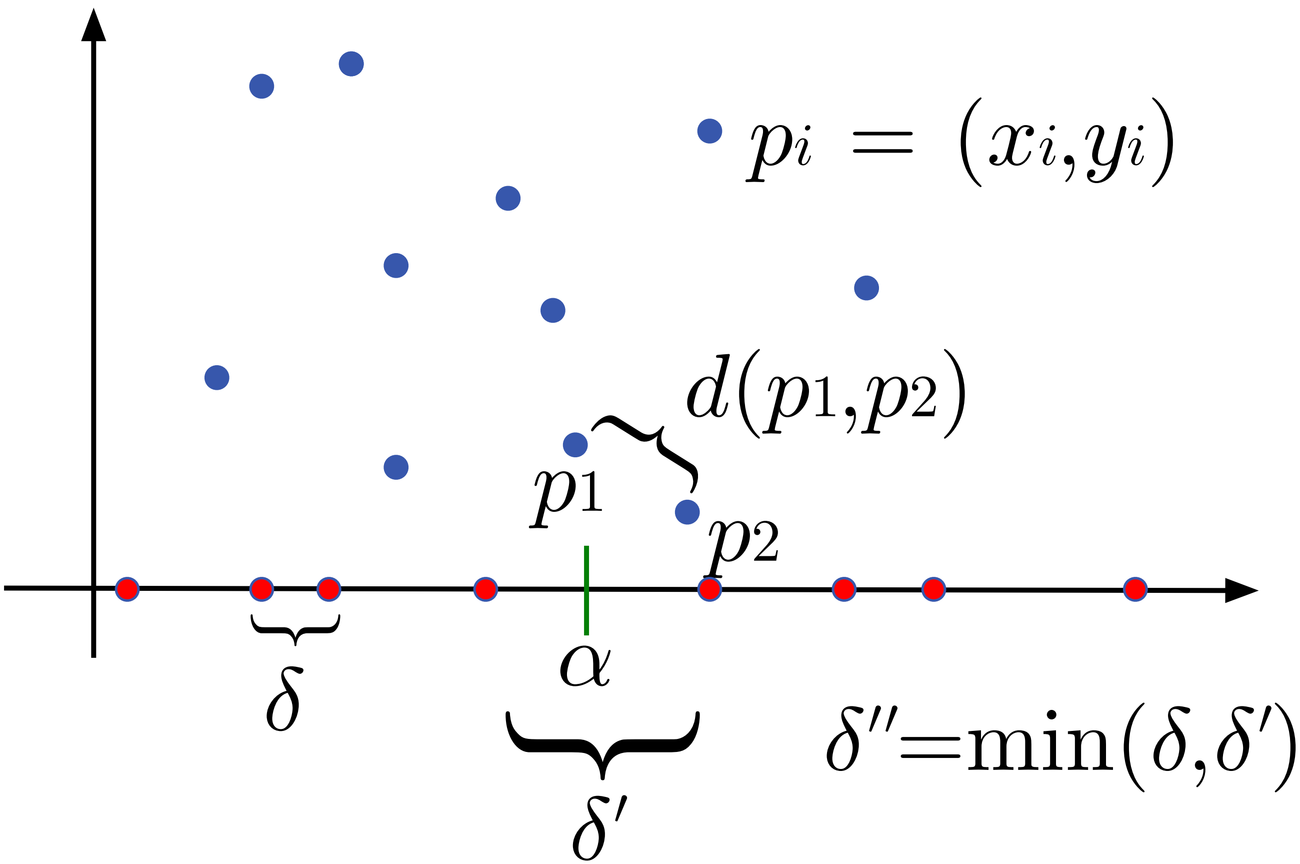
Rückblick: divide and conquer



weitere Algorithmentechniken

- Greedy-Algorithmen
- dynamische Programmierung
- Backtracking
- branch and bound
- Heuristiken und Approximation

Greedy-Algorithmen

- Einsatz bei Auswahl einer Teilmenge von Objekten mit bestimmten Eigenschaften;
- Greedy-Algorithmen wählen aus der Eingabemenge sukzessive Elemente aus und entscheiden *ad hoc*, ob diese zu der Lösungsmenge gehören;
- Entscheidungen sind endgültig und können nicht wieder rückgängig gemacht werden!
- Greedy-Algorithmen suchen in der Regel nur lokal optimale Lösungen, die aber bisweilen auch global die besten sind.

Greedy-Algorithmus: Arbeitsweise

$$\vdots$$

```
SOLUTION ⟵ ∅
 WHILE M ≠ ∅ DO
 BEGIN
   CHOOSE ARBITRARY x ∈ M
   M ⟵ M \ {x}
   IF x MATCHES CRITERION THEN
     SOLUTION ⟵ SOLUTION ∪ {x}
 END
 RETURN SOLUTION
```

$$\vdots$$

Beispiel: Minimales Gerüst

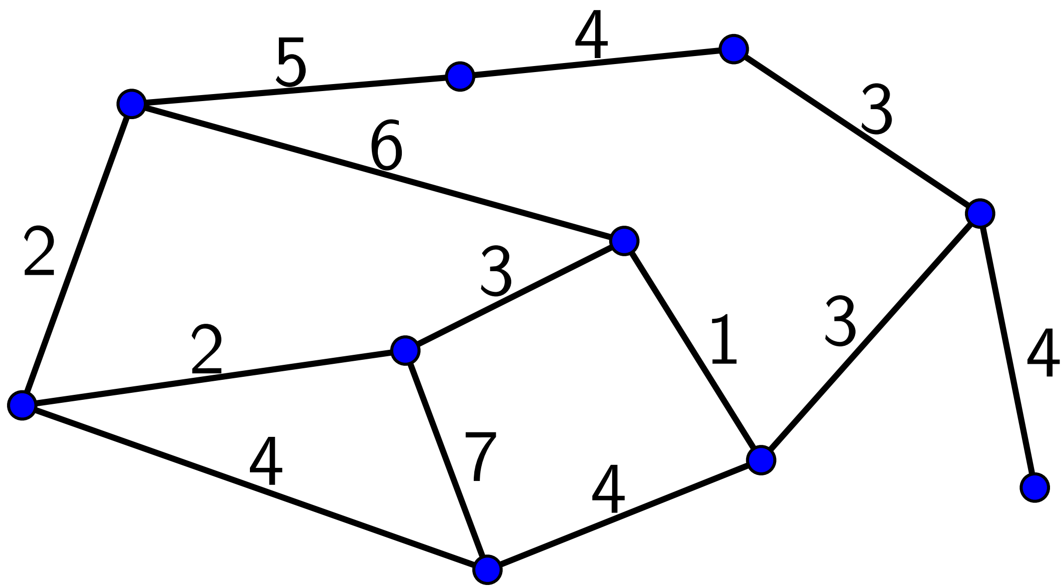
Gegeben:	endlicher Graph $G = (V, E)$, ungerichtet, zusammenhängend; Kanten-Bewertung $l : E \to I\!N$
Gesucht:	Ein Gerüst T von G mit minimaler Länge
Antwort:	Graph des minimalen Gerüstes

- $l(e)$ bezeichne die "Länge" der Kante $e \in E$).
- O.b.d.A. seien $V = \{v_1, v_2, \ldots, v_n\}$, $E = \{e_1, e_1, \ldots, e_m\}$.

Definition: minimales Gerüst

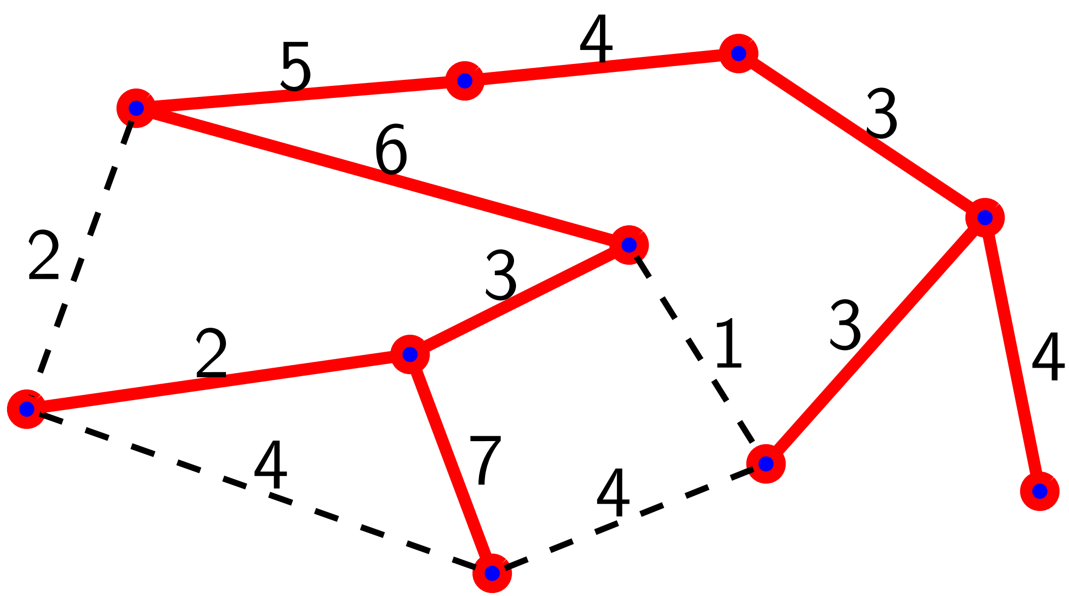
- Ein **Gerüst** von $G = (V, E)$ ist ein Teilgraph $T = (V, F)$ von G, der
  - alle Knotenpunkte von G enthält,
  - zusammenhängend ist und
  - eine minimale Anzahl von Kanten hat.
- T ist ein **Baum** und wird auch als **(auf-)spannender Baum** (*spanning tree*) bezeichnet.
- Die **Länge** von T ist: $l(T) = \sum_{e \in F} l(e)$. Dies ist die *Zielfunktion* für dieses Problem, wobei ein minimales $l(T)$ gesucht wird.
  (Stichwort: Optimierung)

Beispielgraph G

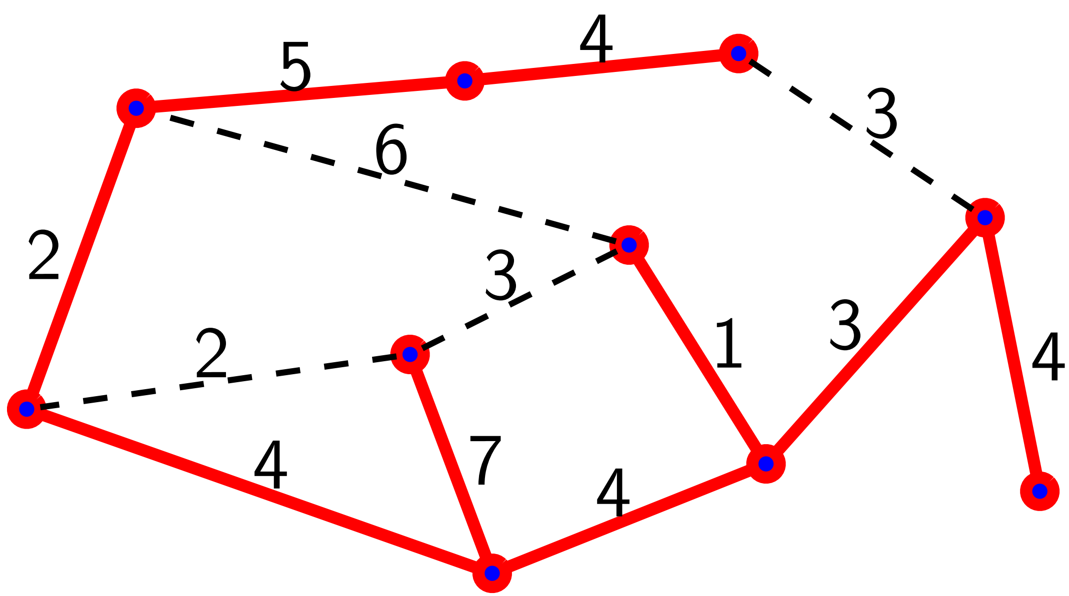


. . . dieser Graph ist zusammenhängend, ungerichtet und kantenbewertet.

Ein Gerüst von G



. . . noch ein Gerüst von G



Lösungsansatz

- **Lösungskandidaten** sind alle möglichen Gerüste T von G.
- Gesucht wird dasjenige mit kleinster Länge.
- Die Lösung ist im allgemeinen nicht eindeutig!

Lösungsraum: $\mathcal{L}_G = \{T_G \mid T_G \text{ ist Gerüst von } G\}$.

Wir suchen eine Lösung für das Problem-Beispiel (G, l):

$T_G^* \in \mathcal{L}_G$ ist *optimale* Lösung aus $\mathcal{L}_G \Leftrightarrow T_G^* \in \mathcal{L}_G$ mit $l(T_G^*) \leq l(T_G)$ für alle $T_G \in \mathcal{L}_G$.

Eigenschaften

- Die *Anzahl der Lösungen* ist $|\mathcal{L}_G|$. Es gilt dabei:

$$1 \leq |\mathcal{L}_G| \leq |V|^{|V|-2} (= n^{n-2}) \qquad \text{mit } |V| = n.$$

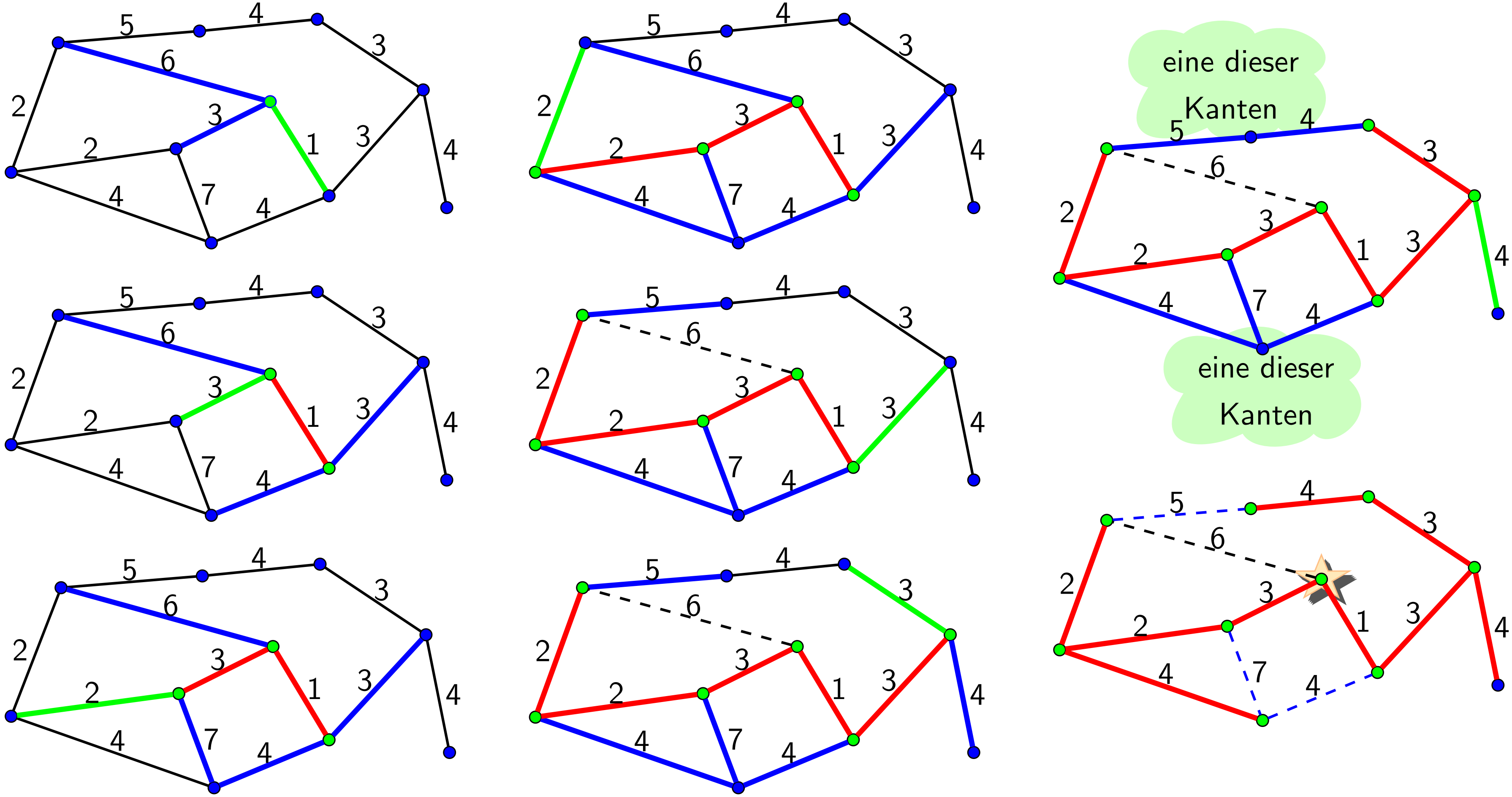
- Dabei ist $|V|^{|V|-2}$ die Anzahl der verschiedenen Gerüste des vollständigen Graphen K_n.

Sei im folgenden $G = (V, E)$ ein ungerichteter Graph mit der Kantenbewertung l.

Algorithmus von PRIM/DIJKSTRA

1. **Initialisierung:** Setze $X := \emptyset; U := \emptyset$.
2. **Wähle** beliebigen Knotenpunkt $v \in V$.
   $X \longleftarrow \{v\}; V \longleftarrow V \setminus \{v\}$.
3. Betrachte alle Kanten $(a, b) \in E \cap (X \times V)$ **Wähle** eine kürzeste Kante (a_1, b_1). Dann gilt:
   $$l(a_1, b_1) \leq l(a, b) \quad \text{für alle } (a, b) \in E \cap (X \times V)$$
4. $X \longleftarrow X \cup \{b_1\}; \quad U \longleftarrow U \cup \{(a_1, b_1)\};$
   $V \longleftarrow V \setminus \{b_1\}$.
5. **Falls** $V \neq \emptyset$ setze mit 3. fort;
6. **STOP**. *Output*: $T = (X, U)$.

Beispiel: PRIM/DIJKSTRA-Alg.



Komplexität: PRIM/DIJKSTRA-Alg.

- Im Schritt 3. sind höchstens m Vergleiche auszuführen. Schritt 3. wird $(n-1)$-mal aufgerufen. $\rightarrow T_A = O(m * n)$
- Die **untere Schranke** ist $\Omega(m)$, da jede Kante mindestens einmal geprüft werden muss.
- Etwas bessere Algorithmen existieren ($O(|E| + |V| \cdot \log |V|)$ durch Ausnutzung der verwendeten Datenstruktur).
- Die Suche nach dem besten Algorithmus wird durch die nachweisbar untere Schranke eingeschränkt!

Korrektheit

Theorem: MINGERÜST liefert ein Minimal- Gerüst des zusammenhängenden Graphen G.

Beweis: Es werden sukzessive Teil-Unter-Graphen (*TU-Graphen*) von G erzeugt: $H_1, H_2, \ldots, H_i, H_{i+1}, \ldots, H_n$ mit $H_1 = (\{v\}, \emptyset)$ und $H_n = (V, U)$.

- Die TU-Graphen sind Bäume:
  - H_1 ist ein Baum.
  - H_i ist Baum $\Rightarrow H_{i+1}$ ist Baum
- Für $i = 1, 2, \ldots, n-1$: Ist H_i TU-Graph eines Minimalgerüsts von G, so gilt dies auch für H_{i+1}.

Korrektheit (2)

- T ist Gerüst von G
  $\Rightarrow \exists e_1 = (a_1, b_1) \in U . a_1 \in V_i, b_1 \in V \setminus V_i$.
- Ist $\tilde{e}$ die Kante, die bei Erzeugung von H_{i+1} aus H_i ausgewählt wird:

  $\tilde{e} = e_1 \quad \Rightarrow \quad$ fertig.

  $\tilde{e} \neq e_1 \quad \Rightarrow \quad \tilde{e} = (\tilde{a}, \tilde{b}) \in E$ mit $\tilde{a} \in V_i$ und $\tilde{b} \in V - V_i$.
- $\tilde{e}$ zu T hinzufügen. Es entsteht ein Kreis C.
- Wegen Schritt 3. muss $l(\tilde{e}) \leq l(e_1)$ gelten.
- Bilde $\tilde{T} = (T \setminus \{e_1\}) \cup \{\tilde{e}\} \quad \Rightarrow \quad l(\tilde{T}) \leq l(T)$.
  Minimalität von $T \Rightarrow \tilde{T}$ ist Minimalgerüst $\Rightarrow H_{i+1}$ ist TU-Graph eines Minimalgerüsts $\tilde{T}$ von G. $\square$

Beispiel: Münzrückgabe

Mit dem Euro-Münzsatz:

34¢ Wechselgeld: 

GREEDY

Mit einem anderen Münzsatz:

Fazit

- Es gibt Probleme, die sich mit *Greedy-Algorithmen* optimal lösen lassen.
- **Aber:** Es gibt auch Probleme, für die *Greedy-Algorithmen* ungeeignet sind!
- **Frage:** Lassen sich die Probleme, die durch *Greedy-Algorithmen* optimal gelöst werden, charakterisieren?
  - Ein *Greedy-Algorithmus* liefert immer eine optimale Lösung, wenn das Problem eine **matroidale Struktur** hat.
  - **Matroide** sind Verallgemeinerungen von Matrizen.

Vereinfachtes Rucksackproblem

Gegeben:	Rucksack mit Maximalgewicht m; Multimenge von Objekten, jeweils mit Gewicht und Wert $o : O \to \mathbb{N}$ mit Objektmenge $O = \{(t_1, g_1, w_1), \ldots (t_n, g_n, w_n)\}$
Gesucht:	Füllung mit maximalem Wert
Antwort:	Liste der Objekte im Rucksack

Beispiel: vereinf. Rucksackproblem

i	Objekttyp	g_i	w_i
1	Nägel	5,0 kg	8,50 €/kg
2	Heftpflaster	0,1 kg	50,00 €/kg
3	Whisky	1,5 kg	38,00 €/kg
4	Toilettenpapier	0,5 kg	4,50 €/kg
5	Käse	9,0 kg	14,50 €/kg
6	Brot	1,0 kg	2,20 €/kg

Maximales Rucksackgewicht: $m = 15kg$

Annahme: Alles ist beliebig portionierbar!

dynamische Programmierung

- Problemlösung durch Zerlegung in Teilprobleme und Zusammenfügen der Lösungen. (**Ähnlichkeit zu *divide and conquer***)
- Dynamische Programmierung eignet sich zur Lösung solcher Probleme, bei denen es sehr *viel weniger* Teilprobleme als mögliche Zerlegungen gibt.
- Häufig ist es günstig, **gefundene Teillösungen in einer Tabelle zu speichern**, so daß jedes Teilproblem nur einmal gelöst wird.
- Anwendung: z.B. *Vergleich des genetische Codes unterschiedlicher Arten von Lebewesen*

dynamische Prog.: Vorgehensweise

1. Bestimme die zu einer optimalen Gesamtlösung führenden Zerlegungen der Eingabe.
2. Definiere rekursiv den Wert der Zielfunktion für
   - die Gesamtlösung und
   - die in Betracht kommenden Teilaufgaben.
3. Berechne diese Werte für alle in Betracht kommenden Teilaufgaben und speichere sie ab.
4. Bestimme eine optimale Gesamtlösung unter Rückgriff auf die vorher berechneten Werte für benutzte Teilaufgaben.

Matrizenmultiplikation

Eine optimale Folge von Entscheidungen erfordert optimale Teilfolgen!

Betrachten wir z.B. n Matrizen der Größe $d_{i-1} \times d_i$ mit $1 \leq i \leq n$.

- Gesucht wird $M = M_1 \cdot \dots \cdot M_n$ mit **minimaler Anzahl von Multiplikationen**.
- Die Matrizenmultiplikation ist assoziativ, also kann das Produkt auf verschiedene Arten berechnet werden, und man kann **durch geschickte Klammerung die Anzahl der skalaren Multiplikationen senken**.

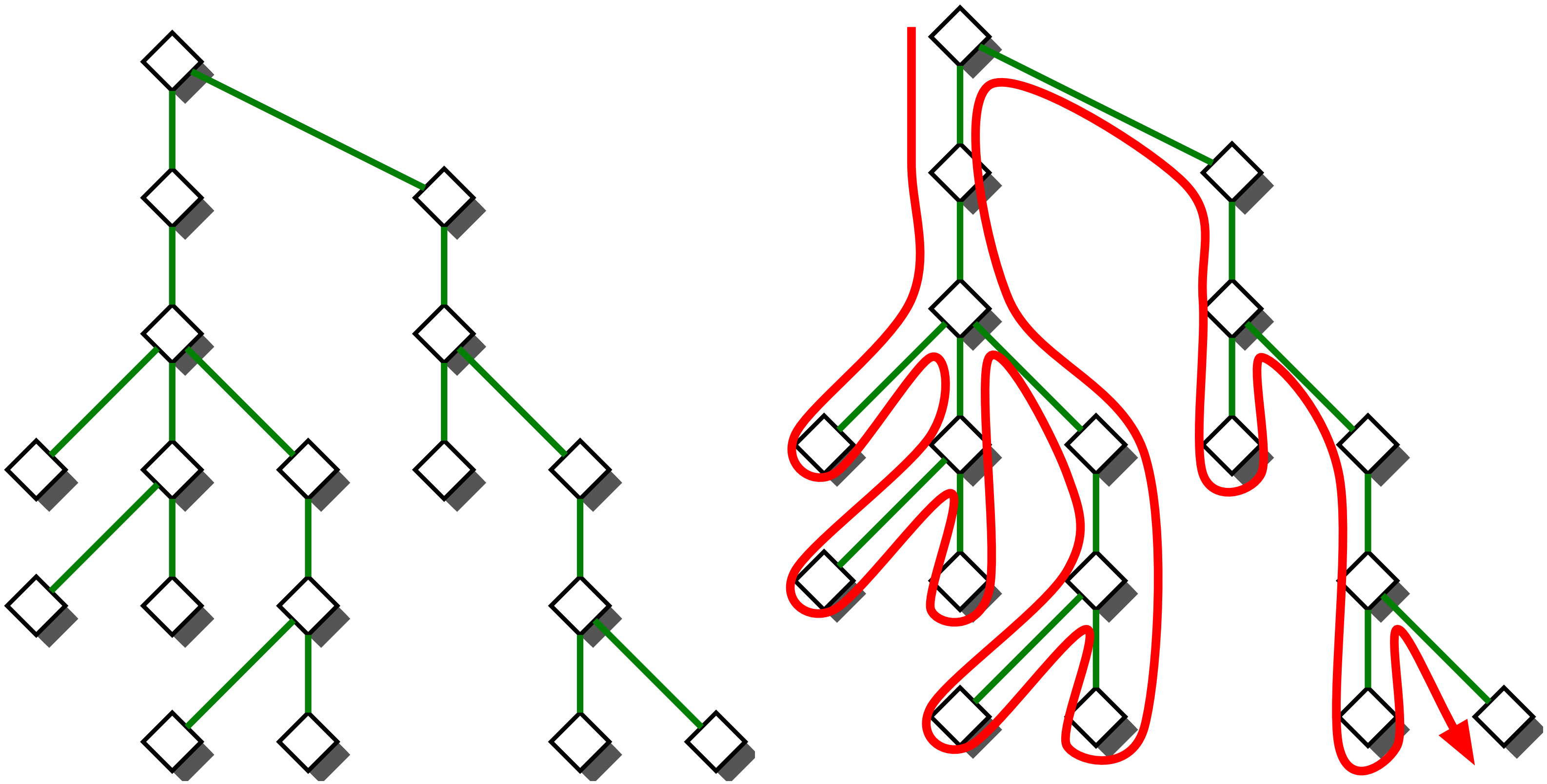
Beispiel: Matrizenmultiplikation

- 4 Matrizen A:[13,5], B:[5,89], C:[89,3] und D:[3,34]:

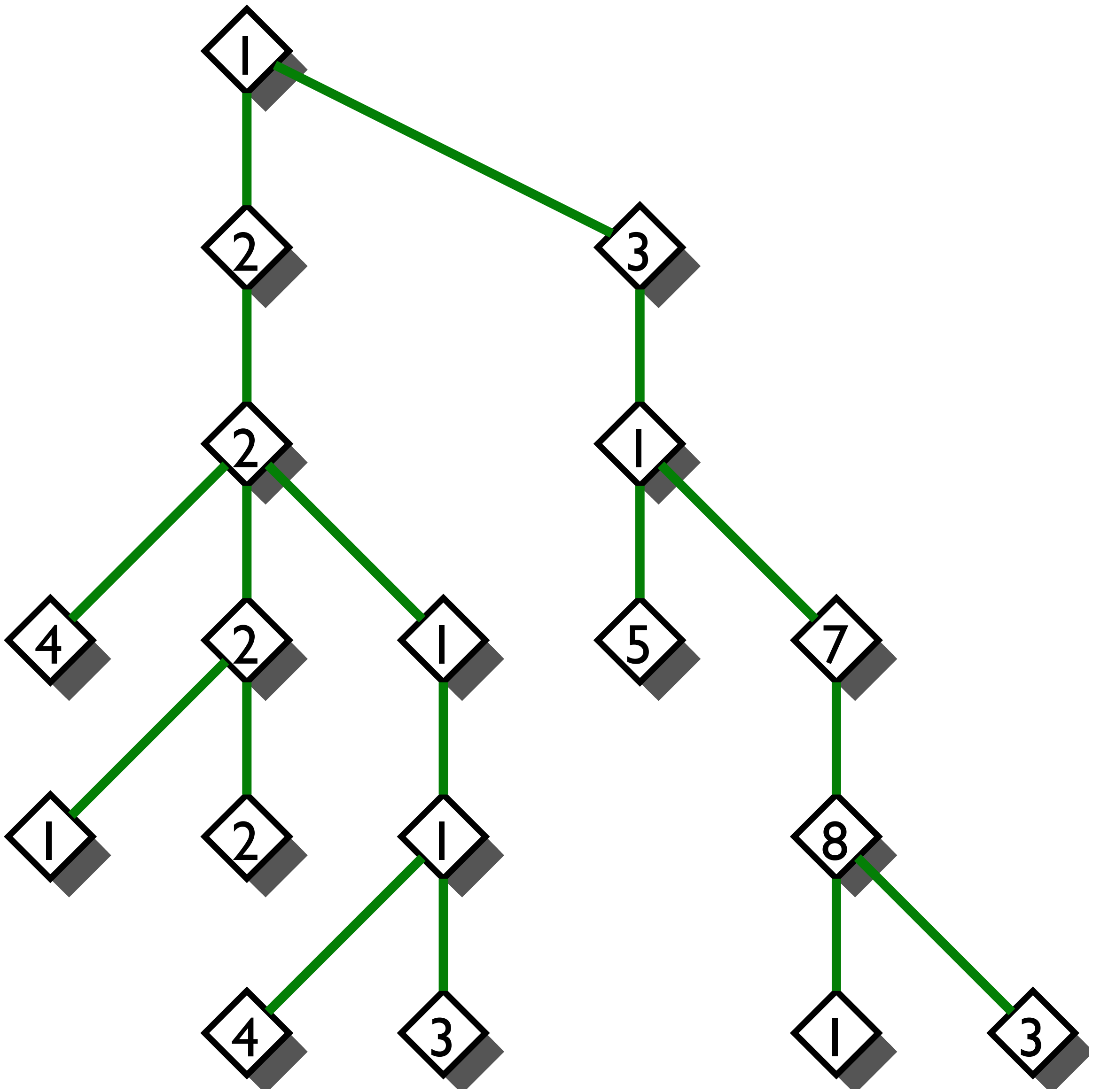
Klammerung	Anzahl der Multiplikationen
$A(B(CD))$	9078+15130+2210 = 26418
$A((BC)D)$	1335+510+2210 = 4055
$(AB)(CD)$	5785+9078+39338 = **54201**
$(A(BC))D$	1335+195+1326 = **2856**
$((AB)C)D$	5785+3471+1326 = 10582

- beste Klammerung ist ca. 19mal schneller
- Lösungen gemeinsamer Teile werden in Tabelle gespeichert

Backtracking



branch and bound



Ausblick

- noch einmal dynamische Programmierung:
  - Eine Lösung für das *Wechselgeldproblem*.
  - Das allgemeine *Rucksackproblem*.
  - Das *Travelling-Salesman-Problem*.
- weitere Algorithmentechniken