mathPAD

Vol 5 No 1, November 1995

In diesem Heft

Redaktion:
Benno Fuchssteiner
Klaus Drescher
Universität-GH Paderborn
D-33095 Paderborn
e-mail:
`benno@uni-paderborn.de`
`kg@uni-paderborn.de`

ISSN 0941-9187
V.i.S.d.P.: Benno Fuchssteiner
Nachdruck gegen Belegexemplar erlaubt

Liebe Leserin, lieber Leser,

etwa ein Jahr ist seit dem Erscheinen der letzten mathPAD verstrichen. Wir waren einfach zu sehr mit *MuPAD* beschäftigt, um an eine neue Ausgabe denken zu können:

- Im November '94 konnte *MuPAD* den Europäischen Hochschul-Softwarepreis in der Kategorie „Mathematik" gewinnen — bei großer internationaler Konkurrenz.
- Im März '95 waren wir einmal mehr auf der CeBIT vertreten, dort haben wir u.a. eine Alpha-Version von *WinMuPAD* präsentiert (der Windows-Version von *MuPAD*).
- Im Juli '95 wurde die neue Version 1.2.2 freigegeben.

Die neue Version 1.2.2 hat entscheidende Fortschritte gebracht, auch wenn sich dies in der Release-Nummer nicht recht ausdrückt. Das mag man unter anderem daran messen, daß *MuPAD* nun 69 der 131 Aufgaben der Test Suite von M. Wester löst (etwa 40 mehr als in der Version 1.2.1). Damit liegt *MuPAD* nun Kopf an Kopf mit den kommerziellen „6 anderen" CA-Systemen — nur die drei anderen „M"-Systeme lösen mehr Aufgaben.

Zu verdanken ist dieser Quantensprung maßgeblich Dr. Paul Zimmermann,[1] der unsere Gruppe für ein Jahr besucht hat. Er streift in seinem Artikel kurz die wesentlichen Neuerungen — für eine ausführliche Beschreibung fehlt leider der Platz. Wir möchten Ihm an dieser Stelle nochmals für seinen tatkräftigen Enthusiasmus danken.

Erwähnt sei noch der Artikel von Sergei O. Ivanov über das Lösen von Ungleichungen. Der begabte junge (14-jährige) Autor aus der Ukraine, der bereits für die letzte Ausgabe einen Artikel schrieb, besucht gerade für längere Zeit unsere Arbeitsgruppe.

Ansonsten gibt es in dieser Ausgabe nicht allzuviel *MuPAD*-spezifisches. Als Forum für Informationen über *MuPAD* haben wir seit einiger Zeit einen eigenen WWW-Server aufgebaut. Die „Adresse" ist

`http://math-www.uni-paderborn.de/MuPAD/`

Dort finden Sie unter anderem alles Wissenswerte zur neuen Version und zur Distribution, schauen Sie doch mal rein!

B.F. & K.D.

[1] INRIA Lorraine, `Paul.Zimmermann@loria.fr`

Maxflow-Algorithmen

Karsten Morisse
Universität GH Paderborn, kamo@uni-paderborn.de

Die Bestimmung eines maximalen Flusses in einem Netzwerk ist eines der klassischen Probleme der Netzwerkflußtheorie. Seit nahezu 40 Jahren beschäftigen sich Wissenschaftler mit diesem Problem.

1 Problemstellung

In der klassischen Netzwerkflußtheorie geht man von der folgenden Situation aus: Es sei $G = (V, E)$ ein gerichteter, zusammenhängender Graph mit Knotenmenge V und Kantenmenge $E \subset V \times V$, wobei $n = |V|$ und $m = |E|$. Ferner seien q und s zwei ausgezeichnete Knoten aus V. Jeder Kante $e \in E$ ist eine positive, reelle Kapazität $\tau(e)$ zugeordnet, welche durch U beschränkt ist. $N = (G, \tau, q, s)$ heißt ein Flußnetzwerk mit Quelle q und Senke s. Eine Abbildung $f : V \times V \to \mathbf{R}$ mit $0 \leq f(i,j) \leq \tau(i,j)$ für alle $(i,j) \in E$ und mit $f(V,i) = f(i,V)$ für alle $i \in V \setminus \{q,s\}$ heißt ein *Fluß* in N. Die erste Bedingung drückt die Zulässigkeit des Flusses aus, d.h. die Kantenkapazitäten werden nicht überschritten. Die zweite Restriktion ist eine Konservativitätsbedingung, die den Verbrauch an Zwischenknoten untersagt. Der *Wert* oder die *Stärke* eines Flusses f ist die Differenz zwischen Ausgangs- und Eingangsfluß in der Quelle q. Beim *Maxflow-Problem* ist nach einem maximalen Fluß von der Quelle zur Senke gefragt.

Das Maxflow-Problem ist ein Spezialfall einer linearen Optimierungsaufgabe und kann somit beispielsweise mit dem Simplex-Algorithmus (z.B. [Dan66] oder Varianten mit speziellen Pivot-Strategien, z.B. [GH90, Gri86, GGT90]) oder einem anderen Verfahren (z.B. [Kar84, Kha79]) gelöst werden. Die schnellsten bekannten Verfahren sind jedoch dedizierte Algorithmen. Eine Vielzahl derartiger Verfahren sind entwickelt worden, von denen ich in diesem Bericht einige vorstellen möchte. Dabei beschränke ich mich auf die Angabe wesentlicher Ideen ohne Beweise. Einen umfassenderen Überblick über die Ergebnisse der Netzwerkflußtheorie zusammen mit vielen Anwendungsbeispielen kann man z.B. in [AMO93] finden.

Man kann die Verfahren aus der Literatur grob in zwei Klassen einteilen. Die älteren Verfahren basieren auf der Suche nach *erweiternden Wegen*. Dabei geht man von einem Fluß f aus und sucht im Netzwerk einen solchen erweiternden Weg von q nach s anhand dessen man f vergrößert. Diese Vorgehensweise hat den Vorteil, das während jeder Phase die Konservativität in den Zwischenknoten gewährleistet ist. Einen anderen Ansatz verfolgen die *Preflow-Push-Algorithmen*. In ihnen betrachtet man eine etwas abgeschwächte Version des Flußbegriffes: ein Preflow entspricht einem Fluß mit der Ausnahme, daß der Eingangsfluß eines Knotens den Ausgangsfluß überschreiten darf. Die dabei auftretenden Überschüsse in den Knoten werden durch Verteilung auf benachbarte Knoten abgebaut. Die Preflow-Push-Algorithmen sind den auf der Suche nach erweiternden Wegen basierenden Algorithmen in verschiedener Hinsicht überlegen. Zum einen zeigen sie sowohl theoretisch wie auch praktisch ein besseres Laufzeitverhalten. Ein zweiter Vorteil ist, daß sie besser für eine parallele Implementation geeignet sind.

2 Erweiternde Wege und blockierende Flüsse

2.1 Der Markierungsalgorithmus von Ford & Fulkerson

Der erste Algorithmus zur Bestimmung eines maximalen Flusses ist der Markierungsalgorithmus von FORD & FULKERSON [FF56]. Er basiert auf der Suche nach erweiternden Wegen. Ausgehend von einem bereits vorhandenen Fluß f bildet man das *Restnetzwerk* N_f, das aus den Knoten von N besteht und die folgenden Kanten besitzt: Ist $f(i,j) < \tau(i,j)$, so nimmt man die Kante (i,j) als Vorwärtskante mit Kapazität $\tau_f(i,j) = \tau(i,j) - f(i,j)$ in N_f auf. Ist $f(i,j) > 0$, so nimmt man (j,i) als Rückwärtskante (entgegen der ursprünglichen Ausrichtung) mit Kapazität $f(i,j)$ in N_f auf. Offensichtlich besitzt dann jede Kante in N_f eine positive Kapazität. Ein *erweiternder Weg* (AP) in N ist ein Weg von q nach s im Restnetzwerk N_f. Mit Hilfe der AP's kann ein Fluß vergrößert werden. Dazu bestimmt man die minimale Restkapazität δ der Kanten auf diesem Weg, und durch die Festsetzung

$$f'(e) := \begin{cases} f(e) + \delta, & e \text{ Vorwärtskante} \\ f(e) - \delta, & e \text{ Rückwärtskante} \end{cases} \tag{1}$$

erhält man einen neuen Fluß f', der einen größeren Wert als f besitzt. Von fundamentaler Bedeutung ist die Tatsache, daß ein Fluß f genau dann maximal ist, wenn es bezüglich f keinen AP in N gibt.

Ein möglicher Ansatz ist es daher, ausgehend vom initialen Fluß f (z.B. dem Nullfluß), solange nach AP's zu suchen und f gemäß (1) zu vergrößern, bis kein solcher Weg mehr gefunden wird. Das Ergebnis ist dann der gesuchte maximale Fluß. Genau diesen Ansatz verfolgt der Markierungsalgorithmus. Er kommt ohne explizite Konstruktion des Restnetzwerkes aus, sondern er verwendet eine modifizierte Breitensuche zur Bestimmung von AP's. Beim Durchlaufen des Netzwerkes werden die Knoten mit Markierungen versehen. Darin ist codiert von welchem Knoten und über welche Art von Kante (Vorwärts- oder Rückwärtskante) ein Knoten erreicht wurde. Ferner ist die minimale Restkapazität des bislang durchlaufenen Weges Teil der Markierung.

Die Laufzeit des Verfahrens hängt in erster Linie von der Auswahl der AP's ab. In der ursprünglichen Formulierung des Markierungsalgorithmus ist keine Strategie zur Bestimmung der AP's angegeben. Ein solcher Weg kann in $\mathcal{O}(m)$ Zeit gefunden werden, er ist jedoch keinesfalls eindeutig. Werden die AP's ungeschickt gewählt, so kann dies zur Konstruktion sehr vieler derartiger Wege führen, da die Vergrößerung des Flusses in jedem Schritt nur sehr gering ist. Die Laufzeit hängt dann nicht nur von der Größe des Netzwerkes ab, sondern auch von den vorkommenden Kapazitäten. Damit hat der Algorithmus exponentiellen Aufwand in der Länge der Eingabe.

Bei irrationalen Kantenkapazitäten kann der Algorithmus allerdings versagen. Einerseits ist die Terminierung nicht garantiert und andererseits konvergiert das Verfahren nicht gegen einen maximalen Fluß[1]. Besitzt der maximale Fluß die Stärke M, so kann man für jedes $d > 1$ ein Netzwerk konstruieren, so daß der Algorithmus gegen einen Fluß mit Stärke $\frac{M}{d}$ konvergiert. Bei ganzzahligen oder rationalen Kapazitäten ist das Ergebnis der maximale Fluß und auch dieser ist dann ganzzahlig bzw. rational.

EDMONDS & KARP [EK72] haben untersucht, wie der Markierungsalgorithmus durch Auswahl spezieller AP's verbessert werden kann. Führt man die Vergrößerung des Flusses entlang eines AP's kürzester Länge (SAP) aus — dazu ist lediglich die Breitensuche zum Finden von AP's geringfügig zu modifizieren, indem man nämlich diejenigen Knoten auswählt, deren Markierung am längsten zurückliegt (dazu kann man die zu bearbeitenden Knoten z.B. in einer Queue speichern) — so führt dies auf einen Algorithmus mit Laufzeit $\mathcal{O}(nm^2)$, der auch im Fall von irrationalen Kantenkapazitäten korrekt arbeitet.

[1] Die kleinsten derartigen Netzwerke besitzen 6 Knoten und 8 Kanten (siehe [Zwi95]).

2.2 Der Algorithmus von Dinic

Die Suche nach SAP's startet beim modifizierten Markierungsalgorithmus jedes Mal von neuem. Hat man jedoch einmal einen kürzesten Weg der Länge k gefunden, so können die nachfolgenden SAP's niemals kürzer als k sein. Diese Tatsache hat DINIC [Din70] zur Verbesserung des Algorithmus ausgenutzt. Zur effizienteren Suche nach SAP's werden *geschichtete Hilfsnetzwerke* verwendet. Solch ein Netzwerk ist ein Teilgraph des Restnetzwerkes und es besteht aus Schichten $V_0 = \{q\}, V_1, \ldots, V_{d-1}, V_d = \{s\}$ von disjunkten Knotenmengen. Jede Kante in diesem Netzwerk führt von einer Schicht V_i in die Schicht V_{i+1}. Durch eine einfache Breitensuche kann ein Schichtennetzwerk in $\mathcal{O}(n + m)$ Zeit konstruiert werden. Ein *blockierender Fluß* ist ein Fluß in einem Schichtennetzwerk, in dem auf jedem Weg von q nach s wenigstens eine Kante mit voller Kapazität durchflossen wird. Dies bedeutet, daß es keinen AP bzgl. dieses Flusses gibt der nur über Vorwärtskanten führt. Schichtennetzwerke haben den Vorteil, daß man in ihnen Wege von q nach s in $\mathcal{O}(n)$ Zeit sehr einfach finden kann, da alle Wege von q nach s in diesem Netzwerk die gleiche Länge haben.

Der Algorithmus besteht aus mehreren Phasen. In jeder Phase wird ein geschichtetes Netzwerk konstruiert und anschließend werden darin AP's bestimmt. Der Algorithmus wäre gegenüber dem Markierungsalgorithmus keine Verbesserung, wenn nach jedem gefundenen AP ein neues Schichtennetzwerk konstruiert werden würde. Stattdessen wird in einem Schichtennetzwerk durch eine Tiefensuche solange nach diesen Wegen gesucht, bis die Senke nicht mehr erreichbar ist. Mit Hilfe der gefundenen Wege wird ein blockierender Fluß gemäß (1) konstruiert, der dann ein blockierender Fluß für das Schichtennetzwerk ist und um den der bisherige Gesamtfluß vergrößert wird. Die Laufzeit einer Phase beträgt $\mathcal{O}(nm)$. Wird in einer Phase kein Weg von q nach s mehr gefunden, so wird die Phase beendet und ein neues Hilfsnetzwerk konstruiert. Dieses hat dann eine größere Anzahl von Schichten als die Hilfsnetzwerke der vorhergehenden Phasen. Der Algorithmus endet, wenn bei der Konstruktion des Hilfsnetzwerkes die Senke von der Quelle aus nicht mehr erreichbar ist. Da es maximal $n - 1$ Phasen geben kann, ergibt sich die Gesamtlaufzeit des DINIC-Algorithmus zu $\mathcal{O}(n^2 m)$.

Durch die Verwendung dynamischer Baumstrukturen haben SLEATOR & TARJAN [ST83] eine Implementation des DINIC-Algorithmus entwickelt, der nur $\mathcal{O}(m \log n)$ Zeit zur Bestimmung eines blockierenden Flusses benötigt. Dies führt zu einem $\mathcal{O}(nm \log n)$ Algorithmus zur Berechnung eines maximalen Flusses. Die verwendete Datenstruktur ist jedoch sehr komplex und aufwendig zu verwalten, und hat sich, obwohl theoretisch in der Laufzeitabschätzung überlegen, gegenüber anderen Methoden praktisch als unterlegen erwiesen (siehe auch Abschnitt 4).

2.3 Preflow-Algorithmus von Karzanov

Beim Algorithmus von DINIC wird bei der Bestimmung blockierender Flüsse pro DFS-Durchlauf nur ein AP bestimmt. Die Berechnung eines blockierenden Flusses benötigt demnach $\mathcal{O}(nm)$ Zeit. KARZANOV [Kar74] konnte diese Laufzeit auf $\mathcal{O}(n^2)$ reduzieren, indem er nicht nur einen AP bestimmt, sondern versucht, mehrere solcher Wege gleichzeitig zu berechnen. Das generelle Schema, die Bestimmung blockierender Flüsse in Schichtennetzwerken, ist identisch zum DINIC-Algorithmus.

Der Algorithmus arbeitet im Gegensatz zu den bisher vorgestellten Verfahren mit Preflows. Ein *Preflow* f ist eine Abbildung $f : V \times V \to \mathbf{R}$ mit $0 \leq f(i,j) \leq \tau(i,j)$ für jedes $(i,j) \in E$ und $f(V,i) \geq f(i,V)$ für alle $i \in V \setminus \{q,s\}$. Die Differenz zwischen Eingangs- und Ausgangsfluß in einem Knoten i bezeichnet man mit *Überschuß* $e_f(i)$. Ein Knoten $i \in V \setminus \{q,s\}$ heißt balanciert, falls $e_f(i) = 0$ gilt. Andernfalls heißt der Knoten *unbalanciert*. Das Restnetzwerk bezüglich eines Preflows f bildet man exakt so wie im Fall von Flüssen. Der KARZANOV-Algorithmus verwendet bis zur Terminierung Preflows. Erst dann sind alle Knoten balanciert, so daß der dann anliegende Preflow ein Fluß ist.

Während der Durchführung des Algorithmus ist jeder Knoten in einem der beiden Zustände *blockiert* oder *frei*. Für einen blockierten Knoten v gilt: auf jedem Weg von v zur Senke gibt es wenigstens eine gesättigte Kante. Das Verfahren führt eine Folge von `Push` und `Balance` Operationen aus. Eine `Push`-Operation versucht unbalancierte, freie Knoten durch Erhöhung des Ausgangsflusses zu balancieren. `Push(i)` schiebt den Fluß von Schicht V_i in die Schicht V_{i+1}. Dabei werden alle Knoten mit positivem Überschuß behandelt. Die `Balance`-Operation reduziert bei unbalancierten, blockierten Knoten den Eingangsfluß. `Balance(i)` versucht die Knoten der Schicht V_i zu balancieren. Dabei wird der Überschuß an Knoten der vorhergehenden Schicht V_{i-1} zurückgegeben. Nach dem Aufruf von `Balance(i)` sind alle Knoten der Schicht V_i balanciert. Der Algorithmus führt wiederholt Vorwärts- und Rückwärtsbewegungen aus. In einer Vorwärtsbewegung wird die `Push`-Operation auf die Schicht mit dem kleinsten Index angewandt, die unbalancierte, freie Knoten enthält. Es werden solange `Push`-Operationen ausgeführt, wie Fluß an eine nachfolgende Schicht abgegeben werden kann. Dem Aufruf von `Push(i)` folgt dann der Aufruf von `Push(i+1)`. Eine Rückwärtsbewegung führt wiederholt die `Balance`-Operation auf die Schicht mit dem größten Index aus, die unbalancierte Knoten enthält. Sobald in einer Vorgängerschicht ein freier Knoten balanciert wird, wird wieder eine Vorwärtsbewegung gestartet.

Für den Korrektheitsnachweis und die Laufzeitabschätzung des Algorithmus sind folgenden Aussagen wichtig:

1. Wird ein Knoten einmal blockiert, so bleibt er es für die restliche Dauer des Algorithmus.
2. Jeder Knoten wird höchstens einmal balanciert.
3. Die Aufrufe `Push(i)` und `Balance(i)` werden höchstens n-mal wiederholt.

Die letzte Aussage liefert die Terminierung des Algorithmus. Zum Zeitpunkt der Terminierung sind alle Knoten balanciert, d.h. der Preflow ist ein Fluß. Nach `Push(0)` ist die Quelle blockiert[2]. Der Knoten bleibt wegen 1. bis zur Terminierung blockiert. Dies bedeutet aber, daß auf jedem Weg von der Quelle zur Senke wenigstens eine Kante gesättigt ist. Folglich ist der Fluß blockierend. Die Laufzeit zur Bestimmung eines blockierenden Flusses beträgt $\mathcal{O}(n^2)$, d.h. die Gesamtlaufzeit des KARZANOV-Algorithmus ist damit $\mathcal{O}(n^3)$.

Eine Kombination des DINIC und KARZANOV-Algorithmus geht auf CHERKASKI [Che77] zurück. Er faßt mehrere Schichten zu einer *Superschicht* zusammen und innerhalb einer solchen Superschicht wird der DINIC-Algorithmus angewendet. Auf das aus den Superschichten bestehende Gesamtnetzwerk wird der Algorithmus von KARZANOV angewendet. Durch diese hybride Technik ergibt sich ein $\mathcal{O}(n^2\sqrt{m})$ Algorithmus. Durch eine geschickte Implementation dieses Verfahrens mittels spezieller Datenstrukturen erzielt GALIL [Gal80] eine Laufzeit von $\mathcal{O}(n^{\frac{5}{3}}m^{\frac{2}{3}})$. Mit Ausnahme von sehr dichten Graphen ($m = \mathcal{O}(n^2)$) war dies eine Verbesserung der bis dahin entwickelten Verfahren zur Lösung des Maxflow-Problemes.

2.4 Der MKM-Algorithmus

Der Algorithmus von MALHOTRA, KUMAR & MAHESHWARI [MKM78] arbeitet ebenfalls in Phasen auf geschichteten Netzwerken und mit blockierenden Flüssen. In jedem Schritt wird versucht, eine möglichst große Menge durch das Netzwerk fließen zu lassen. Zu diesem Zweck wird für einen Knoten $v \in V$ und einen bereits bestehenden Fluß f das Flußpotential $p_f(v)$ bestimmt

$$p_f(v) := \begin{cases} \min\{\sum\limits_{(v,w)\in E} \tau(v,w) - f(v,w), \sum\limits_{(w,v)\in E} \tau(w,v) - f(w,v)\}, & \text{falls } v \neq q, v \neq s, \\ \sum\limits_{(q,i)\in E} \tau(q,i) - f(q,i), & \text{falls } v = q, \\ \sum\limits_{(i,s)\in E} \tau(i,s) - f(i,s), & \text{falls } v = s. \end{cases}$$

Das Flußpotential kann als Restkapazität eines Knotens betrachtet werden, also die Menge, um die der Fluß durch diesen Knoten noch vergrößert werden kann. Man kann den bestehenden Fluß vergrößern, wenn man

[2] q hat „unendlichen“ Überschuß und versucht diesen abzugeben. Dabei werden alle ausgehenden Kanten gesättigt und der Knoten folglich blockiert.

das Flußpotential des *Referenzknotens*, d.h. des Knotens mit minimalem Flußpotential, zum bestehenden Fluß hinzuaddiert. Dazu wird ausgehend vom Referenzknoten das Flußpotential in Richtung der Senke oder in Richtung der Quelle bewegt. Da ein Schichtennetzwerk vorliegt, läßt sich dies einfach ausführen: Jede Ausgangskante eines Knotens führt näher zur Senke; jede Eingangskante führt näher zur Quelle. Der Referenzknoten und alle inzidenten Kanten können nun aus dem Netzwerk gelöscht werden. Da in jedem Schritt innerhalb einer Phase wenigstens ein Knoten gesättigt wird, kann eine Phase maximal aus $\mathcal{O}(n)$ Schritten bestehen. Somit ist die Laufzeit einer Phase und damit die Konstruktion eines blockierenden Flusses ebenfalls $\mathcal{O}(n^2)$. Eine Phase endet, wenn die Quelle oder die Senke aus dem Hilfsnetzwerk gelöscht wird. Wie auch beim DINIC-Algorithmus sind höchstens $n-1$ Phasen notwendig; die Gesamtlaufzeit beträgt demnach wie beim KARZANOV-Algorithmus $\mathcal{O}(n^3)$.

2.5 Der Algorithmus von Galil und Namaad

Obwohl der DINIC-Algorithmus im Vergleich zum modifizierten Markierungsalgorithmus implizit Zusatzinformationen speichert, nämlich eine untere Schranke für die Länge eines SAP's, geht dennoch viel Information verloren. Wird in einer Phase in einem Schichtennetzwerk ein Weg von q nach s gefunden, so wird hierdurch wenigstens eine Kante (i, j) gesättigt und entsprechend gelöscht bzw. als nicht länger benutzbar markiert. Die Wegstücke $q \to \cdots \to i$ und $j \to \cdots \to s$ werden jedoch wieder vergessen. Bei einer späteren Suche können diese Wege (oder Teile davon) gegebenenfalls wiederentdeckt werden. Die dafür notwendige Zeit kann eingespart werden, wenn diese Wege gespeichert werden. Genau diesen Ansatz verfolgt der Algorithmus von GALIL & NAMAAD [GN80]. Zur Speicherung der Wegstücke verwenden sie 2-3-Bäume. Die Laufzeit ihres Verfahrens benötigt $\mathcal{O}(nm \log^2 n)$ Zeit.

2.6 Der Wellen-Algorithmus

Der Wellen-Algorithmus von TARJAN [Tar84] ist ein weiterer Algorithmus zur Berechnung blockierender Flüsse. Es ist eine leicht modifizierte Variante des KARZANOV-Algorithmus aus Abschnitt 2.3. Die Anwendung des Algorithmus ist nicht auf Schichtennetzwerke beschränkt, sondern er kann auf beliebige azyklische Netzwerke angewendet werden. Der Algorithmus arbeitet ebenfalls mit Preflows und unterscheidet zwischen *blockierten* und *freien* Knoten. Der wesentliche Unterschied zum KARZANOV-Algorithmus besteht in der Ausführungsreihenfolge der Behandlung unbalancierter Knoten. Beim Wellen-Algorithmus werden die Knoten des Netzwerkes mit einer topologischen Sortierung versehen, und die Bearbeitungsreihenfolge erfolgt gemäß dieser Sortierung.

Der Algorithmus führt abwechselnd „Wellen" von `Increase`- und `Decrease`-Schritten aus. In jeder Welle werden alle Knoten $\neq q, s$ behandelt. Beim `Increase`-Schritt wird versucht, den Fluß in Richtung Senke zu erhöhen; beim `Decrease`-Schritt wird zum Zwecke der Balancierung der Zufluß reduziert. Zu jedem Zeitpunkt des Algorithmus gilt: Ist ein Knoten v blockiert, so gibt es auf jedem Weg von v zur Senke wenigstens eine gesättigte Kante. Da die Quelle während der Initialisierung blockiert wird, ist der Fluß zum Zeitpunkt der Terminierung blockierend.

Der `Increase`-Schritt verändert den Ausgangsfluß eines Knotens v solange, bis entweder der Überschuß abgebaut ist, der Knoten also balanciert ist, oder kein zusätzlicher Fluß zu einem freien Knoten befördert werden kann. Im zweiten Fall wird v blockiert. Nach Ausführung einer `Increase`-Welle gibt es somit keine unbalancierten, freien Knoten, und jede `Increase`-Welle blockiert wenigstens einen Knoten[3]. Die `Decrease`-Operation balanciert jeden unbalancierten Knoten durch Reduzierung des Zuflusses. Dabei werden gegebenenfalls Knoten in der vorhergehenden Schicht unbalanciert. Zu Beginn von `Decrease` sind jedoch die einzigen unbalancierten Knoten diejenigen, die in der vorhergehenden `Increase`-Welle blockiert wurden.

Es kann höchstens $n-1$ Iterationen der `Increase`- und `Decrease`-Wellen geben, da ein einmal blockierter Knoten stets blockiert bleibt. Folglich gibt es höchstens $(n-1)(n-2)$ Balancierungsversuche. Eine topologische

[3] mit Ausnahme einer zur Terminierung führenden

Sortierung kann in $\mathcal{O}(n+m)$ bestimmt werden (siehe [Kah62]). Durch die Bearbeitung der Knoten in einer `Decrease`-Welle gemäß der umgekehrten topologischen Sortierung, bleibt ein balancierter Knoten bis nach der anschließenden `Increase`-Welle balanciert. Jeder `Increase`-Schritt sättigt eine Kante oder beendet den Balancierungsversuch eines Knotens. Ebenso wird durch einen `Decrease`-Schritt der Fluß über eine Kante zu 0 reduziert, oder aber der Balancierungsversuch des Knotens wird beendet. Folglich ist die Gesamtzahl der `Increase`- und `Decrease`-Schritte durch $\mathcal{O}(2m+n^2)$ beschränkt. Wählt man geeignete Datenstrukturen, z.B. Adjazenz- oder Inzidenzlisten für eingehende und ausgehende Kanten mit direktem Zugriff, so ist die Gesamtlaufzeit $\mathcal{O}(n^2)$ des Wellen-Algorithmus (einschließlich der topologischen Sortierung) gewährleistet. Bettet man den Algorithmus in den DINIC-Algorithmus ein, so liefert dies einen weiteren $\mathcal{O}(n^3)$ Algorithmus für das Maxflow-Problem.

2.7 Algorithmus von Shiloach & Vishkin

Den ersten parallelen Algorithmus für das Maxflow-Problem haben SHILOACH & VISHKIN [SV82] veröffentlicht. Der Algorithmus arbeitet ebenfalls nach dem Grundschema von DINIC (Abschnitt 2.2) und für die Berechnung blockierender Flüsse verwenden sie ähnliche Techniken, wie sie von KARZANOV (Abschnitt 2.3) eingeführt wurden.

Der Algorithmus arbeitet in Phasen. In der ersten Phase werden alle von q ausgehenden Kanten gesättigt. In den folgenden Phasen werden jeweils alle unbalancierten Knoten behandelt. Jeder Knoten versucht durch Weitergabe an nachfolgende Knoten ein Gleichgewicht zu erreichen. Gelingt dies nicht, so wird der verbliebene Überschuß an Vorgänger zurückgegeben. Um die Rückgabe sehr einfach durchführen zu können, gibt es wie beim KARZANOV-Algorithmus für jeden Knoten einen Stack, in dem der Empfang von Fluß protokolliert wird.

Beim parallelen Algorithmus werden alle zu Beginn einer Phase unbalancierten Knoten gleichzeitig behandelt. In der sequentiellen Variante wird die Bearbeitungsreihenfolge durch eine Queue festgelegt. Die Laufzeit des sequentiellen Algorithmus beträgt $\mathcal{O}(n^2)$. Dies führt zu einem weiteren $\mathcal{O}(n^3)$ Algorithmus für das Maxflow-Problem.

3 Preflow-Push-Algorithmen

Die Suche nach parallelen Algorithmen für das Maxflow-Problem war ein wesentlicher Motivationsgrund für die Untersuchung anderer Ansätze zur Lösung des Problems. Die dadurch entstandenen Preflow-Push-Algorithmen gehen auf GOLDBERG & TARJAN [GT88] zurück und basieren auf ähnlichen Ideen wie denen von SHILOACH & VISHKIN. Die aufwendige Verwaltung der geschichteten Hilfsnetzwerke wird dabei jedoch vermieden. Stattdessen werden Abstandsmarkierungen verwendet. Eine Abbildung $d : V \to \mathbf{N}$ heißt *(zulässige) Abstandsmarkierung*, falls gilt: $d(q) = n$, $d(s) = 0$ und für alle Kanten (v, w) des Restnetzwerkes N_f ist $d(v) \leq d(w) + 1$. d dient im Algorithmus als Schranke für die Abschätzung der Entfernung eines Knotens zur Senke bzw. zur Quelle und macht dadurch die Verwendung eines Schichtennetzwerkes überflüssig. Ist $d(v) < n$, so ist $d(v)$ eine untere Schranke für den Abstand von v zur Senke. Ist $d(v) \geq n$, so dient $d(v) - n$ als untere Schranke für den Abstand von der Quelle zu v. In diesem Fall gibt es dann im Restnetzwerk keinen Weg von v zur Senke.

3.1 Generischer Algorithmus

Initial werden die zu q inzidenten Kanten mit voller Kapazität belegt. In jedem Schritt des Algorithmus (ausgenommen die Zeitpunkte der Initialisierung und der Terminierung) gibt es wenigstens einen Knoten mit positivem Überschuß. In jedem Schritt wird ein solcher *aktiver Knoten* ausgewählt und es wird versucht, den vorhandenen Überschuß auf Knoten zu verteilen, die einen geringeren Abstand zur Senke aufweisen. Zur Verteilung des

Überschusses auf andere Knoten werden nur *benutzbare Kanten* verwendet. Dies sind Kanten (i, j) des Restnetzwerkes, für die $d(i) = d(j) + 1$ gilt. Falls der Überschuß nicht an Knoten mit geringerem d-Wert abgegeben werden kann, wird die Markierung erhöht, so daß wenigstens eine neue benutzbare Kante entsteht.

Der generische Algorithmus führt in einer beliebigen Reihenfolge die zwei Basisoperationen `Push` und `Relabel` aus. Diese arbeiten folgendermaßen: Eine `Push`-Operation von v nach w erhöht den Fluß $f(v, w)$ der benutzbaren Kante (v, w), verringert $f(w, v)$ und verändert die Überschüsse $e_f(v)$ und $e_f(w)$ entsprechend. Die `Push`-Operation von v nach w heißt *sättigend*, falls $r_f(v, w) = 0$ nach ihrer Ausführung gilt. Andernfalls heißt sie *nichtsättigend*. Die `Relabel`-Operation ändert den Wert der Abstandsmarkierung gerade so, daß wenigstens eine Kante benutzbar wird. Zu Beginn ist eine initiale Abstandsmarkierung zu wählen. Eine sehr einfache Wahl ist $d(q) = n$ und $d(v) = 0$ für alle übrigen Knoten $v \in V \setminus \{q\}$. Durch eine rückwärts ausgeführte Breitensuche von s aus, kann in $\mathcal{O}(n+m)$ Zeit eine exakte Abstandsmarkierung bestimmt werden, d.h. für jeden Knoten $v \in V \setminus \{q\}$ ist $d(v)$ der Abstand von v nach s. Dies hat keine Auswirkung auf die Komplexität, erweist sich bei einer praktischen Implementation aber als günstig. Durch die Wahl des initialen Preflows und durch die Festsetzung $d(q) = n$ ist gewährleistet, daß nach der Initialisierungsphase von q kein weiterer Fluß ausgeht. In dieser generischen Fassung ist die Anzahl der `Relabel`-Operationen auf $2n^2$ beschränkt, es gibt höchstens nm sättigende `Push`-Operationen und $\mathcal{O}(n^2m)$ nichtsättigende `Push`-Operationen. Durch Auswahl geeigneter Datenstrukturen, die einen schnellen Zugriff auf aktive Knoten und benutzbare Kanten ermöglichen, kann der Preflow-Push-Algorithmus mit einem Laufzeitverhalten von $\mathcal{O}(n^2m)$ implementiert werden. Noch unklar ist jedoch die Korrektheit des Algorithmus, d.h. ist zum Zeitpunkt der Terminierung der dann anliegende Preflow ein maximaler Fluß?

Wenn der Algorithmus terminiert, so gibt es keine aktiven Knoten mehr (denn der Algorithmus läuft in einer Schleife über die aktiven Knoten). Das heißt aber: Der Preflow erfüllt die Konservativitätsbedingung, d.h. es ist ein Fluß. Da die Abstandsmarkierung d während des gesamten Algorithmus zulässig ist, kann es keinen erweiternden Weg bzgl. des dann anliegenden Flusses f in N geben, folglich ist f dann maximal.

3.2 Spezielle Implementationen

Die Laufzeit des Algorithmus ist von der Ausführungsreihenfolge der `Push`- und `Relabel`-Operationen sowie von Implementationsdetails abhängig. Geschieht die Auswahl der aktiven Knoten und der benutzbaren Kanten für eine `Push`-Operation in konstanter Zeit, so ist die Laufzeit von $\mathcal{O}(n^2m)$ gewährleistet. Es geht jedoch besser.

FIFO-Preflow-Push-Algorithmus

Der FIFO-Preflow-Push-Algorithmus speichert die unbalancierten Knoten in einer Queue. Es wird ein Knoten vom Kopf der Queue entfernt und die Push/Relabel-Operation auf ihn angewendet. Diese führt solange die `Push`-Operation auf dem Knoten aus, bis er entweder balanciert ist oder aber sein Überschuß nicht abgebaut werden kann. Dann wird der d-Wert durch die `Relabel`-Operation erhöht. Der Algorithmus terminiert, wenn die Queue leer ist. GOLDBERG & TARJAN [GT88] haben gezeigt, daß die Anzahl der nichtsättigenden `Push`-Operationen höchstens $4n^3$ ist. Damit hat der Algorithmus eine Gesamtlaufzeit von $\mathcal{O}(n^3)$.

Highest-Distance-Algorithmus

Bei der Highest-Distance-Implementation wird der Knoten mit maximalem d-Wert als nächster Knoten ausgewählt. Um den Zugriff in konstanter Zeit auf diese Knoten zu gewährleisten, wird die folgende Datenstruktur verwendet: In einem Array `List` wird unter dem Index k die Menge aller Knoten v abgelegt, für die $d(v) = k$ gilt. Durch Speicherung in einer doppelt verketteten Liste kann Einfügen und Löschen in konstanter Zeit erfolgen. Ferner gibt es einen Index `level`, der der größte Index k ist, so daß `List[k]` nichtleer ist. In jedem Iterationsschritt wählt der Algorithmus einen aktiven Knoten aus `List[level]`, entfernt diesen und versucht den Überschuß vollständig abzubauen. Entweder ist der Knoten danach balanciert oder aber sein d-Wert wurde erhöht. Dieses Verfahren haben ebenfalls GOLDBERG & TARJAN entwickelt, und CHERIYAN & MAHESHWARI [CM89] haben für diesen Algorithmus eine Laufzeit von $\mathcal{O}(n^2\sqrt{m})$ nachgewiesen.

Die angegebenen Schranken für die FIFO- und die Highest-Distance-Methode sind scharf, d.h. man kann Netzwerke konstruieren, so daß zur Bestimmung eines maximalen Flusses $\Theta(n^3)$ bzw. $\Theta(n^2\sqrt{m})$ `Push`-Operationen notwendig sind (in [CM89] sind parametrisierte Worst-Case-Netzwerke angegeben). Ein verbessertes Laufzeitverhalten der FIFO-Methode von $\mathcal{O}(mn \log \frac{n^2}{m})$ konnten GOLDBERG & TARJAN durch die Verwendung einer dynamischen Baumstruktur erzielen. Hierbei wird nicht versucht die Anzahl der `Push`-Operationen zu reduzieren, sondern vielmehr die Zeit zu verringern, die für die `Push`-Operationen aufgewendet werden muß. Allerdings ist der praktische Nutzen dieses Ergebnisses fraglich, da, wie bereits erwähnt, die Behandlung der Datenstruktur einen sehr großen Verwaltungsaufwand notwendig macht (siehe auch Abschnitt 4).

3.3 Skalierungstechniken

Der kritische Zeitfaktor bei den vorgestellten Preflow-Push-Methoden ist die Zahl der nichtsättigenden `Push`-Operationen. AHUJA & ORLIN [AO89] haben eine Variante des Preflow-Push-Algorithmus entwickelt, bei der sie eine Skalierungstechnik zur Reduzierung der Anzahl nichtsättigender `Push`-Operationen nutzen.

Die grundlegende Idee bei ihrem Algorithmus ist es stattdessen, den Überschuß nur von den Knoten aus zu bewegen, bei denen der Überschuß eine bestimmte Größe überschreitet. Weiterhin werden nur solche Knoten mit zusätzlichen Einheiten beliefert, bei denen der dann entstehende Überschuß nicht zu groß wird. Einerseits ist hierdurch gewährleistet, daß bei nichtsättigenden `Push`-Operationen eine relativ große Menge in Richtung der Senke befördert wird. Andererseits wird aber auch vermieden, daß der Überschuß in einem Knoten zu stark ansteigt, der dann vermutlich nicht zur Senke, sondern zurück zur Quelle transportiert werden müsste.

Der Excess-Scaling-Algorithmus

Der Excess-Scaling-Algorithmus führt eine Schleife mit $K = \lceil \log U \rceil + 1$ Skalierungsphasen aus, wobei jeweils ein unterschiedlicher Skalierungsparameter Δ verwendet wird. In jeder Phase werden Knoten mit großem Überschuß ($e_f(v) \geq \Delta/2$) ausgewählt, und die `Push`-Operation wird auf sie (ggf. mehrfach) angewendet. Gibt es mehrere Knoten mit großem Überschuß, wird derjenige mit minimalem d-Wert ausgewählt. Der Push/Relabel-Schritt entspricht bis auf eine Ausnahme dem im generischen Preflow-Push-Algorithmus verwendeten. Anstatt $\delta = \min(e_f(v), \tau_f(v,w))$ Einheiten zu bewegen, werden $\delta = \min(e_f(v), \tau_f(v,w), \Delta - e_f(w))$ Einheiten verschoben. Hierdurch ist einerseits garantiert, daß ein nichtsättigender `Push` wenigstens $\Delta/2$ Einheiten bewegt, und andererseits gilt für den empfangenden Knoten w $e_f(w) \leq \Delta$. Gibt es keinen Knoten mit großem Überschuß, so wird der Skalierungsparameter Δ halbiert und eine neue Skalierungsphase beginnt.

Zum schnellen Zugriff auf aktive Knoten wird eine ähnliche Struktur wie beim Highest-Distance-Algorithmus verwendet. Das Array `List` enthält unter dem Index k alle Knoten v, für die $e_f(v) > \Delta/2$ und $d(v) = k$ gilt. Die Variable `level` ist der kleinste Index k, so daß `List[k]` nichtleer ist. Die Laufzeit des Excess-Scaling-Algorithmus ist $\mathcal{O}(nm + n^2 \log U)$. Unter der Voraussetzung $U = n^{\mathcal{O}(1)}$ ist der Algorithmus eine Verbesserung der bisherigen Verfahren.

Stack-Scaling und Wave-Scaling

Zwei Varianten des Excess-Scaling sind das Stack-Scaling und das Wave-Scaling, die beide von AHUJA, ORLIN & TARJAN [AOT89] entwickelt wurden.

Beim Stack-Scaling wird der Skalierungsparameter Δ beim Übergang von einer Phase zur nächsten um einen Faktor $k \geq 2$ reduziert. Wie auch beim Excess-Scaling wird die `Push`-Operation jeweils auf Knoten mit großem Überschuß angewendet. Bei mehreren Kandidaten wird nun aber der Knoten mit größtem d-Wert ausgewählt. Jeder nichtsättigende `Push` bewegt mindestens Δ/k Einheiten und in keinem Knoten übersteigt der Überschuß Δ. Die Bearbeitung der Knoten erfolgt mittels eines Stacks, in dem die Knoten mit großem Überschuß abgelegt werden. Bei der Wahl $k = \lceil \log U / \log \log U \rceil$ führt der Stack-Scaling-Algorithmus $\mathcal{O}(n^2 \log U / \log \log U)$ nichtsättigende `Push`-Operationen aus und hat eine Gesamtlaufzeit von $\mathcal{O}(nm + \frac{n^2 \log U}{\log \log U})$.

Der Wave-Scaling-Algorithmus verbindet das Stack-Scaling mit dem Wellen-Algorithmus (Abschnitt 2.6). Das Verfahren besteht wie zuvor auch aus mehreren Phasen und verwendet einen zusätzlichen Kontrollparameter l. Ist der totale Überschuß $e_f(V) \geq \frac{n\Delta}{l}$, so wird versucht, den Überschuß in allen aktiven Knoten — in steigender Reihenfolge ihrer d-Werte — durch Anwendung der `Push`-Operation zu eliminieren. Dieser Versuch endet, wenn entweder der Überschuß zu Null reduziert wird oder aber ein `Relabel` erfolgt. Die Ausführung der `Push`-Operationen erfolgt wie beim Stack-Scaling. Dieser Schritt endet, wenn $e_f(V) \leq \frac{n\Delta}{l}$ gilt. Dann wird der Excess-Scaling-Algorithmus angewendet. Bei Wahl von $l = \sqrt{\log U}$ werden $\mathcal{O}(n^2\sqrt{\log U})$ nichtsättigende Push-Operationen ausgeführt und die Gesamtlaufzeit des Verfahrens ist dann $\mathcal{O}(nm + n^2\sqrt{\log U})$. Durch den Einsatz von dynamischen Bäumen kann eine Laufzeit von $\mathcal{O}(nm \log(\frac{n}{m}\sqrt{\log U} + 2))$ nachgewiesen werden.

3.4 Probabilistische Varianten

Eine probabilistische Variante des Preflow-Push-Algorithmus haben CHERIYAN & HAGERUP [CH89] vorgeschlagen. Dabei verwenden sie die dynamische Baumstrukur, Fibonacci-Heaps (vgl. [FT87]) sowie Ideen des Excess-Scaling-Algorithmus. Im Gegensatz zu den übrigen Implementationen des Preflow-Push-Verfahrens gehen sie jedoch nicht von einer festen Reihenfolge innerhalb der Adjazenzlisten der Knoten aus, sondern zu Beginn des Algorithmus und mit jedem `Relabel`-Schritt werden diese Listen zufällig permutiert. Die probabilistische Komponente des Algorithmus besteht in der Bestimmung der aktuellen Kante eines Knotens. Grundlage ihrer Laufzeitabschätzung bildet ein 2-Personen-Spiel auf einem bipartiten Graphen, dessen Spielverhalten so konstruiert wird, daß es das Verhalten des Algorithmus simuliert. Für jede Konstante $\alpha > 0$ löst der Algorithmus das Maxflow-Problem mit mindestens der Wahrscheinlichkeit $1 - n^{-\alpha n^2}$ in $\mathcal{O}(nm + n^2(\log n)^3)$ Zeit. ALON [Alo90] hat die probabilistischen Komponenten des Algorithmus eliminiert und für diese deterministische Fassung eine Laufzeit von $\mathcal{O}(nm + n^{\frac{8}{3}} \log n)$ nachgewiesen. CHERIYAN, HAGERUP & MEHLHORN [CHM90] haben eine modifizierte deterministische Version mit Laufzeit $\mathcal{O}(n^3/\log n)$ sowie für den randomisierten Algorithmus von CHERIYAN & HAGERUP eine Laufzeit von $\mathcal{O}(nm + n^2(\log n)^2)$ nachgewiesen. Durch ein modifiziertes Spielverhalten in dem 2-Personen-Spiel haben KING, RAO & TARJAN [KRT92] eine deterministische Fassung des Maxflow-Algorithmus von CHERIYAN & HAGERUP entwickelt, die eine Laufzeit von $\mathcal{O}(mn + n^{2+\epsilon})$ für eine beliebige Konstante ϵ besitzt. Für $m > n^{1+\epsilon}$ ist dies somit ein $\mathcal{O}(nm)$ Algorithmus für das Maxflow-Problem. Eine weitere Verbesserung der Spielstrategie haben PHILIPS & WESTBROOK [PW93] entwickelt. Ihr Ergebnis führt auf einen Algorithmus mit Laufzeit $\mathcal{O}(nm \log_{\frac{m}{n}} n + n(\log n)^{2+\epsilon})$.

Es bleibt zu zeigen, ob die beschriebenen theoretischen Laufzeitverbesserungen auch in effiziente, praktisch anwendbare Implementationen umsetzbar sind. Die angeführten Laufzeiten nähern sich zwar für eine große Klasse von Graphen der $\mathcal{O}(nm)$-Zeitschranke, es bleibt jedoch weiterhin ein offenes Problem, ob ein Algorithmus mit diesem Laufzeitverhalten für beliebige Netzwerke existiert.

4 Laufzeitvergleiche

In der unten aufgeführten Tabelle 1 sind die Laufzeitverhalten einiger Algorithmen zur Lösung des Netzwerkfluß-Problems aufgelistet. Für die ersten sieben Algorithmen dieser Tabelle hat GALIL [Gal81, GN80] eine Klasse von Netzwerken angegeben, für die diese Verfahren ihr Worst-Case-Laufzeitverhalten annehmen. Für das Highest-

Jahr	Algorithmus	Laufzeit	Referenz
1970	DINIC	$\mathcal{O}(n^2 m)$	[Din70]
1972	EDMONDS & KARP	$\mathcal{O}(nm^2)$	[EK72]
1974	KARZANOV	$\mathcal{O}(n^3)$	[Kar74]
1977	CHERKASKI	$\mathcal{O}(n^2 \sqrt{m})$	[Che77]
1978	MALHOTRA, KUMAR & MAHESHWARI	$\mathcal{O}(n^3)$	[MKM78]
1980	GALIL	$\mathcal{O}(n^{\frac{5}{3}} m^{\frac{2}{3}})$	[Gal80]
1980	GALIL & NAMAAD	$\mathcal{O}(nm \log^2 n)$	[GN80]
1982	SHILOACH & VISHKIN	$\mathcal{O}(n^3)$	[SV82]
1983	SLEATOR & TARJAN	$\mathcal{O}(nm \log n)$	[ST83]
1984	TARJAN	$\mathcal{O}(n^3)$	[Tar84]
1985	GABOW	$\mathcal{O}(nm \log U)$	[Gab85]
1988	GOLDBERG & TARJAN	$\mathcal{O}(n^3)$	[GT88]
1988		$\mathcal{O}(nm \log(\frac{n^2}{m}))$	[GT88]
1989	CHERIYAN & MAHESHWARI	$\mathcal{O}(n^2 \sqrt{m})$	[CM89]
1989	AHUJA & ORLIN	$\mathcal{O}(nm + n^2 \log U)$	[AO89]
1989	AHUJA, ORLIN & TARJAN	$\mathcal{O}(nm + \frac{n^2 \log U}{\log \log U})$	[AOT89]
1989		$\mathcal{O}(nm + n^2 \sqrt{\log U})$	[AOT89]
1989		$\mathcal{O}(nm \log(\frac{n}{m} \sqrt{\log U} + 2))$	[AOT89]
1989	CHERIYAN & HAGERUP	$\mathcal{O}(mn + n^2 (\log n)^3)$	[CH89]
1990	ALON	$\mathcal{O}(nm + n^{\frac{8}{3}} \log n)$	[Alo90]
1990	CHERIYAN, HAGERUP & MEHLHORN	$\mathcal{O}(n^3 / \log n)$	[CHM90]
1991	GOLDBERG, GRIGORIADIS & TARJAN	$\mathcal{O}(nm \log n)$	[GGT90]
1992	KING, RAO & TARJAN	$\mathcal{O}(mn + n^{2+\epsilon})$	[KRT92]
1993	PHILIPS & WESTBROOK	$\mathcal{O}(mn \log_{\frac{m}{n}} n + n \log^{2+\epsilon} n)$	[PW93]

n: Knotenzahl, m: Kantenzahl, U: maximale Kantenkapazität

Tabelle 1: Sequentielle Maxflow-Algorithmen

Distance- und die FIFO-Preflow-Push Methode haben CHERIYAN & MAHESHWARI [CM89] Netzwerke angegeben, bei denen die Algorithmen ihr Worst-Case-Verhalten annehmen.

In einer Reihe von Arbeiten sind Laufzeitvergleiche anhand praktisch durchgeführter Tests veröffentlicht worden. HAMACHER [Ham79] hat die Überlegenheit der Methode von KARZANOV gegenüber dem Algorithmus von EDMONDS & KARP festgestellt. CHEUNG [Che80] hat in seiner Arbeit 8 Algorithmen (u.a. EDMONDS & KARP, DINIC, KARZANOV) miteinander verglichen und dabei Testnetzwerke mit bis zu 1500 Knoten und 7960 Kanten betrachtet. Der Algorithmus von DINIC hat sich dabei, obwohl in seiner Komplexität unterlegen, als effizienteste Methode herausgestellt. Ein ähnliches Ergebnis präsentiert Imai [Ima83]. Er führt seine Untersuchungen anhand mehrerer Klassen von Netzwerken durch und gelangt zu dem Ergebnis, daß der DINIC-Algorithmus und die Methode von KARZANOV ebenbürtig und den übrigen Methoden (EDMONDS & KARP, MKM, GALIL & NAMAAD) überlegen sind. DERIGS & MEIER [DM89] haben Untersuchungen für mehrere Varianten des Preflow-Push-Algorithmus durchgeführt und diese mit verschiedenen Implementationen der Algorithmen von DINIC und KARZANOV verglichen. Interessanterweise stellen sich verschiedene Verfahren bei unterschiedlichen Netzwerkklassen als überlegen heraus. Die Verwendung einer globalen Variante des `Relabel`-Schrittes führt zu einer Reduzierung der notwendigen `Relabel`-Operationen und somit zu einer deutlichen Laufzeitverbesserung (Faktor 3-14). Gegenüber den Algorithmen von DINIC und KARZANOV erweisen sich die Preflow-Push-Methoden als deutlich überlegen (Faktor 2-10). Die Implementation des DINIC-Algorithmus unter Verwendung dynamischer Bäume hat sich bei diesem Test als das langsamste Verfahren herausgestellt, was auf den enormen Aufwand, der zur Verwaltung der komplexen Datenstruktur notwendig ist, zurückzuführen ist. In [AKMO95] werden drei Preflow-Push-Varianten (FIFO, Highest-Distance, Lowest-Distance) mit den Scaling-Algorithmen (Excess-Scaling, Stack-Scaling, Wave-Scaling) und verschiedenen AP-Algorithmen verglichen. Als Testobjekte dienen dabei Netzwerke mit bis zu 10000 Knoten unterschiedlicher Klassen (Random-Netzwerke, Schichtennetzwerke,

Jahr	Algorithmus	Laufzeit	Prozessoren	Referenz
1982	SHILOACH & VISHKIN	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n)$	[SV82]
1988	GOLDBERG & TARJAN	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n)$	[GT88]
1989	CHERIYAN & MAHESHWARI	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	[CM89]
1989	AHUJA & ORLIN	$\mathcal{O}(n^2 \log U \log p)$	$p = \lceil \frac{m}{n} \rceil$	[AO89]
1991	GOLDBERG	$\mathcal{O}(n^2 \log(\frac{2m}{n} + p)(\frac{\sqrt{m}}{p}))$	$p = \mathcal{O}(\sqrt{m})$	[Gol91]
1991		$\mathcal{O}(n^2 \log n(\frac{\sqrt{m}}{p}))$	$p = \mathcal{O}(\sqrt{m})$	[Gol91]

n: Knotenzahl, m: Kantenzahl, U: maximale Kantenkapazität

Tabelle 2: Parallele Maxflow-Algorithmen

Gitternetzwerke, Netzwerke verschiedenster Netzwerk-Generatoren u.a.). Das Highest-Distance-Verfahren stellt sich als das überlegene Verfahren heraus. Die empirische Laufzeit liegt bei $\mathcal{O}(n^{1.5})$. Die Scaling-Algorithmen sind, obwohl theoretisch überlegen, deutlich langsamer.

5 Parallele Algorithmen

Das Maxflow-Problem ist bekanntlich P-vollständig (siehe [GSS82] oder [GR88]). In der Literatur finden sich einige Arbeiten, in denen parallele Ansätze zur Lösung des Maxflow-Problemes untersucht werden. Die auf der Suche nach erweiternden Wegen beruhenden Verfahren enthalten inhärent sequentielle Komponenten und erlauben keine direkte parallele Implementation. Diese Tatsache war ein wesentlicher Motivationsgrund für die Entwicklung der Preflow-Push-Methoden.

Der erste parallele Algorithmus stammt von SHILOACH & VISHKIN [SV82] und basiert auf der Konstruktion blockierender Flüsse nach der Idee von KARZANOV [Kar74]. Das zugrundeliegende Maschinenmodell ist eine SP-RAM. Die Laufzeit beträgt $\mathcal{O}(n^3(\log n)/p)$, wobei $p \leq n$ die Prozessorenanzahl ist und $\mathcal{O}(n^2)$ Speicher benötigt wird. Basierend auf der Preflow-Push-Methode hat GOLDBERG zwei parallele Varianten der Highest-Distance-Methode vorgestellt (siehe [Gol91] oder [Gol93]). Als Maschinenmodell betrachtet er die CREW-PRAM. Eine Methode benötigt bei $p = \mathcal{O}(\sqrt{m})$ Prozessoren und $\mathcal{O}(m + n \log n)$ Platz eine Laufzeit von $\mathcal{O}(n^2 \log(\frac{2m}{n} + p)(\frac{\sqrt{m}}{p}))$. Die zweite Variante benötigt bei gleicher Prozessorenanzahl $\mathcal{O}(m + n)$ Platz und $\mathcal{O}(n^2 \log n(\frac{\sqrt{m}}{p}))$ Zeit. AHUJA & ORLIN [AO89] zeigen, daß für eine parallele Variante des Excess-Scaling-Algorithmus auf einer EREW-PRAM $\mathcal{O}(n^2 \log U \log p)$ Zeit bei $p = \lceil \frac{m}{n} \rceil$ Prozessoren benötigt werden. Eine Laufzeit von $\mathcal{O}(n^2)$ bei $\mathcal{O}(n)$ Prozessoren und $\mathcal{O}(n^2\sqrt{m})$ Nachrichten erreichen CHERIYAN & MAHESHWARI mit einer synchronen, verteilten Variante eines Preflow-Push-Algorithmus.

Literatur

[AKMO95] R.K. Ahuja, M. Kodialam, A.K. Mishra, and J.B. Orlin. Computational Investigations of Maximum Flow Algorithms. Technical report, Sloan School of Management, MIT, Cambridge, MA, 1995.

[Alo90] N. Alon. Generating Pseudo-Random Permutations and Maximum Flow Algorithms. *Information Processing Letters*, 35(4):201–204, 1990.

[AMO93] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1993.

[AO89] R.K. Ahuja and J.B. Orlin. A fast and simple Algorithm for the Maximum Flow Problem. *Operations Research*, 37(5):748–759, September-October 1989.

[AOT89] R.K. Ahuja, J.B. Orlin, and R.E. Tarjan. Improved time bounds for the Maximum Flow Problem. *SIAM Journal of Computing*, 18(5):939–954, October 1989.

[CH89] J. Cheriyan and T. Hagerup. A Randomized Maximum-Flow Algorithm. In *30th Annual Symposium on Foundations of Computer Science*, pages 118–123. IEEE Computer Society Press, 1989.

[Che77] B.V. Cherkaski. Algorithm of Construction of Maximal Flow in Networks with Complexity $\mathcal{O}(|V|^2\sqrt{|E|}$ Operations. *Math. Methods of Solutions of Economic Problems*, 7:117–125, 1977.

[Che80] T.-Y. Cheung. Computational Comparison of Eight Methods for the Maximum Network Flow Problems. *ACM Transactions on Mathematical Software*, 6(1):1–16, March 1980.

[CHM90] J. Cheriyan, T. Hagerup, and K. Mehlhorn. Can a Maximum Flow be Computed in $o(nm)$ Time? In M.S. Paterson, editor, *Automata, Languages, and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 235–248. Springer-Verlag, Berlin-Heidelberg-New York, July 1990.

[CM89] J. Cheriyan and S.N. Maheshwari. Analysis of Preflow Push Algorithms for Maximum Network Flow. *SIAM Journal of Computing*, 18(6):1057–1086, December 1989.

[Dan66] G.B. Dantzig. *Lineare Programmierung und Erweiterungen*. Springer-Verlag, Berlin-Heidelberg-New York, 1966.

[Din70] E.A. Dinic. Algorithm for Solution of a Problem of Maximal Flow in a Network with Power Estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970.

[DM89] U. Derigs and W. Meier. Implementing Goldberg's Max-Flow-Algorithm – A Computational Investigation. *Zeitschrift für Operations Research*, 33:383–403, 1989.

[EK72] J. Edmonds and R.M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19(2):248–264, 1972.

[FF56] L.R. Ford and D.R. Fulkerson. Maximal Flow through a Network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[FT87] M.L. Fredman and R.E. Tarjan. Fibonacci Heaps and their uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[Gab85] H.N. Gabow. Scaling Algorithms for Network Problems. *Journal of Computer and System Sciences*, 31:148–168, 1985.

[Gal80] Z. Galil. An $\mathcal{O}(V^{\frac{5}{3}}E^{\frac{2}{3}})$ Algorithm for the Maximal Flow Problem. *Acta Informatica*, 14:221–242, 1980.

[Gal81] Z. Galil. On the Theoretical Efficiency of various Network Flow Algorithms. *Theoretical Computer Science*, 14:103–111, 1981.

[GGT90] A.V. Goldberg, M.D. Grigoriadis, and R.E. Tarjan. Use of dynamic trees in a network simplex algorithm for the maximum flow problem. *Mathematical Programming*, 50(3):277–290, 1990.

[GH90] D. Goldfarb and J. Hao. A primal Simplex Algorithm that solves the Maximum Flow Problem in at most nm pivots and $\mathcal{O}(n^2m)$ time. *Mathematical Programming*, 47(3):353–365, 1990.

[GN80] Z. Galil and A. Namaad. An $\mathcal{O}(EV\log^2 V)$ Algorithm for the Maximal Flow Problem. *Journal of Computer and System Sciences*, 21(2):203–217, October 1980.

[Gol91] A.V. Goldberg. Processor-efficient implementation of a maximum flow algorithm. *Information Processing Letters*, 38(4):179–185, May 1991.

[Gol93] A.V. Goldberg. Parallel Algorithms for Network Flow Problems. In J.H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 19, pages 813–839. Morgan Kaufmann Publishers, 1993.

[GR88] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[Gri86] M.D. Grigoriadis. An efficient implementation of the network simplex algorithm. *Mathematical Programming Studies*, 26:83–111, March 1986.

[GSS82] L.M. Goldschlager, R.A. Shaw, and J. Staples. The Maximum Flow Problem is Log Space Complete for P. *Theoretical Computer Science*, 21:105–111, 1982.

[GT88] A.V. Goldberg and R.E. Tarjan. A new Approach to the Maximum-Flow Problem. *Journal of the ACM*, 35(4):921–940, October 1988.

[Ham79] H. Hamacher. Numerical Investigations on the Maximal Flow Algorithm of Karzanov. *Computing*, 22:17–29, 1979.

[Ima83] H. Imai. On the practical efficiency of various Maximum Flow Algorithms. *Journal of the Operations Research Society of Japan*, 26(1):61–81, March 1983.

[Kah62] A.B. Kahn. Topological Sorting of Large Networks. *Communications of the ACM*, 5(11):558–562, November 1962.

[Kar74] A.V. Karzanov. Determining the Maximal Flow in a Network with Method of Preflows. *Soviet Mathematics Doklady*, 15:434–437, 1974.

[Kar84] N.K. Karmarkar. A new Polynomial–Time Algorithm for Linear Programming. *Combinatorica*, 4(1):373–395, 1984.

[Kha79] L.G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20(1):191–194, 1979.

[KRT92] V. King, S. Rao, and R.E. Tarjan. A Faster Deterministic Maximum Flow Algorithm. In *Proceedings of the third Annual ACM-SIAM Symposium on Discrete Algorithms*, Orlando,Florida, January 1992. ACM Press.

[MKM78] V.M. Malhotra, M.P. Kumar, and S.N. Maheshwari. An $\mathcal{O}(|V|^3)$ Algorithm for Finding Maximum Flows in Networks. *Information Processing Letters*, 7(6):277–278, October 1978.

[PW93] S. Philips and J. Westbrook. Online Load Balancing and Network Flow. In *25th Annual ACM Symposium on Theory of Computing*, pages 402–411. ACM Press, May 1993.

[ST83] D.D. Sleator and R.E. Tarjan. A Data Structure for Dynamic Trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.

[SV82] Y. Shiloach and U. Vishkin. An $\mathcal{O}(n^2 \log n)$ Parallel MAX-FLOW Algorithm. *Journal of Algorithms*, 3:128–146, 1982.

[Tar84] R.E. Tarjan. A Simple Version of Karzanov's Blocking Flow Algorithm. *Operations Research Letters*, 2(6):265–268, March 1984.

[Zwi95] U. Zwick. The smallest networks on which the Ford-Fulkerson maximum flow procedure may fail to terminate. *Theoretical Computer Science*, 148(1):165–170, August 1995.

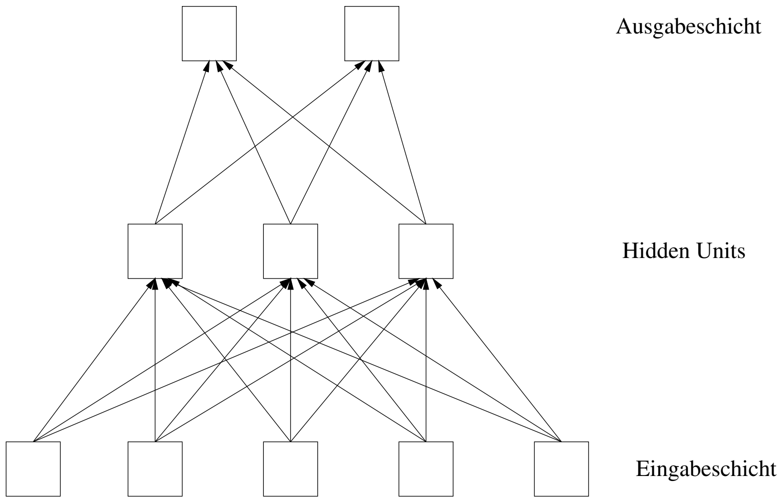
Neuronale Netze

Andreas Kemper
Universität GH Paderborn, linus@uni-paderborn.de

Wie bereits im letzten Beitrag dieser kleinen Reihe beschrieben, unterliegt das Perzeptron Modell einer ganzen Reihe von Einschränkungen bezüglich seiner Anwendbarkeit auf viele elementare Probleme. Diese Einschränkungen werden von mehrschichtigen Feed-Forward-Netzen überwunden.

Mehrschichtige Netze — Backpropagation

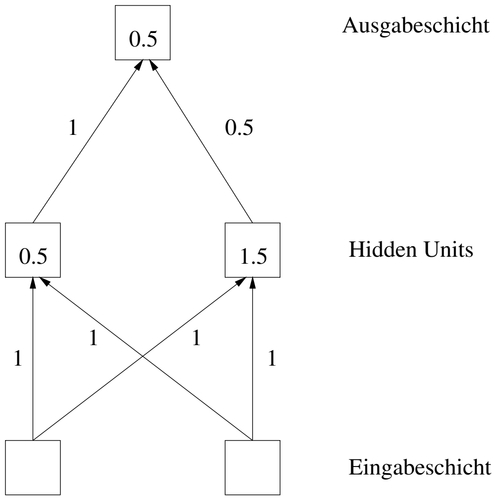
Die Zählweise der Schichten eines mehrschichtigen Netzes ist in der Literatur sehr unterschiedlich. Oft wird die Anzahl der Verbindungsschichten zwischen den Zellen gezählt. Im folgenden bestehe ein n - schichtiges Netz aus n Schichten von Zellen, von denen die erste die Eingabeschicht und die letzte die Ausgabeschicht darstellt. Die einzelnen Zell-Schichten sind durch $n - 1$ Schichten von Verbindungen und dazugehörenden Gewichten verbunden. Die Informationen wandern, mit den Eingabezellen beginnend, Schicht für Schicht durch das Netz, bis die Ausgabeschicht erreicht ist. Jede Zelle einer Schicht ist dabei mit jeder Zelle der folgenden Schicht verbunden. Die Zwischenschichten werden dabei als *Hidden Layer* bezeichnet.



Im Gegensatz zum einfachen Perzeptron kann mit Hilfe eines solchen mehrschichtigen Netzes das XOR Problem, ein Spezialfall des Parity Problems für zwei Elemente, gelöst werden. Das folgende Netz berechnet offensichtlich die XOR Funktion. Es besteht dabei aus binären (0/1) Schwellwertelementen. Die Schranken sind innerhalb der Zellen abgebildet. Die Verbindungen sind mit den Gewichten beschriftet.

Error Backpropagation

Gesucht wird nun eine Methode, die Gewichte innerhalb eines solchen Netzes zu bestimmen. Der Backpropagation Algorithmus stellt eine solche Methode da. Er arbeitet nach dem Prinzip des Überwachten Lernens, das heißt es



gibt eine Lernphase, während der dem Netz Paare aus Eingabemuster und zugehörigem Ausgabemuster präsentiert werden. Die Vorgehensweise des Algorithmus stellt dabei eine Verallgemeinerung der Delta-Regel dar. Die Grundidee besteht dabei aus einem Gradientenabstiegsverfahren.

Zur Erläuterung des Algorithmus wird im folgenden ein dreischichtiges Netzwerk wie in der obigen Abbildung betrachtet. Die Ausgabezellen werden mit O_i bezeichnet, die Zellen der Zwischenschicht mit V_j und die Eingabezellen mit ξ_k. Die w_{jk} bilden die Gewichte der Verbindungen der Eingabeschicht mit der Zwischenschicht, die W_{ij} die entsprechenden Gewichte zwischen dieser Schicht und den Eingabezellen. Außerdem gibt es wieder eine Menge von Eingabemustern ξ^μ, $\mu = 1, \ldots, p$ mit den zugehörigen erwarteten Ausgabemustern ζ^μ. Die ξ_k^μ können sowohl binär ($0/1$, oder $-1/+1$) wie auch reellwertig sein. Die Schranke wird als gleich 0 angenommen. Diese kann immer durch ein weiteres Neuron mit konstantem Ausgabesignal -1 und entsprechendem Gewicht simuliert werden. Die Ausgabefunktion g sollte differenzierbar sein.

Die Zelle j der Zwischenschicht summiert ihre Eingaben beim Anliegen des Musters μ an den Eingabezellen

$$h_j^\mu = \sum_k w_{jk}\xi_k^\mu \tag{2}$$

und erhält damit den Wert

$$V_j^\mu = g(h_j^\mu) = g(\sum_k w_{jk}\xi_k^\mu) \tag{3}$$

Eine Zelle i aus der Ausgabeschicht berechnet damit:

$$h_i^\mu = \sum_j W_{ij}V_j^\mu = \sum_j W_{ij}\, g(\sum_k w_{jk}\xi_k^\mu) \tag{4}$$

Damit ergibt sich der Wert der Zelle zu:

$$O_i^\mu = g(h_i^\mu) = g(\sum_j W_{ij}V_j^\mu) = g(\sum_j W_{ij}\, g(\sum_k w_{jk}\xi_k^\mu)) \tag{5}$$

Um ein Gradientenabstiegsverfahren durchführen zu können, muß zunächst eine Energie- oder Kostenfunktion definiert werden. Hierzu verwendet man die Summe über die Quadrate der Fehler in jeder Zelle der Ausgabeschicht.

$$E(w) = \frac{1}{2}\sum_{\mu i}(\zeta_i^\mu - O_i^\mu)^2 \tag{6}$$

Dieser Fehler ergibt sich dann im Falle des beschriebenen dreischichtigen Netzes zu:

$$E(w) = \frac{1}{2}\sum_{\mu i}(\zeta_i^\mu - g(\sum_j W_{ij} g(\sum_k w_{jk}\xi_k^\mu)))^2 \tag{7}$$

Nun kann gemäß der Regel des Gradientenabstiegs der Änderungswert der Gewichte zwischen der Zwischenschicht und den Ausgabezellen bestimmt werden:

$$\begin{aligned} \Delta W_{ij} &= -\eta \frac{\partial E}{\partial W_{ij}} = \eta \sum_\mu (\zeta_i^\mu - O_i^\mu) g'(h_i^\mu) V_j^\mu \\ &= \eta \sum_\mu \delta_i^\mu V_j^\mu \end{aligned} \tag{8}$$

mit

$$\delta_i^\mu = g'(h_i^\mu)(\zeta_i^\mu - O_i^\mu) \tag{9}$$

Dieses Ergebnis entspricht bis zu dieser Stelle dem Gradientenabstieg beim einfachen Perzeptron, wenn man die Zwischenschicht als Eingabeschicht interpretiert. Beim Backpropagation Algorithmus existieren jedoch weitere Schichten mit den zugehörigen Gewichte, die ebenfalls entsprechend verändert werden. Für die Gewichte der w_{ik}, die die Eingabeschicht mit der Zwischenschicht verbinden, ergibt sich mit Hilfe der Kettenregel die folgende Änderung:

$$\begin{aligned} \Delta w_{jk} = -\eta \frac{\partial E}{\partial w_{jk}} &= -\eta \sum_\mu \frac{\partial E}{\partial V_j^\mu} \frac{\partial V_j^\mu}{\partial w_{jk}} \\ &= \eta \sum_{\mu i} (\zeta_i^\mu - O_i^\mu) g'(h_i^\mu) W_{ij} g'(h_j^\mu) \xi_k^\mu \\ &= \eta \sum_{\mu i} \delta_i^\mu W_{ij} g'(h_j^\mu) \xi_k^\mu \\ &= \eta \sum_\mu \delta_j^\mu \xi_k^\mu \end{aligned} \tag{10}$$

Dabei ergibt sich δ_j^μ zu:

$$\delta_j^\mu = g'(h_j^\mu) \sum_i W_{ij} \delta_i^\mu \tag{11}$$

Dieses Verfahren läßt sich im Falle von Netzen mit mehr als drei Schichten beliebig fortführen. Dabei werden die δ kontinuierlich für jede Schicht des Netzes, mit der Ausgabeschicht beginnend und in Richtung der Eingabeschicht fortfahrend, berechnet.

Die δ jeder neuen Schicht berechnen sich dabei u.a. aus den δ Werten der zuvor berechneten. Die Regel zur Änderung der Gewichte hat dabei immer die Form

$$\Delta_{w_{pq}} = \eta \sum_{\mu} \delta_{output} v_{input} \tag{12}$$

wobei sich *input* und *output* auf die Schichten bezieht, die durch die Gewichte w_{pq} verbunden werden.

Der Fehler wird in Form der δ Werte von Schicht zu Schicht zurückgereicht, wodurch sich die Bezeichnung *Error Backpropagation* erklärt.

Obwohl in der obigen Beschreibung des Verfahrens meist mit der Summe über alle Eingabemuster gearbeitet worden ist, wird das Verfahren in der Praxis meist wiederholt für einzelne Muster der Trainingsmenge durchgeführt. Es wird also für jedes Muster die Differenz aus dem gewünschten und dem tatsächlichen Wert der Ausgabezellen ermittelt und dann der Fehler in Form der δ Werte zur Veränderung der Gewichte durch das Netz zurückgereicht. Die Muster werden dabei zumeist zufällig ausgewählt.

Ein Vorteil des Backpropagation Verfahrens liegt in der Möglichkeit, die Gewichtsveränderungen der Verbindungen zwischen zwei Schichten unabhängig voneinander also parallel berechnen zu können, wenn die δ Werte bekannt sind. Auch vermindert das Verfahren den Aufwand der Berechnung der Ableitungen erheblich.

Als Ausgabefunktion g ist eine differenzierbare Funktion erforderlich, daher sind binäre Schwellwertelemente nicht möglich. Meist wird eine sigmoide Ausgabefunktion verwendet. Zum Beispiel

$$g(h) = f_\beta(h) = \frac{1}{1 * \exp(-2\beta)} \tag{13}$$

oder

$$g(h) = \tanh(\beta h) \tag{14}$$

Im Gegensatz zum Perzeptron existiert für den Backpropagation Algorithmus kein Konvergenzsatz, der die Konvergenz des Verfahrens sicherstellt, wenn eine Lösung existiert. Ebenso wie in anderen Modellen (z.B. dem Hopfield Modell) besteht beim Backpropagation das Problem lokaler Minima. Die Kostenfunktion E ist i.a. eine sehr komplizierte Funktion der Gewichte zwischen den Zellen der verschiedenen Schichten. Daher kann es eine Vielzahl lokaler Minima geben. Ob das System das globale Minimum findet ist in großem Maß von der Form dieses *Fehlergebirges* und der Wahl der Startwerte der Gewichte abhängig. Diese Probleme lassen sich mit der Einführung von Nichtdeterminismus und Methoden wie dem Simulated Annealing behandeln.

Anwendungen

Das Backpropagation Verfahren ist in vielen Anwendungen neuronaler Netze implementiert. Beim sogenannten *Encoder* Problem handelt es sich um ein dreischichtiges Netz mit N Ein- und Ausgabezellen sowie $M \leq N$ Zellen innerhalb der inneren Schicht. Das Netzwerk soll lernen die Aktivierung eines einzelnen Neurons in der Eingabeschicht mit der Aktivierung eines einzelnen Neuron der Ausgabeschicht zu beantworten. Die Nummer dieses Neurons sollte dabei mit der Nummer des aktiven Neurons in der Eingabeschicht übereinstimmen.

Da die Zwischenschicht weniger Neuronen besitzt, bildet sie einen Flaschenhals und das System muß lernen seine Information entsprechend zu *kodieren*, um in dieser Schicht keine Information zu verlieren. Eine Lösung des Problems ist der Binärcode. Er erfordert, daß die Zwischenschicht mindestens $\log_2 N$ Neuronen besitzt. Simulationen haben gezeigt, daß eine solche Lösung mit dem Backpropagation Algorithmus tatsächlich gefunden wird. Es werden jedoch häufig auch alternative Lösungen gefunden.

Eine andere praktische Anwendung ist das *NetTalk* Projekt. Hier wurde versucht, ein Netz derart zu trainieren, daß es englische Wörter korrekt aussprechen kann. Zu diesem Zweck wird ein drei-schichtiges Feed Forward Netz mit 7×29 Eingabeneuronen zur Darstellung von 7 Buchstaben aus einem 29 elementigen Alphabet benutzt. Die mittlere Schicht besteht aus 80 inneren Neuronen, die Ausgabeschicht wird von 26 Ausgabezellen gebildet, die die unterschiedlichen Phoneme darstellen.

Das System wurde mit 1024 Wörtern der englischen Sprache trainiert. Nach 10 Trainingsduchläufen lieferte das Netz bereits eine verständliche Ausgabe. Nach 50 Durchläufen war seine Aussprache zu 95% korrekt. Ein beliebiger Text wurde nun zu 78% korrekt ausgesprochen. Obwohl das NetTalk-Netz kommerziellen Systemen wie dem DEC-talk System unterlegen ist, sind die Erfolge, die mit diesem relativ einfachen Ansatz erzielt worden sind, sehr beachtlicht, wenn man bedenkt, daß ein System wie DEC-talk die Arbeit unzähliger Linguisten über viele Jahre hinweg beinhaltet.

Das Hopfield-Modell

Beim *Hopfield-Modell* handelt es sich im Gegensatz zu den bisher behandelten Netzen nicht um ein Feed-Forward-Netz, sondern um ein vollständig rückgekoppeltes Netz. Durch die Rückkoppelung wirkt jedes Neuron auf die Eingänge aller übrigen Neuronen zurück. Durch die Anwendung der Theorie *wechselwirkender Vielteilchensysteme der Statistischen Physik* konnte Hopfield sein Modell genauer analysieren.

Definition: Hopfield Netz
Ein *Hopfield*-Netz ist ein neuronales Netz $H = (I, W, N, f)$ mit $I = 1, \ldots, N$ Neuronen, $W = \{-1, +1\}$, $N(i) = I$ für alle $i \in I$. Die lokalen Funktionen f_i seien lineare Schwellwertfunktionen mit der Schranke 0 und reellen Gewichten g_{ij} mit $g_{ij} = g_{ji}$.
Jedes Neuron des Netzes besteht aus einem McCulloch-Pitts Neuron mit dem Schwellwert 0 und der Ausgabefunktion $g(h) = \text{sgn}(h)$, so daß die einzelnen Neuronen die Werte 1 oder -1 annehmen können. Jedes Neuron ist mit allen anderen Neuronen verbunden, wodurch sich ein vollständig vernetztes Modell ergibt. Die Aktualisierung des Zustandes der einzelnen Zellen geschieht völlig asynchron. In jedem Schritt wird eine Zelle zufällig ausgewählt und aktualisiert. Das Hopfield-Modell löst das Problem der *Autoassoziation*. Dazu werden eine Anzahl von Mustern mit Hilfe des Netzes gespeichert. Wird nun ein Muster eingegeben, so entwickelt sich das System in Richtung des gespeicherten Musters, das der Eingabe am ähnlichsten ist. So kann bei der Eingabe eines unvollständigen Musters das vollständige Muster assoziiert werden.

Während bei den bisherigen Modellen innerhalb einer Lernphase die Gewichte w_{ik} der Synapsen an eine Menge von Trainingspaare angepaßt wurden, werden bei diesem Modell die Gewichte durch die im Netz gespeicherten Muster vorgegeben und nicht weiter verändert. Das Eingabemuster wird nun durch das Aktualisieren der Zellen solange verändert, bis das Netz einen stabilen Endzustand erreicht hat.

Das Netzwerk bestehe aus N Neuronen und es sollen p Muster gespeichert werden. Um die Wahl der Gewichte in Abhängigkeit dieser zu speichernden Muster zu motivieren, betrachten wir zunächst den Fall der Speicherung eines einzelnen Musters. Dieses Muster sollte stabil sein, das heißt, daß sich das Netzwerk nicht mehr verändern sollte, wenn dieses Muster als Konfiguration der Neuronen vorliegt. Für ein Muster ξ ergibt sich daraus folgende Bedingung:

$$S_i := \text{sgn}\Big(\sum_{j=1}^{N} w_{ij}\xi_j\Big) = \xi_i \quad \text{für alle i} \tag{15}$$

Diese Bedingung ist offensichtlich erfüllt, wenn w_{ij} proportional zu $\xi_i\xi_j$ ist. Daher können wir

$$w_{ij} = \frac{1}{N}\xi_i\xi_j \tag{16}$$

wählen. Im Falle der Speicherung von p Mustern ξ^μ überlagern wir diese Definition, so daß sich

$$w_{ij} = \frac{1}{N}\sum_{\mu=1}^{p}\xi_i^\mu\xi_j^\mu \tag{17}$$

ergibt. Auch diese Regel basiert bei genauem Hinsehen auf der von Hebb aufgestellten Hypothese. Für die Stabilität eines einzelnen Musters ξ^ν ergibt sich in diesem Fall:

$$\mathrm{sgn}(h_i^\nu) = \xi_i^\nu \quad \text{für alle i} \tag{18}$$

wobei h_i^ν gerade die Summe der Eingänge in Zelle i ist, wenn das Netz das Muster ν darstellt:

$$h_i^\nu = \sum_{j=1}^{N} w_{ij}\xi_j^\nu = \frac{1}{N}\sum_{j=1}^{N}\sum_{\mu=1}^{p}\xi_i^\mu\xi_j^\mu\xi_j^\nu \tag{19}$$

Durch Umsortieren der Terme ergibt sich damit:

$$h_i^\nu = \xi_i^\nu \frac{1}{N}\sum_{j=1}^{N}\sum_{\mu\neq\nu}\xi_i^\mu\xi_j^\mu\xi_j^\nu \tag{20}$$

Dieser zweite Term wird als *crosstalk term* bezeichnet. Ist er betragsmäßig kleiner als 1, so kann er das Vorzeichen von h_i^ν nicht verändern. In diesem Fall ist das Muster ν stabil.

Dies hängt natürlich in einem hohen Maße von der Anzahl p der zu speichernden Muster und der Anzahl N der vorhanden Zellen ab. Eine genauere Untersuchung ergibt, daß ab $p \geq 0.138N$ die Menge der Fehler sprunghaft ansteigt.

Ein Nachteil des Hopfield-Modelles besteht darin, daß die Ähnlichkeit zwischen zwei Mustern nur nach der Anzahl der übereinstimmenden Punkte beurteilt wird. Transformationen wie einfache Translationen werden von diesem Modell daher nicht berücksichtigt.

Energiefunktion

Einen anderen Zugang zum Hopfield-Modell stellt die Definition einer Energiefunktion H dar. In neuronalen Netzen existiert eine solche Energiefunktion immer, wenn symmetrische Gewichte vorliegen, d.h. wenn $w_{ij} = w_{ji}$ gilt. Die Energie ist definiert als:

$$H = -\frac{1}{2}\sum_{ij} w_{ij}S_iS_j \tag{21}$$

Hierbei beschreibt S_i den Zustand der Zelle i einer Konfiguration. Die Energiefunktion definiert ein *Potentialgebirge* auf dem Zustandsraum aller Konfigurationen, also aller binären Vektoren. Das Hopfield-Netz ist so konstruiert, daß das System von einem Anfangszustand startend innerhalb dieses Potentialgebirges immer bergab läuft, bis es in einem Tal, das heißt einem lokalen Minimum stecken bleibt. Jedes solche Minimum ist von einem Becken, dem sogenannten *Attraktionsbecken* umgeben, dessen zugehörige Eingabemuster sich alle zu diesem Minimum hin bewegen. Durch die oben beschriebene Wahl der Gewichte w_{ij} wird die Potentiallandschaft gezielt geformt, so daß die zu speichernden Muster gerade den lokalen Minima innerhalb der Becken entsprechen.

Die Energiefunktion nimmt in jedem Schritt des Netzes ab, solange sich die Konfiguration durch diesen Schritt ändert. Findet keine Änderung mehr statt, hat der Algorithmus ein lokales Minumum erreicht.

Jedoch kann die Energiefunktion H noch weitere lokale Minima besitzen, die nicht einem der vorgegebenen Muster zugeordnet sind. Ein Beispiel hierfür sind die zu den Mustern inversen Konfigurationen, bei denen jeder Wert ξ_i^μ durch $-\xi_i^\mu$ ersetzt ist. Diese Muster besitzen dieselbe Energie und aufgrund der Symmetrie des Modells ein vergleichbares Attraktionsbecken. Um diese *falschen* lokalen Minima zu umgehen, gibt es verschiedene Techniken.

Ein anderer Zugang zum Hopfield-Modell geht direkt von einer zu minimierenden Energiefunktion aus und bestimmt aus dieser die Gewichte des Netzes. Starten wir wieder mit dem Fall, daß lediglich ein Muster zu speichern ist und daß die Energie minimal ist, wenn das zu speichernde Muster ξ und die gegenwärtige Konfiguration S

möglichst weit übereinstimmen. Dies ergibt mit dem zunächst anscheinend willkürlich gewählten Faktor $1/2N$ die Energiefunktion:

$$H = -\frac{1}{2N}(\sum_{i=1}^{N} S_i \xi_i)^2 \tag{22}$$

Für den Fall der Speicherung mehrerer Muster ergibt sich durch die Summation über alle zu speichernden Muster die Energiefunktion:

$$H = -\frac{1}{2N}\sum_{\mu=1}^{p}(\sum_{i=1}^{N} S_i \xi_i^{\mu})^2 \tag{23}$$

Durch Ausmultiplizieren ergibt sich exakt die Energiefunktion (21) mit den Gewichten aus (17). Dieses Verfahren ist von prinzipieller Bedeutung. Kann man eine Energiefunktion definieren, die ein konkretes Problem löst, so können die Gewichte des zugehörigen Netzes durch Ausmultiplizieren der Energiefunktion gefunden werden.

Probabilistische Modelle

In starker Anlehnung an die Theorie *wechselwirkender Vielteilchensysteme der Statistischen Physik* kann man eine Form von Nichtdeterminismus in das Hopfield-Modell einführen. Es besteht eine enge Beziehung zwischen einem solchen Hopfield Netzwerk und einfachen Modellen von magnetischen Materialien. Hierbei werden solche Stoffe als aus vielen kleinen atomaren Magneten (*spins*) bestehend betrachtet, die im Falle von *Ising*-Modellen lediglich zwei Richtungen einnehmen können. Diese Richtungen stimmen mit den binären Zuständen der Neuronen des Hopfield-Netzes überein.

Ziel der Einführung dieser Art von Nichtdeterminismus ist es, es dem System zu ermöglichen, sich aus lokalen Minima zu befreien, die nicht den vorgegebenen gespeicherten Mustern entsprechen. Diese nicht erwünschten lokalen Minima haben i. a. eine höhere Energie und sind damit instabiler als die vorgegebenen Minima.

Ebenso wie bei diesen Modellen wird nun ein Nichtdeterminismus eingeführt, in dem die Wahrscheinlichkeit angegeben wird, mit der ein Neuron den Wert 1 bzw. -1 erhält. Die Größe h_i beschreibt hier wie üblich den Wert der mit den Gewichten multiplizierten und aufsummierten Eingänge des Neurons.

$$P(s_i = \pm 1) = f_\beta(\pm h_i) = \frac{1}{1+\exp(\mp 2\beta h_i)} \tag{24}$$

Die Ausgabefunktion, hier f_β wird auch oft als *logistische* Funktion bezeichnet. Dieser Nichtdeterminismus kann als eine Art *Rauschen* aufgefaßt werden, daß von dem Parameter β abhängt. Mit Hilfe von β kann analog zum physikalischen Vorbild eine Art *Pseudo-Temperatur* T definiert werden:

$$\beta = \frac{1}{T} \tag{25}$$

Diese Größe T beeinflußt den Sprung von -1 zu $+1$, den die Funktion f_β in der Nähe der 0 durchführt. Für sehr kleine T nähert sich diese Funktion immer mehr der im deterministischen Modell benutzten sgn Funktion an.

Es kann gezeigt werden, daß Netzwerke, die eine Energiefunktion besitzen, nach einiger Zeit immer in einen stabilen Gleichgewichtszustand kommen. In einem nichtdeterministischen Modell können wir jedoch nicht von absoluter Stabilität der Konfigurationen $\{S_i\}$ sprechen, sondern lediglich von einem Gleichgewicht der Mittelwerte $\{\langle S_i \rangle\}$, in dem sich diese Mittelwerte mit der Zeit nicht mehr verändern. Daher wird zur analytischen Untersuchung solcher Systeme eine Approximation über die Mittelwerte dieser Wechselwirkungen (*Mean Field Theory*) durchgeführt. Das Ergebnis dieser Untersuchung besteht darin, daß es, in Abhängigkeit von der Temperatur T des Systems eine scharfe Grenze gibt, an der das Netzwerk seinen Gleichgewichtszustand verliert. Für Temperaturen unterhalb dieser Grenze befindet sich die Konfiguration eines der vorgegebenen Muster in einem Gleichgewichtszustand. Überschreitet die Temperatur die eben beschriebene Grenze geht dieser Gleichgewichtszustand abrupt verloren. Man spricht hier auch von einem *Phasenübergang*. Diese Beobachtungen gelten für den Fall, daß $p \ll N$ ist.

Optimierungsprobleme

Bisher haben wir Hopfield Netze lediglich als assoziative Speicher kennengelernt, die eine Anzahl von Mustern speichern und dann auf die Eingabe eines beliebigen Musters mit einem der gespeicherten Muster antworten. Hopfield Netze sind jedoch auch in der Lage *kombinatorische Optimierungsprobleme* zu lösen. Hierbei geht es i.a. darum, in einem großen aber endlichen, diskreten Lösungsraum eine bezüglich einer Zielfunktion optimale Lösung zu finden. Neben der Zielfunktion gibt es meist weitere Nebenbedingungen, die eine zulässige Lösung beschreiben. Diese Bedingungen werden in Form weiterer Terme in die Energiefunktion aufgenommen, so daß die Verletzung dieser Bedingungen die Energie erhöht. Mit Hilfe dieser Energiefunktion werden dann die Gewichte des Netzes bestimmt. Zumeist wird hier ein probabilistisches Netz gewählt, um dem System zu ermöglichen, sich aus einem lokalen Minimum zu befreien.
Anhand des *Travelling Salesman* Problems soll diese Vorgehensweise verdeutlicht werden. Zur Erinnerung: Bei diesem Problem sind N Städte und die Entfernungen zwischen den Städten d_{ik} vorgegeben. Die Aufgabe besteht nun darin eine Tour durch alle Städte zu finden, die jede Stadt genau einmal besucht und minimale Länge besitzt. Zu diesem Zweck wird ein neuronales Netz mit N^2 binären Zellen eingesetzt. Wir bezeichnen die Elemente der Matrix mit $n_{i\alpha}$ wobei $i = 1, \dots, N$ für die verschiedenen Städte und $\alpha = 1, \dots, N$ für die Position innerhalb einer Route steht. $n_{i\alpha} = 1$ bedeutet damit, daß die Position α der Tour aus dem Besuch der Stadt i besteht. Anderenfalls trägt $n_{i\alpha}$ den Wert 0. Für die Energiefunktion E ergibt sich damit:

$$
\begin{aligned}
E &= \frac{1}{2}\sum_{i,j,\alpha}^{i\neq k} d_{ik} n_{i\alpha}(n_{k\alpha-1} + n_{k\alpha+1}) \\
&\quad + \frac{A}{2}\sum_{i,\alpha,\beta}^{\alpha\neq\beta} n_{i\alpha} n_{i\beta} + \frac{B}{2}\sum_{i,k,\alpha}^{i\neq k} n_{i\alpha} n_{k\alpha} + \frac{C}{2}(\sum_{i,\alpha} n_{i\alpha} - N)^2
\end{aligned} \tag{26}
$$

Hier beschreibt der erste Term die Länge der aktuellen Tour, der zweite Term ist ungleich 0, wenn eine Stadt mehrfach besucht wird, der dritte Term, wenn einer Position der Tour mehrere zu besuchende Städte zugeordnet sind. Der letzte Term stellt sicher, daß genau N Zellen aktiv sind.

Für die Gewichte des Netzes ergibt sich damit:

$$
\begin{aligned}
w_{i\alpha,k\beta} &= d_{ik}(1 - \delta_{ik})(\delta_{\alpha-1,\beta} + \delta_{\alpha+1,\beta}) \\
&\quad + A(1 - \delta_{\alpha\beta})\delta_{ik} + B(1 - \delta_{ik})\delta_{\alpha\beta} + C
\end{aligned} \tag{27}
$$

Die Simulationsergebnisse bleiben jedoch deutlich hinter den Ergebnissen konventioneller Algorithmen zurück. Dies kann auch durch Veränderungen der Faktoren A, B und C nicht wesentlich verändert werden. Vielmehr müssen diese Faktoren in einem sehr beschränkten Bereich liegen, um überhaupt sinnvolle Ergebnisse zu erhalten.

Simulated Annealing

Diese Methode basiert auf einem physikalischen Phänomen. Erhitzt man ein Metall über seinen Schmelzpunkt hinaus auf und kühlt es dann sehr langsam ab, so nehmen die Moleküle innerhalb des Metalls ein globales Energieminimum an. Von diesem Vorgehen inspiriert wird beim Verfahren des Simulated Annealing die Temperatur T des Hopfield-Netzwerkes langsam gesenkt, um es dem System zu ermöglichen, das globale Minimum der Energiefunktion zu finden. So kann das System bei hoher Temperatur zunächst noch in Konfigurationen mit erheblich höherer Energie wechseln, während dies bei sich langsam senkender Temperatur immer unwahrscheinlicher wird. Ist die Temperatur $T = 0$ erreicht, entspricht das Modell einem deterministischen Hopfield Netz.

Erweiterungen

Das Hopfield-Modell kann auf verschiedene Weise verändert und erweitert werden. So kann eine Diskretisierung des gesamten Modells erforderlich sein, wenn das Netzwerk mit Hilfe elektronischer Schaltkreise realisiert werden soll. Auch eine Beschränkung der Gewichte (*Clipping*) auf einen vorgegebenen Bereich ist möglich. Eine andere Veränderung sieht vor, die Gewichte zwischen zwei beliebigen Zellen mit einer festen Wahrscheinlichkeit c auf 0 zu setzen, bzw. lediglich einen Bruchteil der zunächst existierenden Gewichte beizubehalten (*Dilution*).

Durch Hinzufügen zufälliger Werte zu allen Gewichten können asymmetrische Gewichte erzeugt werden. In einigen Implementationen sind lediglich positive oder negative Gewichte möglich. Dies ist durch Addieren einer Konstante zu allen Gewichten und das Einfügen eines entsprechenden Terms bei der Summierung der Gewichte der einzelnen Zellen möglich.

Eine weitere Abwandlung des Modells stellt die Einführung kontinuierlicher Zellen dar. Ebenso wie im binären Fall kann hier eine Energiefunktion definiert werden, gegen deren lokale Minima das System konvergiert.

Das Hopfield-Netz wird jedoch nicht nur eingesetzt um gegen einen einer Reihe von vorgegeben Attraktionspunkten zu konvergieren. In einer anderen Anwendung wird das Modell in abgewandelter Form dazu benutzt, Sequenzen von Mustern zu speichern und wiederzuerkennen.

Symbolisches Lösen von algebraischen Ungleichungen

Sergei O. Ivanov
Universität GH Paderborn, sergiva@uni-paderborn.de

In diesem Artikel berichte ich über einen Algorithmus zum Lösen von algebraischen Ungleichungen. Desweiteren wird eine Implementation für das Computer-Algebra-System MuPAD [1] vorgestellt.

Einleitung

Dieser Artikel beschreibt einen allgemeinen Algorithmus zur Bestimmung der Lösungsmenge einer algebraischen Ungleichung in Abhängigkeit einer reellwertigen Unbestimmten. Dieser Algorithmus benötigt eine Vielzahl von Rechenschritten, ist aber gut geeignet zur Implementation in ein Computer-Algebra-System (CAS) [1,2].

Der Algorithmus setzt die Existenz einer Prozedur zum symbolischen Lösen von algebraischen Gleichungen voraus. Diese sei im folgenden mit dem Namen `solve` bezeichnet.

Nachdem der Algorithmus beschrieben wurde, werde ich die Implementation des Algorithmus in *MuPAD* anhand einiger Beispiele vorstellen.

Der Algorithmus

Das natürliche Vorgehen zur Bestimmung der Lösungsmenge einer Ungleichung besteht darin, zuerst das Definitionsgebiet der Unbestimmten zu bestimmen.

Das wird wie folgt realisiert: Aus der gegebenen Ungleichung werden eine Reihe spezieller Gleichungen erzeugt (Nenner und Radikanden). Die Lösungen dieser Gleichungen fasse ich mit der Lösung der Ungleichung aufgefaßt als Gleichung zusammen. Auf diese Menge wird die Intervallmethode angewendet, die die Lösungsmenge der Ungleichung liefert.

Für jede Ungleichung es ist möglich, sie in der Form $expression > 0$ darzustellen. Enthält $expression$ Radikanden, so müssen die Gleichungen $Radikand_1 = 0, Radikand_2 = 0, \ldots$ gelöst werden. Enthält $expression$ Brüche, so müssen die Gleichungen $Nenner_1 = 0, Nenner_2 = 0, \ldots$ gelöst werden. Sämtliche gefundenen Lösungen seien in der Menge $defdom$ zusammengefaßt.

Schließlich wird die Gleichung $expression = 0$ gelöst und deren Lösungen zu der Menge $defdom$ zugefügt. Die Lösungen der betrachteten Ungleichung erhält man dann durch die Intervallmethode, angewandt auf die Menge $defdom$.

ALGORITHMUS: *Symbolisches Lösen von Ungleichungen*
EINGABE: $expr_1 > expr_2, x$
AUSGABE: *Lösungsmenge*

```
BEGIN
   expr := expr1-expr2;
   nenner_menge := {Nenner in expr};
   rad_menge := {Radikanden in expr};
   defdom := {};

   for ex in nenner_menge do
      defdom := defdom union solve(ex=0, x);
   end_for;

   for ex in rad_menge do
      defdom := defdom union solve(ex=0, x);
   end_for;

   sls := solve(expr=0, x);
   interval_menge := {};

   for k from 1 to nops(sls)-1 do
      interv := sls[k]..sls[k+1];
      cp := Punkt aus interv
      cpl := sls[k]; # linke Grenze #
      cpr := sls[k+1]; # rechte Grenze #

      if expr in x=cpl > 0 then
         interval_menge := interval_menge union {cpl..cpl};
      end_if;
      if expr in x=cpr > 0 then
         interval_menge := interval_menge union {cpr..cpr};
      end_if;
      if expr in x=cp > 0 then
         interval_menge := interval_menge union
                                  {Open(cpl)..Open(cpr)};
      end_if;
   end_for;

   return( interval_menge );
END
```

Wir betrachten das folgende Beispiel einer Ungleichung in x:

$$x\frac{\sqrt{x+5}}{\sqrt{x+6}} < 0.$$

Wir transformieren diese Ungleichung auf die Form

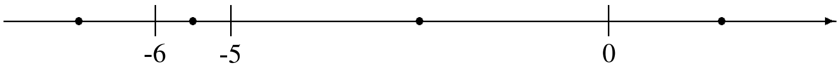
$$-x\frac{\sqrt{x+5}}{\sqrt{x+6}} > 0$$

und lösen zuerst: $x+5=0, x+6=0, \sqrt{x+6}=0$. Das liefert uns uns die Menge $defdom = \{-6,-5\}$. Zu dieser fügen wir noch die Lösungen der Gleichung

$$-x\frac{\sqrt{x+5}}{\sqrt{x+6}} = 0,$$

also $x = 0$ und $x = -5$ hinzu: $defdom = \{-6, -5, 0\}$.

Wir betrachten die Elemente dieser Menge auf der reellen Zahlengerade:



und testen die Ungleichung mit jeweils einem Punkt aus jedem dieser Intervalle und mit deren Intervallgrenzen. Die Lösungen liegen im Intervall $(-5, 0)$.

Beispiele

Ich stelle die Implementation des oben beschriebenen Algorithmus in *MuPAD* anhand einiger Beispiele vor.

Das Paket trägt den Namen `ineqs`. Die Prozedur `ineqsolve` ist die Hauptprozedur in `ineqs` und liefert eine Menge von Ausdrücken vom Typ `IntervalSet` als Beschreibung der Lösungsmenge der Ungleichung zurück. Dabei beschreibt ein Ausdruck der Form `a..b` das geschlossene Intervall $[a..b]$ und der Ausdruck `Open(a)..Open(b)` das offene Intervall $(a..b)$. Ferner existieren die Kombinationen der Form `a..Open(b)` und `Open(a)..b`.

```
>> loadlib("ineqs");

                                        TRUE

>> ineqsolve(x*sqrt(x+5)/sqrt(x+6)<0,x);

                              [Open(-5)..Open(0)]

>>  ineqsolve(x*sqrt((x+5)/(x+6))<0,x);

         [Open(-infinity)..Open(-6), Open(-5)..Open(0)]

>> ineqsolve(abs(abs(x^2-3*x+2)-1)> x-2,x);

                                  1/2
   [Open(-infinity)..Open(2     + 1), Open(3)..Open(infinity)]

>> ineqsolve(abs(x^2-abs(x))<1/4,x);

--      /    1/2        \
|       |   2           |
|  Open| - ---- - 1/2 |..Open(-1/2), Open(-1/2)..Open(1/2),
--      \    2          /

                  /  1/2        \ --
                  | 2           |  |
   Open(1/2)..Open| ---- + 1/2 |  |
                  \  2          / --
```

```
>> ineqsolve(
      sqrt(x^2-8*x+15)+sqrt(x^2+2*x-15)>sqrt(4*x^2-18*x+18),x
);

                 [Open(17/3)..Open(infinity)]

>> ineqsolve(
       sqrt(2*x-1)+sqrt(3*x-2) < sqrt(4*x-3)+sqrt(5*x-4),x
);

                  [Open(1)..Open(infinity)]
```

Zusammenfassung

Der vorgestellte Algorithmus basiert auf der Prozedur zum symbolischen Lösen von algebraischen Gleichungen. Wird die Klasse von Gleichungen, welche durch diese Prozedur gelöst werden können, erweitert, so erhöht sich automatisch die Funktionalität des vorgestellten Algorithmus.

Literatur

[1] B. FUCHSSTEINER ET. AL., *MuPAD Benutzerhandbuch,* Birkhauser Verlag, Basel, 1993.

[2] S. O. IVANOV, *Some Variations on the Mandelbrot Set*, mathPAD, vol. 4, No. 3, Dezember 1994, s. 31–34.

New features in *MuPAD* 1.2.2

Paul Zimmermann
INRIA Lorraine, Paul.Zimmermann@loria.fr

This document describes the new features of MuPAD 1.2.2 with respect to the previous version (1.2.1).

All these new features can be obtained on-line with the command `?news`. This document does not describe the changes of functionality, that are described by `?changes`.

New functions in the standard library

For more information about these new functions, you can call the corresponding on-line help with `?function`.

Aliases. The new function `alias` enables you to define (parameterized) aliases, which makes *MuPAD* programs much easier to write (and to read):

```
>> alias(head(l)=op(l,1)):
>> alias(tail(l)=subsop(l,1=null())):
>> l:=[a,b,c,d]:
>> head(l), tail(l);

                         a, [b, c, d]
```

If the left hand side of the equation given to `alias` is a function call, then the arguments are considered as free variables, like with the macro `define` in the C programming language.

Limits, series and asymptotic expansions. The new function `series` computes general Puiseux series expansions:

```
>> series(sin(sqrt(x)),x=0);

                     3/2     5/2
                1/2   x       x          3
               x    - ---- + ---- + O(x )
                      6      120
```

The objects returned by `series` can be directly manipulated with arithmetic operations, for example to get the inverse of the above expansion, one can simply write:

```
>> 1/%;

                1/2       3/2
 1       x      7 x          2
---- + ---- + ------ + O(x )
 1/2     6      360
x
```

The new function `limit` implements the MRV-algorithm (most rapidly varying) developed by D. Gruntz and G. Gonnet [1]. It is the best known algorithm today:

```
>> limit((exp(x*exp(-x)/(exp(-x)+exp(-2*x^2/(x+1))))-exp(x))/x,x=infinity);

                                  -exp(2)
```

The `asympt` command enables to compute generalized expansions (not necessarily of Puiseux type):

```
>> asympt(sin(1/x+exp(-x))-sin(1/x),x);

                                         / exp(-x) \
           exp(-x)    exp(-x)     | ------- |
exp(-x) - ------- + ------- + O|    6    |
               2          4       \    x     /
            2 x       24 x
```

Bessel functions. The new functions `besselJ` and `besselY` correspond to the Bessel functions of first and second kind respectively. Both these functions can be evaluated numerically in the complex plane:

```
>> besselJ(2,2.0+I), besselY(2,2.0+I);

      0.4126719082 + 0.2659739228 I, - 0.5737407339 + 0.4104130982 I
```

Collecting coefficients. The new function `collect` collects the coefficients of the same power in a polynomial.

```
>> p := x*y+z*x*y+y*x^2-z*y*x^2+x+z*x:
>> collect(p,[x,y]);

                                           2
x (z + 1) + x y (z + 1) + x  y (- z + 1)
```

An additional third argument can be mapped onto the collected coefficients:

```
>> collect(p,[x],Factor);

                                    2
    x (y + 1) (z + 1) - x  y (z - 1)
```

Combining expressions. The function `combine` is the inverse of `expand`. It puts powers of the same expression together, . . . With a given second argument, it does a specialized job:

```
>> combine(x^a*x^b), combine(sqrt(2)*sqrt(3),sqrt);

                                a + b   1/2
                               x     , 6
```

Decomposing polynomials. The function `decompose` tries to decompose a polynomial f, i.e. to find polynomials g and h such that $f = g \circ h$.

```
>> decompose(x^6+6*x^4+x^3+9*x^2+3*x-5);

                               2            3
                           x + x  - 5, 3 x + x
```

The main application of the polynomial decomposition is to solve polynomial equations. To solve $f = 0$, one first determines the roots α_i of $g = 0$, then one solves $h = \alpha_i$. This is done automatically by `solve`, and enables *MuPAD* to solve irreducible equations of degree 6 or higher like the above one.

Factorization. The new function `Factor` is an interactive interface for the library function `factor`. It returns a *MuPAD* expression instead of a list:

```
>> Factor(x^3-3*x+2);

                                           2
                             (x + 2) (x - 1)
```

Extended gcd. The function `gcdex` implements the extended Euclidean algorithm for two univariate polynomials:

```
>> gcdex(x^3+1,x^2+2*x+1,x);

                                         x
                           x + 1, 1/3, - - + 2/3
                                         3
```

Infinity. The symbol `infinity` now represents a positive real infinite value. It is implemented as a domain. Simplifications like `infinity+infinity=infinity` and `infinity+c = infinity` where c is a constant are done automatically.

Integration. There is now a function `int` in the standard library to compute indefinite integrals. As in version 1.2.1, it implements Risch's algorithm for elementary functions, but one now gets an output expressed with the same functions as the input function if possible:

```
>> int(cos(x)^3*sin(x)^2,x);

                        sin(x)   sin(3 x)   sin(5 x)
                        ------ - -------- - --------
                          8         48         80
```

In what concerns definite integration, only the polynomial case is currently implemented:

```
>> int(a*x^2+b*x+c,x=0..1);

                                  a   b
                                  - + - + c
                                  3   2
```

but one can evaluate numerically proper integrals without singularities:

```
>> float(int(sin(1+cos(x)),x=0..1));

                                0.9542771279
```

Extending functionality. The function maprat enables you to apply a given procedure, which normally applies only to a restricted class of expressions, to a wider class of expressions. Suppose for example that you want to factor $\sin^2(1) - 1$. The command factor fails because this is not a polynomial with unknown variables.

```
>> maprat(sin(1)^2-1,Factor);

                          (sin(1) + 1) (sin(1) - 1)
```

It works as follows: firstly maprat calls the companion function rationalize to make the expression rational, by substituting non-rational subexpressions by new variables:

```
>> rationalize(sin(1)^2-1);

                              2
                            X6  - 1, {X6 = sin(1)}
```

Then factor is called on the transformed expression, here $X_6^2 - 1$, and finally the original subexpressions are substituted back. Some additional arguments can be given to both maprat and rationalize in order to accept a different class of expressions (the default class is that of rational expressions).

Partial fractions. The partfrac command computes the partial fraction decomposition of a rational function:

```
>> partfrac(x^4/(x^3-3*x+2));

                         11          16           1
                 x + --------- + --------- + ----------
                     9 (x - 1)   9 (x + 2)            2
                                             3 (x - 1)
```

Polynomial equations. The `pdioe` command solves polynomial equations of the form $ap + bq = c$:

```
>> pdioe(x+1,x^2+1,x-1,x);

                                   x, -1
```

Primitive part. The function `primpart` computes the primitive part of a polynomial with respect to a list of unknowns:

```
>> primpart(6*x^3*y + 3*x*y + 9*y, [x]);

                                      3
                                x + 2 x  + 3
```

Radical denesting. The function `radsimp` simplifies expressions involving nested square roots. The following nice example is taken from [2]:

```
>> radsimp(sqrt(14+3*sqrt(3+2*sqrt(5-12*sqrt(3-2*sqrt(2))))));

                                    1/2
                                   2    + 3
```

Random polynomials. The function `randpoly` generates a random polynomial, given a list of unknowns and a given type of coefficients. For example to generate a random univariate polynomial with complex coefficients, simply write:

```
>> loadlib("domains"): DIGITS:=3:
>> randpoly([x], Numerical, Terms=2);

                                   3                     5
   poly((- 6.29e-1 - 5.35e-1 I) x  + (1.02 + 1.32 I) x , [x], Numerical)
```

Rectangular form. The function `rectform` puts a complex expression in rectangular form, i.e. in the form $a + ib$. By default all variables are taken as complex, however one can give as second argument a set of real variables:

```
>> rectform(sin(x));

            sin(Re(x)) cosh(Im(x)) + (cos(Re(x)) sinh(Im(x))) I
```

Simplification. The function `simplify` is a general simplifier. Like `combine`, it can be given a second argument to specify which kind of simplification one wants:

```
>> simplify((exp(x)-1)/(exp(x/2)+1),exp);
```

```
        / x \
     exp| - | - 1
        \ 2 /
```

Solving equations. The function `solve` is the main interface to solve equations and systems. It can be overloaded to solve over domains, for example `ode::solve` solves ordinary differential equations. The standard function can solve polynomial equations, either directly for degree less or equal to four, or by factorization (see `factor`) or decomposition (see `decompose`). It can also solve systems of linear and algebraic equations, using either the `linalg` package or the `groebner` package:

```
>> solve({x^2+y^2=1,a*x+b*y=0},{x,y});

        -- --        b y                  2    2   2    2       -- --
        |  |   x = - ---, y = RootOf(- a  + y  (a  + b ), y)   |  |
        -- --         a                                         -- --
```

Summation. The function `sum` performs both definite and indefinite summation. It implements Abramov's algorithm for rational functions:

```
>> sum(1/(k^2+21*k),k=1..infinity);

                                 18858053/108636528
```

and Gosper's algorithms for hypergeometrics:

```
>> sum((-1)^k*binomial(n,k),k);
                                                      k
                                k binomial(n, k) (-1)
                              - ----------------------
                                          n
```

For definite sums, Zeilberger's algorithm is used, which may return a linear recurrence (only very easy recurrences are solved). For example, if you try to compute $F_n = \sum_{k=0}^{n} \binom{n}{k}^2 \binom{n+k}{k}^2$, which was used by Apéry to prove the irrationality of $\zeta(3)$, then you will get:

```
>> sum(binomial(n,k)^2*binomial(n+k,k)^2,k=0..n);

      / {                          3
      | {                u(n) (n + 1)
rsolve| { u(n + 2) = - ------------- +
      | {                         3
      \ {                  (n + 2)

                                   2        }        \
   u(n + 1) (2 n + 3) (51 n + 17 n  + 39)   }        |
   ------------------------------------- }, u(n)  |
                            3               }        |
                     (n + 2)                }        /
```

permute	generates all permutations of a list
powerset	generates all subsets of a set
stirling1	Stirling numbers of the first kind
stirling2	Stirling numbers of the second kind

Figure 1: The functions of the `combinat` package.

TEX formatting. The function `TeX` converts a *MuPAD* expression to TEX format, for example:

```
>> print(Unquoted,TeX(int(exp(-x^2/sigma^2),x=0..infinity)));

   \int_{0}^{\infty} \mbox{exp}\left(- x^2 \frac{1}{\sigma^2}\right) d x
```

Printing infos. The function `userinfo` enables the programmer to write some informations in his program, which are printed when the user increases the corresponding information level using the command `setuserinfo`. Several `userinfo` commands are already written in the library, whence one can get some details about the computations done after `setuserinfo(Any,1)` for example, which sets the global level to 1.

New packages

Colors. The package `RGB` allows to call colors by their name in plot commands. Try for example:

```
>> loadlib("RGB"):
>> plot2d([Mode=Curve,[x,sin(x)],x=[0,PI],Color=[Flat,RGB::Chocolate]]);
```

The list of all the 194 color names is given by `RGB::ColorNames()`.

Combinatorics. The package `combinat` (see Fig. 1) is designed to assist combinatorial computations, for example compute all permutations of a given list of expressions, or all subsets of a given set:

```
>> loadlib("combinat"):
>> combinat::powerset({a,b,c});

          {{}, {b}, {c}, {a}, {b, c}, {a, b}, {a, c}, {a, b, c}}
```

Differential equations. *MuPAD* can now solve ordinary differential equations, using the `ode` package. It works as follows. One first defines an equation with the `ode` command:

```
>> loadlib("ode"):
>> ode(x^2*diff(y(x),x)+3*x*y(x)=sin(x)/x,y(x));
                /             2                  sin(x)        \
             ode| 3 x y(x) + x  diff(y(x), x) = ------, y(x) |
                \                                  x           /
```

cylindricalplot	plot in cylindrical coordinates
fieldplot	plot vector fields
implicitplot	implicit two-dimensional plot
polarplot	plot in polar coordinates
sphericalplot	plot in spherical coordinates
xrotate	plot surfaces of revolution around the x-axis
yrotate	plot surfaces of revolution around the y-axis

Figure 2: The functions of the `plotlib` package.

ChiSquare	values for a Chi square distribution
mean	average of a list of values
normal	values for a normal distribution
stdev	standard deviation
variance	variance of a list of values

Figure 3: The functions of the `stats` package.

which creates an object of the domain `ode`, then one solves this equation with the function `solve`, whose overloading mechanism automatically calls the method `solve` of the domain `ode`:

```
>> solve(%);
                                   -- C1    cos(x) --
                                   |  -- - ------  |
                                   |   3      3    |
                                   -- x      x     --
```

The main advantage of this process is that the user does not have to remember the name of a special function to solve differential equations; moreover other methods can be easily defined for the `ode` domain, for example a method `diff` could compute the differential equation satisfied by the derivative of $y(x)$. The `ode` package recognizes autonomous equations, separable equations, linear equations and systems, simple first-order equations, and finds polynomial equations of linear equations by undetermined coefficients. The only function of the `ode` package to be exported is currently `solve`.

Graphics. The `plotlib` package contains several functions (see Fig. 2) to produce special graphics. For example the well-known cardiod is obtained by:

```
>> loadlib("plotlib"):
>> plotlib::polarplot([[1-cos(t),t],t=[0,2*PI],Grid=[50]]);
```

Statistics. The `stats` package is an aid for the analysis of data values (see Fig. 3). To compute the standard deviation of a list of data, simply call `stats::stdev` as follows:

```
>> loadlib("stats"):
>> stats::stdev([4.3,4.5,5.1,4.2,5.7,5.3,4.0]);
```

```
                              0.5921251936
```

Networks. Finally, the `Network` package contains several functions (see Fig. 4) to deal with directed graphs. For example to get the shortest path from 1 to 4 in the graph with edges $1 \rightarrow 2$, $1 \rightarrow 3$, $2 \rightarrow 3$, $2 \rightarrow 4$, $3 \rightarrow 4$, $3 \rightarrow 5$, $4 \rightarrow 5$, you simply write

```
>> loadlib("Network"):
>> N1:=Network([1,2,3,4,5],[[1,2],[1,3],[2,3],[2,4],[3,4],[3,5],[4,5]]):
>> Network::ShortPath(N1,1,Path)[4];

                              [1, 2, 4]
```

therefore the shortest path from 1 to 4 is of length 2 and goes through node 2.

New functions in existing packages

The `linalg` package contains many new functions: `cholesky`, `curl`, `divergence`, `eigenValues`, `eigenVectors`, `extractMatrix`, `grad`, `isHermitian`, `isOrthogonal`, `isPosDef`, `jacobian`, `jordanForm`, `normalize`, `randomMatrix` and `vectorPotential`:

```
>> loadlib("linalg"): M:=SquareMatrix(2):
>> A:=M([[1,2],[3,4]]):
>> linalg::eigenValues(A);

                     --   1/2             1/2       --
                     |  33              33           |
                     |  ----- + 5/2, - ----- + 5/2   |
                     --   2               2         --
```

A lot of other functions have been improved, in particular `linearSolve`.

The `numlib` package contains three new functions: `decimal` computes the infinite decimal representation of a rational, `pollard` implements Pollard's ρ method to factor an integer, and `proveprime` proves the primality of an integer using the ECPP method (Elliptic Curve Primality Proving), giving a primality certificate (see the help page of `proveprime`) that can be checked independently, for example using `check`:

```
>> loadlib("numlib"):
>> numlib::proveprime(1009);

               [1009, 67, [947], 177, 118, 1, 392, [947]]

>> numlib::check(%);

                                  TRUE
```

New domains and domain constructors

All the examples of this section suppose that the `domains` package has been loaded, by the command `loadlib("domains")`.

The domain constructor `AlgebraicExtension` enables to deal with algebraic numbers and functions. For example to get the normal form of $2\phi/(\phi^3+1)$ when $\phi^2-\phi-1=0$, one writes:

```
>> Qphi := AlgebraicExtension(Rational, x^2-x-1);

                                                2
             AlgebraicExtension(Rational, - x + x  - 1 = 0, x)

>> Qphi(2*x)/Qphi(x^3+1);

                                   x - 1
```

The domain constructor `Product` enables to create products of identical domains:

```
>> A:=Product(Rational,3);
>> A::random();

                        [764/979, 221/72, 229/220]
```

The domain constructor `TruncatedPowerSeries` implements arithmetic of truncated power series over a given coefficient field:

```
>> S:=TruncatedPowerSeries(x,Rational):
>> s:=S(1+x/2+x^2/6+O(x^3));
>> 1/s;

                                  2
                              x   x            3
                            - - + -- + 1 + O(x )
                              2   12
```

The domain `Interval` enables to compute with floating-point intervals:

```
>> Interval(-2..3)^3;

                                  -8..27
```

This domain can be used as a field in other domain constructors, for example in matrices:

```
>> MI:=SquareMatrix(2,Interval):
>> A:=MI::random();
```

AddEdge	adds one or several edges to a network
AddVertex	adds one or several vertices to a network
AllShortPath	shortest pathes for every pair of nodes
ChangeEdge	changes weight and capacity of one or several edges
ChangeVertex	changes the weight of one or several vertices in a network
Complete	generates a complete network
ConvertSSQ	converts a network into a single source single sink network
Cycle	generates a cycle
DelEdge	deletes one or several edges from a network
DelVertex	deletes one or several vertices from a network
ECapacity	returns the table of capacity
EWeight	returns the table of edge weight
Edge	returns a list with all edges
Epost	returns a table with adjacency lists for outgoing edges
Epre	returns a table with adjacency lists for incoming edges
InDegree	returns the indegree for nodes
IsEdge	checks whether an edge is contained in a network
IsVertex	checks whether a vertex is contained in a network
LongPath	longest pathes from one single node
MaxFlow	computes a maximal flow through a network
MinCost	computes a minimal cost flow
MinCut	computes a minimal cut
new	generates a new network
OutDegree	returns the outdegree for nodes
PrintGraph	print all information about a network
Random	generates a random network
ShortPath	shortest pathes from one single node
ShortPathTo	shortest pathes to one single node
ShowGraph	plots a network
TopSort	topological sorting of the nodes
VWeight	returns the table of vertex weight
Vertex	returns a list with all vertices

Figure 4: The functions of the `Network` package.

```
      +-                                                  -+
      |   0.4861272868..8.464924731,    6.451886974..6.686135983   |
      |                                                            |
      | -1.880224722..-1.080638751, 0.8005639644e-1..0.5365006573 |
      +-                                                  -+

>> 1/A;

   +-                                                              -+
   |   -1.631429498..1.218977401  , -1.660594478e1..-0.2165163261e-1 |
   |                                                                 |
   | 0.6314770003e-1..0.2681791695,   0.2840710648e-1..1.207364448   |
   +-                                                              -+
```

New methods in existing domains

A new method `random` has been added in all existing domain constructors, in order to generate a random element of this domain:

```
>> loadlib("domains"):
>> Rational::random(), Float::random();

                                  229/220, -0.9401748532
```

References

[1] GRUNTZ, D. *On Computing Limits in a Symbolic Manipulation System*. PhD thesis, Swiss Federal Institute of Technology Zürich, 1995.

[2] WESTER, M. A review of CAS mathematical capabilities. *Computer Algebra Nederland Nieuwsbrief 13* (Dec. 1994), 41–48.

Die *MuPAD*-Bibliothek `plotlib`

Thorsten Schulze
Universität GH Paderborn, heino@uni-paderborn.de

In diesem Artikel werden die Funktionen der *MuPAD*-Bibliothek `plotlib` vorgestellt und anhand von Beispielen erläutert. Diese Routinen dienen dazu, Gafiken in Polar-, Zylinder- und Kugelkoordinaten, zwei-dimensionale Vektorfelder als auch Rotationsflächen zwei-dimensionaler Kurven und implizite Funktionen darzustellen.

Die *MuPAD*-Bibliothek `plotlib` umfaßt zur Zeit die folgenden Routinen:

- `polarplot`
  erzeugt Grafiken in Polarkoordinaten.
- `cylindricalplot`
  erzeugt Grafiken in Zylinderkoordinaten.
- `sphericalplot`
  erzeugt Grafiken in Kugelkoordinaten.
- `xrotate`
  dient zur grafischen Darstellung von Rotationsflächen, welche sich durch Rotation einer zwei-dimensionalen Kurve um die x-Achse ergeben.
- `yrotate`
  dient zur grafischen Darstellung von Rotationsflächen, welche sich durch Rotation einer zwei-dimensionalen Kurve um die y-Achse ergeben.
- `fieldplot`
  dient zur grafischen Darstellung von zwei-dimensionalen Vektorfeldern.
- `implicitplot`
  dient zur grafischen Darstellung von zwei-dimensionalen, implizit definierten Funktionen.

Um diese Routinen in *MuPAD* zu verwenden, muß die Bibliothek geladen werden, was mit Hilfe des Befehles `loadlib("plotlib")` geschieht. Mit Hilfe des Befehles `export(plotlib)` können die Funktionen exportiert werden, so daß anstelle von `plotlib::polarplot()` auch direkt `polarplot()` aufgerufen werden kann.

Im folgenden werden die einzelnen Routinen anhand von *MuPAD*-Befehlen vorgestellt. Bei fast allen Funktionen (mit Ausnahme von `implicitplot()`) können dabei beliebig viele Objekte zu einer einzelnen Grafik zusammengefaßt werden. Der generelle Aufbau eines solchen *MuPAD*-Befehls ist dabei an die üblichen Befehle `plot2d()` und `plot3d()` angelehnt und kann folgendermaßen zusammengefaßt werden:

```
plot2d(<scene_option, ...> [ object <, object_option, ...>], ...);
```

Hierbei beschreibt `<scene_option, ...>` die Optionen, welche für die gesamte Grafik gelten sollen, wie z. B. den Achsenstil oder die Skalierung. Die Angabe dieser Argumente ist optional, was durch < > angedeutet werden soll. Im Anschluß an die Szene-Optionen sind die einzelnen Objekte zu definieren, wobei jedes Objekt in einer eigenen Liste zu spezifizieren ist. Objekte setzen sich aus notwendigen Argumenten (`object`) und optionalen Spezifikationen (`object_option`) zusammen, welche dazu dienen, die grafische Ausgabe eines solchen Objektes zu beschreiben, also bspw. in welcher Farbe oder welchem Stil ein Objekt ausgegeben werden soll; genauere Informationen zu den möglichen Optionen können in [1] nachgelesen werden.

Polar-, Zylinder- und Kugelkoordinaten

Grafiken in Polarkoordinaten werden mit Hilfe der Routine `polarplot()` erzeugt. Die notwendigen Objekt-Optionen sind dabei:

- Eine Liste bestehend aus zwei Ausdrücken, welche den Radius und den Winkel der Stützstellen beschreiben. Die beiden Ausdrücke können maximal von einer Variablen abhängig sein.
- Der Bereich, in dem die unabhängige Variable ausgewertet werden soll.

Die Befehle, um bspw. zwei Kreise mit unterschiedlichen Radien darzustellen, lauten:

```
>> loadlib("plotlib"):
   export(plotlib):
   polarplot([[1, angle], angle = [0, 2*PI]],
             [[2, angle], angle = [0, 2*PI]]);
```

Im obigen Befehl wurden jeweils nur die notwendigen Optionen der beiden Objekte spezifiziert. Wie jedoch bereits erwähnt, gibt es eine Vielzahl von Optionen, welche die grafische Ausgabe eines solchen Befehles beeinflußen. Die beiden folgenden Befehle erzeugen die in der Abbildung 5 dargestellten Grafiken.

```
>> loadlib("plotlib"):
   export(plotlib):
   polarplot
       (Axes = Origin, Ticks = 5,
        AxesOrigin = [0, YMin],
        [[sin(3*u), u],
          u = [0, PI],
          Grid = [100]
        ]);
```

```
>> loadlib("plotlib"):
   export(plotlib):
   polarplot
       (Axes = Corner, Ticks = 5,
        [[sin(2*u), u],
          u = [0, PI],
          Grid = [100]
        ]);
```

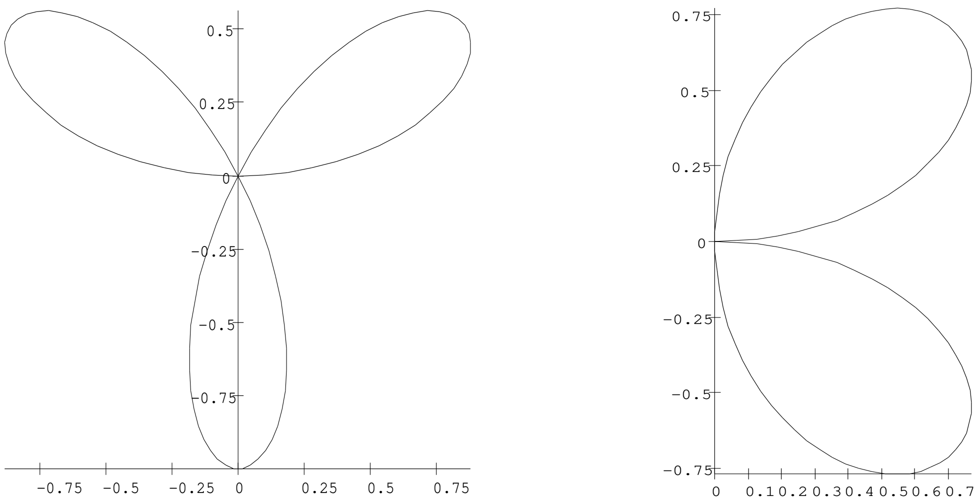


Abbildung 5: Beispiele für `polarplot()`

Mit Hilfe der Routine `cylindricalplot()` können Grafiken in Zylinderkoordinaten erzeugt werden. Die notwendigen Optionen eines Objektes sind dabei gegeben durch:

- Eine Liste bestehen aus drei Einträgen, welche den Radius, sowie den horizontalen Winkel und die Höhe der einzelnen Stützstellen beschreiben. Diese Ausdrücke können maximal von zwei freien Variablen abhängig sein.
- Zwei Bereiche, in denen die unabhängigen Variablen ausgewertet werden.

Die beiden folgenden Beispielbefehle erzeugen die in Abbildung 6 dargestellten Grafiken. Dabei enthält der Befehl auf der linken Seite, der zur Erzeugung des Zylinders dient, nur die notwendigen Objekt-Attribute. Im Befehl auf der rechten Seite hingegen wurden noch die Optionen `Axes` und `CameraPoint` für die Szene-Optionen und die Spezifikationen `Grid`, `Style` und `Color` verwendet, um die grafische Ausgabe zu beeinflußen.

```
>> loadlib("plotlib"):
   export(plotlib):

   cylindricalplot
       ([[1, angle, height],
          angle = [0, 2*PI],
          height = [-1, 1]
        ]);
```

```
>> loadlib("plotlib"):
   export(plotlib):

   cylindricalplot
       (Axes = None,
        CameraPoint = [16, -8, 41],
        [[u*v, 2*v, -3*cos(u^2)]],
          u = [-2, 2], v = [0, PI],
          Grid = [30, 30]
        ]);
```

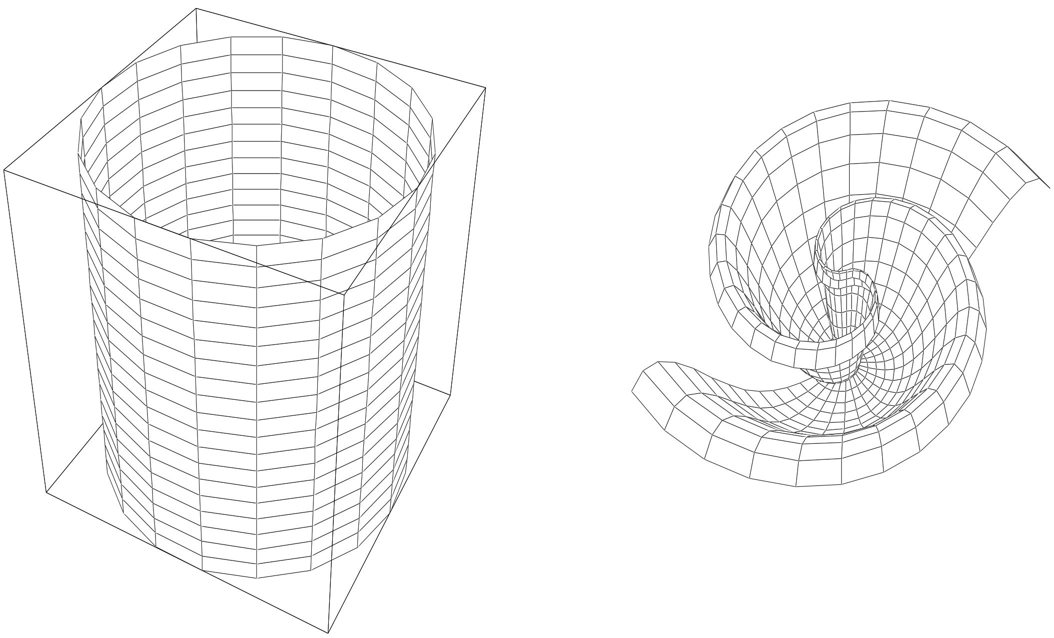


Abbildung 6: Beispiele für `cylindricalplot()`

Die notwendigen Optionen der Routine `sphericalplot()` sind ähnlich wie die der zuvor beschriebenen Funktion `cylindricalplot()`. Ein Objekt in Kugelkoordinaten wird beschrieben durch:

- Eine Liste bestehend aus drei Ausdrücken, welche den Radius, sowie den horizontalen und vertikalen Winkel beschreiben.
- Zwei Bereiche, in denen die unabhängigen Variablen evaluiert werden.

Die beiden folgenden Befehle erzeugen die in Abbildung 7 dargestellten Grafiken. Dabei sind im Befehl auf der linken Seite wiederum nur die notwendigen Optionen eines Objektes enthalten.

```
>> loadlib("plotlib"):
   export(plotlib):

   sphericalplot
       ([[1, phi, theta],
          phi = [0, PI],
          theta = [0, PI]
        ]);
```

```
>> loadlib("plotlib"):
   export(plotlib):

   sphericalplot
       (Axes = Box,
        CameraPoint = [15, 13, 14],
        [[(1.3)^z*sin(u), z, u],
          z = [-1,2*PI], u = [0,PI],
          Grid = [30, 30]
        ]);
```

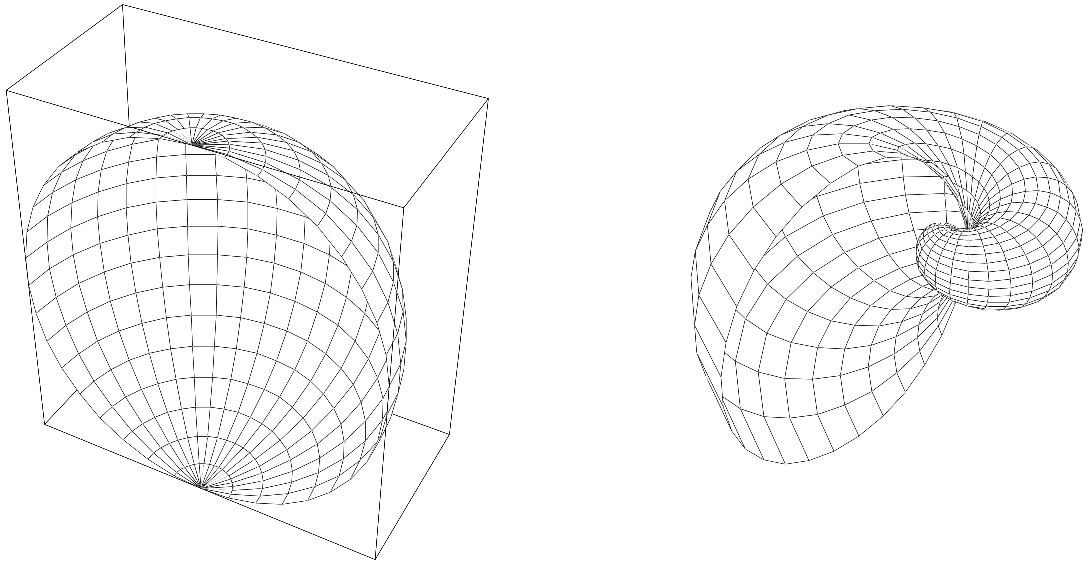


Abbildung 7: Beispiele für `sphericalplot()`

Rotationsflächen

Wie zuvor schon angedeutet, dienen die Routinen `xrotate()` und `yrotate()` dazu, Rotationsflächen darzustellen, die man erhält, wenn man zwei-dimensionale Kurven um die x-Achse bzw. um die y-Achse rotieren läßt. Beide Routinen erwarten dabei die folgenden notwendigen Eingaben:

- Eine Liste, in der die zwei-dimensionale Kurve, deren Rotationsfläche berechnet werden soll, parametrisiert wird.

- Einen Bereich für die unabhängige Variable dieser Parametrisierung.
- Einen Bereich, den der Rotationswinkel durchlaufen soll.

Will man bspw. die in Abbildung 5 dargestellten zwei-dimensionalen Kurven um die x- bzw. die y-Achse rotieren lassen, so sind die folgenden Befehle notwendig — dabei ist zu beachten, daß die Parametrisierung dieser Kurven nun in kartesischen Koordinaten einzugeben ist.

```
>> loadlib("plotlib"):
   export(plotlib):

   xrotate
       (Axes = None,
        [[sin(3*x)*cos(x),
          sin(3*x)*sin(x)],
          x = [0, PI],
          angle = [0, 2*PI],
          Grid = [50, 50]
     ]):
```

```
>> loadlib("plotlib"):
   export(plotlib):

   yrotate
       (Axes = None,
        [[sin(2*x)*cos(x),
          sin(2*x)*sin(x)],
          x = [0, PI],
          angle = [0, 3/2*PI],
          Grid = [50, 50]
       ]):
```

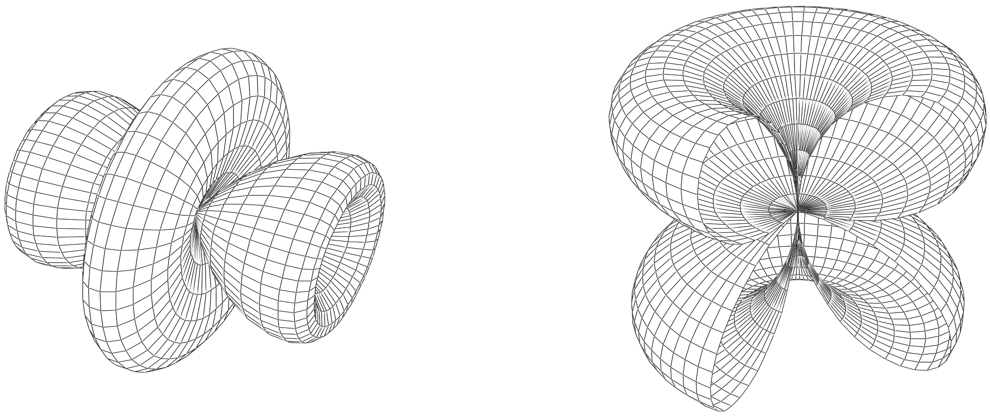


Abbildung 8: Beispiele für `xrotate()` und `yrotate()`

Vektorfelder

Um zwei-dimensionale Vektorfelder grafisch darzustellen, sollte die Routine `fieldplot()` verwendet werden. Die notwendigen Eingaben dieser Funktion sind:

- Eine Liste bestehend aus zwei Ausdrücken, welche das darzustellende Vektorfeld beschreiben. Die Ausdrücke können von maximal zwei Variablen abhängig sein.
- Zwei Bereiche, welche die Wertebereiche der unabhängigen Variablen angeben.

- Eine Liste bestehend aus zwei positiven ganzen Zahlen größer als 2, welche die Anzahl der Stützstellen angeben, an denen die obigen Ausdrücke ausgewertet werden.

Als Beispiel betrachte man die Gleichung

$$\varphi_{tt} + \sin(\varphi) = 0, \tag{28}$$

welche die dynamische Entwicklung eines Pendels beschreibt. Durch Einführung der folgenden Abkürzungen

$$x = \varphi \text{ und } y = \varphi_t$$

können wir Gleichung (28) auch folgendermaßen schreiben:

$$\begin{pmatrix} x \\ y \end{pmatrix}_t = \begin{pmatrix} y \\ -\sin(x) \end{pmatrix}. \tag{29}$$

Will man nun das in Gleichung (29) aufgestellte Vektorfeld grafisch darstellen, so sind in *MuPAD* die folgenden Befehle einzugeben:

```
>> loadlib("plotlib"):
   export(plotlib):

   fieldplot(Axes = Corner, Ticks = 8, Scaling = UnConstrained,
             [[y, -sin(x)], x = [-2*PI, 2*PI], y = [-PI, PI],
               Grid = [40, 40]]);
```

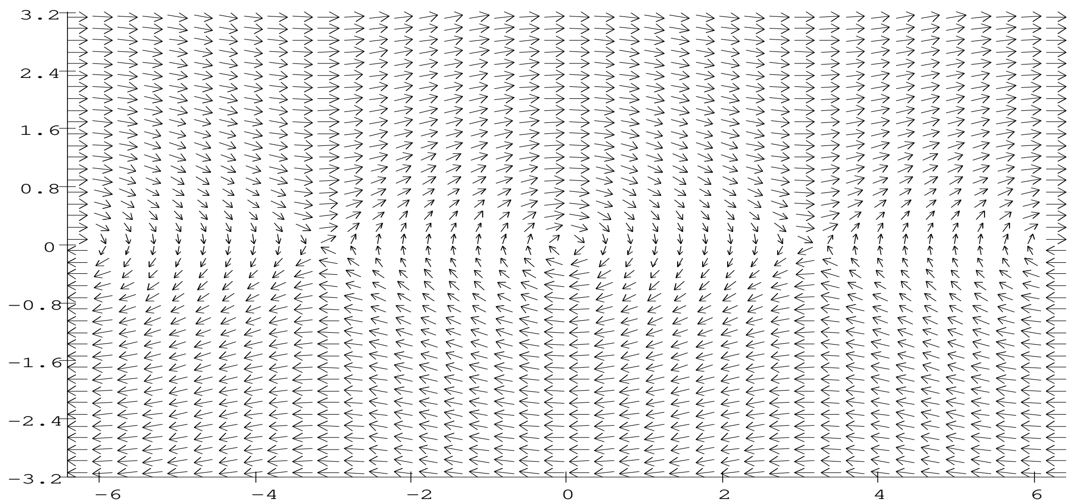


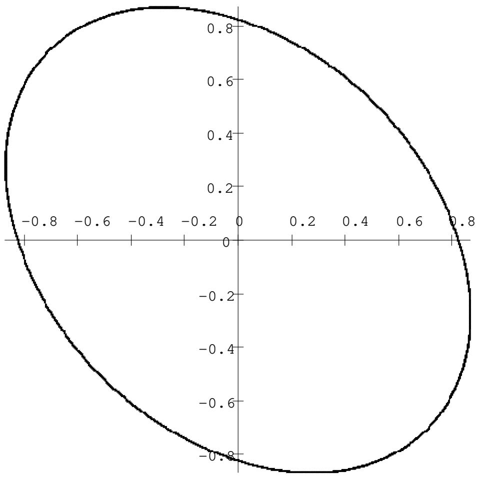
Abbildung 9: Beispiel für `fieldplot()`

Implizite Funktionen

Die Funktion `implicitplot()` stellt den Graphen einer implizit definierten Funktion $f(x, y) = 0$ dar. Anzugeben sind hierbei die implizit definierte Funktion als Prozedur oder als sogenannte *pure function* (vgl. [1]), sowie die Bereiche, in denen die unabhängigen Variablen ausgewertet werden sollen. Die grafische Darstellung einer impliziten Funktion setzt sich aus einer Reihe von Rechtecken zusammen, in denen die Nullstellen dieser Funktionen liegen. Zur Berechnung dieser Rechtecke wird der *Bisektions-Exklusions-Algorithmus* verwendet (siehe auch [2]). Die Anzahl der Unterteilungen der Intervalle kann dabei durch einen optionalen Parameter gesteuert werden. Die beiden folgenden Befehle erzeugen die in Abbildung 9 dargestellten Grafiken.

```
>> loadlib("plotlib"):
   export(plotlib):

   f := func(x^2+y^2-cos(x+y),
             x, y):

   implicitplot(f, -1..1, -1..1, 9):
```

```
>> loadlib("plotlib"):
   export(plotlib):

   g := func((x^2+y^2)^3-(x^2-y^2)^2,
             x, y):

   implicitplot(g, -1..1, -1..1, 9):
```



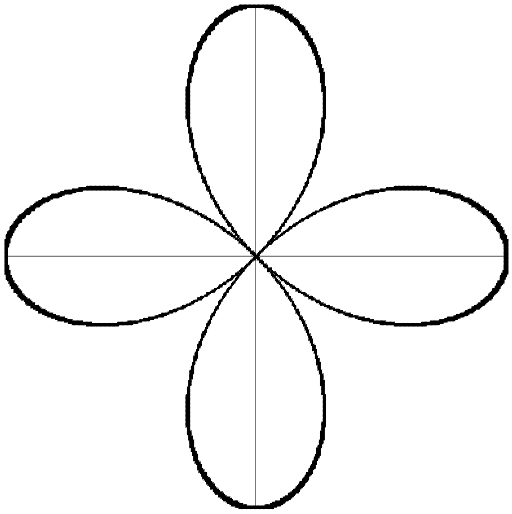


Abbildung 10: Beispiele für `implicitplot()`

Literatur

[1] B. Fuchssteiner, et. al., MuPAD-Benutzerhandbuch, Birkhäuser 1994

[2] Computing the real roots of a polynomial by the exclusion algorithm, Numerical Algorithms, No. 4, 1993

Surfin' the Web

Interessante, nützliche und kuriose Plätze im „global village“.

Für die Tips in dieser Ausgabe danken wir Prof. Dr. Klaus-Dieter Bierstedt[1] (bi) und Andreas Sorgatz[2] (so).

Mathematik allgemein

`http://elib.zib-berlin.de:88/Math-Net/Links/math.html`
„Math-Net Links to the Mathematical World“ (noch experimentell). Ausgezeichnete Sammlung des Konrad-Zuse-Zentrums (ZIB) Berlin quasi aller Links, die einen Mathematiker fachlich interessieren: Math-Web (incl. Scientific Computing), Math-Net (mathematical institutions, mainly in Germany), The Mathematical Museum, Electronic Publishing, Internet Resources. — Wenn man einen Eintrag für die „Hotlist“ oder die „Bookmarks“ braucht, von dessen Links aus man „alles“ erreichen kann: Hier ist er. (bi)

`http://elib.zib-berlin.de/Cimu`
Homepage der „International Mathematical Union (IMU)“ in Zusammenarbeit mit dem ZIB in Berlin — mit vielen nützlichen Links. (bi)

`http://www.EMIS.de/`
„European Mathematical Information Service (EMIS)“ der Europäischen Mathematischen Gesellschaft (EMS) in Kooperation mit dem FIZ Karlsruhe/Zbl. Math. — ebenfalls viele interessante Links. (bi)

`http://e-math.ams.org/`
„The American Mathematical Society's Homepage on e-Math“ Die AMS bietet hier sehr viel Interessantes an, darunter ihr Bulletin und ihre Notices. — Man kann dabei auch sehen, was die amerikanischen Mathematiker z.Zt. noch alles bewegt (ethical guidelines etc.). (bi)

`http://camel.cecm.sfu.ca/home.html`
Die Kanadische Math. Ges. hat elektronisch ebenfalls viel zu bieten. — Allerdings dauert es manchmal etwas länger, bis man „durchkommt“... (bi)

`http://archives.math.utk.edu/`
Mathematics Archives unterhält eine ftp Archiv mit mathematischer Software. Neben Informationen zu diesem Archiv sind hier auch viele Referenzen zu mathematischen Instituten und Arbeitsgruppen zu finden. (so)

Computeralgebra

`http://math-www.uni-paderborn.de/MuPAD/`
Hier findet der geneigte Leser stets die neuesten Informationen zur Entwicklung von *MuPAD*. Auûerdem gibt es einen FAQ Service (Installation, Known-Bugs, unterstützte Systeme, ...) und mehr. (so)

`http://www.uni-karlsruhe.de/~CAIS/`
Es werden Mitteilungen und Rundbriefe der Fachgruppe 2.2.1 „Computeralgebra“ der GI, GAMM und DMV dargeboten. Neben einer sehr ausführlichen Tagungsliste gibt es Informationen zu Arbeitsgruppen und Diskussionsforen. (so)

[1] `klausd@plato.uni-paderborn.de`
[2] `http://math-www.uni-paderborn.de/~andi/`

http://www.can.nl/
Dies ist ein Knoten des Computer Algebra Information Network (CAIN). Hier sind Referenzen zu interessanten Servern im Bereich der Computeralgebra aufgeführt, insbesondere zu den lokalen Seiten der Computer Algebra Nederland (CAN). (so)

Orientierung im WWW-Dschungel

http://cuiwww.unige.ch/meta-index.html
Enthält ein Formular zum Zugriff auf die wichtigsten „search engines" im Internet. (so)

Getting Away

http://www.uni-karlsruhe.de/~rail/
Mit Bahnverbindungen in Deutschland sowie Referenzen auf vergleichbare Dienste im Ausland. Besonders interessant ist ein WWW Formular, mit dem Anfragen an den RailServer gestellt werden können. Die möglichen Bahnverbindungen werden dann per email zugestellt. (so)

http://www.tkz.fh-rpl.de/tii/air/air.html
Der TII Flugverkehr Service enthält Referenzen zu Servern wichtiger Airlines sowie viele interessante Informationen für Flugreisende. Auch hier bieten einige Airlines WWW Formulare für Reiseauskünfte an. (so)

Tennis

http://atptour.com/data/press/news.html
„International Tennis Weekly (ITW): Official Newsletter of the ATP Tour" Berichte der ATP (Association of Tennis Professionals) zu den Herren-Tennisturnieren der vergangenen Woche, Auslosung für die Turniere der laufenden Woche usw. (bi)

http://www.tennisserver.com/WTA/WTA.html
„WTA Tour Notes & Netcords" Wöchentliche Berichte der „Women's Tennis Association" zu den Damen-Tennisturnieren; analog zum ITW, aber weniger ausführlich. (bi)

http://www.pathfinder.com/cgi-bin/SI/genlist/ten/lgns
„Sports Access from Sports Illustrated: Tennis News" erstklassige Quelle für die aktuellsten Tennis-Resultate. (Herren und Damen: laufende Turniere, Weltranglisten, Davis-Cup etc.) (bi)

http://www.mindspring.com/~csmith/TennisSupp.html#News
„Tennis News Sources" Links zu anderen interessanten Tennis-Informationen im Internet. (bi)

Fussball

http://ls2-www.informatik.uni-dortmund.de/Buli/Buli.html
„Bundesliga Server of Lehrstuhl Informatik II, Dortmund" Hier stellt Thomas Hofmeister stets sehr aktuell (samstags „live") Bundesliga-Resultate, Tabellen, Statistiken sowie Links zu anderen Fussball-Servern bereit — vorbildlich, mit viel Einsatz. (bi)

Teletexte

`http://134.109.112.4:9090/telenet/ARD/111.next/1.html`
Seite 112 des laufenden ARD/ZDF-Teletextes (Nachrichten-Überblick, Seite 1) (bi)

`http://134.109.112.4:9090/telenet/ARD/201.prev/1.html`
Seite 200 (wie oben) (Sport-Überblick, Seite 1) (bi)

`http://134.109.112.4:9090/telenet/ARD/422/1.html`
Seite 422 (wie oben) (aktuelle DAX-Werte an der Frankfurter Börse) — die letzten 3 Seiten sind experimentell, nicht immer aktuell, Links zu anderen Seiten sind nicht immer verläßlich. (bi)

`http://www.telecom.at/orf/teletext/210.htm`
„ORF World Wide Text" (Österreich), Seite 210 (Sport) — ausgezeichnet, up to date (vor allem beim Tennis). (bi)

`http://www.omroep.nl/cgi-bin/tt/nos/page/t/645-1`
NOS (Holland) Teletext, Seite 645/1 (ATP-Weltrangliste) — ausgezeichnet und stets up to date, aber von der Sprache her natürlich ein bißchen gewöhnungsbedürftig. (bi)

Trost

`http://www.dcs.ed.ac.uk/home/jhb/whisky/`
The Edinburgh Malt Whisky Tour bietet eine Vielzahl an Informationen zum Thema Whisky und ist eine wahre Fundgrube für Freunde eines guten Single Malts. (so)

Und sonst?

`http://math-www.uni-paderborn.de/HTML/Allerlei.html`
Eine sortierte Liste interessanter WepPages. (so)

Kalender

Modern Group Analysis VI

The 6th International Conference on Modern Group Analysis, Johannesburg, Südafrika, 15.–20. Januar 1996.

Arbeiten sind einzureichen bis:
1. August 1995

Kontakt:
Ms. F. Vawda
Dept. Comp. Applied Mathematics, University of the Witwatersrand
P O Wits 2050, Johannesburg, South Africa
Fax: (+27)(11) 339 79 65
Tel.: (+27)(11) 716 3444, (+27)(11) 716 3923
E-mail: `fatima@gauss.cam.wits.ac.za`

Rhine Workshop on CA

Fifth Rhine Workshop on Computer Algebra, Saint-Louis, Frankreich, 1.–3. April 1996

Arbeiten sind einzureichen bis:
31. Oktober 1995

Kontakt:
WWW: `http://avalon.ira.uka.de/iaks-calmet/conf/rwca96.html`

DISCO '96

International Symposium on Design and Implementation of Symbolic Computation Systems, Karlsruhe, 18.–20. September 1996

Arbeiten sind einzureichen bis:
19. Februar 1996

Kontakt:
WWW: `http://iaks-www.ira.uka.de/iaks-calmet/conf/disco.html`

MEGA 96

The Fourth International Symposium on Effective Methods in Algebraic Geometry, Eindhoven, Niederlande, 4.–8. Juni 1996

Arbeiten sind einzureichen bis:
16. Februar 1996

Kontakt:
Arjeh M. Cohen, Hans Cuypers, Hans Sterk
Fac. Wiskunde en Informatica
Technical University Eindhoven
Postbox 513
5600 MB Eindhoven
Niederlande
Tel.: 31–40–2472965, 2472727, 2473121
Fax: 31–40–2435810
E–mail: `mega96@win.tue.nl`
WWW: `http://www.win.tue.nl/win/mega96`

ISSAC '96

International Symposium on Symbolic and Algebraic Computation, Zürich, Schweiz, 24.–26. Juli 1996

Arbeiten sind einzureichen bis:
8. Januar 1996

Kontakt:
Bob Caviness
Computer & Information Sciences
103 Smith Hall
University of Delaware
Newark DE 19716, USA
E-mail: `issac96@cis.udel.edu`
WWW: `http://www.inf.ethz.ch/ISSAC96/ISSAC96.html`

CALISCE '96

Third International Conference on Computer Aided Learning and Instruction in Science and Engineering, Donostia — San Sebastian, Spanien, 29.–31. Juli 1996

Arbeiten sind einzureichen bis:
15. Dezember 1995

Kontakt:
I. Fernandez-Castro
Informatika Fakultatea 649 P.K.
E-20080 Donostia, Spanien
Tel.: (+ 34) (43) 21 80 00
Fax: (+ 34) (43) 21 93 06
E-mail: `calisce96@si.ehu.es`
WWW: `http://www.sc.ehu.es/calisce96.html`