Seminar Ergebnisprüfung zur Laufzeit

Lehr- und Forschungsgebiet Theoretische Informatik

IP=PSPACE

Joachim Kneis

joachim.kneis@post.rwth-aachen.de

29. Januar 2004

Zusammenfassung

Zuerst nutzen wir probabilistische Turingmaschinen sowie die Klasse AM als alternative Beschreibung von IP um IP $\subseteq$ PSPACE zu zeigen. Danach beweisen wir, daß QBF PSPACE-Vollständig ist und führen das LFKN Protokoll am Beispiel der Permanente ein. Im Anschluss daran wenden wir eine Arithmetisierung für Formeln an, um ein modifiziertes LFKN Protokoll für QBF zu entwerfen und zeigen so PSPACE $\subseteq$ IP. Abschließend werden die Folgerungen aus IP=PSPACE für das Gebiet der Programmverifikation diskutiert.

Inhaltsverzeichnis

1 Einführung **3**
1.1 Motivation 3
1.2 Grundlegende Definitionen 3

2 IP $\subseteq$ PSPACE **4**
2.1 PPSPACE 4
2.1.1 probabilistische Turingmaschinen 5
2.2 PPSPACE = PSPACE 6
2.2.1 PPSPACE $\subseteq$ PSPACE 6
2.2.2 PPSPACE $\supseteq$ PSPACE 6
2.3 AM $\subseteq$ PPSPACE 7

3 IP $\supseteq$ PSPACE **8**
3.1 QBF 8
3.1.1 Definition 8
3.1.2 QBF ist PSPACE-vollständig 9
3.2 LFKN Protokoll 9
3.2.1 Permanente 10
3.2.2 Idee des LFKN-Protokolls 10
3.2.3 Protokoll 11
3.3 modifiziertes LFKN Protokoll für QBF 12
3.3.1 Arithmetisierung 12
3.3.2 Protokoll 12

4 Anwendungen in der Programmverifikation **13**

5 Zusammenfassung **14**

1 Einführung

1.1 Motivation

Eines der wichtigsten Problem der theoretischen Informatik ist die Frage, für welche Sprachen effiziente Beweise der Mitgliedschaft eines Wortes existieren. Im Mittelpunkt stehen dabei die Klassen P und NP, aber auch PSPACE ist als Obermenge von NP von Bedeutung. Da auf diesem Gebiet noch fast alle Fragen ungeklärt sind, wurden verschiedene modifizierte Modelle eingeführt. Eines der wichtigsten dabei sind *interactive proofs*, in denen die Worte einer Sprache nur noch mit hoher Wahrscheinlichkeit akzeptiert werden müssen. Die Stärke dieser Beweise liegt dabei in der Kommunikation zwischen zwei Spielern. Dieses Modell, das sicher auch für die Praxis relevant ist, liegt in der Mächtigkeit leicht über der Klasse NP, zumindest wurde es mit dieser Absicht konstruiert. In dieser Hinsicht ist das Resultat IP=PSPACE auch heute noch von großer Bedeutung, zeigt es doch einerseits, daß die Klassen NP und PSPACE nicht weit auseinander liegen. Anderseits resultiert daraus auch, daß die Menge der Sprachen, die man als *effizient beweisbar* bezeichnen kann, wesentlich größer ist als die klassische Sichtweise vermuten läßt. Hierzu ist noch zu bemerken, daß die Klasse MIP, die auch noch als effizient betrachtet werden kann, sogar gleich NEXPTIME ist. Damit sind fast alle praktischen Probleme auf die eine oder andere Art *effizient beweisbar*, ein insgesamt sehr ermutigendes Ergebnis. Hier befassen wir uns nur mit den Klassen IP und PSPACE, die Ausarbeitung basiert dabei in wesentlichen Teilen auf dem Artikel von Lazlo Babai [Bab90].

1.2 Grundlegende Definitionen

Die Klasse PSPACE sollte bekannt sein, zur Vollständigkeit geben wir hier noch einmal kurz die Definition an:
Def: PSPACE
PSPACE = $\{\mathcal{L}|$existiert polynomiell platzbeschränkte TM A mit $L(A) = \mathcal{L}\}$

Im Gegensatz zu PSPACE ist die Klasse IP nicht weithin bekannt. Das Modell wurde 1985 von Goldwasser, Micali und Rackoff eingeführt [GMR85]. Man beachte, daß zur selben Zeit das Modell der Arthur-Merlin Spiele (die Klasse AM) von Lazlo Babai eingeführt wurde [Bab85]. Beide Modelle unterscheiden sich nur in der Art der Zufallsbits (public coin / private coin) und rückblickend ist es nicht überraschend, daß beide Modelle die gleichen Sprachen beschreiben. Wir werden hier daher je nach Bedarf das passende Modell nutzen.

Um IP zu definieren, werden wir zuerst ein interaktives Beweissystem (IPS) einführen:
Def: interactive proof system
Ein interaktives Beweissystem (IPS, interactive proof system), ist ein Paar zweier Turingmaschinen P und V wie folgt: P (Prover) ist unbeschränkt, V (Verifier) deterministisch, polynomiell zeitbeschränkt. P und V greifen dabei auf verschiedene Bänder (Arbeitsbänder, Bänder mit Zufallsbits, Kommunikationsbänder $P \to V$ und $V \to P$) wie in Abbildung1 dargestellt zu. Sei nun A, B ein IPS und $\mathcal{L}$ eine Sprache. Dann ist (A, B) ein IPS für $\mathcal{L}$ falls:

- $x \in \mathcal{L} \Rightarrow$ B hält und akzeptiert mit Wahrscheinlichkeit $\geq \frac{2}{3}$

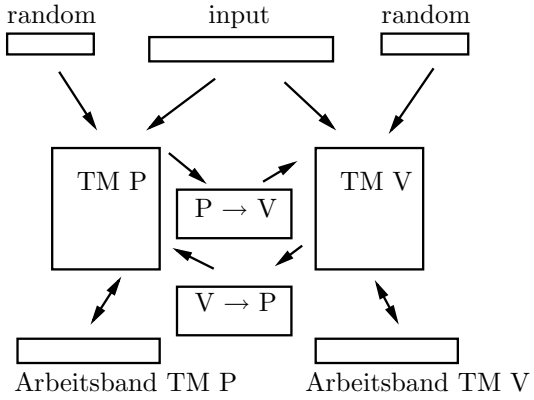

Abbildung 1: IPS

- $x \notin \mathcal{L} \Rightarrow$ B hält und akzeptiert mit Wahrscheinlichkeit $\leq \frac{1}{3}$

Man beachte, daß durch mehrfaches Anwenden die Zuverlässigkeit des Systems beliebig erhöht werden kann. Mit diesen Hilfsmitteln können wir nun die Klasse IP beschrieben:

Def: IP PSPACE = $\{\mathcal{L}|$existiert IPS(A, B) für $\mathcal{L}\}$

Die Klasse AM unterscheidet sich von IP nur dadurch, daß A und B auf das selbe Band mit Zufallsbits zugreifen und es gilt

$$IP = AM$$

2 IP $\subseteq$ PSPACE

Unserer Anschauung nach liegt IP knapp oberhalb von NP, die Abschätzung nach PSPACE sollte daher leicht fallen. Da es schwierig ist, das Prinzip der private coins in einem Modell zu verwirklichen, das nur aus einer Turingmaschine besteht, verwenden wir hier die Klasse AM. Allerdings stellt sich die Frage, wie der potentiell unbeschränkte Prover durch eine platzbeschränkte Turingmaschine simuliert werden kann. Um dieses Problem zu umgehen, wird nun erst eine neue Charakterisierung von PSPACE eingeführt, die einen wesentlich leichteren Vergleich ermöglicht.

2.1 PPSPACE

Unser Ziel ist nun, eine alternative Beschreibung platzbeschränkter Turingmaschinen zu finden, die dem Modell von IP ähnelt. Ein passendes Modell liefern hier die probabilistische Turingmaschinen, die von Papadimitriou entwickelt wurden [Pap83]. Über diese probabilistischen Turingmaschinen definieren wir dann eine neue Klasse von Sprachen PPSPACE. Danach wird gezeigt, daß diese neue Klasse mit PSPACE identisch ist und gleichzeitig leicht als Obermenge

von AM angesehen werden kann. Wir zeigen also ($\star$) und ($\star\star$) in

$$IP = AM \underbrace{\subseteq}_{\star} PPSPACE \underbrace{=}_{\star\star} PSPACE$$

2.1.1 probabilistische Turingmaschinen

Def: Sei P eine Turingmaschine. Ohne Einschränkung habe P bei jedem Schritt die Wahl zwischen genau zwei Möglichkeiten. Dann heißt P probabilistische Turingmaschine, falls sie in allen ungeraden Berechnunsschritten eine zufällige Wahl trifft und in allen geraden Schritten (nichtdeterministisch) unbeschränkt entscheidet.

Eine probabilistische Turingmaschine P und eine Eingabe x implizieren einen Berechnungsbaum T von P auf x, indem jeder Pfad in T einer möglichen Berechnung entspricht. Ohne Einschränkung repräsentieren nur die Blätter akzeptierende bzw. verwerfende Zustände. Falls T kein vollständiger Baum ist, kann er leicht durch Hinzufügen von Transitionen von akzeptierenden (verwerfenden) Zuständen in sich selbst zu einem vollständigen Baum erweitert werden. Knoten, die Entscheidungen in geraden (ungeraden) Schritten zugeordnet sind, werden berechnende (probabilistische) Knoten genannt.

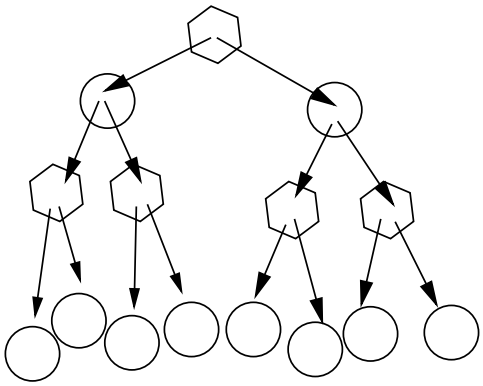

Abbildung 2: Berechnungsbaum

P akzeptiert nun x genau dann, falls in jedem berechnendem Knoten von T ein Ast entfernt werden kann, sodass in dem entstehenden Teilbaum mehr akzeptierende als verwerfende Blätter existieren. Ein solcher Teilbaum heißt akzeptierender Teilbaum von P auf x. Man beachte, daß in jedem solchen Teilbaum das Verhältnis von akzeptierenden und verwerfenden Blättern gleich der Wahrscheinlichkeit ist, daß P in einem Lauf auf x einen akzeptierenden Zustand erreicht (unter der Bedingung, daß P in geraden Schritten genau die Entscheidung trifft, deren Ast nicht entfernt wurde).

Sei P eine probabilistischen Turingmaschine.
$L(P) := \{x \in \Sigma^* |$ es existiert ein akzeptierender Teilbaum von P auf $x\}$.
PPSPACE $:= \{L |$ es existiert eine probabilistische Turingmaschine P mit $L(P) = L\}$

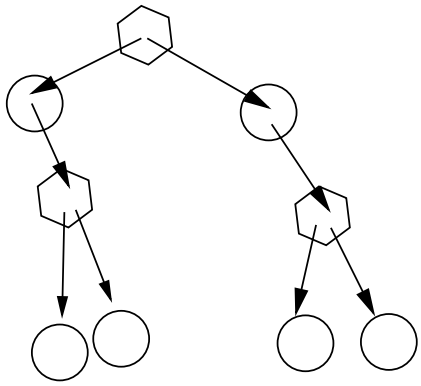

Abbildung 3: akzeptierender Teilbaum

2.2 PPSPACE = PSPACE

Um später AM mit PSPACE abzuschätzen zu können, zeigen wir nun erst:
Satz: PPSPACE = PSPACE
Beweis: Wir zeigen dazu beide Richtungen:

2.2.1 PPSPACE $\subseteq$ PSPACE

Hier muss zu einer gegebenen probabilistischen Turingmaschine P eine polynomiell platzbeschränkte Turingmaschine N konstruiert werden, die P simuliert. Wir konstruieren N wie folgt:

Gegeben P und eine Eingabe x, berechnet N zuerst den kompletten Berechnungsbaum T von P auf x. Anstatt nun zu versuchen, die Entscheidungen von P zu berechnen, wird in T einfach ein Teilbaum nichtdeterministisch geraten. Falls der geratene Teilbaum ein akzeptierender Teilbaum ist, akzeptiert sonst verwirft N.

Es ist sofort klar, daß P x genau dann akzeptiert, wenn auch ein Lauf existiert, in dem N x akzeptiert.

$$\Rightarrow L(P) = L(N)$$

2.2.2 PPSPACE $\supseteq$ PSPACE

Wir nutzen jetzt aus, daß wir nur eine deterministische polynomiell platzbeschränkte Turingmaschine D auf einer Eingabe x simulieren müssen. Diese Simulation werden wir in den unbeschränkten Schritten durchführen, die randomisierten werden vollständig ignoriert.

Gegeben D und x arbeitet P wie folgt: Der ersten Schritt ist ein probabilistscher, das erste Zufallsbit entspricht also entweder 0 oder 1

- Falls es 0 ergibt, verhält sich P unabhängig der weiteren Zufallsbits immer wie D. Falls also D auf x einen akzeptierenden Zustand erreicht, dann auch

P (und umgekehrt). Wichtig ist, daß alle Berechnunsmöglichkeiten gleich lang sind (D ist deterministisch).

- Falls das erste Zufallsbit einer 1 entspricht, ignoriert P D und erreicht genau dann einen akzeptierenden Zustand, falls alle weiteren Zufallsbits ebenfalls 1 sind. Diese Berechnungen werden dabei der Länge der ersten Möglichkeit angeglichen.

Betrachtet man nun den Berechnungsbaum von P auf x, dann entspricht er der Grafik 4. Die Blätter $x_1, \ldots, x_n$ sind genau dann akzeptierende Blätter , falls D die Eingabe x akzeptiert. Die Blätter $y_1, \ldots, y_{n-1}$ sind immer verwerfende Blätter und das Blatt y_n ist immer ein Akzeptierendes.

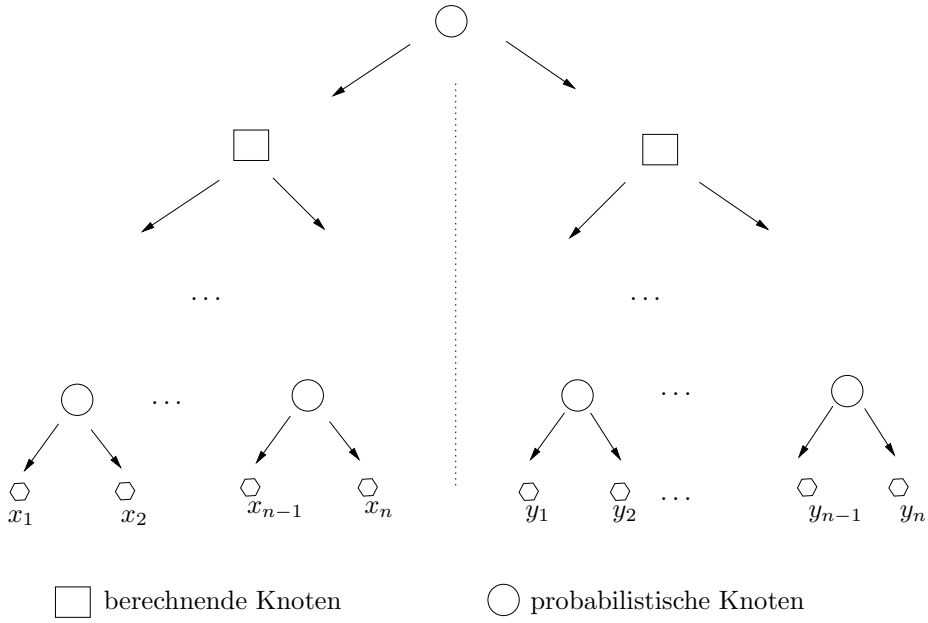

Abbildung 4: Berechnungsbaum

Es ist leicht zu sehen, daß genau dann ein akzeptierender Teilbaum existiert, falls D x akzeptiert. In diesem Fall sind alle Blätter im linken Teilbaum akzeptierende und ergeben zusammen mit dem rechtesten Blatt, die Möglichkeit einen akzeptierenden Teilbaum zu konstruieren. Akzeptiert D dagegen nicht, enthält der Baum genau ein akzeptierendes Blatt.

$$\Rightarrow \text{PPSPACE} = \text{PSPACE}$$

2.3 AM $\subseteq$ PPSPACE

Mit der jetzigen Formulierung von PPSPACE stellt sich ein direkter Vergleich mit AM immer noch schwierig da. Zwar sind Zufallsbits und unbeschränkte

Komponenten in beiden Modellen enthalten, aber es bleiben noch Kommunikation zwischen zwei Turingmaschinen und akzeptierende Teilbäume als gravierende Unterschiede. Um zu dem gewünschten Resultat zu kommen, betrachten wir daher PPSPACE aus einem anderen Blickwinkel.

Betrachten wir einen Lauf der probabilistischen Turingmaschine P auf x. Falls für diese Instanz ein akzeptierender Berechnungsbaum existiert und sich P entsprechend verhält, erreicht sie in mehr als der Hälfte aller Fälle ein akzeptierendes Blatt. Die Existenz eines akzeptierenden Baumes ist also äquivalent zur Forderung, daß die Turingmaschine P mit Wahrscheinlichkeit $\frac{1}{2}$ akzeptiert, da alle Berechnungen im Baum gleich lang sind.

Weiterhin können wir berechnenden Schritte und probabilistische Schritte auf zwei kommunizierende Turingmaschinen aufteilen. Einen unbeschränkten Prover für berechnende und einen polynomiell zeitbeschränkten Verifier, der in Abhängigkeit von Zufallsbits nur 0 oder 1 sendet.

Vergleichen wir dies nun mit der Klasse AM bleiben nur zwei Unterschiede:

1. Der Verifier der Klasse AM darf eine Strategie verfolgen
2. AM verlangt eine Wahrscheinlichkeit von $\frac{2}{3}$ zum Akzeptieren

Der erste Punkt kann jedoch ignoriert werden, da der Prover die Strategie ebenfalls berechnen kann und dann nur noch die Zufallsbits vom Verifier erhalten muss (Eine Strategie, die nicht komplett zufällig ist, bringt keine Vorteile). Offensichtlich schränkt der zweite Punkt AM gegenüber PPSPACE nur ein, sodaß sich folgern läßt:

$$\text{IP} = \text{AM} \subseteq \text{PPSPACE} = \text{PSPACE}$$

3 IP $\supseteq$ PSPACE

Diese Inklusion ist wesentlicher aufwendiger zu beweisen, wir werden dabei in drei Schritten vorgehen. Zuerst führen wir das Entscheidungsproblem QBF ein und beweisen, daß es PSPACE-vollständig ist [SM73]. Können wir dann für dieses Problem ein IPS angeben, folgt IP $\supseteq$ PSPACE mittels Reduktion aller Problem auf QBF. Um ein solches IPS zu finden, wird dann das LFKN Protokoll für die Permanente einer Matrix eingeführt. Wir zeigen, daß dieses LFKN Protokoll ein gültiges IPS ist. Schließlich wandeln wir das LFKN-Protokoll so um, daß es zur Lösung von QBF genutzt werden kann.

3.1 QBF

3.1.1 Definition

Das Entscheidungsproblem, das ab jetzt im Mittelpunkt unserer Bemühungen steht, läßt sich als Erweiterung des bekannten SAT um Quantoren beschreiben. Die Eingabe ist eine quantifizierte boolsche Formel φ (kurz QBF) und das Problem ist zu Entscheiden, ob φ erfüllbar ist oder nicht. Intuitiv erhöht sich die

Schwierigkeit dadurch, daß Aussagen der Form *für alle x existiert y sodaß für alle z* beliebig geschachtelt werden können und es daher nicht mehr reicht, eine Lösung zu raten. Formal definieren wir QBF als:

$$\begin{aligned} QBF &= \{\varphi \subset \Sigma_{Logic} | \varphi \text{ gültige Formel }, \varphi \text{ erfüllbar}\} \\ \Sigma_{Logic} &= \{\wedge, \vee, \exists, \forall, \neg\} \cup \{x_i | i \in \mathbb{N}\} \end{aligned}$$

3.1.2 QBF ist PSPACE-vollständig

Wir zeigen die PSPACE-Vollständigkeit von QBF nicht durch Reduktion eines anderen Problems, sondern direkt für alle anderen Turingmaschinen T mit $L(T) \in$ PSPACE. Das heißt, wir müssen ein Verfahren angeben, das zu einer gegebenen polynomiell platzbeschränkten Turingmaschine T und Eingabe x eine Formel konstruiert, die genau dann erfüllbar ist, falls T x akzeptiert. Unser Verfahren ähnelt dabei stark dem Beweis des Satzes von Cook.

- Die Zustände von T, Kopfposition und Bandinhalt werden durch Variablen dargestellt.
- Formeln für gültige Konfigurationen, Startzustände (für feste Eingabe x) $I_x(U)$ und Endzustände $F(U)$ sind leicht zu konstruieren.
- Gegeben zwei Konfigurationen U, V können wir leicht eine Formel $A_0(U, V)$ angeben, die genau dann erfüllt ist, falls T von U nach V in einem Berechnungsschritt gelangt.
- Dann ist die Formel $A_k(U, V) = \exists W \exists Y \exists Z[(U = Y \wedge W = Z) \vee (W = Y \wedge V = Z)] \longrightarrow A_k(Y, Z)$ genau dann erfüllt, falls T in 2^k Schritten von U nach V gelangt.
- Da T polynomiell-platzbeschränkt ist, verläuft die Berechnung von T auf x in t Schritten mit $t \leq 2^{p(|x|)}$ für ein Polynom p. Damit kann jede Berechnung durch die Formel $A_{p(|x|)}$ ausgedrückt werden.

Für die Berechnung von T generieren wir nun die Formel $\varphi = \exists U \exists V I_x(U) \wedge F(V) \wedge A_{p(|x|)}(U, V)$. Es ist leicht ersichtlich, daß φ genau dann erfüllbar ist, falls T die Eingabe x akzeptiert. Damit ist gezeigt, daß sich jedes Problem in PSPACE auf QBF reduzieren läßt. Da QBF offensichtlich in PSPACE liegt, folgt direkt, daß QBF PSPACE-vollständig ist.

3.2 LFKN Protokoll

Um später ein Protokoll für QBF entwerfen zu können, betrachten wir nun das LFKN Protokoll [LFKN89]. Dieses wurde ursprünglich für die Berechnung der Permanente entwickelt, kann jedoch leicht an QBF angepasst werden [Sha89, Sha90]. Zum einfacheren Verständnis wird zuerst das unveränderte Protokoll erläutert und danach die Anpassung an QBF erklärt. Am Beispiel der Permanente wird auch die Mächtigkeit dieses Werkzeuges deutlich, denn bisher scheint die Berechnung der Permanente nicht in NP zu liegen.

3.2.1 Permanente

Die Permanente ist ähnlich zur Determinante einer Matrix definiert und kann ebenfalls durch eine Reihenentwicklung berechnet werden. Allerdings ist bisher keine effizientere Methode zur Berechnung der Permanente bekannt.

Def: Für eine eine 2×2 Matrix A ist die Permanente $Per(A)$ definiert als $Per(A) = a_{11}a_{22} + a_{21}a_{12}$. Für eine $n \times n$ Matrix mit $n > 2$ ist die Permanente definiert als $Per(A) = \sum_{i=1}^{n} a_{1i}Per(A_{1,i})$. Dabei sei A_{ij} die Matrix, die aus A durch Streichen der Zeile i und Spalte j entsteht.

Dies entspricht der Reihenentwicklung der Determinante bei der alle negativen Vorzeichen durch ungerade Zeilen/Spalten ignoriert werden. ($Det(A) = \sum_{i=1}^{n}(-1)^{i+1}a_{1i}Per(A_{1,i})$)

3.2.2 Idee des LFKN-Protokolls

Das Problem bei der Reihenentwicklung der Permanente ist offensichtlich, daß für eine $n \times n$ Matrix auch n Teilprobleme gelöst werden müssen, die alle nur minimal leichter als das Ursprungsproblem sind. Dadurch erhält man schnell einen Aufwand in der Größenordnung von $n!$.

Die Idee des LFKN-Protokolls ist nun, statt in einem Schritt die Permanenten von verschiedenen Teilmatrizen $A_1, \ldots, A_n$ zu berechnen, alle Teilmatrizen zu einer neuen Matrix D zu verschmelzen und dann deren Permanente zu berechnen. Dabei wird D so konstruiert, daß ein Fehler bei der Ursprungsmatrix A auch mit hoher Wahrscheinlichkeit bei D auftritt. Am Anfang ist eine Matrix A gegeben, der Prover veröffentlicht dann $Per(A)$.

Betrachten wir für eine $n \times n$ Matrix A zuerst eine leichte Vereinfachung. Die Permanente einer Matrix A entstehe nur aus zwei kleineren Matizen (anstelle vobn n):

$$Per(A) = a_1Per(A_1) + a_2Per(A_2) \tag{1}$$

Sei $D(x) = xA_1 + (1 - x)A_2$. Aus der Reihenentwicklung folgt dann, daß $Per(D(x))$ ein Polynom vom Grad $\leq n$ ist. Der Prover veröffentlicht nun $Per(A)$ und das Polynom $p(x) := Per(D(x))$, beispielsweise indem er alle Koeffizienten verschickt. Der Verifier kann nun leicht $Per(A_1)$ und $Per(A_2)$ berechnen und damit prüfen, ob die Gleichung 1 erfüllt ist. Im nächsten Schritt wählt der Verifier zufällig ein y und berechnet damit die Matrix $D(y)$. Außerdem kann er mittels des vom Prover erhaltenen Polynoms p den Wert $p(y)$ berechnen. Im nächsten Schritt wird dann mit $D(y)$ und dem Wert $p(y)$ fortgefahren. Sobald die Matrix klein genug ist, kann der Verifier den Wert der Permanente direkt bestimmen und mit $p(y)$ vergleichen.

- Falls der Prover sowohl die Permanente der Matrix A als auch das Polynom p korrekt angegeben hat, gilt natürlich für die Polynome $p(x) = Per(D(x))$. Damit ergibt der Test des Verifiers immer das korrekte Ergebnis.

- Falls der Prover bei $Per(A)$ betrogen hat, muss auch das Polynom $p(x)$ falsch sein. Damit gilt dann $p(x) \neq Per(D(x))$. Da beide Polynome den Grad kleiner gleich n haben, ist auch der Grad von $p(x) - Per(D(x))$ kleiner gleich n. Daher hat das Polynom maximal n Nullstellen, also ergibt der Vergleich von $p(x)$ und $Per(D(x))$ an der Stelle y nur in n Fällen ein falsches Ergebnis. Falls der Verifier y aus einer Menge Y wählt, ist damit die Wahrscheinlichkeit, daß ein Betrug nicht auffällt, kleiner als $\frac{n}{|Y|}$.

Um die Permanente einer Matrix mittels

$$Per(A) = a_1 Per(A_1) + \ldots + a_n Per(A_n) \tag{2}$$

zu berechnen, muss man nun obiges Verfahren $n-1$ mal anwenden, um alle Teilmatrizen zu einer Matrix verschmelzen. Falls der Prover dabei betrügt (also einen falschen Wert für $Per(A)$ veröffentlicht), muss er mit hoher Wahrscheinlichkeit auch bei der kleineren Matrix $D(x)$ betrügen. Auf diese Weise kann man in n Schritten eine Matrix generieren, deren Permanente der Verifier direkt berechnen kann.

3.2.3 Protokoll

Sei für $n \times n$ Matrix A:

$$\begin{aligned} p_k(x_1, \ldots, x_k) &= Per(x_k \cdot p_{k-1}(x_1, \ldots, x_{k-1}) + (1 - x_n) \cdot A_k) \\ p_1(x_1) &= Per(x_1 \cdot A_1 + (1 - x_1) \cdot A_2) \end{aligned}$$

und

$$\begin{aligned} D_k(x_1, \ldots, x_k) &= x_k \cdot p_{k-1}(x_1, \ldots, x_{k-1}) + (1 - x_n) \cdot A_k \\ D_1(x_1) &= x_1 \cdot A_1 + (1 - x_1) \cdot A_2 \end{aligned}$$

Protokoll zur Berechnung der Permanente von A:

Eingabe: Matrix A mit $Per(A) = a_1 Per(A_1) + \ldots + a_n Per(A_n))$ (3)

1	Prover	sendet $Per(A)$ und $p_{n-1}(x_1, \ldots, x_{n-1})$
2	Verifier	berechnet mit $p_{n-1}(x_1, \ldots, x_{n-1})$ die Werte für $Per(A_1), \ldots, Per(A_n)$ und testet ob Gleichung 3 erfüllt ist
3	Verifier	wählt $y_1, \ldots, y_n$ und berechnet Matrix $D_{n-1}(y_1, \ldots, y_{n-1})$ und $p_{n-1}(y_1, \ldots, y_{n-1})$
4	Verifier	testet ob $Per(D_{n-1}(y_1, \ldots, y_{n-1}) = p_{n-1}(y_1, \ldots, y_{n-1})$ falls n klein genug sonst wird Protokoll mit $D_{n-1}(y_1, \ldots, y_{n-1})$ und $p_{n-1}(y_1, \ldots, y_{n-1}$ erneut ausgeführt

Wir müssen nun zeigen, daß der Verifier insgesamt polynomiell zeitbeschränkt ist. Dazu stellen wir fest, daß Schritt 2 in $O(n^2)$ ausführbar ist. Schritt 3 benötigt $O(n^3)$ da n Matrizen addiert werden müssen. Der letzte Schritt kann dann in $O(n)$ berechnet werden. Da das Protokoll insgesamt maximal n mal aufgerufen wird, folgt daß der Verifier polynomiell zeitbeschränkt ist.

Die Fehlerwahrscheinlichkeit hängt nur von der Größe des Suchraums für die Zufallszahlen ab. Mit polynomiell vielen Zufallsbits für jede Zufallszahl läßt sich damit die Fehlerwahrscheinlichkeit beliebig verkleinern. Damit ist das LFKN Protokoll ein gültiges IPS für die Berechnung der Permanente, also $Per \in$ IP.

3.3 modifiziertes LFKN Protokoll für QBF

Wenn wir das LFKN Protokoll näher betrachten, stellen wir fest, daß auf Polynomen und Termen gerechnet wurde, ohne spezielle Matrixeigenschaften der Permanente auszunutzen. So könnten wir um zum Beispiel ein Protokoll für die Determinante zu erhalten, indem wir $Per()$ mit $Det()$ austauschen. Wenn wir also QBF in eine entsprechende Form bringen können, erhalten wir automatisch ein passendes IPS.

3.3.1 Arithmetisierung

Die Arithmetisierung einer Formel φ zu einem Polynom $\tilde{\varphi}$ definieren wir induktiv durch

- $\varphi(x) = x_i \rightarrow \quad \tilde{\varphi}(x) = x_i$
- $\varphi(x) = \bar{\psi} \rightarrow \quad \tilde{\varphi}(x) = 1 - \tilde{\psi}$
- $\varphi(x) = \psi_1 \wedge \psi_2 \rightarrow \quad \tilde{\varphi}(x) = \tilde{\psi}_1 \cdot \tilde{\psi}_2$
- $\varphi(x) = \psi_1 \vee \psi_2 \rightarrow \quad \tilde{\varphi}(x) = \tilde{\psi}_1 + \tilde{\psi}_2$
- $\varphi(x) = \exists x_1, \ldots, x_s \psi(x_1, \ldots, x_s) \rightarrow \tilde{\varphi}(x) = \sum_{x_1 \in \{0,1\}}, \ldots, \sum_{x_s \in \{0,1\}} \tilde{\psi}(x_1, \ldots, x_s)$
- $\varphi(x) = \forall x_1, \ldots, x_s \psi(x_1, \ldots, x_s) \rightarrow \tilde{\varphi}(x) = \prod_{x_1 \in \{0,1\}}, \ldots, \prod_{x_s \in \{0,1\}} \tilde{\psi}(x_1, \ldots, x_s)$

Wir können ohne Einschränkung davon ausgehen, daß die Ausgangsformel φ keine freien Variablen enthält, denn diese können sonst durch Existenzquantoren gebunden werden ohne die Erfüllbarkeit zu beeinflussen. Wenden wir die Arithmetisierung auf eine solche Formel φ an, so ist φ genau dann unerfüllbar, falls $\tilde{\varphi} = 0$. (Dies kann leicht per Induktion über den Formelaufbau von φ bewiesen werden).

3.3.2 Protokoll

Ohne Einschränkung tritt jede Variable nur in einem Quantor auf. Sei f_0 die Arithmetisierung der Formel φ, dann definieren wir induktiv

$$\begin{aligned} f_i(x_1, \ldots, x_i) &=: \sum_{x_{i+1} \in \{0,1\}} f_{i+1}(x_1, \ldots, x_{i+1}) \\ \text{bzw. } f_i(x_1, \ldots, x_i) &=: \prod_{x_{i+1} \in \{0,1\}} f_{i+1}(x_1, \ldots, x_{i+1}) \end{aligned}$$

Das heißt f_i entsteht aus f_{i-1}, indem die Variable x_i nicht mehr durch gebunden ist sondern als Parameter der Formel auftritt. Mit dieser Konstruktion gilt

$$f_i(x_1, \ldots, x_i) = f_{i+1}(x_1, \ldots, x_i, 0) + f_{i+1}(x_1, \ldots, x_i, 1) \tag{4}$$

$$\text{bzw. } f_i(x_1, \ldots, x_i) = f_{i+1}(x_1, \ldots, x_i, 0) \cdot f_{i+1}(x_1, \ldots, x_i, 1) \tag{5}$$

Dann ergibt sich das Protokoll im ersten Schritt

1	Prover	sendet f_0 und $f_1(x_1)$
2	Verifier	berechnet $f_1(0)$ und $f_1(1)$ und testet ob Gleichung (4) erfüllt ist
3	Verifier	wählt y_1 und berechnet $f_1(y_1)$
4	Verifier	testet ob $f_1(y_1)$ den korrekten Wert ergibt falls $f_1(y_1)$ leicht zu berechnen sonst wird Protokoll mit $f_1(y_1)$ erneut ausgeführt

bzw. im i-ten Schritt

1	Prover	sendet $f_i(y_1, \ldots, y_i)$ und $f_{i+1}(y_1, \ldots, y_i, x_{i+1})$
2	Verifier	berechnet $f_{i+1}(y_1, \ldots, y_i, 0)$ und $f_{i+1}(y_1, \ldots, y_i, 1)$ und testet ob Gleichung (4) erfüllt ist
3	Verifier	wählt y_{i+1} und berechnet $f_{i+1}(y, \ldots, y_{i+1})$
4	Verifier	testet ob $f_{i+1}(y, \ldots, y_{i+1})$ den korrekten Wert ergibt falls $f_{i+1}(y, \ldots, y_{i+1})$ leicht zu berechnen (i groß) sonst wird Protokoll mit $f_{i+1}(y, \ldots, y_{i+1})$ erneut ausgeführt

Für Laufzeit und Fehlerwahrscheinlichkeit gelten die selben Argumente wie für das ursprüngliche LFKN-Protokoll. Allerdings tritt im Gegensatz zum nicht modifizierten Protokoll noch folgendes Problem auf: Die Polynome $f_{i+1}(y_1, \ldots, y_i, 0)$ und $f_{i+1}(y_1, \ldots, y_i, 1)$ können exponentiellen Grad haben , zum Beispiel

$$f_i(y_1, \ldots, y_i,) \quad := \quad \prod_{x_{i+1} \in \{0,1\}} \prod_{x_{i+2} \in \{0,1\}} \ldots \prod_{x_n \in \{0,1\}} (x_n \cdot y_i)$$

Dieses Problem entsteht immer dann auf, wenn eine Variable in der Ursprungsformel φ hinter vielen Allquantoren auftritt. Um dies zu verhindern muss die Formel modifiziert werden. Die Idee ist dabei, hinter jedem Allquantor eine Substitution aller Variablen durchzuführen. Dadurch tritt dann jede Variable maximal 2 mal hinter einem Allquantor auf. Von links nach rechts wird jedes Auftreten eines Allquantors wie folgt ersetzt

$$\forall x_i \psi(x_1, \ldots x_n) \longrightarrow \forall x_i \exists (x'_1, \ldots, x'_n)(x_1 = x'_1) \wedge \ldots \wedge (x_n = x'_n)\psi(x'_1, \ldots, x'_n)$$

Wir müssen nun nur noch eine Arithmetisierung für $(x_i = x'_i)\psi$ angeben:

$$(x_i = x'_i)\psi \longrightarrow [(x_i x'_i) + (1 - x_i)(1 - x'_i)]\tilde{\psi}$$

Weiterhin muss der Prover alle Polynome f_i in eine normierte Form bringen, damit der Verifier $f_i(x_1, \ldots, x_i) = f_{i+1}(x_1, \ldots, x_i, 0) + f_{i+1}(x_1, \ldots, x_i, 1)$ (bzw. $f_i(x_1, \ldots, x_i) = f_{i+1}(x_1, \ldots, x_i, 0) \cdot f_{i+1}(x_1, \ldots, x_i, 1)$) effizient testen kann. Die übliche normierte Darstellung ist dazu ausreichend. Außerdem können während der Berechnung exponentiell große Werte auftreten. Um dies zu verhindern, berechnet der Prover eine Primzahl p und daß Protokoll wird in F_p durchgeführt. Dazu muss der Verifier testen, ob p Primzahl ist oder der Prover einen Zeugen senden. Das daraus entstehende modifizierte Protokoll ist ein gültiges IPS für QBF darstellt, somit gilt QBF $\in$ IP. Zusammen mit den vorigen Resultaten folgt

$$IP = PSPACE$$

4 Anwendungen in der Programmverifikation

Wir zeigen nun zum Abschluss noch, wie die vorgestellten Mittel zur Programmverifikation eingesetzt werden können. Unser Ansatz läßt sich auf jedes Problem

in PSPACE anwenden. Aus Gründen der Übersichtlichkeit beschrieben wir die Verfahren dabei für die Permanente, für andere Probleme müssen die Werte angepasst werden.

instance checking Gegeben einen Algorithmus A für die Permanente. Indem der Algorithmus die Rolle des Provers spielt und wir den Part des Verifiers übernehmen, kann leicht getestet werden, ob der Algorithmus auf einer Eingabe x das korrekte Ergebnis liefert [BK88].

Fehlerhäufigkeit Durch Anwenden des instance checkings auf n^c zufällig generierten Matrizen kann sichergestellt werden, daß der Algorithmus mit Wahrscheinlichkeit $\geq 1 - n^{-2}$ korrekt arbeitet.

self-correcting Um die Permanente einer Matrix A zu bestimmen, wird zufällig eine Matrix B gewählt und die Permanenten von $A + iB$ für alle $i = 1, \ldots, n$ bestimmt werden. Durch Interpolation läßt sich dann der Wert der Permanente von A interpoliert werden. Falls der Algorithmus maximal mit Wahrscheinlichkeit n^{-2} einen Fehler macht, läßt sich durch k-faches Wiederholen sicherstellen, daß die Ausgabe des Algorithmus mit Wahrscheinlichkeit $\geq 1 - 2^{-k}$ korrekt ist [BLR90].

5 Zusammenfassung

Wir haben IP = PSPACE gezeigt und damit die Menge der effizient beweisbaren Sprachen im Vergleich zur klassischen Sicht stark erweitert. Weiterhin haben wir das LFKN-Protokoll eingeführt und gezeigt, wie es an QBF angepasst werden kann. Dadurch ergaben sich auch verschiedene Möglichkeiten der Programmverifikation für Sprachen in PSPACE.

Offenen bleibt, ob die Inklusion NP $\subseteq$ PSPACE echt ist und ob es zwischen NP und PSPACE bzw. IP liegende Klassen gibt. Weiterhin ist zu beachten, daß der Prover im LFKN Protokoll unbeschränkt ist. Die Frage stellt sich, inwieweit diese Mächtigkeit noch eingeschränkt werden kann.

Literatur

[Bab85] L. Babai. Trading group theory for randomness. In *Proceedings of the 17-th ACM Symp. on Theory of Computing*, 1985.

[Bab90] L. Babai. E-mail and the unexpected power of interaction. In *Proceedings of the 5-th Structure in Complexity Theory*, 1990.

[BK88] M. Blum and S. Kannan. Designing programs that check their work. In *Proceedings of the 21-th ACM Symp. on Theory of Computing*, 1988.

[BLR90] M. Blum, M. Luby, and R. Rubinfeld. Selftesting and selfcorrecting programs, with applications to numerical programs. In *Proceedings of the 22-th ACM Symp. on Theory of Computing*, 1990.

[GMR85] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the 17-th ACM Symp. on Theory of Computing*, 1985.

[LFKN89] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. The polynomial time hierarchy has interactive proofs, 1989.

[Pap83] C.H. Papadimitriou. Games against nature. In *Proceedings of the 24-th IEEE Symp. on Foundations of Computer Science*, 1983.

[Sha89] A. Shamir. IP = PSPACE, e-mail announcement , 1989.

[Sha90] A. Shamir. IP=PSPACE (interactive proof=polynomial space). In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 1990.

[SM73] L.J. Stockmeyer and A.R. Meyer. Word problems requiring exponential time. In *Proceedings of the 5-th ACM Symp. on Theory of Computing*, 1973.