NP-Probleme

Hans Jürgen Ohlbach

Keywords:

Empfohlene Vorkenntnisse: Turingmaschine

Inhaltsverzeichnis

1 Einleitung

In diesem Miniskript geht es um Algorithmenprobleme, die noch vor gar nicht langer Zeit als unlösbar galten, und auch heute noch extrem „Harte Nüsse“ sind. Aus Sicht der Theoretische Informatik sind aber weniger die Probleme selbst, sondern die Untersuchungsmethoden das Interessante. Wir werden Techniken kennenlernen, mit denen man auch neue Probleme daraufhin untersuchen kann, wie schwierig sie sind.

Konkret geht es um die Klasse der NP-vollständigen Probleme. NP steht für *Nichtdeterministisch Polynomial*. Das bedeutet, wenn man eine Lösung des Problems *rät* (nichtdeterministisch), dann kann man in *polynomieller* Zeit testen, ob die Lösung korrekt ist. Da unsere Computer ja keine Hellseher sind, bedeutet das in der Praxis, dass man die richtige Lösung mit passenden Suchverfahren *suchen* muss.

Das „vollständig“ in NP-vollständig bedeutet, dass sich alle Problem in dieser Klasse ineinander transformieren lassen. D.h. ein Lösungsverfahren eines Problems A in dieser Klasse kann für alle anderen Probleme B in dieser Klasse benutzt werden, indem man ein B-Problem in ein A-Problem transformiert, dort löst, und dann die Lösung wieder zurücktransformiert. Aus Effizienzgründen macht man das i.A. nicht so, sondern man entwickelt für jede Problemklasse entsprechend angepasste Algorithmen. Allerdings sind die Prinzipien, nach denen diese Algorithmen funktionieren, immer die gleichen. Alle müssen eine Lösung suchen, d.h. so lange alternative Lösungskandidaten testen, bis die die richtige gefunden haben. Da es i.A. exponentiell viele Lösungskandidaten gibt, ist das sehr aufwendig, und man würde gerne Algorithmen haben, die zielgerichtet in polynomieller Zeit die richtige Lösung finden.

Bisher ist es noch niemanden gelungen, solch ein Verfahren zu finden, oder zu beweisen, dass es das nicht geben kann. Das Problem: „gibt es ein solches Verfahren, oder gibt es keines“, ist als P/NP-Problem in die Informatikgeschichte eingegangen. Es ist eines der sog. Milleniumprobleme, für deren Lösung es 1 Million $ gibt.

Zunächst machen wir einen kleine Exkurs in die Komplexitätstheorie, um die grundlegenden Begriffe zu verstehen, und damit auch die Probleme selber genauer zu verstehen. Dann führen wir die „Mutter“ aller NP-vollständigen Probleme ein, das SAT-Problem. Um zu zeigen, dass das SAT-Problem NP-vollständig ist, muss man zeigen, wie man beliebige NP-Algorithmen in SAT-Probleme umkodiert. Da man das mittels Turingmaschinen macht, sollte unbedingt die Theorie der Turingmaschine bekannt sein. Ausgehend von dem SAT-Problem kann man für eine ganze Reihe weiterer Probleme die NP-Vollständigkeit zeigen. Für die Beispiele von NP-vollständigen Problemen, die in Kap. 5 gezeigt werden, sind die NP-Vollständigkeitsbeweise komplex, und nicht jeder Leser muss sich da durch kämpfen. Ein NP-Vollständigkeitsbeweis für ein neues Problem ist aber insofern extrem nützlich, als man davor bewahrt wird, nach höchstwahrscheinlich nicht existierenden effizienten Algorithmen zu suchen.

In den letzten beiden Kapitel schließlich werden konkrete Algorithmenansätze zur Lösung dieser Probleme vorgestellt.

2 Exkurs in Komplexitätstheorie

In der Komplexitätstheorie untersucht man Algorithmen und Probleme danach, wieviel Speicherplatz und wieviel Zeit sie benötigen, und zwar in Abhängigkeit von der Größe der Eingabe.

Die Komplexität eines Problems und die Komplexität eines Algorithmus hängen folgendermaßen zusammen:

Definition 1 (Komplexität von Problemen) *Die Komplexität eines Problems ist die Komplexität des* bestmöglichen *Algorithmus, der dieses Problem löst.*

Hat man also eine Algorithmus für ein Problem P, und dessen Komplexität analysiert, heißt das noch lange nicht, dass das auch die Komplexität des Problems ist. Es könnte ja noch einen besseren Algorithmus geben, der das Problem schneller löst oder mit weniger Speicherplatz. Um von der Komplexität eines Algorithmus auf die Komplexität des Problems zu schließen, muss man also nachweisen, dass es keinen besseren Algorithmus für dieses Problem geben kann. Die Komplexitätstheorie stellt Techniken bereit, um solche Nachweise zu führen.

Eingaben: Die Komplexitätsangaben orientieren sich an der Größe der Eingaben. Dafür gibt es allerdings verschiedene Maße. Ist die Eingabe eine Zeichenkette, dann ist die Größe einfach die Länge der Zeichenkette. Ist die Eingabe allerdings eine oder mehrere natürliche Zahlen, dann gibt es verschiedene Möglichkeiten, deren Größe zu messen:

unär: Die Größe ist die Zahl selbst. Falls die Zahl n ist, dann entspräche das z.B. n Striche als Eingabe, oder n Felder auf dem Band einer Turingmaschine.

binär: Die Größe der Eingabe entspricht der Anzahl Bits in der Binärdarstellung der Zahl. Für eine Zahl n braucht man ca. $\log_2(n)$ Bits. Zwischen unär und binär ist also ein Faktor $\log_2(n)$.

konstant: Wenn die Verarbeitung der Zahlen unabhängig von ihrer Größe ist, dann kommt es nur auf die Anzahl der Eingaben an, nicht auf deren Größe.

Verarbeitung: Um die Komplexität zu bestimmen kann man entweder den maximal benötigten Speicherplatz, oder die maximal benötigte Zeit bestimmen. Als Speicherplatz zählt man die elementaren Speichereinheiten, die zur Verfügung sind. Bei der Turingmaschine sind es die Zellen auf dem Band. In unseren heutigen Computern würde man die Bytes oder die Worte zählen. Man kann auch abstrakter bleiben, und irgendwelche abstrakten Basiseinheiten annehmen.

Für die Zeitmessung zählt man die Anzahl Operationen, von denen man annimmt, dass sie immer gleich viel Zeit brauchen. In der Turingmaschine sind es die Einzelschritte der Maschine. In unseren heutigen Computern wären es die Maschinenbefehle. Auch hier kann man abstrakter bleiben, und abstrakte Elementarschritte zählen, von denen man annehmen kann, dass sie gleich viel Zeit brauchen.

Auch hier bildet die Verarbeitung von Zahlen wieder ein Problem. Passen die Zahlen in ein feste Anzahl von Bits, dann zählt deren Verarbeitung, z.B. die Zuweisungsoperation i.A. als einen Schritt.

Verarbeitet man allerdings beliebig große Zahlen (in Java BigInteger), dann erfordert schon eine Zuweisungsoperation mehrere Schritte, abhängig von der Größe der Zahl.

Wenn man also Komplexitätsangaben liest, hängt deren Interpretation u.U. davon ab, wie genau gezählt wird.

Wachstumsverhalten: Man möchte also für eine Eingabe der Größe n den Berechnungsaufwand (Zeit oder Platz) als Funktion $f(n)$ bestimmen. Als Beispiel könnte für Algorithmus A die Funktion f sein: $f_A(n) = n^2$, und für einen Algorithmus B: $f_B(n) = n^2 + n$. Für kleine n macht das einen Unterschied. Je größer das n wird, desto kleiner wird allerdings der Unterschied.

Woran man daher viel mehr interessiert ist, ist das *Wachstumsverhalten*. Was passiert, wenn man z.B. die Eingabegröße verdoppelt? Verdoppelt man n von 100 auf 200, dann verändert sich f_A im Beispiel oben von 100000 auf 400000. f_B verändert sich von 100100 auf 400200. Der relative Unterschied ist minimal und wird immer kleiner, je größer das n wird. Daher spielt für das Wachstumsverhalten in der Funktion $f_B(n) = n^2+n$ der lineare Summand n eigentlich keine Rolle, und wird gerne weggelassen. Konstante Faktoren, wie z.B. bei $f(n) = 2n$ oder $f(n) = 4n$ spielen für das Wachstumsverhalten schon gar keine Rolle. In beiden Fällen bedeutet die Verdopplung von n die Vervierfachung von $f(n)$. Daher lässt man konstante Faktoren ebenfalls weg.

Landau-Notation: Funktionen, die asymptotisch (für $n \mapsto \infty$) das gleiche Wachstumsverhalten zeigen, werden daher in eine Klasse zusammengefasst. Mit der sog. *Landau Notation* schreibt man das so: für eine Funktion $g(n)$ bezeichnet $\mathcal{O}(g)$ die Menge aller Funktionen, die das gleiche Wachstumsverhalten wie g haben. Formal:

$$f \in \mathcal{O}(g) \text{ gdw. } \limsup_{n\mapsto\infty} \frac{|f(x)|}{|g(x)|} < \infty$$

$\limsup$ ist dabei der *größte Häufungspunkt*. Das hat eine Bedeutung wenn die Funktion oszilliert. Bei monton wachsenden Funktionen, wie es bei den Komplexitätsberechnungen i.A. der Fall ist, ist $\limsup$ einfach der normale Grenzwert.

Beispiele:

Für eine Konstante k ist $kn^2 \in \mathcal{O}(n^2)$, da $\limsup_{n\mapsto\infty} \frac{|kn^2|}{|n^2|} = k < \infty$

Des weiteren ist $n^2 + kn \in \mathcal{O}(n^2)$, da $\limsup_{n\mapsto\infty} \frac{|n^2 + kn|}{|n^2|} = \limsup_{n\mapsto\infty} 1 + \frac{|k|}{|n|} = 1 < \infty.$

Für ein beliebiges Polygon $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$ gilt
$f \in \mathcal{O}(n^k)$, da $\limsup_{n\mapsto\infty} \frac{|a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0|}{|n^k|} = \limsup_{n\mapsto\infty} a_k + a_{k-1}\frac{1}{|n|} + \ldots a_0 \frac{1}{|n^k|} = a_k < \infty$

Bei Polynomen reicht also der höchste Exponent für Komplexitätsangaben. Im allgemeinen ist es der am schnellsten wachsende Term, der für die Komplexitätsangaben relevant ist.

Ein Gefühl für die unterschiedlichen Wachstumsverhalten bekommt man mit der folgenden Tabelle. Die letzte Spalte listet Beispielalgorithmen, die die entsprechende *Zeitkomplexität* aufweisen.

Notation	Bedeutung	anschauliche Erklärung	Beispielalgorithmus
$f \in \mathcal{O}(1)$	f ist beschränkt	f ist unabhängig von der Größe der Eingabe	Zugriff auf ein Arrayelement
$f \in \mathcal{O}(\log(n))$	f wächst logarithmisch	f wächst fast konstant wenn sich die Eingabe verdoppelt	Binäre Suche im sortieren Array
$f \in \mathcal{O}(\sqrt{(n)})$	f wächst als Wurzelfunktion	f wächst um das Doppelte wenn sich die Eingabe vervierfacht	Primzahltest mittels Division
$f \in \mathcal{O}(n)$	f wächst linear	f wächst proportional zur Eingabe	Suche in unsortierter Liste
$f \in \mathcal{O}(n \log(n))$	f wächst superlinear	f wächst ganz leicht überproportional zur Eingabe	Effiziente Suchalgorithmen
$f \in \mathcal{O}(n^2)$	f wächst quadratisch	f vervierfacht sich bei Verdopplung der Eingabe	Suchalgorithmen wie Selection Sort
$f \in \mathcal{O}(n^3)$	f wächst kubisch	f verachtfacht sich bei Verdopplung der Eingabe	CYK-Algorithmus für Typ 2-Sprachen
$f \in \mathcal{O}(2^n)$	f wächst exponentiell	f verdoppelt sich bei einer Eingabe mehr	Algorithmen für das SAT-Problem
$f \in \mathcal{O}(n!)$	f wächst faktoriell	f wächst um $n+1$ bei einer Eingabe mehr	Einfacher Algorithmus für das Problem des Handlungsreisenden

2.1 Komplexitätsklassen

Mit Hilfe der Funktionsklassen $\mathcal{O}(f)$ kann man jetzt die Zeitkomplexität und die Speicherkomplexität sowohl von Algorithmen als auch Problemen angeben. Dabei ist, wie schon gesagt, die Komplexität eines Problems die Komplexität des bestmöglichen Algorithmus für dieses Problem.

Im folgenden konzentrieren wir uns auf die Komplexität der *Probleme*, nicht der Algorithmen. Man unterscheidet zunächst vier grobe Klassen:

DTime(f) =
Menge der Probleme, die sich *deterministisch* mit einer Zeitmessfunktion $t \in \mathcal{O}(f)$ lösen lassen.

NTime(f) =
Menge der Probleme, die sich *nichtdeterministisch* mit einer Zeitmessfunktion $t \in \mathcal{O}(f)$ lösen lassen.

DSpace(f) = Menge der Probleme, die sich *deterministisch* mit einer Speicherplatzmessfunktion $s \in \mathcal{O}(f)$ lösen lassen.

NSpace(f) = Menge der Probleme, die sich *nichtdeterministisch* mit einer Speicherplatzmessfunktion $s \in \mathcal{O}(f)$ lösen lassen.

Die nichtdeterministischen Versionen besagen, dass, wenn das Problem eine Lösung hat, und der Algorithmus die richtige Lösung *rät*, deren Überprüfung mit $\mathcal{O}(f)$ Aufwand machbar ist.

Wenn ein Verfahren $f(n)$ viel Speicherplatz benötigt, dann muss ja jede Speicherzelle davon mindestens einmal bearbeitet worden sein, was mindestens eine Zeiteinheit kostet. Daher braucht man immer

mehr Zeiteinheiten, als Platzeinheiten. Wenn man also ein Problem z.B. in linearer Zeit lösen kann, dann kann man in dieser Zeit auch höchstens linear viel Platz bearbeiten, nicht mehr. Das bedeutet $DTime(f) \subseteq DSpace(f)$ und $NTime(f) \subseteq NSpace(f)$

In der Komplexitätstheorie hat man für verschiedene Teilklassen davon spezielle Namen eingeführt, die in der Literatur oft benutzt werden. Einige davon sind:

$$\begin{array}{lcl} L & = & DSpace(\log(n)) \\ NL & = & NSpace(\log(n)) \\ P & = & \bigcup_{k\geq 1} DTime(n^k) \\ NP & = & \bigcup_{k\geq 1} NTime(n^k) \\ PSPACE & = & \bigcup_{k\geq 1} DSpace(n^k) = \bigcup_{k\geq 1} NSpace(n^k) \\ & & \text{(Satz von Savitch)} \end{array}$$

Die Klasse P besteht also aus allen Problemen, für die es einen deterministischen Algorithmus gibt, der das Problem in *polynomieller* Zeit löst, wobei der Grad des Polynoms egal ist.

Die Klasse NP besteht also aus allen Problemen, für die es einen nichtdeterministischen Algorithmus gibt, der einen Lösungskandidaten für das Problem in *polynomieller* Zeit überprüfen kann, wobei der Grad des Polynoms egal ist.

Jedes Problem, welches man in polynomieller Zeit *deterministisch* lösen kann, kann man natürlich auch in polynomieller Zeit *nichtdeterministisch* lösen. Daher gilt $P \subseteq NP$.

Die Klasse $PSPACE$ besteht aus den Problemen, für die es einen Algorithmus gibt, egal ob deterministisch oder nichtdeterministisch, der das Problem mit polynomiellem Platzbedarf löst.

Um ein Problem in eine dieser Klassen einsortieren zu können, muss man also den bestmöglichen Algorithmus dafür kennen. In einigen Fällen, z.B. für das Sortierproblem, kennt man in der Tat bestmögliche Algorithmen. In anderen Fällen kennt man zwar Algorithmen, aber ob darunter tatsächlich der bestmögliche ist, ist derzeit unbekannt. Dies trifft insbesondere auf die Klassen P und NP zu. Bisher gilt $P \subseteq NP$. Es könnte sich aber auch $P = NP$ herausstellen (was kaum jemand glaubt).

3 Das SAT-Problem

Der Startpunkt für die Problemklasse der NP-vollständigen Problem ist das *SAT-Problem*. SAT steht für *satisfiability* und bedeutet *aussagenlogische Erfüllbarkeit*. Ohne Rückgriff auf Aussagenlogik oder Boolesche Algebra kann man das SAT-Problem folgendermaßen verstehen:

- Man hat zunächst Variablen, die Werte 0 oder 1 annehmen können (alternativ falsch oder wahr).
- Diese Variablen kann man negieren. D.h. wenn p eine Variable ist, dann bedeutet $-p$ ihre Negation. Die Negation dreht die Werte der Variablen um. Aus 0 wird 1 und aus 1 wird 0. Variable zusammen mit ihrer Negation nennt man *Literale*. p und $-p$ sind beides Literale.

Wenn l ein Literal ist, dann kann man das *Komplement* l' von l bilden, indem man l folgendermaßen negiert: Falls $l = p$, dann ist $l' = -p$, und falls $l = -p$ dann ist $l' = p$.

- Eine *Liste* von Literalen nennt man *Klausel*. $(p, -q, r, s, -t)$ könnte eine solche Klausel sein. Beim SAT-Problem interpretiert man solche Listen von Literalen in sog. *Konjunktiver Normalform disjunktiv*. Das bedeutet, die Literale sind mit dem logischen *oder* verbunden. Wenn die Variablen einen konkreten Wert haben, dann hat die Klausel auch einen Wert. Dieser Wert ist 1, genau dann wenn *mindestens eines* der Literale den Wert 1 hat.

  Bsp.: Die Klausel (p) hat den Wert 1, falls p den Wert 1 hat.
  Die Klausel $(-p)$ hat den Wert 1, falls p den Wert 0 hat.
  Die Klausel (p, q) hat den Wert 1, falls p oder q oder beide den Wert 1 haben.
  Die leere Klausel $()$ hat per Definitionem den Wert 0.

  Für eine Klausel, die auf den Wert 1 abgebildet wird, sagt man auch, die *Klausel wird wahr*. Falls sie auf 0 abgebildet wird, sagt man auch, die *Klausel wird falsch*.

- Mehrere Klauseln bilden eine *Klauselmenge*. Im SAT-Problem sind alle Klauseln einer Klauselmenge mit dem logischen *und* verbunden. Damit kann man auch einer Klauselmenge einen Wert zuordnen, wenn die einzelne Klauseln einen Wert haben: Die Klauselmenge hat den Wert 1, genau dann wenn *alle* Klauseln den Wert 1 haben.

Das SAT-Problem besteht nun darin, für eine Klauselmenge C über den Variablen $p_1, \ldots, p_n$ herauszufinden, ob es eine *Belegung*, d.h. Werte für die Variablen gibt, so dass C den Wert 1 bekommt. Eine solche Belegung heißt auch *Modell* der Klauselmenge. Manche Klauselmengen haben ein Modell, die heißen *erfüllbar*, manche eben auch nicht, die heißen *unerfüllbar*.

Beispiele:
Die Klauselmenge $\{(p)\}$ hat offensichtlich ein Modell, nämlich $p = 1$.

Die Klauselmenge $\{(p), (-p)\}$ hat kein Modell. Für $p = 1$ wird die zweite Klausel $(-p)$ falsch, und für $p = 0$ wird die erste Klausel (p) falsch.

Um Klammern zu sparen, schreiben wir ab jetzt die Klauseln in eine Zeile, und Klauselmengen als Zeilen untereinander.

Die Klauselmenge
p, q
$p, -q$
$-p, q$
$-p, -q$
hat ebenfalls kein Modell. Alle vier Möglichkeiten 1. $p = 1, q = 1$, 2. $p = 1, q = 0$, 3. $p = 0, q = 1$ und 4. $p = 0, q = 0$ machen jeweils eine der Klauseln falsch.

Lässt man eine der Klauseln weg, dann bekommt man eine erfüllbare Klauselmenge.

Z.B. die Klauselmenge
p, q
$p, -q$
$-p, q$

hat das Modell $p = 1, q = 1$.

Bei diesen kleinen Beispielen sieht das noch ganz harmlos aus. Aber man sollte bedenken, für Klauselmengen über n Variablen, sind es 2^n mögliche Belegungen, von denen vielleicht nur eine einzige die richtige ist. Die muss man finden.

SAT-Probleme entstehen in vielen Anwendungsbereichen, insbesondere in der Verifikation von Chip-Entwürfen, d.h. dem Nachweis, dass ein Mikrochip-Entwurf auch das tut was er soll. Klauselmengen über *hunderten* von Variablen sind da nichts ungewöhnliches.

Ein NP-Algorithmus für das SAT-Problem ist ganz einfach.

1. Rate eine Variablenbelegung

2. Überprüfe, ob in jeder Klausel mindestens ein Literal ist, welches mit dieser Belegung zu 1 wird.

Die Überprüfung geht sogar in linearer Zeit. Daher ist das SAT-Problem auf jeden Fall in der Klasse NP. Es könnte aber auch in der kleineren Klasse P sein, wenn man einen polynomiellen Algorithmus dafür finden würde.

4 NP-Härte des SAT-Problems

NP-Härt des SAT-Problems bedeutet, dass man jedes andere Problem in der Klasse NP in ein SAT-Problem transformieren kann, dort lösen, und die Lösung zurücktransformieren kann auf eine Lösung des Ausgangsproblems. Die Transformation selbst darf natürlich nicht zu aufwendig sein. Sie muss in polynomieller Zeit machbar sein. NP-Härte des SAT-Problems zusammen mit dem Fakt, dass das SAT-Problem in NP liegt, bedeutet die *NP-Vollständigkeit* des SAT-Problems. (NP-Härte allein reicht nicht. Das SAT-Problem könnte ja in einer noch viel schlimmeren Klasse liegen.)

Diese Eigenschaft des SAT-Problems bedeutet, dass ein Lösungsverfahren für das SAT-Problem automatisch auch für alle anderen Probleme in der NP-Klasse benutzt werden kann.

Wie beweist man so etwas?

Die Schlüsselidee ist folgende: Wenn ein Problem in der Klasse NP ist, dann gibt es einen nichtdeterministisch polynomiellen Algorithmus dafür. Diesen kann man in einer nichtdeterministischen Turingmaschine implementieren. Jetzt muss man es schaffen, diese nichtdeterministischen Turingmaschine so in ein SAT-Problem zu transformieren, dass die Lösung des SAT-Problems die Information enthält, wie die Turingmaschine den Nichtdeterminismus steuern kann, so dass sie bei Alternativen immer die richtige Wahl trifft. Dann ist sie in polynomieller Zeit fertig.

Der Kern des Beweises ist also die Transformation einer nichtdeterministichen Turingmaschine in ein SAT-Problem.

4.1 Turingmaschine $\mapsto$ SAT-Problem

Die Beschreibung der Transformation einer beliebigen Turingmaschine in ein SAT-Problem ist sehr aufwendig und technisch. Wir illustrieren die Vorgehensweise daher an einer einfachen Maschine, der Einerkomplement-Maschine M. Diese Maschine geht einmal von links nach rechts durch eine Bitfolge, und dreht alle Bits um. Z.B. aus der Bitfolge 00011001 macht sie 11100110. Diese Maschine ist deterministische, was aber für die Transformation in ein SAT-Problem unerheblich ist.

Die Rechenregeln für diese Maschine sind

$$z, 0 \mapsto z, 1, R, \qquad z, 1 \mapsto z, 0, R \qquad z, \_ \mapsto e, \_, N$$

Wir nehmen eine einfache Eingabe: 01_, die die Maschine in 10_ umwandelt.

Zunächst ist es wichtig zu bestimmen, wieviele Schritte die Maschine macht. Für dieses Beispiel macht sie die Schritte 0,1,2,3. Im allgemeinen Fall eines NP-Problems, welches in der Turingmaschine implementiert wird, wissen wir, dass bei einer optimalen Steuerung des Nichtdeterminismus polynomiell viele Schritte notwendig sind. Für jede konkrete Eingabe kann man also die benötigte Schrittzahl konkret angeben: Zeitpunkt 0 bis Zeitpunkt t. Im Beispiel ist $t = 3$.

Außerdem wissen wir, wieviele und welche Zustände die Maschine braucht. Im Beispiel sind es die Zustände z und e.

Daraus kann man folgende SAT-Variablen für die Zustände erzeugen:

$$Z_{0z}, Z_{1z}, Z_{2z}, Z_{3z}, Z_{0e}, Z_{1e}, Z_{2e}, Z_{3e}.$$

Eine Variablenbelegung wie $Z_{2e} = 1$ soll bedeuten: die Maschine befindet sich bei Zeitpunkt 2 in Zustand e, oder allgemein $Z_{tx} = 1$ bedeutet: die Maschine befindet sich bei Zeitpunkt t in Zustand x.

Als nächstes bestimmen wir die SAT-Variablen für die Bandpositionen: P_{ti}.
$P_{ti} = 1$ bedeutet: (M befindet sich bei Zeitpunkt t auf Bandposition i)

Dafür brauchen wir im Beispiel: $P_{00}, P_{01}, P_{02}, P_{10}, P_{11}, P_{12}, P_{20}, P_{21}, P_{22}, P_{30}, P_{31}, P_{32}$

Die letzte Gruppe von SAT-Variablen beschreibt den Bandinhalt nach zu jedem Zeitpunkt:
$B_{tia} = 1$ bedeutet: der Bandinhalt zum Zeitpunkt t an Position i ist das Bit a.

Im Beispiel brauchen wir:
B_{000}, B_{001}, $B_{00\_}$, B_{010}, B_{011}, $B_{01\_}$, B_{020}, B_{021}, $B_{02\_}$,
B_{100}, B_{101}, $B_{10\_}$, B_{110}, B_{111}, $B_{11\_}$, B_{120}, B_{121}, $B_{12\_}$,
B_{200}, B_{201}, $B_{20\_}$, B_{210}, B_{211}, $B_{21\_}$, B_{220}, B_{221}, $B_{22\_}$,
B_{300}, B_{301}, $B_{30\_}$, B_{310}, B_{311}, $B_{31\_}$, B_{320}, B_{321}, $B_{32\_}$.

Mit diesen Variablen kann man den kompletten Zustand der Maschine in jedem einzelnen Zeitpunkt beschreiben, indem man für jede Variable eine Belegung angibt. Es sollte auch klar sein, welche Variablen man für die Kodierung eines beliebig anderen NP-Problems wählen muss.

Zur Formulierung der Arbeitsweise der Maschine benutzen wir bekannte logische Verknüpfungen, die sich aber alle auf direkte Weise in SAT-Klauseln umschreiben lassen.

$p \wedge q \Rightarrow r \wedge s$ wird zu Klausel $-p, -q, r$ und $-p, -q, s$.
Wenn also eine Belegung $p = 1, q = 1$ wählt, dann ist $-p = 0, -q = 0$, so dass die beiden Klauseln nur wahr werden können wenn $r = 1$ und $s = 1$ ist. Analoges gilt wenn, die Prämisse, d.h. der Teil links von $\Rightarrow$ weniger oder mehr Literale enthält.
$p \Leftrightarrow q$ wird zu den beiden Klauseln $-p, q$ und $-q, p$.
Eine Belegung, diese diese beiden Klauseln wahr macht muss also entweder $p = q = 1$ oder $p = q = 0$ sein.

Randbedingungen für die Maschine: Bestimmte Belegungen beschreiben unmögliche Zustände der Maschine. Z.B. kann sich die Maschine in einem Zeitpunkt nicht in zwei verschiedenen Zuständen befinden. D.h. eine Belegung wie $Z_{0z} = 1$ und $Z_{0e} = 1$ muss ausgeschlossen werden. Dies verhindert man durch entsprechende Klauseln.

- Die Maschine befindet sich zu jedem Zeitpunkt in genau einem Zustand:
  $Z_{0z} \Leftrightarrow -Z_{0e}$
  $Z_{1z} \Leftrightarrow -Z_{1e}$
  $Z_{2z} \Leftrightarrow -Z_{2e}$
  $Z_{3z} \Leftrightarrow -Z_{3e}$.
- Der Schreib-Lesekopf befindet sich zu jedem Zeitpunkt auf genau einer Bandposition:
  $P_{00} \Rightarrow -P_{01}, -P_{02}$ Wenn er sich zum Zeitpunkt 0 auf Position 0 befindet, kann er sich nicht auf Position 1 und nicht auf Position 2 befinden.
  $P_{10} \Rightarrow -P_{11} \wedge -P_{12}$
  $P_{20} \Rightarrow -P_{21} \wedge -P_{22}$
  $P_{30} \Rightarrow -P_{31} \wedge -P_{32}$
  Das führt man fort für die Positionen 1 und 2.
- Zu jedem Zeitpunkt kann sich auf jeder Bandposition nur genau ein Zeichen befinden:
  $B_{000} \Rightarrow -B_{001} \wedge -B_{00\_}$ Wenn er sich zum Zeitpunkt 0 auf Position 0 eine 0 befindet, kann sich zu diesem Zeitpunkt und an dieser Position keine 1 und kein Blank befinden.

  Das führt man fort für alle Zeitpunkte und Bandpositionen.

Diese Randbedingunen hängen nur von der Anzahl der Schritte, der Zustände, und der Bandlänge ab. Daraus kann man die Klauseln automatisch erzeugen.

Anfangsbedingungen: Die Maschine startet bei Zeitpunkt 0, im Zustand z, bei Position 0, und mit Bandinhalt $01\_$. Mit den SAT-Variablen formuliert man das als Klauseln:

Z_{0z}	Bei Zeitpunkt 0 ist die Maschine in Zustand z.
P_{00}	Bei Zeitpunkt 0 steht der Schreib-Lesekopf an Position 0.
B_{000}	Bei Zeitpunkt 0 steht auf Position 0 die 0.
B_{011}	Bei Zeitpunkt 0 steht auf Position 1 die 1.
$B_{02\_}$	Bei Zeitpunkt 0 steht auf Position 2 die das Blank.

Dass diese Klauseln nur aus einem Literal bestehen, zwingt jede erfüllende Belegung, diese Variablen mit 1 zu belegen, d.h. $Z_{0z} = 1$ muss sein, usw.

Auch diese Klauseln kann man für ein beliebiges NP-Problem aus einer gegebenen Anfangsbelegung des Bandes automatische berechnen.

Das Programm Schließlich müssen wir die Rechenregeln der Turingmaschine übersetzen:

$z, 0 \mapsto z, 1, R$**:** bedeutet ja: wenn die Machine im Zustand z die 0 liest, dann bleibt sie im Zustand z, ersetzt die 0 durch die 1, und geht eine Position nach rechts. Das gilt für jeden Zeitpunkt und für jede Bandposition.

Die erste daraus erzeugte Formel ist dann:
$Z_{0z} \wedge P_{00} \wedge B_{000} \Rightarrow Z_{1z} \wedge P_{11} \wedge B_{101}$
mit der Bedeutung: Wenn M in Zeitpunkt 0 im Zustand z ist ($Z_{0z} = 1$), und der Schreib-Lesekopf in Zeitpunkt 0 auf Position 0 ist ($P_{00} = 1$), und in Zeitpunkt 0 and Position 0 die 0 steht ($B_{000} = 1$), dann muss die Maschine im Zeitpunkt 1 immer noch in Zustand z sein ($Z_{1z} = 1$ wird erzwungen), und im Zeitpunkt 1 muss an Position 0 die 1 stehen ($B_{101} = 1$ wird erzwungen), und im Zeitpunkt 1 ist der Schreib-Lesekopf um 1 nach rechts gerückt, steht also auf Position 1 ($P_{11} = 1$ wird erzwungen).

Entsprechende Klauseln muss man für alle Zeitpunkte und Bandpositionen hinzufügen.

$z, 1 \mapsto z, 0, R$**:** wird entsprechend zu
$Z_{0z} \wedge P_{00} \wedge B_{001} \Rightarrow Z_{1z} \wedge P_{11} \wedge B_{100}$
und das für alle Zeitpunkt und Positionen.

$z, \_ \mapsto e, \_, N$**:** wird zu
$Z_{0z} \wedge P_{00} \wedge B_{00\_} \Rightarrow Z_{1e} \wedge P_{10} \wedge B_{10\_}$
und das auch für alle Zeitpunkt und Positionen.

Da für jeden Regeltyp festliegt, welche Klauseln ihn beschreiben, kann man jedes Maschinenprogramm der Turingmaschine automatisch in SAT-Klauseln übersetzen.

Frameaxiome: Schließlich muss man noch beschreiben, was sich bei den einzelnen Rechenschritten *nicht* ändert (das sind die berüchtigten Frameaxiome). Z.B. wenn sich bei Zeitpunkt 0 der Schreib-Lesekopf *nicht* auf Position 1 befindet, dann hat sich bei Zeitpunkt 1 der Bandinhalt auf Position 1 *nicht* geändert:
$-P_{01} \wedge B_{010} \Rightarrow B_{110}$
$-P_{01} \wedge B_{011} \Rightarrow B_{111}$
Das muss man für alle Zeitpunkte, Positionen und Bandinhalte generieren.

Alles zusammen ergibt das eine sehr große Klauselmenge. Wegen der Variablen B_{tij} kann man die Anzahl auf $\mathcal{O}(n^3)$ abschätzen, wobei n das Produkt aus Bandlänge, Schrittzahl und Anzahl der Zeichen ist. D.h. die Transformation Turingmaschine $\mapsto$ SAT-Problem ist *polynomiell*.

Im Originalbeweis wurde gezeigt, dass ein Modell der transformierten Turingmaschine, d.h. eine Belegung, die alle Klauseln wahr macht, die Information für die korrekte und erfolgreiche Steuerung für die Turingmaschine enthält.

Im Beispiel der Einerkomplementmaschine ergibt sich ein Modell wo die folgenden Variablen auf 1 abgebildet werden.

$Z_{0z}, P_{00}, B_{000}, B_{011}, B_{02\_}$ (Startzustand)
$Z_{1z}, P_{11}, B_{101}, B_{111}, B_{12\_}$ (1. Bit umgedreht)
$Z_{2z}, P_{22}, B_{201}, B_{210}, B_{22\_}$ (2. Bit umgedreht)
$Z_{3e}, P_{32}, B_{301}, B_{310}, B_{32\_}$ (Ende)

Daraus kann man genau die Abfolge von Schritten der Maschine ablesen:

Zeitpunkt	Zustand	Position	Bandinhalt
0	z	0	01_
1	z	1	11_
2	z	2	10_
3	e	3	10_

Insbesondere kann man den Bandinhalt im Endzustand auslesen. Man braucht also die Turingmaschine gar nicht mehr.

Diese Maschine ist deterministisch. Daher gibt es nur dieses eine Modell. Bei nichtdeterministischen Maschinen kann es eines oder mehrere Modelle geben. Jedes davon beschreibt eine erfolgreiche Abfolge von Schritten für die Steuerung der Maschine, und man kann den Bandinhalt im Endzustand direkt ablesen.

Zusammenfassung:
Man kann also *jedes* Problem in NP, für das es also einen nichtdeterministischen Algorithmus gibt, folgendermaßen lösen:
1. Implementiere den Algorithmus in einer Turingmaschine.
2. Schreibe die Eingabe auf das Band.
3. Transformiere die Turingmaschine mit der Eingabe in eine Klauselmenge.
4. Berechne für die Klauselmenge ein Modell.
5. Extrahiere aus dem Modell den Bandinhalt im Endzustand. Das ist das Ergebnis der Berechnung.

Die algorithmische Schwierigkeit steckt in Schritt 4, der Berechnung des SAT-Modells. Dafür hat man bisher leider nur exponentielle Algorithmen. Falls es irgendjemand schaffen könnte, dafür einen polynomiellen Algorithmus zu finden, könnte man mit dieser Technik jedes Problem in NP polynomiell lösen. Dann wäre $P = NP$ und derjenige, der das schafft, bekommt die 1 Million $. Auch für einen Beweis, dass das unmöglich ist, gibt es die 1 Million $.

Nebenbemerkung: Wenn wir die Transformation der Einerkomplementmaschine in ein SAT-Problem exakt Schritt für Schritt gemacht hätten (was man automatisieren kann), und dann nachgewiesen, dass das Modell auch wirklich stimmt, dann hätten wir *verifiziert*, dass die Maschine tatsächlich das Einerkomplement berechnet, *allerdings nur für die Bitfolge 01_*. Wollte man es *für alle* Bitfolgen verifizieren, bräuchten man ein logisches System mit einem *für alle* Quantor, also mindestens Prädikatenlogik erster Stufe.

5 Weitere NP-vollständige Probleme

Zu Beginn wurde erwähnt, dass das SAT-Modell die „Mutter“ aller NP-vollständigen Probleme ist. Das ist so zu verstehen, dass nur für dieses Problem der komplizierte Beweis über die Turingmaschine gemacht wurde. Die Beweise für alle anderen NP-vollständigen Probleme leitet man direkt oder indirekt aus dem SAT-Problem ab, und zwar folgendermaßen:

Gegeben ein Problem P_{Neu}. Um zunächst dessen NP-Härte zu zeigen:

1. Finde ein geeignetes NP-vollständiges Problem P_{Alt} (z.B. das SAT-Problem).
2. Finde eine Transformation $T(P_{Alt})$ des alten Problems in das neue Problem P_{Neu} mit folgenden Eigenschaften:
   - die Transformation ist polynomiell und
   - das alte Problem P_{Alt} hat eine Lösung genau dann wenn das transformierte Problem $T(Alt)$ eine Lösung hat. (Dazu zeigt man, wie man die Lösungen hin und her transformiert).

Wenn man das geschafft hat, ist die Argumentation immer die gleiche: Falls es einen polynomiellen Algorithmus A für das neue Problem gäbe, dann könnte man das alte Problem ebenfalls polynomiell lösen:

1. Man transformiert das alte Problem polynomiell in das neue Problem,
2. löst das neue Problem mit dem polynomiellen Algorithmus A und
3. transformiert die Lösung zurück.

Das ist zunächst kein Widerspruch zur NP-Vollständigkeit des alten Problems, da ja bisher nicht bewiesen wurde, dass NP-vollständige Probleme nicht polynomiell lösbar sind.

Es zeigt nur, dass ein polynomieller Algorithmus für das neue Problem auch für das alte Problem taugt. Wenn jemand also einen solchen Algorithmus finden würde, dann wäre $P = NP$, und derjenige bekäme auch die 1 Million $.

Um insgesamt die NP-Vollständigkeit des neuen Problems P_{Neu} zu zeigen, muss man neben der NP-Härte (wie oben) noch zeigen, dass das Problem in NP liegt. Dazu braucht man nur zu zeigen, dass eine geratene Lösung in polynomieller Zeit überprüft werden kann. Falls man das nicht zeigt, bedeutet das, dass das neue Problem u.U. in einer noch schlimmeren Komplexitätsklasse als NP liegt.

Zusammenfassend: die NP-Vollständigkeit eines Problems zeigt man
1. indem man zeigt, dass das Problen in NP liegt, und
2. indem man die NP-Härte zeigt, am besten nach der oben skizzierten Transformationstechnik.

Eine ganze Reihe von Problemen wurden bisher untersucht, und als NP-vollständig bewiesen.

5.1 Das 3-SAT-Problem

Dies ist eine einfachere Variante des SAT-Problems, wo jede Klausel nur drei Literale hat.

Das Problem liegt offensichtlich auch in NP, denn es kann mit dem gleichen nichtdeterministischen Rate-und-Teste Verfahren gelöst werden, wie das SAT-Problem selbst.

Um die NP-Härte zu zeigen, wählen wir als P_{Alt} das uns bisher einzig bekannte NP-vollständige Problem, nämlich das SAT-Problem selbst. Für die Transformation SAT $\mapsto$ 3-SAT müssen wir zeigen, wie eine beliebig lange Klausel in eine oder mehrere Klauseln mit 3 Literalen transformiert werden kann, sodass die Erfüllbarkeit erhalten bleibt.

Der Trick besteht in der Einführung von Hilfsvariablen.

Beispiel: für die Klausel p, q, r, s führen wir eine Hilfsvariable x ein und ersetzen die Klausel p, q, r, s durch die zwei 3-Literal Klauseln p, q, x und $-x, r, s$.

Jetzt argumentiert man folgendermaßen:
1. Angenommen, eine Belegung macht p, q, r, s wahr.
Fall 1: $p = 1$ oder $q = 1$ Dann wählen wir $x = 0$ und bekommen eine Belegung, die auch $-x, r, s$ wahr macht.
Fall 2: $p = 0$ und $q = 0$ dann muss aber $r = 1$ oder $s = 1$. In diesem Fall wählen wir $x = 1$ und machen damit beide neuen Klauseln wahr.

2. Angenommen, eine Belegung macht die beiden neuen Klauseln wahr.
Fall 1: $x = 1$ dann ist $-x = 0$ und daher muss entweder $r = 1$ oder $s = 1$ sein. Auf jeden Fall ist dann p, q, r, s wahr.
Fall 2: $x = 0$ dann muss entweder $p = 1$ oder $q = 1$ sein. Auf jeden Fall ist dann p, q, r, s ebenfalls wahr.

So funktioniert die Transformation für Klauseln mit 4 Literalen und der Beweis, dass die Erfüllbarkeit erhalten wird. Bei Klauseln mit $n > 4$ Literalen macht man die Transformation schrittweise. Mit einer ersten Hilsvariablen reduziert man die Klausel auf $n - 1$ Literale. Mit einer zweiten Hilfsvariablen auf $n - 2$ Literale usw.

Insgesamt braucht man so viele Hilfsvariablen, und damit neue Klauseln, wie das alte Problem zu viele Literale in den Klauseln hat. Die Transformation ist also polynomiell.

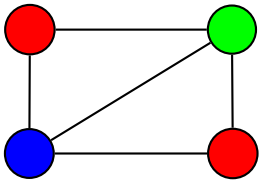
Fazit: das 3-SAT-Problem ist NP-vollständig

5.2 Graph-Coloring

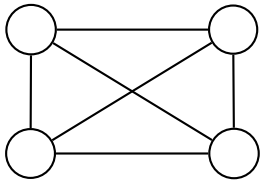
Das Problem ist folgendes: gegeben ist ein ungerichteter Graph, sowie drei Farben (allgemein n Farben). Zu entscheiden ist: kann man die Knoten des Graphen mit den drei Farben so einfärben, dass Nachbarknoten nicht die gleiche Farbe haben?

Beispiele:

dieser Graph ist färbbar



dieser nicht



Eine NP-Algorithmus für das Graph-Coloring Problem hat man schnell gefunden: Man rät eine Färbung und testet dann, ob alle Nachbarknoten verschiedene Farbe haben. Da es bei n Knoten maximal $\mathcal{O}(n^2)$ Nachbarpaare gibt, ist der Test polynomiell.

Also ist Graph-Coloring in NP.

NP-Härte: Zum Nachweis der NP-Härte müssen wir, nach dem obigen Rezept, ein bekanntes NP-vollständiges Problem finde, das wir in ein Graph-Coloring-Problem transformieren können. Wir probieren es mit dem 3-SAT-Problem, welches ja als NP-vollständig nachgewiesen wurde.

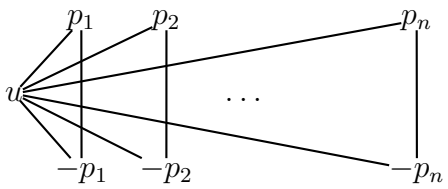
Ausgangspunkt ist demnach eine Klauselmenge C über den Variablen $p_1, \ldots, p_n$, deren Klauseln exakt 3 Literale haben (durch Verdopplung der Literale kann man das immer erreichen).

Wir zeigen, dass man C so in ein Graph-Coloring-Problem $T(C)$ transformieren kann, dass C erfüllbar ist genau dann wenn $T(C)$ eine Färbung mit 3 Farben hat.

Als Farben wählen wir Rot, Wahr und Falsch.

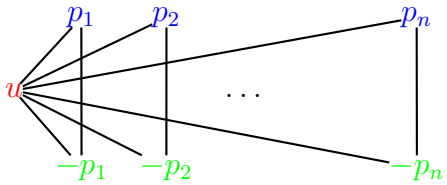
Im ersten Schritt erzeugt man einen Teilgraphen, dessen Färbung sicher stellt, dass für keine Variable p sowohl $T(p)$ als auch $T(-p)$ die gleiche Farbe bekommen.

Der Teilgraph sieht folgendermaßen aus:



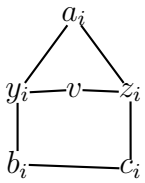
Dieser Teilgraph stellt sicher, dass die Nachbarknoten p_i und $-p_i$ nie die gleiche Farbe bekommen.

Eine mögliche Färbung wäre z.B.

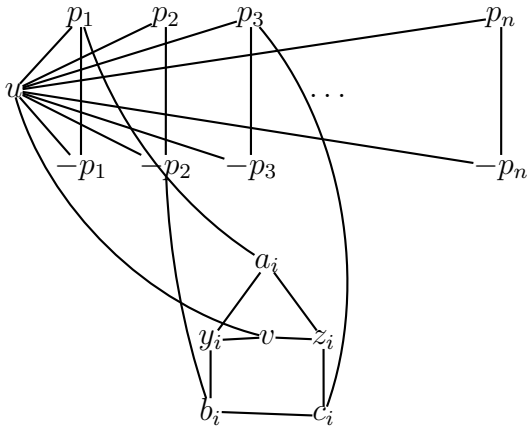


Alle Literale $p_1, \dots, p_n$ wären mit Wahr gefärbt. Alle Literale $-p_1, \dots, -p_n$ wären mit Falsch gefärbt.

Im nächsten Schritt müssen die Klauseln transformiert werden.
Dazu wird zunächst ein gemeinsamer Knoten v für alle Klauseln eingeführt.
Für jede Klausel k_i wird ein Teilgraph mit Knoten a_i, b_i, c_i, y_i, z_i erzeugt mit folgender Struktur:

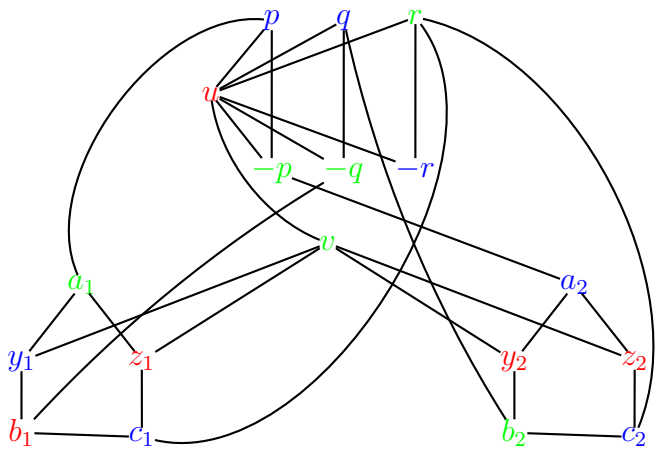


Der Knoten v wird mit dem Knoten u im obigen Teilgraphen verbunden. Die Knoten a_i, b_i, c_i im „Klauselgraphen“ werden mit den Knoten im obigen Teilgraphen verbunden, und zwar entsprechend der Literale in der Klausel k_i. Angenommen $k_i = p_1, -p_2, p_3$. Dann sie sieht die Verbindung so aus:



Diese Transformation ist sogar linear in der Anzahl Literale.

Betrachten wir eine ganz konkrete Klauselmenge wie $p, -q, r$ und $-p, q, r$ Sie hat u.A. das Modell $p = 1, q = 1, r = 0$. Dafür sieht der Graph mit entsprechender Färbung dann so aus:



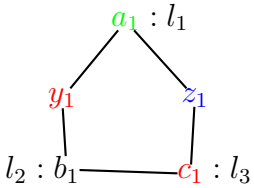
Der formale Beweis ist folgendermaßen:

$\Rightarrow$ Angenommen, die Klauselmenge hat ein Modell.

Dann kann man den Knoten u Rot färben, und die Literalknoten p_i und $-p_i$ entsprechend dem Modell. (Das färbt den oberen Teilgraphen). Den Knoten v kann man Falsch färben. Für die Knoten y_i und z_i im „Klauselgraphen“ bleiben dann die Farben Rot und Wahr.

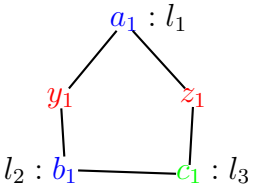
Betrachten wir eine Klausel $k_i = l_1, l_2, l_3$ (die l_i sind Literale, positive oder negative Variablen). Es ergeben sich drei Fälle:

$l_1 = 1$**:** Dann wählt man die Färbung des „Klauselgraphen“ folgendermaßen:

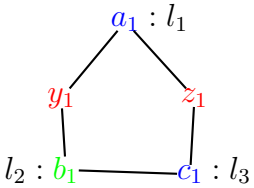


und die Farbe von b_1 wählt man Wahr oder Falsch je nach Wert von l_2.

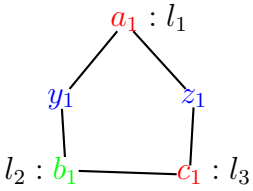
$l_1 = 0, l_2 \neq l_3$**:** Die Färbung ist dann



oder



$l_1 = 0, l_2 = l_3 = 1$**:** Die Färbung ist dann

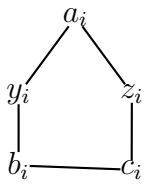


Den Fall $l_1 = 0, l_2 = l_3 = 0$ gibt es nicht, da sonst die Klausel falsch wäre.

$\Leftarrow$ Angenommen der Graph hat eine Färbung.

Da man die drei Farben beliebig rotieren kann, können wir *ohne Beschränkung der Allgemeinheit* annehmen, dass der Knoten u die Farbe Rot hat und v die Farbe Falsch. Dann haben die Literalknoten p_i und $-p_i$ entweder die Farbe Wahr oder Falsch, und nicht beides zugleich. Daraus ergibt sich direkt eine Belegung: falls p_i die Farbe Wahr hat, dann wird $p_i = 1$, ansonsten $p_i = 0$. Wir müssen zeigen, dass diese Belegung die Klauselmenge wahr macht.

Angenommen es wird eine Klausel $k_i = l_1, l_2, l_3$ falsch, d.h. $l_1 = l_2 = l_3 = 0$. Damit, und wegen v = Falsch hätte man dann für alle Knoten in dem „Klauselgraphen“ nur die Farben Rot und Wahr. Es gibt aber keine Möglichkeit, diesen Graphen mit nur zwei Farben einzufärben:



Daher kann k_i nicht falsch sein. Also wird die Klauselmenge wahr.

Man kann also jede 3-SAT-Klauslemenge in ein Graph-Coloring-Problem transformieren, die Färbung berechnen und daraus dann das Modell für die Klauselmenge ablesen.

Ergo: Das Graph-Coloring-Problem ist NP-hart, und zusammen mit Graph-Coloring $\in NP$ gilt:
Das Graph-Coloring-Problem ist NP-vollständig.

Man kann also das Graph-Coloring Problem selbst wieder als Ausgangspunkt für weitere Transformationen machen.

5.3 Das Rucksackproblem

Das Problem heißt Rucksackproblem, weil man damit berechnen kann, ob und wie man einen Rucksack mit verschiedenen Gegenständen füllen kann.

Gegeben sind natürliche Zahlen $a_1, \ldots, a_n$ (z.B. die Gewichte der Objekte für den Rucksack) sowie eine natürliche Zahl b (die Kapazität des Rucksacks).

Gefragt: Gibt es eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $\Sigma_{i \in I} a_i = b$?
oder alternativ formuliert: welche Objekte passen in den Rucksack, so dass seine Kapazität exakt ausgeschöpft wird?

Ein NP-Algorithmus ist wieder leicht gefunden: Man rät die Indexmenge I und überprüft. $\Sigma_{i \in I} a_i = b$. Das geht in linearer Zeit.

Also ist das Rucksackproblem in NP.

NP-Härte: Um die NP-Härte zu beweisen, transformieren wir wieder das 3-SAT-Problem, diesmal in ein Rucksack Problem.

Ausgangspunkt ist also eine 3-SAT-Klauselmenge mit m Klauseln und n Variablen $p_1, \ldots p_n$. Daraus müssen wir die Zahlen $a_1, \ldots, a_n$ und die Zahl b erzeugen.

Die Zahl b ist gegeben durch $b = \underbrace{444\ldots444}_{m}\underbrace{11\ldots11}_{n}$.

Bei 3 Klauseln mit 5 Variablen wäre also $b = 44411111$, einfach als natürliche Zahl gelesen.

Die Zahlen $a_1, \ldots, a_n$ sind alle von der Struktur: $x_1 \ldots x_m y_1 \ldots y_n$, wobei x_i Ziffern von $0 \ldots 3$ sind und die y_i entweder 0 oder 1. Wir illustrieren die Konstruktion mit der Klauselmenge C:
$p_1, -p_3, p_5$
$-p_1, p_4, p_5$
$-p_2, -p_2, -p_5$.

Diese Zahlen werden in 4 Gruppen erzeugt.

Gruppe 1 : die Zahlen $v_1, \ldots, v_n$ mit $v_i = x_1 \ldots x_m 0 \ldots 0 \underbrace{1}_{i} 0 \ldots 0$ stehen für die positiven Vorkommen der Variablen in den Klauseln. Und zwar ist x_k = Anzahl der positiven Vorkommnisse der Variablen p_i in Klausel k.

Für die Klauselmenge C wären das

$$v_1 = 100\ 10000$$
$$v_2 = 000\ 01000$$
$$v_3 = 000\ 00100$$
$$v_4 = 010\ 00010$$
$$v_5 = 110\ 00001$$

Gruppe 2 : die Zahlen $v'_1, \ldots, v'_n$ werden genauso erzeugt, nur zählen sie die negativen Vorkommnisse der Variablen.

Für die Klauselmenge C wären das

$$v'_1 = 010\ 10000$$
$$v'_2 = 002\ 01000$$
$$v'_3 = 100\ 00100$$
$$v'_4 = 000\ 00010$$
$$v'_5 = 001\ 00001$$

Die hintere Gruppe von Ziffern identifiziert eindeutig die Variable. Die Ziffern in der ersten Gruppe zählen die Vorkommnisse in den jeweiligen Klauseln. Da die Klauseln 3 Literale haben, können diese Ziffern nur 0,1,2 oder 3 sein.

Gruppe 3 : Jetzt brauchen wir für die Addition auf die Ziffern 4 in der Zahl b noch ein paar Resevezahlen. Die erste Gruppe von Reservezahlen braucht man wenn zur Addition auf die 4 noch eine 1 fehlt. Für die Klauselmenge C wären das:

$$c_1 = 100\ 00000$$
$$c_2 = 010\ 00000$$
$$c_3 = 001\ 00000$$

Gruppe 4 : Die zweite Gruppe von Reservezahlen braucht man wenn zur Addition auf die 4 noch eine 2 fehlt. Für die Klauselmenge C wären das:

$$d_1 = 200\ 00000$$
$$d_2 = 020\ 00000$$
$$d_3 = 002\ 00000$$

Die Transformation erzeugt $2(m+n)$ viele Zahlen, ist also polynomiell.

$\Rightarrow$ Angenommen die Klauselmenge D hat ein Modell M. Wir müssen zeigen, dass es dann eine Lösung des generierten Rucksackproblems gibt.

Die Indexmenge I kann jetzt folgendermaßen ausgewählt werden: Falls $p_i = 1$ dann ist $v_i \in I$, ansonsten $v'_i \in I$.

Für die Beispielklauseln C wäre ein Modell: $p_1 = 1, p_2 = 1, p_3 = 1, p_4 = 1, p_5 = 0$. In I wäre dann:

$$v_1 = 100\ 10000$$
$$v_2 = 000\ 01000$$
$$v_3 = 000\ 00100$$
$$v_4 = 010\ 00010$$
$$v'_5 = 001\ 00001$$

Die Summe ist: 111 11111. Um exakt auf $b = 444\ 11111$ aufzusummieren braucht man noch aus den Gruppen 3 und 4: $c_1 + c_2 + c_3 + d_1 + d_2 + d_3 = 333\ 00000$. Das ist die Lösung des Rucksackproblems.

Für eine beliebige Klauselmenge sorgt die Bedingung, dass in jedem Modell für jede Variable p entweder $p = 0$ oder $p = 1$ ist, dafür das für jedes i entweder v_i oder v'_i gewählt wird, aber nicht beide. Die hinteren Ziffern addieren sich daher immer zur 1.

In den vorderen m Ziffern steht jede Ziffer an Position j für die j-te Klausel, und die Ziffer gibt an wie oft die Variable darin vorkommt. Die größte Zahl an der Position j kann nur 3 sein. In allen anderen Zahlen steht dann an der Position j die 0. Summiert man alle Zahlen auf, dann ist die größte Zahl, die in den erste m Ziffern der Summe vorkommen kann, die 3. Mit entsprechenden Zahlen aus Gruppe 3 und 4 kann man daher immer auf die Zahl b kommen.

$\Leftarrow$ Angenommen das Rucksackproblem hat die Lösung I.
Zunächst kann man feststellen, dass die Ziffern in den Zahlen so klein sind, dass bei der Summierung keine Überträge vorkommen.

Die Zahlen in I summieren zu $b = \underbrace{444\ldots444}_{m}\underbrace{11\ldots11}_{n}$. Die 1en im hinteren Zifferblock können nur von Zahlen aus den Gruppen 1 und 2 kommen, und zwar entweder ein v_i oder ein v'_i, aber nicht beide. Daraus lesen wir eine Belegung ab: Falls $v_i \in I$, dann ist $p_i = 1$, und falls $v'_i \in I$ dann ist $p_i = 0$.

Angenommen diese Belegung macht die Klausel $C_k = l_1, l_2, l_3$ falsch. Falls $l_i = p_j = 0$, dann ist $v'_j \in I$, und v'_j hat an der Stelle k die Ziffer 0 (da ja $-p_j$ nicht in C_k vorkommt).

Falls $l_i = -p_j = 0$, dann ist $v_j \in I$, und v_j hat an der Stelle k die Ziffer 0 (da ja p_j nicht in C_k vorkommt.).

Beim Aufsummieren entsteht also an der Stelle j eine 0. Durch Hinzunehmen von Zahlen aus der Gruppe 3 und 4 kann man aber höchstens an dieser Stelle eine 3 bekommen, keine 4.

Also ist die Annahme, dass die Belegung die Klausel falsch macht selbst falsch.

Also ist die Belegung ein Modell für die ganze Klauselmenge.

Man kann also eine 3-SAT-Klauselmenge in ein Rucksackproblem transformieren, und aus dessen Lösung ein Modell für die Klauselmenge ablesen.

Ergo: Das Rucksackproblem ist NP-hart, und zusammen mit Rucksackproblem $\in NP$ gilt:
Das Rucksackproblem ist NP-vollständig.

Damit kann man das Rucksackproblem als Ausgangspunkt für weitere Transformationen nehmen, was wir im folgenden Abschnitt tun.

5.4 Das Paritionsproblem

Dieses Problem ist ähnlich zum Rucksackproblem. Es geht darum, herauszufinden, ob man eine Menge von Objekten exakt auf zwei Behälter aufteilen kann.

Formal:
Gegeben: Natürliche Zahlen $a_1, \ldots a_n$.
Gefragt: Gibt es eine Teilmenge $J \subseteq \{1, \ldots, n\}$ mit $\Sigma_{i \in J} a_i = \Sigma_{i \notin J} a_i$?

Auch hier ist der NP-Algorithmus offensichtlich: Rate die Indexmenge J und überprüfe die Summen. Das geht in linearer Zeit.

Also ist das Paritionsproblem in NP

NP-Härte:
Um die NP-Härte zu zeigen transformieren wir ein Rucksackproblem in ein Partitionsproblem.

Sei also $(a_1, \ldots, a_k, b)$ ein Rucksackproblem mit $M = \Sigma_{i=1}^{k} a_k$. Wir transformieren das Rucksackproblem in ein Partitionsproblem:

$$(a_1, \ldots, a_k, b) \mapsto (a_1, \ldots, a_k, M - b + 1, b + 1)$$

$\Rightarrow$ Sei I eine Lösung des Rucksackproblems.
Also ist $\Sigma_{i \in I} = b$ und $\Sigma_{i \notin I} = M - b$.

Wir haben also $\Sigma_{i \in I} + M - b + 1 = M + 1$ und $\Sigma_{i \notin I} + b + 1 = M + 1$. Das bedeutet, $I \cup \{k+1\}$ ist eine Lösung die Partitionsproblems.

$\Leftarrow$ Sei J eine Lösung des Partitionsproblems.
Entweder $M - b + 1$ oder $b + 1$ müssen in J liegen. Wenn beide drin sind wird die eine Summe zu groß, wenn beide nicht drin sind wird die andere Summe zu groß. Angenommen, $M - b + 1$

liegt in J, und J' sei J ohne diese Zahl. Wir haben also $\Sigma_{i\in J'}a_i + M - b + 1 = \Sigma_{i\notin J'}a_i + b + 1$
$\Rightarrow \Sigma_{i\in J'}a_i + M = \Sigma_{i\notin J'}a_i + 2b$
Wegen $M = \Sigma_{i\in J'}a_i + \Sigma_{i\notin J'}a_i$
$\Rightarrow 2\Sigma_{i\in J'}a_i + \Sigma_{i\notin J'} = \Sigma_{i\notin J'}a_i + 2b$
$\Rightarrow 2\Sigma_{i\in J'}a_i = 2b$
$\Rightarrow \Sigma_{i\in J'}a_i = b.$
J' ist also eine Lösung des Rucksackproblems.

Falls $b + 1$ in J liegt, geht der Beweis analog.

Man kann also eine Rucksackproblem in ein Partitionsproblem transformieren, und aus dessen Lösung gibt eine Lösung des Rucksackproblems.

Ergo: Das Paritionsproblem ist NP-hart, und zusammen mit Paritionsproblem $\in NP$ gilt:
Das Paritionsproblem ist NP-vollständig.

Damit kann man das Paritionsproblem als Ausgangspunkt für weitere Transformationen nehmen, was wir im folgenden Abschnitt tun.

5.5 Das Bin Packing-Problem

Das Bin-Packing-Problem erweitert das Partitionsproblem von 2 Behälter auf k Behälter. Außerdem müssen die k Behälter nicht ganz voll werden.

Gegeben: k Behälter derselben Größe $b \in \mathbb{N}$. „Objekte“ $a_1, \ldots, a_n \leq b$.

Gefragt: Können die Objekte so auf die Behälter verteilt werden, dass die sie nicht überlaufen?
D.h. gibt es eine Paritionierung $I_1, \ldots I_k \subseteq \{1, \ldots, n\}$ mit $\Sigma_{j\in I_i}\, a_j \leq b$ für alle i?

Ein NP-Algorithmus ist wiederum: rate die Partitionierung, und teste die Summen. Das geht in linearer Zeit.

Also ist das Bin Packing-Problem in NP

NP-Härte:
Um die NP-Härte zu zeigen transformieren wir ein Partitionsproblem in ein Bin Packing-Problem.

Sei also $(a_1, \ldots, a_k)$ ein Partitionsproblem. Zunächst einmal hat dieses Problem überhaupt nur eine Lösung wenn $M = \Sigma_{i=1}11k\, a_i$ gerade ist. Sonst bekommt man keine zwei gleich großen Teile. Dieses Problem wird auf ein Bin Packing-Problem mit zwei Behältern transformiert.

$$(a_1, \ldots, a_k) \mapsto \begin{cases} \text{Behältergröße} & b = M/2 \\ \text{Behälterzahl} & k = 2 \\ \text{Objekte} & a_1, \ldots a_k \end{cases}$$

$\Rightarrow$ Wenn das Partitionsproblem eine Lösung hat, dann kann man die Objekte auf die beiden Behälter verteilen, und jeder wird exakt bis zur Kapazitätsgrenze $M/2$ gefüllt. Also hat das Bin Packing-Problem eine Lösung.

$\Leftarrow$ Wenn das Bin Packing-Problem eine Lösung hat, dann muss diese Lösung so sein, dass beide Behälter exakt bis zur Kapazitätsgrenze $M/2$ gefüllt sind. Wenn einer weniger gefüllt wäre, dann wäre die Gesamtsumme kleiner als M. D.h. einige Objekte wären übrig. Das wäre keine Lösung. Also gibt die Aufteilung auf die Behälter eine Lösung des Partitionsproblems.

Ergo: Das Bin Packing-Problem ist NP-hart, und zusammen mit Bin Packing-Problem $\in NP$ gilt: **Das Bin Packing-Problem ist NP-vollständig**.

5.6 Noch weitere NP-vollständige Probleme

In der Literatur findet man eine stetig wachsende Menge an NP-vollständigen Problemen. Viele davon sind aus der Graphentheorie.

Clique: Gibt es in einem ungerichteten Graphen eine Teilmenge von *mindestens* k Knoten, die alle miteinander verbunden sind?

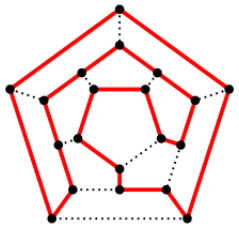
Wenn die Knoten des Graphen Menschen sind, und die Kanten sind die *man-kennt-sich* Beziehung, dann ist eine Clique eine Teilmenge dieser Menschen, wo jeder jeden kennt (daher wohl auch der Name Clique).

Knotenüberdeckung: Gibt es in einem ungerichteten Graphen eine Teilmenge von Knoten mit höchstens k Elementen, deren Knoten insgesamt mit allen Kanten verbunden sind?

Wenn wiederum die Knoten des Graphen Menschen sind, und die Kanten sind die *man-kennt-sich* Beziehung, dann wäre eine Knotenüberdeckung eine Teilmenge K dieser Menschen, so dass jeder Mensch in der Menge von einem Menschen in K gekannt wird.

Ungericheter Hamilton-Kreis:

Gibt es in einem ungerichteten Graphen einen *Hamiltonkreis*, das ist dabei eine Art Kreis, der alle Knoten des Graphen enthält (und keinen Knoten zweimal durchläuft oder überschreitet)?



Gericheter Hamilton-Kreis: Gibt es in einem gerichteten Graphen einen *Hamiltonkreis*?

Traveling Salesman: Gegeben n Städte (mit Entfernungen dazwischen) und eine Länge k. Gibt es eine Rundreise durch alle Städte der maximalen Länge k?

In den nächsten beiden Kapiteln werden die gängigen Ansätze für die Lösung NP-vollständiger Probleme vorgestellt. Sie werden jeweils am SAT-Problem, am Graph-Coloring-Problem (kurz GC-Problem) und am Bin Packing-Problem konkretisiert.

6 Random Walk Ansätze für NP-vollständige Probleme

Die Idee der Random Walk Ansätzen nimmt die Nichtdeterministisch-Polynomielle (NP) Definition nahezu wörtlich: Man startet mit einem Lösungskandidaten, der mehr oder wenig zufällig ist und testet, ob der Kandidat tatsächlich eine Lösung ist. Falls nicht, verändert man den Kandidaten punktuell und testet wieder. Das wiederholt man solange, bis man eine Lösung gefunden hat.

Falls das Problem eine Lösung hat, hofft man, dass der Algorithmus irgendwann auf die Lösung stößt. Falls das Problem aber keine Lösung hat, terminiert der Algorithmus nicht.

Diese Vorgehensweise kann natürlich nur dann erfolgreich sein, wenn man die punktuelle Veränderung steuern kann, so dass man der Lösung immer näher kommt.

Dazu benötigt man ein *Gütemaß*, welches dem Algorithmus sagt, wie nah er schon an der Lösung ist. Das Gütemaß hängt natürlich vom Problem ab.

Ein Vorschlag für das Gütemaß ist:

Sat-Problem: die Anzahl der Klauseln, die noch falsch sind (alle Literale sind 0).

GC-Problem: die Anzahl der Kanten mit Farbkonflikten.

Bin Packing: die Summe oder die Gesamtgröße der noch nicht einsortierten Objekte. Wenn man die Behälter möglichst gleichmäßig füllen möchte, könnte man als Gütemaß z.B. das Tupel (die gemittelten freien Kapazitäten, die Größe der noch nicht einsortierten Objekte) nehmen.

Für die Gütemaße muss man eine Festlegung treffen, wann es eine Lösung repräsentiert. Z.B. wenn es 0 ist, oder wenn eine Komponente eines Tupels 0 ist, ist eine Lösung gefunden.

Die Startkonfiguration: Ein wichtiger Punkt ist auch, dass man die Lösungskandidaten zu Beginn nicht wirklich zufällig wählt, sondern so, dass man schon möglichst nahe an die Lösung kommt.

Folgendes sollte helfen:

Sat-Problem: Bei der Entscheidung, ob man für eine Variable mit $p = 1$ oder $p = 0$ startet, wählt man die Variante, die in mehr Klauseln vorkommt. Wenn also p in 5 Klauseln vorkommt und $\neg p$ in 7 Klauseln, wählt man $p = 0$. Damit macht man zu Beginn schon möglichst viele Klauseln wahr.

GC-Problem: Bei der Entscheidung, ob man einen Knoten rot, grün oder blau färbt, nimmt man die Farbe, die mit den schon eingefärbten Nachbarknoten die wenigsten Konflikte erzeugt.

Bin Packing: Hier könnte man die Objekte der Größe nach in die einzelnen Behälter nacheinander verteilen, die großen Objekte zuerst.

Operationen: Nach dem Start hat man also einen Lösungskandidaten, und kann das Gütemaß berechnen. Für die punktuellen Veränderungen benötigt man lokale Operationen, die die Lösungskandidaten manipulieren:

Sat-Problem: Ändere die Belegung einer Variablen (Flip)

GC-Problem: Ändere die Farbe eines Knotens

Bin Packing: Hier gibt es verschiedene Operationen, z.B.:
- Vertausche zwei Objekte zwischen zwei Behältern
- Vertausche ein Objekte aus einem Behältern mit einem noch nicht einsortierten Objekt.

Für die Auswahl der zielführendsten Operation berechnet man die Änderung des Gütemaßes. Die Operation, die das Gütemaß am nähesten zur Lösung bring, wird ausgewählt. Dies ist ein Suchproblem, welches man aber durch geeignete Datenstrukturen und Indexing extrem beschleunigen kann. Hierbei zeigt sich das Können des Programmierers!

Leider kann die Suche in einen Zustand geraten, wo noch keine Lösung gefunden ist, aber auch durch eine punktuelle Operation keine Verbesserung des Gütemaßes möglich ist. Dies ist das bekannte und gefürchtete Phänomen der *lokalen Minima*. Betrachtet man die Menge aller möglichen lokalen Operationen, und das sich daraus ergebende Gütemaß, dann ist das eine Art Gebirge mit Bergen und Tälern. Man sucht das Tal auf Meereshöhe (Gütemaß = 0). Es kann aber viele höhere Täler geben, die in alle Richtungen durch Berge eingeschlossen sind. Über diese Berge muss man weg kommen. Hierfür gibt es verschiedene Strategien:

Zufallsschritte: In regelmäßigen Abständen, oder auch nur dann, wenn man in einem lokalen Minimum ist, macht man zufällige Änderungen am Lösungskandidaten, die nicht durch das Gütemaß motiviert sind. Dabei hofft man, über die Hindernisse zu springen und in ein tieferes Tal zu kommen. Das verhindert auch, dass das Verfahren in eine Schleife läuft.

Simulated Annealing: Hierbei erlaubt man, dass bei jeder lokalen Änderung das Gütemaß auch in gewissem Rahmen schlechter werden darf. Die Größe der erlaubten Verschlechterung ist zu Beginn recht hoch, und wird dann nach und nach verringert, so dass man sich irgendwann nicht mehr verschlechtern darf. Dabei hofft man, dass man zu Beginn über die ganz hohen Berge springen kann. Wenn man in die Nähe der Lösung kommt, sollte man aber gezielter in die Richtung der Lösung gehen.

Die konkrete Ausgestaltung dieser Algorithmen ist kaum anders als durch Experimente zu bestimmen, und hängt oft auch von den konkreten Problemklassen der Anwendung ab.

Generell kann man aber feststellen, dass Random Walk schneller zu einer Lösung kommt, wenn es mehr Lösungen gibt (bei SAT mehr erfüllende Belegungen, bei GC mehr Färbungen, bei Bin Packing mehr Verteilungen).

Es gibt sehr gute Random Walk Implementierungen für das SAT-Problem, welche Probleme mit Millionen Klauseln lösen können.

7 Vollständige Verfahren für NP-vollständige Probleme

Vollständige Verfahren sind solche, die garantiert immer terminieren und das Ergebnis melden, auch wenn das Problem keine Lösung hat. Dafür muss das Verfahren garantieren, dass alle Lösungskandidaten *systematisch* ausprobiert werden.

Für NP-Probleme gibt es i.A. eine Reihe von Variablen, für die Werte aus einem endlichen Wertebereich ausgewählt müssen. Bei SAT-Problemen sind das die booleschen Variablen, für die 0 oder 1 gewählt werden muss. Beim Graph Coloring-Problem sind es die Knoten, für die eine von den n Farben gewählt werden muss. Beim Bin Packing-Problem sind es die Objekte, für die einer der Behälter gewählt werden muss.

Jede Wahl für eine Variable hat i.A. Konsequenzen für andere Variablen, deren Wahlmöglichkeiten eingeschränkt werden. Dies kann man sich zu Nutze machen, indem man diese Konsequenzen berechnet, um von vorneherein nutzlose Wahlmöglichkeiten auszuschließen. Man nennt das *Constraint Propagation*.

Man baut dazu die Lösungskandidaten systematisch auf, indem man für jede einzelne Variable eine Entscheidung trifft, die man u.U. aber später wieder ändern muss.

Folgende Schritte werden dabei ausgeführt:

1. Wähle eine Alternative für eine minimale Erweiterung des Lösungskandidaten
2. Berechne die Konsequenzen dieser Erweiterung (Constraint Propagation).
3. Falls der Lösungskandidat widersprüchlich wird, mache die letzte Erweiterung rückgängig (Backtracking) und wähle eine andere Alternative.

Wichtig ist dabei, welche Erweiterung man wählt. Das kann entscheidend für die Performanz des Verfahrens sein.

Wir illustrieren das wieder an den drei NP-vollständigen Problemen.

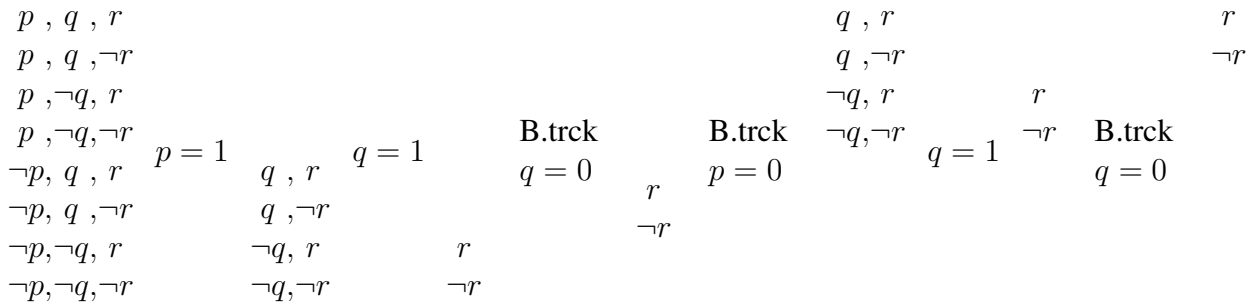
7.1 SAT-Solving

Hierbei muss für jede Variable der Wert 0 oder 1 festgelegt werden. Bei dem Verfahren, welches auf Davis und Patnam zurückgeht, startet man mit einer Variablen p, und wählt z.B. als erstes $p = 1$. Die unmittelbaren Konsequenzen dieser Entscheidung sind, dass alle Klauseln, in denen p positiv vorkommt auf 1 abgebildet werden, und daher nicht weiter betrachtet werden müssen. Aus allen Klauseln, in denen $\neg p$ vorkommt, kann $\neg p$ gestrichen werden, da $\neg p = 0$ ist, und um die Klausel auf 1 abzubilden, muss eines der anderen Literale 1 werden. Bei der Wahl $p = 0$ ist es genau umgekehrt. Falls jetzt schon eine Klausel leer geworden ist, oder es zwei Klauseln mit einem Literal gibt, die widersprüchlich sind, ist das ein Widerspruch. Dann war die Wahl $p = 1$ falsch, und man kann schließen, dass, wenn es überhaupt eine Lösung gibt, $p = 0$ sein muss. Falls keine Klausel leer

geworden ist, wählt man eine weitere Variable q und erweitert die Lösung mit $q = 1$. Jetzt wiederholt sich die Prozedur.

Das Verfahren terminiert mit einer erfüllenden Belegung, wenn alle Klauseln auf 1 abgebildet wurden, d.h. die weiter zu betrachtende Klauselmenge ist leer geworden. Es terminiert mit „unerfüllbar“ wenn alle Alternativen zu Widersprüchen geführt haben.

Wir illustrieren das an folgendem Beispiel:



Jetzt sind alle Kombinationen ausprobiert worden. Jede hat auf einen Widerspruch geführt. Daher ist die Klauselmenge nicht erfüllbar.

Constraint Propagation: Neben den einfachen Löschungen als Konsequenzen der Wahl für die Variablen gibt es noch eine ganze Reihe weiterer Operationen, die die Klauselmenge vereinfachen.

Beispiele sind:

Unit Propagation: Wenn eine Klausel auf ein Literal geschrumpft ist (Unit Klausel), dann kann man das Komplement des Literals aus allen Klauseln löschen.

$$\frac{p \qquad \neg p, Rest}{Rest}$$

Das kann einen Schneeballeffekt von weiteren Löschungen erzeugen.

Subsumption: Wenn eine Klausel eine Teilmenge einer anderen Klausel ist, kann man die andere Klausel löschen.

Resolution: Wenn es eine Klausel p, X (X beliebig) gibt, und eine zweite Klausel $\neg p, X, Y$, dann kann man das $\neg p$ aus der zweiten Klausel löschen.

$$\frac{p, X \qquad \neg p, X, Y}{X, Y}$$

Natürlich können die Vorzeichen auch umgekehrt sein.

Bei all diesen Operationen ist zu beachten, dass die Klauseln ungeordnet sind. Daher können die beteiligten Literale irgendwo in der Klausel stecken.

Es gibt noch weitere Vereinfachungsoperationen, die man in das Constraint Propagation einbauen kann. Im konkreten Fall muss man jedoch testen, wie teuer diese zu implementieren sind. Brauchen sie zu viel Aufwand, dann lohnt es sich nicht.

Auswahlstrategie: Ganz entscheidend für die Performanz des Verfahrens ist die Wahl der nächsten Variablen, für die der Wert festgelegt werden soll. Man kann z.B. die Variable wählen, die am wenigsten häufig vorkommt. Dann schränkt man die Möglichkeiten für die anderen Variablen am wenigsten ein. Oder man kann die Variable wählen, die am häufigsten vorkommt. Dann findet man die Widersprüche schneller. Man kann auch versuchen, eine Variable zu finden, bei der das Constraint Propagation am stärksten wirkt (ist aber aufwendig).

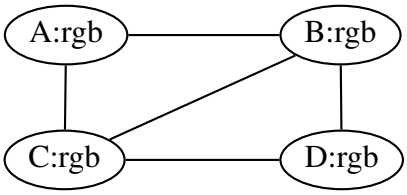
Seit vielen Jahren gibt es die *SAT-Competition*, bei der die besten SAT-Solver gegeneinander antreten.

7.2 Graph Coloring

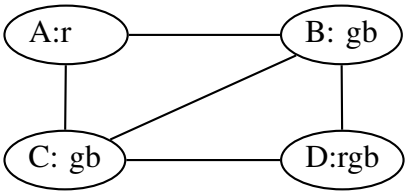
Hierbei muss man für jeden Knoten eine der drei (allgemein n) Farben festlegen. Man fängt mit einem Knoten an, wählt eine Farbe und berechnet die Konsequenzen (Constraint Propagation). Dann wählt man den nächsten Knoten usw. Falls zwei benachbarte Knoten die gleiche Farbe bekommen, muss man wieder backtracken und die nächste Farbe probieren.

Für das Constraint Propgation ist es nützlich, wenn man sich an jedem Knoten die noch wählbaren Farben merkt.

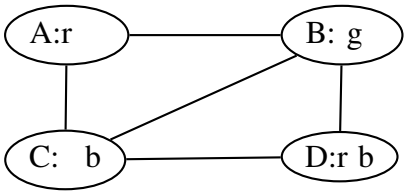
Wir illustrieren das an einem Beispiel. Bei den Knoten notieren wir nicht nur den Knotennamen, sondern auch die noch erlaubten Farben. Zu Beginn sind alle Farben erlaubt.



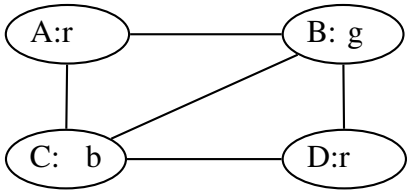
Wir starten die Suche, indem wir für den Knoten A die Farbe r wählen, und dann für die Nachbarknoten diese Farbe als Wahlmöglichkeit löschen (Constraint Propagation)



Für den Knoten B könne wir jetzt die Farbe g wählen, und g aus den Nachbarknoten streichen.



Bei Knoten C ist nur noch die Farbe b übrig, was die Wahlmöglichkeit für den Nachbarknoten D reduziert. Die Constraint Propagation geht also weiter.



Das ist auch jetzt schon die Lösung. In diesem Fall war kein Backtracking nötig, was aber die große Ausnahme ist.

In dem Beispiel war die Auswahlstrategie für den nächsten Knoten und die nächste Farbe einfach nach der vorgegebenen Reihenfolge. Für komplexe Graphen sollte man das aber heuristisch steuern. Man könnte als nächsten Knoten einen mit möglichst wenig / möglichst vielen Nachbarn wählen. Nur Experimente können hier bestimmen, was am günstigsten ist.

7.3 Bin Packing

Beim Bin Packing-Problem sind die Objekte die Variablen, und die Behälter die möglichen Werte. Für jede Variable merkt man sich daher die Liste der noch erlaubten Behälter. Man startet also, indem man ein Objekt in einen Behälter legt. Dies kann bewirken, dass der restliche Platz für andere Objekte nicht ausreicht. Daher kann man den Behälter aus deren Liste streichen (Constraint Propagation). Ist für eine Variable die Liste leer geworden, kann dieses Objekt nicht mehr untergebracht werden, was Backtracking nötig macht.

Das Vorgehen ist dann ganz analog zum Vorgehen beim Graph Coloring Problem. Nur die Details des Constraint Propagation sind anders.