

# 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.

- 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.
- 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  $k$ ?  
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  $U \subseteq V$  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  $k \cdot (n - 1) \leq k \cdot n$  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  $\{u, v\} \in E$ . 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  $v \in \ddot{U}$ .

- 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**.

Das Ziel: Löse ein Entscheidungsproblem.

- Wir möchten feststellen, ob sich in der Menge  $U$  aller **potentiellen** Lösungen eine **tatsächliche** Lösung 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  $\phi$  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.



- 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 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.

- (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  $v_1, \dots, v_k$ .
  - ▶ Ein Kind  $v_j$  wird disqualifiziert, wenn  $v_j$  definitiv keine tatsächliche Lösungen enthält.
  - ▶ Sollte allerdings in  $v_j$  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  $B$ ?
- 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

$\alpha$  sei eine KNF-Formel.

Bestimme eine erfüllende Belegung, wenn  $\alpha$  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  $J \subseteq \{1, \dots, n\}$  und  $b: J \rightarrow \{0, 1\}$ :
  - 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  $\alpha$  ein und wiederholen dieses Vorgehen, bis wir einen Widerspruch gefunden haben (und  $v$  wird disqualifiziert) oder bis alle verbliebenen Klauseln mindestens zwei-elementig sind.

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

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

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 erfolgsversprechendstes Blatt.

Damit durchsucht Backtracking den Baum nach dem **Tiefensuche**-Verfahren.



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  $v_1, \dots, v_k$ , 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.

Für den Bounding-Schritt wird eine untere Schranke benötigt:

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

$$\text{unten}(v) \leq f(y)$$

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

- Wann kann der Knoten  $v$  „abgeschnitten“ werden?  
Wenn  $\text{unten}(v) \geq f(y_0)$  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.

- (1) Eine Lösung  $y_0$  wird mit Hilfe einer **Heuristik** berechnet.
- (2) Wiederhole, solange es aktivierte Blätter gibt:
  - (2a) Wähle das **erfolgsversprechendste** aktivierte Blatt  $v$  von  $\mathcal{B}$ .
  - (2b) **Branching**: Wende den Branching Operator  $B$  auf  $v$  an, um die Kinder  $v_1, \dots, v_k$  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  $\text{unten}(v_i) < f(y_0)$ , 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
  - ▶ und die untere Schranke  $\text{unten}(v)$  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  $g_1, \dots, g_n \in \mathbb{R}$  und Werten  $w_1, \dots, w_n \in \mathbb{N}$  ist in einen Rucksack mit Gewichtsschranke  $G \in \mathbb{R}$  zu packen.

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

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

$$\frac{w_1}{g_1} \geq \frac{w_2}{g_2} \geq \dots \geq \frac{w_n}{g_n}.$$

- Wenn  $\sum_{i=1}^k g_i \leq G < \sum_{i=1}^{k+1} g_i$ :
  - ▶ Packe die Objekte  $1, \dots, k$  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.
- Angeregt durch den Greedy Algorithmus definieren wir **Knoten** durch die Paare  $(J, i)$ :
  - die Objekte aus  $J \subseteq \{1, \dots, i\}$  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.

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 (funktionalen) Greedy Algorithmus auf die Objekte  $i + 1, \dots, n$  mit der neuen Gewichtsschranke  $G'$  an.
  - ▶ Da der Greedy Algorithmus eine optimale Lösung des funktionalen Rucksackproblems berechnet und
  - ▶ da der optimale Wert des funktionalen Problems mindestens so groß wie der optimale Wert des ganzzahligen Problems ist,
  - ▶ haben wir die gewünschte obere Schranke gefunden.

Löse das lineare 0-1 Programm

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

- **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.



- 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 ....

## Ein 0-1 Programm

$$\min \sum_{r=1}^n \sum_{s \neq r} d_{r,s} \cdot x_{r,s} \quad \text{so dass } \sum_{r, r \neq s} x_{r,s} = 2 \text{ für jede Stadt } s,$$
$$\sum_{r \in S, s \notin S} x_{r,s} \geq 2 \text{ für jede Teilmenge } S \subseteq V$$

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

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

- 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-Eliminationsbedingungen erfüllt sind, dann führe den **Branching-Schritt** aus:
  - ▶ Bestimme eine fraktionale Komponente  $e$  von  $opt$ .
  - ▶ Erzwingen die Kante  $e$  oder verbiete sie.
- Durchsuche den Branch & Bound Baum mit Tiefensuche.