

Universität Paderborn  
Fakultät für Elektrotechnik, Informatik und Mathematik  
Fachgruppe Algorithmen und Komplexität  
Fürstenallee 11  
33102 Paderborn

Ausarbeitung  
Der Computer programmiert sich selbst

Sven Köhler  
Matrikelnummer 6089897  
18. Januar 2005

Betreuer: Dr. Martin Ziegler

## Zusammenfassung

Der hier vorgestellte Algorithmus  $M_{p^*}$  löst jedes wohl-definierte Problem  $p^*$ . Die Laufzeit beträgt dabei asymptotisch nur das 5 fache der Zeit des „schnellsten“ Algorithmus plus einiger additiver Terme niedriger Ordnung. Der Algorithmus  $M_{p^*}$  verteilt dabei die Rechenressourcen optimal an 3 Teilprogramme: Suche nach Algorithmen, die das Problem *beweisbar* lösen, Evaluieren der Laufzeitschranken, Simulation des schnellsten Algorithmus. Dabei wird ausgenutzt, dass die Menge der korrekten Beweise aufgezählt werden kann.

## Inhaltsverzeichnis

<b>1</b>	<b>Einleitung und Hauptergebnis</b>	<b>1</b>
<b>2</b>	<b>Speed-Up Algorithmus SIMPLE</b>	<b>2</b>
<b>3</b>	<b>Anwendbarkeit von <math>M_{p^*}</math></b>	<b>2</b>
<b>4</b>	<b>Der Algorithmus <math>M_{p^*}</math></b>	<b>3</b>
<b>5</b>	<b>Laufzeitanalyse</b>	<b>6</b>
<b>6</b>	<b>Schlussfolgerungen</b>	<b>8</b>

# 1 Einleitung und Hauptergebnis

Das Suchen und Finden von schnellen Algorithmen ist eine schwierige Aufgabe. Oft wird versucht schnelle auf Probleme oder Problemklassen zugeschnittene Algorithmen zu konstruieren. Die meisten Probleme lassen sich allerdings darauf reduzieren, dass für eine gegebenen Wert  $x \in X$  ein dazugehöriger Wert  $y \in Y$  berechnet werden soll. Ein Algorithmus für solche Probleme muss also im Wesentlichen eine Abbildung  $f : X \mapsto Y$  berechnen.  $f$  kann dabei durch eine formale logische oder mathematische Spezifikation gegeben sein.  $f$  kann allerdings auch in Form eines Algorithmus gegeben sein.

Ideal wäre es, den schnellsten Algorithmus für ein Problem zu finden. Allerdings zeigt Blum's Speed-Up Theorem [1] [2], dass nicht immer ein schnellster Algorithmus existiert. Blum's Speed-Up Theorem sagt im Wesentlichen aus, dass ein Problem existiert für das eine Folge immer schneller werdender Algorithmen existiert.

Hier wird nun ein Algorithmus von Marcus Hutter vorgestellt, der nur Algorithmen untersucht, die beweisbar ein gegebenes Problem lösen und eine effizient berechenbare Laufzeitschranke aufweisen. Es gilt folgendes:

**Theorem 1.1.** [3, THEOREM 1]

Sei  $p^*$  eine formale Spezifikation oder ein Algorithmus anhand dessen  $p^*(x)$  zu berechnen ist. Sei nun  $p$  ein Algorithmus für den **beweisbar**  $\forall x : p(x) = p^*(x)$  und  $\text{time}_p(x) \leq t_p(x)$  gilt. Dann berechnet der Algorithmus  $M_{p^*}$  den Wert  $p^*(x)$  in Laufzeit

$$\text{time}_{M_{p^*}}(x) \leq 5 \cdot t_p(x) + d_p \cdot \text{time}_{t_p}(x) + c_p$$

wobei die Konstanten  $c_p$  und  $d_p$  von  $p$  abhängen, aber nicht von  $x$ .

Der Ausdruck  $\text{time}_f(x)$  bezeichnet dabei die tatsächliche Laufzeit die benötigt wird, um die Funktion  $f$  an der Stelle  $x$  oder den Algorithmus  $f$  für Eingabe  $x$  auszuwerten. Die Funktion  $t_p$  berechnet also obere Schranken für die Laufzeit des Algorithmus  $p$ .

Das Theorem sagt aus, dass egal mit welchem (noch so schnellem) Algorithmus  $p$  man  $M_{p^*}$  vergleicht,  $M_{p^*}$  asymptotisch maximal 5-mal so langsam ist. Allerdings muss man besonders auf den Begriff „beweisbar“ achten, denn dahinter verbergen sich Einschränkungen, die in Kapitel 4 näher erläutert werden. In der Summe taucht außerdem der Term  $\text{time}_{t_p}(x)$  auf, also die Laufzeit um die Laufzeitschranke von  $p$  zu berechnen. Diese Laufzeit könnte asymptotisch schneller wachsen als  $t_p(x)$ . Es genügt allerdings, wenn  $t_p(x)$  nur eine obere Schranke für die Laufzeit von  $p$  ist, und diese sollte effizient zu berechnen sein. Außerdem wird davon ausgegangen, dass Laufzeitschranken für praktische Probleme effizient berechnet werden können, so dass die Funktion  $\text{time}_{t_p}(x)$  asymptotisch schwächer wächst, als  $\text{time}_p(x)$ .

Der praktische Nutzen von  $M_{p^*}$  wird etwas durch die Konstante  $c_p$  gemindert. Diese kann recht gross werden, und damit schlägt das asymptotische Laufzeitverhalten von  $M_{p^*}$  erst recht spät durch. Die Konstante  $c_p$  wird in Kapitel 5 genauer abgeschätzt.

## 2 Speed-Up Algorithmus SIMPLE

Einleitend sei hier ein Beispiel eines Speed-Up-Algorithmus gegeben. Der Algorithmus SIMPLE ist ein Speed-Up-Algorithmus für Invertierungsprobleme. Ein Invertierungsproblem besteht darin, zu einer Funktion  $g : Y \mapsto X$  und einem vorgegebenen  $x \in X$  ein  $y \in Y$  zu finden, welches die Gleichung  $g(y) = x$  erfüllt.  $g$  muss dabei in Form eines Algorithmus gegeben sein.

Um nun die Funktion  $g$  zu invertieren, führt SIMPLE *alle* Algorithmen parallel aus. Um alle Algorithmen aufzählen zu können, genügt es eine Gödelisierung zu kennen. Es ergibt sich die Aufzählung  $(p_k)_{k \in \mathbb{N}}$  aller Algorithmen. SIMPLE führt nun in jedem zweiten Rechenschritt  $p_1$  aus,  $p_2$  in jedem zweiten Schritt der übrigen unbenutzten Schritte,  $p_3$  wieder in jedem zweiten Schritt der unbenutzten Schritte, usw. — es ergibt sich folgendes Schema: 12131214121312151213121412131216...

Die Algorithmen  $p_k$  werden also jeweils mit dem Bruchteil  $2^{-k}$  der gesamten Rechenzeit simuliert. Hält einer der Algorithmen  $p_k$  mit Ausgabe  $y_k$ , so wird dieses Ergebnis zuerst auf  $g(y_k) = x$  getestet. Eine Überprüfung ist nötig, da natürlich die Ausgabe irgendeines Algorithmus  $p_k$  auch *irgendetwas* sein kann und nicht notwendigerweise eine Lösung des Invertierungsproblems darstellt. Bei dem ersten korrekten  $y_k$  hält SIMPLE.

**Lemma 2.1.** [4, SEITE 503] *Existiert ein  $k$ , so dass der Algorithmus  $p_k$  die Funktion  $g$  in Laufzeit  $t(x)$  invertiert, so invertiert SIMPLE die Funktion  $g$  in Laufzeit  $c \cdot t(x) \in O(t(x))$ . Die Konstante  $c$  hängt dabei nur von  $k$  ab.*

Der Algorithmus SIMPLE ist also asymptotisch genauso schnell, wie jeder andere (noch so schnelle) Algorithmus  $p_k$ , schließlich simuliert SIMPLE irgendwann den Algorithmus  $p_k$ , allerdings mit nur dem Anteil  $2^{-k}$  der gesamten Rechenzeit.  $p_k$  wird dabei erst ab dem  $2^{k-1}$ -ten Schritt von SIMPLE simuliert. Dazu kommt noch eine Überprüfung des Ergebnisses von  $p_k$  und damit ein Aufruf von  $g$ . Es ergibt sich daraus in etwa die Laufzeit:

$$\begin{aligned} & 2^k \cdot \text{time}_{g \circ p_k}(x) + 2^{k-1} \\ = & 2^k \cdot (\text{time}_{p_k}(x) + \text{time}_g(p_k(x))) + 2^{k-1} \\ \leq & 2^k \cdot (\text{time}_{p_k}(x) + \text{time}_g(p_k(x)) + 1) \\ = & 2^k \cdot (t(x) + \text{time}_g(p_k(x)) + 1) \end{aligned}$$

Es wird außerdem davon ausgegangen, dass die Überprüfung eines Ergebnisses durch den Aufruf von  $g$  effizienter möglich ist, als das Invertieren der Funktion durch  $p_k$ . Daher wird  $\text{time}_g(p_k(x)) + 1 \leq t(x)$  angenommen und es bleibt im Wesentlichen wie im Lemma behauptet die Laufzeit  $2^{k+1} \cdot t(x) \in O(t(x))$ .

## 3 Anwendbarkeit von $M_{p^*}$

Als Beispielanwendung von  $M_{p^*}$  soll hier das Beispiel der Multiplikation zweier  $n \times n$ -Matrizen dienen. Dieses Beispiel wurde gewählt, da es sich nicht um ein Invertierungs-

problem handelt. Matrixmultiplikation lässt sich auch nicht sinnvoll in ein solches umwandeln, da die Überprüfung der Lösung das Berechnen der Lösung voraussetzt.

Für Matrixmultiplikation gibt es einen Trivialalgorithmus  $p$ . Dieser benötigt maximal die Laufzeit  $2n^3 \in O(n^3)$ . Dieser kann auch als Spezifikation  $p^*$  des Problems dienen. Es existiert allerdings auch ein Algorithmus  $p'$  zur Matrixmultiplikation mit Laufzeit  $c \cdot n^{2.81} \in O(n^{2.81})$ . Die Konstante  $c$  ist allerdings so gross, dass dieser Algorithmus in der Praxis nur für sehr große  $n$  angewendet werden kann.

Nach Theorem 1.1 ist  $M_{p^*}$  maximal 5-mal langsamer als der Algorithmus  $p'$ . Theorem 1.1 kann allerdings erst angewendet werden, nachdem man sich klar gemacht hat, dass die Funktion  $t_{p'}(x) \geq c \cdot n^{2.81}$  durch runden und abschätzen effizient zu berechnen ist. Allerdings ist der Faktor  $c$  in der Laufzeit für  $p'$  so hoch, dass  $p'$  in der Praxis kaum benutzt wird, da der Algorithmus  $p$  mit der Laufzeit  $O(n^3)$  für kleine  $n$  schneller ist. Darüber hinaus macht es wenig Sinn  $M_{p^*}$  zu benutzen, da der Algorithmus  $p'$  bereits bekannt ist.

Es besteht natürlich immernoch die Möglichkeit, dass ein Algorithmus  $p''$  existiert, der geringere Laufzeit (z.B.  $O(n^2 \log(n))$ ) benötigt, als der Algorithmus  $p'$ .  $M_{p^*}$  wäre auch in Bezug auf diesen Algorithmus maximal 5-mal langsamer. Allerdings können Konstanten  $c_p$  und  $d_p$  in der Laufzeit von  $M_{p^*}$  so gross werden, dass sich der Einsatz  $M_{p^*}$  erst für sehr große  $n$  lohnt. Allerdings hat sich schon bei anderen Speed-Up Algorithmen wie Levin Search gezeigt, dass diese erfolgreich implementiert und genutzt werden können [5] [6].

## 4 Der Algorithmus $M_{p^*}$

Der Algorithmus  $M_{p^*}$  iteriert nicht wie SIMPLE über eine Aufzählung aller Algorithmen, sondern über eine Aufzählung aller (korrekten) Beweise in einem formalen axiomatischen System. Dies mag komisch erscheinen, da bekannt ist, dass Computer bzw. Turingmaschinen keine Beweise herleiten können (Unentscheidbarkeit der Arithmetik, Gödelscher Unvollständigkeitssatz [7, S. 221–225]). Man muss sich allerdings klar machen, dass Beweise verifiziert werden können. Dazu wird ein Beweis als Binärstring (z.B. Textdatei) aufgefasst, der aus einer Folge von entsprechend kodierten Formeln besteht. Ein Beweis kann nun auf seine Korrektheit bezüglich eines formalen axiomatischen Systems überprüft werden. Dieses besteht aus einem formalen logisches System  $(\forall, \lambda, y_i, c_i, f_i, R_i, \rightarrow, \wedge, =, \dots)$ , einer Menge von Axiomen sowie einer Menge von Schlussregeln.

Als Beispiel eines Systems vom Axiomen sollen hier die natürlichen Zahlen dienen. Im Jahr 1889 wurden die folgenden Axiome erstmals von Giuseppe Peano angegeben. Das Peanosche Axiomensystem für die natürlichen Zahlen [8] lautet:

- 1 ist eine natürliche Zahl
- Zu jeder natürlichen Zahl gibt es genau eine andere natürliche Zahl, die ihr Nachfolger ist

- 1 ist kein Nachfolger einer Zahl
- Verschiedene Zahlen haben stets verschiedene Nachfolger
- Für eine beliebige Eigenschaft der natürlichen Zahlen gilt:  
Hat die 1 diese Eigenschaft, und hat für jede natürliche Zahl, die diese Eigenschaft besitzt, auch ihr Nachfolger diese Eigenschaft, so haben alle natürlichen Zahlen diese Eigenschaft

Das letzte Axiom ist das Induktionsaxiom, welches die Grundlage für die Beweismethode vollständige Induktion bildet. Aus diesem Axiomensystem gehen allerdings weder die Addition noch Multiplikation hervor. Diese müssen separat definiert werden. Auch die Darstellung der natürlichen Zahlen  $(1, 2, 3, \dots, 10, 11, \dots)$  als Dezimalzahlen muss gesondert definiert werden, um Dezimalzahlen in den Beweisen benutzen zu können. Dazu kommen Axiome und Regeln für Quatoren und Operatoren wie z.B.  $\forall, \exists, =, \neq, >, <$ . Weitere Regeln können dem System hingefügt werden, solange diese keine Widersprüche erzeugen.

Um einen Beweis zu verifizieren überprüft die Maschine einfach, welche der gültigen Regeln von Formel zu Formel angewendet wurde. Ist der Beweis zu kompliziert aufgeschrieben, d.h. wurden mehrere Regeln auf einmal angewendet, so wird der selbe Beweis inklusive zusätzlicher Zwischenschritte unter den längeren Binärstrings wieder auftauchen und gefunden. Somit liefert das Verifizieren aller Binärstrings eine Aufzählung aller korrekten Beweise.

Algorithmen werden innerhalb der Beweise ebenfalls als Binärstrings (z.B. Quelltext) dargestellt. Dies ist kein Problem, denn eine universelle Turingmaschine  $U$  berechnet bei Eingabe des Algorithmus  $p$  in Form eines Binärstrings genau dieselbe Funktion wie der Algorithmus  $p$ .

Für das logische System muss außerdem gelten:

- Es kann ein Prädikat  $u$  definiert werden, so dass die Formel  $\forall x : u(p, x) = u(p^*, x)$  für zwei Algorithmen  $p$  und  $p^*$  genau dann gilt, wenn  $\forall x : U(p, x) = U(p^*, x)$  gilt bzw. wenn  $p$  dieselbe Funktion berechnet wie  $p^*$
- Ein Prädikat  $tm$  kann definiert werden, so dass die Formel  $tm(p, x) = n$  genau dann gilt, wenn die Laufzeit von  $U(p, x)$  genau  $n$  ist ( $\text{time}_U(p, x) = n$ )

---

#### Algorithmus $M_{p^*}$

---

Initialisiere  $L := \emptyset, t_{fast} := \infty, p_{fast} := p^*$ .  
 Starte Algorithmen A und B mit jeweils 10% der Rechenzeit.  
 Starte Algorithmus C mit 80% der Rechenzeit.

---

Der Algorithmus  $M_{p^*}$  startet seine 3 Teilalgorithmen, nachdem er 3 globale Variablen initialisiert hat.  $L$  ist die Liste der bisher gefundenen Algorithmen inkl. deren Laufzeitschranken.  $t_{fast}$  ist die Laufzeitschranke des bisher schnellsten Algorithmus  $p_{fast}$ .

---

**Algorithmus A**

---

**für**  $i := 1, 2, 3, \dots$  **tue**

Nehme  $i$ -ten Binärstring.

**wenn** der Binärstring ein korrekter Beweis ist, **dann**

Isoliere letzte Formel im Beweis.

Hat die Formel die Form  $\forall x : u(p, x) = u(p^*, x) \wedge u(t_p, x) \geq tm(p, x)$  ?

(für beliebige Binärstrings  $p$  und  $t_p$ )

**wenn ja, dann**

Füge  $(p, t_p)$  der Menge  $L$  hinzu.

**Ende**

**Ende**

**Ende**

---

Der Algorithmus *A* nimmt nun der Reihe nach alle Binärstrings und sucht die korrekten Beweise heraus. Er zählt also alle korrekten Beweise auf. Wird dann der Beweis gefunden, dass ein Algorithmus  $p$ , dessen Laufzeit durch die Funktion  $t_p$  beschränkt ist, die selbe Funktion berechnet wie  $p^*$ , dann wird dieser der Liste  $L$  hinzugefügt.

---

**Algorithmus B**

---

**für** alle  $(p, t_p) \in L$  **tue**

Starte  $U(t_p, x)$  parallel mit Bruchteil  $2^{-l(p)-l(t_p)}$  der Rechenzeit.

**wenn**  $U$  für ein  $t_p$  mit  $U(t_p, x) < t_{fast}$  hält, **dann**

Setze  $t_{fast} := U(t_p, x)$ .

Setze  $p_{fast} := p$ .

**Ende**

**Ende**

---

Der Algorithmus *B* rechnet mit Hilfe der Funktionen  $t_p$  die Laufzeitschranken aller gefundenen Algorithmen  $p$  für die Eingabe  $x$  parallel aus. *B* wartet dabei unter Umständen auf *A*, wenn noch keine Algorithmen gefunden wurden. Je länger die Algorithmen und deren Laufzeitschrankenfunktion sind, desto weniger Rechenzeit wird der Berechnung zugeteilt. Sobald eine Laufzeitschranke berechnet wurde, wird diese mit  $t_{fast}$  verglichen. Falls ein neuer schnellster Algorithmus gefunden wurde, werden  $t_{fast}$  und  $p_{fast}$  entsprechend neu gesetzt. Es ist zu beachten, dass nicht mehr als 100% Rechenzeit vergeben werden kann. Daher muss die Kraft'sche Ungleichung  $\sum_{(p, t_p) \in L} 2^{-l(p)-l(t_p)}$  erfüllt sein [9]. Dies lässt sich allerdings leicht durch die Verwendung von Präfix Codes erreichen.

Der Algorithmus *C* simuliert den bisher schnellsten gefundenen Algorithmus maximal  $k$  Schritte lang. Hat der Algorithmus nach weniger als  $k$  Schritten gehalten, so werden die Algorithmen *A* und *B* beendet, und das endgültige Resultat liegt vor. Ansonsten wird  $k$  um den Faktor 2 erhöht und die Schleife nocheinmal durchlaufen.

---

**Algorithmus C**

---

**für**  $k := 1, 2, 4, 8, 16, 32, \dots$  **tue**  
    Weise Variable  $p$  aktuelles  $p_{fast}$  zu.  
    Führe  $U(p, x)$  für  $k$  Schritte aus.  
    **wenn**  $U(p, x)$  in weniger als  $k$  Schritten gehalten hat, **dann**  
        Stoppe  $A$  und  $B$ .  
        Gebe Resultat  $U(p, x)$  zurück.  
    **Ende**  
**Ende**

---

## 5 Laufzeitanalyse

Sei  $p$  ein Algorithmus der beweisbar (bzgl. des formalen axiomatischen Systems) äquivalent zu  $p^*$  ist, und dessen Laufzeit  $\text{time}_p$  dabei ebenfalls beweisbar durch die Funktion  $t_p$  beschränkt ist.

**Algorithmus A)** Sei  $b$  die Länge des kürzesten Beweises, der für  $p$  und  $t_p$  gefunden werden kann. Um einen Beweis der Länge  $b$  aufzuschreiben (nicht herleiten!) braucht man lediglich  $O(b)$  viele Rechenschritte. Um diesen Beweis der Länge  $b$  auf Korrektheit zu überprüfen, muss man im Wesentlichen überprüfen, welche Formeln Axiomen entsprechen und ob die Schlussregeln korrekt angewendet wurden. Dies kostet  $O(b^2)$  Rechenschritte. Es gibt  $2^{b+1}$ -viele Beweise der Länge  $\leq b$ . Daher benötigt der Algorithmus  $A$  maximal  $2^{b+1} \cdot O(b^2)$  Rechenschritte, bis er auf den Beweis für  $p$  und  $t_p$  trifft. Da  $A$  allerdings nur 10% der Rechenzeit von  $M_{p^*}$  bekommt, halten wir die Laufzeit wie folgt fest:

$$T_A \leq 10 \cdot 2^{b+1} \cdot O(b^2)$$

**Algorithmus B)** Um  $t_p(x)$  zu berechnen, werden nach Definition  $\text{time}_{t_p}(x)$  viele Rechenschritte benötigt. Für diese Berechnung steht weiterhin nur der Bruchteil  $2^{-l(p)-l(t_p)}$  der Rechenzeit zur Verfügung. Weiterhin muss  $B$  auf Algorithmus  $A$  warten, bis dieser  $t_p$  überhaupt zu  $L$  hinzugefügt hat. Halten wir die Laufzeit daher wie folgt fest:

$$T_B \leq T_A + 10 \cdot 2^{l(p)+l(t_p)} \cdot \text{time}_{t_p}(x)$$

**Algorithmus C)** Die Laufzeit von  $C$  ist entscheidend für die Laufzeit von  $M_{p^*}$ , denn erst wenn  $C$  beendet wird, werden auch  $A$  und  $B$  beendet.  $C$  berechnet darüber hinaus die eigentliche Ausgabe. Um nun die Laufzeit von  $C$  abschätzen zu können, müssen die folgenden Fälle betrachtet werden:

- Es wurde ein anderer schneller Algorithmus  $p'$  gefunden, wodurch  $C$  hält, bevor  $B$  die Berechnung von  $t_p(x)$  abgeschlossen hat. Es gilt dann:

$$T_C \leq T_B$$

- Die Schleifenvariable  $k$  habe den Wert  $k_0$  und  $C$  simuliert in diesem Durchgang einen von  $p$  verschiedenen Algorithmus  $p'$ , während  $B$  mit der Berechnung  $t_p(x)$

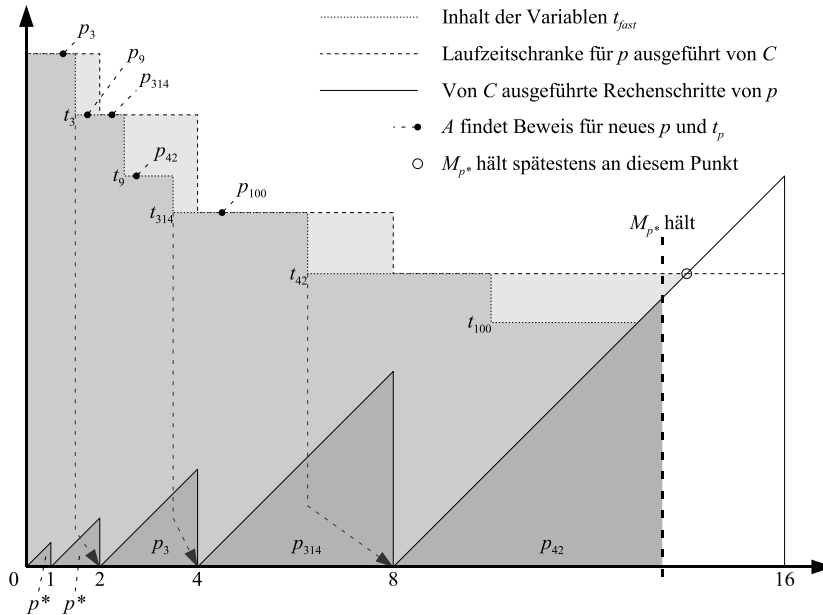


Abbildung 1: Dieses Schaubild zeigt den Verlauf einer Ausführung von  $M_{p^*}$ . Die horizontale Achse beschreibt dabei die von  $M_{p^*}$  konsumierte Rechenzeit. Die vertikale Achse bemisst die verschiedenen auftretenden Laufzeiten. Die untere dunklere gepunktete Stufenentreppe stellt den Inhalt von  $t_{fast}$  dar. Die hellere gestrichelte Stufenentreppe zeigt die Laufzeit des gerade von  $C$  ausgeführten  $p$ . Die Beschriftung der horizontalen Achse zeigt auch das  $k$ , welches  $C$  gerade benutzt. Aus der Simulation von  $k$  Schritten des aktuellen  $p_{fast}$  ergibt sich die Sägezahnkurve. Klar ist, dass  $M_{p^*}$  spätestens hält, wenn sich die gestrichelte Linie und die Sägezahnkurve schneiden. Da die  $t_{fast}$  allerdings nur obere Schranken darstellen, kann  $M_{p^*}$  wie in diesem Beispiel auch früher halten.

fertig wird und  $t_{fast}$  sowie  $p_{fast}$  entsprechend setzt. Es gilt dann  $k_0 \leq 80\% \cdot T_B$ . Wenn nun  $C$  nach diesem Durchgang hält, da  $p'$  terminierte, dann gilt:

$$T_C \leq \frac{2}{80\%} k_0 \leq 2T_B$$

- Wenn  $C$  allerdings im Durchgang  $k_0$  nicht hält, aber  $2k_0 \geq t_{fast}$  gilt, dann wird die Simulation von  $p$  beim nächsten Durchgang halten. Es gilt dann  $timet_p(x) \leq t_{fast} \leq 2k_0$  und damit:  

$$T_C \leq \frac{4}{80\%} k_0 \leq 4T_B$$
- Wenn nicht  $2k_0 \geq t_{fast}$  gilt, dann warten wir auf  $k > k_0$  mit  $\frac{1}{2}k \leq t_p(x) < k$ . Es gilt daher  $80\% \cdot T_C \leq 2k \leq 4t_{fast} \leq 4t_p(x)$  und damit  $T_C \leq 5t_p(x)$

Die 4 Fälle zeigen, dass die Laufzeit von  $C$  und damit die Laufzeit von  $M_{p^*}$  nach oben durch  $4T_B$  sowie  $5t_p(x)$  beschränkt ist. Es gilt damit:

$$\begin{aligned} \text{time}_{M_{p^*}}(x) &= T_C \\ &\leq \max\{4T_B, 5t_p(x)\} \\ &\leq 4T_B + 5t_p(x) \\ &\leq 5t_p(x) + d_p \cdot \text{time}_{t_p}(x) + c_p \end{aligned}$$

mit  $d_p = 40 \cdot 2^{l(p)+l(t_p)}$  und  $c_p = 40 \cdot 2^{b+1} \cdot O(b^2)$ , wobei  $b$  die Länge des kürzesten Beweises für  $p$  und  $t_p$  ist.  $d_p$  und  $c_p$  hängen neben  $p$  auch von  $p^*$  ab, allerdings wird  $p^*$  als gegebene Konstante betrachtet und nicht als Parameter. Daher können diese bei asymptotischer Betrachtungen der Laufzeit vernachlässigt werden.

Dadurch, dass hier und in Theorem 1.1 auf *beweisbar* äquivalente Algorithmen eingeschränkt wurde, ist sichergestellt, dass der Algorithmus  $M_{p^*}$  den Algorithmus  $p$  irgendwann findet. Die hier errechnete und in Theorem 1.1 angegebene Laufzeit von  $M_{p^*}$  ist dabei die maximale Laufzeit, die  $M_{p^*}$  zum Finden und Anwenden von  $p$  benötigt.

## 6 Schlussfolgerungen

Der Algorithmus  $M_{p^*}$  kombiniert 3 Verfahren:

1. Suche in der Menge der korrekten Beweise nach Algorithmen und deren Laufzeitschrankenfunktion
2. Suche des schnellsten Algorithmus durch Berechnen der Laufzeitschranken
3. Stückweise Simulation des bisher schnellsten gefundenen Algorithmus

Dabei wird die Rechnerzeit so geschickt verteilt, dass die in Kapitel 4 errechnete Laufzeit möglich wird.  $M_{p^*}$  lässt sich dabei nicht wie SIMPLE nur auf Invertierungsprobleme anwenden, sondern auf alle Probleme der Form  $f : X \mapsto Y$ . Der Algorithmus  $M_{p^*}$  ist also ein sehr allgemeiner Speed-Up Algorithmus.

Bei der Berechnung der Laufzeit  $M_{p^*}$  wurden großzügige Annahmen über das zugrundeliegende Rechenmodell gemacht.  $M_{p^*}$  führt massiv Teilprogramme parallel aus, was allerdings in der Praxis zu Performanceverlust führen würde. Dem kann entgegen gewirkt werden, indem die Teilprogramme nicht parallel, sondern nacheinander ausgeführt werden: jedes Programm erst 1, dann 2, dann 4, 8, 16, 32, ... Schritte. Dies erhöht die Laufzeit des Algorithmus B auf das 4-fache.

$M_{p^*}$  lässt sich allerdings wegen der hohen Konstanten  $d_p$  und  $c_p$  nur sehr schwer in der Praxis nutzen. Allerdings macht die Tatsache Mut, dass sich andere Algorithmen wie Levin Search trotz großer Faktoren umsetzen ließen [5] [6].

Darüberhinaus ist das formale axiomatische System ausschlaggebend dafür, welche Algorithmen  $p$  der Algorithmus  $M_{p^*}$  finden kann und für welche  $p$  das Theorem 1.1 gilt. Das formale System muss also gut gewählt werden.

## Literatur

- [1] BLUM, M.: A Machine-Independent Theory of the Complexity of Recursive Functions. In: *Journal of the ACM* 14 (1967), April, Nr. 2, S. 322–336. – ISSN 0004–5411
- [2] BLUM, M.: On Effective Procedures for Speeding Up Algorithms. In: *Journal of the ACM* 18 (1971), April, Nr. 2, S. 290–305
- [3] HUTTER, M.: The Fastest and Shortest Algorithm for All Well-Defined Problems. In: *International Journal of Foundations of Computer Science* 13 (2002), Juni, Nr. 3, 431–443. <http://www.hutter1.de/ai/pfastprg.htm>
- [4] LI, M. ; VITÁNYI, P.: *An introduction to Kolmogorov complexity and its applications*. 2. Auflage. Springer, 1997
- [5] SCHMIDHUBER, J.: Discovering Neural Nets with Low Kolmogorov Complexity and High Generalization Capability. In: *Neural Networks* 10 (1997), Nr. 5, S. 857–873
- [6] SCHMIDHUBER, J. ; ZHAO, J. ; WIERING, W.: Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement. In: *Machine Learning* 28 (1997), S. 105–130
- [7] BOOLOS, G. S. ; BURGESS, J. P. ; JEFFREY, R. C.: *Computability and Logic*. vierte. Cambridge University Press, 2002
- [8] HILBERT, D. ; ACKERMANN, W.: *Grundzüge der theoretischen Logik*. 6. Auflage. Berlin : Springer-Verlag, 1949
- [9] KRAFT, L. G.: *A Device for Quantizing, Grouping and Coding Amplitude Modified Pulses*. Cambridge, MA, Diplomarbeit, 1949