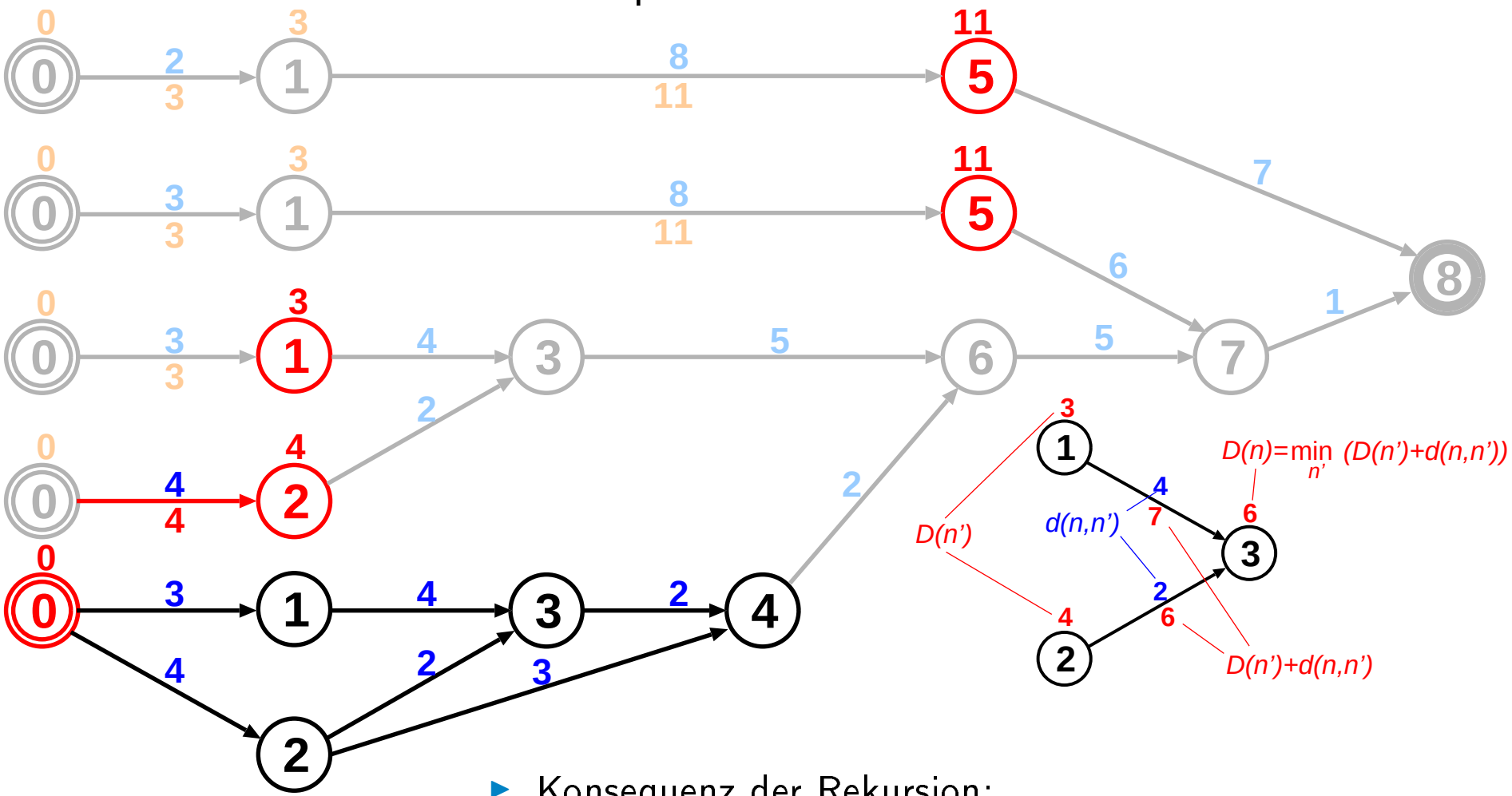
...oder wie man DP nicht implementieren sollte




- Konsequenz der Rekursion:
  sukzessive Aufspaltung des Graphen

# Beispiel: Teilen und Herrschen - *Top-Down*...

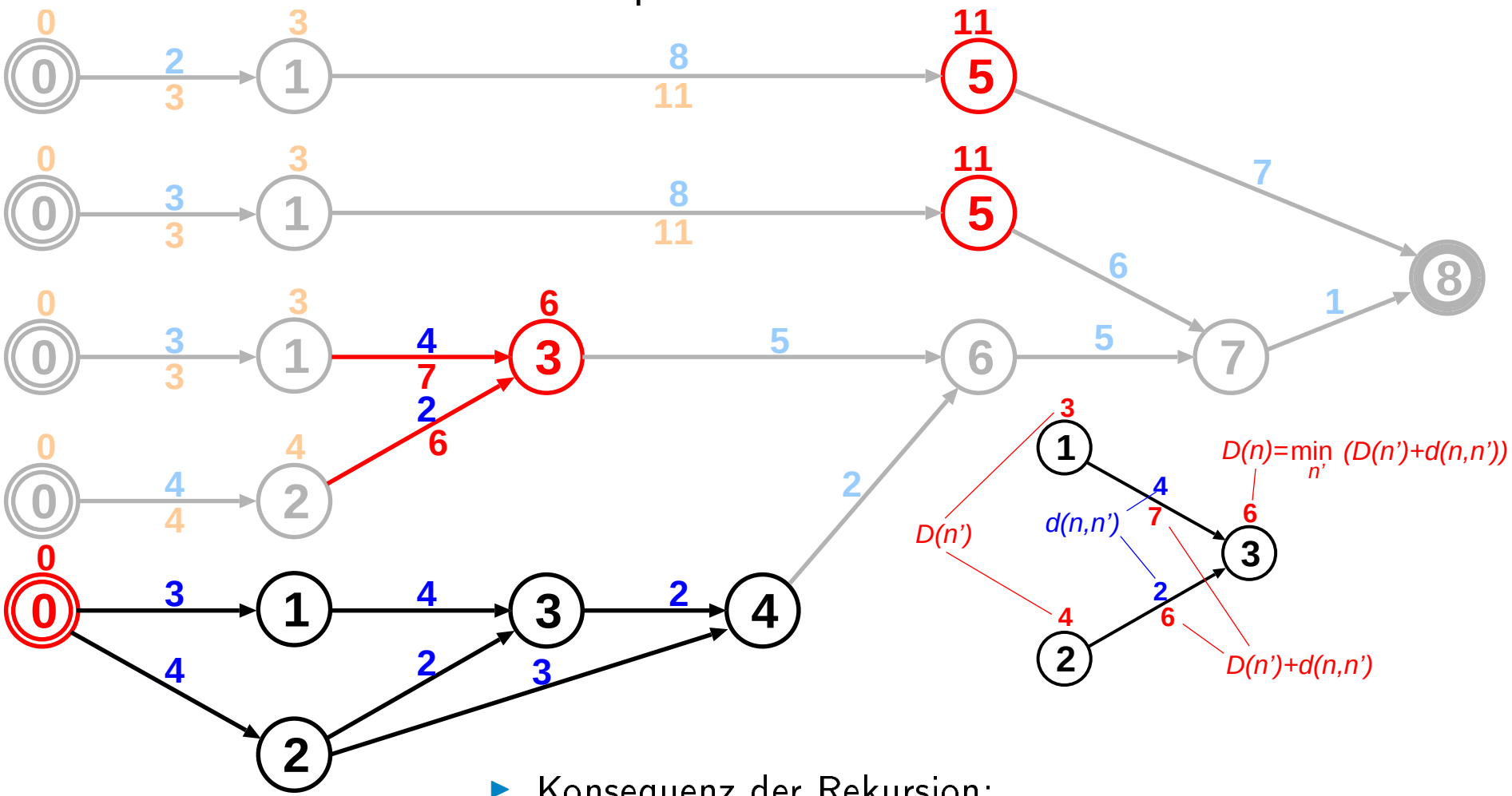
...oder wie man DP nicht implementieren sollte




- Konsequenz der Rekursion:
  sukzessive Aufspaltung des Graphen

# Beispiel: Teilen und Herrschen - *Top-Down*...

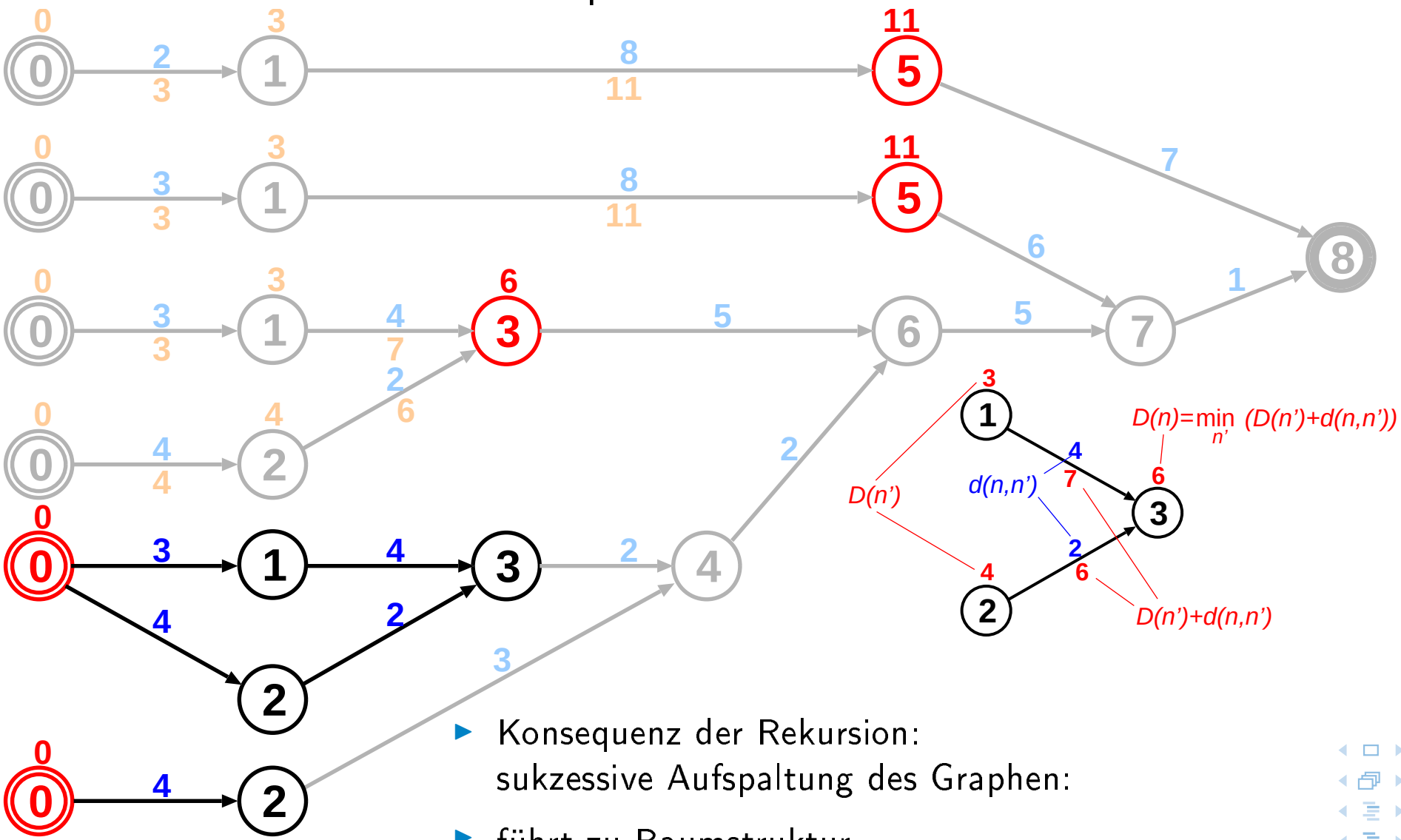
...oder wie man DP nicht implementieren sollte




- Konsequenz der Rekursion:
  sukzessive Aufspaltung des Graphen

# Beispiel: Teilen und Herrschen - *Top-Down*...

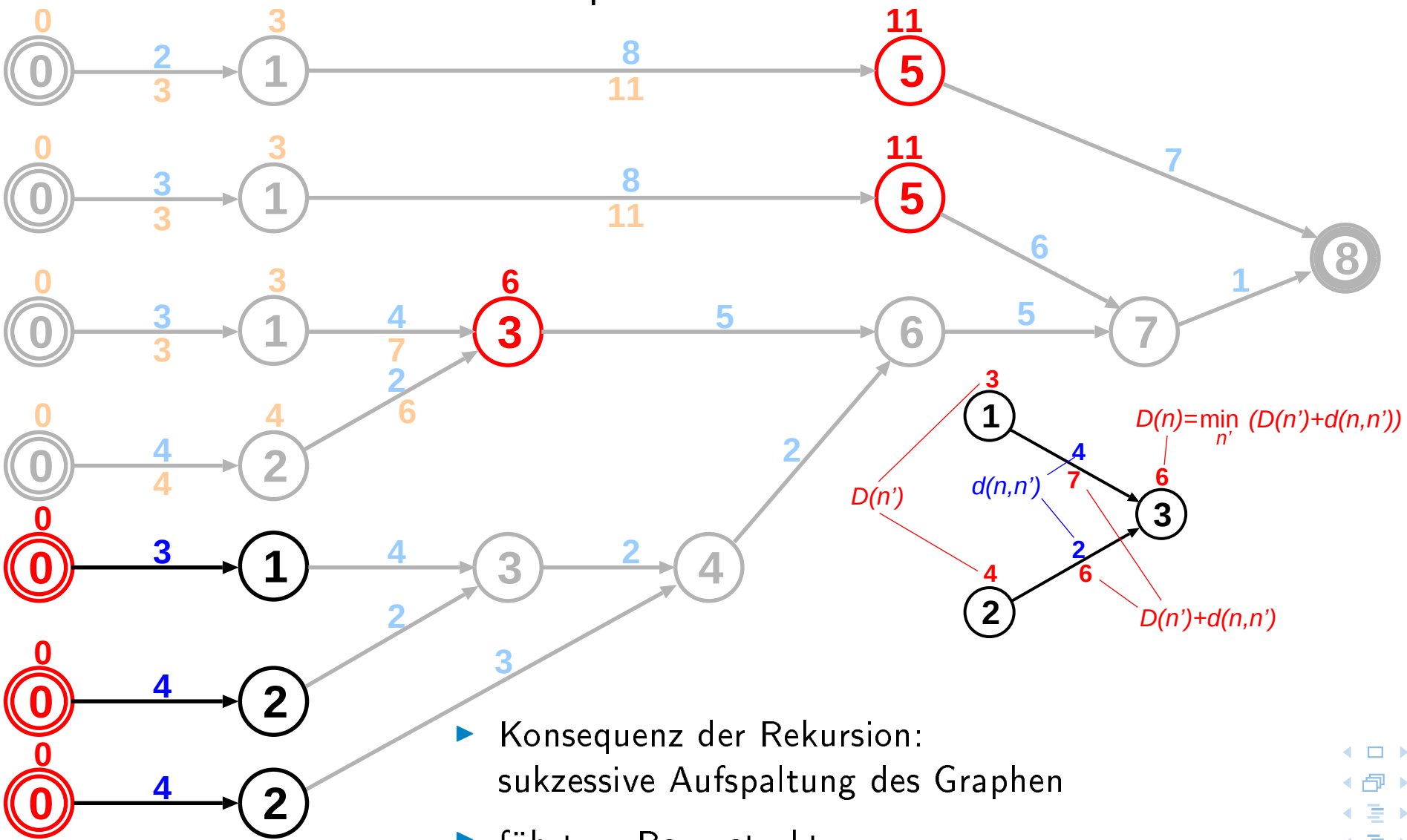
...oder wie man DP nicht implementieren sollte




- Konsequenz der Rekursion:
  sukzessive Aufspaltung des Graphen:
- führt zu Baumstruktur

# Beispiel: Teilen und Herrschen - *Top-Down*...

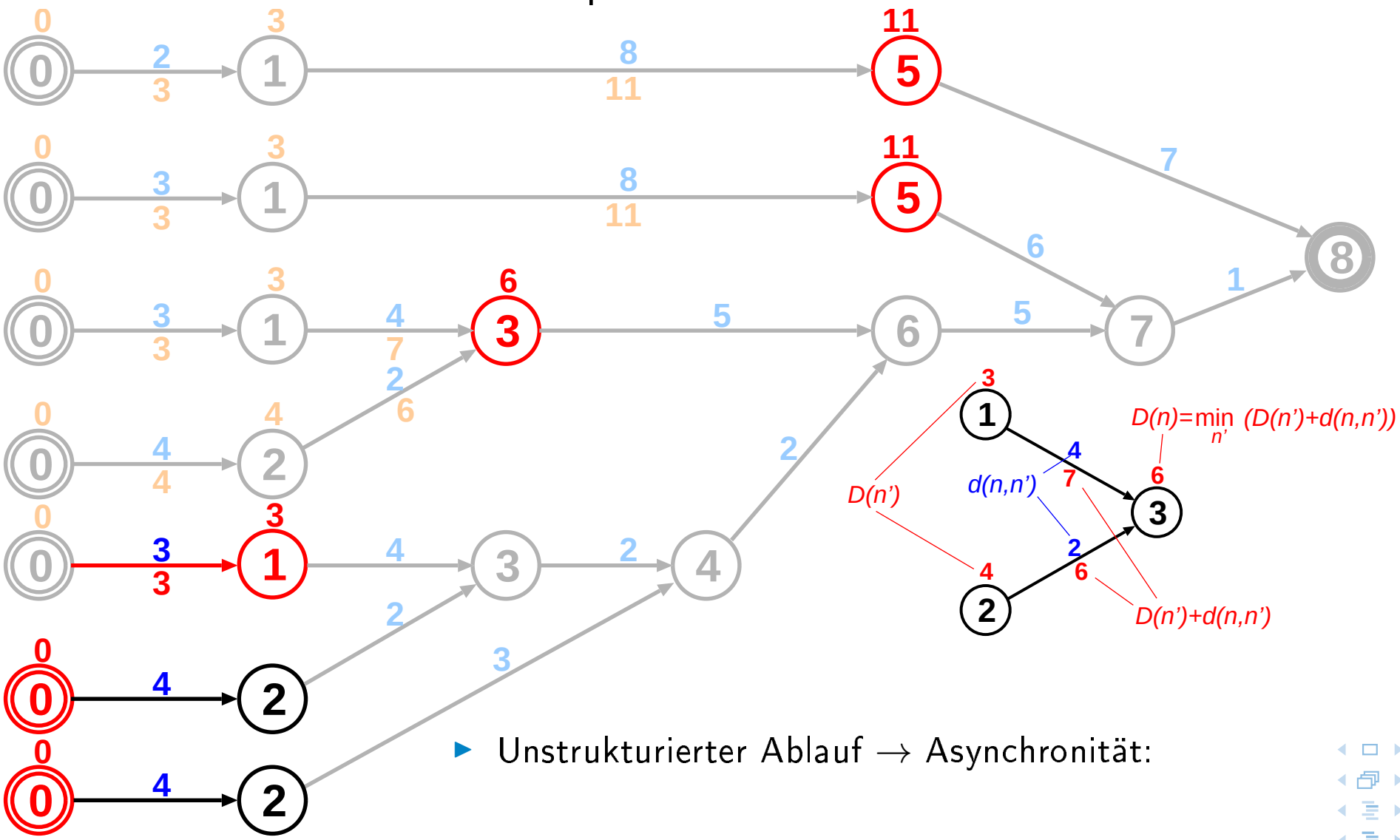
...oder wie man DP nicht implementieren sollte




- Konsequenz der Rekursion: sukzessive Aufspaltung des Graphen
- führt zu Baumstruktur

# Beispiel: Teilen und Herrschen - *Top-Down*...

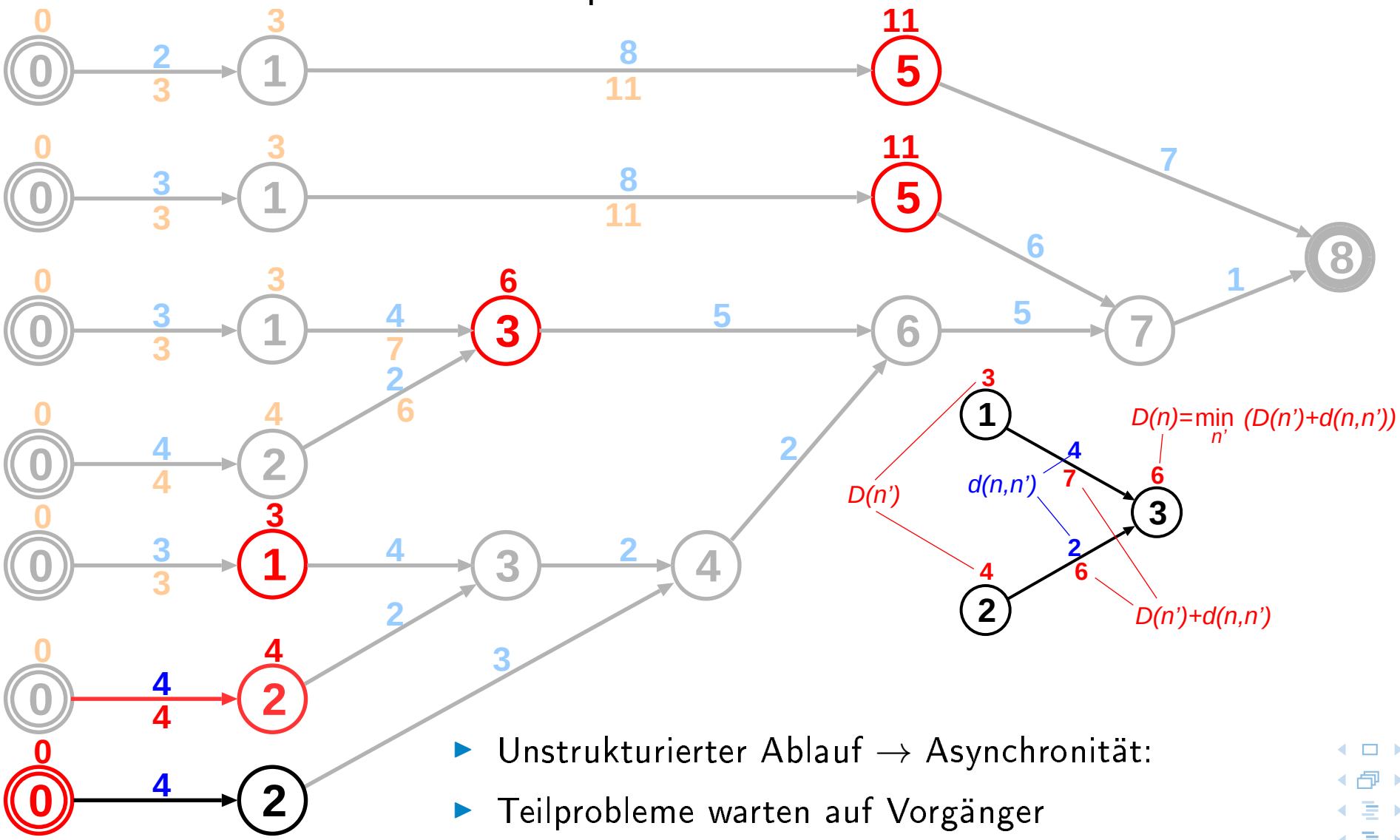
...oder wie man DP nicht implementieren sollte




- Unstrukturierter Ablauf → Asynchronität:

# Beispiel: Teilen und Herrschen - *Top-Down*...

...oder wie man DP nicht implementieren sollte



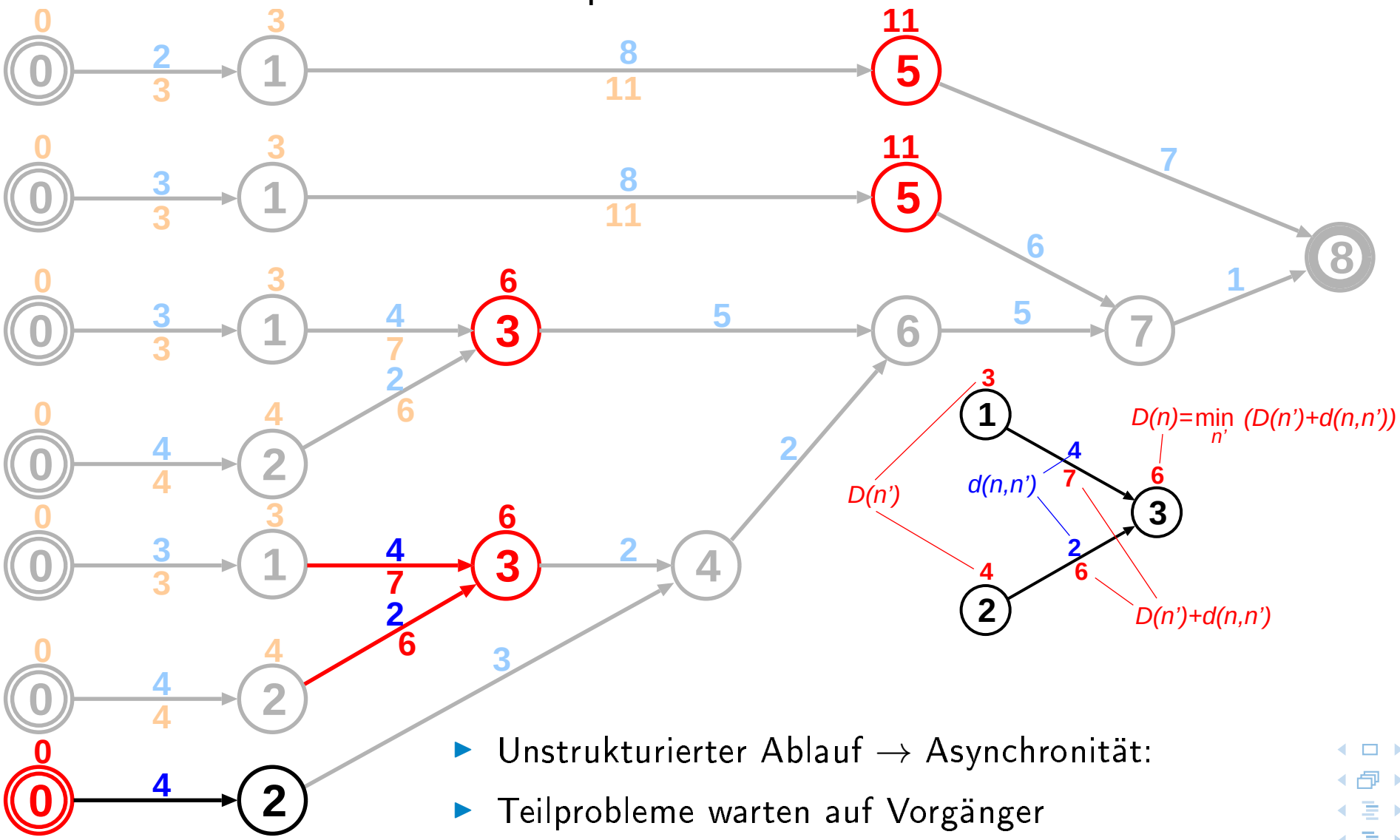




- Unstrukturierter Ablauf → Asynchronität:
- Teilprobleme warten auf Vorgänger

# Beispiel: Teilen und Herrschen - *Top-Down*...

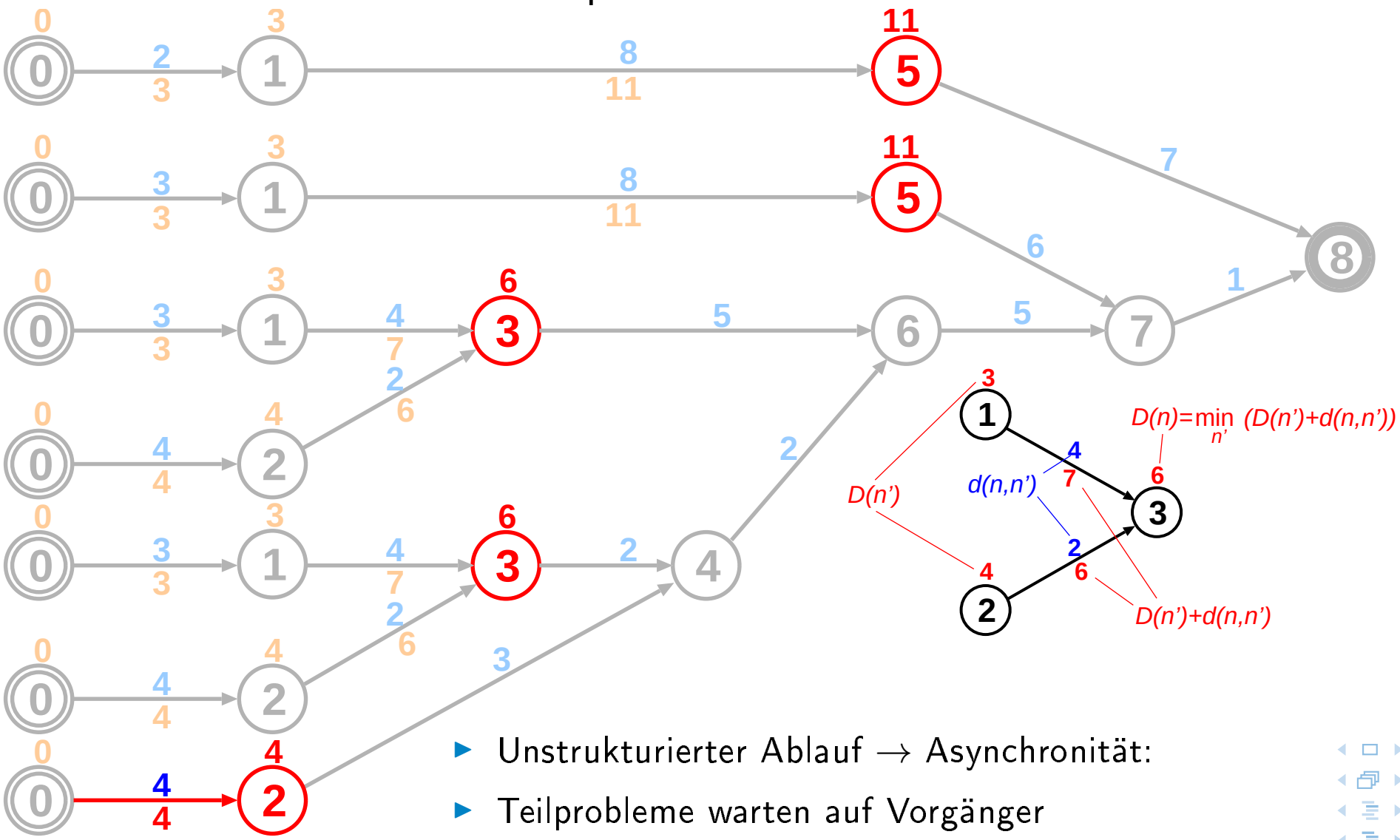
...oder wie man DP nicht implementieren sollte




- Unstrukturierter Ablauf → Asynchronität:
- Teilprobleme warten auf Vorgänger

# Beispiel: Teilen und Herrschen - *Top-Down*...

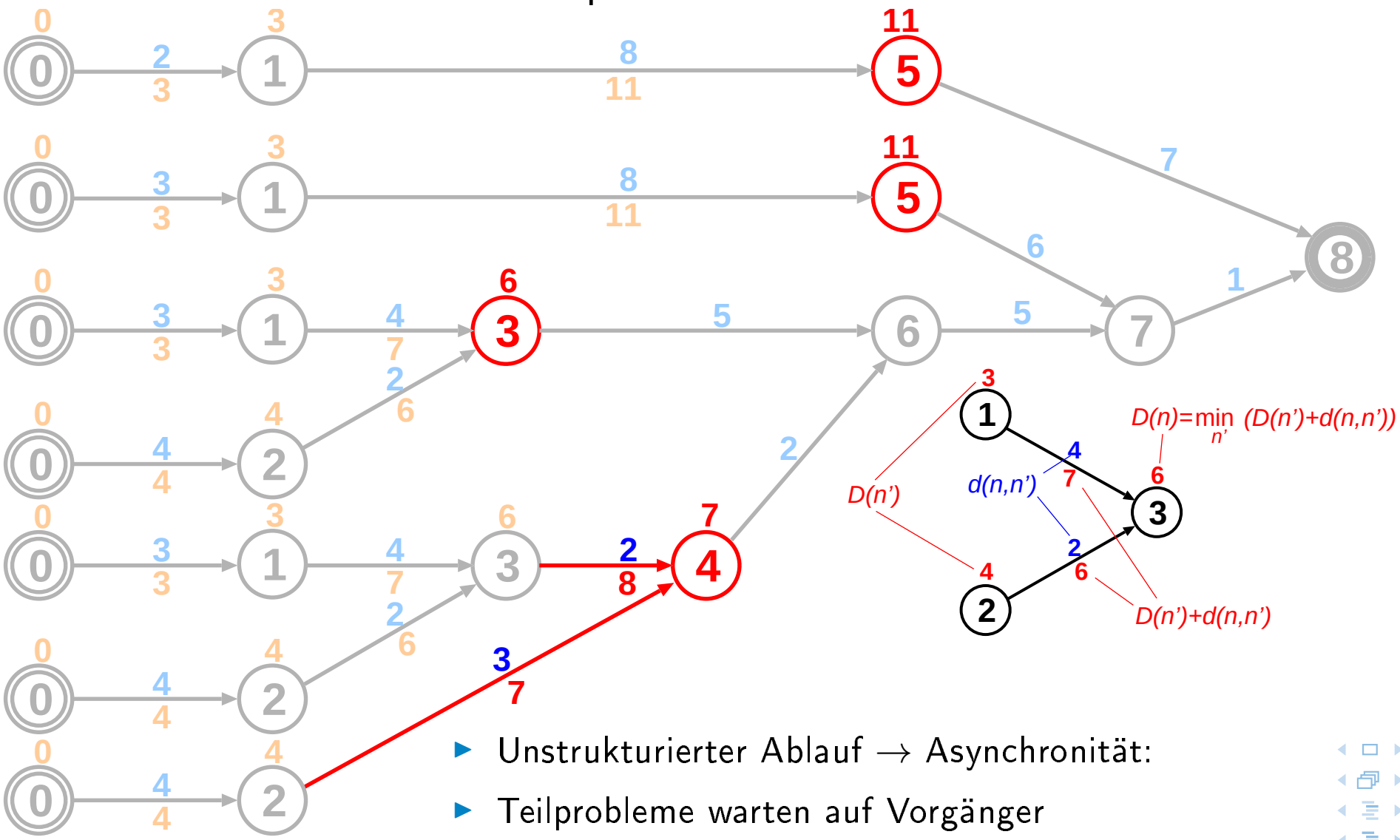
...oder wie man DP nicht implementieren sollte




- Unstrukturierter Ablauf → Asynchronität:
- Teilprobleme warten auf Vorgänger

# Beispiel: Teilen und Herrschen - *Top-Down*...

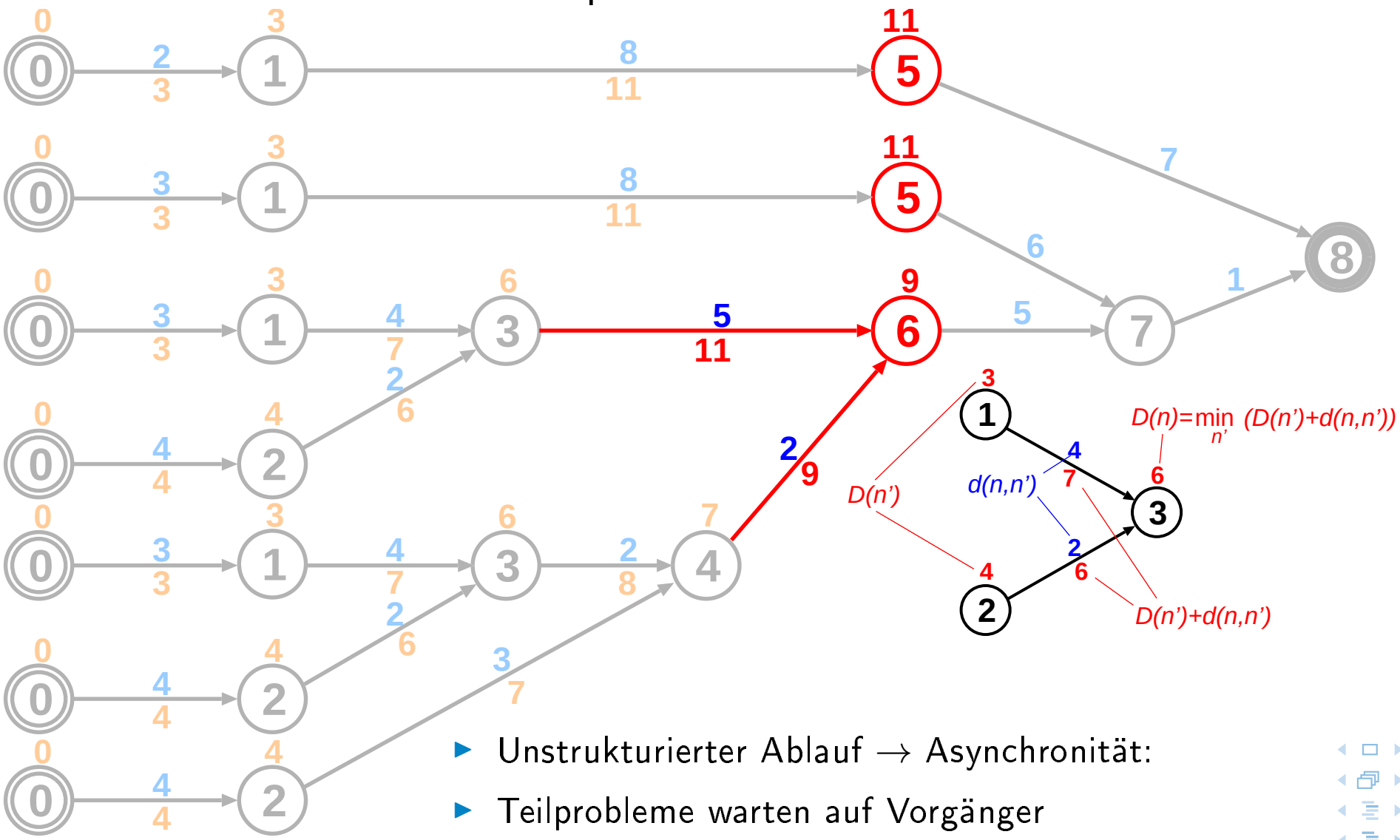
...oder wie man DP nicht implementieren sollte




- Unstrukturierter Ablauf → Asynchronität:
- Teilprobleme warten auf Vorgänger

# Beispiel: Teilen und Herrschen - *Top-Down*...

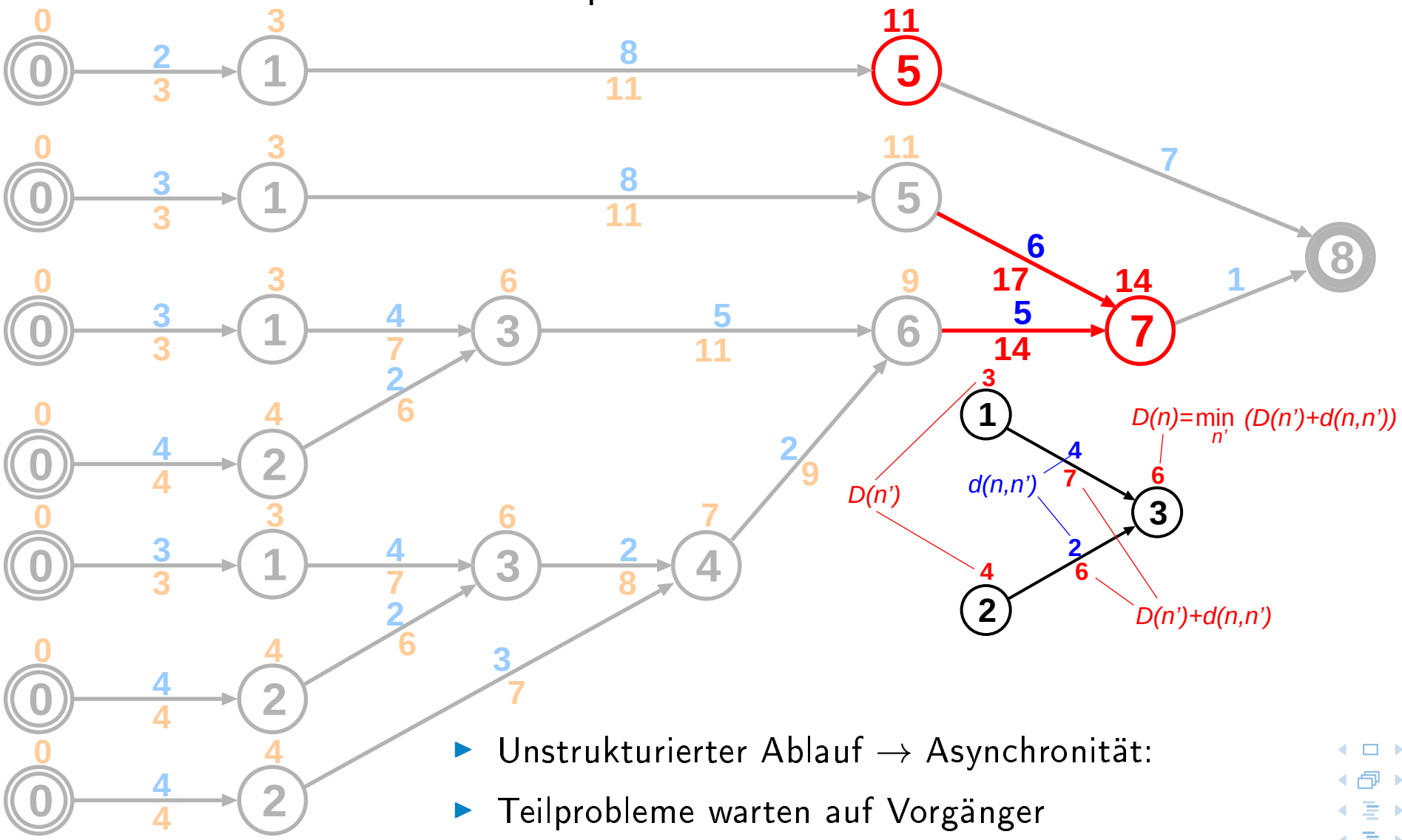
...oder wie man DP nicht implementieren sollte




- Unstrukturierter Ablauf → Asynchronität:
- Teilprobleme warten auf Vorgänger

# Beispiel: Teilen und Herrschen - *Top-Down*...

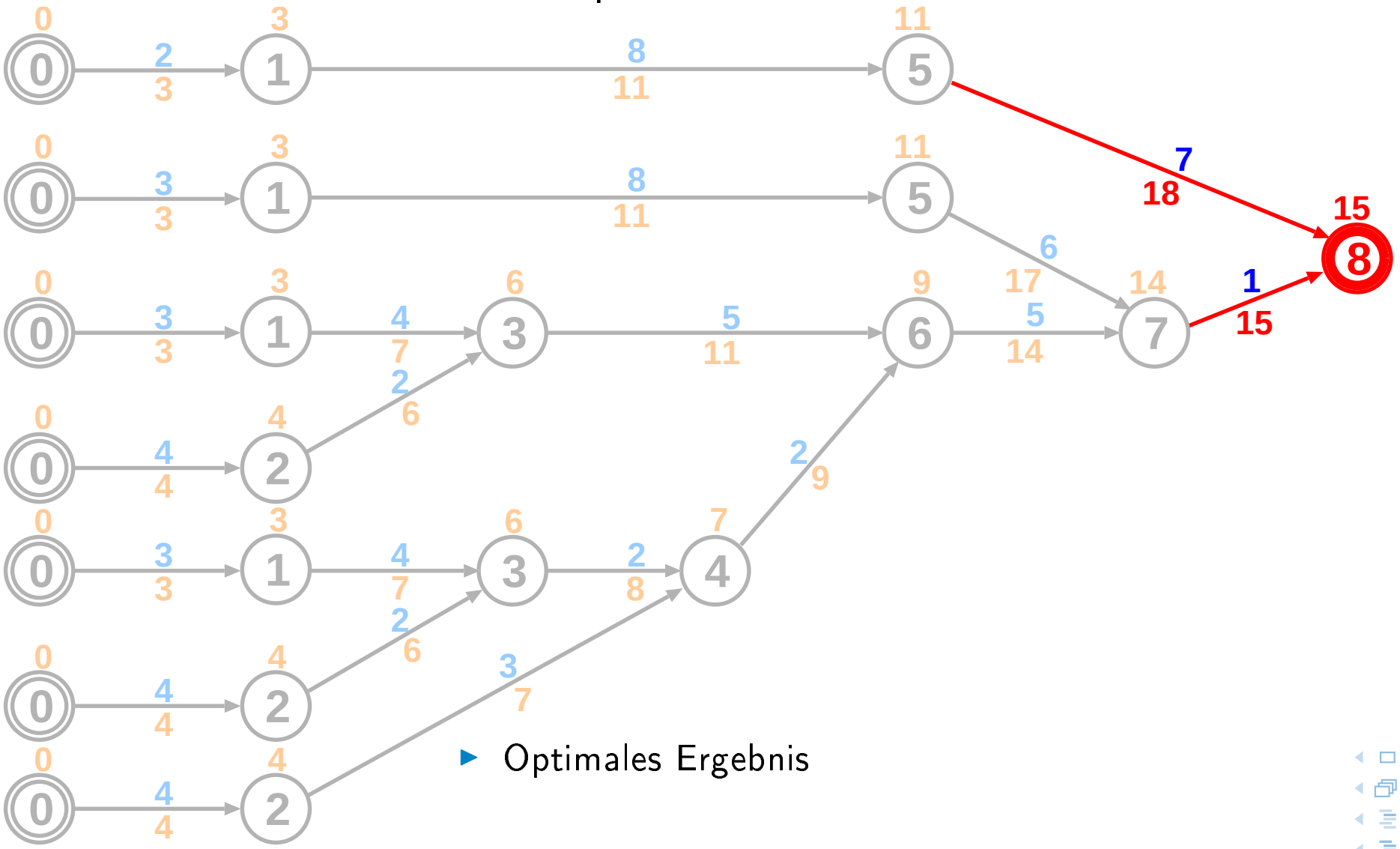
...oder wie man DP nicht implementieren sollte




- Unstrukturierter Ablauf → Asynchronität:
- Teilprobleme warten auf Vorgänger

# Beispiel: Teilen und Herrschen - *Top-Down*...

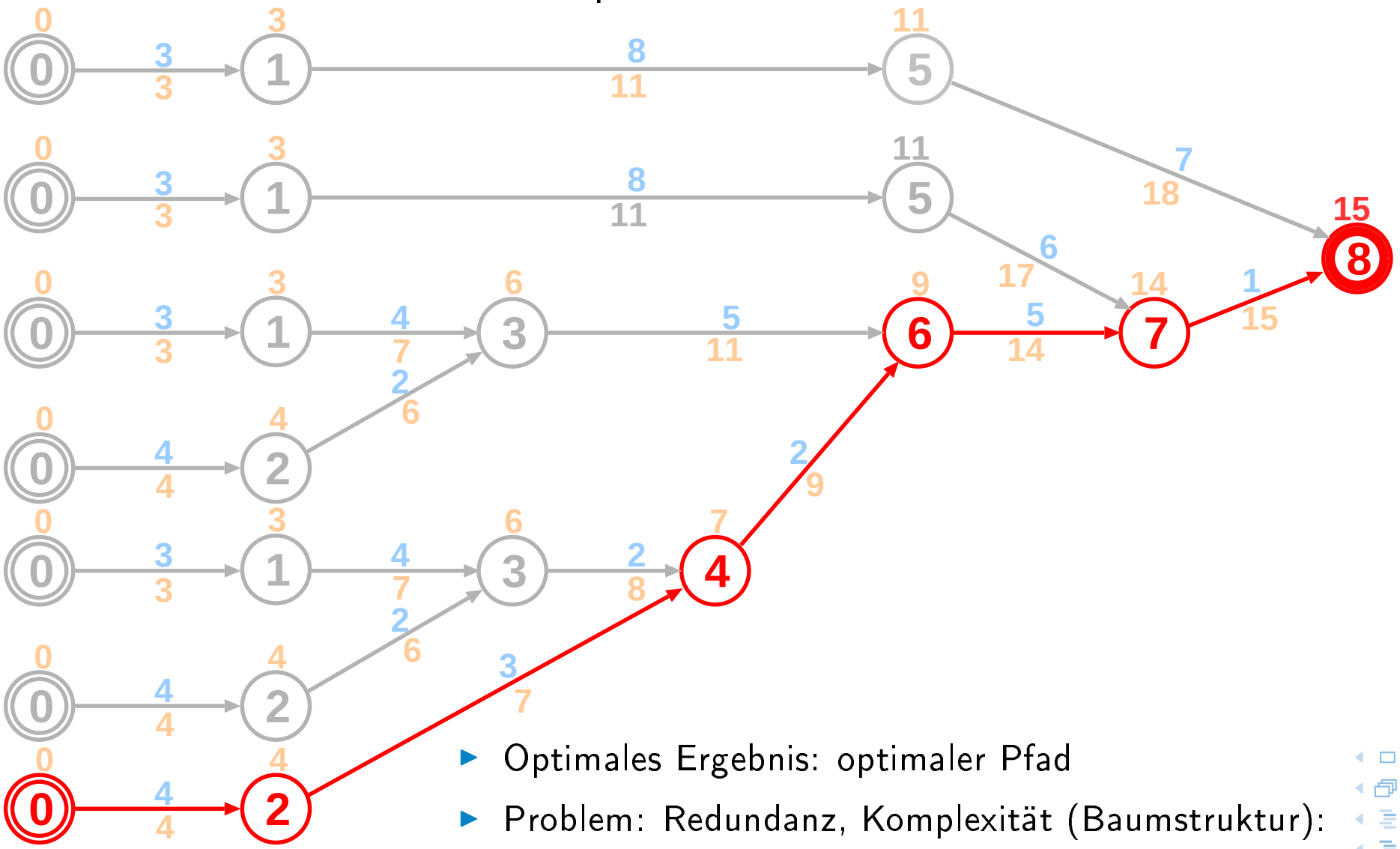
...oder wie man DP nicht implementieren sollte




- Optimales Ergebnis

# Beispiel: Teilen und Herrschen - *Top-Down*...

...oder wie man DP nicht implementieren sollte




- Optimales Ergebnis: optimaler Pfad
- Problem: Redundanz, Komplexität (Baumstruktur):
  # Knoten × mittl. # einlaufende Kanten: 21 Schritte

# Dynamische Programmierung

Lösung der dynamischen Programmierung:

- Rekursiv: naive Implementierung $\rightarrow$ nicht zeiteffizient
- Rekursiv: Memorieren, Tabellierung von Zwischenergebnissen
  $\rightarrow$ Zeiteffizienz: Vergewissern Sie sich
- Iterativ: Tabellierung und kontrollierter Ablauf $\rightarrow$ effizient

# Anwendung DP auf Editierdistanz

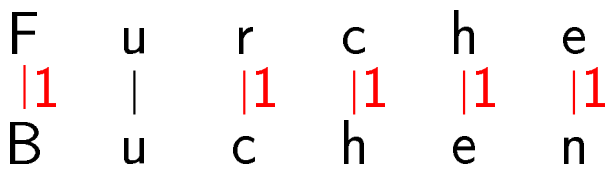
Approximatives *String-Matching*

- Motivation:
  - Fehlerratenberechnung in der automatischen Spracherkennung
  - Tippfehler-tolerantes *String-Matching*
  - DNA-Sequenzierung
- Editierdistanz (Levenshtein Distanz):
  - Minimale Anzahl an elementaren Operationen, um eine Symbolfolge in eine andere Symbolfolge umzuwandeln
  - Elementare Editieroperationen:
    - Streichungen
    - Einfügungen
    - Ersetzungen
  - Kosten pro elementarer Editieroperation: 1
- Randbedingungen:
  - Kontinuität: kein Überspringen von Symbolen
    - Zuordnung allein über Editieroperationen
  - Monotonie der Symbolsequenzen

# Editierdistanz: Beispiel und Definition

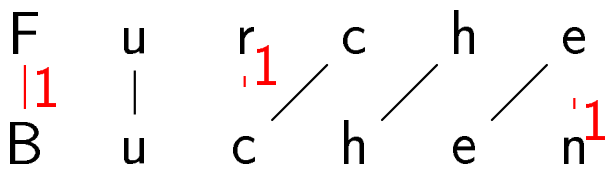
Beispiel: Vergleich der Wörter “Buchen” und “Furche”

Lineare Zuordnung:




- Anzahl Editieroperationen: 5

Optimale Zuordnung:




- Anzahl Editieroperationen: 3

Ziel:

- Zuordnung mit minimaler Anzahl Editieroperationen

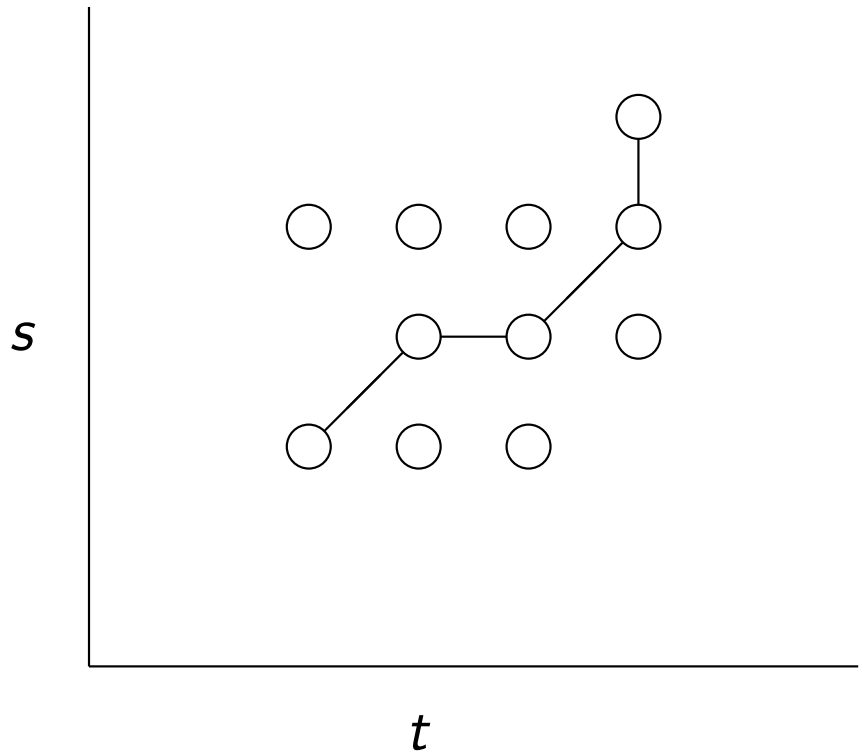
Streichungen, Einfügungen und Ersetzungen sollen gleiche Kosten haben:

$$\begin{aligned} \text{Kosten} &= \text{Anzahl Editieroperationen} \\ &= \text{Streichungen} + \text{Einfügungen} + \text{Ersetzungen} \end{aligned}$$

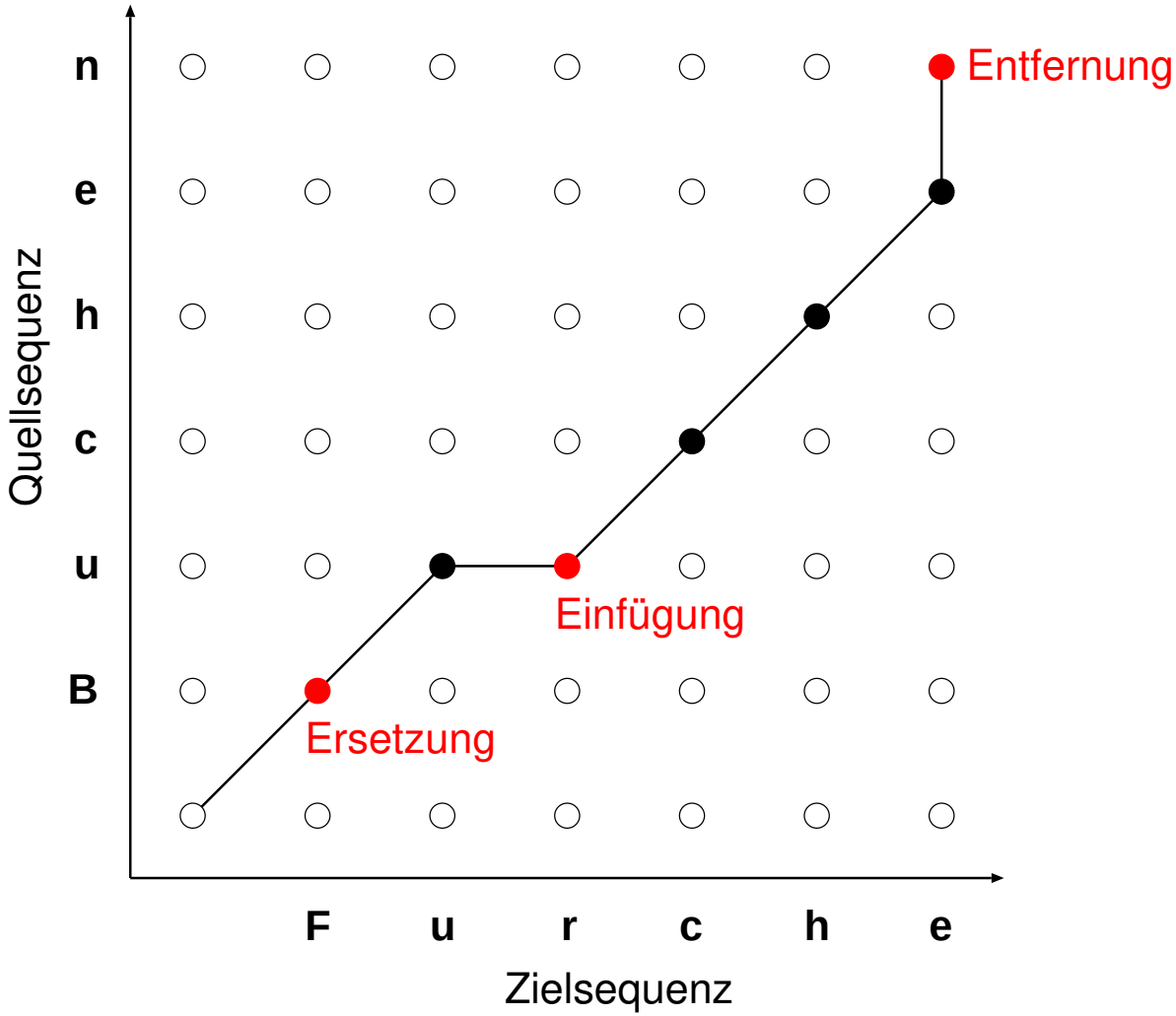
# Editierdistanz: Formalisierung

- Gegeben:
  Quellsymbolfolge: $y_1 \ldots y_s \ldots y_S$ (zu editieren)
  Zielsymbolfolge: $x_1 \ldots x_t \ldots x_T$ (zu erzielen)

- Diagramm zur Darstellung unterschiedlicher Anpassungen:



# Editierdistanz: Beispiel



# Editierdistanz: DP-Hilfsfunktion

- Definition lokaler Kostenfunktion:

$$d(t,s,t',s') = \begin{cases} 1 & \text{falls } t = t'+1 \text{ und } s = s' \quad \text{(Einfügung)} \\ 1 & \text{falls } t = t' \text{ und } s = s'+1 \quad \text{(Entfernung)} \\ 1-\delta_{x_t,y_s} & \text{falls } t = t' \text{ und } s = s' \quad \text{(Ersetzung/korrekt)} \\ \infty & \textit{sonst} \end{cases}$$

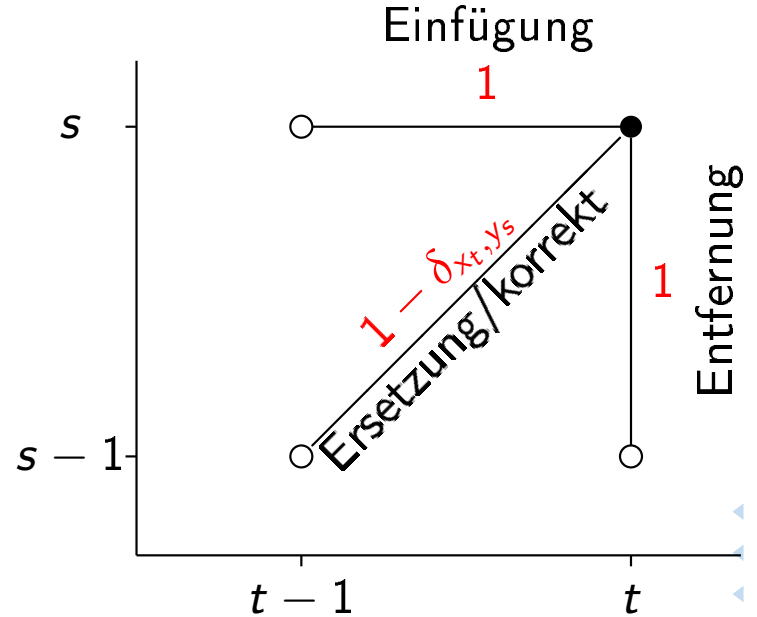
mit der *Kronecker*-Deltafunktion $\delta_{x,y} = \begin{cases} 1 & x = y \\ 0 & \textit{sonst} \end{cases}$

- Definition einer Hilfsfunktion:
  $D(t,s)$: Editierdistanz zw. $x_1^t$ und $y_1^s$

$$D(t,s) := \min_{\substack{N,t_0^N,s_0^N: \\ t_N=t \\ s_N=s}} \sum_{n=1}^{N} d(t_n,s_n,t_{n-1},s_{n-1})$$

Randbedingungen: $t_0 = 0$, $s_0 = 0$
(im Folgenden nicht genannt)

- Gesamtlösung ergibt sich
  für t=T und s=S: $D(T,S)$



Erlaubte Schritte bei der Zuordnung

# Editierdistanz: Herleitung DP-Rekursion

$$
\begin{aligned}
D(t,s) &= \min_{\substack{N, t_0^N, s_0^N: \\ t_N = t \\ s_N = s}} \sum_{n=1}^{N} d(t_n, s_n, t_{n-1}, s_{n-1}) \\
&= \min_{\substack{N, t_0^N, s_0^N: \\ \cancel{t_N = t} \\ \cancel{s_N = s}}} \Bigg( \underbrace{\sum_{n=1}^{N-1} d(t_n, s_n, t_{n-1}, s_{n-1})}_{\text{keine Abhängigkeit von } t_N, s_N} + d(\underbrace{t_N}_{=t}, \underbrace{s_N}_{=s}, \underbrace{t_{N-1}}_{=:\tau}, \underbrace{s_{N-1}}_{=:\sigma}) \Bigg) \\
&\qquad \Big| \text{ Substitutionen: } N' = N-1,\ \tau = t_{N-1} = t_{N'},\ \sigma = s_{N-1} = s_{N'} \\
&= \min_{\tau,\sigma} \Bigg( \underbrace{\min_{\substack{N', t_0^{N'}, s_0^{N'}: \\ t_{N'} = \tau \\ s_{N'} = \sigma}} \sum_{n=1}^{N'} d(t_n, s_n, t_{n-1}, s_{n-1})}_{= D(\tau,\sigma)} + d(t,s,\tau,\sigma) \Bigg) \\
&= \min_{\tau,\sigma} \big( D(\tau,\sigma) + d(t,s,\tau,\sigma) \big) \\
&= \min_{\delta_t, \delta_s \in \Delta} \big( D(t-\delta_t, s-\delta_s) + d(t,s,t-\delta_t,s-\delta_s) \big) \\
&\qquad \Big| \text{ mit der Menge an möglichen Schritten } \Delta = \{(0,1),(1,0),(1,1)\} \\
&= \min \big\{ D(t-1,s)+1,\ D(t,s-1)+1,\ D(t-1,s-1)+1-\delta_{x_t,y_s} \big\}
\end{aligned}
$$

Herleitung Rekursion Dynamische Programmierung Editierdistanz:

Hilfsfunktion: Editierdistanz zwischen Teilfolgen $x_1^t$ und $y_1^s$

$$D(t,s) := \min_{N, t_0^N, s_0^N: \; t_N = t, s_N = s} \sum_{n=1}^{N} d(t_n, s_n, t_{n-1}, s_{n-1})$$

| separiere letzten Summanden für n=N:

$$:= \min_{N, t_0^N, s_0^N: \; t_N = t, s_N = s} \left( \sum_{n=1}^{N-1} d(t_n, s_n, t_{n-1}, s_{n-1}) + d(t_N, s_N, t_{N-1}, s_{N-1}) \right)$$

$t_N = t$, $s_N = s$, $t_{N-1} = \tau$, $s_{N-1} = \sigma$

Substitution: $N' := N-1$, $t_{N-1} = \tau = t_{N'}$, $s_{N-1} = \sigma = s_{N'}$

$$= \min_{\tau, \sigma} \left( \min_{N', t_0^{N'}, s_0^{N'}: \; t_{N'} = \tau, s_{N'} = \sigma} \sum_{n=1}^{N'} d(t_n, s_n, t_{n-1}, s_{n-1}) + d(t, s, \tau, \sigma) \right)$$

$$= D(\tau, \sigma)$$

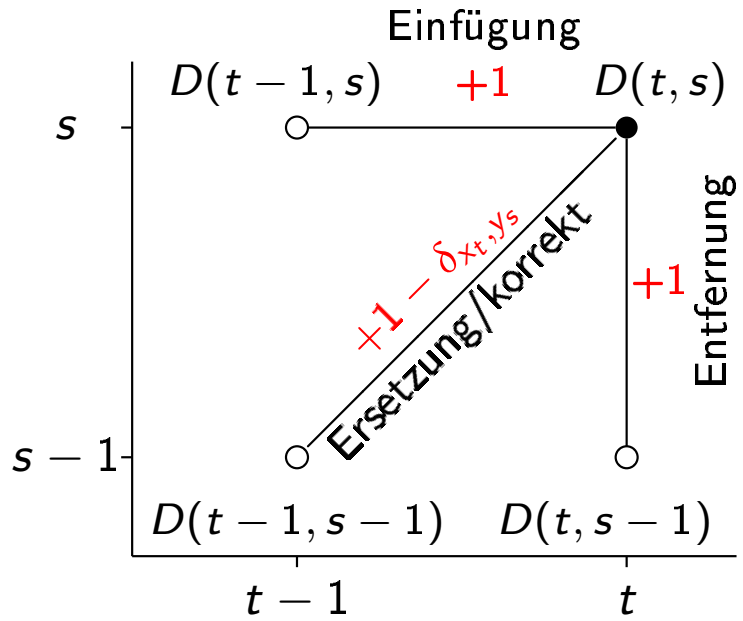
$$= \min_{\tau, \sigma} \left( D(\tau, \sigma) + d(t, s, \tau, \sigma) \right)$$

# Editierdistanz: Durchführung Dynamische Programmierung

- Rekursion:

$$\begin{aligned} D(t,s) &= \min\{D(t-1,s)+1, D(t,s-1)+1, \\ &\qquad D(t-1,s-1)+(1-\delta_{x_t,y_s})\} \\ &= 1+\min\{D(t-1,s-1)-\delta_{x_t,y_s}, \\ &\qquad D(t-1,s), D(t,s-1)\} \end{aligned}$$

für $0 < t \leq T,\ 0 < s \leq S$ mit $\delta_{x,y} = \begin{cases} 0 & x \neq y \\ 1 & x = y \end{cases}$




- Initialisierung:
  - $D(0,0) = 0$ (Startpunkt vor Beginn beider Symbolfolgen)
  - $D(0,s) = s$ mit $s \in [1,S]$ (anfängliche Folge von Streichungen)
  - $D(t,0) = t$ mit $t \in [1,T]$ (anfängliche Folge von Einfügungen)

# Editierdistanz: DP-Algorithmus

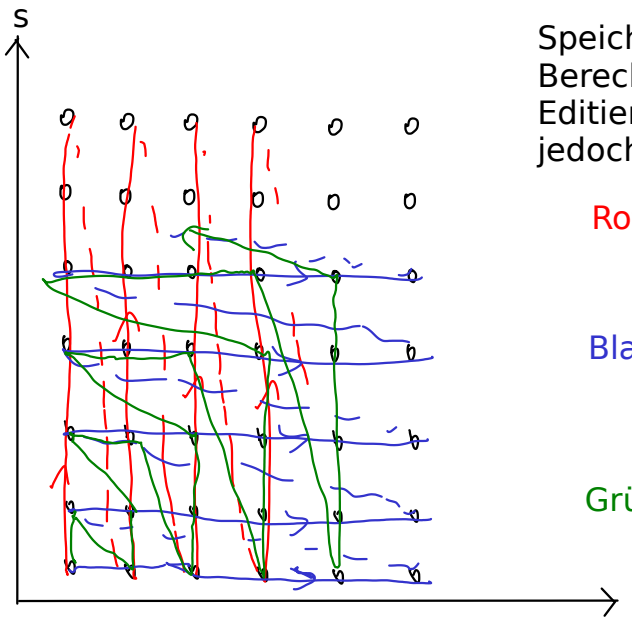
**Array** D(1:T, 1:S)
**for** $s = 0, .., S$
  D(0,s)=s
**for** $t = 1, .., T$
  D(t,0)=t
  **for** $s = 1, .., S$
    $D(t,s) = \min_{(\delta_t,\delta_s)\in\Delta} \left(D(t-\delta_t, s-\delta_s) + d(t,s,t-\delta_t,s-\delta_s)\right)$

- Komplexität: $T \cdot S$ (Zeit & Speicher)
- Speicherkomplexität ohne Bestimmung der Zuordnung: $\min\{T, S\}$
- *Backpointer* speichert lokale Entscheidungen der Rekursion:

$$B(t,s) = \arg \min_{(\delta_t,\delta_s)\in\Delta} \left(D(t-\delta_t, s-\delta_s) + d(t,s,t-\delta_t,s-\delta_s)\right)$$

  - erlaubt Zurückverfolgung der optimalen Editierfolge

## Reihenfolge der Abarbeitung und Speicherbedarf bei der Berechnung der Editierdistanz:




Speicherbedarf für verschiedene Abfolgen der Berechnung der Hilfsfunktion, wenn nur die Editierdistanz berechnet werden soll, nicht jedoch die optimale Zuordnung:

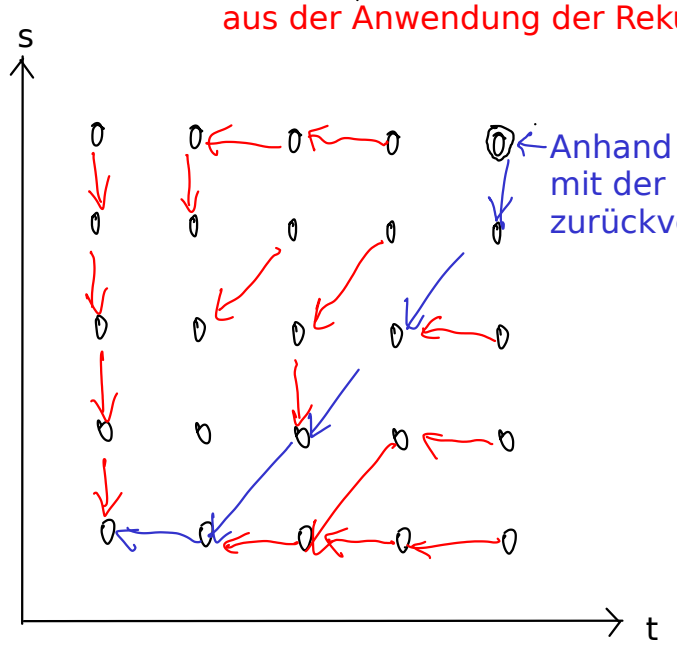
Rot: Iteration synchron in t: Speicherbedarf O(S), da für die Berechnung einer Spalte allein die direkt vorhergehende Spalte benötigt wird

Blau: Iteration synchron in s: Speicherbedarf O(T), da für die Berechnung einer Zeile allein die direkt vorhergehende Zeile benötigt wird

Grün: Iteration gemischt: Speicherbedarf O(T+S)

## Illustration zur Verwendung von Backpointern:

Jeder Gitterpunkt (t,s) hat einen eindeutigen besten Vorgänger aus der Anwendung der Rekursionsgleich der dynamischen Programmierung.




Anhand der Backpointer lässt sich die Zuordnung mit der minimalen Anzahl an Editoperationen zurückverfolgen.

# Dynamische Programmierung: Anwendungen

Beispiele für die Anwendung dynamischer Programmierung:

- Triangulation von Polygonen
- Matrizen-Kettenmultiplikation
- Cocke-Younger-Kasami Parsing:
  - Wortproblem bei kontextfreien Sprachen
- Rucksackproblem
- Traveling Salesman Problem:
  - DP-Strukturierung: Kardinalitäten der Mengen schon besuchter Orte
  - Hilfsfunktion: Funktion der Menge schon besuchter Orte und des letzten erreichten Ortes
  - Komplexität kombinatorisch $\mathcal{O}(N!) \rightarrow$ DP: $\mathcal{O}(N^2 2^N)$

# Dynamische Programmierung: Zusammenfassung

- Konzept zur Algorithmenentwicklung:
- Optimierungsprobleme
- Sukzessiv zerlegbar in Teilprobleme
- Ausnutzung/Vermeidung von Redundanz
- Strukturiertes Vorgehen

Quelle: U. Schöning: *Algorithmik*, Spektrum Akademischer Verlag, Heidelberg - Berlin, 2001.