Exakte Algorithmen

Exakte Algorithmen

Bisher haben wir 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 das Branch & Bound Verfahren.
- Gewinnstrategien in Zwei-Personen Spielen berechnen wir mit der Alpha-Beta Suche.
- Backtracking, Branch & Bound und Alpha-Beta 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.
- 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 höchstens $n - 1$ Kanten sein.
- Also hat G höchstens $k \cdot (n - 1) \leqslant k \cdot n$ Kanten.

Der Algorithmus VC(G, k)

(1) Setze $U = \emptyset$.

(2) Wenn G mehr als $k \cdot n$ Kanten hat, dann hat G keine Überdeckung der Größe k. Der Algorithmus bricht in diesem Fall mit einer „erfolglos“ Meldung ab.
Ansonsten, für $k = 0$, brich mit einer „erfolgreich“ Meldung 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 $U = U \cup \{u\}$ und brich mit der Nachricht „erfolgreich“ ab.
- Ansonsten rufe **VC(G − v, k − 1)** auf. Wenn die Antwort „erfolgreich“ ist, dann setze $U = U \cup \{v\}$ und brich mit der Nachricht „erfolgreich“ ab.
- Wenn beide Aufrufe erfolglos sind, dann brich mit der Nachricht „erfolglos“ ab.
  // Die Überdeckung U muss einen Endpunkt der Kante $\{u, v\}$
  // besitzen. Zuerst wird die Option $u \in U$ rekursiv untersucht, und
  // bei Mißerfolg auch die Option $v \in U$.

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.

- 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 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 $\mathcal{B}$.

- 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.

(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. (Animation)

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.
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, 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: 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 (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

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.
- 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.

(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
  - 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.

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.
- 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 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.

M-TSP

Im metrischen Traveling Salesman Problem ist eine kürzeste Rundreise für n Städte zu bestimmen, wobei die Distanzen zwischen den Städten die Dreiecksungleichungen erfüllen.

- Wir beginnen mit dem „Brute Force“ Ansatz: Wir zählen alle möglichen Permutationen auf und bestimmen dann die beste zugehörige Rundreise.
- Dumme Frage: Wie zählt man alle Permutationen von n Objekten in Zeit $O(n!)$ auf?
- Wir lösen ein noch allgemeineres Problem, nämlich wir zählen alle einfachen Wege einer fixen Länge s in einem gerichteten oder ungerichteten Graphen G auf.
  - Wähle den vollständigen (ungerichteten) Graphen $G = V_n$ mit n Knoten und wähle $s = n - 1$: Die Wege der Länge $n - 1$ in V_n entsprechen genau den Permutationen von n Objekten.

Aufzählung aller Wege

Wir verwenden einen zur Tiefensuche ähnlichen Ansatz.

- Wenn wir bereits einen einfachen Weg

$$(a_1, \ldots, a_i) = w$$

erzeugt haben, dann haben wir die Knoten $a_1, \ldots, a_i$ markiert: Sämtliche Verlängerungen von w müssen neue Knoten durchlaufen.

- Wenn wir alle Verlängerungen von w berechnet haben, dann geben wir a_i wieder frei, heben also die Markierung auf.

Das Programm

```
void wege (int u, int s)
// Alle in u beginnenden Wege der Länge s werden ausgegeben.
    { // u wird markiert.
    nummer[u] = nr++; //Anfänglich ist nr=0 und nummer[u]=-1.
    if (nr == s) gib das nr-Array aus; else
        for (v = 0; v < n; v ++)
        // alle unmarkierten Nachfolger werden besucht:
            if ( (nummer[v] == -1) && (A[u][v]))
            // A ist die Adjazenzmatrix von G
                wege (v,s);
    nr –; nummer [u] = -1;} // u wird freigegeben.
```

- Wenn alle Permutationen aufzuzählen sind, dann verwalte alle nicht markieren Knoten in einer Schlange.
  - In `wege(u,s)`: Zuerst füge ein Trennsymbol ein. Freigegebene Nachbarn von u werden nach dem Trennzeichen eingefügt und ein mehrfacher Besuch während `wege(u,s)` kann verhindert werden.
  - Leere die Schlange in der for-Schleife. Laufzeit $= O(n!)$.

Branch & Bound für M-TSP: Die Anfangslösung

- Wir haben bereits die Spannbaum-Heuristik kennengelernt.
- Eine erfolgreichere Heuristik ist die Nearest-Insertion Regel:
  - Man beginnt mit einer kürzesten Kante.
  - Eine Rundreise wird iterativ, Kante für Kante, konstruiert.
  - In einem Iterationsschritt wähle die Stadt, die unter allen noch nicht erfassten Städten einen kleinsten Abstand zur gegenwärtigen partiellen Rundreise besitzt.
  - Füge die Stadt zwischen ein Paar benachbarter Städte der partiellen Rundreise ein: Wähle das Paar so, dass der Längenanstieg minimal ist.
  - Man kann zeigen dass die Nearest-Insertion Regel 2-approximativ ist.

1-Bäume

- Wähle eine beliebige Stadt s und berechne einen minimalen Spannbaum $B_{\min}$ für alle Städte bis auf Stadt s.
- Füge s zu $B_{\min}$ hinzu: Verbinde s mit den beiden nächstliegenden Städten.
- Wir haben den 1-Baum $B^*_{\min}$ erhalten.

  Warum 1-Baum? $B^*_{\min}$ hat einen einzigen Kreis.
- Das Gewicht $l(B^*_{\min})$ des 1-Baums ist unsere neue untere Schranke.
- Warum ist $l(B^*_{\min})$ eine untere Schranke?

Eine verbesserte untere Schranke I

Geht es noch besser? Ersetze die Distanzen $d_{r,s}$ zwischen r und s durch $d^*_{r,s} = d_{r,s} + \lambda_r + \lambda_s$ mit beliebigen λ_r.

- Wie verändert sich die Länge $L(R)$ einer Rundreise R?
  Die neue Länge ist
$$L^*(R) = L(R) + 2 \cdot \sum_{r=1}^{n} \lambda_r.$$
- Wie verändert sich die Länge $l(B)$ eines 1-Baums B?
  - $\text{grad}_B(r)$ sei die Anzahl der Nachbarn von r in B.
  - Die neue Länge ist
$$l^*(B) = l(B) + \sum_{r=1}^{n} \lambda_r \cdot \text{grad}_B(r).$$
- Es ist $L^*(R) \geqslant \min_B l^*(B)$. Also folgt
$$L(R) + 2 \cdot \sum_{r=1}^{n} \lambda_r \geqslant \min_B l(B) + \sum_{r=1}^{n} \lambda_r \cdot \text{grad}_B(r).$$

Eine verbesserte untere Schranke II

Aus $L(R) + 2 \cdot \sum_{r=1}^{n} \lambda_r \geqslant \min_B l(B) + \sum_{r=1}^{n} \lambda_r \cdot \text{grad}_B(r)$ folgt

$$L(R) \geqslant f(\lambda_1, \ldots, \lambda_n) := \min_B l(B) + \sum_{r=1}^{n} \lambda_r \cdot (\text{grad}_B(r) - 2).$$

- Maximiere $f(\lambda_1, \ldots, \lambda_n)$ und die untere Schranke wird so groß wie möglich.

  $\max_{\lambda \in \mathbb{R}^n} f(\lambda)$ wird die Held-Karp Schranke genannt.
- Wie schwierig ist die Maximierung?
  - f ist ein Minimum linearer Funktionen und f ist damit konkav.
  - f hat also nur globale Optima.
- Iterative Heuristik: *subgradiente Optimierung*

Branch & Bound für M-TSP

- Der Bounding-Schritt:
  - Berechne die Held-Karp Schranke $\max_{\lambda \in \mathbb{R}^n} f(\lambda)$.
  - Bestimme einen für die Distanzen $d^*_{r,s} = d_{r,s} + \lambda_r + \lambda_s$ minimalen 1-Baum B_{opt}.
  - B_{opt} ist zusammenhängend mit genau einem Kreis. B_{opt} ist eine Rundreise oder es gibt eine Stadt s mit mindestens drei Nachbarn.
- Die Kanten $\{r, s\}, \{t, s\}$ seien die längsten, mit s inzidenten Kanten in B_{opt}.
- Der Branching-Schritt produziert drei Kinder-Probleme.
  (1) Die Kante $\{r, s\}$ wird verboten.
  (2) $\{r, s\}$ muss durchlaufen werden, aber $\{t, s\}$ ist verboten.
  (3) Sowohl $\{r, s\}$ wie auch $\{t, s\}$ müssen durchlaufen werden. Alle anderen mit s inzidenten Kanten werden verboten.

  Das Ursprungsproblem wird disjunkt zerlegt und B_{opt} wird in jedem Teilproblem ausgeschaltet.

Branch & Bound für M-TSP: Der Algorithmus

- Die Teilprobleme werden durch Paare (S, T) beschrieben:
  - Die Kanten in $S \subseteq \{ \{u, v\} \mid u \neq v\}$ sind erzwungen,
  - die Kanten in $T \subseteq \{ \{u, v\} \mid u \neq v\}$ sind verboten.
- Der Bounding-Schritt bestimmt einen minimalen 1-Baum B_{opt} mit maximalem Gewicht.
- Der Branching-Schritt schaltet B_{opt} aus.
- Der Branch & Bound Baum wird mit Tiefensuche erzeugt:

  Die zuletzt erzeugten Teilprobleme werden weiterverarbeitet, sind also die vielversprechendsten Teilprobleme.

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 $x_i \in \{0, 1\}$.
  - 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

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

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:
  - 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.

Zwei-Personen Spiele

- Zwei Spieler, Alice und Bob, sind beteiligt.
- Alice zieht zuerst. Danach alternieren die Spieler.
- Forderung: Alle Spiele sind endlich und enden mit einer Auszahlung an Alice.

Das Ziel

Bestimme eine Gewinnstrategie für Alice, also eine Strategie, die ihr eine größtmögliche Auszahlung garantiert.

Aber eine effiziente Bestimmung ist für nicht-triviale Spiele nicht möglich. Versuche das unter den Umständen Beste zu erreichen.

Der Spielbaum $\mathcal{B}$

Wir definieren den Spielbaum $\mathcal{B}$.

- Die Wurzel r entspricht der Ausgangsstellung und ist mit dem zuerst ziehenden Spieler, also mit Alice markiert.
- Wenn der Knoten v mit dem aktuell ziehenden Spieler markiert ist und der Stellung S_v entspricht, dann erzeugen wir für jeden möglichen Zug des ziehenden Spielers ein Kind w von v:
  - w ist mit dem Gegenspieler markiert und entspricht der durch den gerade ausgeführten Zug veränderten Stellung S_v.
- Wenn hingegen das Spiel in v entschieden ist, dann wird v zu einem Blatt, und wir markieren v mit der Auszahlung $A(v)$ an Alice.
- Unser Ziel: Werte $\mathcal{B}$ aus, um die größtmögliche, von Alice erzwingbare Auszahlung zu bestimmen.

Die Minimax Auswertung von $\mathcal{B}$ für den Knoten v

(1) Wenn v ein Blatt ist, dann gib den Wert $A(v)$ zurück.

(2) Wenn Alice in v zieht, dann
- setze $\text{Max} = -\infty$,
- durchlaufe alle Kinder w von v und setze $\text{Max} = \max\{\text{Max}, \text{Minimax}(w)\}$.
  // Alice wählt den für sie besten Zug.
- Gib den Wert Max zurück.

(3) Wenn Bob in v zieht, dann
- setze $\text{Min} = \infty$,
- durchlaufe alle Kinder w von v und setze $\text{Min} = \min\{\text{Min}, \text{Minimax}(w)\}$.
  //Der Zug von Bob minimiert die Auszahlung an Alice.
- Gib den Wert Min zurück.

// Die Minimax-Auswertung erfolgt mittels Tiefensuche.
// Wenn v die Wurzel ist, dann ist Max der Wert der größten,
// von Alice erreichbaren Auszahlung.

Die Minimax-Auswertung ist redundant!

- Alice möge im Knoten v ziehen.
- Der Knoten u sei ein Vorfahre von v und Bob ziehe in u.
- Zum Zeitpunkt der Auswertung von v besitze die Min-Variable von u den Wert β.

- Wir betrachten alle Kinder w von v, also alle Züge von Alice.
- Wenn Alice für irgendein Kind w eine Auszahlung mindestens β erzwingt, dann kann Bob unbeschadet v verhindern: (Animation)

  Z. B. kann er im Knoten u so ziehen, dass v nicht erreichbar ist und dennoch die Auswertung von Alice auf höchstens β beschränkt ist.
- Die Auswertung von v kann abgebrochen werden!

Um möglichst frühzeitig abbrechen zu können:

Was sollte der Wert von β sein, bevor v besucht wird?

β ist das Minimum der Min-Variablen, über alle mit Bob markierten Vorfahren von v.

Und wenn Bob am Zug ist?

(1) Der Knoten v sei mit Bob markiert,
(2) Der Vorfahre u von v sei mit Alice markiert.
(3) Der Wert der Max-Variable von u sei α, bevor v besucht wird.

- Wenn Bob für irgendein Kind w von v eine Auszahlung von höchstens α erzwingen kann, dann kann Alice v ohne Schaden für sich umgehen. Brich die Evaluierung von v ab. (Animation)
- Um möglichst frühzeitig abbrechen zu können: Was sollte der Wert von α sein, bevor v besucht wird?

  α ist das Maximum der Max-Variablen, über alle mit Alice markierten Vorfahren von v.

Alpha-Beta (v, α, β)

(1) Wenn v ein Blatt ist, dann gib den Wert $A(v)$ zurück.

(2) Ansonsten durchlaufe alle Kinder w von v:

- Wenn v mit Alice markiert ist, dann setzte
  $\alpha = \max\{\ \alpha,\ \text{Alpha-Beta}(w, \alpha, \beta)\ \}$;
  Wenn $\alpha \geqslant \beta$, dann terminiere mit Antwort α.
  // Bob wird das Erreichen von v unbeschadet verhindern können.
- Wenn v mit Bob markiert ist, dann setzte
  $\beta = \min\{\ \beta,\ \text{Alpha-Beta}(v, \alpha, \beta)\ \}$;
  Wenn $\alpha \geqslant \beta$, dann terminiere mit Antwort β.
  // Alice wird das Erreichen von v unbeschadet verhindern können.

(3) Gib den Wert α zurück, wenn v mit Alice markiert ist. Ansonsten gib den Wert β zurück.

Was berechnet Alpha-Beta(v, α, β)?

A sei die größte von Alice erreichbare Auszahlung im Teilbaum mit Wurzel v

- Wenn v mit Alice markiert ist, dann wird $\max\{\alpha, A\}$ ausgegeben, falls $A \leqslant \beta$.
- Wenn v mit Bob markiert ist, dann wird $\min\{\beta, A\}$ ausgegeben, falls $\alpha \leqslant A$.

- Beweis: Führe eine Induktion über die Tiefe des Spielbaums.
- Wenn $v = r$ die Wurzel des Spielbaums ist:

  Alpha-Beta($r, -\infty, \infty$) gibt den Wert der größten von Alice erreichbaren Auszahlung zurück.

Ein Vergleich zwischen Minimax und Alpha-Beta

Eine Modell-Situation

$\mathcal{B}$ sei ein vollständiger b-ärer Spielbaum der Tiefe d.

- Der Spielbaum hat $\Theta(b^d)$ Knoten.
- Es gibt eine Alpha-Beta Auswertung von $\mathcal{B}$, die höchstens $\text{opt} = b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$ Knoten besucht. (Übungsaufgabe)
  Im besten Fall wird die Anzahl besuchter Knoten also von $\Theta(b^d)$ auf $\Theta(\sqrt{b^d})$ reduziert. (Animation)
  Im Vergleich zur Minimax-Auswertung kann damit die Anzahl simulierter Züge, bei gleichen Ressourcen, verdoppelt werden!
- Aber mehr ist nicht möglich: Jede Alpha-Beta Auswertung von $\mathcal{B}$ besucht mindestens opt Knoten.
- Experimentelle Erfahrung: Der „Best Case“ wird approximativ erreicht. Warum?

Schachprogramme

Es gibt ungefähr 38^{84} verschiedene Schach-Stellungen und selbst eine Reduktion auf 38^{42} hilft nicht wirklich.

- Schachprogramme versuchen eine möglichst weit gehende Vorausschau, um die Stärke eines Zuges beurteilen zu können.
- Die Grobstruktur eines Schachprogramms besteht deshalb aus
  - einer Bewertungsprozedur, die Stellungen mit reellen Zahlen bewertet
  - und einem Suchalgorithmus besteht, der die Konsequenz eines jeden möglichen Zugs für die „nächsten“ Züge vorausberechnet.
- Der Suchalgorithmus basiert meistens auf Varianten des Alpha-Beta Algorithmus. Alpha-Beta setzt seine Suche mit der am höchsten bewerteten Nachfolgestellung fort. (Web)
- Der Best Case für die Alpha-Beta Suche ist nicht weiter überraschend, wenn die Bewertungsprozedur gut ist.