NP-vollständig – was nun?

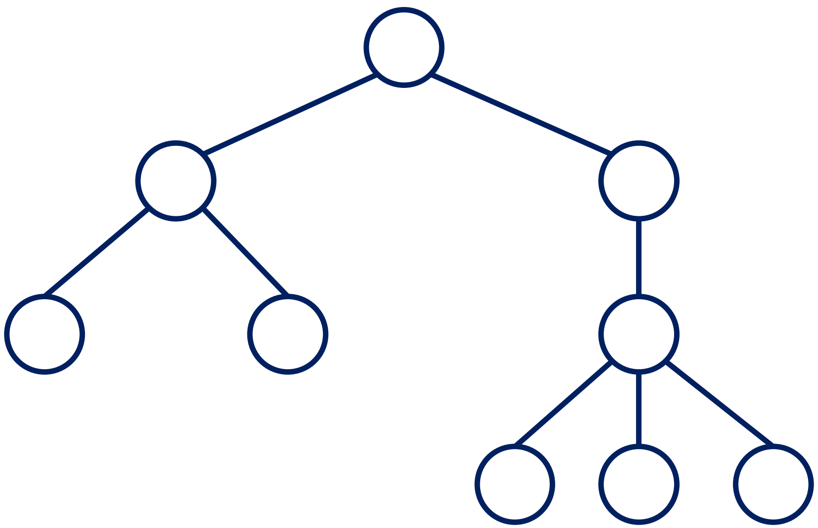
- Spezialfall?

- Heuristiken
  - ✓ Lokale Verbesserung
  - ✓ Backtracking
  - ✓ Branch-and-Bound

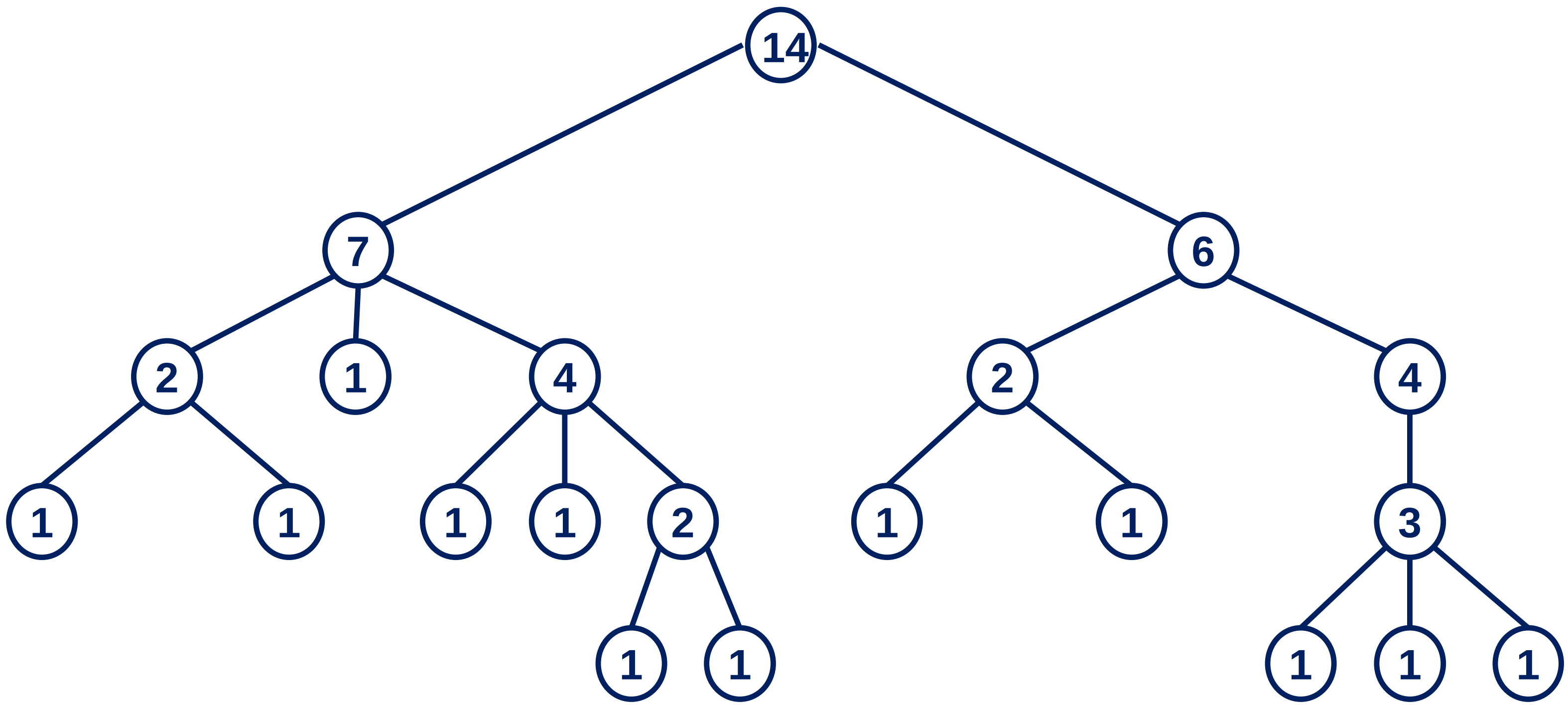
- Approximationsalgorithmen
  (für Optimierungsprobleme)

Clique auf Bäumen

Die größte Clique in einem Baum mit ≥ 2 Knoten hat Größe 2, denn jede Clique der Größe ≥ 3 besitzt einen Kreis.



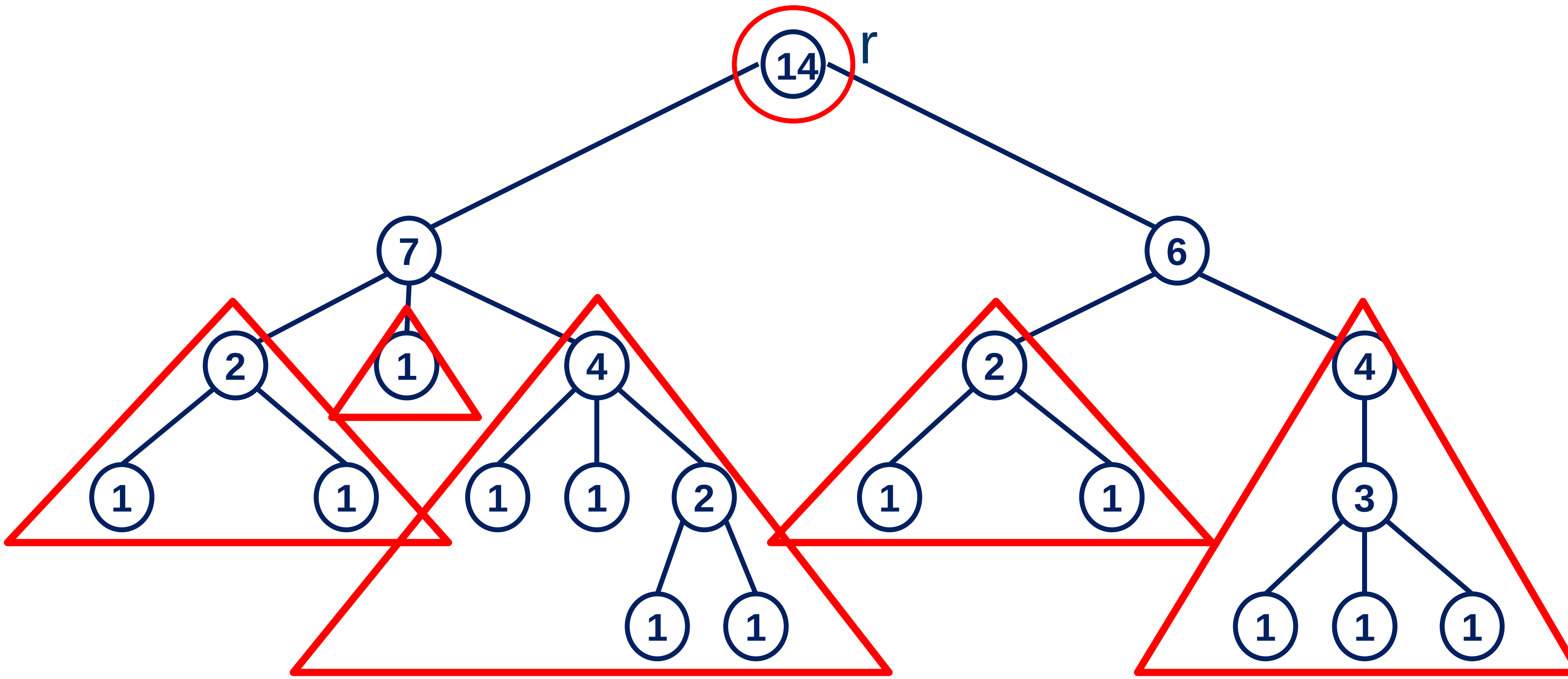
Unabhängige Menge auf Bäumen



Unabhängige Menge: Menge von Knoten, die nicht durch Kanten verbunden sind.

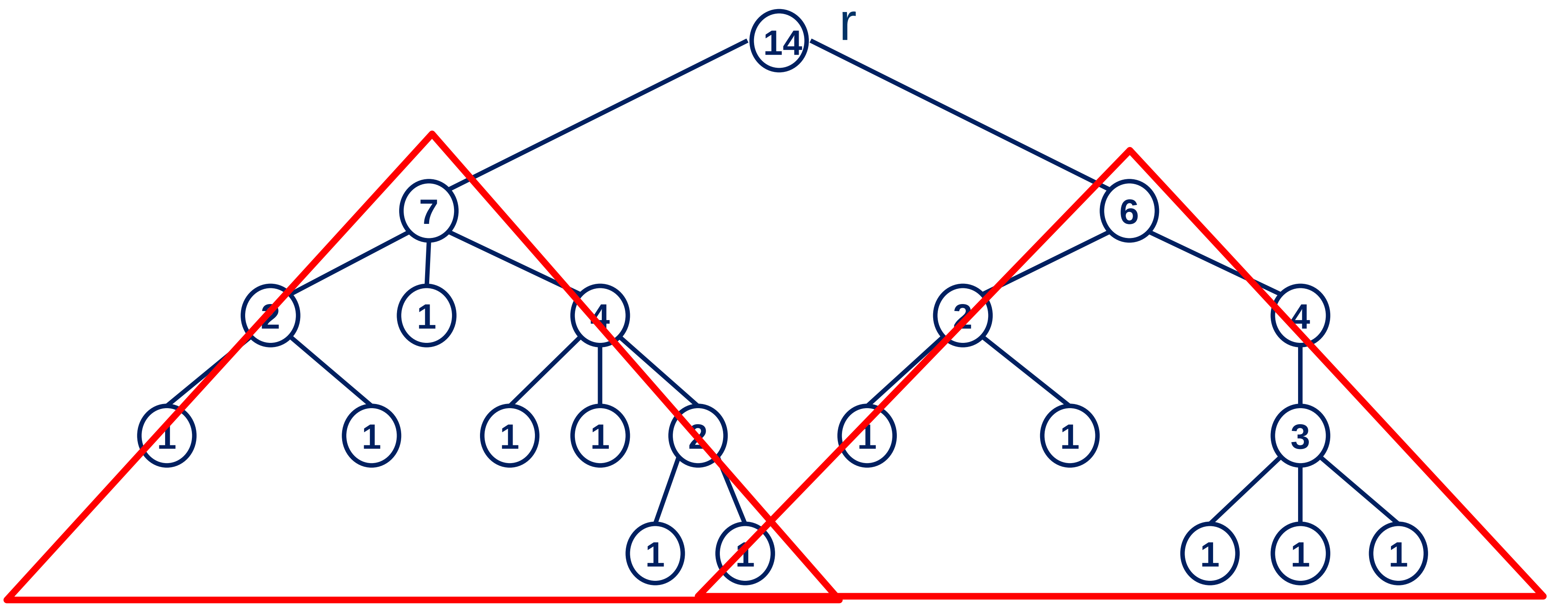
Gesucht: **maximale** unabhängige Menge.

Unabhängige Menge auf Bäumen



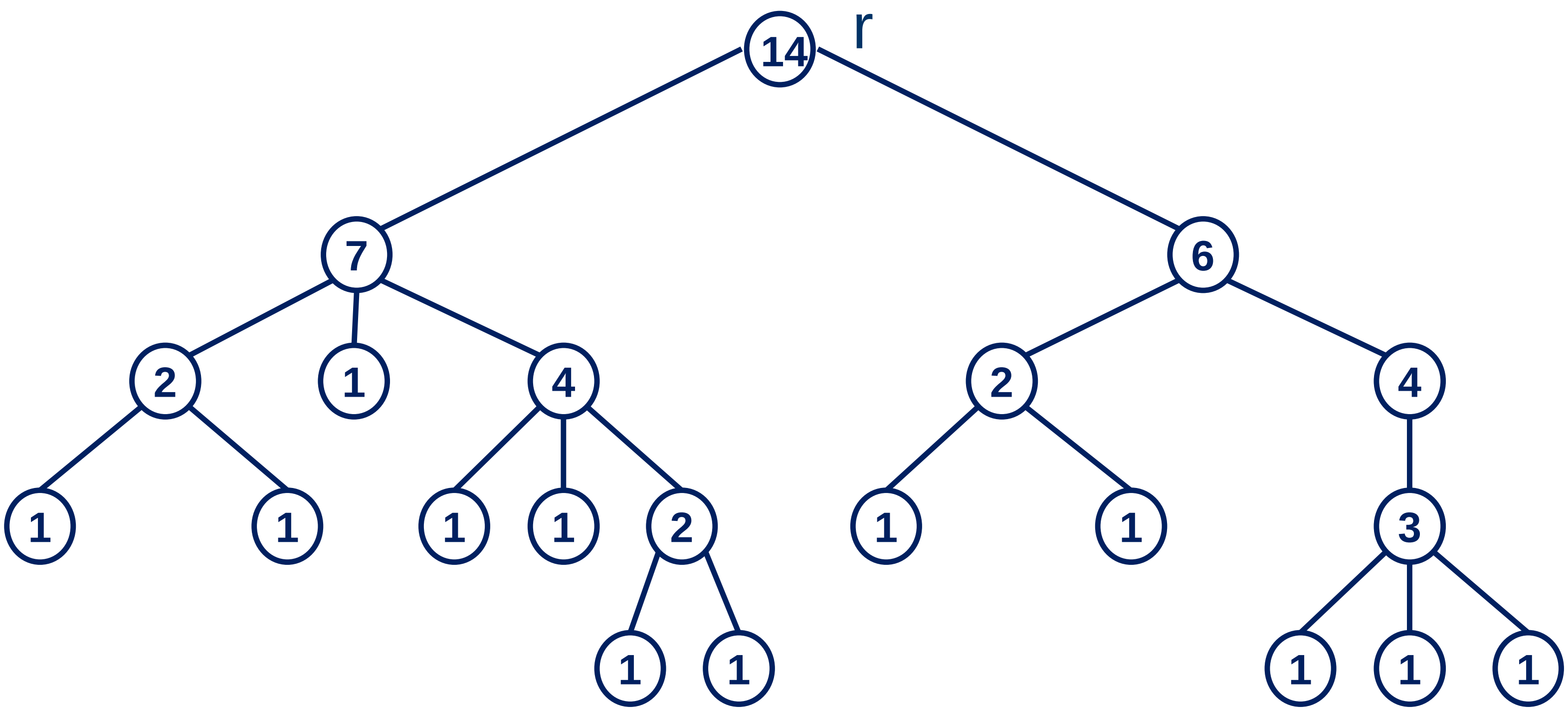
Maximale unabhängige Menge U_1 mit Wurzel r: {r} vereinigt mit maximalen unabhängigen Mengen der Teilbäume, deren Wurzeln die Enkel von r sind.

Unabhängige Menge auf Bäumen



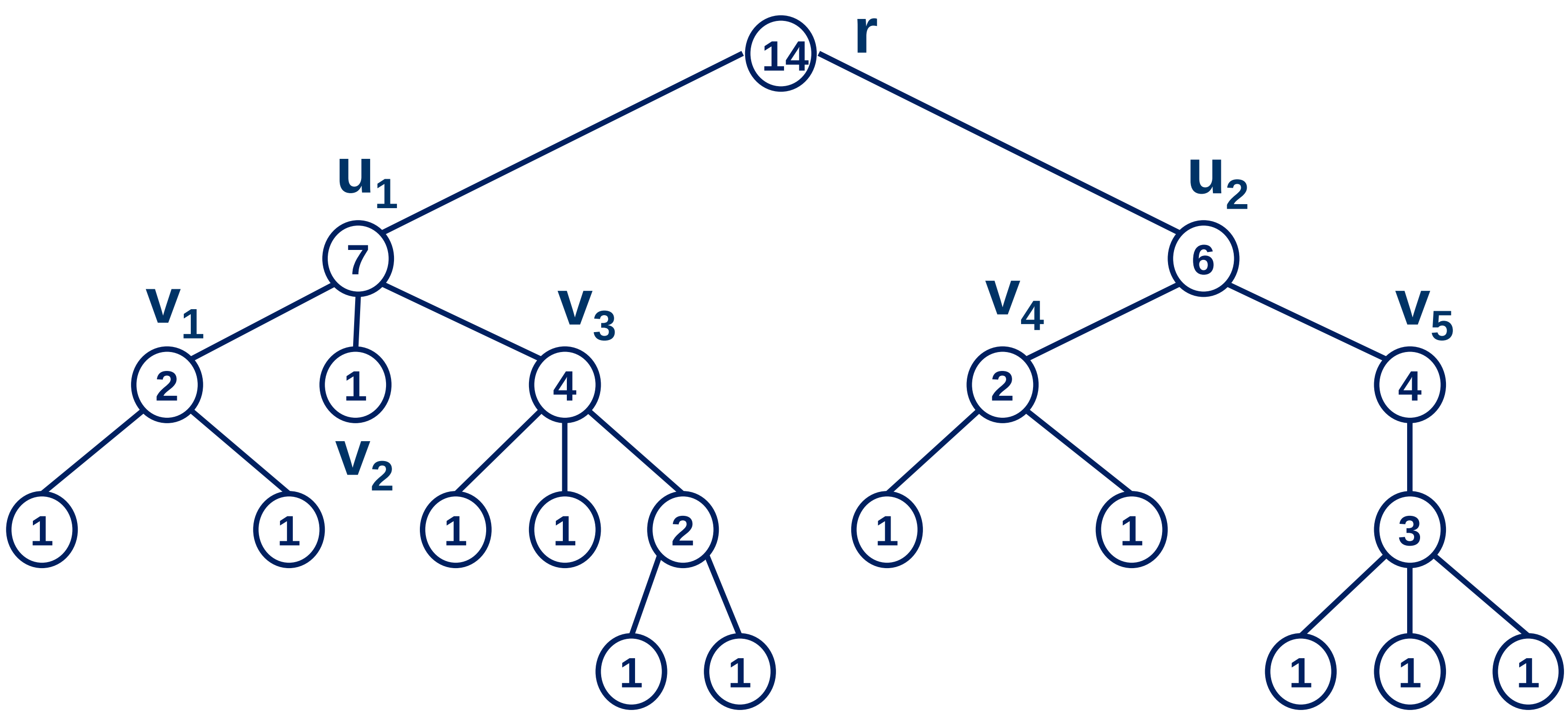
Maximale unabhängige Menge U_2 ohne Wurzel r: Vereinigung mit maximalen unabhängigen Mengen der Teilbäume, deren Wurzeln die Kinder von r sind.

Unabhängige Menge auf Bäumen



Berechnung von U_1 und U_2: rekursiv (dynamisches Programm)
Maximale unabhängige Menge U: $\max\{U_1,U_2\}$.
Dynamisches Programm: bereche maximale unabhängige Mengen der Knoten des Baums bottom-up.

Unabhängige Menge auf Bäumen



Dynamisches Programm: **bereche maximale unabhängige Mengen der Knoten des Baums bottom-up.**

Beispiel: $OPT(r)=\max\{ OPT(u_1)+OPT(u_2)\ ,$

$$1+OPT(v_1)+OPT(v_2)+OPT(v_3)+OPT(v_4)+OPT(v_5)\}$$

Die Sprache 2SAT

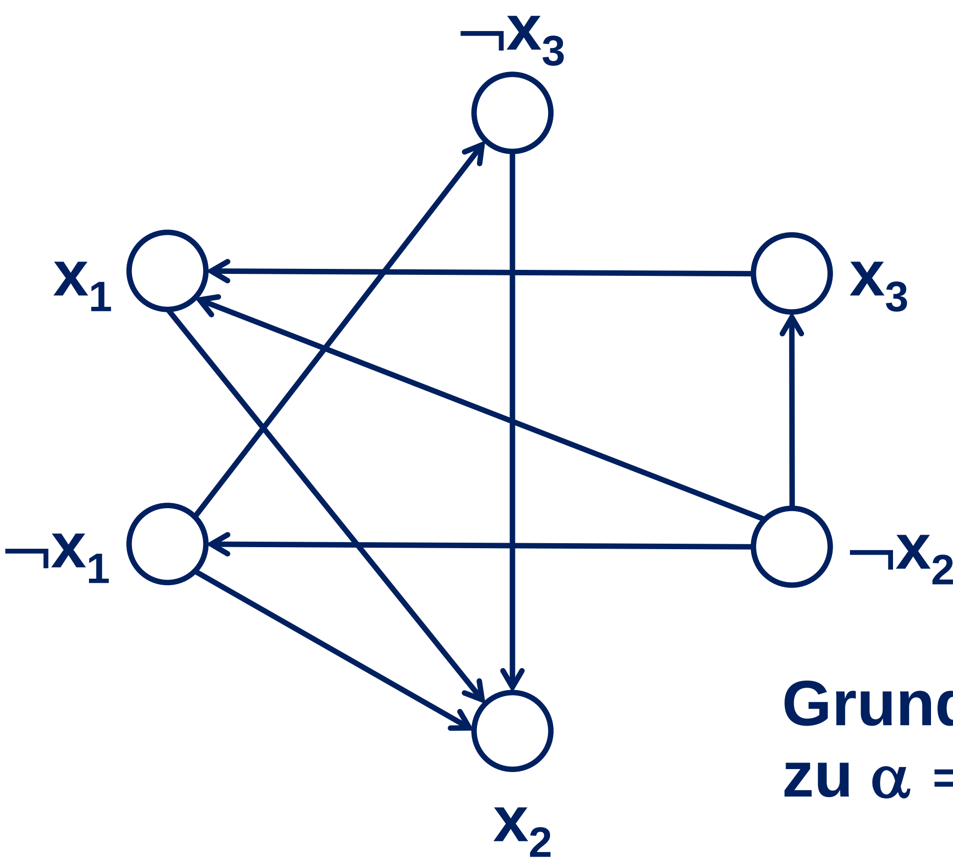
2SAT $:= \{\langle\phi\rangle \mid \phi$ **ist erfüllbare 2-KNF Formel.**$\}$

Satz 4.2 **2SAT** $\in$ **P.**

2SAT und Graphen - Beispiel

$\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$

G(φ):



$(\alpha,\beta) \in G(\phi)$

$\Updownarrow$

$(\neg\alpha \vee \beta)$ **Klausel in** ϕ

oder

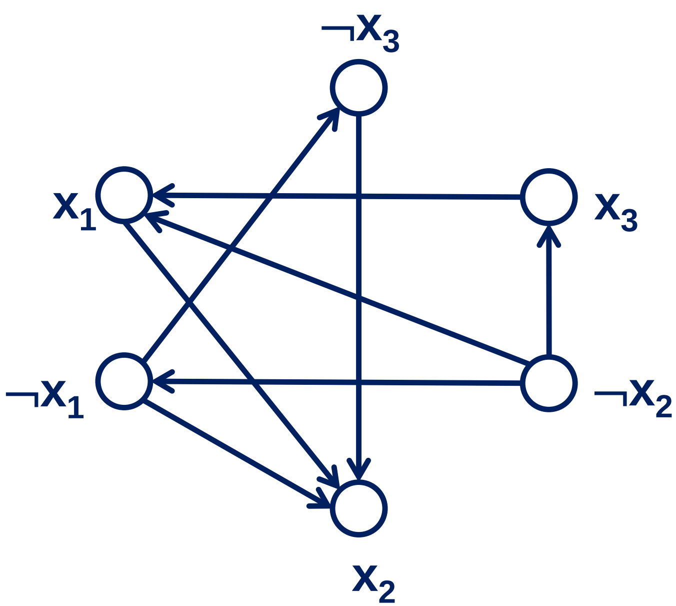
$(\beta \vee \neg\alpha)$ **Klausel in** ϕ

Grund: $(\neg\alpha \vee \beta)$ **äquivalent zu** $\alpha \Rightarrow \beta$ **(log. Implikation)**

2SAT und Graphen

$\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$

G(ϕ):



$\Leftarrow$: Angenommen, es gibt x, so dass in G(ϕ) Pfade von x nach $\neg$x und von $\neg$x nach x existieren.

Pfad von x nach $\neg$x:

$$x \Rightarrow y_1 \Rightarrow y_2 \Rightarrow \ldots \Rightarrow y_k \Rightarrow \neg x$$

Pfad von $\neg$x nach x:

$$\neg x \Rightarrow z_1 \Rightarrow z_2 \Rightarrow \ldots \Rightarrow z_l \Rightarrow x$$

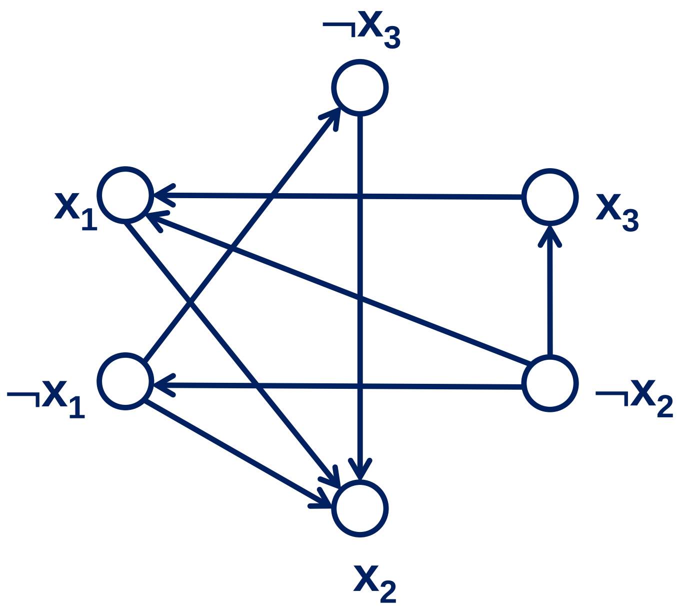
Beide Implikationsketten können nicht wahr sein, d.h. ϕ unerfüllbar.

Lemma 4.1 ϕ ist unerfüllbar genau dann, wenn es eine Variable x gibt, so dass in G(ϕ) Pfade von x nach $\neg$x und von $\neg$x nach x existieren.

2SAT und Graphen

$\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$

$G(\phi)$:

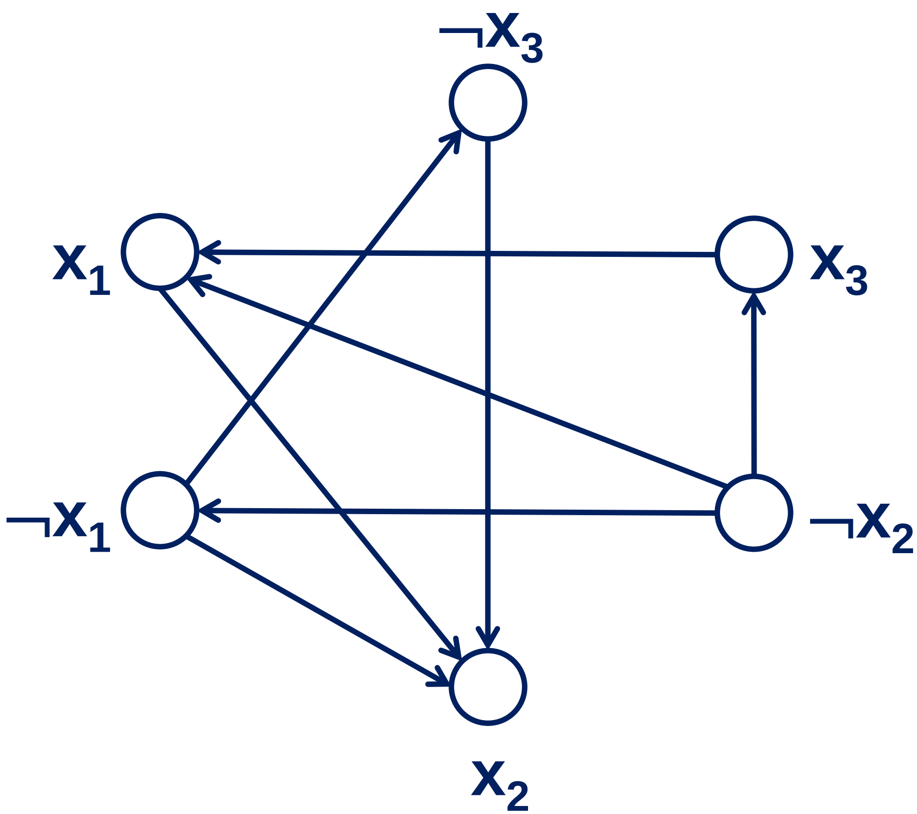


$\Rightarrow$: **Betrachte folgenden Algorithmus: Solange es in $G(\phi)$ noch einen Knoten α gibt, von dem $\neg\alpha$ nicht erreicht werden kann und dessen Wert noch nicht feststeht, definiere Wert von α und aller von α erreichbaren Knoten als 1, definiere den Wert der Negationen dieser Knoten als 0.**

Lemma 4.1 ϕ ist **unerfüllbar** genau dann, wenn es eine Variable x gibt, so dass in $G(\phi)$ Pfade von x nach $\neg$x und von $\neg$x nach x existieren.

2SAT und Graphen - Beispiel

$$\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$$

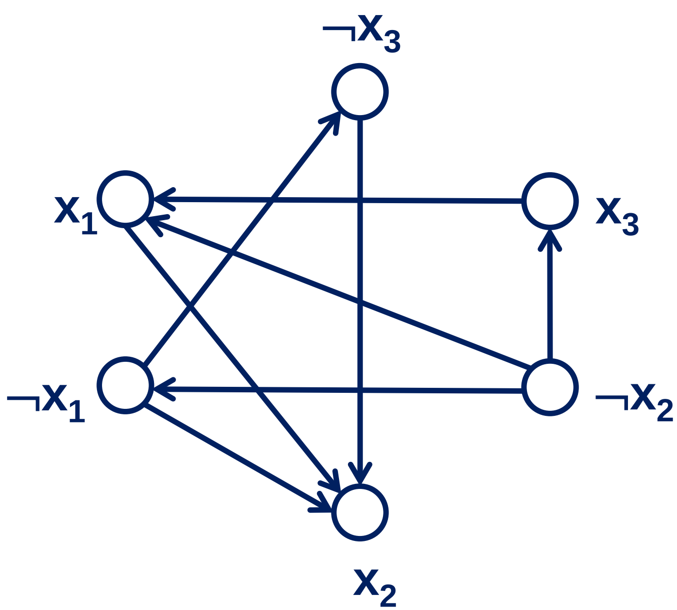


Solange es in $G(\phi)$ noch einen Knoten α gibt, von dem $\neg\alpha$ nicht erreicht werden kann und dessen Wert noch nicht feststeht, definiere Wert von α und aller von α erreichbaren Knoten als 1, definiere den Wert der Negationen dieser Knoten als 0.

2SAT und Graphen

$\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$

G(ϕ):



$\Rightarrow$: Betrachte folgenden Algorithmus: Solange es in G(ϕ) noch einen Knoten α gibt, von dem $\neg\alpha$ nicht erreicht werden kann und dessen Wert noch nicht feststeht, definiere Wert von α und aller von α erreichbaren Knoten als 1, definiere den Wert der Negationen dieser Knoten als 0.

Da es keine Variable x gibt, so dass in G(ϕ) Pfade von x nach $\neg$x und von $\neg$x nach x existieren, muss dieser Algorithmus erfolgreich darin sein, allen Knoten Werte zuzuweisen.

2SAT und Graphen

Zu zeigen: jedem Knoten wird nur einmal ein Wert zugeordnet.

Wir wissen: $(\alpha,\beta) \in G(\phi) \Leftrightarrow (\neg\alpha \vee \beta)$ Klausel in ϕ
Damit

$$(\alpha,\beta) \in G(\phi) \Leftrightarrow (\neg\beta,\neg\alpha) \in G(\phi)$$

Gibt es also einen Pfad von α nach γ in $G(\phi)$, dann auch von $\neg\gamma$ nach $\neg\alpha$.

Angenommen, γ werde zum erstenmal ein Wert wegen α zugeordnet. Dann muss $\neg\gamma$ noch unbelegt sein. Fall 1: $\neg\gamma$=0: dann müsste γ bereits (mit 1) belegt sein, was die Annahme widerlegt.

2SAT und Graphen

Zu zeigen: jedem Knoten wird nur einmal ein Wert zugeordnet.

Wir wissen: $(\alpha,\beta) \in G(\phi) \Leftrightarrow (\neg\alpha \vee \beta)$ Klausel in ϕ
Damit

$$(\alpha,\beta) \in G(\phi) \Leftrightarrow (\neg\beta,\neg\alpha) \in G(\phi)$$

Gibt es also einen Pfad von α nach γ in $G(\phi)$, dann auch von $\neg\gamma$ nach $\neg\alpha$.

Angenommen, γ werde zum erstenmal ein Wert wegen α zugeordnet. Dann muss $\neg\gamma$ noch unbelegt sein. Fall 2: $\neg\gamma$=1: Dann müsste wegen eines Pfades von $\neg\gamma$ nach $\neg\alpha$ auch bereits $\neg\alpha$ und damit α belegt sein.

2SAT und Graphen

Zu zeigen: es kann keine Kante $(\alpha,\beta) \in G(\phi)$ geben, für die $\alpha=1$ und $\beta=0$ ist (d.h. alle Klauseln sind wahr).

Wir wissen: $(\alpha,\beta) \in G(\phi) \Leftrightarrow (\neg\alpha \vee \beta)$ Klausel in ϕ
Damit

$$(\alpha,\beta) \in G(\phi) \Leftrightarrow (\neg\beta,\neg\alpha) \in G(\phi)$$

Gibt es also einen Pfad von α nach γ in $G(\phi)$, dann auch von $\neg\gamma$ nach $\neg\alpha$.

Angenommen, im Algo wird von α aus ein γ erreicht mit $\gamma=0$. Da es einen Pfad von $\neg\gamma$ nach $\neg\alpha$ gibt und $\neg\gamma=1$ ist, muss daher bereits $\neg\alpha=1$ und damit $\alpha=0$ sein, was der Annahme widerspricht.

Die Sprache 2SAT

2SAT := $\{\langle\phi\rangle \mid \phi$ **ist erfüllbare 2-KNF Formel.**$\}$

Satz 4.2 **2SAT** $\in$ **P.**

Optimierungsprobleme

Ein Optimierungsproblem Π ist definiert durch:

1. **Menge $\mathcal{I}$ von Instanzen,**
2. **Für jede Instanz eine Menge zulässiger Lösungen F(I),**
3. **Eine Funktion w, die jeder zulässigen Lösung s einen positiven Wert w(s) zuordnet.**

Falls $w(s) \in \mathbb{N}$ für alle zulässigen Lösungen s ist, nennen wir Π ein diskretes Optimierungsproblem.

Optimierungsprobleme

Ein Optimierungsproblem Π ist definiert durch:

1. **Menge $\mathcal{I}$ von Instanzen,**
2. **Für jede Instanz eine Menge zulässiger Lösungen F(I),**
3. **Eine Funktion w, die jeder zulässigen Lösung s einen positiven Wert w(s) zuordnet.**

Ziel bei Instanz I: Finde zulässige Lösung $s \in F(I)$ mit w(s) möglichst groß (Maximierungsproblem) oder w(s) möglichst klein (Minimierungsproblem).

Optimierungsproblem RS_{opt}

1. Instanzen: $I = \langle G, W, g \rangle$, $G = \{g_1, \ldots, g_n\}$, $W = \{w_1, \ldots, w_n\}$.

2. Zulässige Lösungen: $S \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in S} g_i \leq g$.

3. Wert einer zulässigen Lösung: $w(S) = \sum_{i \in S} w_i$.

4. RS_{opt} ist Maximierungsproblem.

Erinnerung: RS_{ent} ist NP-vollständig.

Gäbe es einen Polynomialzeitalgorithmus für RS_{opt}, dann wäre P=NP.

Optimierungsproblem TSP_{opt}

1. Instanzen: $I = \langle \Delta \rangle$, $\Delta = (d_{ij})$, $1 \leq i,j \leq n$.

2. Zulässige Lösungen: Permutationen π auf $\{1,\ldots,n\}$.

3. Wert einer zulässigen Lösung: $w(\pi) = \sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(1)}$.

4. TSP_{opt} ist Minimierungsproblem.

Erinnerung: TSP_{ent} ist NP-vollständig.

Gäbe es einen Polynomialzeitalgorithmus für TSP_{opt}, dann wäre P=NP.

Approximationsalgorithmus

- Ein **Approximationsalgorithmus** A für ein Optimierungs-problem Π ist ein Algorithmus, der bei Eingabe I in Zeit polynomiell in der Größe der Instanz eine zulässige Lösung $s \in F(I)$ berechnet.

- **A(I)** bezeichnet Ausgabe von A bei Eingabe I.

- **opt(I)** bezeichnet Wert einer optimalen zulässigen Lösung für Instanz I.

Approximationsfaktor

- **Approximationsfaktor** von A bei Eingabe I ist definiert als

$$\delta_A(I) = \frac{w(A(I))}{opt(I)}$$

Approximationsfaktor

- **Approximationsfaktor** von A bei Eingabe I ist definiert als

$$\delta_A(I) = \frac{w(A(I))}{opt(I)}$$

- A hat Approximationsfaktor k, wenn für jede Instanz I gilt
  - $\delta_A(I) \geq k$ bei einem Maximierungsproblem,
  - $\delta_A(I) \leq k$ bei einem Minimierungsproblem.

Approximationsfaktor

- **Approximationsfaktor** von A bei Eingabe I ist definiert als

$$\delta_A(I) = \frac{w(A(I))}{opt(I)}$$

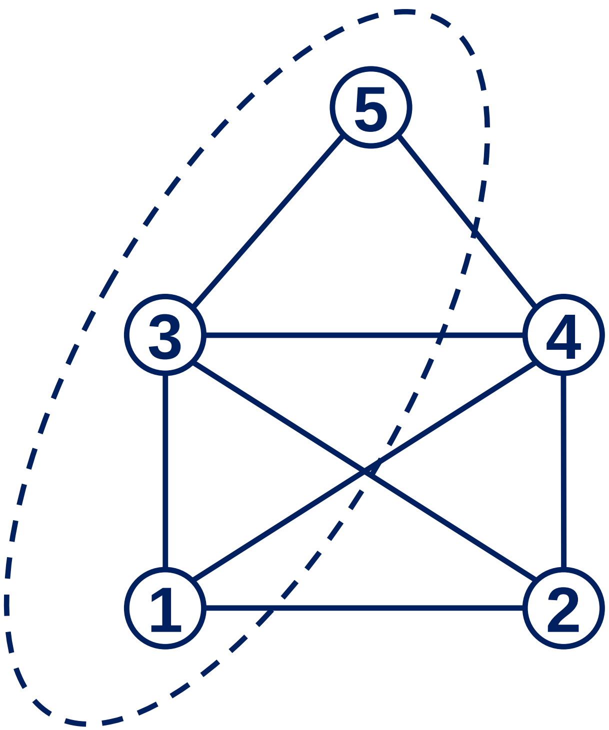
- A hat Approximationsfaktor k, wenn für jede Instanz I gilt
  - $\delta_A(I) \geq k$ bei einem Maximierungsproblem,
  - $\delta_A(I) \leq k$ bei einem Minimierungsproblem.

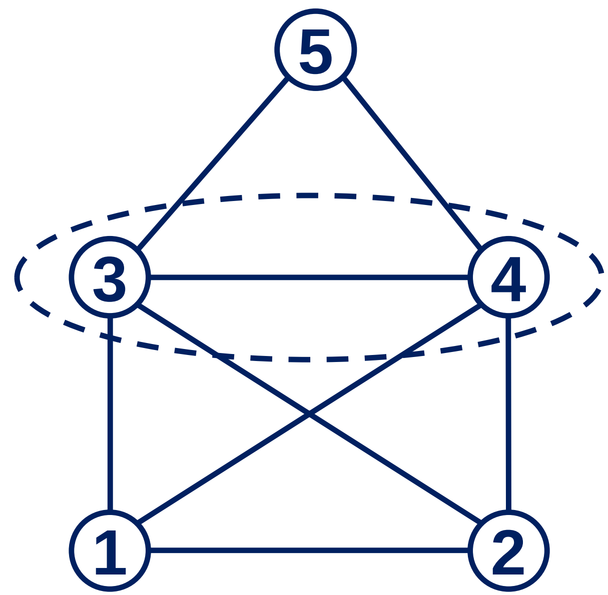
- $\delta_A(I) \leq 1$ bei Maximierungsproblem.
- $\delta_A(I) \geq 1$ bei Minimierungsproblem.

Schnitte in Graphen

Graph G = (V,E), V =$\{$1,…,n$\}$.

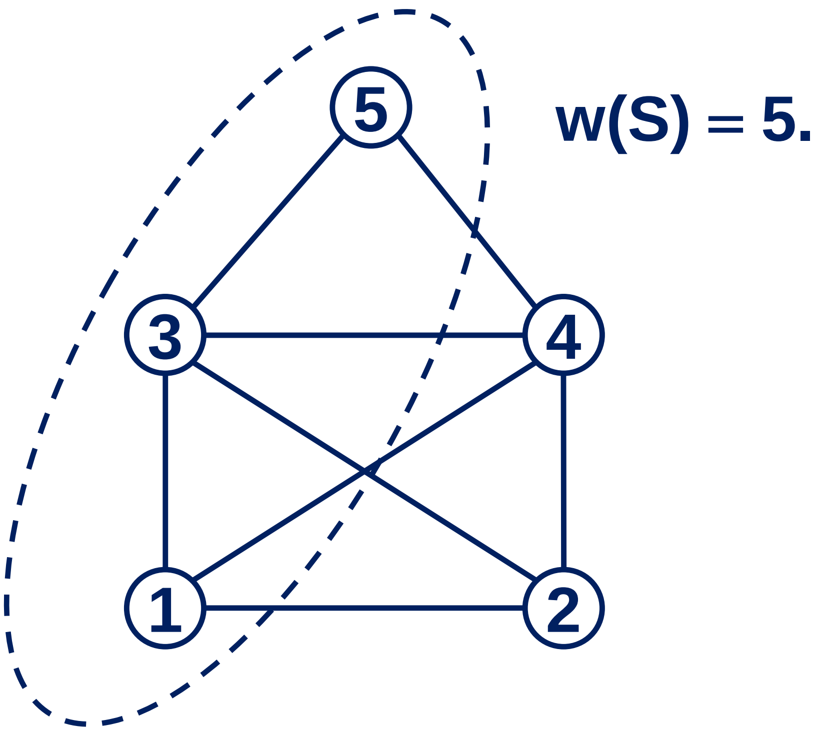
Schnitt in G ist Teilmenge S $\subseteq$ $\{$1,…,n$\}$.

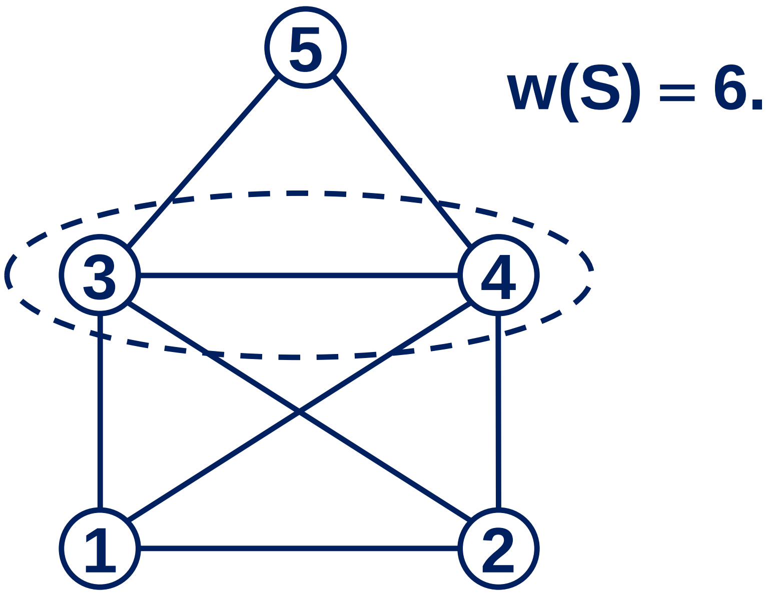




Schnitte in Graphen

Wert $w(S)$ eines Schnitts = Anzahl Kanten mit Knoten in S und V\S.





Optimierungsproblem Max-Cut

1. **Instanzen: $G = (V,E)$, $V = \{1,\ldots,n\}$.**
2. **Zulässige Lösungen: $S \subseteq \{1,\ldots,n\}$.**
3. **Wert einer zulässigen Lösung: $w(S) =$ Wert des Schnitts S.**
4. **Max-Cut ist Maximierungsproblem.**

Cut := $\{\langle G, k\rangle \mid$ G besitzt einen Schnitt der Größe mind. $k\}$

Cut ist NP-vollständig.

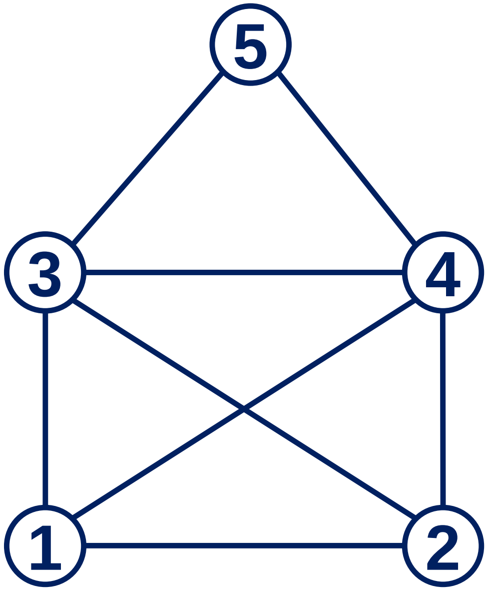
Approximationsalgorithmus $LI_{\text{Max-Cut}}$

$LI_{\text{Max-Cut}}$ bei Eingabe $\langle G \rangle$, $G=(V,E)$:

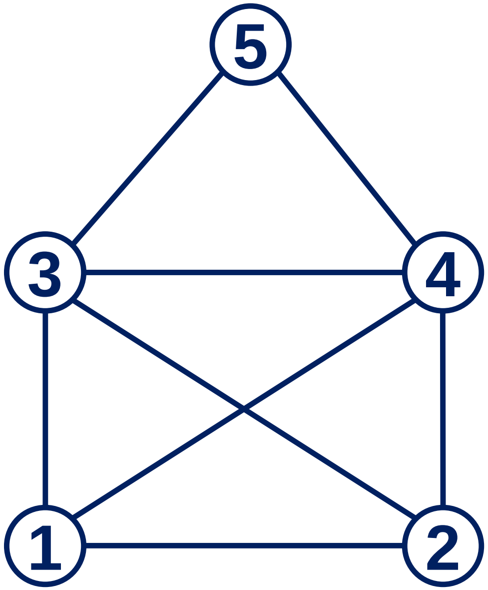
1. $S:=\emptyset$.
2. Solange es ein $v \in V$ gibt, so dass $w(S\Delta\{v\}) > w(S)$,
3. Setze $S:=S\Delta\{v\}$
4. Ausgabe S.

$$S\Delta\{v\} := \begin{cases} S \cup \{v\} & \text{falls } v \notin S \\ S \setminus \{v\} & \text{falls } v \in S \end{cases}$$

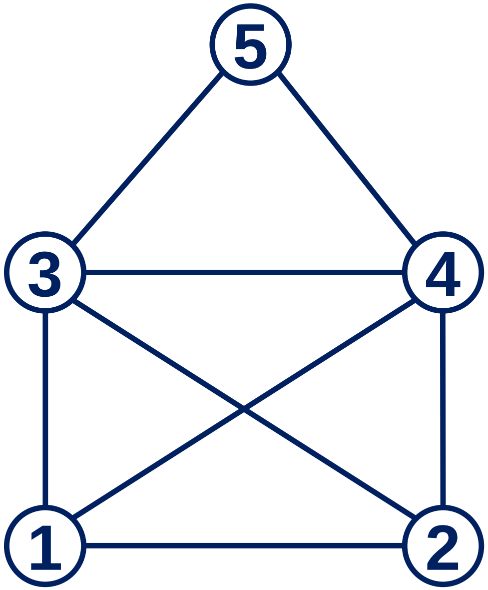
$LI_{\text{Max-Cut}}$ - Beispiel



$LI_{Max\text{-}Cut}$ - Beispiel



$LI_{\text{Max-Cut}}$ - Beispiel



Approximationsalgorithmus $LI_{Max\text{-}Cut}$

$LI_{Max\text{-}Cut}$ **bei Eingabe $\langle G \rangle$, $G = (V,E)$:**

1. $S := \emptyset$.
2. Solange es ein $u \in V$ gibt, so dass $w(S \Delta \{v\}) > w(S)$,
3. Setze $S := w(S \Delta \{v\})$
4. Ausgabe S.

Satz 4.3 Algorithmus $LI_{Max\text{-}Cut}$ hat Approximationsfaktor 1/2.

$LI_{\text{Max-Cut}}$ – Idee der Analyse

Satz 4.3 Algorithmus $LI_{\text{Max-Cut}}$ hat Approximationsfaktor 1/2.

- Zeige obere (untere) Schranke für Wert der optimalen Lösung.

- $opt(G) \leq |E|$.

- Zeige untere (obere) Schranke für Wert berechneter Lösung.

- $w(A(G)) \geq |E|/2$.
- Für jeden Knoten v geht mindestens die Hälfte seiner Kanten über den Schnitt. Sonst ist dieser verbesserbar.

Optimierungsproblem TSP_{opt}

1. Instanzen: $I = \langle \Delta \rangle$, $\Delta = (d_{ij})$, $1 \le i,j \le n$.

2. Zulässige Lösungen: Permutationen π auf $\{1,\ldots,n\}$.

3. Wert einer zulässigen Lösung: $w(\pi) = \sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(1)}$.

4. TSP_{opt} ist Minimierungsproblem.

Erinnerung TSP_{ent} ist NP-vollständig.

Optimierungsproblem ETSP_{opt}

ETSP (Euclidean Traveling Salesman Problem):

1. **Städte s_i sind Punkte im P^2, $s_i=(s_{ix},s_{iy})$.**
2. **d_{ij} gegeben durch euklidische Distanz, d.h.**

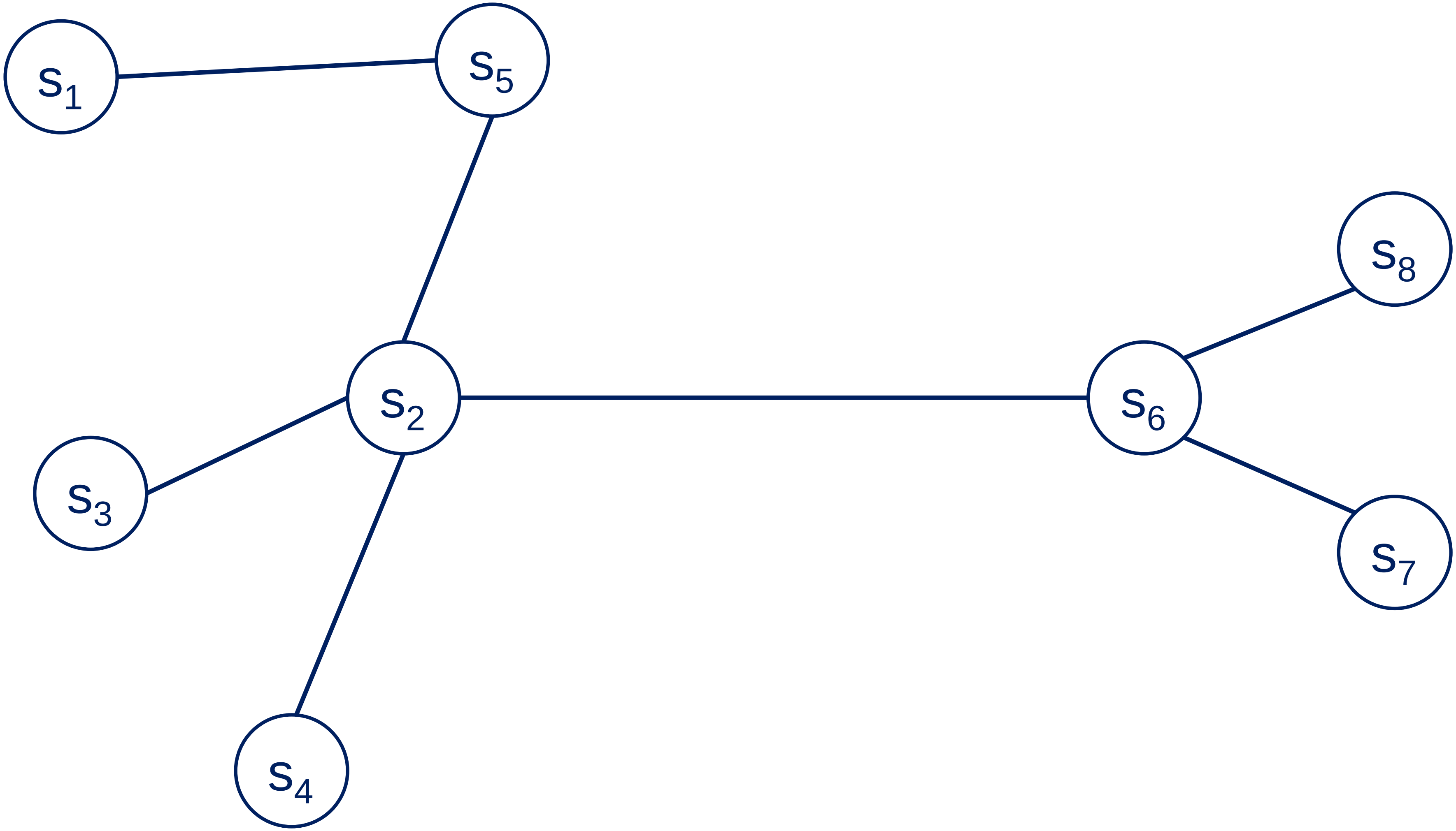
$$d_{ij}=((s_{ix}-s_{jx})^2+(s_{iy}-s_{jy})^2)^{1/2}$$

3. **Zulässige Lösungen: Permutationen π auf $\{1,\ldots,n\}$.**
4. **Wert einer zulässigen Lösung: $w(\pi) = \sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(1)}$..**
5. **ETSP_{opt} ist Minimierungsproblem.**

Minimale Spannbäume

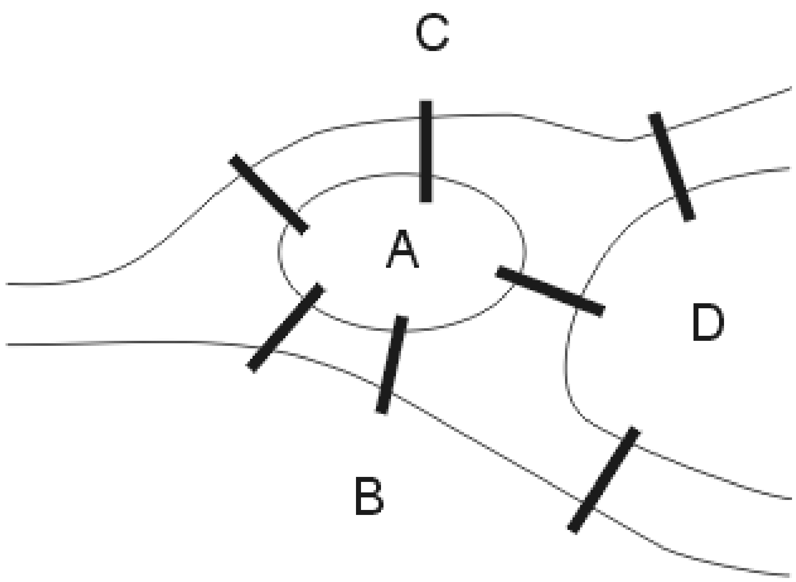
- $G = (V,E,w)$ gewichteter Graph.
- Spannbaum ist Baum auf Knotenmenge V, so dass jede Baumkante in E enthalten ist.
- Gesamtgewicht eines Spannbaums ist Summe seiner Kantengwichte.
- **Minimaler Spannbaum** für G ist Spannbaum mit minimalem Gesamtgewicht.
- Minimaler Spannbaum für Punkte im $\mathbb{P}^2$ ist minimaler Spannbaum für vollständigen Graphen mit euklidischen Distanzen als Kantengewichten.

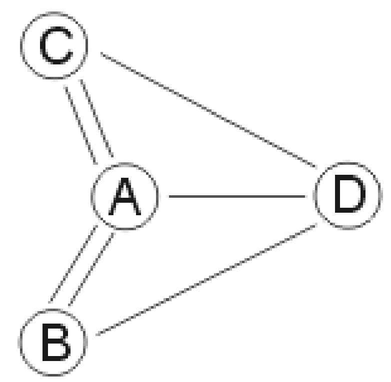
Minimaler Spannbaum - Bespiel



Eulerkreise

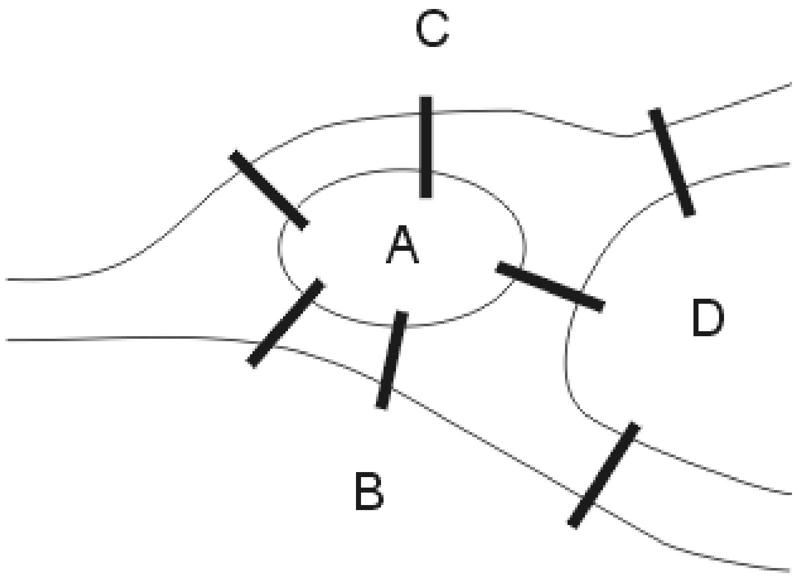
- **G = (V,E) Graph.**
- **Eulerkreis ist Kreis auf Knotenmenge V, der jede Kante aus E genau einmal enthält.**

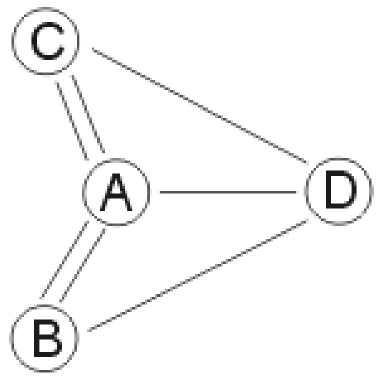




Eulerkreise

Lemma 4.4 **Ein zusammenhängender Graph G = (V,E) besitzt genau dann einen Eulerkreis, wenn jeder Knoten in V geraden Grad besitzt.**





Minimale Spannbäume und Eulerkreise

Beobachtung Minimale Spannbäume und Eulerkreise können in polynomieller Zeit berechnet werden.

Konstruktion eines Eulerkreises:
wiederhole
- starte bei beliebigem Knoten v mit noch unbenutzten Kanten
- wiederhole
  - gehe entlang einer der unbenutzten Kanten

  bis wieder zurück bei v
- baue den neuen Kreis in den bisherigen Kreis ein

bis alle Kanten benutzt worden sind

Approximationsalgorithmus MSB

1. Konstruiere einen minimalen Spannbaum T auf den Punkten $s_1, \ldots, s_n$.

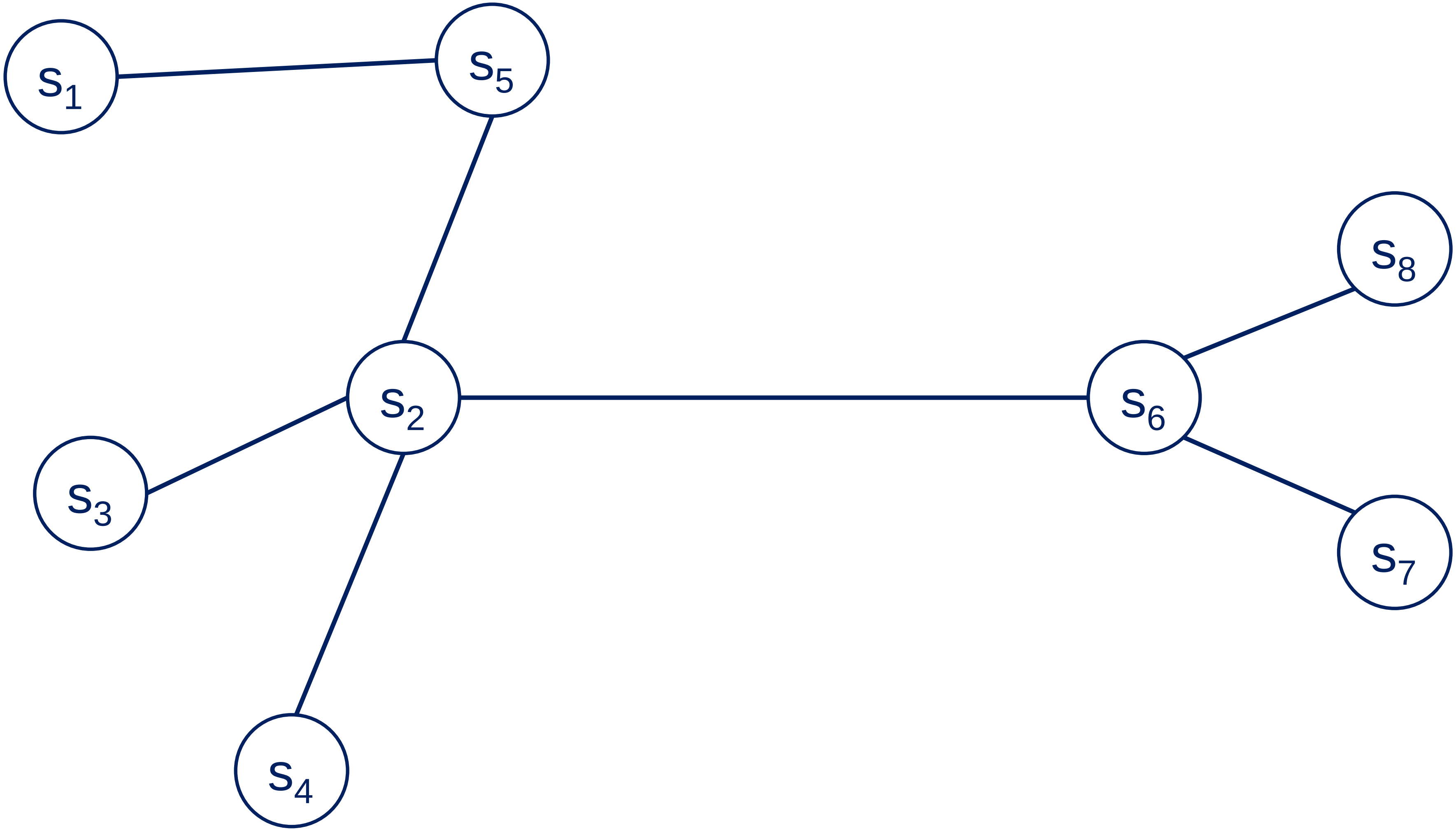
2. Konstruiere aus T einen Graph H, indem jede Kante in T verdoppelt wird.

3. Finde in H einen Eulerkreis K.

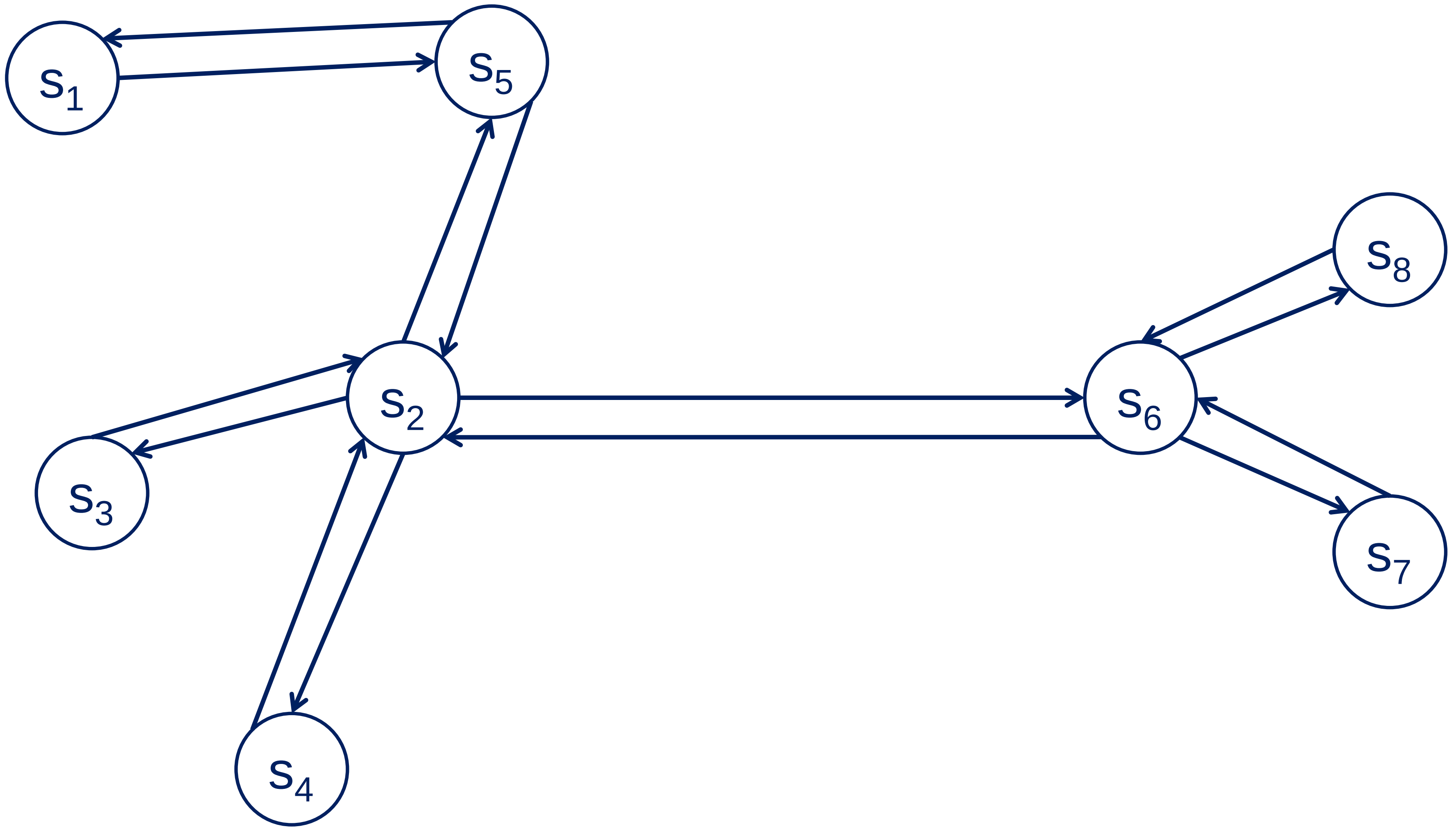
4. Berechne die Reihenfolge $s_{\pi(1)}, \ldots, s_{\pi(n)}$ des ersten Auftretens der Knoten $s_1, \ldots, s_n$ in K.

5. Gib $\pi = (\pi(1), \ldots, \pi(n))$ aus.

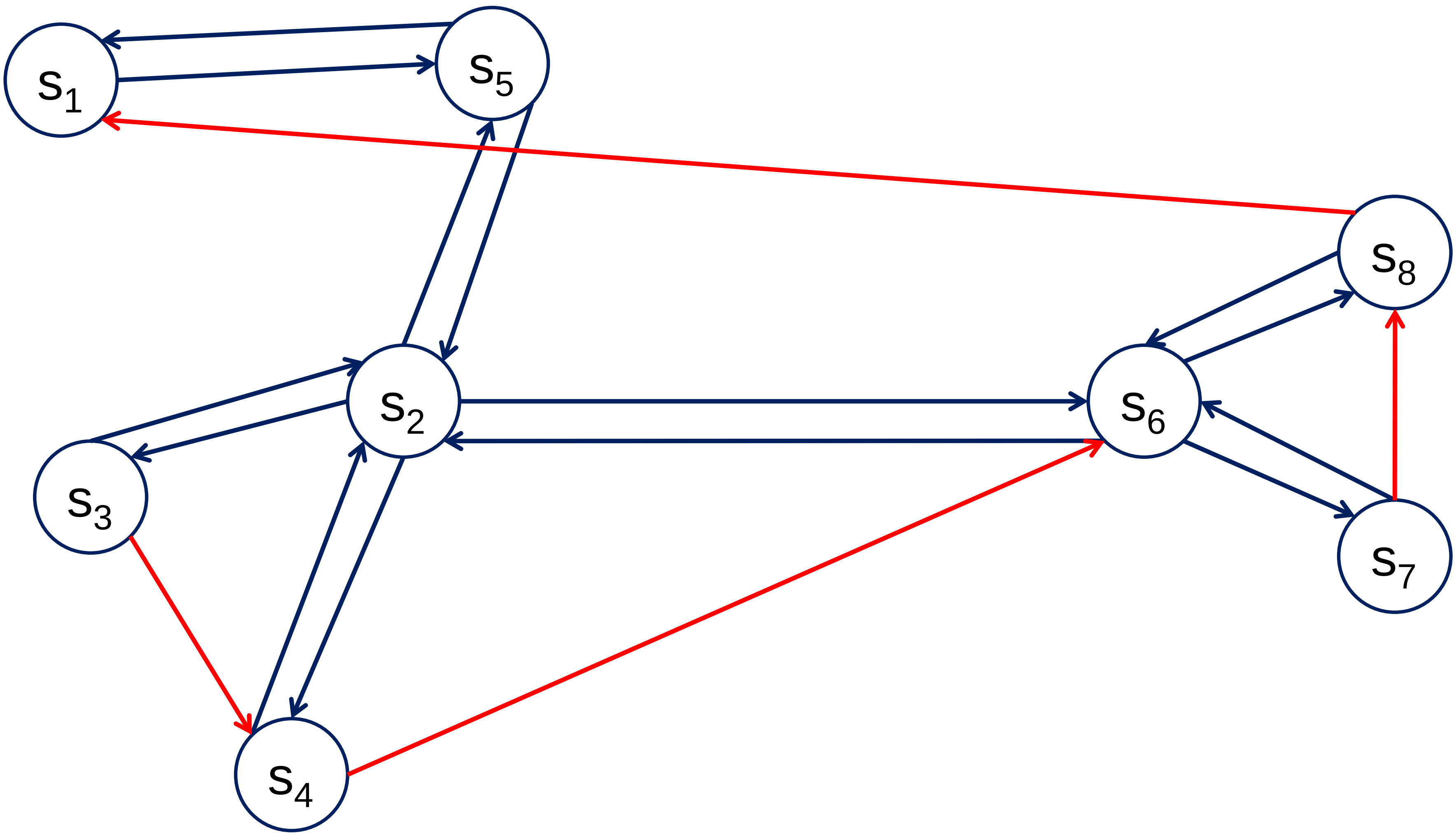
Approximation von ETSP - Spannbaum



Approximation von ETSP - Eulerkreis



Approximation von ETSP - Rundreise



Approximationsalgorithmus MSB

1. Konstruiere einen minimalen Spannbaum T auf den Punkten $s_1, \ldots, s_n$.
2. Konstruiere aus T einen Graph H, indem jede Kante in T verdoppelt wird.
3. Finde in H einen Eulerkreis K.
4. Berechne die Reihenfolge $s_{\pi(1)}, \ldots, s_{\pi(n)}$ des ersten Auftretens der Knoten $s_1, \ldots, s_n$ in K.
5. Gib $\pi = (\pi(1), \ldots, \pi(n))$ aus.

Satz 4.5 Algorithmus MSB hat Approximationsgüte 2.

Approximationsalgorithmus MSB

Satz 4.5 Algorithmus MSB hat Approximationsgüte 2.

Beweis

- Sei opt(I) die Länge einer optimalen Rundreise R.
- Sei w(T) das Gewicht eines minimalen Spannbaums.
- Durch Entfernung einer beliebigen Kante aus R wird R zu einem Spannbaum.
- Daher ist w(T) ≤ opt(I).
- Der Eulerkreis hat daher Gesamtlänge ≤ 2opt(I).
- Da die Euklidische Distanz eine Metrik ist, gilt die Dreiecksungleichung.
- Abkürzungen im Eulerkreis verkürzen daher die Länge.
- Daher ist MSB(I) ≤ 2opt(I).

Optimierungsproblem RS_{opt}

1. Instanzen: $I = \langle G,W,g\rangle$, $G = \{g_1,\ldots,g_n\}$, $W = \{w_1,\ldots,w_n\}$.

2. Zulässige Lösungen: $S \subseteq \{1,\ldots,n\}$ mit $\sum_{i \in S} g_i \leq g$.

3. Wert einer zulässigen Lösung: $w(S) = \sum_{i \in S} w_i$.

4. RS_{opt} ist Maximierungsproblem.

Erinnerung RS_{ent} ist NP-vollständig.

RS_{opt} und Dynamisches Programmieren

$F_j(i) :=$ minimales Gewicht einer Teilmenge der ersten j Gegenstände mit Wert mindestens i, $0 \le j \le n$, $i \in \mathbb{Z}$.

$F_j(i) = \infty$, falls solche Menge nicht existiert.

Lemma 4.6 Sei opt Wert einer optimalen Lösung. Dann gilt $opt = \max\{i \mid F_n(i) \le g\}$.

Rs_{opt} und Dynamisches Programmieren

Lemma 4.7

1. $F_j(i) = 0$ für $i \leq 0$.
2. $F_0(i) = \infty$ für $i > 0$.
3. $F_j(i) = \min\{F_{j-1}(i),\ g_j + F_{j-1}(i - w_j)\}$ für $i,j > 0$.

Objekt j wird nicht genommen.

Objekt j wird genommen.

Algorithmus ExactKnapsack

1. Setze $i := 0$.
2. Wiederhole die folgenden Schritte bis $F_n(i) > g$.
3. Setze $i := i + 1$.
4. Für $j = 1, \ldots, n$
5. Setze $F_j(i) = \min\{F_{j-1}(i), g_j + F_{j-1}(i - w_j)\}$.
6. Ausgabe $i-1$.

Algorithmus ExactKnapsack

j \ i	0	1	2	3	4	5	6	…
0	0	∞	∞	∞	∞	∞	∞	∞
1	0	*	*	*	*			
2	0	*	*	*	*			
3	0	*	*	*	$F_j(i)$			
4	0	*	*	*				
5	0	*	*	*				
6	…	*	*	*				

$$F_j(i) = \min\{F_{j-1}(i), g_j + F_{j-1}(i - w_j)\}$$

Algorithmus ExactKnapsack

j \ i	0	1	2	3	4	5	6	…
0	0	∞	∞	∞	∞	∞	∞	∞
1	0	*	*	*	*			
2	0	*	*	*	*			
3	0	*	*	*	$F_j(i)$			
4	0	*	*	*				
5	0	*	*	*				
…	0	*	*	*				

$F_{j-1}(i)$

$$F_j(i) = \min\{F_{j-1}(i),\ g_j + F_{j-1}(i - w_j)\}$$

Algorithmus ExactKnapsack

j \ i	0	1	2	3	4	5	6	...
0	0	∞	∞	∞	∞	∞	∞	∞
1	0	*	*	*	*			
2	0	*	*	*	*			
3	0	*	*	*	$F_j(i)$			
4	0	*	*	*				
5	0	*	*	*				
...	0	*	*	*				

$F_{j-1}(i - w_j)$

$F_{j-1}(i)$

$$F_j(i) = \min\left\{F_{j-1}(i),\ g_j + F_{j-1}(i - w_j)\right\}$$

Algorithmus ExactKnapsack

1. Setze $i := 0$.
2. Wiederhole die folgenden Schritte bis $F_n(i) > g$.
3. Setze $i := i + 1$.
4. Für $j = 1, \ldots, n$
5. Setze $F_j(i) = \min\{F_{j-1}(i), g_j + F_{j-1}(i - w_j)\}$.
6. Ausgabe $i-1$.

Satz 4.8 Algorithmus ExactKnapsack hat Laufzeit polynomiell in opt und polynomiell in der Eingabegröße.

Algorithmus ScaledKnapsack(ε)

1. Setze $i := 0$.
2. Wähle $k \in N$.
3. Für $j = 1,\ldots,n$, setze $w_j(k) := w_j / k$.
4. Berechne mit ExactKnapsack die optimale Lösung $opt(k)$ und die optimale Teilmenge $S(k)$ des Rucksackproblems mit Werten $w_j(k)$ und unveränderten Gewichten g_j und g.
5. Gib $opt^* := \sum_{j \in S(k)} w_j$ als Lösung aus.

Algorithmus ScaledKnapsack(ε)

1. Setze $i := 0$.
2. Wähle $k \in N$.
3. Für $j = 1,\ldots,n$, setze $w_j(k) := \lfloor w_j / k \rfloor$.
4. Berechne mit ExactKnapsack die optimale Lösung $opt(k)$ und die optimale Teilmenge $S(k)$ des Rucksackproblems mit Werten $w_j(k)$ und unveränderten Gewichten g_j und g.
5. Gib $opt^* := \sum_{j \in S(k)} w_j$ als Lösung aus.

Algorithmus ScaledKnapsack(ε)

1. Setze $i := 0$.
2. Setze $k := \max\left\{1, \left\lfloor \omega w_{max} / n \right\rfloor\right\}$.
3. Für $j = 1,\ldots,n$, setze $w_j(k) := \left\lfloor w_j / k \right\rfloor$.
4. Berechne mit ExactKnapsack die optimale Lösung $opt(k)$ und die optimale Teilmenge $S(k)$ des Rucksackproblems mit Werten $w_j(k)$ und unveränderten Gewichten g_j und g.
5. Gib $opt^* := \sum_{j \in S(k)} w_j$ als Lösung aus.

Algorithmus ScaledKnapsack(ε)

1. Setze $i := 0$.
2. Setze $k := \max\left\{1, \left\lfloor \varepsilon \cdot w_{max} / n \right\rfloor\right\}$.
3. Für $j = 1, \ldots, n$, setze $w_j(k) := \left\lfloor w_j / k \right\rfloor$.
4. Berechne mit ExactKnapsack die optimale Lösung $opt(k)$ und die optimale Teilmenge $S(k)$ des Rucksackproblems mit Werten $w_j(k)$ und unveränderten Gewichten g_j und g.
5. Gib $opt^* := \sum_{j \in S(k)} w_j$ als Lösung aus.

Satz 4.9 ScaledKnapsack(ε) hat Approximationsgüte $1 - \varepsilon$.

Algorithmus ScaledKnapsack(ε)

1. Setze $i := 0$.
2. Setze $k := \max\{1, \lfloor \varepsilon \cdot w_{max}/n \rfloor\}$.
3. Für $j = 1,\ldots,n$, setze $w_j(k) := \lfloor w_j/k \rfloor$.
4. Berechne mit ExactKnapsack die optimale Lösung $opt(k)$ und die optimale Teilmenge $S(k)$ des Rucksackproblems mit Werten $w_j(k)$ und unveränderten Gewichten g_j und g.
5. Gib $opt^* := \sum_{j \in S(k)} w_j$ als Lösung aus.

Satz 4.9 **ScaledKnapsack(ε) hat Approximationsgüte 1 –ε.**

Satz 4.10 **ScaledKnapsack(ε) hat Laufzeit polynomiell in der Eingabegröße und in 1/ε.**

Ein Unmöglichkeitsergebnis

Satz 4.11 Wenn es einen polynomiellen Approximationsalgorithmus A mit konstanter Approximationsgüte r für das allgemeine TSP_{opt} Problem gibt, dann ist P=NP.

Beweis

- Sei TSP[r] das Problem, TSP_{opt} mit Approximationsgüte r zu lösen.
- Zu zeigen: Hamilton $\leq_p$ TSP[r] für alle $r \in \mathbb{N}$.

Approximationsschemata

Definition 4.12 Ein **Approximationsschema** A für ein Optimierungsproblem Π ist ein Algorithmus, der bei Eingabe ε mit $0<\varepsilon<1$ das gegebene Problem mit Approximationsgüte

- $1-\varepsilon$ bei Maximierungsproblemen

beziehungsweise

- $1+\varepsilon$ bei Minimierungsproblemen

lösen kann.

- Ein Approximationsschema A ist ein **polynomielles Approximationsschema (PTAS)**, wenn A für jedes *konstante* ε eine Laufzeit hat, die polynomiell in der Größe der Instanz ist.
- Ein Approximationsschema A ist ein **streng polynomielles Approximationsschema (FPTAS)**, wenn A für jedes ε eine Laufzeit hat, die polynomiell in der Größe der Instanz *und* ε ist.

Approximationsschemata

- Ein Approximationsschema A ist ein **polynomielles Approximationsschema (PTAS)**, wenn A für jedes *konstante* ε eine Laufzeit hat, die polynomiell in der Größe der Instanz ist.
- Ein Approximationsschema A ist ein **streng polynomielles Approximationsschema (FPTAS)**, wenn A für jedes ε eine Laufzeit hat, die polynomiell in der Größe der Instanz *und* ε ist.

Beispiele für Laufzeiten:

- PTAS: $O(|I|^{1/\varepsilon})$ oder $O(2^{1/\varepsilon} \cdot |I|)$
- FPTAS: $O((1/\varepsilon)^2 \cdot |I|^3)$

Approximationsschemata

Aus Satz 4.10 folgt: Das Rucksackproblem hat ein FPTAS.

Warum fordern wir nicht auch eine Laufzeit, die polynomiell in $\log(1/\varepsilon)$ ist?

- **Angenommen, es gibt ein FPTAS mit Laufzeit $O(\text{poly}(|I|, \log(1/\varepsilon)))$ für ein diskretes Optimierungsproblem Π, dessen Entscheidungsvariante (ist $\text{opt}(I)\geq k$ bzw. $\text{opt}(I)\leq k$?) NP-vollständig ist.**

- **Dann kann damit jede Instanz $I'=\langle I,k\rangle$ der Entscheidungsvariante in polynomieller Zeit gelöst werden, da $|k|\leq|I'|$ ist und es daher ausreicht, $\varepsilon=1/2^{|I'|}$ zu setzen (so dass $\varepsilon<1/k$ ist) um herauszufinden, ob $\text{opt}(I)\geq k$ bzw. $\text{opt}(I)\leq k$ ist:**
  - **Denn ist $\text{opt}(I)\geq k$, dann ist $(1-\varepsilon)\text{opt}(I)>k-1$ (und damit die Ausgabe des FPTAS mindestens k), und sonst ist $\text{opt}(I)<k$ (und damit auch die Ausgabe des FPTAS kleiner als k).**
  - **Analoges gilt für die Frage „Ist $\text{opt}(I)\leq k$?“.**

- **D.h. wir hätten dann gezeigt, dass P=NP ist.**

Approximationsschemata

Als Konsequenz unserer Überlegungen folgt:

Satz 4.13 Sei Π ein diskretes Optimierungsproblem und sei für ein **festes** k die Entscheidungsvariante „Ist zur Eingabe I von Π der Wert opt(I)$\leq$k?“ falls Π ein Minimierungsproblem ist, bzw. „Ist zur Eingabe I von Π der Wert opt(I)$\geq$k?“ falls Π ein Maximierungs-problem ist, NP-vollständig. Gibt es ein PTAS für Π, dann ist P=NP.

Beweis: Es reicht, ε so zu setzen, dass $\varepsilon<1/k$ ist.

Approximationsschemata

Beobachtung:

Sei maxnr(I) die größte Zahl in einer Instanz I.

- **Ist maxnr(I) für eine Instanz des Rucksackproblems nur polynomiell groß in |I|, dann kann I in polynomieller Zeit gelöst werden.**
- **Für das Hamiltonkreis Problem ist maxnr(I)=1 für jede Instanz I (die Knoten und Kanten haben keine Gewichte), aber trotzdem ist das Problem NP-vollständig.**

Definition 4.14 **Ein NP-vollständiges Entscheidungsproblem L heißt stark NP-vollständig, wenn es ein Polynom q gibt, so dass $L_q = \{ I \in L \mid maxnr(I) \le q(|I|) \}$ NP-vollständig ist. Gibt es kein solches Polynom, dann heißt L schwach NP-vollständig.**

Approximationsschemata

Definition 4.14 Ein NP-vollständiges Entscheidungsproblem L heißt **stark NP-vollständig**, wenn es ein Polynom q gibt, so dass $L_q=\{ I \in L \mid maxnr(I) \leq q(|I|) \}$ NP-vollständig ist. Gibt es kein solches Polynom, dann heißt L **schwach NP-vollständig**.

Beispiele:

- Hamiltonkreis und Clique sind stark NP-vollständig.
- TSP_{ent} ist stark NP-vollständig.
- RS_{ent} ist schwach NP-vollständig.

Allgemein besteht eine enge Beziehung zwischen starker NP-Vollständigkeit und der Unmöglichkeit, ein (F)PTAS anzugeben.

Approximationsschemata

Satz 4.15 Sei Π ein diskretes Optimierungsproblem. Wenn es ein Polynom $q(x_1,x_2)$ gibt, so dass für alle Instanzen I gilt, dass $opt(I) \le q(|I|, maxnr(I))$ ist, dann folgt aus der Existenz eines FPTAS für Π, dass es einen pseudopolynomiellen exakten Algorithmus (d.h. einen Algorithmus mit Laufzeit $O(poly(|I|, maxnr(I)))$) für Π gibt.

Beweis: Wähle $\varepsilon < 1/q(|I|, maxnr(I))$.

Wenn $maxnr(I) = O(poly(|I|))$ ist, dann ergibt dieser Satz sogar eine polynomielle Laufzeit für den exakten Algorithmus, was uns zu folgender Aussage führt.

Satz 4.16 Wenn es für eine diskrete Optimierungsvariante eines stark NP-vollständigen Entscheidungsproblems ein FPTAS gibt, dann ist P=NP.

Zusammenfassung

- Laufzeit/Zeitkomplexität einer DTM
- Klassen für Zeitkomplexität
- Vergleich Zeitkomplexität 1-Band und Mehrband DTM
- Klasse P als Klasse effizient lösbarer Probleme
- Beispiele für Sprachen in P, Pfad, RelPrim
- RS_{ent}, $TSP_{ent} \in P$?

Zusammenfassung

- Verifizierbarkeit als gemeinsame Eigenschaft
- Verifizierer und Klasse NP
- Nichtdeterministische TMs (NTMs)
- L(N) für NTM N
- Laufzeit von NTMs
- Alternative Definition von NP über NTMs

Zusammenfassung

- Schwierigste Probleme in NP?
- Polynomielle Reduktionen
- NP-Vollständigkeit
- Satz von Cook-Levin, SAT NP-vollständig
- Weitere NP-vollständige Sprachen, 3SAT, Clique,…

Zusammenfassung

- Spezialfälle wie 2SAT
- Approximationsalgorithmen
- Approximationsfaktor/-güte
- Beispiele Max-Cut, TSP_{opt}, RS_{opt}
- Approximationsschemas
- Starke und schwache NP-Vollständigkeit