Aussagenlogikbasierte Suchprobleme

Wintersemester 2001/02

Universität zu Köln

Patrick Holz

(Skript zur Vorlesung von Dr. Bert Randerath)

Version vom: 13. Dezember 2003

Inhaltsverzeichnis

1 Einführende Beispiele

Beispiel 1: Erfüllbarkeitsproblem SAT

Input: Boolesche Formel in KNF

$$F = (x_{11} \vee \cdots \vee x_{1m}) \wedge \cdots \wedge (x_{n1} \vee \cdots \vee x_{nm})$$

$$x_{ij} \in \{a_1, \overline{a_1}, a_2, \ldots, a_n, \overline{a_n}\}$$

Frage: Gibt es eine Belegung t: $\{a_1, \ldots, a_n\} \rightarrow \{true, false\}$ die F erfüllt? (NP-vollständig)

Beispiel 2: Kontradiktionsproblem

Input: wie Beispiel 1

Frage: Gibt es *keine* Belegung t, die F erfüllt? (Co-NP-vollständig)

Beispiel 3: Handlungsreisendenproblem HRP/TSP

Input: nxn - Kostenmatrix C = (c[i,j]) mit 0 ≤i,j≤n und (c[i,j]) > 0

Gesucht: Permutation $\hat{\pi} \in S_n$ so dass für alle $\pi \in S_n$ gilt:

$$cost(\hat{\pi}) \leq cost(\pi) := \sum_{i=0}^{n-1} c[\pi(i), \pi(i+1) \text{mod n}]$$

D.h. $\hat{\pi}$ repräsentiert eine kürzeste Rundreise. (NP-hart)

Beispiel 4: $n^2 - 1$ - Puzzle

Input: Zwei Konfigurationen S und Z von $n^2 - 1$ verschiedenen Plättchen auf einem Brett der Länge n

Frage: Lassen sich die Plättchen auf dem Brett derart verschieben, dass man von S zu Z gelangt? (in polynomieller Zeit lösbar ⇒∈ P)

Beispielkonfigurationen:

a) S:=

2	3	1
4	5	6
7	8	

Z:=

1	2	3
4	5	6
7	8	

Hierzu existiert eine Lösung (der Einfachheit halber werden die jeweiligen Positionen des freien Feldes angegeben, Bezeichnung anhand der Reihenfolge gemäß Z): 6, 3, 2, 1, 4, 5, 6, 3, 2, 1, 4, 1, 2, 3, 6, 9

b) S:=

4	2	3
1	5	6
7	8	

Z:=

1	2	3
4	5	6
7	8	

Hierzu existiert *keine* Lösung

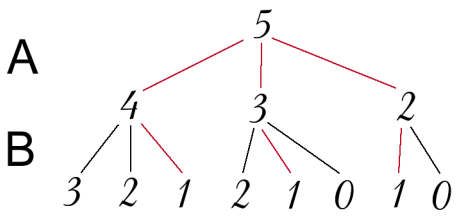
Beispiel 5: Nimm-Spiel

Input: n Streichhölzer, zwei Spieler A und B nehmen abwechselnd 1-3 Streichhölzer; wer das letzte Streichholz nimmt, hat verloren.

Frage: Hat Spieler B eine Gewinnstrategie, wenn A beginnt? (in n Schritten entscheidbar)

Lösung: B hat eine Gewinnstrategie, falls n = 4i + 1 (ansonsten gewinnt immer A)

Spielbaum für n=5 (rote Linien = Gewinnstrategie von B):



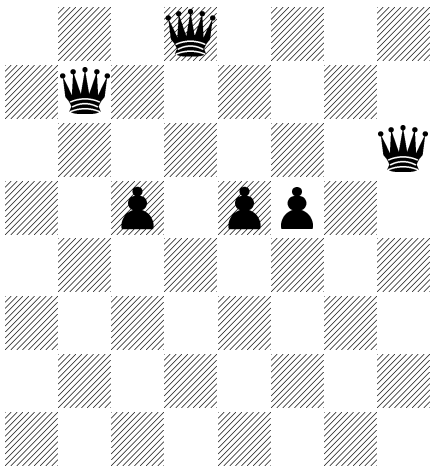
Beispiel 6: 8-Damen-Problem

Aufgabe: Plaziere 8 Damen auf einem Schachbrett derart, dass sich keine 2 Damen gegenseitig bedrohen.

Vorgehen: Plaziere die Damen Zeile für Zeile und bewerte anhand einer Heuristik. Das Ziel der Heuristik ist es, die vielversprechendste Möglichkeit auszuwählen. Beispiele für Heuristiken:

- f(x) sei die Anzahl der unattackierten Felder bzgl. der bereits besetzten Felder einschließlich des zu bewertenden Feldes x
- g(x) sei die minimale Anzahl der unattackierten Felder pro noch freier Zeile

Als Beispielkonfiguration sei gegeben:



Dabei symbolisieren die Damen die bereits gesetzten Damen und die Bauern die möglichen Positionen für die vierte Dame.

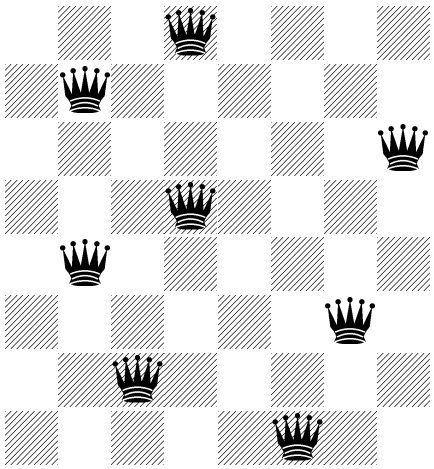
Nun berechnet man die Heuristiken f und g jeweils für die Spalte c, e und f:

f(c) = 8 g(c) = 1

f(e) = 9 g(e) = 1

f(f) = 10 g(f) = 2

Somit weisen beide Heuristiken die Spalte f als optimale Spalte aus. Im weiteren Verlauf würde jedoch ersichtlich, dass dies nicht zu einer Lösung führt, die Heuristiken sind also scheinbar ungeeignet. Eine komplette Lösung sieht wie folgt aus (es gibt mehrere):



2 Backtracking (uninformiertes Suchen)

2.1 Algorithmische Paradigmen

Obige Beispiele zeigen, dass wir es häufig mit NP-vollständigen Problemen, Problemen in Co-NP oder P-Space zu tun haben. Diese Konzepte stellen aber „worst-case“ -Szenarien dar. D.h. aller Voraussicht nach (P≠NP) ist es unmöglich, für die obigen Probleme effiziente Algorithmen zu finden.
Nun ist es für praktische Anwendungen meistens von größerer Bedeutung, ob ein Algorithmus existiert, der die Probleme im mittleren oder „typischen“ Fall effizient löst („average-case“ -Analysen).
Um diese Ziele zu erreichen, werden algorithmische Paradigmen wie Backtracking, Branch & Bound, $\alpha - \beta-$Suche u.a. verwendet, um in potentiell

exponentiell (in der Inputlänge) großen Suchbäumen durch „Abschneiden“ ganzer Teilbäume die Suchzeit drastisch zu beschleunigen.
Dieses Abschneiden geschieht mit Hilfe von explizit oder implizit vorhandenen, problemspezifischen Constraints (Bedingungen, Prädikate). Eine weitere Beschleunigung lässt sich durch problemabhängige Heuristiken zur Lösungssuche erzielen.

2.2 Das Prinzip von Backtracking

Backtracking ist eine Strategie (ein Paradigma), die nach der Methode „Trial and Error“ systematisch versucht, eine oder mehrere zulässige Lösung(en) für diskrete Optimierungsprobleme zu generieren, indem Schritt für Schritt Entscheidungsfolgen verlängert werden, bis eine Lösung gefunden wird oder die aktuelle Entscheidungsfolge ein Constraint verletzt. In letzterem Fall wird die zuletzt gefällte Einzelentscheidung zurückgenommen und die so verkürzte Entscheidungsfolge in eine andere Richtung verlängert (falls möglich).

2.3 Backtracking-Algorithmus für das SAT-Problem

Definitionen und Notationen:

- Var = $\{a_1, \ldots, a_n\}$ sei eine Menge von n Boole'schen Variablen
- Variable $a_i \in$ Var oder deren Negation $\overline{a_i}$ heißt Literal über i
- L = $\{a_1, \ldots, a_n, \overline{a_1}, \ldots, \overline{a_n}\}$ ist die Literalmenge über Var
- Disjunktion c = $x_1, \ldots, x_n$ von Literalen $x_i \in$ L heißt Klausel
- $\square$ bezeichne die leere Klausel

- cl(n) sei die Menge aller Klauseln über Var
- cl(n,s) sei die Menge aller Klauseln über Var mit der festen Länge s
- $cl(n)^r$ / $cl(n,s)^r$ ist die Konjunktion von r Klauseln aus cl(n) / cl(n,s) = Menge der KNF von r Klauseln aus cl(n) / cl(n,s)
- Repräsentiere $F \in cl(n)^r$ als geordnete Multimenge

$$F = \{\{x_{11}, \ldots, x_{1s_1}\}, \ldots, \{x_{r1}, \ldots, x_{rs_r}\}\}$$

- partielle Abbildung t: V→L mit $t(a_i) \in \{a_i, \overline{a_i}\}$ heißt partielle Belegung von V mit Wahrheitswerten
- t erfüllt Klausel c, falls $t \cap c \neq \emptyset \Rightarrow$ kein t erfüllt die leere Klausel
- $F_t := \{c - \overline{t} | c \in F, c \cap t \neq \emptyset\}$ sind Formeln, die aus F unter t entstehen; t erfüllt F, falls $F_t = \emptyset \Rightarrow$ F erfüllbar, falls $F_t = \emptyset$

Beispiel für F_t:
Sei F = $(a \vee b) \wedge (c \vee \overline{b})$ und sei b = true ($\Rightarrow$ t(b) = b).
$\Rightarrow F_t = c$
Für t = b,c wäre F erfüllt.

2.4 Satz von Cook

Das Problem der Erfüllbarkeit einer Inputformel F $\in \bigcup_{n,r \geq 0} cl(n,s)^r$ ist

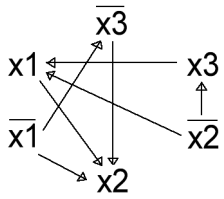
a) NP-vollständig, falls s > 2
b) linear, falls s $\leq$ 2

Beweis zu b):

für s = 1 $\surd$ (unit clause rule)

für s = 2 Splitting der Klauseln in Implikationen: $(x \vee y) \Leftrightarrow (\overline{x} \rightarrow y) \wedge (\overline{y} \rightarrow x)$ und Bildung des entsprechenden Digraphen. Beispiel:

Sei F = $(x_1 \vee x_2) \wedge (x_1 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2) \wedge (x_2 \vee x_3) \Rightarrow$ Digraph:



Lemma: F ist nicht erfüllbar, wenn es eine Variable x_i gibt, so dass die Wege von x_i nach $\overline{x_i}$ und umgekehrt existieren. Die einfachste Methode, dies zu prüfen, ist die Zerlegung des Digraphen in starke Zusammenhangskomponenten, in denen dann jeweils auf die Eigenschaft getestet wird.

2.5 Backtracking-Algorithmus für ERF

1. Lies Input-Formel ein.
2. Setze Variable n := Anzahl der Klauseln
3. Setze Laufvariable l := 0
4. push t = $\emptyset$ auf den Keller K (auf K befinden sich partielle Belegungen)
5. repeat
6. pop top Belegung t von K
7. l := n - (kleinster Index i der a_i in t) + 1

8. if $\square \notin F_t$

9. then if $F_t = \emptyset$ then melde „t ist Lösung“

10. else push $t_1 := t \cup \{a_{n-l}\}$ und $t_2 := t \cup \{\overline{a_{n-l}}\}$

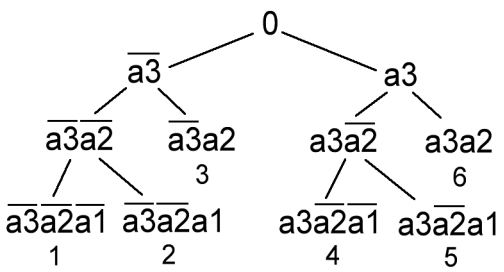
11. until empty(K)

2.6 Backtracking-Bäume

Sei t ein vollständiger Baum der Tiefe n. ERF mit F $\in cl(n)^r$ definiert eindeutig einen Teilbaum T(F) von T, dessen Knoten mit partiellen Belegungen t markiert sind.

- T(F) enthält immer den Wurzelnoten v von T, v ist mit t = $\emptyset$ markiert.
- Ist v innerer Knoten von T(F) und mit t markiert, so gilt immer: $\square \notin F_t \neq \emptyset$ und der linke und rechte Sohn v_l und v_r von v existieren, markiert mit $t \cup \{\overline{a_{n-|t|}}\}$ bzw. $t \cup \{a_{n-|t|}\}$.
- Blätter von T(F) sind mit Belegungen t markiert, für die entweder $F_t = \emptyset$ (Lösungsblatt) oder $\square \in F_t$ (Nicht-Lösungsblatt) gilt.

T(F) heißt „Backtracking-Baum“ von F unter ERF. T(F) hat $\frac{|T(F)|+1}{2}$ Blätter.
Beispiel:
$F = \{\{a_1, a_2\}, \{\overline{a_2}, a_3\}\}$



Für Blätter 1, 3 und 4 gilt: $\square \in F_t \Rightarrow$ keine Lösung.
Für Blätter 2, 5 und 6 gilt: $F_t = \emptyset \Rightarrow$ Lösung.
Die Laufzeit von ERF mit Inputformel F $\in cl(n)^r$ ist beschränkt durch $O((n + |F|) \cdot |T(F)|)$
Der Erwartungswert $E(|T|)$ ist in Polynomzeit berechenbar.

2.7 Das allgemeine Backtracking

Beim ERF ist die Anzahl der Variablen (= Anzahl der zu treffenden Entscheidungen) durch das Inputproblem fest vorgegeben. Zusätzlich gilt, dass es genau zwei Entscheidungsmöglichkeiten (true / false) gibt. Dies muss nicht immer gelten.
Beim 8-Puzzle zum Beispiel kann es sein, dass man im ersten Schritt (um von S zu Z zu gelangen) 4 Möglichkeiten hat, im zweiten 3, dann wieder 4 usw., darüber hinaus ist die Anzahl der benötigten Schritte nicht eindeutig, daher gelten zyklische Lösungen als unerwünscht. Das Beispiel zeigt, welche Eigenschaften Probleme haben können:

- Die Zahl der zu treffenden Entscheidungen ist nicht eindeutig durch das Inputproblem vorgegeben, verschiedene Lösungen des gleichen Inputs können unterschiedliche Längen haben.
- Die Zahl der Möglichkeiten pro Entscheidung ist endlich, kann aber von Entscheidung zu Entscheidung variieren.
- Zur Vermeidung unendlich großer Entscheidungsbäume muss stets Zyklenfreiheit gefordert werden

Das allgemeine Backtracking enthält eine Schranke „max_Tiefe“ , die angibt, bis zu welcher Tiefe Entscheidungsbäume maximal aufgebaut werden. Dies

dient bei Problemen wie dem 8-Puzzle dazu, das Laufzeitverhalten zu beschränken und kurze Lösungen zu präferieren.
Das nachfolgende algorithmische Schema Backtracking (eine Verallgemeinerung von ERF) berechnet alle zulässigen Lösungen k mit Tiefe(k) $\leq$ max_Tiefe eines Inputbeispiels der Klasse π. Dabei wird ein Keller „Offen“ benutzt, auf dem k gespeichert wird:

1. Lies Input-Problem I sowie max_Tiefe ein.
2. k := $\emptyset$; push(k,Offen); knotenzahl := 1;
3. repeat
4. k := top(Offen); pop(Offen);
5. if k Lösung von I then melde „K ist Lösung“
6. else if Tiefe(k) < max_Tiefe
7. then berechne alle zulässigen, zykelfreien Verlängerungen k_1 bis k_s von k;
8. for i := 1 to s do push(k_i,Offen); knotenzahl++;
9. until empty(Offen)

Auch hier definiert Backtracking zum Input I $\in \pi$ einen Backtracking-Baum T(I), dessen Knotenzahl $|T(I)|$ in Backtracking der Variablen „knotenzahl“ entspricht. Die Laufzeit von Backtracking beträgt $O(\text{p(I,max\_Tiefe)} \cdot |T(I)|)$ für ein Polynom p. Man benötigt nun noch eine erwartungstreue Schätzung für $|T(I)|$:

- Wähle in T(I) zufällig einen Weg W = $(\emptyset = k^{(0)} \rightarrow k^{(1)} \rightarrow \cdots \rightarrow k^{(m)})$ von der Wurzel bis zu einem Blatt $k^{(m)}$ von T(I) mit m $\leq$ max_Tiefe
- Bestimme für jedes $k^{(i)}$ aus W die Anzahl s_i der (zulässigen) Söhne in T(I) mit $0 \leq i \leq m$
- Definiere auf derartigen Wegen in T(I) die Zufallsvariable Z(W) := $1 + \sum_{i=0}^{m-1} \prod_{j=0}^{i} s_j$

Es gilt: E(Z) = $|T(I)|$, d.h. das Verfahren schätzt die Knotenzahl im Mittel richtig ab, und zwar mit polynomiellem Aufwand.

3 Branch & Bound (B&B)

3.1 Motivation

Backtracking sucht im Backtracking-Baum nach einem Lösungsblatt mit der Methode Depth First Search (DFS), also mit uninformierter Suche. Im Unterschied dazu expandiert die Methode B&B (eine Modifikation von Backtracking) jeweils die Vielversprechendsten der bereits erzeugten Knoten von T(I). B&B wird als Lösungsmethode eingesetzt, um diskrete Optimierungsprobleme (z.B. HRP) zu lösen.

3.2 B&B am Beispiel des HRP/TSP

Hilfseinschränkung: Die Dreiecksungleichung soll gelten.
Sei G(V,E) ein vollständiger, ungerichteter Graph mit V=0,...,n-1 mit Kostenfunktion c: E $\rightarrow \Re^+$

Gesucht: Eine kürzeste Rundreise $\pi = (0, \pi(1), \pi(2), \ldots, \pi(n-1), 0)$ für eine Permutation π von V mit $\pi(0) = 0$, d.h.:

$$cost(\pi) := \sum_{i=0}^{n-1} c(\pi(i), \pi(i+1 \bmod n)) \doteq min$$

Ähnlich wie Backtracking definiert B&B einen Suchbaum T(G,c), dessen Knoten jeweils fixierte Entscheidungsfolgen (Teilrundreisen) repräsentieren. Ein Knoten K in T(G,c), der die Teilrundreise $0 \to \pi(1) \to \cdots \to \pi(j-1)$ repräsentiert, erhält bei der Expansion genau (n-j) Söhne. Um zu beurteilen, welcher der erzeugten, aber noch nicht expandierten Knoten K am „vielversprechendsten“ ist, führe Bewertungsfunktion f: Knoten(T(G,c)) $\to \Re^+$ ein. Forderungen an f:

- a) Falls k eine Rundreise $0 \to \pi(1) \to \cdots \to \pi(n-1) \to 0$ repräsentiert, so soll gelten: f(k) = cost(π)
- b) Ist k' Sohn von k in T(G,c), so soll gelten: f(k) $\leq$ f(k')

Realisiere f(k) beim HRP wie folgt: f(k) := g(k) + h(k) für einen Knoten k, wobei

$$(i)\ \text{g(k)} := \sum_{i=0}^{j-1} c(\pi(i), \pi(i+1)) + \begin{cases} c(\pi(n-1), \pi(0)) & \text{falls j=n (d.h. k ist Rundreise)} \\ 0 & \text{sonst} \end{cases}$$

$\Rightarrow$ g(k) gibt die Kosten der bereits getroffenen Einzelentscheidungen an

(ii) h(k) $:= h_1(k) + h_2(k)$, wobei:

- $h_1(k) := min_{x,y \in V - V(k)}(c(x,0) + c(\pi(j-1), y))$ falls $j < n$
  $\Rightarrow$ $h_1(k)$ ist untere Kostenschranke für eine Kante in den Knoten 0 sowie eine Kante aus dem Knoten j-1

- $h_2(k)$:= Kosten eines billigsten aufspannenden Baumes (MST) B für den Graphen G - V(k) mit Kostenfunktion c (zu ermitteln z.B. mittels Prim-Algorithmus)
  $\Rightarrow$ $h_2(k)$ ist untere Kostenschranke für die Kosten, die durch Vervollständigen der Teilrundreise k zu einer Rundreise im Restgraphen G - V(k) noch entstehen können

Sind g(k) und h(k) wie oben definiert, so erfüllt f(k) = g(k) + h(k) die Forderungen a) und b) an f. Beweis:
zu a) nach Def.
zu b) 1. Fall: k' ist keine Rundreise

$$\text{Sei } k : 0 \to \pi(1) \to \cdots \to \pi(j-1)$$

von $\pi(j-1)$ führt eine Kante e_1 in den Restgraphen V - V(k) und von diesem führt wieder eine Kante e_0 zum Knoten 0

$$\text{Sei } k' : 0 \to \pi(1) \to \cdots \to \pi(j-1) \xrightarrow{e'} \pi(j)$$

von $\pi(j)$ führt eine Kante e'_1 in den Restgraphen V - V(k') und von diesem führt wieder eine Kante e'_0 zum Knoten 0
Es gilt nun:

- g(k') = g(k) + c(e')
- $\to$ i) cost(B) $\leq$ cost(B') + c(e'_1), da B' $\cup\{e'_1\}$ aufspannender Baum von G - V(k) ist
- ii) $c(e_0) + c(e_1) \leq c(e'_0) + c(e')$

Damit lässt sich nun zeigen, dass f(k) $\leq$ f(k'):

- f(k') = g(k') + h_1(k') + h_2(k')
- $\Leftrightarrow$ f(k') = g(k) + c(e') + c(e_0') + c(e_1') + cost(B')
- mit i) und ii) folgt: f(k') $\geq$ g(k) + c(e_0) + c(e_1) + cost (B)
- $\Leftrightarrow$ f(k') $\geq$ f(k)

3.3 B&B im allgemeinen Rahmen

Ein Suchalgorithmus zur Bestimmung einer günstigsten Lösung einer Input-Instanz I eines Problems P, der eine Knotenbewertungsfunktion f verwendet, heißt „Branch&Bound-Algorithmus“ . B&B arbeitet mit der Methode Best First Search (BFS).
Hat f die Form f(k) = g(k) + h(k), wobei g(k) gleich der Summe der Kosten der bisher getroffenen Entscheidungen (= Summe der Kantenkosten auf dem Weg vom Startknoten zu k in T(I)) ist und h(k) eine heuristische Schätzung der Kosten der noch zu treffenden Entscheidungen, um von k zu einem von k erreichbaren „besten“ Zielknoten Z zu gelangen, so nennen wir diesen Algorithmus A*.
A* verwendet eine Priority Queue (PQ) „Offen“ , die folgende Operationen unterstützt:

- insert(k, Offen) fügt k in Offen ein
- k := min(Offen) weist k das kleinste Element aus Offen zu
- deletemin(Offen) löscht das kleinste Element aus Offen

Zusätzlich verwendet A* eine Liste „Closed“ , in der expandierte Knoten verwaltet werden.

Algorithmus A*:

1. insert (s, Offen)
2. repeat
3. k := min(Offen); deletemin(Offen); insert(k, Closed);
4. if k Zielknoten then melde „k ist Zielknoten“
5. else erzeuge alle Söhne k_i von k und setze zeiger(k_i) auf k
6. for i := 1 to n
7. do berechne f(k_i)
8. if $k_i \notin$ Offen and $k_i \notin$ Closed
9. then insert(k_i, Offen)
10. else if (alter Wert von f(k_i)) $> f(k_i)$
11. then zeiger(k_i) auf k;
12. if ($k_i \in$ Closed)
13. then delete (k_i, Closed)
14. insert (k_i, Offen)
15. until empty(Offen) or Zielknoten gefunden

Eigenschaften des A*-Algorithmus':

Zu jedem Zeitpunkt der Rechnung bilden die Knoten aus Closed und Offen zusammen mit s und den aktuellen Zeigern einen gerichteten Baum T. Kantenrichtung ist die entgegen gesetzte Richtung, Blätter(T) = {Offen} und

innere Knoten (T) = s ∪ {Closed}.
Dieser Baum wird auch als aktueller Durchlaufbaum bezeichnet. Die Knoten, die zu früheren Durchlaufbäumen T' gehörten, bilden mit den Knoten aus Offen und Closed den aktuellen B&B-Graphen $G_{akt} \subseteq$ G.
Zulässigkeit von A*:
Bedingungen an G (Suchbaum) und h (Heuristik) garantieren, dass A* angewandt auf die Inputinstanz I immer einen minimalen Kostenweg findet. Die Bedingungen lauten:

1. Jeder Knoten in G hat eine endliche Anzahl an Nachfolgern
2. Alle Kanten in G haben ein positives Gewicht, das größer als ein $\epsilon > 0$ ist
3. Für alle Knoten k des Suchgraphen G gilt h(k) ≤ h*(k), wobei h*(k) die Kosten eines billigsten Weges von k zu Z sind. Man sagt hierfür: „h ist eine zulässige Heuristik“

Theorem 1: Erfüllen G und h die Bedingungen, so bestimmt A* eine optimale Lösung (falls eine existiert).
Lemma: In jedem Schritt vor Terminierung von A* existiert ein Knoten k* in Offen mit den folgenden Eigenschaften:

1. k* liegt auf einem optimalen Weg zu einem Zielknoten
2. A* hat einen optimalen Weg zu k* gefunden
3. f(k*) ≤ f*(s) (= Kosten eines billigsten Weges von s „über s“ zu einem Zielknoten)

Beweis des Lemmas: Mittels vollständiger Induktion bzgl. der Anzahl m der von A* expandierten Knoten.
Ind.-Anfang: m = 1 $\Rightarrow$ k* = s, hierfür gelten offensichtlich alle 3 Eigenschaften.
Ind.-Schritt: Angenommen, die Behauptung gilt nur für m expandierte Knoten. Wenn ein noch zu expandierender Knoten k* sich in Offen befindet und nicht optimal ist, muss er alle drei Bedingungen daher erfüllen. Falls k* optimal ist, werden alle Söhne von k* in Offen eingefügt, wobei einer der Söhne von k* ebenfalls auf einem optimalen Weg sein muss (sonst wäre k* auch nicht optimal). Setzte nun k* auf den optimalen Sohn von k*. Da h eine zulässige Heuristik ist, muss weiterhin gelten: f(k*) $\leq$ f*(k*). k* liegt auf dem optimalen Weg, somit gilt f*(k*) $\leq$ f*(s) und damit auch die dritte Bedingung.
Beweis des Theorems 1:
Teiltheoreme:

- a) A* terminiert, wenn Zielknoten existiert
- b) A* terminert dann in einem optimalen Kostenweg

zu a): Falls A* nicht terminiert, fährt A* immer fort, Knoten auf Offen zu expandieren und wird auch jede vorher gesetzte max_Tiefe überschreiten, aufgrund der Annahme, dass in jedem Knoten ein endlicher Verzweigungsgrad existiert. Da die Kosten jeder Kante $> \epsilon$ ist, werden die g-Werte aller Knoten von Offen den Wert von f*(s) überschreiten. Dies widerspricht dem Lemma.
zu b): A* terminiert entweder, wenn Offen leer ist oder wenn ein Zielknoten erreicht wurde. Angenommen, A* terminiert in einem nicht-optimalen Zielknoten k_z. mit f*(k_z) > f*(s). Bevor A* in k_z expandiert, muss allerdings

nach dem Lemma k* in Offen existieren mit f(k*) $\leq$ f*(s). Dies ist ein Widerspruch.

Theorem 2 (ohne Beweis):

Falls A_1* und A_2* zwei Varianten von A* sind mit $h_1 < h_2$ für alle Nicht-Zielknoten, so nennen wir A_2* besser informiert als A_1*. Dann hat nach Terminierung von A_1* der Algorithmus jeden Zielknoten, den A_2* gefunden hat, auch gefunden.

4 Lokale Suchverfahren

4.1 Hill-Climbing (HC) und Monte-Carlo-HC

HC wird auch als „Greedy-Methode“ bezeichnet.

4.1.1 Einführung in HC

Typische Problemstellung:

Gegeben: Endliche Menge X von Konfigurationen (Zuständen mit einer Kostenfunktion c: X $\rightarrow \Re$)

Gesucht: Globales Minimum $\hat{x}$ bzgl. c $\Rightarrow$ c($\hat{x}$) = min{c(x) | x$\in$ X}

Beispiel HRP/TSP:

Gegeben: Menge S von n Städten, Distanzfunktion dist: S X S $\rightarrow \Re_{>0}$, dist(i,i) = 0

Gesucht: kürzeste Rundreise X, die gestartet bei Stadt 1 alle Städte genau einmal besucht und wieder bei 1 endet

S_n = Menge der Permutationen auf {1,. . . ,n}

Konfigurationsmenge X = {x = $(i_1, i_2, \ldots, i_n, i_1)$ | x ∈ S_n}
Kostenfunktion c = $\sum_{j=1}^{n} cost(i_j, i_{j+1})$ mit $i_{n+1} = i_1$
⇒ |X| = (n-1)!
Auf x Nachbarschaftsrelation G definiert durch G: x $\stackrel{G}{\rightarrow} G_x$ (Sprechweise: x' ∈ G_x ⇒ x' ist Nachbar von x. Interpretiere G als gerichteten Graphen G = (X,A) mit Kantenmenge A := {(x,x') | x ∈ X und x' ∈ G_x}
Beispiel HRP:
Sei x = $(i_1, i_2, \ldots, i_{k-1}, i_k, i_{k+1}, \ldots, i_{l-1}, i_l, i_{l+1}, \ldots, i_n, i_1)$ mit $2 \leq k \leq l \leq n$
Setze $G_x = \{(i_1, i_2, \ldots, i_{k-1}, i_l, i_{k+1}, \ldots, i_{l-1}, i_k, i_{l+1}, \ldots, i_n, i_1)\}$ mit $2 \leq k \leq l \leq n$
⇒ $|G_x| = \binom{n-1}{2} = O(n^2)$

4.1.2 Algorithmus HC

1. Input: Probleminstanz I = (X, G, c) mit
   - X endlicher Konfigurationsraum
   - G Nachbarschaftsrelation
   - c: X → ℜ Kostenfunktion
2. Wähle Startkonfiguration x ∈ X
3. <u>while</u> $\exists x' \in G_x : c(x') < c(x)$ <u>do</u> x = x'

Ausgabe: Lokales Minimum x ∈ X

Die Lösung x, die HC zu I berechnet, ist im Allgemeinen suboptimal. HC findet dieses Minimum in O(|X|) Schritten. Das Ziel ist nun die Überwindung lokaler Minima, ein erster Ansatz ist das Monte-Carlo-HC.

Algorithmus Monte-Carlo-HC:

1. Input: Probleminstanz I = (X, G, c)
2. for i=1 to r
3. do wähle Startkonfiguration $x_i \in$ X
4. berechne mittels HC lokales Minimum $\tilde{x_i}$ zu x_i
5. Bestimme Minimum $\tilde{x}$ aller $\tilde{x_i}$ mit $1 \leq i \leq r$
6. Ausgabe: Konfiguration $\tilde{x}$

Frage: Wie ist r zu wählen, damit $\tilde{x}$ mit Wahrscheinlichkeit $\geq 1 - \alpha$ ($0 \leq \alpha \leq 1$) globales Minimum ist?
Antwort (ohne Beweis): r = $\lceil -ln(\alpha) \cdot p^{-1} \rceil$ mit p = Wahrscheinlichkeit, dass man von einem zufällig gewählten x $\in$ X durch HC zu einem globalen Minimum gelangt

4.2 Probabilistic Hill-Climbing (PHC)

PHC erweitert HC um ein probabilistisches Konzept, wobei Übergänge zu schlechteren Konfigurationen kontrolliert möglich sind. PHC verzichtet weitgehend darauf, problemspezifische Eigenschaften auszunutzen. Vom Nachbarschaftsgraphen G wird allerdings gefordert, dass er stark zusammenhängend ist.

Anmerkung: Die Methode „Simulated Annealing“ (PHC) ist bei einem physikalischen Verfahren, Kristalle aus Siliziumoxid zu züchten, abgeschaut worden. Analogie:

physikalisches System	Optimierungsproblem
Zustand	Konfiguration x
Energie	Kostenfunktion c
Grundzustand	globales Minimum $\tilde{x}$
schnelles Abkühlen	HC
langsames, sorgfältiges Abkühlen	PHC

Algorithmus PHC:

1. Input: Probleminstanz I = (X, G, c), Anfangstemperatur $t_0 > 0$
2. Wähle Startkonfiguration $x_0 \in$ X
3. x = x_0; t = t_0;
4. while Abbruchbedingung nicht erfüllt
5. do while stationäre Wahrscheinlichkeitsverteilung erreicht = False
6. do wähle zufälligen Nachbarn y $\in G_x$
7. berechne Δ_{xy} := c(y) - c(x)
8. if $\Delta_{xy} < 0$ then x = y
9. else setze x = y mit W'keit $e^{\frac{-\Delta_{xy}}{t}}$
10. t = t - d · t (mit 0 < d < 1)

Ausgabe: Konfiguration x
(Bemerkung: $0 < exp(\frac{-\Delta_{xy}}{t}) < 1$)

5 Suche in Spielbäumen

5.1 Spieldefinition

Wir betrachten die Klasse der „endlichen 2-Personen-Nullsummenspiele mit vollständiger Information“ , die im einzelnen folgende Eigenschaften aufweist:

- zwei Spieler Max und Min treffen abwechselnd Entscheidungen
- in jeder Spielsituation gibt es eine endliche Anzahl möglicher nächster Entscheidungen und das Spiel endet nach einer endlichen Entscheidungsfolge
- Terminalknoten K_t sind mit Wert$(K_t) \in \Re$ bewertet, der die Auszahlung an Spieler Max bemisst
- beide Spieler haben vollständige Information, d.h. Kenntnis der bisherigen Entscheidungsfolge

Bemerkung: Knoten von T, wo Max (bzw. Min) am Zuge ist, heißen Max-Knoten (bzw. Min-Knoten).
Definition: Eine Strategie T^+ von Max ist ein Teilbaum von T, der S enthält. Für jeden inneren Max-Knoten K von T^+ enthält T^+ genau einen Sohn K' von K in T und für jeden inneren Min-Knoten K von T^+ enthält T^+ alle Söhne von K in T (analog T^-).
Lemma: T^+ und T^- haben genau ein Blatt K_t gemeinsam, bezeichnet durch

$K_t = (T^+, T^-)$

Das Ziel von Max ist nun eine größtmögliche Auszahlung Wert(T^+, T^-) bei optimaler Gegenstrategie T^- von Min. Analog ist das Ziel von Min eine minimale Auszahlung Wert(T^+, T^-) bei optimaler Gegenstrategie T^+ von Max:

$$(*)Max: \overset{min}{T^-}\ Wert(\hat{T^+}, T^-) = \overset{maxmin}{T^+ T^-}\ Wert(T^+, T^-)$$

$$(**)Min: \overset{max}{T^+}\ Wert(T^+, \hat{T^-}) = \overset{minmax}{T^- T^+}\ Wert(T^+, T^-)$$

Satz von Kuhn: Sind $\hat{T^+}$ und $\hat{T^-}$ wie in (*) und (**) gewählt, so gilt:

$$Wert(T) := Wert(\hat{T^+}, \hat{T^-}) = \overset{maxmin}{T^+ T^-}\ Wert(T^+, T^-) = \overset{minmax}{T^- T^+}\ Wert(T^+, T^-)$$

Wert(T) heißt Wert des Spiels T. (Beweis durch Induktion über Tiefe(T))

5.2 Bestimmung des Spielwerts von T

Wir berechnen für jeden Knoten K den „Minimax“ -Wert V(K). Dabei ist V(K) wie folgt definiert:

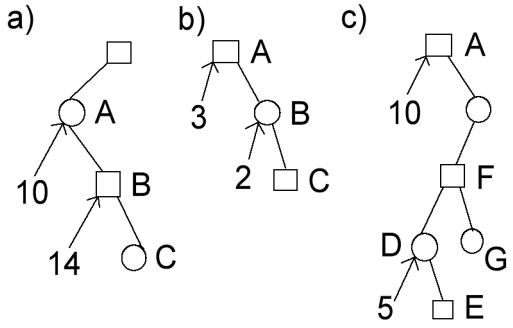
- K ist Blatt $\Rightarrow$ V(K) = Wert(K)
- K ist kein Blatt, hat Söhne $K_1, \ldots, K_m$ und ist Max_Knoten $\Rightarrow$ V(K) = max{V(K_1), …, V(K_m)}
- K ist kein Blatt, hat Söhne $K_1, \ldots, K_m$ und ist Min_Knoten $\Rightarrow$ V(K) = min{V(K_1), …, V(K_m)}

Der Minimax-Wert der Wurzel S V(S) ist dann gleich dem Spielwert von T. Mittels eines geeigneten Algorithmus' kann der Spielwert in O($|T|$) berechnet werden.

5.3 $\alpha - \beta$−Suche

Es ist sehr häufig überflüssig, ganz T zu durchsuchen, um V(S) zu berechnen.

Betrachte folgende Konstellationen in Spielbäumen:



- Fall a): C braucht nicht mehr untersucht zu werden, da V(A) $\leq$ 10 („β-cutoff“)
- Fall b): C braucht nicht mehr untersucht zu werden, da V(A) $\geq$ 3 („α-cutoff“)
- Fall c): E braucht nicht mehr untersucht zu werden, da V(D) $\leq$ 5 („deep-cutoff“)

Das Abschneiden von Teilbäumen (pruning) und das Aktualisieren der $\alpha-\beta$-Werte (bound updating) ist im nachfolgenden Algorithmus beschrieben in Form einer rekursiven Prozedur V(k, α, β) mit k = Knoten und $\alpha, \beta \in \Re(\alpha \leq \beta)$. Dabei gilt:

$$V(k,\alpha,\beta) := \begin{cases} V(k) & \text{falls } \alpha < V(k) < \beta \\ \alpha & \text{falls } V(k) \leq \alpha \\ \beta & \text{falls } V(k) \geq \beta \end{cases}$$

Der Minimax-Wert V(S) der Wurzel S von T wird durch den Aufruf V(S, -∞, +∞) bestimmt.

Algorithmus Alpha-Beta:

1. function V(k, α, β): Real;
2. L := Anzahl Söhne von k
3. if k Terminalknoten then V := Wert(k);
4. else for i := 1 to L do
5. VC := V(K_i, α, β)
6. if k Max_Knoten and VC > α then α := VC;
7. if k Min_Knoten and VC < β then β := VC;
8. if $\alpha \geq \beta$ then return(VC); //cutoff
9. if k Max_Knoten then V := α else V := β;

Das Laufzeitverhalten von Alpha-Beta wird stark von der Reihenfolge, in der die Blätter dursucht werden, bestimmt. Frage: Wieviele Söhne eines Knotens sind durchschnittlich zu durchsuchen, bis es zu einem cutoff kommt?
Seien A und B Min_Knoten und V(A) bekannt. B hat Söhne B_1 bis B_n. Dann kommt es zu einem α-cutoff bei B_i, wenn für B_1 bis B_{i-1} gilt: V(B_j) > V(A) mit $1 \leq j \leq i-1$ und V(B_i) $\leq$ V(A).
Annahme: Mit Wahrscheinlichkeit p ist V(B_j) > V(A) unabhängig von j.
$\Rightarrow$ Die mittlere Anzahl zu berechnender Söhne von L Söhnen $B_1, \ldots, B_L$ bis zum ersten α-cutoff ist:

$$D(L,p) = \sum_{k=1}^{L} k \cdot p^{k-1} \cdot (1-p) = (1-p) \cdot \sum_{k=1}^{L} k \cdot p^{k-1} \leq \frac{1}{1-p}$$

5.4 0-1-wertige Spielbäume

$1 \hat{=}$ Max gewinnt

$0 \hat{=}$ Max verliert

Algorithmus Solve:

```
1. function V(J := Knoten in T): {0,1}
2. if J Blatt(T) then return Wert(J);
3.         else if J Max_Knoten
4.                   then while ∃ unbesichtigter Sohn J' von J
5.                                do if V(J') = 1 then return 1
6.                         return 0
7.                   else while ∃ unbesichtigter Sohn J' von J
8.                                do if V(J') = 0 then return 0
9.                         return 1
```

Aufruf im Hauptprogramm: V(S)

Gesucht: Wahrscheinlichkeit, mit der Max ein Spiel mit zufälligen 0-1-Blattbelegungen gewinnt.

Seien b,d $\in \aleph$ (d gerade). T(d,b) bezeichnet den vollständigen Baum der Tiefe d mit Verzweigungsgrad b. S (= Wurzel von T) sei ein Max_Knoten.

T(d,b,p_0) (mit $0 < p_0 < 1$) sei die Menge aller Spielbäume der Gestalt T(d,b), deren b^d Blätter unabhängig mit 0 und 1 bewertet sind, wobei für ein Blatt

k gilt: Prob(Wert(k) = 1) = p_0
⇒ Wahrscheinlichkeit, einen Spielbaum T(d,b,p_0) zu ziehen ist:

$$p_0^{\#1-Blätter} \cdot (1-p_0)^{\#0-Blätter}$$

Unter allen Algorithmen, die für T ∈ T(d,b,p_0) den Minimax-Wert V(S) berechnen, hat Solve die kürzeste erwartete Laufzeit.

6 Erlernbare Klassen / Konzept-Erlernbarkeit

6.1 Einleitung

Gesucht: Klasse von Konzepten, die in polynomieller Zeit von Lernprotokollen erlernt werden können.
Bemerkung: Die Konzeptklassen, die wir hier untersuchen, sind durch spezielle Programme (hier: Boole'sche Formeln) definiert.
Lernprotokolle für Boole'sche Formeln:

- Seien $p_1, \ldots, p_n$ Boole'sche Variablen mit $p_i \in \{0,1\}$, V = $\{p_1, \ldots, p_n\}$
- t: V → $\{0,1,*\}^n$ sei Belegung, wobei t(p_i) = * bedeutet, dass p_i undefiniert unter t ist
- Boole'sche Funktion F: $\{0,1\}^n \rightarrow \{0,1\}$ heißt Konzept, falls F auf $\{0,1,*\}$ definiert ist und gilt: F(t) = 1 ⇔ F(t') = 1 für alle Belegungen t', die mit den in t definierten Stellen übereinstimmen
- Wahrscheinlichkeitsverteilung D auf der Menge T = $\{t \in \{0,1,*\}^n$ mit F(t) = 1$\}$ gibt die relative Häufigkeit an, mit der t ∈ T erzeugt wird ⇒ $\sum_{t \in T} D(t) = 1$

- Lernprotokolle geben dem Lernenden Zugriff auf die nachfolgenden Prozeduren:

  1. Examples:
     - wird ohne Input-Parameter aufgerufen
     - liefert mit Wahrscheinlichkeit D(t) eine Belegung t ∈ T (d.h. F(t) = 1) pro Aufruf
  2. Oracle(t):
     - wird mit Input-Parameter t aufgerufen, gibt F(t) zurück
     - Oracle ist typischerweise eine Datenbank und wird mit kritischen Werten aufgerufen

Definition Erlernbarkeit: Eine Klasse X von Programmen (z.B. Boole'schen Funktionen) mit einer Menge $p_1, \ldots, p_n$ von Boole'schen Variablen als Input und einer Ausgabe $\in \{0, 1, *\}$ heißt erlernbar bzgl. eines gegebenen Lernprotokolls, falls es einen Algorithmus A gibt, der das Protokoll aufruft und folgende Eigenschaften hat:

1. polynomielle Laufzeit, beschränkt in einem justierbaren Parameter h, der Variablenzahl n und in den Längenparametern des zu lernenden Programms
2. für alle f ∈ X und alle D auf T deduziert A mit Wahrscheinlichkeit $\geq 1 - \frac{1}{h}$ ein Programm g ∈ X mit folgenden Eingenschaften:
   (a) $\forall t : g(X) = 1 \rightarrow f(X) = 1$
   (b) $\sum_{t \in T, g(t) \neq 1} D(t) \leq \frac{1}{h}$

Erfüllt A die Eigenschaften 1 und 2, so sprechen wir von „Lernen bei einseitiger Fehlerbegrenzung“ .
Falls 2.a) wie folgt ersetzt wird:
„Sei $\overline{D}$ Wahrscheinlichkeitsverteilung auf $\overline{T} = \{0, 1, *\}^n - T$ und g(X) erfüllt neben b) die Eigenschaft $\sum_{t \in \overline{T}, g(t)=1} \overline{D}(t) \leq \frac{1}{h}$“
dann spricht man von Lernen mit zweiseitiger Fehlerbegrenzung.

Beispiel:
$X = \{q_{i1}, \ldots, q_{is} : q_{ij} \in \{p_{ij}, \overline{p_{ij}}\}, 1 \leq i_1 < i_2 < \cdots \leq n, j > 0\}$
= Menge der Konjunktionen über V
$|X| = 3^n$

Erlernen von $f = p_1$

A erhält durch wiederholten Aufruf von Examples eine Belegung t ∈ T, die durch D gesteuert wird. D sei derart, dass mit hoher Wahrscheinlichkeit Belegungen t ∈ T generiert werden, die mit (1,1,...) beginnen.
Falls A nur solche Beispiele gesehen hat, würde A $f = (p_1, p_2)$ deduzieren.
Gegenmaßnahme: Oracle-Aufrufe mit (1,0,...).

6.2 Kombinatorische Überlegungen

Eine Folge unabhängiger Einzelexperimente heißt Bernoulli-Experiment, falls jedes Einzelexperiment nur zwei mögliche Ergebnisse (Erfolg = 1, Misserfolg = 0) mit den Eintrittswahrscheinlichkeiten p für 1 und (1-p) für 0 hat (Ziehen mit Zurücklegen). Ein Bernoulli-Experiment der Länge l besteht aus l Einzelexperimenten.
Sei $h \in \Re^{>1}$ und $s \in \aleph$, dann bezeichnet L(h,s) = min {l: in einem Bernoulli-Experiment der Länge l, in dem Erfolg mit Wahrscheinlichkeit $\frac{1}{h}$ eintritt, ist

die Wahrscheinlichkeit, dass weniger als s-mal das Ereignis Erfolg gezogen wurde, $< \frac{1}{h}\}$.

Satz: $\forall s \geq 1 \wedge h > 1 : L(h,s) \leq 2h \cdot (s + ln(h))$

Beweis:

Bekannte Schranken aus der Analysis:

1. $\forall x > 0 : (1 + \frac{1}{x})^x < e$
2. $\forall x > 0 : (1 - \frac{1}{x})^x < \frac{1}{e}$
3. (Chernoff-Schranke:) Die Wahrscheinlichkeit, dass in einem Bernoulli-Experiment der Länge m mit Wahrscheinlichkeit p für Erfolg höchstens k-mal Erfolg gezogen wird, wobei $k < mp$, ist:

   $$\leq \left(\frac{m - mp}{m - k}\right)^{m-k} \cdot \left(\frac{mp}{k}\right)^k \qquad (**)$$

   durch Umformung und aus 2. folgt:

   $$(**) < e^{-mp+k} \cdot \left(\frac{mp}{k}\right)^k$$

   weitere Substitution ergibt:

   $$(**) < \frac{1}{h}$$

Beispiel: Examples liefert positive Belegungen t für ein Programm F. Bestimme Approximation der Menge P der Variablen von F, wobei jede Variable in P mindestens in einem positiven Beispiel t ∈ T (d.h. F(t) = 1) vorkommt (P = Menge der relevanten Variablen für F).

⇒ Vorgehen: Erzeuge L(h,$|P|$) positive Belegungen mit Examples und berechne die Menge P' der Variablen, die durch die so erzeugten positiven Beispiele bestimmt sind. Nach Definition von L(h,$|P|$) hat P' dann mit Wahrscheinlichkeit $\geq 1 - \frac{1}{h}$ folgende Eigenschaft:

Wird ein weiteres positives Beispiel t mit Examples generiert, so ist die Wahrscheinlichkeit, dass t eine Variable aus P - P' definiert, $< \frac{1}{h}$.

6.3 Erlernbarkeit von k-CNF-Formeln

Wir betrachten hier ausschließlich CNF-Formeln mit $\leq k$ Literalen pro Klausel. Elemente aus k-CNF heißen auch „Konzepte“ . Sei k-CNF =

$$\bigcup_{r>0} \left[\bigcup_{s \leq k} cl(n,s) \right]^T$$

Satz: Für jedes k $\in \aleph$ ist die Klasse der k-CNF-Ausdrücke erlernbar mit Hilfe des Algorithmus A, der $\tilde{L} = L[h, (2n)^{k+1}]$ Aufrufe von Examples und keinen von Oracle macht.
Der Algorithmus startet mit Formel g, die Produkt aller möglichen Klauseln der Länge $\leq k$ über der Literalmenge L ist. Dann gibt es höchstens

$$\sum_{i=0}^{k} \binom{2n}{i} \leq \sum_{i=0}^{k} (2n)^i < (2n)^{k+1}$$

Produktterme in g.

- A ruft dann Examples $\tilde{L}$-mal auf und liefern $\tilde{L}$ positive Beispiele der das zu erlernende Konzept repräsentierenden Formel f $\in$ k-CNF
- jedes positive Beispiel t von f definiert eindeutig eine Literalmenge $L_t \subseteq L$, wobei $p_i \in L_t$, falls i-ter Eintrag in t=1 und $\overline{p_i} \in L_t$, falls i-ter Eintrag in t=0 ist
- streiche aus der verbleibenden Klauselmenge von g jede Klausel M, für die gilt: $M \cap L_t = \emptyset$, denn in diesem Fall gilt: $L_t \not\Rightarrow M$

Unser Deduktionsalgorithmus A, der k-CNF erlernt, sieht so aus:

1. berechne g
2. for i = 1 to $L(h, 2n)^{k+1}$) do
3. t := Examples
4. for all M $\in$ g do
5. if $L_t \not\Rightarrow M$ then streiche M aus g
6. gib erlernte Formel g aus

Zeige: Die Klasse k-CNF wird von A gelernt.
Während der Berechnung von A wird g sukzessive verändert. Dabei erfüllt das aktuell berechnete g (g_{akt}) folgende Eigenschaft:

$$\{t : L_t \Rightarrow g_{akt}\} \subseteq \{t : L_t \Rightarrow f\}$$

Beweis:
Für die Startformel g_0 gilt: $\{t : L_t \Rightarrow g_0\} = \emptyset$.
Sei B := $\pi M(M \in g : f \Rightarrow M)$.

- $\Rightarrow$ B $\subseteq g_{akt}$ zu jedem Zeitpunkt der Rechnung
- $\Rightarrow \{t : L_t \Rightarrow g_{akt}\} \subseteq \{t : L_t \Rightarrow B\}$
- $\Rightarrow$ Nach Konstruktion von B: f $\Rightarrow$ B
- $\Rightarrow$ B $\Rightarrow$ f

Zu zeigen: Ausgabe g von A erfüllt Definition von Erlernbarkeit.

$$X_{g_{akt}} := \sum_{t:L_t \Rightarrow f, L_t \not\Rightarrow g_{akt}} D(t)$$

ist im Laufe der Rechnung monoton fallend.
Fall 1: Im Laufe der Rechnung gilt für ein $X_{g_{akt}} \leq \frac{1}{h} \Rightarrow$ fertig.
Fall 2: Für jedes $X_{g_{akt}}$ gilt: $X_{g_{akt}} > \frac{1}{h}$.
Aufgrund der Definition von $L[h, (2n)^{k+1}]$ angewandt auf die Bernoulli-Experimente von A ist die Wahrscheinlichkeit, dass bei einer Erfolgswahrscheinlichkeit von $> \frac{1}{h}$ pro Einzelexperiment, weniger als $(2n)^{k+1}$ Klauseln aus der Startformel g_0 eliminiert werden, $< \frac{1}{h}$.

7 Komplexitätsklassen

7.1 Die Klasse P

Probleme der Klasse P sind polynomzeitbeschränkt und gelten als „effizient lösbare Probleme“ (J. Edmonds).
Beispiel: Linear Programming (LP)
Instanz: Ganzzahlige Vektoren $V_i = (v_i[1], \ldots, v_i[n])1 \leq i \leq n$ und $D = (d[1], \ldots, d[n])$ sowie $C = (c[1], \ldots, c[n])$ und ganze Zahl B.
Frage: Existiert ein Vektor $x = [x_1, \ldots, x_n]$ von rationalen Zahlen, sodass $C \cdot x > B$ und $V_i \cdot x < d[i]$ für alle $i : 1 \leq i \leq n$
LP ist in P.

Problemreduktion:
Definition: Falls x und y zwei Entscheidungsprobleme sind, so ist eine Transformation von x nach y eine DTM-berechenbare Funktion f: $\{0,1\}^* \rightarrow \{0,1\}^*$, sodass x eine Antwort „ja“ unter x hat, gdw. f(x) eine Antwort „ja“ unter y hat.
Bemerkung: Der Reduktionsbegriff kann benutzt werden, um Nicht-Mitgliedschaft

zu einer Komplexitätsklasse zu zeigen; man benötigt ein Problem M, welches nicht in der Klasse enthalten ist und kann die Nicht-Mitgliedschaft eines unbekannten Problems durch Transformation auf M zeigen.

Vollständigkeit Definition: Angenommen, wir haben ein Problem X, eine Klasse C von Problemen und eine Klasse R von Reduktionen; falls $Y \subseteq_R X$ für alle $Y \in C$, dann bezeichnen wir X als hartes Problem bzgl. C (z.B. ist TSP NP-hart). Falls $X \in C$, dann bezeichnen wir X als vollständiges Problem von C.

Die Klasse P enthält die Klasse L der Logspace-berechenbaren Probleme (z.B. Palindrome-Berechnung). Allgemein gilt:

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE \hat{=} NPSPACE$$

Als echte Untermenge ist lediglich bewiesen, dass $L \subset PSPACE$.

7.2 Die Klasse NP

Die Klasse NP ist definiert als die Menge der lösbaren Entscheidungsprobleme, die durch eine NTM in Polynomzeit gelöst werden. Interpretation: „Rate eine Lösung und verifiziere in Polynomzeit“ .
Das Erfüllbarkeitsproblem (ERF) ist NP-vollständig.
Beispiel 1: Not-all-equal-ERF
Instanz: Gegebene Boolesche Formel in 3-KNF.
Frage: Existiert eine erfüllende Belegung, sodass in jeder Klausel weder alle Literale False oder True gesetzt sind?
Not-all-equal-ERF ist NP-vollständig.
Beispiel 2: 3-Färbungsproblem:

Instanz: Graph G = (V,E)

Frage: Existiert eine 3-Färbung der Knotenmenge V, so dass jede Farbklasse eine Kante erhält?

Zu zeigen: 3-Färbungsproblem ist NP-vollständig.

1.) Problem ist in NP $\surd$

2.) Vollständigkeit: Reduktion auf not-all-equal-ERF:

Gegeben eine Instanz des Problems nae-ERF, in der n Klauseln mit maximal je 3 Literalen enthalten sind. Dann zeigt eine Reduktion, dass eine „ja“ -Instanz des 3-Färbungsproblems äquivalent ist zu einer „ja“ -Instanz für das nae-ERF.

$\Rightarrow$ 3-Färbungsproblem ist NP-vollständig.

7.3 Die Klasse PSPACE mit Schwerpunkt auf Spielprobleme

Definition: PSPACE ist die Menge aller Sprachen, die auf einer polynomplatzbeschränkten, deterministischen Turing-Maschine erkannt werden.

Bemerkung: Savitch hat bewiesen, dass NPSPACE = PSPACE gilt.

Die Rolle des ERF für die Klasse NP hat das „Quantifizierte ERF“ für die Klasse PSPACE. 1973 wurde gezeigt, dass QBF-ERF PSPACE-vollständig ist.

QBF-ERF:

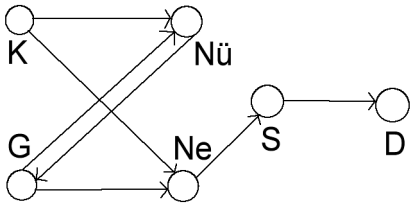
Gegeben: Boolesche Formel F in KNF und Boolesche Variablen $x_1 \dots x_n$.

Frage: Gibt es eine Wahrheitsbelegung für x_1, so dass für alle Belegungen von x_2 eine Belegung x_3 existiert, so dass ... (alternierende Existenz- und Allquantoren); formal:

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \dots Q_n x_n = F$$

Diese Klasse enthält viele Spielprobleme (2-Personen-Spiele). Auch QBF kann als 2-Personen-Spiel interpretiert werden mit Spieler $\exists$ und Spieler $\forall$. Ein Zug besteht daraus, den Wahrheitswert der nächsten Variablen festzulegen. Das Spiel ist nach n Schritten beendet und es gewinnt $\exists$, falls F immer wahr ist. Ansonsten gewinnt $\forall$. D.h. das Problem ist die Existenzfrage einer Gewinnstrategie für $\exists$.
Weitere Beispiele: Nimm, Tic-Tac-Toe, Schach, Dame, Go, Geography

Nähere Betrachtung des Beispiels Geography:
Gegeben: Zwei Spieler und alternierend werden Städtenamen angegeben, so dass die nächste Stadt mit dem letzten Buchstaben der vorherigen Stadt beginnt. Wer keine neue Stadt mehr weiß, verliert.
Beispiel: Städtepool Köln, Nürnberg, Giessen, Neuss, „Schwerind“ , Düsseldorf:



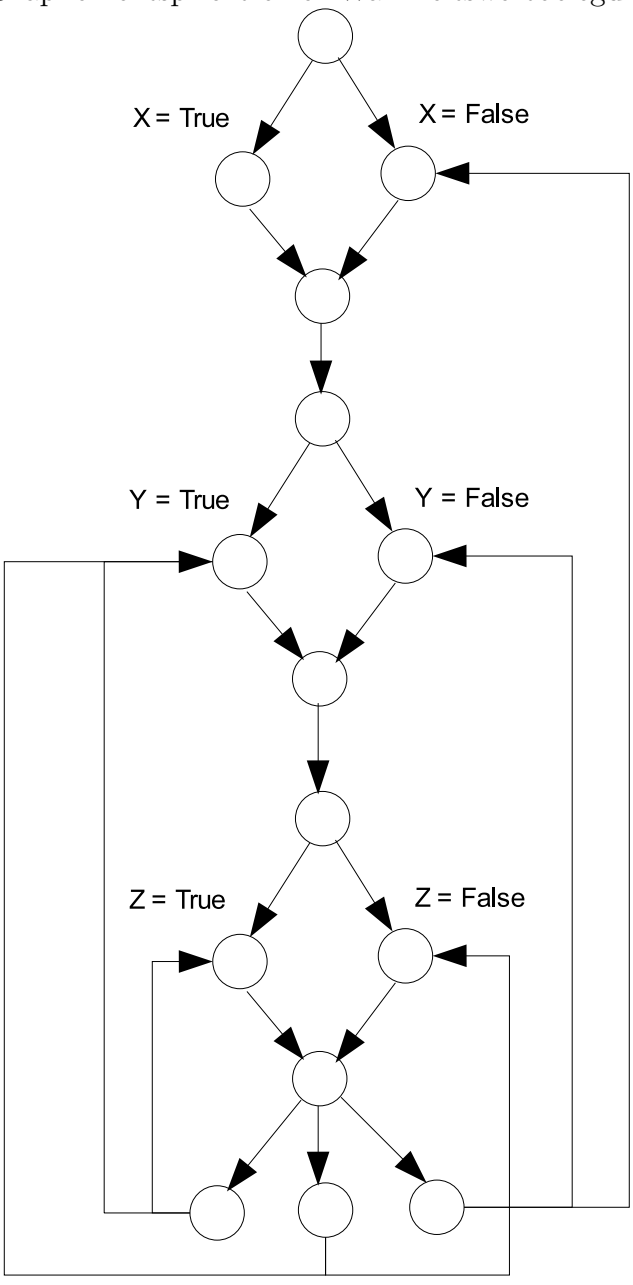
Umformulierung des Spiels:
Gegeben sei ein gerichteter Graph G = (V,E), dessen Knoten alle Städte der Welt darstellen sollen. so dass zwei Städte i und j durch eine Kante (i,j) verbunden sind, falls der letzte Buchstabe der Stadt i gleich dem Anfangsbuchstaben der Stadt j ist. Alternierend werden Knoten ausgewählt, so dass ein nicht verlängerbarer Weg entsteht. Gewonnen hat der, der den letzten Knoten gewählt hat.
Frage: Existiert eine Gewinnstrategie für Spieler 1 (= Spieler $\exists$)?

Satz: Geography ist PSPACE-vollständig.

1.) Geography ist in PSPACE $\surd$

2.) Reduktion von QBF-ERF auf Geography:

Beispiel-Konstruktion: F $= \exists x \forall y \exists z[(\overline{x} \vee \overline{y}) \wedge (y \vee z) \wedge (y \vee \overline{z})]$ Ein Weg im Graphen entspricht einer Wahrheitswertbelegung der Variablen:



Dabei werden die Knoten und Kanten gegensätzlich zur Wahl der Spieler markiert. Z.B. setze Spieler 1 die Variable x = true, dann wird im Graphen x = false markiert. O.B.d.A. gehen wir im Folgenden davon aus, dass Spieler 2 (= Spieler $\forall$) nach der letzten Variablenzuweisung am Zug ist. Er kann sich nun zwischen allen Klauseln von F entscheiden. Wählt er eine Klausel, die von der (gegensätzlichen) Belegung erfüllt wird, so kann Spieler 1 aufgrund der Konstruktion nicht mehr ziehen und hat verloren.
Beispiel: Gewählt wurde der Weg x, y und $\overline{z}$. Daher wurden $\overline{x}$, $\overline{y}$ und z markiert. Spieler 2 wählt nun die Klausel $(\overline{x} \vee \overline{y})$ und hat damit gewonnen, da beide Knoten bereits markiert wurden.