Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

Universität zu Lübeck

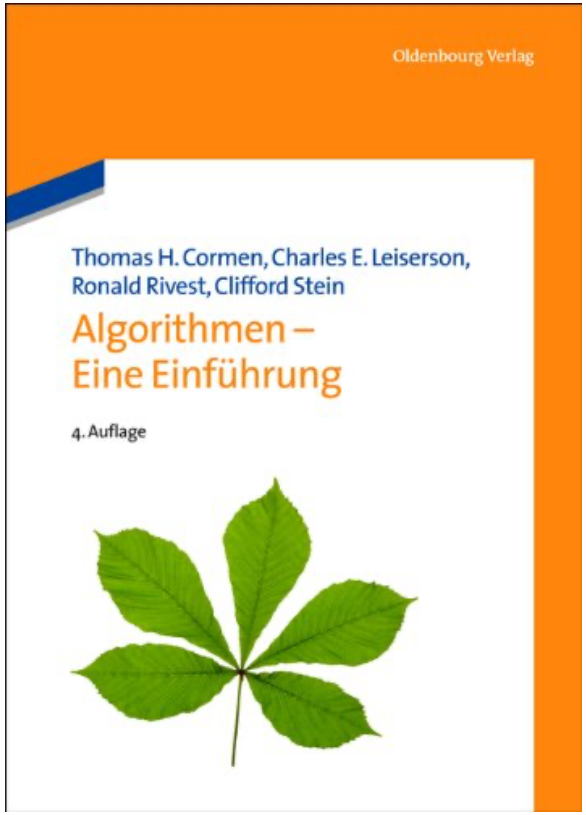
Institut für Informationssysteme

Felix Kuhr (Übungen)

sowie viele Tutoren

Danksagung

- Animationen wurden übernommen aus dem Kurs: CS 3343 Analysis of Algorithms von Jianhua Ruan
- Inhalte sind angelehnt an



Algorithmen-Entwurfsmuster

Beispielproblemklasse: Optimierungsprobleme

- Beladungsprobleme
- Anordnungsprobleme
- Planungsprobleme

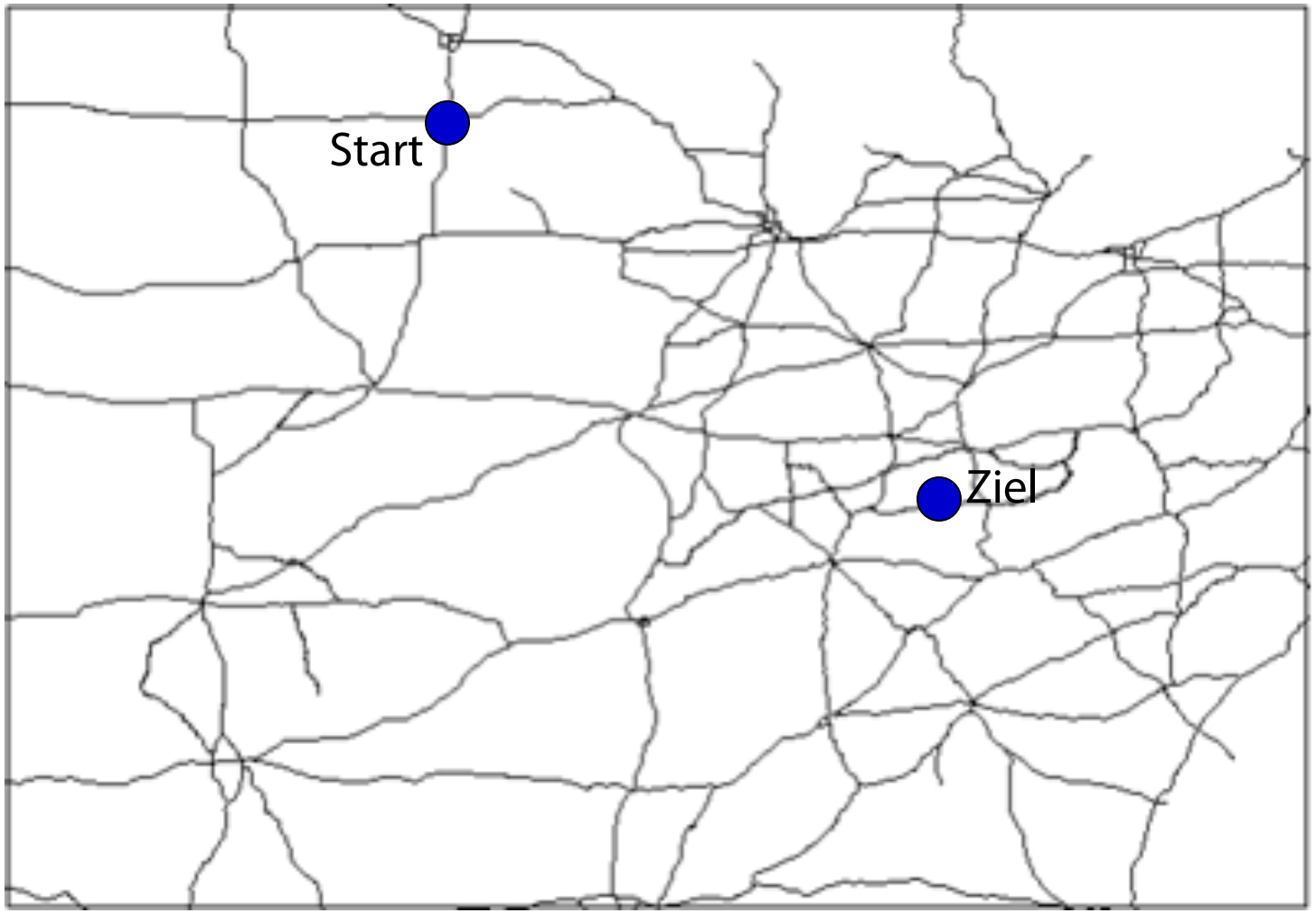
Naiver Ansatz (Brute-Force) ist fast immer kombinatorisch oder die optimale Lösung wird nicht gefunden (Unvollständigkeit)

Entwurfsziel: Vermeidung von Kombinatorik
unter Beibehaltung der Vollständigkeit

Überblick

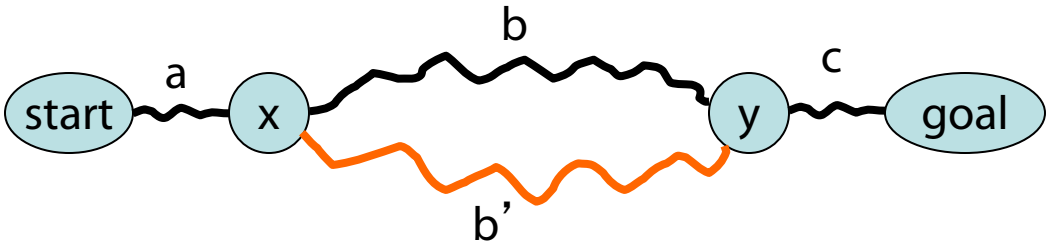
- Dynamische Programmierung
  - Name historisch begründet (sollte gut klingen, kein Bezug zum heutigen Begriff der Programmierung)
  - Bellmans Optimalitätsprinzip
  - Fragestellung: Wie können Problemlösungen aus Lösungen für Teilprobleme hergeleitet werden, so dass Vollständigkeit erreicht wird
- Gierige Algorithmen (Greedy Algorithms)
  - Verfolgung nur des augenscheinlich aktuell günstigsten Wegs zum Ziel
  - Fragestellung: Wann sind gierige Algorithmen vollständig?

Kürzeste Wege als Optimierungsproblem betrachtet



Bellmans Optimalitätsprinzip

Behauptung: Falls ein Weg start⟶goal optimal ist, muss jeder Teilweg x⟶y, der auf dem optimalen Pfad liegt, optimal sein



a + b + c ist kürzester Weg

b' < b

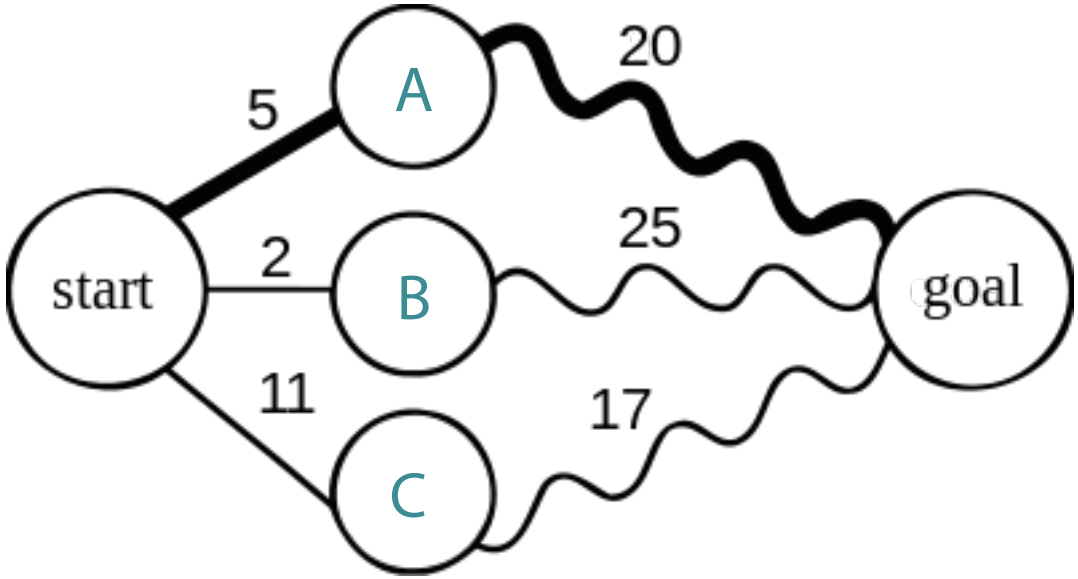
a + b' + c < a + b + c

Beweis durch Widerspruch:

- Falls Teilweg b zwischen x und y nicht kürzester Weg, können wir ihn durch kürzeren Weg b' ersetzen
- Dadurch wird der Gesamtweg kürzer und der betrachtete Weg start⟶goal ist nicht optimal: Widerspruch
- Also muss b der kürzeste Weg von x nach y sein

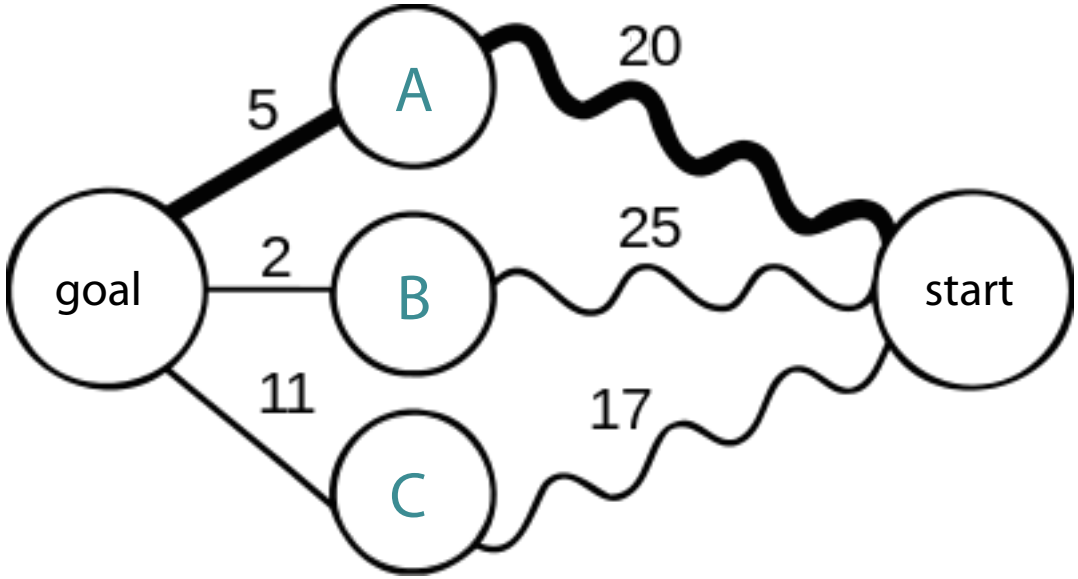
R. Bellman, *On the Theory of Dynamic Programming*, Proceedings of the National Academy of Sciences, **1952**

Problem des kürzesten Pfads



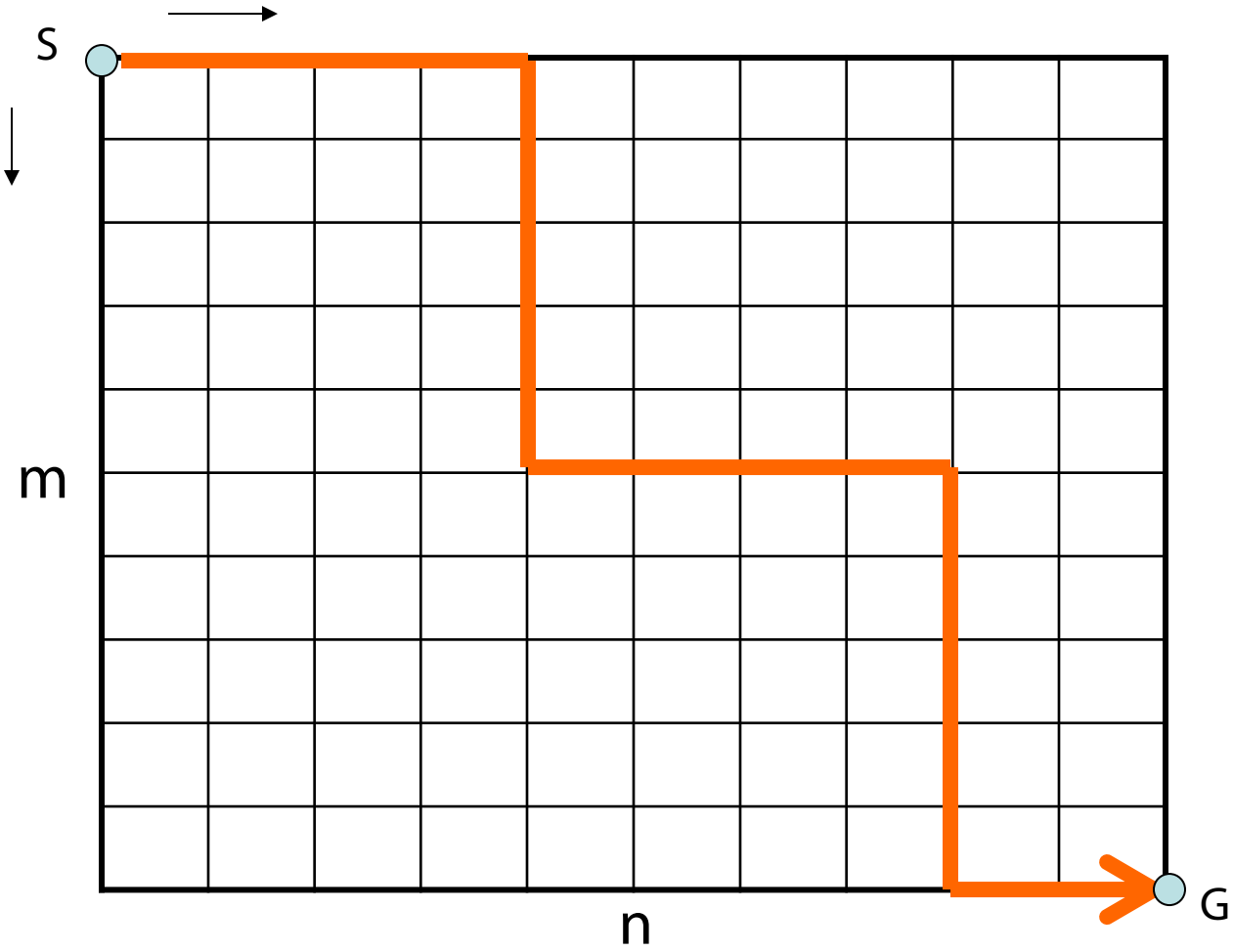
$$SP(start, goal) = \min \begin{cases} dist(start, A) + SP(A, goal) \\ dist(start, B) + SP(B, goal) \\ dist(start, C) + SP(C, goal) \end{cases}$$

Problem des kürzesten Pfads



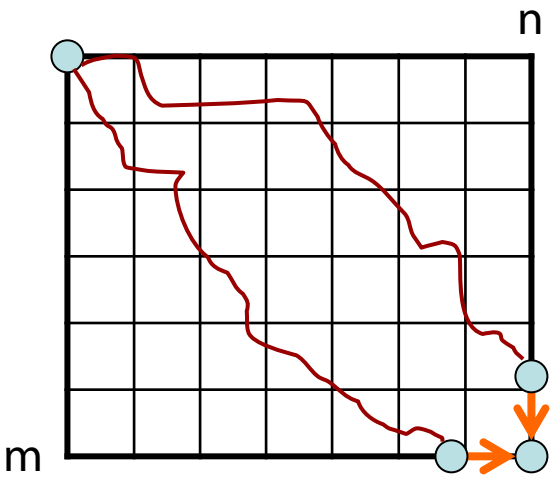
$$SP(start, goal) = \min \begin{cases} dist(start, A) + SP(A, goal) \\ dist(start, B) + SP(B, goal) \\ dist(start, C) + SP(C, goal) \end{cases}$$

Ein spezielles Szenario für den kürzesten Pfad



Jeder Kante sind (verschiedene) Kosten zugeordnet. Schritte nur nach rechts oder nach unten möglich. Ziel: Kürzester Weg von S nach G.

Betrachtung eines Lösungsansatzes

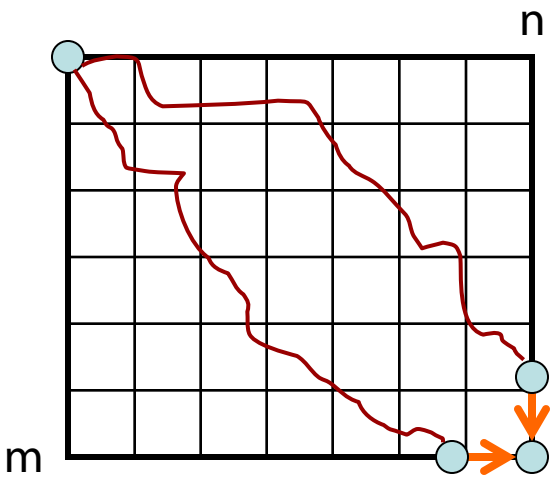


Annahme: Kürzester Pfad gefunden

- Folgende Kanten kommen vor
  - (m, n-1) nach (m, n) Fall 1
  - (m-1, n) nach (m, n) Fall 2
- Fall 1:
  - Suche kürzesten Pfad von (0, 0) nach (m, n-1)
  - Length(SP(0, 0, m, n-1)) + dist(m, n-1, m, n) ist die kürzeste Pfadlänge
- Fall 2:
  - Suche kürzesten Pfad von (0, 0) nach (m-1, n)
  - Length(SP(0, 0, m-1, n)) + dist(m-1, n, m, n) ist die kürzeste Pfadlänge
- Wir wissen nicht, welcher Fall eintreten wird
  - Falls wir die zwei Pfade haben, können wir aber vergleichen
  - Dann lässt sich der kürzeste Pfad leicht bestimmen

Rekursiver Lösungsansatz

Sei F(i, j) = length(SP(0, 0, i, j)). => F(m, n) die Länge des SP von (0, 0) nach (m, n)

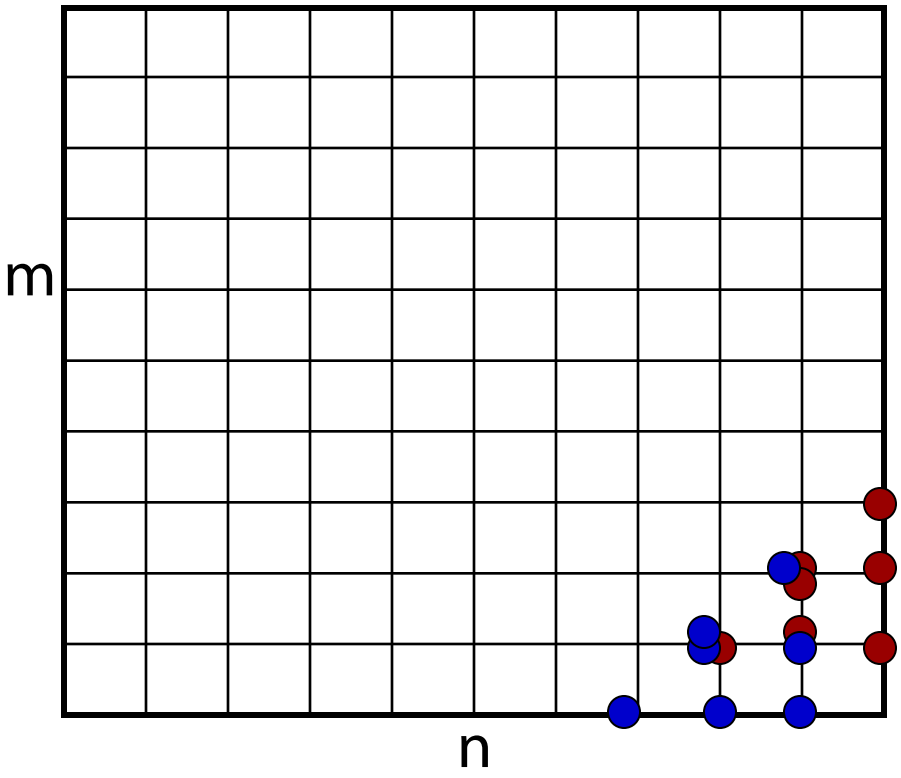


$$F(m, n) = \min \begin{cases} F(m-1, n) + dist(m-1, n, m, n) \\ F(m, n-1) + dist(m, n-1, m, n) \end{cases}$$

Generalisierung

$$F(i, j) = \min \begin{cases} F(i-1, j) + dist(i-1, j, i, j) \\ F(i, j-1) + dist(i, j-1, i, j) \end{cases}$$

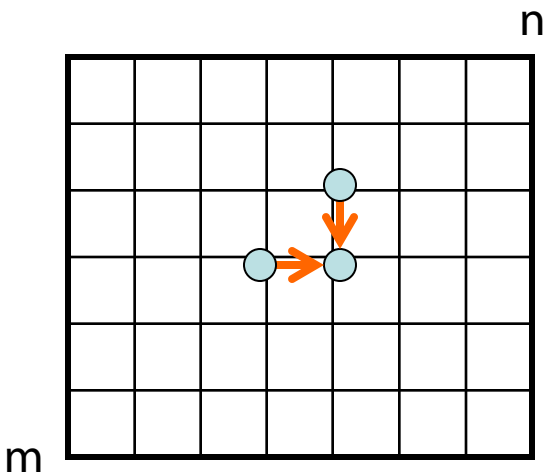
Vermeide Neuberechnungen



- Um den kürzesten Pfad von (0, 0) nach (m, n) zu finden, benötigen wir die kürzesten Pfade nach (m-1, n) und (m, n-1)
- Bei rekursivem Aufruf werden Lösungen immer wieder neu berechnet
- Strategie:
  Löse Teilprobleme in richtiger Reihenfolge, so dass redundante Berechnungen vermieden werden

Prinzip der Dynamischen Programmierung

Sei F(i, j) = SP(0, 0, i, j). => F(m, n) die Länge von SP von (0, 0) nach (m, n)

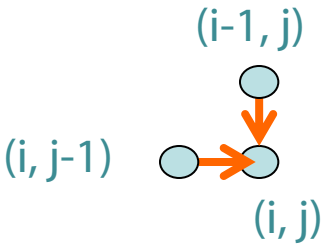


$$F(i, j) = \min \begin{cases} F(i-1, j) + dist(i-1, j, i, j) \\ F(i, j-1) + dist(i, j-1, i, j) \end{cases}$$

Grenzfall: i = 0 oder j = 0
Leicht zu identifizieren

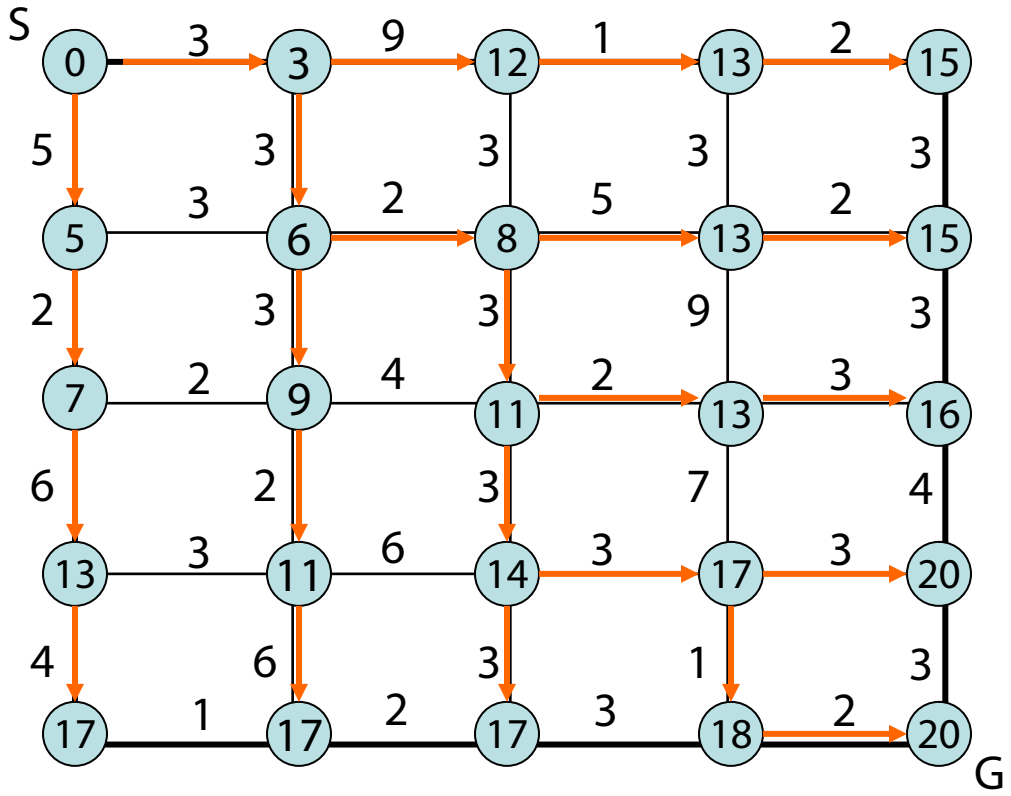
i = 1 .. m, j = 1 .. n

Anzahl der Unterprobleme: m · n



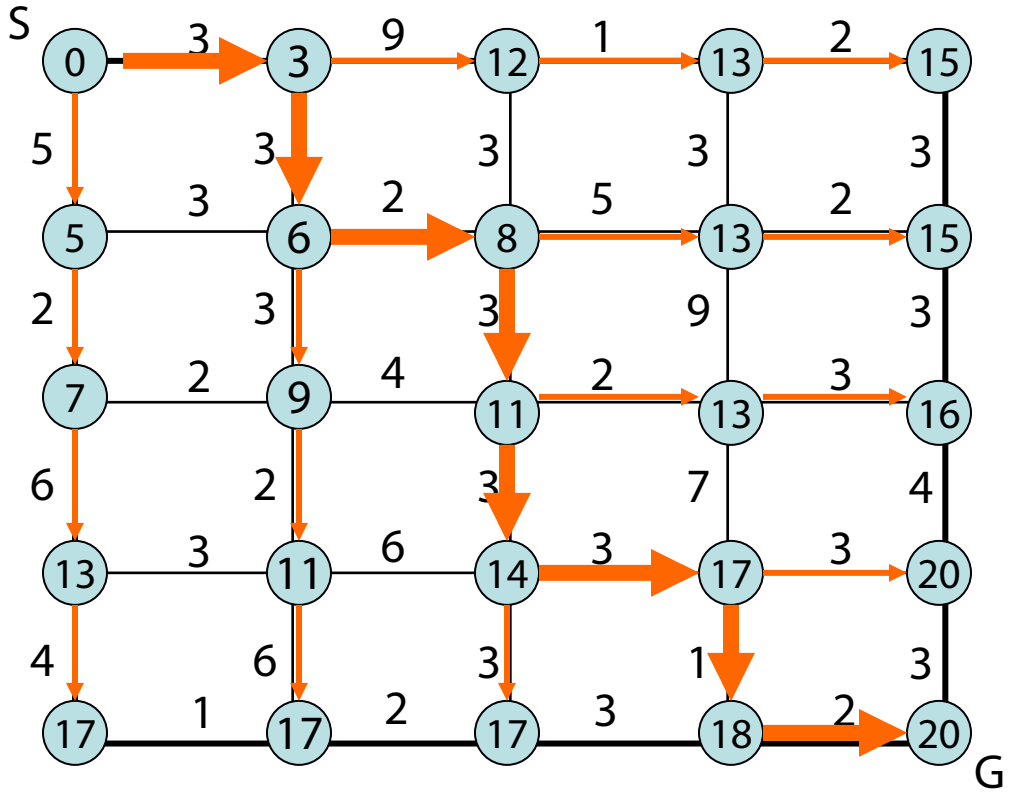
Datenabhängigkeit definiert Anordnung der Teilprobleme; hier von links nach rechts, von oben nach unten

Dynamische Programmierung: Illustration



$$F(i, j) = \min \begin{cases} F(i-1, j) + dist(i-1, j, i, j) \\ F(i, j-1) + dist(i, j-1, i, j) \end{cases}$$

Rückverfolgung zur Bestimmung der Lösung



Kürzester Pfad hat die Länge 20

Elemente der Dynamischen Programmierung

- Optimale Unterstrukturen
  - Optimale Lösungen des ursprünglichen Problems enthält optimale Lösungen von Teilproblemen
- Überlappung von Teilproblemen
  - Einige Teilprobleme kommen in vielen Lösungen vor
  - Formulierung von Lösungen als Rekurrenz auf Teillösungen
- Speicherung und Wiederverwendung
  - Wähle Ordnung der Teilprobleme sorgfältig aus, so dass jedes Teilproblem eines höheren Problems vorher gelöst wurde und die Lösung wiederverwendet werden kann

Überblick über betrachtete Probleme

- Sequenzausrichtungsprobleme (mehrere Varianten)
  - Vergleich zweier Zeichenketten auf Ähnlichkeit
    - Definition von Ähnlichkeit nötig
    - Nützlich in vielen Anwendungen
      - Vergleich von DNA-Sequenzen, Aufsätzen, Quellcode
  - Beispielproblem: Longest-Common-Subsequence (LCS)
- Anordnungsprobleme
- Planungsprobleme (Scheduling)
- Rucksackproblem (mehrere Varianten)