Exakte Algorithmen

Exakte Algorithmen

Für NP-harte Probleme haben wir bisher mit Hilfe von Approximationsalgorithmen versucht, gute Lösungen zu erhalten.

Exakte Algorithmen

Für NP-harte Probleme haben wir bisher mit Hilfe von Approximationsalgorithmen versucht, gute Lösungen zu erhalten. Jetzt bestehen wir auf optimalen Lösungen.

Exakte Algorithmen

Für NP-harte Probleme haben wir bisher mit Hilfe von Approximationsalgorithmen versucht, gute Lösungen zu erhalten. Jetzt bestehen wir auf optimalen Lösungen.

- Um herauszufinden, ob ein schwieriges Entscheidungsproblem eine Lösung besitzt, werden wir Backtracking einsetzen.

Exakte Algorithmen

Für NP-harte Probleme haben wir bisher mit Hilfe von Approximationsalgorithmen versucht, gute Lösungen zu erhalten. Jetzt bestehen wir auf optimalen Lösungen.

- Um herauszufinden, ob ein schwieriges Entscheidungsproblem eine Lösung besitzt, werden wir Backtracking einsetzen.
- Für die Lösung von Optimierungsproblemen benutzen wir die Branch & Bound und Branch & Cut Verfahren.

Exakte Algorithmen

Für NP-harte Probleme haben wir bisher mit Hilfe von Approximationsalgorithmen versucht, gute Lösungen zu erhalten. Jetzt bestehen wir auf optimalen Lösungen.

- Um herauszufinden, ob ein schwieriges Entscheidungsproblem eine Lösung besitzt, werden wir Backtracking einsetzen.
- Für die Lösung von Optimierungsproblemen benutzen wir die Branch & Bound und Branch & Cut Verfahren.
- Backtracking und Branch & Bound/Cut versuchen, den Lösungsraum intelligent zu durchsuchen.

Exakte Algorithmen

Für NP-harte Probleme haben wir bisher mit Hilfe von Approximationsalgorithmen versucht, gute Lösungen zu erhalten. Jetzt bestehen wir auf optimalen Lösungen.

- Um herauszufinden, ob ein schwieriges Entscheidungsproblem eine Lösung besitzt, werden wir Backtracking einsetzen.
- Für die Lösung von Optimierungsproblemen benutzen wir die Branch & Bound und Branch & Cut Verfahren.
- Backtracking und Branch & Bound/Cut versuchen, den Lösungsraum intelligent zu durchsuchen. Trotzdem müssen wir im Allgemeinen auf den massiven Einsatz von Rechnerressourcen gefasst sein.

Die Entdeckung weniger Ausreißer

- Wir führen n Tests aus, wobei die Ergebnisse einiger weniger Tests stark verfälscht sind.

Die Entdeckung weniger Ausreißer

- Wir führen n Tests aus, wobei die Ergebnisse einiger weniger Tests stark verfälscht sind.
- Wir bauen einen ungerichteten Graphen mit n Knoten und setzen eine Kante $\{i, j\}$ ein, wenn die Ergebnisse des $i$ten und $j$ten Test zu stark voneinander abweichen.

Die Entdeckung weniger Ausreißer

- Wir führen n Tests aus, wobei die Ergebnisse einiger weniger Tests stark verfälscht sind.
- Wir bauen einen ungerichteten Graphen mit n Knoten und setzen eine Kante $\{i, j\}$ ein, wenn die Ergebnisse des $i$ten und $j$ten Test zu stark voneinander abweichen.
  - Die wenigen stark verfälschten Tests „sollten“ dann einerseits einem kleinen Vertex Cover entsprechen und

Die Entdeckung weniger Ausreißer

- Wir führen n Tests aus, wobei die Ergebnisse einiger weniger Tests stark verfälscht sind.
- Wir bauen einen ungerichteten Graphen mit n Knoten und setzen eine Kante $\{i, j\}$ ein, wenn die Ergebnisse des $i$ten und $j$ten Test zu stark voneinander abweichen.
  - Die wenigen stark verfälschten Tests „sollten“ dann einerseits einem kleinen Vertex Cover entsprechen und
  - ein kleinstes Vertex Cover „sollte“ andererseits genau aus allen Ausreißern bestehen.

Die Entdeckung weniger Ausreißer

- Wir führen n Tests aus, wobei die Ergebnisse einiger weniger Tests stark verfälscht sind.
- Wir bauen einen ungerichteten Graphen mit n Knoten und setzen eine Kante $\{i, j\}$ ein, wenn die Ergebnisse des $i$ten und $j$ten Test zu stark voneinander abweichen.
  - Die wenigen stark verfälschten Tests „sollten“ dann einerseits einem kleinen Vertex Cover entsprechen und
  - ein kleinstes Vertex Cover „sollte“ andererseits genau aus allen Ausreißern bestehen.
- Löse das Vertex Cover Problem exakt, wenn kleine Überdeckungen existieren!

Wenn wir Zusatzwissen haben

- Wir betrachten das Vertex Cover Problem VC für einen ungerichteten Graphen $G = (V, E)$ mit n Knoten.

Wenn wir Zusatzwissen haben

- Wir betrachten das Vertex Cover Problem VC für einen ungerichteten Graphen $G = (V, E)$ mit n Knoten.
- Wir nehmen an, dass G ein Vertex Cover der (relativ kleinen) Größe höchstens k besitzt

Wenn wir Zusatzwissen haben

- Wir betrachten das Vertex Cover Problem VC für einen ungerichteten Graphen $G = (V, E)$ mit n Knoten.
- Wir nehmen an, dass G ein Vertex Cover der (relativ kleinen) Größe höchstens k besitzt und suchen nach exakten Algorithmen mit einer Laufzeit der Form

$$f(k) \cdot \text{poly}(n).$$

Wenn wir Zusatzwissen haben

- Wir betrachten das Vertex Cover Problem VC für einen ungerichteten Graphen $G = (V, E)$ mit n Knoten.
- Wir nehmen an, dass G ein Vertex Cover der (relativ kleinen) Größe höchstens k besitzt und suchen nach exakten Algorithmen mit einer Laufzeit der Form

$$f(k) \cdot \text{poly}(n).$$

- Es genügt eine Lösung des Entscheidungsproblems

„Hat G eine Überdeckung der Größe k?“

Wenn wir Zusatzwissen haben

- Wir betrachten das Vertex Cover Problem VC für einen ungerichteten Graphen $G = (V, E)$ mit n Knoten.
- Wir nehmen an, dass G ein Vertex Cover der (relativ kleinen) Größe höchstens k besitzt und suchen nach exakten Algorithmen mit einer Laufzeit der Form

$$f(k) \cdot \text{poly}(n).$$

- Es genügt eine Lösung des Entscheidungsproblems

„Hat G eine Überdeckung der Größe k?“

Durch Binärsuche können wir dann das Optimum mit $\log_2 k$-maliger Lösung des Entscheidungsproblems berechnen.

Wenn kleine Überdeckungen existieren

Wann hat G eine Überdeckung der Größe k?

Wenn kleine Überdeckungen existieren

Wann hat G eine Überdeckung der Größe k?
Nur wenn G nicht zu viele Kanten besitzt.

Wenn kleine Überdeckungen existieren

Wann hat G eine Überdeckung der Größe k?
Nur wenn G nicht zu viele Kanten besitzt.

Wenige Kanten bei kleinen Überdeckungen

Der Graph G habe n Knoten und einen Vertex Cover der Größe k.
Dann hat G höchstens $k \cdot n$ Kanten.

Wenn kleine Überdeckungen existieren

Wann hat G eine Überdeckung der Größe k?
Nur wenn G nicht zu viele Kanten besitzt.

Wenige Kanten bei kleinen Überdeckungen

Der Graph G habe n Knoten und einen Vertex Cover der Größe k.
Dann hat G höchstens $k \cdot n$ Kanten.

Warum?

Wenn kleine Überdeckungen existieren

Wann hat G eine Überdeckung der Größe k?
Nur wenn G nicht zu viele Kanten besitzt.

Wenige Kanten bei kleinen Überdeckungen

Der Graph G habe n Knoten und einen Vertex Cover der Größe k.
Dann hat G höchstens $k \cdot n$ Kanten.

Warum?

- Eine Überdeckung $\ddot{U} \subseteq V$ muss einen Endpunkt für jede Kante des Graphen besitzen.

Wenn kleine Überdeckungen existieren

Wann hat G eine Überdeckung der Größe k?
Nur wenn G nicht zu viele Kanten besitzt.

Wenige Kanten bei kleinen Überdeckungen

Der Graph G habe n Knoten und einen Vertex Cover der Größe k.
Dann hat G höchstens $k \cdot n$ Kanten.

Warum?

- Eine Überdeckung $\ddot{U} \subseteq V$ muss einen Endpunkt für jede Kante des Graphen besitzen.
- Ein Knoten kann aber nur Endpunkt von max. $n - 1$ Kanten sein.

Wenn kleine Überdeckungen existieren

Wann hat G eine Überdeckung der Größe k?
Nur wenn G nicht zu viele Kanten besitzt.

Wenige Kanten bei kleinen Überdeckungen

Der Graph G habe n Knoten und einen Vertex Cover der Größe k.
Dann hat G höchstens $k \cdot n$ Kanten.

Warum?

- Eine Überdeckung $\ddot{U} \subseteq V$ muss einen Endpunkt für jede Kante des Graphen besitzen.
- Ein Knoten kann aber nur Endpunkt von max. $n - 1$ Kanten sein.
- Also hat G höchstens $k \cdot (n-1) \leqslant k \cdot n$ Kanten.

Der Algorithmus VC(G, k)

(1) Setze $\ddot{U} = \emptyset$.

Der Algorithmus VC(G, k)

(1) Setze $\ddot{U} = \emptyset$.

(2) Wenn G mehr als $k \cdot n$ Kanten hat, dann hat G keine Überdeckung der Größe k.

Der Algorithmus VC(G, k)

(1) Setze $\ddot{U} = \emptyset$.

(2) Wenn G mehr als $k \cdot n$ Kanten hat, dann hat G keine Überdeckung der Größe k. Dann brich mit der Nachricht „erfolglos“ ab.

Der Algorithmus VC(G, k)

(1) Setze $\ddot{U} = \emptyset$.

(2) Wenn G mehr als $k \cdot n$ Kanten hat, dann hat G keine Überdeckung der Größe k. Dann brich mit der Nachricht „erfolglos“ ab.
Ansonsten, für $k = 0$, brich mit der Nachricht „erfolgreich“ ab.

Der Algorithmus VC(G, k)

(1) Setze $\ddot{U} = \emptyset$.

(2) Wenn G mehr als $k \cdot n$ Kanten hat, dann hat G keine Überdeckung der Größe k. Dann brich mit der Nachricht „erfolglos“ ab.
Ansonsten, für $k = 0$, brich mit der Nachricht „erfolgreich“ ab.
// G hat also jetzt höchstens $k \cdot n$ Kanten.

Der Algorithmus VC(G, k)

(1) Setze $\ddot{U} = \emptyset$.

(2) Wenn G mehr als $k \cdot n$ Kanten hat, dann hat G keine Überdeckung der Größe k. Dann brich mit der Nachricht „erfolglos“ ab.
Ansonsten, für $k = 0$, brich mit der Nachricht „erfolgreich“ ab.
// G hat also jetzt höchstens $k \cdot n$ Kanten.

(3) Wähle eine beliebige Kante $\{u, v\} \in E$.

Der Algorithmus VC(G, k)

(1) Setze $\ddot{U} = \emptyset$.

(2) Wenn G mehr als $k \cdot n$ Kanten hat, dann hat G keine Überdeckung der Größe k. Dann brich mit der Nachricht „erfolglos“ ab.
Ansonsten, für $k = 0$, brich mit der Nachricht „erfolgreich“ ab.
// G hat also jetzt höchstens $k \cdot n$ Kanten.

(3) Wähle eine beliebige Kante $\{u, v\} \in E$. Rufe **VC(G** $-$ **u**, **k** $-$ **1)** auf.

Der Algorithmus VC(G, k)

(1) Setze $\ddot{U} = \emptyset$.

(2) Wenn G mehr als $k \cdot n$ Kanten hat, dann hat G keine Überdeckung der Größe k. Dann brich mit der Nachricht „erfolglos“ ab.
Ansonsten, für $k = 0$, brich mit der Nachricht „erfolgreich“ ab.
// G hat also jetzt höchstens $k \cdot n$ Kanten.

(3) Wähle eine beliebige Kante $\{u, v\} \in E$. Rufe **VC(G − u, k − 1)** auf.

- Wenn die Antwort „erfolgreich“ ist,

Der Algorithmus VC(G, k)

(1) Setze $\ddot{U} = \emptyset$.

(2) Wenn G mehr als $k \cdot n$ Kanten hat, dann hat G keine Überdeckung der Größe k. Dann brich mit der Nachricht „erfolglos“ ab.
Ansonsten, für $k = 0$, brich mit der Nachricht „erfolgreich“ ab.
// G hat also jetzt höchstens $k \cdot n$ Kanten.

(3) Wähle eine beliebige Kante $\{u, v\} \in E$. Rufe **VC(G − u, k − 1)** auf.

- Wenn die Antwort „erfolgreich“ ist, dann setze $\ddot{U} = \ddot{U} \cup \{u\}$ und brich mit der Nachricht „erfolgreich“ ab.

Der Algorithmus VC(G, k)

(1) Setze $\ddot{U} = \emptyset$.

(2) Wenn G mehr als $k \cdot n$ Kanten hat, dann hat G keine Überdeckung der Größe k. Dann brich mit der Nachricht „erfolglos“ ab.
Ansonsten, für $k = 0$, brich mit der Nachricht „erfolgreich“ ab.
// G hat also jetzt höchstens $k \cdot n$ Kanten.

(3) Wähle eine beliebige Kante $\{u, v\} \in E$. Rufe **VC(G − u, k − 1)** auf.

- Wenn die Antwort „erfolgreich“ ist, dann setze $\ddot{U} = \ddot{U} \cup \{u\}$ und brich mit der Nachricht „erfolgreich“ ab.
- Ansonsten rufe **VC(G − v, k − 1)** auf.

Der Algorithmus VC(G, k)

(1) Setze $\ddot{U} = \emptyset$.

(2) Wenn G mehr als $k \cdot n$ Kanten hat, dann hat G keine Überdeckung der Größe k. Dann brich mit der Nachricht „erfolglos“ ab.
Ansonsten, für $k = 0$, brich mit der Nachricht „erfolgreich“ ab.
// G hat also jetzt höchstens $k \cdot n$ Kanten.

(3) Wähle eine beliebige Kante $\{u, v\} \in E$. Rufe **VC(G − u, k − 1)** auf.

- Wenn die Antwort „erfolgreich“ ist, dann setze $\ddot{U} = \ddot{U} \cup \{u\}$ und brich mit der Nachricht „erfolgreich“ ab.
- Ansonsten rufe **VC(G − v, k − 1)** auf. Wenn die Antwort „erfolgreich“ ist,

Der Algorithmus VC(G, k)

(1) Setze $\ddot{U} = \emptyset$.

(2) Wenn G mehr als $k \cdot n$ Kanten hat, dann hat G keine Überdeckung der Größe k. Dann brich mit der Nachricht „erfolglos“ ab.
Ansonsten, für $k = 0$, brich mit der Nachricht „erfolgreich“ ab.
// G hat also jetzt höchstens $k \cdot n$ Kanten.

(3) Wähle eine beliebige Kante $\{u, v\} \in E$. Rufe **VC(G − u, k − 1)** auf.

- Wenn die Antwort „erfolgreich“ ist, dann setze $\ddot{U} = \ddot{U} \cup \{u\}$ und brich mit der Nachricht „erfolgreich“ ab.
- Ansonsten rufe **VC(G − v, k − 1)** auf. Wenn die Antwort „erfolgreich“ ist, dann setze $\ddot{U} = \ddot{U} \cup \{v\}$ und brich mit der Nachricht „erfolgreich“ ab.

Der Algorithmus VC(G, k)

(1) Setze $\ddot{U} = \emptyset$.

(2) Wenn G mehr als $k \cdot n$ Kanten hat, dann hat G keine Überdeckung der Größe k. Dann brich mit der Nachricht „erfolglos“ ab.
Ansonsten, für $k = 0$, brich mit der Nachricht „erfolgreich“ ab.
// G hat also jetzt höchstens $k \cdot n$ Kanten.

(3) Wähle eine beliebige Kante $\{u, v\} \in E$. Rufe **VC(G − u, k − 1)** auf.

- Wenn die Antwort „erfolgreich“ ist, dann setze $\ddot{U} = \ddot{U} \cup \{u\}$ und brich mit der Nachricht „erfolgreich“ ab.
- Ansonsten rufe **VC(G − v, k − 1)** auf. Wenn die Antwort „erfolgreich“ ist, dann setze $\ddot{U} = \ddot{U} \cup \{v\}$ und brich mit der Nachricht „erfolgreich“ ab.
- Wenn beide Aufrufe erfolglos: Brich mit der Nachricht „erfolglos“ ab.

Der Algorithmus VC(G, k)

(1) Setze $\ddot{U} = \emptyset$.

(2) Wenn G mehr als $k \cdot n$ Kanten hat, dann hat G keine Überdeckung der Größe k. Dann brich mit der Nachricht „erfolglos“ ab.
Ansonsten, für $k = 0$, brich mit der Nachricht „erfolgreich“ ab.
// G hat also jetzt höchstens $k \cdot n$ Kanten.

(3) Wähle eine beliebige Kante $\{u, v\} \in E$. Rufe **VC(G − u, k − 1)** auf.

- Wenn die Antwort „erfolgreich“ ist, dann setze $\ddot{U} = \ddot{U} \cup \{u\}$ und brich mit der Nachricht „erfolgreich“ ab.
- Ansonsten rufe **VC(G − v, k − 1)** auf. Wenn die Antwort „erfolgreich“ ist, dann setze $\ddot{U} = \ddot{U} \cup \{v\}$ und brich mit der Nachricht „erfolgreich“ ab.
- Wenn beide Aufrufe erfolglos: Brich mit der Nachricht „erfolglos“ ab.
// Die Überdeckung $\ddot{U}$ muss einen Endpunkt der Kante $\{u, v\}$
// besitzen. Zuerst wird die Option $u \in \ddot{U}$ rekursiv untersucht,

Der Algorithmus VC(G, k)

(1) Setze $\ddot{U} = \emptyset$.

(2) Wenn G mehr als $k \cdot n$ Kanten hat, dann hat G keine Überdeckung der Größe k. Dann brich mit der Nachricht „erfolglos“ ab.
Ansonsten, für $k = 0$, brich mit der Nachricht „erfolgreich“ ab.
// G hat also jetzt höchstens $k \cdot n$ Kanten.

(3) Wähle eine beliebige Kante $\{u, v\} \in E$. Rufe **VC(G − u, k − 1)** auf.

- Wenn die Antwort „erfolgreich“ ist, dann setze $\ddot{U} = \ddot{U} \cup \{u\}$ und brich mit der Nachricht „erfolgreich“ ab.
- Ansonsten rufe **VC(G − v, k − 1)** auf. Wenn die Antwort „erfolgreich“ ist, dann setze $\ddot{U} = \ddot{U} \cup \{v\}$ und brich mit der Nachricht „erfolgreich“ ab.
- Wenn beide Aufrufe erfolglos: Brich mit der Nachricht „erfolglos“ ab.
// Die Überdeckung $\ddot{U}$ muss einen Endpunkt der Kante $\{u, v\}$
// besitzen. Zuerst wird die Option $u \in \ddot{U}$ rekursiv untersucht, und
// bei Misserfolg auch die Option $v \in \ddot{U}$.

Laufzeit-Analyse

- Der Algorithmus erzeugt mit seinem ersten Aufruf VC(G, k) einen Rekursionsbaum.

Laufzeit-Analyse

- Der Algorithmus erzeugt mit seinem ersten Aufruf VC(G, k) einen Rekursionsbaum.
- Der Rekursionsbaum ist binär und hat Tiefe k.

Laufzeit-Analyse

- Der Algorithmus erzeugt mit seinem ersten Aufruf VC(G, k) einen Rekursionsbaum.
- Der Rekursionsbaum ist binär und hat Tiefe k. Es gibt höchstens 2^k rekursive Aufrufe mit Zeit höchstens $O(n)$ pro Aufruf.

Laufzeit-Analyse

- Der Algorithmus erzeugt mit seinem ersten Aufruf VC(G, k) einen Rekursionsbaum.
- Der Rekursionsbaum ist binär und hat Tiefe k. Es gibt höchstens 2^k rekursive Aufrufe mit Zeit höchstens $O(n)$ pro Aufruf.

Der Algorithmus überprüft in Zeit $O(2^k \cdot n)$, ob ein Graph mit n Knoten einen Vertex Cover der Größe höchstens k besitzt.

Laufzeit-Analyse

- Der Algorithmus erzeugt mit seinem ersten Aufruf $\mathrm{VC}(G, k)$ einen Rekursionsbaum.
- Der Rekursionsbaum ist binär und hat Tiefe k. Es gibt höchstens 2^k rekursive Aufrufe mit Zeit höchstens $O(n)$ pro Aufruf.

Der Algorithmus überprüft in Zeit $O(2^k \cdot n)$, ob ein Graph mit n Knoten einen Vertex Cover der Größe höchstens k besitzt.

Der Algorithmus scheint nur mäßig intelligent zu sein,

Laufzeit-Analyse

- Der Algorithmus erzeugt mit seinem ersten Aufruf VC(G, k) einen Rekursionsbaum.
- Der Rekursionsbaum ist binär und hat Tiefe k. Es gibt höchstens 2^k rekursive Aufrufe mit Zeit höchstens $O(n)$ pro Aufruf.

Der Algorithmus überprüft in Zeit $O(2^k \cdot n)$, ob ein Graph mit n Knoten einen Vertex Cover der Größe höchstens k besitzt.

Der Algorithmus scheint nur mäßig intelligent zu sein, aber er ist dennoch wesentlich schneller als die Überprüfung aller $\binom{n}{k}$ möglichen k-elementigen Teilmengen,

Laufzeit-Analyse

- Der Algorithmus erzeugt mit seinem ersten Aufruf VC(G, k) einen Rekursionsbaum.
- Der Rekursionsbaum ist binär und hat Tiefe k. Es gibt höchstens 2^k rekursive Aufrufe mit Zeit höchstens $O(n)$ pro Aufruf.

Der Algorithmus überprüft in Zeit $O(2^k \cdot n)$, ob ein Graph mit n Knoten einen Vertex Cover der Größe höchstens k besitzt.

Der Algorithmus scheint nur mäßig intelligent zu sein, aber er ist dennoch wesentlich schneller als die Überprüfung aller $\binom{n}{k}$ möglichen k-elementigen Teilmengen, falls k nicht zu groß ist.

Laufzeit-Analyse

- Der Algorithmus erzeugt mit seinem ersten Aufruf VC(G, k) einen Rekursionsbaum.
- Der Rekursionsbaum ist binär und hat Tiefe k. Es gibt höchstens 2^k rekursive Aufrufe mit Zeit höchstens $O(n)$ pro Aufruf.

Der Algorithmus überprüft in Zeit $O(2^k \cdot n)$, ob ein Graph mit n Knoten einen Vertex Cover der Größe höchstens k besitzt.

Der Algorithmus scheint nur mäßig intelligent zu sein, aber er ist dennoch wesentlich schneller als die Überprüfung aller $\binom{n}{k}$ möglichen k-elementigen Teilmengen, falls k nicht zu groß ist.

Man bezeichnet das Forschungsgebiet von Problemen mit fixierten Parametern auch als parametrisierte Komplexität.

Backtracking

Das Ziel: Löse ein Entscheidungsproblem.

Backtracking

Das Ziel: Löse ein Entscheidungsproblem.

- Wir möchten feststellen, ob sich in der Menge U aller potentiellen Lösungen

Backtracking

Das Ziel: Löse ein Entscheidungsproblem.

- Wir möchten feststellen, ob sich in der Menge U aller potentiellen Lösungen eine tatsächliche Lösungen befindet.

Backtracking

Das Ziel: Löse ein Entscheidungsproblem.

- Wir möchten feststellen, ob sich in der Menge U aller potentiellen Lösungen eine tatsächliche Lösungen befindet.
- Baue Lösungen schrittweise aus partiellen Lösungen auf:

Backtracking

Das Ziel: Löse ein Entscheidungsproblem.

- Wir möchten feststellen, ob sich in der Menge U aller potentiellen Lösungen eine tatsächliche Lösungen befindet.
- Baue Lösungen schrittweise aus partiellen Lösungen auf: Eine partielle, noch nicht vollständig entwickelte Lösung ist eine Menge potentieller Lösungen.

Backtracking

Das Ziel: Löse ein Entscheidungsproblem.

- Wir möchten feststellen, ob sich in der Menge U aller potentiellen Lösungen eine tatsächliche Lösungen befindet.
- Baue Lösungen schrittweise aus partiellen Lösungen auf: Eine partielle, noch nicht vollständig entwickelte Lösung ist eine Menge potentieller Lösungen.
- Zum Beispiel für eine KNF-Formel ϕ

Backtracking

Das Ziel: Löse ein Entscheidungsproblem.

- Wir möchten feststellen, ob sich in der Menge U aller potentiellen Lösungen eine tatsächliche Lösungen befindet.
- Baue Lösungen schrittweise aus partiellen Lösungen auf: Eine partielle, noch nicht vollständig entwickelte Lösung ist eine Menge potentieller Lösungen.
- Zum Beispiel für eine KNF-Formel ϕ entsprechen Belegungen den potentiellen Lösungen,

Backtracking

Das Ziel: Löse ein Entscheidungsproblem.

- Wir möchten feststellen, ob sich in der Menge U aller potentiellen Lösungen eine tatsächliche Lösungen befindet.
- Baue Lösungen schrittweise aus partiellen Lösungen auf: Eine partielle, noch nicht vollständig entwickelte Lösung ist eine Menge potentieller Lösungen.
- Zum Beispiel für eine KNF-Formel ϕ entsprechen Belegungen den potentiellen Lösungen, erfüllende Belegungen den tatsächlichen Lösungen

Backtracking

Das Ziel: Löse ein Entscheidungsproblem.

- Wir möchten feststellen, ob sich in der Menge U aller potentiellen Lösungen eine tatsächliche Lösungen befindet.
- Baue Lösungen schrittweise aus partiellen Lösungen auf: Eine partielle, noch nicht vollständig entwickelte Lösung ist eine Menge potentieller Lösungen.
- Zum Beispiel für eine KNF-Formel ϕ entsprechen Belegungen den potentiellen Lösungen, erfüllende Belegungen den tatsächlichen Lösungen und partielle Belegungen den partiellen Lösungen.

Backtracking

Das Ziel: Löse ein Entscheidungsproblem.

- Wir möchten feststellen, ob sich in der Menge U aller potentiellen Lösungen eine tatsächliche Lösungen befindet.
- Baue Lösungen schrittweise aus partiellen Lösungen auf: Eine partielle, noch nicht vollständig entwickelte Lösung ist eine Menge potentieller Lösungen.
- Zum Beispiel für eine KNF-Formel ϕ entsprechen Belegungen den potentiellen Lösungen, erfüllende Belegungen den tatsächlichen Lösungen und partielle Belegungen den partiellen Lösungen.
- Backtracking ist eine Suche auf dem Raum aller potentiellen Lösungen,

Backtracking

Das Ziel: Löse ein Entscheidungsproblem.

- Wir möchten feststellen, ob sich in der Menge U aller potentiellen Lösungen eine tatsächliche Lösungen befindet.
- Baue Lösungen schrittweise aus partiellen Lösungen auf: Eine partielle, noch nicht vollständig entwickelte Lösung ist eine Menge potentieller Lösungen.
- Zum Beispiel für eine KNF-Formel ϕ entsprechen Belegungen den potentiellen Lösungen, erfüllende Belegungen den tatsächlichen Lösungen und partielle Belegungen den partiellen Lösungen.
- Backtracking ist eine Suche auf dem Raum aller potentiellen Lösungen, die frühzeitig erkennt, wenn eine partielle Lösung sich nicht in eine tatsächliche Lösung erweitern lässt.

Backtracking: Der Branching Operator B

- Wie organisiert Backtracking die Suche im Raum U der potentiellen Lösungen?

Backtracking: Der Branching Operator B

- Wie organisiert Backtracking die Suche im Raum U der potentiellen Lösungen?
- Backtracking arbeitet mit einem Branching-Operator B, der U in mehreren Schritten zerlegt:

Backtracking: Der Branching Operator B

- Wie organisiert Backtracking die Suche im Raum U der potentiellen Lösungen?
- Backtracking arbeitet mit einem Branching-Operator B, der U in mehreren Schritten zerlegt:
  - Wenn wir bereits die Teilmenge $v \subseteq U$ erhalten haben, dann beschreibt $B(v)$ eine Zerlegung $v = \bigcup_j v_j$ von v.

Backtracking: Der Branching Operator B

- Wie organisiert Backtracking die Suche im Raum U der potentiellen Lösungen?
- Backtracking arbeitet mit einem Branching-Operator B, der U in mehreren Schritten zerlegt:
  - Wenn wir bereits die Teilmenge $v \subseteq U$ erhalten haben, dann beschreibt $B(v)$ eine Zerlegung $v = \bigcup_j v_j$ von v.
  - Die partielle Lösung v wird zu den partiellen Lösungen v_j erweitert.

Backtracking: Der Branching Operator B

- Wie organisiert Backtracking die Suche im Raum U der potentiellen Lösungen?
- Backtracking arbeitet mit einem Branching-Operator B, der U in mehreren Schritten zerlegt:
  - Wenn wir bereits die Teilmenge $v \subseteq U$ erhalten haben, dann beschreibt $B(v)$ eine Zerlegung $v = \bigcup_j v_j$ von v.
  - Die partielle Lösung v wird zu den partiellen Lösungen v_j erweitert.
  - Wenn v eine tatsächliche Lösung enthält, dann enthält auch mindestens ein Kind v_j eine tatsächlichen Lösung,

Backtracking: Der Branching Operator B

- Wie organisiert Backtracking die Suche im Raum U der potentiellen Lösungen?
- Backtracking arbeitet mit einem Branching-Operator B, der U in mehreren Schritten zerlegt:
  - Wenn wir bereits die Teilmenge $v \subseteq U$ erhalten haben, dann beschreibt $B(v)$ eine Zerlegung $v = \bigcup_j v_j$ von v.
  - Die partielle Lösung v wird zu den partiellen Lösungen v_j erweitert.
  - Wenn v eine tatsächliche Lösung enthält, dann enthält auch mindestens ein Kind v_j eine tatsächlichen Lösung, denn der Branching-Operator bestimmt eine Zerlegung von v.

Der Backtracking Baum $\mathcal{B}$

Der Branching-Operator definiert einen Backtracking Baum $\mathcal{B}$.

Der Backtracking Baum $\mathcal{B}$

Der Branching-Operator definiert einen Backtracking Baum $\mathcal{B}$.

- Anfänglich besteht $\mathcal{B}$ nur aus der Wurzel $v = U$.

Der Backtracking Baum $\mathcal{B}$

Der Branching-Operator definiert einen Backtracking Baum $\mathcal{B}$.

- Anfänglich besteht $\mathcal{B}$ nur aus der Wurzel $v = U$.
- Wenn v ein Blatt ist und die Menge v nicht ein-elementig ist,

Der Backtracking Baum $\mathcal{B}$

Der Branching-Operator definiert einen Backtracking Baum $\mathcal{B}$.

- Anfänglich besteht $\mathcal{B}$ nur aus der Wurzel $v = U$.
- Wenn v ein Blatt ist und die Menge v nicht ein-elementig ist, dann erhält v die durch $B(v)$ beschriebenen Teilmengen $v_i \subseteq v$ als Kinder.

Der Backtracking Baum $\mathcal{B}$

Der Branching-Operator definiert einen Backtracking Baum $\mathcal{B}$.

- Anfänglich besteht $\mathcal{B}$ nur aus der Wurzel $v = U$.
- Wenn v ein Blatt ist und die Menge v nicht ein-elementig ist, dann erhält v die durch $B(v)$ beschriebenen Teilmengen $v_i \subseteq v$ als Kinder.
- Ein-elementige Mengen werden nicht expandiert:

Der Backtracking Baum $\mathcal{B}$

Der Branching-Operator definiert einen Backtracking Baum $\mathcal{B}$.

- Anfänglich besteht $\mathcal{B}$ nur aus der Wurzel $v = U$.
- Wenn v ein Blatt ist und die Menge v nicht ein-elementig ist, dann erhält v die durch $B(v)$ beschriebenen Teilmengen $v_i \subseteq v$ als Kinder.
- Ein-elementige Mengen werden nicht expandiert:
  - Ein Blatt von $\mathcal{B}$ entspricht einer potentiellen Lösung,

Der Backtracking Baum $\mathcal{B}$

Der Branching-Operator definiert einen Backtracking Baum $\mathcal{B}$.

- Anfänglich besteht $\mathcal{B}$ nur aus der Wurzel $v = U$.
- Wenn v ein Blatt ist und die Menge v nicht ein-elementig ist, dann erhält v die durch $B(v)$ beschriebenen Teilmengen $v_i \subseteq v$ als Kinder.
- Ein-elementige Mengen werden nicht expandiert:
  - Ein Blatt von $\mathcal{B}$ entspricht einer potentiellen Lösung,
  - während ein innerer Knoten einer partiellen Lösungen,

Der Backtracking Baum $\mathcal{B}$

Der Branching-Operator definiert einen Backtracking Baum $\mathcal{B}$.

- Anfänglich besteht $\mathcal{B}$ nur aus der Wurzel $v = U$.
- Wenn v ein Blatt ist und die Menge v nicht ein-elementig ist, dann erhält v die durch $B(v)$ beschriebenen Teilmengen $v_i \subseteq v$ als Kinder.
- Ein-elementige Mengen werden nicht expandiert:
  - Ein Blatt von $\mathcal{B}$ entspricht einer potentiellen Lösung,
  - während ein innerer Knoten einer partiellen Lösungen, also einer Menge potentieller Lösungen entspricht.

Backtracking: Der Algorithmus

(1) Anfänglich besteht der Backtracking Baum nur aus der (nicht disqualifizierten) Wurzel v.

Backtracking: Der Algorithmus

(1) Anfänglich besteht der Backtracking Baum nur aus der (nicht disqualifizierten) Wurzel v.
(2) Wiederhole, solange es nicht disqualifizierte Blätter gibt

Backtracking: Der Algorithmus

(1) Anfänglich besteht der Backtracking Baum nur aus der (nicht disqualifizierten) Wurzel v.

(2) Wiederhole, solange es nicht disqualifizierte Blätter gibt
   - Wähle das erfolgversprechendste Blatt v.

Backtracking: Der Algorithmus

(1) Anfänglich besteht der Backtracking Baum nur aus der (nicht disqualifizierten) Wurzel v.

(2) Wiederhole, solange es nicht disqualifizierte Blätter gibt
   - Wähle das erfolgversprechendste Blatt v.
   - Wende den Branching Operator auf v an.

Backtracking: Der Algorithmus

(1) Anfänglich besteht der Backtracking Baum nur aus der (nicht disqualifizierten) Wurzel v.

(2) Wiederhole, solange es nicht disqualifizierte Blätter gibt
   - Wähle das erfolgversprechendste Blatt v.
   - Wende den Branching Operator auf v an. Wir erhalten die Kinder $v_1, \ldots, v_k$.

Backtracking: Der Algorithmus

(1) Anfänglich besteht der Backtracking Baum nur aus der (nicht disqualifizierten) Wurzel v.

(2) Wiederhole, solange es nicht disqualifizierte Blätter gibt
- Wähle das erfolgversprechendste Blatt v.
- Wende den Branching Operator auf v an. Wir erhalten die Kinder $v_1, \ldots, v_k$.
- Ein Kind v_i wird disqualifiziert, wenn v_i definitiv keine tatsächliche Lösungen enthält.

Backtracking: Der Algorithmus

(1) Anfänglich besteht der Backtracking Baum nur aus der (nicht disqualifizierten) Wurzel v.

(2) Wiederhole, solange es nicht disqualifizierte Blätter gibt
- Wähle das erfolgversprechendste Blatt v.
- Wende den Branching Operator auf v an. Wir erhalten die Kinder $v_1, \ldots, v_k$.
- Ein Kind v_i wird disqualifiziert, wenn v_i definitiv keine tatsächliche Lösungen enthält.
- Sollte allerdings in v_i eine tatsächliche Lösung gefunden werden, dann wird der Algorithmus mit einer Erfolgsmeldung abgebrochen.

Backtracking: Der Algorithmus

(1) Anfänglich besteht der Backtracking Baum nur aus der (nicht disqualifizierten) Wurzel v.

(2) Wiederhole, solange es nicht disqualifizierte Blätter gibt
   - Wähle das erfolgversprechendste Blatt v.
   - Wende den Branching Operator auf v an. Wir erhalten die Kinder $v_1, \ldots, v_k$.
   - Ein Kind v_i wird disqualifiziert, wenn v_i definitiv keine tatsächliche Lösungen enthält.
   - Sollte allerdings in v_i eine tatsächliche Lösung gefunden werden, dann wird der Algorithmus mit einer Erfolgsmeldung abgebrochen.

(3) Brich mit einer Erfolglos-Meldung ab.

Backtracking: Worauf ist zu achten?

- Wie definiert man den Branching Operator B?

Backtracking: Worauf ist zu achten?

- Wie definiert man den Branching Operator B?
- Was ist ein erfolgversprechendstes Blatt?

Backtracking: Worauf ist zu achten?

- Wie definiert man den Branching Operator B?
- Was ist ein erfolgversprechendstes Blatt?
- Wie entdeckt man, dass ein Knoten disqualifiziert werden kann?

Backtracking: Worauf ist zu achten?

- Wie definiert man den Branching Operator B?
- Was ist ein erfolgversprechendstes Blatt?
- Wie entdeckt man, dass ein Knoten disqualifiziert werden kann?

Damit Backtracking mehr als eine einfache Suche im Raum aller potentiellen Lösungen ist,

Backtracking: Worauf ist zu achten?

- Wie definiert man den Branching Operator B?
- Was ist ein erfolgversprechendstes Blatt?
- Wie entdeckt man, dass ein Knoten disqualifiziert werden kann?

Damit Backtracking mehr als eine einfache Suche im Raum aller potentiellen Lösungen ist, muss wertvolle Suchzeit durch Disqualifikation eingespart werden!

Backtracking für KNF-SAT

α sei eine KNF-Formel.

Backtracking für KNF-SAT

α sei eine KNF-Formel.
Bestimme eine erfüllende Belegung, wenn α erfüllbar ist.

Backtracking für KNF-SAT

α sei eine KNF-Formel.
Bestimme eine erfüllende Belegung, wenn α erfüllbar ist.

- Das Universum $U = \{0, 1\}^n$ ist die Menge aller Belegungen.

Backtracking für KNF-SAT

α sei eine KNF-Formel.
Bestimme eine erfüllende Belegung, wenn α erfüllbar ist.

- Das Universum $U = \{0,1\}^n$ ist die Menge aller Belegungen.
- Wir beschreiben einen Knoten v des Backtracking Baums $\mathcal{B}$ durch das Paar (J, b) mit $J \subseteq \{1, \ldots, n\}$ und $b : J \to \{0,1\}$:

Backtracking für KNF-SAT

α sei eine KNF-Formel.
Bestimme eine erfüllende Belegung, wenn α erfüllbar ist.

- Das Universum $U = \{0,1\}^n$ ist die Menge aller Belegungen.
- Wir beschreiben einen Knoten v des Backtracking Baums $\mathcal{B}$ durch das Paar (J, b) mit $J \subseteq \{1, \ldots, n\}$ und $b : J \to \{0,1\}$:

  In v werden alle partiellen Belegungen $x \in \{0,1,*\}^n$ mit $x_j = b(j)$ für alle $j \in J$ und mit $x_j = *$ für $j \notin J$ erfasst.

Backtracking für KNF-SAT

α sei eine KNF-Formel.
Bestimme eine erfüllende Belegung, wenn α erfüllbar ist.

- Das Universum $U = \{0,1\}^n$ ist die Menge aller Belegungen.
- Wir beschreiben einen Knoten v des Backtracking Baums $\mathcal{B}$ durch das Paar (J, b) mit $J \subseteq \{1, \ldots, n\}$ und $b : J \to \{0,1\}$:

  In v werden alle partiellen Belegungen $x \in \{0,1,*\}^n$ mit $x_j = b(j)$ für alle $j \in J$ und mit $x_j = *$ für $j \notin J$ erfasst.
- Wann wird ein Knoten v mit Beschriftung (J, b) disqualifiziert?

Backtracking für KNF-SAT

α sei eine KNF-Formel.
Bestimme eine erfüllende Belegung, wenn α erfüllbar ist.

- Das Universum $U = \{0,1\}^n$ ist die Menge aller Belegungen.
- Wir beschreiben einen Knoten v des Backtracking Baums $\mathcal{B}$ durch das Paar (J, b) mit $J \subseteq \{1, \ldots, n\}$ und $b : J \to \{0,1\}$:

  In v werden alle partiellen Belegungen $x \in \{0,1,*\}^n$ mit $x_j = b(j)$ für alle $j \in J$ und mit $x_j = *$ für $j \notin J$ erfasst.
- Wann wird ein Knoten v mit Beschriftung (J, b) disqualifiziert?
  - Wir setzen die durch (J, b) definierten Wahrheitswerte ein und suchen nach einelementigen Klauseln,

Backtracking für KNF-SAT

α sei eine KNF-Formel.
Bestimme eine erfüllende Belegung, wenn α erfüllbar ist.

- Das Universum $U = \{0,1\}^n$ ist die Menge aller Belegungen.
- Wir beschreiben einen Knoten v des Backtracking Baums $\mathcal{B}$ durch das Paar (J, b) mit $J \subseteq \{1, \ldots, n\}$ und $b : J \to \{0,1\}$:

  In v werden alle partiellen Belegungen $x \in \{0,1,*\}^n$ mit $x_j = b(j)$ für alle $j \in J$ und mit $x_j = *$ für $j \notin J$ erfasst.
- Wann wird ein Knoten v mit Beschriftung (J, b) disqualifiziert?
  - Wir setzen die durch (J, b) definierten Wahrheitswerte ein und suchen nach einelementigen Klauseln, die dann den Wert „ihrer“ Variablen festlegen.

Backtracking für KNF-SAT

α sei eine KNF-Formel.
Bestimme eine erfüllende Belegung, wenn α erfüllbar ist.

- Das Universum $U = \{0,1\}^n$ ist die Menge aller Belegungen.
- Wir beschreiben einen Knoten v des Backtracking Baums $\mathcal{B}$ durch das Paar (J, b) mit $J \subseteq \{1, \ldots, n\}$ und $b : J \to \{0,1\}$:

  In v werden alle partiellen Belegungen $x \in \{0,1,*\}^n$ mit $x_j = b(j)$ für alle $j \in J$ und mit $x_j = *$ für $j \notin J$ erfasst.
- Wann wird ein Knoten v mit Beschriftung (J, b) disqualifiziert?
  - Wir setzen die durch (J, b) definierten Wahrheitswerte ein und suchen nach einelementigen Klauseln, die dann den Wert „ihrer“ Variablen festlegen.
  - Wir setzen die „erzwungenen“ Wahrheitswerte in α ein und wiederholen dieses Vorgehen, bis

Backtracking für KNF-SAT

α sei eine KNF-Formel.
Bestimme eine erfüllende Belegung, wenn α erfüllbar ist.

- Das Universum $U = \{0,1\}^n$ ist die Menge aller Belegungen.
- Wir beschreiben einen Knoten v des Backtracking Baums $\mathcal{B}$ durch das Paar (J, b) mit $J \subseteq \{1, \ldots, n\}$ und $b : J \to \{0,1\}$:

  In v werden alle partiellen Belegungen $x \in \{0,1,*\}^n$ mit $x_j = b(j)$ für alle $j \in J$ und mit $x_j = *$ für $j \notin J$ erfasst.
- Wann wird ein Knoten v mit Beschriftung (J, b) disqualifiziert?
  - Wir setzen die durch (J, b) definierten Wahrheitswerte ein und suchen nach einelementigen Klauseln, die dann den Wert „ihrer“ Variablen festlegen.
  - Wir setzen die „erzwungenen“ Wahrheitswerte in α ein und wiederholen dieses Vorgehen, bis wir einen Widerspruch gefunden haben

Backtracking für KNF-SAT

α sei eine KNF-Formel.
Bestimme eine erfüllende Belegung, wenn α erfüllbar ist.

- Das Universum $U = \{0,1\}^n$ ist die Menge aller Belegungen.
- Wir beschreiben einen Knoten v des Backtracking Baums $\mathcal{B}$ durch das Paar (J, b) mit $J \subseteq \{1, \ldots, n\}$ und $b : J \to \{0,1\}$:

  In v werden alle partiellen Belegungen $x \in \{0,1,*\}^n$ mit $x_j = b(j)$ für alle $j \in J$ und mit $x_j = *$ für $j \notin J$ erfasst.
- Wann wird ein Knoten v mit Beschriftung (J, b) disqualifiziert?
  - Wir setzen die durch (J, b) definierten Wahrheitswerte ein und suchen nach einelementigen Klauseln, die dann den Wert „ihrer“ Variablen festlegen.
  - Wir setzen die „erzwungenen“ Wahrheitswerte in α ein und wiederholen dieses Vorgehen, bis wir einen Widerspruch gefunden haben (und v wird disqualifiziert)

Backtracking für KNF-SAT

α sei eine KNF-Formel.
Bestimme eine erfüllende Belegung, wenn α erfüllbar ist.

- Das Universum $U = \{0,1\}^n$ ist die Menge aller Belegungen.
- Wir beschreiben einen Knoten v des Backtracking Baums $\mathcal{B}$ durch das Paar (J, b) mit $J \subseteq \{1, \ldots, n\}$ und $b : J \to \{0,1\}$:

  In v werden alle partiellen Belegungen $x \in \{0,1,*\}^n$ mit $x_j = b(j)$ für alle $j \in J$ und mit $x_j = *$ für $j \notin J$ erfasst.
- Wann wird ein Knoten v mit Beschriftung (J, b) disqualifiziert?
  - Wir setzen die durch (J, b) definierten Wahrheitswerte ein und suchen nach einelementigen Klauseln, die dann den Wert „ihrer“ Variablen festlegen.
  - Wir setzen die „erzwungenen“ Wahrheitswerte in α ein und wiederholen dieses Vorgehen, bis wir einen Widerspruch gefunden haben (und v wird disqualifiziert) oder bis alle verbliebenen Klauseln mindestens zwei-elementig sind.

Der Branching Operator für KNF-SAT

Die Grundidee

Formeln mit kurzen Klauseln können am ehesten als Sackgassen,

Der Branching Operator für KNF-SAT

Die Grundidee

Formeln mit kurzen Klauseln können am ehesten als Sackgassen, also als nicht erfüllbar erkannt werden.

Der Branching Operator für KNF-SAT

Die Grundidee

Formeln mit kurzen Klauseln können am ehesten als Sackgassen, also als nicht erfüllbar erkannt werden.

- Wir müssen den Branching Operator für ein beliebiges Blatt $v = (J, b)$ definieren.

Der Branching Operator für KNF-SAT

Die Grundidee

Formeln mit kurzen Klauseln können am ehesten als Sackgassen, also als nicht erfüllbar erkannt werden.

- Wir müssen den Branching Operator für ein beliebiges Blatt $v = (J, b)$ definieren.
  - Dazu setzen wir die durch (J, b) definierten Wahrheitswerte in α ein und

Der Branching Operator für KNF-SAT

Die Grundidee

Formeln mit kurzen Klauseln können am ehesten als Sackgassen, also als nicht erfüllbar erkannt werden.

- Wir müssen den Branching Operator für ein beliebiges Blatt $v = (J, b)$ definieren.
  - Dazu setzen wir die durch (J, b) definierten Wahrheitswerte in α ein und wählen eine kürzeste Klausel k.

Der Branching Operator für KNF-SAT

Die Grundidee

Formeln mit kurzen Klauseln können am ehesten als Sackgassen, also als nicht erfüllbar erkannt werden.

- Wir müssen den Branching Operator für ein beliebiges Blatt $v = (J, b)$ definieren.
  - Dazu setzen wir die durch (J, b) definierten Wahrheitswerte in α ein und wählen eine kürzeste Klausel k.
  - Wir wählen eine beliebige Variable x_i in k und erzeugen die beiden Kinder zu den zusätzlichen Belegungen $x_i = 0$ und $x_i = 1$.

Der Branching Operator für KNF-SAT

Die Grundidee

Formeln mit kurzen Klauseln können am ehesten als Sackgassen, also als nicht erfüllbar erkannt werden.

- Wir müssen den Branching Operator für ein beliebiges Blatt $v = (J, b)$ definieren.
  - Dazu setzen wir die durch (J, b) definierten Wahrheitswerte in α ein und wählen eine kürzeste Klausel k.
  - Wir wählen eine beliebige Variable x_i in k und erzeugen die beiden Kinder zu den zusätzlichen Belegungen $x_i = 0$ und $x_i = 1$.
- Das erfolgversprechendste Blatt ist ein nicht disqualifiziertes Blatt mit einer kürzesten Klausel.

Backtracking für Subset Sum

Subset Sum:

Backtracking für Subset Sum

Subset Sum: Zahlen $t_1, \ldots, t_n \in \mathbb{N}$ und ein Zielwert $Z \in \mathbb{N}$ sind gegeben.

Backtracking für Subset Sum

Subset Sum: Zahlen $t_1, \ldots, t_n \in \mathbb{N}$ und ein Zielwert $Z \in \mathbb{N}$ sind gegeben. Es ist festzustellen, ob es eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} t_i = Z$ gibt.

Backtracking für Subset Sum

Subset Sum: Zahlen $t_1, \ldots, t_n \in \mathbb{N}$ und ein Zielwert $Z \in \mathbb{N}$ sind gegeben. Es ist festzustellen, ob es eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} t_i = Z$ gibt.

- Wir sortieren die Zahlen absteigend und können annehmen, dass $t_1 \geqslant \cdots \geqslant t_n$ gilt.

Backtracking für Subset Sum

Subset Sum: Zahlen $t_1, \ldots, t_n \in \mathbb{N}$ und ein Zielwert $Z \in \mathbb{N}$ sind gegeben. Es ist festzustellen, ob es eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} t_i = Z$ gibt.

- Wir sortieren die Zahlen absteigend und können annehmen, dass $t_1 \geqslant \cdots \geqslant t_n$ gilt.
- Wir beschreiben einen Knoten v des Backtracking Baums durch ein Paar (I', i) mit $I' \subseteq \{1, \ldots, i\}$:

Backtracking für Subset Sum

Subset Sum: Zahlen $t_1, \ldots, t_n \in \mathbb{N}$ und ein Zielwert $Z \in \mathbb{N}$ sind gegeben. Es ist festzustellen, ob es eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} t_i = Z$ gibt.

- Wir sortieren die Zahlen absteigend und können annehmen, dass $t_1 \geqslant \cdots \geqslant t_n$ gilt.
- Wir beschreiben einen Knoten v des Backtracking Baums durch ein Paar (I', i) mit $I' \subseteq \{1, \ldots, i\}$: Bisher wurden alle Zahlen t_j mit $j \in I'$ aufgenommen.

Backtracking für Subset Sum

Subset Sum: Zahlen $t_1, \ldots, t_n \in \mathbb{N}$ und ein Zielwert $Z \in \mathbb{N}$ sind gegeben. Es ist festzustellen, ob es eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} t_i = Z$ gibt.

- Wir sortieren die Zahlen absteigend und können annehmen, dass $t_1 \geqslant \cdots \geqslant t_n$ gilt.
- Wir beschreiben einen Knoten v des Backtracking Baums durch ein Paar (I', i) mit $I' \subseteq \{1, \ldots, i\}$: Bisher wurden alle Zahlen t_j mit $j \in I'$ aufgenommen.
- v wird disqualifiziert,

Backtracking für Subset Sum

Subset Sum: Zahlen $t_1, \ldots, t_n \in \mathbb{N}$ und ein Zielwert $Z \in \mathbb{N}$ sind gegeben. Es ist festzustellen, ob es eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} t_i = Z$ gibt.

- Wir sortieren die Zahlen absteigend und können annehmen, dass $t_1 \geqslant \cdots \geqslant t_n$ gilt.
- Wir beschreiben einen Knoten v des Backtracking Baums durch ein Paar (I', i) mit $I' \subseteq \{1, \ldots, i\}$: Bisher wurden alle Zahlen t_j mit $j \in I'$ aufgenommen.
- v wird disqualifiziert,
  - falls der Zielwert „überschossen“ wird

Backtracking für Subset Sum

Subset Sum: Zahlen $t_1, \ldots, t_n \in \mathbb{N}$ und ein Zielwert $Z \in \mathbb{N}$ sind gegeben. Es ist festzustellen, ob es eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} t_i = Z$ gibt.

- Wir sortieren die Zahlen absteigend und können annehmen, dass $t_1 \geqslant \cdots \geqslant t_n$ gilt.
- Wir beschreiben einen Knoten v des Backtracking Baums durch ein Paar (I', i) mit $I' \subseteq \{1, \ldots, i\}$: Bisher wurden alle Zahlen t_j mit $j \in I'$ aufgenommen.
- v wird disqualifiziert,
  - falls der Zielwert „überschossen“ wird ($\sum_{j \in I'} t_j > Z$) oder

Backtracking für Subset Sum

Subset Sum: Zahlen $t_1, \ldots, t_n \in \mathbb{N}$ und ein Zielwert $Z \in \mathbb{N}$ sind gegeben. Es ist festzustellen, ob es eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} t_i = Z$ gibt.

- Wir sortieren die Zahlen absteigend und können annehmen, dass $t_1 \geqslant \cdots \geqslant t_n$ gilt.
- Wir beschreiben einen Knoten v des Backtracking Baums durch ein Paar (I', i) mit $I' \subseteq \{1, \ldots, i\}$: Bisher wurden alle Zahlen t_j mit $j \in I'$ aufgenommen.
- v wird disqualifiziert,
  - falls der Zielwert „überschossen“ wird ($\sum_{j \in I'} t_j > Z$) oder
  - falls der Zielwert nicht mehr erreichbar ist

Backtracking für Subset Sum

Subset Sum: Zahlen $t_1, \ldots, t_n \in \mathbb{N}$ und ein Zielwert $Z \in \mathbb{N}$ sind gegeben. Es ist festzustellen, ob es eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} t_i = Z$ gibt.

- Wir sortieren die Zahlen absteigend und können annehmen, dass $t_1 \geqslant \cdots \geqslant t_n$ gilt.
- Wir beschreiben einen Knoten v des Backtracking Baums durch ein Paar (I', i) mit $I' \subseteq \{1, \ldots, i\}$: Bisher wurden alle Zahlen t_j mit $j \in I'$ aufgenommen.
- v wird disqualifiziert,
  - falls der Zielwert „überschossen“ wird ($\sum_{j \in I'} t_j > Z$) oder
  - falls der Zielwert nicht mehr erreichbar ist ($\sum_{j \in I'} t_j + \sum_{j=i+1}^{n} t_j < Z$).

Backtracking für Subset Sum

Subset Sum: Zahlen $t_1, \ldots, t_n \in \mathbb{N}$ und ein Zielwert $Z \in \mathbb{N}$ sind gegeben. Es ist festzustellen, ob es eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} t_i = Z$ gibt.

- Wir sortieren die Zahlen absteigend und können annehmen, dass $t_1 \geqslant \cdots \geqslant t_n$ gilt.
- Wir beschreiben einen Knoten v des Backtracking Baums durch ein Paar (I', i) mit $I' \subseteq \{1, \ldots, i\}$: Bisher wurden alle Zahlen t_j mit $j \in I'$ aufgenommen.
- v wird disqualifiziert,
  - falls der Zielwert „überschossen“ wird ($\sum_{j \in I'} t_j > Z$) oder
  - falls der Zielwert nicht mehr erreichbar ist ($\sum_{j \in I'} t_j + \sum_{j=i+1}^{n} t_j < Z$).
- Die absteigende Sortierung erleichtert die Disqualifikation.

Der Branching Operator für Subset Sum

Sei v ein Blatt mit Beschreibung (l', i).

Der Branching Operator für Subset Sum

Sei v ein Blatt mit Beschreibung (l', i).

- Der Branching Operator fügt die beiden Kinder mit den Beschreibungen

Der Branching Operator für Subset Sum

Sei v ein Blatt mit Beschreibung (I', i).

- Der Branching Operator fügt die beiden Kinder mit den Beschreibungen

$$(I' \cup \{i+1\}, i+1) \text{ und } (I', i+1)$$

ein:

Der Branching Operator für Subset Sum

Sei v ein Blatt mit Beschreibung (I', i).

- Der Branching Operator fügt die beiden Kinder mit den Beschreibungen

$$(I' \cup \{i+1\}, i+1) \text{ und } (I', i+1)$$

ein: Entweder wird die Zahl t_{i+1} aufgenommen oder nicht.

Der Branching Operator für Subset Sum

Sei v ein Blatt mit Beschreibung (I', i).

- Der Branching Operator fügt die beiden Kinder mit den Beschreibungen

$$(I' \cup \{i+1\}, i+1) \text{ und } (I', i+1)$$

ein: Entweder wird die Zahl t_{i+1} aufgenommen oder nicht.

- Wir wählen das Blatt (I', i), das am weitesten festgelegt ist, als erfolgversprechendstes Blatt.

Der Branching Operator für Subset Sum

Sei v ein Blatt mit Beschreibung (I', i).

- Der Branching Operator fügt die beiden Kinder mit den Beschreibungen

$$(I' \cup \{i+1\}, i+1) \text{ und } (I', i+1)$$

ein: Entweder wird die Zahl t_{i+1} aufgenommen oder nicht.

- Wir wählen das Blatt (I', i), das am weitesten festgelegt ist, als erfolgversprechendstes Blatt.
Damit durchsucht Backtracking den Baum nach dem Tiefensuche-Verfahren.

Branch & Bound: Der Branching-Operator B

Unser Ziel: Löse das allgemeine Minimierungsproblem

$$\text{minimiere } f(x), \text{ so dass Lösung}(x).$$

Branch & Bound: Der Branching-Operator B

Unser Ziel: Löse das allgemeine Minimierungsproblem

$$\text{minimiere } f(x), \text{ so dass Lösung}(x).$$

- B zerlegt eine Menge von Lösungen in disjunkte Teilmengen.

Branch & Bound: Der Branching-Operator B

Unser Ziel: Löse das allgemeine Minimierungsproblem

$$\text{minimiere } f(x), \text{ so dass Lösung}(x).$$

- B zerlegt eine Menge von Lösungen in disjunkte Teilmengen.
- Die wiederholte Anwendung des Branching Operators erzeugt dann einen Branch & Bound Baum $\mathcal{B}$:

Branch & Bound: Der Branching-Operator B

Unser Ziel: Löse das allgemeine Minimierungsproblem

$$\text{minimiere } f(x), \text{ so dass Lösung}(x).$$

- B zerlegt eine Menge von Lösungen in disjunkte Teilmengen.
- Die wiederholte Anwendung des Branching Operators erzeugt dann einen Branch & Bound Baum $\mathcal{B}$:
  - Die Wurzel von $\mathcal{B}$ entspricht der Menge aller Lösungen.
  - Ist v ein Knoten von $\mathcal{B}$ mit Lösungsmenge $\mathcal{L}(v)$, dann hat v die Kinder $v_1, \ldots, v_k$, deren Lösungsmengen $\mathcal{L}(v_i)$ die Lösungsmenge $\mathcal{L}(v)$ disjunkt zerlegen.

Branch & Bound: Der Branching-Operator B

Unser Ziel: Löse das allgemeine Minimierungsproblem

minimiere $f(x)$, so dass Lösung(x).

- B zerlegt eine Menge von Lösungen in disjunkte Teilmengen.
- Die wiederholte Anwendung des Branching Operators erzeugt dann einen Branch & Bound Baum $\mathcal{B}$:
  - Die Wurzel von $\mathcal{B}$ entspricht der Menge aller Lösungen.
  - Ist v ein Knoten von $\mathcal{B}$ mit Lösungsmenge $\mathcal{L}(v)$, dann hat v die Kinder $v_1, \ldots, v_k$, deren Lösungsmengen $\mathcal{L}(v_i)$ die Lösungsmenge $\mathcal{L}(v)$ disjunkt zerlegen.
- Der Baum $\mathcal{B}$ ist viel zu groß: Sein Wachstum muss eingeschränkt werden.

Der Bounding-Schritt

Für den Bounding-Schritt wird eine untere Schranke benötigt:

- Für jeden Knoten v von $\mathcal{B}$ nehmen wir an, dass eine untere Schranke `unten(v)` gegeben ist, so dass

$$\texttt{unten}(v) \leqslant f(y)$$

für jede Lösung $y \in \mathcal{L}(v)$ gilt.

Der Bounding-Schritt

Für den Bounding-Schritt wird eine untere Schranke benötigt:

- Für jeden Knoten v von $\mathcal{B}$ nehmen wir an, dass eine untere Schranke `unten(v)` gegeben ist, so dass

$$\texttt{unten}(v) \leqslant f(y)$$

  für jede Lösung $y \in \mathcal{L}(v)$ gilt.
- Wann kann der Knoten v „abgeschnitten“ werden?

  Wenn $\texttt{unten}(v) \geqslant f(y_0)$ für eine aktuelle beste Lösung y_0 gilt!

Der Bounding-Schritt

Für den Bounding-Schritt wird eine untere Schranke benötigt:

- Für jeden Knoten v von $\mathcal{B}$ nehmen wir an, dass eine untere Schranke `unten(v)` gegeben ist, so dass

$$\texttt{unten}(v) \leqslant f(y)$$

  für jede Lösung $y \in \mathcal{L}(v)$ gilt.
- Wann kann der Knoten v „abgeschnitten“ werden?

  Wenn $\texttt{unten}(v) \geqslant f(y_0)$ für eine aktuelle beste Lösung y_0 gilt!
- Neben den unteren Schranken benötigen wir also auch eine Heuristik, die eine gute Lösung y_0 berechnet.

Branch & Bound: Der Algorithmus

(1) Eine Lösung y_0 wird mit Hilfe einer **Heuristik** berechnet.

Branch & Bound: Der Algorithmus

(1) Eine Lösung y_0 wird mit Hilfe einer **Heuristik** berechnet.
(2) Wiederhole, solange es aktivierte Blätter gibt:
  (2a) Wähle das **erfolgversprechendste** aktivierte Blatt v von $\mathcal{B}$.

Branch & Bound: Der Algorithmus

(1) Eine Lösung y_0 wird mit Hilfe einer **Heuristik** berechnet.

(2) Wiederhole, solange es aktivierte Blätter gibt:

  (2a) Wähle das **erfolgversprechendste** aktivierte Blatt v von $\mathcal{B}$.

  (2b) **Branching:** Wende den Branching Operator B auf v an, um die Kinder $v_1, \ldots, v_k$ zu erhalten. Inspiziere die k Teilmengen nacheinander:

Branch & Bound: Der Algorithmus

(1) Eine Lösung y_0 wird mit Hilfe einer **Heuristik** berechnet.

(2) Wiederhole, solange es aktivierte Blätter gibt:

  (2a) Wähle das **erfolgversprechendste** aktivierte Blatt v von $\mathcal{B}$.

  (2b) **Branching:** Wende den Branching Operator B auf v an, um die Kinder $v_1, \ldots, v_k$ zu erhalten. Inspiziere die k Teilmengen nacheinander:

  - ⋆ Wenn es offensichtlich ist, dass v_i eine Lösung y_i enthält, die besser als y_0 ist, dann setze $y_0 = y_i$ und deaktiviere gegebenenfalls Blätter.

Branch & Bound: Der Algorithmus

(1) Eine Lösung y_0 wird mit Hilfe einer **Heuristik** berechnet.

(2) Wiederhole, solange es aktivierte Blätter gibt:

  (2a) Wähle das **erfolgversprechendste** aktivierte Blatt v von $\mathcal{B}$.

  (2b) **Branching:** Wende den Branching Operator B auf v an, um die Kinder $v_1, \ldots, v_k$ zu erhalten. Inspiziere die k Teilmengen nacheinander:

  - ⋆ Wenn es offensichtlich ist, dass v_i eine Lösung y_i enthält, die besser als y_0 ist, dann setze $y_0 = y_i$ und deaktiviere gegebenenfalls Blätter.
  - ⋆ Ansonsten führe den **Bounding-Schritt** durch: Nur wenn $\texttt{unten}(v_i) < f(y_0)$, wird v_i aktiviert.

Branch & Bound: Der Algorithmus

(1) Eine Lösung y_0 wird mit Hilfe einer **Heuristik** berechnet.

(2) Wiederhole, solange es aktivierte Blätter gibt:

  (2a) Wähle das **erfolgversprechendste** aktivierte Blatt v von $\mathcal{B}$.

  (2b) **Branching:** Wende den Branching Operator B auf v an, um die Kinder $v_1, \ldots, v_k$ zu erhalten. Inspiziere die k Teilmengen nacheinander:

  - ⋆ Wenn es offensichtlich ist, dass v_i eine Lösung y_i enthält, die besser als y_0 ist, dann setze $y_0 = y_i$ und deaktiviere gegebenenfalls Blätter.
  - ⋆ Ansonsten führe den **Bounding-Schritt** durch: Nur wenn $\texttt{unten}(v_i) < f(y_0)$, wird v_i aktiviert.

(3) Gib die Lösung y_0 als optimale Lösung aus.

Was ist zu beachten?

- Damit der Bounding-Schritt erfolgreich ist,
  - muss die Anfangslösung y_0 möglichst nahe am Optimum liegen

Was ist zu beachten?

- Damit der Bounding-Schritt erfolgreich ist,
  - muss die Anfangslösung y_0 möglichst nahe am Optimum liegen
  - und die untere Schranke unten(v) muss die Qualität der besten Lösung in v möglichst gut voraussagen.

Was ist zu beachten?

- Damit der Bounding-Schritt erfolgreich ist,
  - muss die Anfangslösung y_0 möglichst nahe am Optimum liegen
  - und die untere Schranke unten(v) muss die Qualität der besten Lösung in v möglichst gut voraussagen.
- Die Wahl eines erfolgversprechendsten Blatts bestimmt die Suche in $\mathcal{B}$ und damit den Speicherplatzverbrauch.

Was ist zu beachten?

- Damit der Bounding-Schritt erfolgreich ist,
  - muss die Anfangslösung y_0 möglichst nahe am Optimum liegen
  - und die untere Schranke unten(v) muss die Qualität der besten Lösung in v möglichst gut voraussagen.
- Die Wahl eines erfolgversprechendsten Blatts bestimmt die Suche in $\mathcal{B}$ und damit den Speicherplatzverbrauch.
  - Tiefensuche schont den Speicherplatzverbrauch. Allerdings wird die schnelle Entdeckung guter Lösungen leiden, da stets mit dem letzten, und nicht mit dem besten Knoten gearbeitet wird.

Was ist zu beachten?

- Damit der Bounding-Schritt erfolgreich ist,
  - muss die Anfangslösung y_0 möglichst nahe am Optimum liegen
  - und die untere Schranke unten(v) muss die Qualität der besten Lösung in v möglichst gut voraussagen.
- Die Wahl eines erfolgversprechendsten Blatts bestimmt die Suche in $\mathcal{B}$ und damit den Speicherplatzverbrauch.
  - Tiefensuche schont den Speicherplatzverbrauch. Allerdings wird die schnelle Entdeckung guter Lösungen leiden, da stets mit dem letzten, und nicht mit dem besten Knoten gearbeitet wird.
  - Der große Speicherverbrauch schließt Breitensuche als ein praktikables Suchverfahren für große Bäume aus.

Was ist zu beachten?

- Damit der Bounding-Schritt erfolgreich ist,
  - muss die Anfangslösung y_0 möglichst nahe am Optimum liegen
  - und die untere Schranke unten(v) muss die Qualität der besten Lösung in v möglichst gut voraussagen.
- Die Wahl eines erfolgversprechendsten Blatts bestimmt die Suche in $\mathcal{B}$ und damit den Speicherplatzverbrauch.
  - Tiefensuche schont den Speicherplatzverbrauch. Allerdings wird die schnelle Entdeckung guter Lösungen leiden, da stets mit dem letzten, und nicht mit dem besten Knoten gearbeitet wird.
  - Der große Speicherverbrauch schließt Breitensuche als ein praktikables Suchverfahren für große Bäume aus.
  - In der „best first search“ wird der Knoten v mit der niedrigsten unteren Schranke gewählt. Man versucht also, schnell gute Lösungen zu erhalten.

Was ist zu beachten?

- Damit der Bounding-Schritt erfolgreich ist,
  - muss die Anfangslösung y_0 möglichst nahe am Optimum liegen
  - und die untere Schranke unten(v) muss die Qualität der besten Lösung in v möglichst gut voraussagen.
- Die Wahl eines erfolgversprechendsten Blatts bestimmt die Suche in $\mathcal{B}$ und damit den Speicherplatzverbrauch.
  - Tiefensuche schont den Speicherplatzverbrauch. Allerdings wird die schnelle Entdeckung guter Lösungen leiden, da stets mit dem letzten, und nicht mit dem besten Knoten gearbeitet wird.
  - Der große Speicherverbrauch schließt Breitensuche als ein praktikables Suchverfahren für große Bäume aus.
  - In der „best first search“ wird der Knoten v mit der niedrigsten unteren Schranke gewählt. Man versucht also, schnell gute Lösungen zu erhalten.
  - Häufig werden Varianten der Tiefensuche und der best-first search kombiniert.

Branch & Bound für das Rucksackproblem

Eine möglichst wertvolle Auswahl von n Objekten mit Gewichten $g_1, \ldots, g_n \in \mathbb{R}$ und Werten $w_1, \ldots, w_n \in \mathbb{N}$ ist in einen Rucksack mit Gewichtsschranke $G \in \mathbb{R}$ zu packen.

Branch & Bound für das Rucksackproblem

Eine möglichst wertvolle Auswahl von n Objekten mit Gewichten $g_1, \ldots, g_n \in \mathbb{R}$ und Werten $w_1, \ldots, w_n \in \mathbb{N}$ ist in einen Rucksack mit Gewichtsschranke $G \in \mathbb{R}$ zu packen.

Ein Greedy Algorithmus löst das fraktionale Rucksackproblem exakt.

Branch & Bound für das Rucksackproblem

Eine möglichst wertvolle Auswahl von n Objekten mit Gewichten $g_1, \ldots, g_n \in \mathbb{R}$ und Werten $w_1, \ldots, w_n \in \mathbb{N}$ ist in einen Rucksack mit Gewichtsschranke $G \in \mathbb{R}$ zu packen.

Ein Greedy Algorithmus löst das fraktionale Rucksackproblem exakt.
(Hier dürfen Anteile $0 \leqslant x_i \leqslant 1$ des $i$ten Objekts eingepackt werden.)

Branch & Bound für das Rucksackproblem

Eine möglichst wertvolle Auswahl von n Objekten mit Gewichten $g_1, \ldots, g_n \in \mathbb{R}$ und Werten $w_1, \ldots, w_n \in \mathbb{N}$ ist in einen Rucksack mit Gewichtsschranke $G \in \mathbb{R}$ zu packen.

Ein Greedy Algorithmus löst das fraktionale Rucksackproblem exakt. (Hier dürfen Anteile $0 \leqslant x_i \leqslant 1$ des $i$ten Objekts eingepackt werden.)

- Zuerst werden die Objekte absteigend, nach ihrem „Wert pro Kilo“ sortiert,

Branch & Bound für das Rucksackproblem

Eine möglichst wertvolle Auswahl von n Objekten mit Gewichten $g_1, \ldots, g_n \in \mathbb{R}$ und Werten $w_1, \ldots, w_n \in \mathbb{N}$ ist in einen Rucksack mit Gewichtsschranke $G \in \mathbb{R}$ zu packen.

Ein Greedy Algorithmus löst das fraktionale Rucksackproblem exakt. (Hier dürfen Anteile $0 \leqslant x_i \leqslant 1$ des $i$ten Objekts eingepackt werden.)

- Zuerst werden die Objekte absteigend, nach ihrem „Wert pro Kilo“ sortiert, also nach

$$\frac{w_1}{g_1} \geqslant \frac{w_2}{g_2} \geqslant \cdots \geqslant \frac{w_n}{g_n}.$$

Branch & Bound für das Rucksackproblem

Eine möglichst wertvolle Auswahl von n Objekten mit Gewichten $g_1, \ldots, g_n \in \mathbb{R}$ und Werten $w_1, \ldots, w_n \in \mathbb{N}$ ist in einen Rucksack mit Gewichtsschranke $G \in \mathbb{R}$ zu packen.

Ein Greedy Algorithmus löst das fraktionale Rucksackproblem exakt. (Hier dürfen Anteile $0 \leqslant x_i \leqslant 1$ des $i$ten Objekts eingepackt werden.)

- Zuerst werden die Objekte absteigend, nach ihrem „Wert pro Kilo“ sortiert, also nach

$$\frac{w_1}{g_1} \geqslant \frac{w_2}{g_2} \geqslant \cdots \geqslant \frac{w_n}{g_n}.$$

- Wenn $\sum_{i=1}^{k} g_i \leqslant G < \sum_{i=1}^{k+1} g_i$:

Branch & Bound für das Rucksackproblem

Eine möglichst wertvolle Auswahl von n Objekten mit Gewichten $g_1, \ldots, g_n \in \mathbb{R}$ und Werten $w_1, \ldots, w_n \in \mathbb{N}$ ist in einen Rucksack mit Gewichtsschranke $G \in \mathbb{R}$ zu packen.

Ein Greedy Algorithmus löst das fraktionale Rucksackproblem exakt. (Hier dürfen Anteile $0 \leqslant x_i \leqslant 1$ des $i$ten Objekts eingepackt werden.)

- Zuerst werden die Objekte absteigend, nach ihrem „Wert pro Kilo“ sortiert, also nach

$$\frac{w_1}{g_1} \geqslant \frac{w_2}{g_2} \geqslant \cdots \geqslant \frac{w_n}{g_n}.$$

- Wenn $\sum_{i=1}^{k} g_i \leqslant G < \sum_{i=1}^{k+1} g_i$:
  - Packe die Objekte $1, \ldots k$ ein

Branch & Bound für das Rucksackproblem

Eine möglichst wertvolle Auswahl von n Objekten mit Gewichten $g_1, \ldots, g_n \in \mathbb{R}$ und Werten $w_1, \ldots, w_n \in \mathbb{N}$ ist in einen Rucksack mit Gewichtsschranke $G \in \mathbb{R}$ zu packen.

Ein Greedy Algorithmus löst das fraktionale Rucksackproblem exakt. (Hier dürfen Anteile $0 \leqslant x_i \leqslant 1$ des $i$ten Objekts eingepackt werden.)

- Zuerst werden die Objekte absteigend, nach ihrem „Wert pro Kilo“ sortiert, also nach

$$\frac{w_1}{g_1} \geqslant \frac{w_2}{g_2} \geqslant \cdots \geqslant \frac{w_n}{g_n}.$$

- Wenn $\sum_{i=1}^{k} g_i \leqslant G < \sum_{i=1}^{k+1} g_i$:
  - Packe die Objekte $1, \ldots k$ ein und
  - fülle den Rucksack mit dem entsprechenden Anteil am Objekt $k+1$.

Eine Implementierung für das Rucksackproblem

- Wir berechnen eine **Anfangslösung** mit Hilfe des dynamischen Programmieralgorithmus für das Rucksackproblem.

Eine Implementierung für das Rucksackproblem

- Wir berechnen eine **Anfangslösung** mit Hilfe des dynamischen Programmieralgorithmus für das Rucksackproblem.
- Angeregt durch den Greedy Algorithmus definieren wir **Knoten** durch die Paare (J, i):

  die Objekte aus $J \subseteq \{1, \ldots, i\}$ können, ohne die Kapazität G zu überschreiten, in den Rucksack gepackt werden.

Eine Implementierung für das Rucksackproblem

- Wir berechnen eine **Anfangslösung** mit Hilfe des dynamischen Programmieralgorithmus für das Rucksackproblem.
- Angeregt durch den Greedy Algorithmus definieren wir **Knoten** durch die Paare (J, i):

  die Objekte aus $J \subseteq \{1, \ldots, i\}$ können, ohne die Kapazität G zu überschreiten, in den Rucksack gepackt werden.

- Der **Branching Operator** erzeugt die Kinder $(J, i+1)$ und $(J \cup \{i+1\}, i+1)$:

Eine Implementierung für das Rucksackproblem

- Wir berechnen eine **Anfangslösung** mit Hilfe des dynamischen Programmieralgorithmus für das Rucksackproblem.
- Angeregt durch den Greedy Algorithmus definieren wir **Knoten** durch die Paare (J, i):

  die Objekte aus $J \subseteq \{1, \ldots, i\}$ können, ohne die Kapazität G zu überschreiten, in den Rucksack gepackt werden.

- Der **Branching Operator** erzeugt die Kinder $(J, i+1)$ und $(J \cup \{i+1\}, i+1)$: Entweder wird das $i+1$.ste Objekt ausgelassen oder eingepackt.

Eine Implementierung für das Rucksackproblem

- Wir berechnen eine **Anfangslösung** mit Hilfe des dynamischen Programmieralgorithmus für das Rucksackproblem.
- Angeregt durch den Greedy Algorithmus definieren wir **Knoten** durch die Paare (J, i):

  die Objekte aus $J \subseteq \{1, \ldots, i\}$ können, ohne die Kapazität G zu überschreiten, in den Rucksack gepackt werden.

- Der **Branching Operator** erzeugt die Kinder $(J, i+1)$ und $(J \cup \{i+1\}, i+1)$: Entweder wird das $i+1$.ste Objekt ausgelassen oder eingepackt.
- Wir bestimmen ein **erfolgversprechendstes Blatt** als die aktuell wertvollste Bepackung eines nicht disqualifizierten Blatts.

Eine obere Schranke für das Rucksack Problem

Das Rucksackproblem ist ein Maximierungsproblem

Eine obere Schranke für das Rucksack Problem

Das Rucksackproblem ist ein Maximierungsproblem und Branch & Bound benötigt eine obere statt einer unteren Schranke.

Eine obere Schranke für das Rucksack Problem

Das Rucksackproblem ist ein Maximierungsproblem und Branch & Bound benötigt eine obere statt einer unteren Schranke.

- Für die partielle Lösung (J, i) berechnen wir die Restkapazität $G' = G - \sum_{j \in J} g_j$ und

Eine obere Schranke für das Rucksack Problem

Das Rucksackproblem ist ein Maximierungsproblem und Branch & Bound benötigt eine obere statt einer unteren Schranke.

- Für die partielle Lösung (J, i) berechnen wir die Restkapazität $G' = G - \sum_{j \in J} g_j$ und
- wenden den (fraktionalen) Greedy Algorithmus auf die Objekte $i + 1, \ldots, n$ mit der neuen Gewichtsschranke G' an.

Eine obere Schranke für das Rucksack Problem

Das Rucksackproblem ist ein Maximierungsproblem und Branch & Bound benötigt eine obere statt einer unteren Schranke.

- Für die partielle Lösung (J, i) berechnen wir die Restkapazität $G' = G - \sum_{j \in J} g_j$ und
- wenden den (fraktionalen) Greedy Algorithmus auf die Objekte $i + 1, \ldots, n$ mit der neuen Gewichtsschranke G' an.
  - Da der Greedy Algorithmus eine optimale Lösung des fraktionalen Rucksackproblems berechnet und

Eine obere Schranke für das Rucksack Problem

Das Rucksackproblem ist ein Maximierungsproblem und Branch & Bound benötigt eine obere statt einer unteren Schranke.

- Für die partielle Lösung (J, i) berechnen wir die Restkapazität $G' = G - \sum_{j \in J} g_j$ und
- wenden den (fraktionalen) Greedy Algorithmus auf die Objekte $i + 1, \ldots, n$ mit der neuen Gewichtsschranke G' an.
  - Da der Greedy Algorithmus eine optimale Lösung des fraktionalen Rucksackproblems berechnet und
  - da der optimale Wert des fraktionalen Problems mindestens so groß wie der optimale Wert des ganzzahligen Problems ist,

Eine obere Schranke für das Rucksack Problem

Das Rucksackproblem ist ein Maximierungsproblem und Branch & Bound benötigt eine obere statt einer unteren Schranke.

- Für die partielle Lösung (J, i) berechnen wir die Restkapazität $G' = G - \sum_{j \in J} g_j$ und
- wenden den (fraktionalen) Greedy Algorithmus auf die Objekte $i + 1, \ldots, n$ mit der neuen Gewichtsschranke G' an.
  - Da der Greedy Algorithmus eine optimale Lösung des fraktionalen Rucksackproblems berechnet und
  - da der optimale Wert des fraktionalen Problems mindestens so groß wie der optimale Wert des ganzzahligen Problems ist,
  - haben wir die gewünschte obere Schranke gefunden.

Branch & Cut

Löse das lineare 0-1 Programm

$$\min \sum_{i=1}^{n} c_i \cdot x_i \quad \text{so dass} \quad \sum_{j=1}^{n} a_{i,j} \cdot x_j \geqslant b_i \text{ für alle } i$$

$$x_1, \ldots, x_n \in \{0, 1\}.$$

Branch & Cut

Löse das lineare 0-1 Programm

$$\min \sum_{i=1}^{n} c_i \cdot x_i \quad \text{so dass} \quad \sum_{j=1}^{n} a_{i,j} \cdot x_j \geqslant b_i \text{ für alle } i$$
$$x_1, \ldots, x_n \in \{0, 1\}.$$

- ACHTUNG: Die 0-1 Programmierung wie auch die ganzzahlige Programmierung sind im Allgemeinen extrem schwierig.

Branch & Cut

Löse das lineare 0-1 Programm

$$\min \sum_{i=1}^{n} c_i \cdot x_i \quad \text{so dass} \quad \sum_{j=1}^{n} a_{i,j} \cdot x_j \geqslant b_i \text{ für alle } i$$
$$x_1, \ldots, x_n \in \{0, 1\}.$$

- ACHTUNG: Die 0-1 Programmierung wie auch die ganzzahlige Programmierung sind im Allgemeinen extrem schwierig.
- Erfolgsaussichten nur für Programme mit eingeschränkter Struktur.

Branch & Cut

Löse das lineare 0-1 Programm

$$\min \sum_{i=1}^{n} c_i \cdot x_i \quad \text{so dass} \quad \sum_{j=1}^{n} a_{i,j} \cdot x_j \geqslant b_i \text{ für alle } i$$

$$x_1, \ldots, x_n \in \{0, 1\}.$$

- ACHTUNG: Die 0-1 Programmierung wie auch die ganzzahlige Programmierung sind im Allgemeinen extrem schwierig.
- Erfolgsaussichten nur für Programme mit eingeschränkter Struktur.
- Branch & Cut ist das erfolgreichste Verfahren.

Branch & Cut

Löse das lineare 0-1 Programm

$$\min \sum_{i=1}^{n} c_i \cdot x_i \quad \text{so dass} \quad \sum_{j=1}^{n} a_{i,j} \cdot x_j \geqslant b_i \text{ für alle } i$$

$$x_1, \ldots, x_n \in \{0, 1\}.$$

- ACHTUNG: Die 0-1 Programmierung wie auch die ganzzahlige Programmierung sind im Allgemeinen extrem schwierig.
- Erfolgsaussichten nur für Programme mit eingeschränkter Struktur.
- Branch & Cut ist das erfolgreichste Verfahren.
  - Vorgehen ähnlich zu Branch & Bound.

Branch & Cut

Löse das lineare 0-1 Programm

$$\min \sum_{i=1}^{n} c_i \cdot x_i \quad \text{so dass} \quad \sum_{j=1}^{n} a_{i,j} \cdot x_j \geqslant b_i \text{ für alle } i$$

$$x_1, \ldots, x_n \in \{0, 1\}.$$

- ACHTUNG: Die 0-1 Programmierung wie auch die ganzzahlige Programmierung sind im Allgemeinen extrem schwierig.
- Erfolgsaussichten nur für Programme mit eingeschränkter Struktur.
- Branch & Cut ist das erfolgreichste Verfahren.
  - Vorgehen ähnlich zu Branch & Bound.
  - Der Cutting-Schritt ersetzt den Bounding-Schritt.

Branch & Cut

Löse das lineare 0-1 Programm

$$\min \sum_{i=1}^{n} c_i \cdot x_i \quad \text{so dass} \quad \sum_{j=1}^{n} a_{i,j} \cdot x_j \geqslant b_i \text{ für alle } i$$

$$x_1, \ldots, x_n \in \{0, 1\}.$$

- ACHTUNG: Die 0-1 Programmierung wie auch die ganzzahlige Programmierung sind im Allgemeinen extrem schwierig.
- Erfolgsaussichten nur für Programme mit eingeschränkter Struktur.
- Branch & Cut ist das erfolgreichste Verfahren.
  - Vorgehen ähnlich zu Branch & Bound.
  - Der Cutting-Schritt ersetzt den Bounding-Schritt.
  - Der Branching-Schritt bleibt erhalten.

Branch & Cut: Der Algorithmus

- Der Cutting-Schritt entfernt alle 0-1 Bedingungen $x_i \in \{0, 1\}$.

Branch & Cut: Der Algorithmus

- Der Cutting-Schritt entfernt alle 0-1 Bedingungen $x_i \in \{0, 1\}$.
  - Bestimme die optimale Lösung `opt` des linearen Programms.

Branch & Cut: Der Algorithmus

- Der Cutting-Schritt entfernt alle 0-1 Bedingungen $x_i \in \{0, 1\}$.
  - Bestimme die optimale Lösung `opt` des linearen Programms.
  - Wenn `opt` 0-1 wertig ist, dann ist `opt` optimal.

Branch & Cut: Der Algorithmus

- Der Cutting-Schritt entfernt alle 0-1 Bedingungen $x_i \in \{0, 1\}$.
  - Bestimme die optimale Lösung `opt` des linearen Programms.
  - Wenn `opt` 0-1 wertig ist, dann ist `opt` optimal.
    Ansonsten bestimme eine lineare Bedingung, die von `opt` verletzt wird, aber

Branch & Cut: Der Algorithmus

- Der Cutting-Schritt entfernt alle 0-1 Bedingungen $x_i \in \{0, 1\}$.
  - Bestimme die optimale Lösung `opt` des linearen Programms.
  - Wenn `opt` 0-1 wertig ist, dann ist `opt` optimal.
    Ansonsten bestimme eine lineare Bedingung, die von `opt` verletzt wird, aber von keiner 0-1 Lösung.

Branch & Cut: Der Algorithmus

- Der Cutting-Schritt entfernt alle 0-1 Bedingungen $x_i \in \{0, 1\}$.
  - Bestimme die optimale Lösung `opt` des linearen Programms.
  - Wenn `opt` 0-1 wertig ist, dann ist `opt` optimal.
    Ansonsten bestimme eine lineare Bedingung, die von `opt` verletzt wird, aber von keiner 0-1 Lösung.
  - Füge die Bedingung hinzu und wiederhole den Prozess.

Branch & Cut: Der Algorithmus

- Der Cutting-Schritt entfernt alle 0-1 Bedingungen $x_i \in \{0, 1\}$.
  - Bestimme die optimale Lösung `opt` des linearen Programms.
  - Wenn `opt` 0-1 wertig ist, dann ist `opt` optimal.
    Ansonsten bestimme eine lineare Bedingung, die von `opt` verletzt wird, aber von keiner 0-1 Lösung.
  - Füge die Bedingung hinzu und wiederhole den Prozess.
- Aber das Auffinden verletzter Bedingungen ist schwierig.

Branch & Cut: Der Algorithmus

- Der Cutting-Schritt entfernt alle 0-1 Bedingungen $x_i \in \{0, 1\}$.
  - Bestimme die optimale Lösung `opt` des linearen Programms.
  - Wenn `opt` 0-1 wertig ist, dann ist `opt` optimal.
    Ansonsten bestimme eine lineare Bedingung, die von `opt` verletzt wird, aber von keiner 0-1 Lösung.
  - Füge die Bedingung hinzu und wiederhole den Prozess.
- Aber das Auffinden verletzter Bedingungen ist schwierig.
- Wenn keine „gute“ verletzte Bedingung gefunden wird, dann führe einen Branching-Schritt durch.

Branch & Cut: Der Algorithmus

- Der Cutting-Schritt entfernt alle 0-1 Bedingungen $x_i \in \{0, 1\}$.
  - Bestimme die optimale Lösung `opt` des linearen Programms.
  - Wenn `opt` 0-1 wertig ist, dann ist `opt` optimal.
    Ansonsten bestimme eine lineare Bedingung, die von `opt` verletzt wird, aber von keiner 0-1 Lösung.
  - Füge die Bedingung hinzu und wiederhole den Prozess.
- Aber das Auffinden verletzter Bedingungen ist schwierig.
- Wenn keine „gute“ verletzte Bedingung gefunden wird, dann führe einen Branching-Schritt durch.
  - Wähle eine fraktionale Komponente von `opt` und lege sie auf die 0-, bzw. die 1-Alternative fest.

Branch & Cut: Der Algorithmus

- Der Cutting-Schritt entfernt alle 0-1 Bedingungen $x_i \in \{0, 1\}$.
  - Bestimme die optimale Lösung `opt` des linearen Programms.
  - Wenn `opt` 0-1 wertig ist, dann ist `opt` optimal.
    Ansonsten bestimme eine lineare Bedingung, die von `opt` verletzt wird, aber von keiner 0-1 Lösung.
  - Füge die Bedingung hinzu und wiederhole den Prozess.
- Aber das Auffinden verletzter Bedingungen ist schwierig.
- Wenn keine „gute“ verletzte Bedingung gefunden wird, dann führe einen Branching-Schritt durch.
  - Wähle eine fraktionale Komponente von `opt` und lege sie auf die 0-, bzw. die 1-Alternative fest.
  - `opt` ist für jedes Kinder-Problem ausgeschaltet.

Branch & Cut: Der Algorithmus

- Der Cutting-Schritt entfernt alle 0-1 Bedingungen $x_i \in \{0, 1\}$.
  - Bestimme die optimale Lösung `opt` des linearen Programms.
  - Wenn `opt` 0-1 wertig ist, dann ist `opt` optimal.
    Ansonsten bestimme eine lineare Bedingung, die von `opt` verletzt wird, aber von keiner 0-1 Lösung.
  - Füge die Bedingung hinzu und wiederhole den Prozess.
- Aber das Auffinden verletzter Bedingungen ist schwierig.
- Wenn keine „gute“ verletzte Bedingung gefunden wird, dann führe einen Branching-Schritt durch.
  - Wähle eine fraktionale Komponente von `opt` und lege sie auf die 0-, bzw. die 1-Alternative fest.
  - `opt` ist für jedes Kinder-Problem ausgeschaltet.
- Wähle ein Kind und führe den Cutting Schritt aus ....

Branch & Cut für M-TSP I

Ein 0-1 Programm

$$\min \sum_{r=1}^{n} \sum_{s \neq r} d_{r,s} \cdot x_{r,s} \qquad \text{so dass}$$

Branch & Cut für M-TSP I

Ein 0-1 Programm

$$\min \sum_{r=1}^{n} \sum_{s \neq r} d_{r,s} \cdot x_{r,s} \qquad \text{so dass} \sum_{r, r \neq s} x_{r,s} = 2 \text{ für jede Stadt } s,$$

Branch & Cut für M-TSP I

Ein 0-1 Programm

$$\min \sum_{r=1}^{n} \sum_{s \neq r} d_{r,s} \cdot x_{r,s}$$

so dass $\sum_{r, r \neq s} x_{r,s} = 2$ für jede Stadt s,

$$\sum_{r \in S, s \notin S} x_{r,s} \geqslant 2 \text{ für jede Teilmenge } S \subseteq V$$

Branch & Cut für M-TSP I

Ein 0-1 Programm

$$\min \sum_{r=1}^{n} \sum_{s \neq r} d_{r,s} \cdot x_{r,s}$$

so dass $\sum_{r, r \neq s} x_{r,s} = 2$ für jede Stadt s,

$$\sum_{r \in S, s \notin S} x_{r,s} \geqslant 2 \text{ für jede Teilmenge } S \subseteq V$$

und $x_{r,s} \in \{0, 1\}$ für alle $r \neq s$.

Branch & Cut für M-TSP I

Ein 0-1 Programm

$$\min \sum_{r=1}^{n} \sum_{s \neq r} d_{r,s} \cdot x_{r,s}$$

so dass $\sum_{r, r \neq s} x_{r,s} = 2$ für jede Stadt s,

$$\sum_{r \in S, s \notin S} x_{r,s} \geqslant 2 \text{ für jede Teilmenge } S \subseteq V$$

und $x_{r,s} \in \{0, 1\}$ für alle $r \neq s$.

- Die Lösungen entsprechen Rundreisen.

Branch & Cut für M-TSP I

Ein 0-1 Programm

$$\min \sum_{r=1}^{n} \sum_{s \neq r} d_{r,s} \cdot x_{r,s}$$

so dass $\sum_{r, r \neq s} x_{r,s} = 2$ für jede Stadt s,

$$\sum_{r \in S, s \notin S} x_{r,s} \geqslant 2 \text{ für jede Teilmenge } S \subseteq V$$

und $x_{r,s} \in \{0, 1\}$ für alle $r \neq s$.

- Die Lösungen entsprechen Rundreisen.
- Wenn wir die „Teilmengen-Bedingungen“ auslassen, dann erhalten wir disjunkte Kreise als Lösungen.

Branch & Cut für M-TSP I

Ein 0-1 Programm

$$\min \sum_{r=1}^{n} \sum_{s \neq r} d_{r,s} \cdot x_{r,s}$$

so dass $\sum_{r, r \neq s} x_{r,s} = 2$ für jede Stadt s,

$$\sum_{r \in S, s \notin S} x_{r,s} \geqslant 2 \text{ für jede Teilmenge } S \subseteq V$$

und $x_{r,s} \in \{0, 1\}$ für alle $r \neq s$.

- Die Lösungen entsprechen Rundreisen.
- Wenn wir die „Teilmengen-Bedingungen“ auslassen, dann erhalten wir disjunkte Kreise als Lösungen. Man spricht deshalb von Subtour-Eliminationsbedingungen.

Branch & Cut für M-TSP I

Ein 0-1 Programm

$$\min \sum_{r=1}^{n} \sum_{s \neq r} d_{r,s} \cdot x_{r,s}$$

so dass $\sum_{r, r \neq s} x_{r,s} = 2$ für jede Stadt s,

$$\sum_{r \in S, s \notin S} x_{r,s} \geqslant 2 \text{ für jede Teilmenge } S \subseteq V$$

und $x_{r,s} \in \{0, 1\}$ für alle $r \neq s$.

- Die Lösungen entsprechen Rundreisen.
- Wenn wir die „Teilmengen-Bedingungen“ auslassen, dann erhalten wir disjunkte Kreise als Lösungen. Man spricht deshalb von Subtour-Eliminationsbedingungen.
  - Viel zu viele Subtour-Eliminationsbedingungen.

Branch & Cut für M-TSP I

Ein 0-1 Programm

$$\min \sum_{r=1}^{n} \sum_{s \neq r} d_{r,s} \cdot x_{r,s}$$

so dass $\sum_{r, r \neq s} x_{r,s} = 2$ für jede Stadt s,

$$\sum_{r \in S, s \notin S} x_{r,s} \geqslant 2 \text{ für jede Teilmenge } S \subseteq V$$

und $x_{r,s} \in \{0, 1\}$ für alle $r \neq s$.

- Die Lösungen entsprechen Rundreisen.
- Wenn wir die „Teilmengen-Bedingungen“ auslassen, dann erhalten wir disjunkte Kreise als Lösungen. Man spricht deshalb von Subtour-Eliminationsbedingungen.
  - Viel zu viele Subtour-Eliminationsbedingungen.
  - Enferne alle

Branch & Cut für M-TSP I

Ein 0-1 Programm

$$\min \sum_{r=1}^{n} \sum_{s \neq r} d_{r,s} \cdot x_{r,s}$$

so dass $\sum_{r, r \neq s} x_{r,s} = 2$ für jede Stadt s,

$$\sum_{r \in S, s \notin S} x_{r,s} \geqslant 2 \text{ für jede Teilmenge } S \subseteq V$$

und $x_{r,s} \in \{0, 1\}$ für alle $r \neq s$.

- Die Lösungen entsprechen Rundreisen.
- Wenn wir die „Teilmengen-Bedingungen“ auslassen, dann erhalten wir disjunkte Kreise als Lösungen. Man spricht deshalb von Subtour-Eliminationsbedingungen.
  - Viel zu viele Subtour-Eliminationsbedingungen.
  - Enferne alle Subtour-Elimantionsbedingungen und 0-1 Bedingungen.

Branch & Cut für M-TSP II

- Der Cutting-Schritt:

Branch & Cut für M-TSP II

- Der Cutting-Schritt:
  - Berechne eine optimale Lösung `opt` des linearen Programms und

Branch & Cut für M-TSP II

- Der Cutting-Schritt:
  - Berechne eine optimale Lösung `opt` des linearen Programms und bestimme eine von `opt` verletzte Subtour-Eliminationsbedingung.

Branch & Cut für M-TSP II

- Der Cutting-Schritt:
  - Berechne eine optimale Lösung `opt` des linearen Programms und bestimme eine von `opt` verletzte Subtour-Eliminationsbedingung.
  - Füge die Bedingung hinzu

Branch & Cut für M-TSP II

- Der Cutting-Schritt:
  - Berechne eine optimale Lösung `opt` des linearen Programms und bestimme eine von `opt` verletzte Subtour-Eliminationsbedingung.
  - Füge die Bedingung hinzu und wiederhole den Prozess.

Branch & Cut für M-TSP II

- Der Cutting-Schritt:
  - Berechne eine optimale Lösung `opt` des linearen Programms und bestimme eine von `opt` verletzte Subtour-Eliminationsbedingung.
  - Füge die Bedingung hinzu und wiederhole den Prozess.
- Wenn alle Subtour-Eliminiationsbedingungen erfüllt sind, dann führe den Branching-Schritt aus:

Branch & Cut für M-TSP II

- Der Cutting-Schritt:
  - Berechne eine optimale Lösung `opt` des linearen Programms und bestimme eine von `opt` verletzte Subtour-Eliminationsbedingung.
  - Füge die Bedingung hinzu und wiederhole den Prozess.
- Wenn alle Subtour-Eliminiationsbedingungen erfüllt sind, dann führe den Branching-Schritt aus:
  - Bestimme eine fraktionale Komponente e von `opt`.

Branch & Cut für M-TSP II

- Der Cutting-Schritt:
  - Berechne eine optimale Lösung `opt` des linearen Programms und bestimme eine von `opt` verletzte Subtour-Eliminationsbedingung.
  - Füge die Bedingung hinzu und wiederhole den Prozess.
- Wenn alle Subtour-Eliminiationsbedingungen erfüllt sind, dann führe den Branching-Schritt aus:
  - Bestimme eine fraktionale Komponente e von `opt`.
  - Erzwinge die Kante e oder verbiete sie.

Branch & Cut für M-TSP II

- Der Cutting-Schritt:
  - Berechne eine optimale Lösung `opt` des linearen Programms und bestimme eine von `opt` verletzte Subtour-Eliminationsbedingung.
  - Füge die Bedingung hinzu und wiederhole den Prozess.
- Wenn alle Subtour-Eliminiationsbedingungen erfüllt sind, dann führe den Branching-Schritt aus:
  - Bestimme eine fraktionale Komponente e von `opt`.
  - Erzwinge die Kante e oder verbiete sie.
- Durchsuche den Branch & Bound Baum mit Tiefensuche.