Universität Würzburg
Institut für Informatik
Research Report Series

Künstliche Neuronale Netze für die Wegoptimierung in ATG Leiterplattentestern

K. Leibnitz* **K. Tutschku*** **U. Rothaug‡**

Report No. 146 Juli 1996

* *Bayerische Julius Maximilians Universität Würzburg*
Institut für Informatik, Lehrstuhl für Informatik III
Am Hubland, 97074 Würzburg, Germany
Tel.: +49-931-8885513, Fax: +49-931-8884601
e-mail: {leibnitz,tutschku}@informatik.uni-wuerzburg.de

‡ *ATG Test Systems GmbH*
Zum Schlag 3, 97877 Wertheim–Reicholzheim
Tel.: +49-9342-291-0, Fax.: +49-9342-39510
e-mail: uwe@atg.uucp

Zusammenfassung:

In diesem Bericht werden die Ergebnisse der Implementation eines neuentwickelten Wegoptimierungsalgorithmus Leiterplattentestsystemen der Firma ATG vorgestellt. Der verwendete neuronale Algorithmus baut auf selbstorganisierenden Karten vom Typ *Flexmap* [Fri91] auf und ist in der Lage das *Traveling Salesman Problem (TSP)* in nahezu linearer Zeit zu lösen. Der Bericht beschreibt sowohl die Erweiterung des Flexmap Algorithmus, als auch die nötigen Vorverarbeitungsschritte um das Verfahren in der Praxis anzuwenden. Zusätzlich wird die Laufzeit des neuronalen Verfahrens mit der von konventionellen Heuristiken zur Wegsuche verglichen.

1. Einleitung

Gedruckte Leiterplatten, sog. „*Printed Circuits Boards*“ *(PCB)*, bilden die Grundlage für die Serienfertigung moderner elektronischer Geräte. Leiterplatten dienen dabei im wesentlichen zwei Zwecken. Zum einen bilden sie die Montageoberfläche für die elektronischen Bauteile des Gerätes und zum anderen werden durch leitende Verbindungen, den sogenannten „Leiterbahnen“, auf der nichtleitenden Oberfläche der Platte, die notwendige elektrische Verschaltung der Bauteile realisiert. Bei der Herstellung von Leiterplatten können jedoch Fehler an den Leiterbahnen, nicht völlig ausgeschlossen werden. Die Bestückung einer schadhaften Platine würde zu einem defekten Gerät führen und dem Hersteller einen Verlust einbringen. Deshalb ist es notwendig, so frühzeitig wie möglich im Produktionsprozeß fehlerhafte Leiterplatten zu erkennen und auszusortieren. Die Firma *ATG Test Systems GmbH, Wertheim–Reicholzheim*, entwickelt Testgeräte, die mittels elektrischer Verfahren die Korrektheit der leitenden Verbindungen überprüfen. Alle nach Design durch Leiterbahnen miteinander verbundene Elemente einer Leiterplatte, wie z.B. *Surface Mounted Devices (SMD)* ($\hat{=}$ Testpunkte des elektrischen Testsystems) werden in der Fachterminologie *Netz* genannt. Der elektrische Test hat daher im wesentlichen zwei Aufgaben:

Verbindungstest: Der Verbindungstest prüft, ob zwischen allen Punkten eines Netzes eine elektrische Verbindung besteht.

Isolationstest: Aufgabe des Isolationstests ist es zu prüfen, ob verschiedene Netze miteinander kurzgeschlossen sind.

Die Testpunkte werden dabei mit elektrischen Sonden, sogenannten *Fingern* abgetastet, die sich auf mehreren Schienen *(Rails)* unabhängig voneinander bewegen können.

Ein wichtiges Entscheidungskriterium für den Kauf eines Testsystems ist die Laufzeit die für einen Test benötigt wird. Diese Zeitdauer hängt direkt von der Anzahl der Testpunkte ab. Die kürzeste Testzeit kann nur durch Anfahren der Testpunkte in optimaler Reihenfolge, d.h. minimale Fahrweglänge der Finger, erreicht werden. Dieses Problem der Fahrwegoptimierung wird in der Informatik als *Traveling Salesman Problem (TSP)* bezeichnet. Von dieser Problemklasse ist jedoch bekannt, daß sie NP-vollständig ist und deshalb im *worst case* $O(2^N)$ Schritte zur Berechnung der optimalen Lösung benötigt.

Ein Kunde der Fa. ATG Test Systems, die Fa. Hitachi in Tokio, hat spezielle Produkte für den elektrischen Test, sogenannte MCM (= Multi Chip Module). Dies sind Träger von mehreren Chips in Mainframe Rechnern und enthalten bis 250.000 Testpunkte. Erste Versuche mit konventionellen Wegoptimierungsalgorithmen, bei denen die Anzahl der Testpunkte quadratisch einen Anstieg der Rechenzeit des Algorithmus bewirkt, resultierten in Rechenzeiten von über 20 Stunden. Hitachi stellte aber die Vorbedingung an ATG, daß die Optimierung innerhalb einer Arbeitsschicht eines Arbeiters von Hitachi ($\hat{=}$ 8h) beendet sein muß, da dieser Arbeiter den ganzen Vorgang überwachen muß.

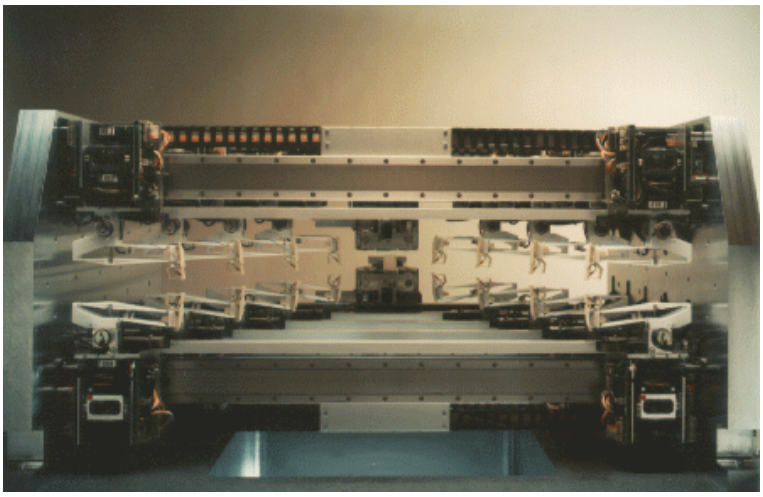


Abbildung 1: Außenansicht ATG Fingertester A2/16

Diese strenge Vorschrift ist deshalb entstanden, weil die internen Herstellkosten eines einzigen MCM Produktes je nach Produkt zwischen DM 80.000.– und DM 300.000,– betragen. Falsche Datenüberarbeitung kann aber zur Zerstörung der sehr empfindlichen Produktoberfläche führen, was nicht mehr reparabel ist.

In der Zusammenarbeit des Lehrstuhls für Informatik III der Universität Würzburg mit der ATG Test Systems GmbH sollte nun ein effektives und praktikables Verfahren zur Wegeoptimierung des Multi-Head Systems ATG Fingertester mit 16 Testköpfen, entwickelt werden. Aus oben aufgeführten Gründen wurde ein neuronaler Algorithmus, der eine suboptimale Lösung für die Wegsuche in akzeptabler findet, implementiert. Der neue Algorithmus wird im folgenden als *Modflex* bezeichnet. Es stellt ein Verfahren zur Lösung von TSP mittels selbstorganisierender neuronaler Karten dar. Der Algorithmus basiert auf *Flexmap* von B. Fritzke [Fri91]. Das ursprüngliche Verfahren wurde jedoch an einigen Stellen derart verändert, daß große Datenmengen besser zu verarbeiten sind. Eine genaue Beschreibung der Algorithmen *Modflex* und *Flexmap* wurde bereits in [RYL94] veröffentlicht.

In diesem Bericht soll in Kapitel 2 zuerst auf die Verbesserungen an *Modflex* eingegangen werden. Im Anschluß daran, in Kapitel 3, die Routinen des Programms `modatgtest`, einer angepassten Version von *Modflex* an die *ATG Testersoftware*, beschrieben. Den Abschluß bildet Kapitel 5, welches einen Ausblick auf mögliche Erweiterungen gibt. Im Anhang A werden die im Rahmen dieser Zusammenarbeit erstellten Programme kurz beschrieben.

2. Modflex – Modifiziertes Flexmap

Der *Modflex* Algorithmus stellt einen neuronalen Ansatz zur Optimierung der Weglänge bei einem TSP Problem dar. *Modflex* ist eine Weiterentwicklung von *Flexmap* von B. Fritzke [Fri91] und stellt ebenfalls ein neuronales Verfahren mit einer wachsenden Zellstruktur dar.

Eine ringförmige Kette von Neuronen passt sich im Laufe des „Lernvorgangs“ an die Struktur des Eingaberaumes mit den Städten an. Die Neuronenpositionen in der Eingabeebene entsprechen nach dem Lernen den Positionen der Städte. Durch die ringförmige Nachbarschaftsbeziehung der Neuronen wird vom Algorithmus eine Tour, die alle Städte durchläuft, erzeugt. Eine wesentliche Eigenschaft des Algorithmus ist es, daß die Anzahl benötigter Neuronen mit der Zeit zunimmt. Man spricht in diesem Fall von einer wachsenden neuronalen Struktur. Neuronen werden sukzessive in die Kette eingefügt, um eine Zuordnung eines nächsten Neurons *(Best Matching Unit, BMU)* zu jeder Stadt zu ermöglichen. Nachdem genau eine BMU für jede Stadt gefunden ist, endet der Algorithmus.

Der Kernalgorithmus von *Modflex* ist in Abbildung 2 dargestellt. Im Gegensatz zu *Flexmap* werden hier die Referenzzähler der Neuronen betrachtet und nicht die *Hits*, d.h. die Anzahl wie oft ein Neuron als BMU für eine bestimmte Stadt bisher ausgewählt wurde. Das *Pinnen* des Neurons auf die Position der Stadt nach einer festen Anzahl von Hits wirkte sich beim *Flexmap* Verfahren negativ auf die Elastizität des Netzes aus und wurde deshalb entfernt.

2.1. Veränderungen am Algorithmus

Bei den Untersuchungen in [RYL94] wurden die Testversuche mit einer relativ kleinen Anzahl an Städten (544) betrachtet. Aus diesem Grund ist ein Nachteil des Algorithmus nicht aufgefallen. Bei größeren Datensätzen traten vermehrt *Crossovers*, d.h. Überschneidungen der Wege, auf, die sich danach nicht mehr auflösten. Dieser Nachteil konnte jedoch durch eine Veränderung des urspurünglichen *Modflex* Verfahrens behoben werden. Der Nachteil des alten Algorithmus lag an der festen lokalen Suchtiefe k. Dieser muß bei einer größeren Städteanzahl C angepasst werden.

Eine Anzahl C von Städten wird sequentiell ausgewählt (vgl. Abbildung 2, Zeile 7) und nach jeweils n Schritten wird ein neues Neuron erzeugt und in die Kette eingefügt. Die Suche nach der BMU in Zeile 8 erfolgt während den ersten C Iterationen immer global, da zu diesem Zeitpunkt noch keine Stadt einer BMU zugeordnet ist. Nach C Durchläufen sind bereits $\frac{C}{n}$ Neuronen zusätzlich erzeugt. Für $C \gg n$ reicht somit die bei Fritzke festgesetzte Suchtiefe von k nicht mehr aus. Aufgrund der im Laufe des Algorithmus abnehmenden Dynamik bei den Neuronenbewegungen sind nach ungefähr $2 \cdot C$ Schritten die Nachbarschaftsbeziehungen der Neuronen derart stabilisert, daß die

```
1  begin
2    Mische die Reihenfolge der Städte;
3    Initialisiere ein Dreieck mit 3 Neuronen an beliebigen Positionen;
4    ready := 0;
5    while ready ≠ Anzahl StädteC do
6          for iteration := 0 to Iterationsschritten do
7              Wähle nächste Stadt c aus;
8              bmu^neu := Neuron mit geringstem Abstand zu c;
                 /*
                       Dabei wird die gesamte Neuronenkette durchsucht, falls c.bmu
9                      nicht gesetzt war, ansonsten werden nur ab c.bmu höchstens k
                       Neuronen in beide Richtungen betrachtet
                 */
10             if bmu^neu ≠ c.bmu
11               then bmu^neu.ref := bmu^neu.ref + 1;
12                    if bmu^neu.ref = 1
13                      then ready := ready + 1;
14                    fi
15                    if bmu^neu.ref = 2
16                      then ready := ready − 1;
17                    fi
18                    if c.bmu existierte bereits
19                      then c.bmu.ref := c.bmu.ref − 1;
20                           if c.bmu.ref = 1
21                             then ready := ready + 1;
22                           fi
23                           if c.bmu.ref = 0
24                             then ready := ready − 1;
25                           fi
26                    fi
27                    c.bmu := bmu_neu;
28             fi
29             Bewege bmu^neu um e_bmu nach c;
30             Bewege nächsten 2 Nachbarn von bmu^neu um e_nbr(i) in Richtung c;
31         od
32         e_worst := Neuron mit dem höchsten Wert ref;
33         Erzeuge ein neues Neuron neu an Position e_worst;
34         Verschiebe neu um 1/3 des Abstandes zum Vorgänger in Neuronenkette;
35         Verschiebe e_worst um 1/3 des Abstandes zum nachfolgenden Neuron;
36   od
37 end
```

Abbildung 2: Der *Modflex* Algorithmus

Veränderung einer BMU nur noch lokal stattfindet. Eine geringere konstante Suchtiefe ist nun ausreichend. Für einen beliebigen Iterationsschritt t und der aktuellen Anzahl an Neuronen N lautet daher die verbesserte Suchtiefe k:

$$k(t) = \begin{cases} \frac{N}{2} & t < n \\ k_0 + \frac{C}{n} - (N - \frac{C}{n}) & C \leq t < 2 \cdot C \\ k_0 & 2 \cdot C \leq t \end{cases} \quad (1)$$

Die Verwendung von Gleichung 1 ergab bessere Ergebnisse bei der Optimerung mit *Modflex* und *Flexmap*. Bei 100000 Testpunkten sind auf den ersten Blick keine großen Crossovers zu entdecken (siehe Abb. 3).

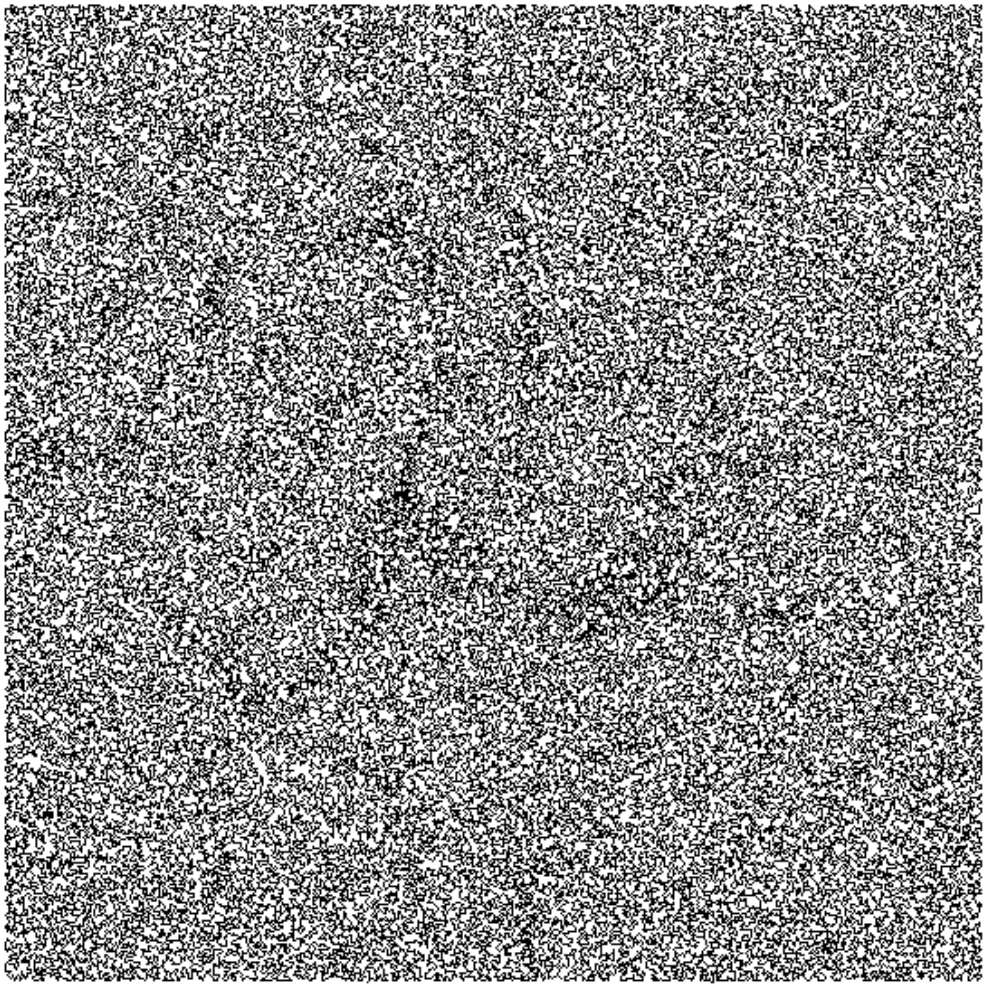


Abbildung 3: Ergebnis von Modflex mit 100000 Testpunkten

2.1.1. Konvergenzverhalten

Das Konvergenzverhalten von *Modflex* und *Flexmap* verbesserte sich ebenfalls. Der Einsatz von *Flexmap* bei großen Punktmengen und konstantem k Wert, wie er in [Fri91] beschrieben ist, lieferte oft kein Ergebnis. Nach Verwendung der angepassten k Funktion traten diese Fälle nicht mehr auf.

2.2. Testergebnisse

Die Auswirkungen des verbesserten k Wertes sollten analog zu den Ergebnissen auf Seite 12 in [RYL94] mit Testläufen unterschiedlicher Punktemengen untersucht werden. Es wurden dazu Datensätze mit 10000 bis 50000 zufällig verteilten Punkten erstellt und mit den gleichen Verfahren wie in [RYL94] optimiert. Die resultierenden Werte für Weglänge und benötigte Rechenzeit befinden sich in den folgenden Tabellen. Für den realen Einsatz von *Modflex* in *ATG Fingertestern* stellt dabei die Rechenzeit das wichtigere Kriterium dar. Ein eventuell längerer Weg kann dagegen eher in Kauf genommen werden. Alle Untersuchungen wurden mit dem Programm `modflex` (siehe auch Anhang A.1) auf einer *SUN SparcStation 20* unter dem Betriebssystem *Solaris 2.4* durchgeführt.

Städte	Neuronen	Weglänge	Zeit/s
10000	—	817144	113
20000	—	1151847	519
30000	—	1414011	1307
40000	—	1628489	2040
50000	—	1816876	3250

Tabelle 1: Testergebnisse mit dem Hungry Algorithmus

Die mit dem *Hungry* Verfahren ermittelten Werte (Tabelle 1) dienten als Referenz für die anderen beiden Verfahren, da diese Methode bei Fingertestern bereits implementiert war und auch bisher zur Optimierung eingesetzt wurde.

Städte	Neuronen	Weglänge	Zeit/s
10000	21269	830676	309
20000	42149	1251998	676
30000	63374	1536397	1100
40000	83127	1757131	2040
50000	99907	2025152	3227

Tabelle 2: Testergebnisse mit dem Flexmap Algorithmus

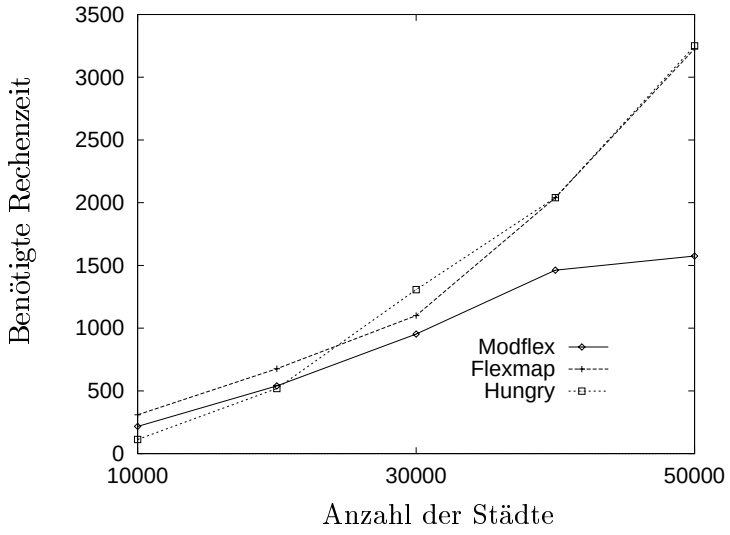
Bei den Ergebnissen des *Flexmap* Verfahrens (Tabelle 2) erhöhten sich die Wegstrecken

gegenüber der *Hungry* Methode jeweils um ungefähr 10%. Das lineare Rechenzeitverhalten von *Flexmap* kam mit der Zunahme der Städte besser zur Geltung. Ab etwa 30000 Testpunkten lieferte *Flexmap* ein schnelleres Ergebnis als das konventionelle Verfahren.

Städte	Neuronen	Weglänge	Zeit/s
10000	16746	823209	217
20000	32694	1170292	540
30000	49501	1442687	953
40000	66314	1667615	1462
50000	82469	1882222	1575

Tabelle 3: Testergebnisse mit dem Modflex Algorithmus

Die Weglängen bei *Modflex* waren etwa um 10% kürzer als bei *Flexmap* und nur minimal länger als bei *Hungry*. Die geringere Anzahl an erzeugten Neuronen gegenüber *Flexmap* schlug sich auch in der ungefähr um die Hälfte kürzeren Berechnungszeit nieder. Die graphischen Darstellungen (Abbildung 4(a) und (b)) der Tabellen 1, 2 und 3 verdeutlichen den Vorteil von *Modflex* gegenüber den anderen beiden Verfahren vor allem bei besonders großen Städtemengen. Damit sind die geplanten Anforderungen an den Algorithmus voll erfüllt.



(a) Benötigte Rechenzeit in Sekunden

(b) Berechnete Weglänge

Abbildung 4: Graphische Darstellungen der Rechenzeit bzw. Weglänge

3. Die AFT-Bibliothek

Das in Kapitel 2 erstellte Verfahren wurde aus zwei Gründen entwickelt:

1. Optimierung der Durchlaufzeit bei Verbindungstests *(Continuity Test)* mit dem *ATG Fingertester*.
2. Optimierung der Rechenzeit des Algorithmus.

Dazu war es notwendig das Program `modflex` an die bereits bestehende *ATG Fingertester (AFT) Parserbibliothek* [HR95] anzupassen. Als Grundgerüst diente das Demonstrationsprogramm `demo`, das alle erforderlichen Einleseroutinen der verschiedenen Parameterdateien durchführte und lediglich um die Optimierungsroutinen zu ergänzen war.

Eine globale Koordination der Optimierung über alle Rails des Boards erwies sich jedoch als nicht durchführbar, statt dessen wurde entschieden, eine Methode nach dem Prinzip *Divide–and–Conquer* zu verwenden, das Problem auf mehrere Teilprobleme aufzuteilen: *Modflex* sollte die Wege eines Fingerpaares jeweils innerhalb einzelner Rails optimieren und Railübergänge gesondert betrachten.

3.1. Veränderungen am Programm `modflex`

In der Entwicklungsphase von *Modflex* war es nötig, durch viele frei wählbare Kommandozeilenoptionen die einzelnen Parameter im Algorithmus steuern zu können. Die Auswirkungen der unterschiedlichen Parametereinstellungen konnten anhand der Graphikausgabe unter X11 mitverfolgt werden. Im Laufe der Untersuchungen in Abschnitt 2.2 bildeten sich einige sinnvolle Defaulteinstellungen für globale Variablen heraus, die im Quellcode als globalen Konstanten festgesetzt sind. Es folgt eine Auflistung der konfigurierbaren globalen Variablen und Schnittstellenfunktionen.

3.1.1. Globale Variablen

In der Datei `modflex.c` läßt sich die Optimierung mittels folgender Parameter beeinflussen:

`unsigned k0 = 50;` Setzt die lokale Tiefe bei der Suche nach der BMU auf 50 fest. Dieser Wert wird erst nach $2 \cdot C$ Iterationen angenommen (siehe auch Gleichung 1). Wird dieser Wert zu niedrig gesetzt, ergeben sich Crossovers, wie in Abschnitt 2.1 beschrieben. Bei einem zu großen Wert k_0 wird die Iteration

unnötig gebremst und der Geschwindigkeitsvorteil des Algorithmus zunichte gemacht.

`unsigned nDistr = 100;` Mit diesem Parameter wird die Anzahl der Iterationschritte festgelegt. Nach jeweils n Iterationen wird ein neues Neuron in die Neuronenkette eingefügt. Ein hoher Wert erzeugt weniger Neuronen, benötigt jedoch mehr Rechenzeit.

`double e_bmu = 0.03;` Mit der Variablen e_{bmu} wird die Stärke der Verschiebung der BMU in Richtung zugehörige Stadt bestimmt. (Zeile 31 in Abbildung 2). Die Funktion $e_{nbr}()$, die eine Anpassung der Nachbarn von BMU in der Neuronenkette vornimmt, wird ebenfalls durch e_{bmu} festgelegt. Ein zu hoher oder zu niedriger Wert stört die Elastizität des Netzes derart, daß eine Konvergenz nicht stattfinden kann.

3.1.2. Anwendungsschnittstellen

Die Headerdatei `modflex.h` beinhaltet die Schnittstellen zum Aufruf der Funktion

```
void Modflex(City *cities, unsigned city_count)
```

sowie die Typendeklarationen der benötigten Strukturen `City` und `Neuron`. Zum Zeitpunkt des Aufrufes der Funktion `Modflex()`, müssen die Städtedaten in dem Feld `cities` existieren und Speicher für das Feld alloziert sein. Der zweite Parameter `city_count` beinhaltet die Anzahl der übergebenen Städte.

Der weitere Inhalt der Datei ist in Abbildung 5 dargestellt. Bei Fingertestern, die nicht mit 2 Finger pro Rail arbeiten, muß in der Präprozessoranweisung

```
#define FINGER 2
```

der Wert entsprechend angepasst werden.

Die Struktur `Neuron` konnte von der ursprünglichen `modflex` Version übernommen werden. Die Datenstruktur `City`, die die Koordinaten der Städte speichert, musste um folgende beiden Felder ergänzt werden:

`Id`: beinhaltet die Identifizierungsnummer des Punktes aus den *ATF Netzdaten*
`Rail`: speichert den Rail, der den Testpunkt enthält.

```
#ifndef MODFLEX_H
#define MODFLEX_H

#define FINGER 2

#ifndef Min
#define Min(x, y) ((x) < (y)) ? (x) : (y)
#endif

#ifndef Max
#define Max(x, y) ((x) > (y)) ? (x) : (y)
#endif

#ifndef Sqr
#define Sqr(x)    ((x) * (x))
#endif

typedef struct Neuron
{
    struct Neuron *next, *prev;
    double XPos[FINGER], YPos[FINGER];
    unsigned RefCount;
} Neuron;

typedef struct City
{
    double XPos[FINGER], YPos[FINGER];
    unsigned Id[FINGER];
    unsigned Rail[FINGER];
    struct Neuron *Bmu;
} City;

/* Function interface */

void Modflex(City *, unsigned);

#endif
```

Abbildung 5: Die Headerdatei `modflex.h`

3.2. Das Programm `modatgtest`

Bei dem realen Einsatz von *Modflex* in einem Fingertester muß die in Abschnitt 3.1 angegebene Optimierungsroutine mit Paaren von Netzpunkten eines *ATF Files* aufgerufen werden. Die Einleseroutinen der Testpunkte, sowie der Testerparameter wurden aus der *AFT Parser Bibliothek* verwendet. Das Demonstartionsprogramm `demo` zur Library aus [HR95], das lediglich alle erforderlichen Parameterfiles einliest, fungierte dabei als Grundlage für das Programm `modatgtest`. Nach dem fehlerfreien Einlesen der Testerparameter wird mit dem Aufruf der Funktion

```
int Optimize_ATF(AtfData *atf, AFTData *aft)
```

der Optimierungsprozeß gestartet. Die Implementation dieser Funktion befindet sich in der Datei `cities.c`. Im folgenden soll nun eine kurze Beschreibung der verwendeten Routinen gegeben werden.

3.2.1. Die Funktion `int Optimize_ATF(AtfData *, AFTData *)`

Diese Funktion koordiniert die Optimierung der *ATF* Netzpunkte mit *Modflex*. Der Algorithmus in Pseudocode befindet sich in Abbildung 6.

Ein Board kann Testpunkte auf Vorder– oder Rückseite besitzen. Alle y Koordinaten auf der Rückseite des Boards werden in einem ersten Schritt um den maximalen y Wert der Vorderseite erhöht, um somit nur eine Seite bertachten zu müssen. Die sogenannten *Durchsteiger*, d.h. Punkte, die auf beiden Seiten des PCB Kontaktierflächen besitzen, werden jeweils einer von den beiden Seiten zugeteilt. Die Zuordnung erfolgt dabei indirekt proportional zu der Anzahl der auf beiden Seiten vorhandenen Testpunkte.

Das Einlesen aller *ATF Testpunkte* erfolgt netzweise, wobei jeder Punkt zuerst einer Oberflächenseite zugeordnet wird und danach seine *AFT Koordinaten* berechnet werden. Dies wird von der Funktion `TranslateCoordinates(NetzPktList *, View *)` durchgeführt, die für einen Netzpunkt, unter Berücksichtigung des physikalischen Offsets aus den *Sample*–Daten, sowie der Rotationsmatrix aus den *View*–Spezifikationen den zugehörigen *AFT Punkt* erzeugt. Dies ist nötig, weil die physikalischen Koordinaten auf dem Tester nicht den idealen Koordinaten des Design entsprechen. Die Nummer des Rails, in dem sich jeweils ein Punkt befindet, dient als Index bei der Zuordnung der *AFT Punkte* an eine *RailList*. Nachdem ein Netz vollständig eingelesen wurde, werden mit `DumpRailList()` aus sämtlichen Einträgen von *RailList* Punktepaare zu einer Struktur `City` zusammengefasst. Der Zusammenhang der verwendeten Datenstrukturen wird in Abbildung 7 verdeutlicht.

Da ein Netz sich nicht notwendigerweise nur innerhalb eines Rails befindet, wird es häufig vorkommen, daß eine `City` aus zwei *ATF Punkten* besteht, die in verschiedenen

```
1  begin
2    RailCount := Anzahl der Rails des Testers;
3    SideOffset := Maximaler y Wert auf der Vorderseite des Boards;
4    /*
       Dieser Wert wird zu allen y Koordinaten auf der Rückseite des
5      Boards
       addiert, um somit nur eine Testseite betrachten zu müssen.
6    */
7    for alle y Koordinaten der Rails auf der Rückseite des Boards do
8        Erhöhe y Koordinate um SideOffset;
9    od
10   RailList[] := Array von ATF Punkten der Größe RailCount ;
11   CityCount[] := Array von Integer Werten der Größe 2^RailCount;
12   cities[] := Array von City ebenfalls der Größe 2^RailCount;
13   CountFixPkt() zählt die Anzahl der Punkte.
14   /*
       Durchsteiger können somit indirekt proportional zu der Anzahl der
15     Punkte auf der Oberfläche zugeordnet werden.
16   */
17   for netz := alle ATF Netze do
18       for punkt := alle ATF Punkte in netz do
19           if punkt ist ein Durchsteiger
20             then
21                 Ordne punkt einer Oberflächenseite zu;
22           fi
23           if punkt befindet sich auf der Rückseite des Boards
24             then
25                 Erhöhe y Koordinate von punkt um SideOffset;
26           fi
27           punkt := TranslateCoordinates(punkt);
28           railNumber := Nummer des Rails, das punkt enthält;
29           Füge punkt nach x Koordinate sortiert in RailList[railNumber] ein;
30       od
31       DumpRailList() erzeugt die cities[] aus den Punktepaaren in RailList[];
32       ClearRailList() löscht alle Eintragungen in RailList[];
33   od
34   for i := 0 to 2^RailCount do
35       if CityCount[i] > 2
36         then
37             Modflex(cities[i], CityCount[i]);
38       fi
39   od
40   MergeCities() bereitet Ausgabe im ATF Task Format vor;
41 end
```

Abbildung 6: Ablauf der Funktion Optimize_ATF()

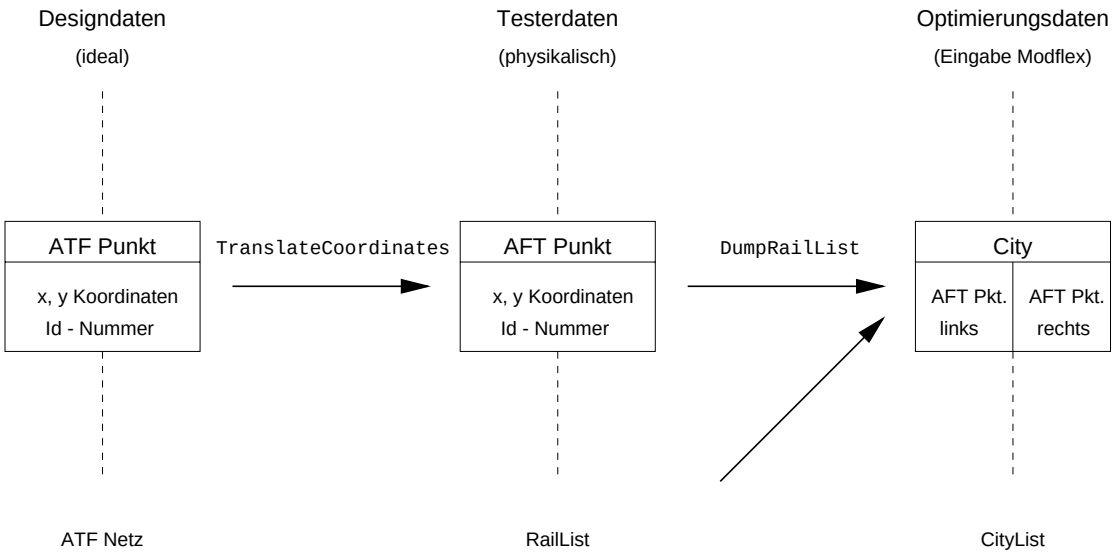


Abbildung 7: Zusammenhang der verwendeten Datenstrukturen

Rails liegen. Aus diesem Grund wird das Feld `cities`, das alle Punktepaare beinhaltet, ein Größe von $2^{RailCount}$ besitzen. Der jeweilige Index von `cities` wird durch bitweise UND-Verknüpfung der Binärdarstellungen der Railnummern ermittelt. Ein Beispiel für die Erstellung der `cities` für ein Netz mit den Punkten $1 \ldots 6$ ist in Abbildung 8 zu sehen.

Links im Bild ist die physikalische Lage der Punkte innerhalb der Rails dargestellt. Danach erfolgt eine Zuordnung der Punkte an die *RailList*. Aus diesen Listen werden die Punktepaare gebildet und in die entsprechenden *CityList* eingefügt. Die Felder in *CityList*, die nie besetzt werden, sind in der Skizze mit grauer Schraffur gekennzeichnet. Der genaue Ablauf bei der Erstellung der Städte aus *RailList* wird in Abschnitt 3.2.2 näher erläutert.

Nachdem alle Netze auf diese Weise eingelesen wurden, wird sequentiell für alle `cities` der *CityList* die Funktion `Modflex()` aufgerufen. Am Ende des Programms muß nur noch die optimierte *CityList* mittels `MergeCities()` im *AFT Task Format* ausgegeben werden und an den Fingertester weitergeleitet werden.

3.2.2. Die Funktion `void DumpRailList()`

Nachdem alle Punkte eines Netzes eingelesen wurden, wird in `Optimize_ATF()` (Abbildung 6, Zeile 32) die Funktion `DumpRailList()` aufgerufen. Die Aufgabe dieser Funktion ist es aus der *RailList* eines Netzes die Punktepaare zu bilden, die als Städte für *Modflex* in Frage kommen. Dabei werden die `cities` aus den Einträgen in der *RailList* folgen-

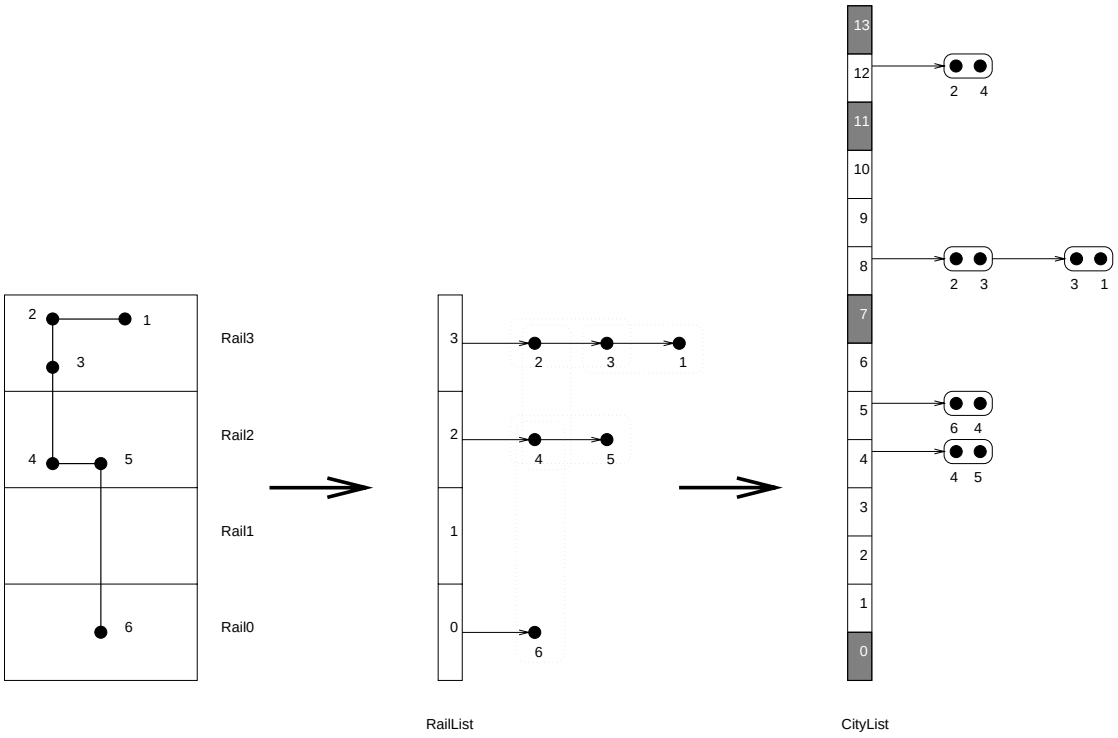


Abbildung 8: Beispiel für die Erstellung von `cities`

dermaßen zusammengestellt:

- jeder Punkt und sein Nachfolger innerhalb einer *RailList*
- jeder erste Punkt einer nichtleeren *RailList* und der erste Punkt der nächstgrößeren, nichtleeren *RailList*.

Der Algorithmus sieht dabei wie in Abbildung 9 aus.

3.2.3. Die Funktion `void MergeCities(AFTData *)`

Nach der Optimierung der *CityList* müssen die Daten aufbereitet werden, um dem Tester im *ATF Task Format* vorzuliegen. Die einzelnen Kommandos an die Testerfinger werden den Rails als *Tasks* übermittelt. Die Rails sollten dabei möglichst gleichmäßig ausgelastet werden, um einen Stillstand von Fingern zu vermeiden. Es bietet sich daher an, die Zuordnung der Tasks nicht sequentiell, sondern *interlaced* durchzuführen. Die Reihenfolgen innerhalb der `cities` müssen jedoch beibehalten werden, da ansonsten die vorherige Optimierung zunichte gemacht wird.

```
1  begin
2    for i := 0 to RailCount do
3      if RailList[i] nicht leer
4        then
5          for pkt := RailList[i].first to RailList[i].last.prev do
6            c := Erzeuge Stadt aus (pkt, pkt.next);
7            Füge c in cities[2^i] ein;
8          od
9          c̄ := Erzeuge Stadt aus (RailList[i].first, RailList[j].first);
10         Füge c̄ in cities[2^i + 2^j] ein ;
11         /*
12           wobei j der Index der nächstgrößeren, nichtleeren RailList ist.
13         */
14     fi
15   od
16 end
```

Abbildung 9: Ablauf der Funktion DumpRailList()

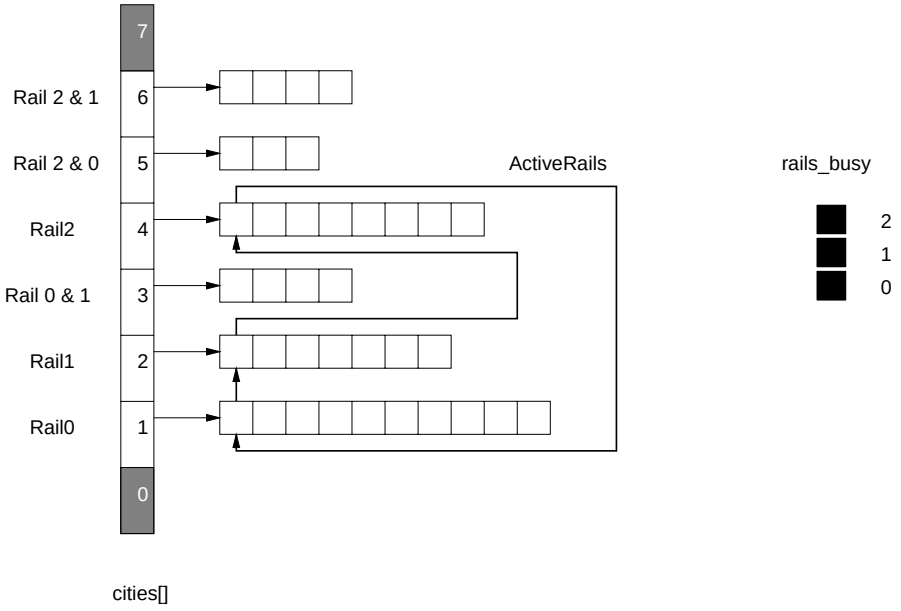


Abbildung 10: Anfangskonfiguration bei `MergeCities()`

```
1  begin
2    RailsLeft[] := Array von Integer der Größe 2^RailCount;
3    Initialisiere RailsLeft[] mit den Werten aus CityCount[];
4    for i := 0 to RailCount do
5        Füge cities[2^i] in die Ringliste ActiveRails ein;
6    od
7    Drucke Header des Taskfiles;
8    tmp := ActiveRails.first;
9    while ActiveRails nicht leer do
10           while tmp.offset ≠ RailsLeft[tmp.index] do
11               city := cities[tmp.index][tmp.offset];
12               Ausgabe von city;
13               tmp.offset := tmp.offset + 1;
14               tmp := tmp.next;
15           od
16           Lösche tmp aus ActiveRails;
17           max := Index der größten RailList[], dessen Rail nicht aktiv ist;
18           Füge cities[max] in ActiveRails ein;
19   od
20 end
```

Abbildung 11: Ablauf der Funktion MergeCities()

Es wird mit *ActiveRails* eine neue Ringliste eingeführt, die jeweils auf die aktuell zu bearbeitende `City` verweist. Initialisiert wird *ActiveRails* mit den `cities`, die komplett innerhalb eines Rails vorzufinden sind (siehe Abbildung 10). Die *ActiveRails* werden der Reihe nach bearbeitet bis eine von ihnen, z.B. `cities[j]` vollständig abgearbeitet ist. Danach wird `cities[j]` aus *ActiveRails* entfernt und `cities[k]` hinzugefügt, wobei `k` der Index des `cities` Feldes mit den meisten Einträgen ist, und nicht in aktiven Rails liegt. Dieser Vorgang wiederholt sich solange, bis die gesamte *CityList* bearbeitet ist.

4. Erste Resultate der Implementierung

Erste Tests des *Modflex* Algorithmus auf dem ATG Fingertester M2/16, der zur Zeit bei der Fa. Hitachi in Kanagawa bei Tokio installiert ist, brachten die folgenden Resultate im Vergleich des Hungry Algorithmus mit *Modflex*. Die Berechnungen wurden auf dem *Sun Sparc 20* Clone AXIL 20 unter dem Betriebssystem *Solaris 2.4* durchgeführt.

Als Testdaten wurden Daten von Protyp Keramik Boards verwendet, mit denen dann reale Tests auf dem Fingertester durchgeführt werden. In die unten aufgeführte Testzeit ist die Zeit für Messungen mit eingerechnet, die im Gegensatz zur Zeit, die für den Weg benötigt wird, allerdings nahezu keine Rolle spielt.

Produktname		XSH01	XSJ21
Testpunkte		89534	62215
Testnetze		30265	24510
Testzeit/s	*Hungry*	3790	2838
	Modflex	3809	2890
Rechenzeit/s	*Hungry*	7876	4271
	Modflex	3132	1763

Nach den ersten erfolgreichen Versuchen mit Prototypen, die noch relativ wenige Testpunkte enthalten, wurden auch Versuche mit realen MCM Produkten gestartet. Das erste Board hatte den Namen HAN11 mit 237000 Testpunkten bei 97000 Netzen. Leider zeigte sich, daß die Software noch Probleme mit den großen Datenmengen hat, was aber nicht auf den Algorithmus, sondern auf Speicherallokierungsfehler zurückzuführen ist. Das Programm wird zur Zeit überarbeitet.

Aus den oben durchgeführten Tests läßt sich eindeutig der Vorteil von *Modflex* erkennen. Bei nahezu gleicher Testzeit wurde die Rechenzeit des Algorithmus drastisch reduziert. Es lassen sich aber auch Probleme erkennen, deren Beseitigung auch zu einer Reduzierung der Weglänge und damit der Testzeit führen würde. Darauf soll im nächsten Kapitel eingegangen werden.

5. Erweiterungsmöglichkeiten

Ein großes Problem bei der Optimierung stellt sich derzeit in der `MergeCities()` Routine. Der Grad an Parallelisierung bei der Ausgabe nimmt im Lauf der Zeit stetig ab und es müssen immer mehr Testfinger untätig stillstehen. Dies liegt darin begründet, daß die gesamte *CityList* nicht gleichmäßig mit Testpunkten belegt ist. Während ein Feld in *CityList* abgearbeitet wird, blockiert es mindestens ein Rail. Dies schränkt die Anzahl der möglichen Kombinationen an nichtaktiven Rails für die Auswahl einer nächsten *CityList*

ein. Da dieses Problem bei der Resourcenverwaltung der Betriebssysteme in ähnlicher Form auftritt, können vielleicht z.B. Verfahren aus dem Scheduling von Prozessen eingesetzt werden.

Eine weiterer Nachteil könnte in der Aufteilung der Optimierung auf mehrere kleinere Teilprobleme liegen. Die Datenmengen werden nicht so groß sein, wie bei der Betrachtung des gesamten Boards und der Vorteil von *Modflex* gegenüber dem *Hungry* Verfahren wird kaum noch ins Gewicht fallen. Der *Hungry* Algorithmus wies bei kleinen Datensätzen besseres Rechenzeitverhalten auf als die neuronalen Methoden. Hier sind ebenfalls Verbesserungen möglich, um eine globale Wegoptimierung zu realisieren.

Zusammenfassend betrachtet, stellt *Modflex* unserer Meinung nach einen praktikablen Ansatz für die Wegoptimierung innerhalb eines *ATG Fingertesters* dar. Bei der Verwendung großer Datenmengen zeichnet sich ein deutlicher Gewinn an Rechenzeit gegenüber konventionellen Verfahren ab. Nach der Beseitigung oben aufgeführter Problemstellen, erscheint es möglich zu sein, den Verbindungstest in wesentlich kürzerer Zeit als bisher durchführen zu können.

A. Entwickelte Programme

A.1. Das Programm `modflex`

A.1.1. Technische Daten

Das Programm `modflex` wurde in ANSI C mit dem *GNU gcc* der *Free Software Foundation* entwickelt und auf folgenden Systemen erfolgreich getestet:

- SUN SparcStation 20 unter Solaris 2.4
- DEC Alpha 3000 AXP unter OSF/1 3.0
- Intel PC486 unter Linux 1.2.11

Die Graphikroutinen wurden unter X11 mit dem *XToolkit* und dem *Athena Widget Set* implementiert.

A.1.2. Format der Eingabedatei

Die Städtedaten in der Eingabedatei werden wie folgt eingelesen:

```
city_0[0] city_0[1] ... city_0[RailCount - 1]
city_1[0] city_1[1] ... city_1[RailCount - 1]
...
city_n[0] city_n[1] ... city_n[RailCount - 1]
```

Als Trennzeichen zwischen den einzelnen Koordinatenwerten können beliebige *Whitespace* Zeichen, wie Leerzeichen, Tabulator oder Zeilenumbruch fungieren.

A.1.3. Kommandozeilenoptionen

Eine Liste mit den zulässigen Parametern wird mit `modflex -h` ausgegeben:

```
modflex (version 2.0)
a self organizing neural map for optimization
Kenji Leibnitz, Jan. 1995
```

```
usage:  modflex [switches]

-F [file]       use input data file
-e [num]        value of ebmu           (default: 0.030000)
-f [num]        number of fingers       (default: 1)
-h              this help page
-k [num]        local search depth      (default: 50)
-l [file]       use logfile             (default: stderr)
-m [num]        use method:             (default: 0)
                0 = modflex, 1 = flexmap, 2 = hungry
-n [num]        distribution steps      (default: 100)
-q              quiet mode: don't use X11 graphics
-r [num]        generate random distribution
-s [num]        save intermediate results
-u [num]        update graphics         (default: 10)
-v [file]       view data file only
```

Für eine detailliertere Beschreibung der Kommandozeilenparameter wird auf die Manual Page verwiesen, es soll an dieser Stelle nur noch auf die Menüpunkte der interaktiven Version eingegangen werden. Der Aufruf von `modflex` mit einem gültigen Datenfile öffnet ein Fenster, das in Abbildung 12 dargestellt ist.

Als Menüpunkte stehen dem Benutzer danach folgende Kommandos zur Verfügung:

quit — beendet die Optimierung und bewirkt eine Rückkehr auf Betriebssystemebene.

length — liefert die gesamte Weglänge des Netzes zurück. Diese wird über die euklidische Distanz berechnet.

run — startet den Optimierungsprozeß mit dem jeweils ausgewählten Verfahren.

shuffle — ordnet die Städte in zufälliger Reihenfolge an.

modflex — bewirkt eine Änderung des Optimierungsverfahrens. Durch Anklicken dieses Menüpunktes wird die aktive Auswahl in folgender Reihenfolge geändert:
modflex → flexmap → hungry

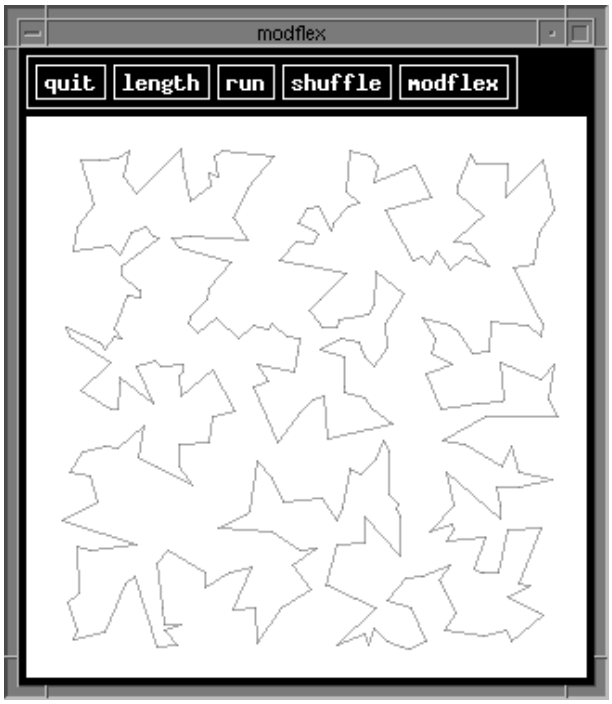


Abbildung 12: Screenshot von `modflex`

A.2. Das Programm `modatgtest`

Das Programm `modatgtest` wurde ebenfalls unter den in Anhang A.1.1 angegebenen Bedingungen implementiert. Der Aufruf von `modatgtest` muß in folgender Form erfolgen:

`modatgtest <tester_name> <board_name>`

Alle *AFT* Parameterdateien werden dann in den Verzeichnissen `./BOARDS/<tester_name>` bzw. `./BOARDS/<board_name>` erwartet.

Literatur

[Fri91] Bernd Fritzke. Flexmap – a neural network for the travelling salesman problem with linear time and space complexity. Singapore, 1991. IJCNN.

[HR95] Claus-Peter Hübner-Rauch. *Struktur- und Schnittstellenbeschreibung für AFT-Format-Parser*. Softwarebüro Hübner-Rauch, Uissigheim, März 1995.

[RYL94] Uwe Rothaug, Eugene Yanenko, and Kenji Leibnitz. Artificial neural networks used for way optimization in mutli–head systems in application to electrical flying probe testers. *Fachberichte Informatik*, 88, September 1994.

Verantwortlich: Die Vorstände des Institutes für Informatik.

[90] U. Hertrampf. *On Simple Closure Properties of #P.* Oktober 1994.

[91] H. Vollmer und K. W. Wagner. *Recursion Theoretic Characterizations of Complexity Classes of Counting Functions.* November 1994.

[92] U. Hinsberger und R. Kolla. *Optimal Technology Mapping for Single Output Cells.* November 1994.

[93] W. Nöth und R. Kolla. *Optimal Synthesis of Fanoutfree Functions.* November 1994.

[94] M. Mittler und R. Müller. *Sojourn Time Distribution of the Asymmetric $M/M/1//N$ – System with LCFS-PR Service.* November 1994.

[95] M. Ritter. *Performance Analysis of the Dual Cell Spacer in ATM Systems.* November 1994.

[96] M. Beaudry. *Recognition of Nonregular Languages by Finite Groupoids.* Dezember 1994.

[97] O. Rose und M. Ritter. *A New Approach for the Dimensioning of Policing Functions for MPEG-Video Sources in ATM-Systems.* Januar 1995.

[98] T. Dabs und J. Schoof. *A Graphical User Interface For Genetic Algorithms.* Februar 1995.

[99] M. R. Frater und O. Rose. *Cell Loss Analysis of Broadband Switching Systems Carrying VBR Video.* Februar 1995.

[100] U. Hertrampf, H. Vollmer und K. W. Wagner. *On the Power of Number-Theoretic Operations with Respect to Counting.* Januar 1995.

[101] O. Rose. *Statistical Properties of MPEG Video Traffic and their Impact on Traffic Modeling in ATM Systems.* Februar 1995.

[102] M. Mittler und R. Müller. *Moment Approximation in Product Form Queueing Networks.* Februar 1995.

[103] D. Rooß und K. W. Wagner. *On the Power of Bio-Computers.* Februar 1995.

[104] N. Gerlich und M. Tangemann. *Towards a Channel Allocation Scheme for SDMA-based Mobile Communication Systems.* Februar 1995.

[105] A. Schömig und M. Kahnt. *Vergleich zweier Analysemethoden zur Leistungsbewertung von Kanban Systemen.* Februar 1995.

[106] M. Mittler, M. Purm und O. Gihr. *Set Management: Synchronization of Prefabricated Parts before Assembly.* März 1995.

[107] A. Schömig und M. Mittler. *Autocorrelation of Cycle Times in Semiconductor Manufacturing Systems.* März 1995.

[108] A. Schömig und M. Kahnt. *Performance Modelling of Pull Manufacturing Systems with Batch Servers and Assembly-like Structure.* März 1995.

[109] M. Mittler, N. Gerlich und A. Schömig. *Reducing the Variance of Cycle Times in Semiconductor Manufacturing Systems.* April 1995.

[110] A. Schömig und M. Kahnt. *A note on the Application of Marie's Method for Queueing Networks with Batch Servers.* April 1995.

[111] F. Puppe, M. Daniel und G. Seidel. *Qualifizierende Arbeitsgestaltung mit tutoriellen Expertensystemen für technische Diagnoseaufgaben.* April 1995.

[112] G. Buntrock, und G. Niemann. *Weak Growing Context-Sensitive Grammars.* Mai 1995.

[113] J. García and M. Ritter. *Determination of Traffic Parameters for VPs Carrying Delay-Sensitive Traffic.* Mai 1995.

[114] M. Ritter. *Steady-State Analysis of the Rate-Based Congestion Control Mechanism for ABR Services in ATM Networks.* Mai 1995.

[115] H. Graefe. *Konzepte für ein zuverlässiges Message-Passing-System auf der Basis von UDP.* Mai 1995.

[116] A. Schömig und H. Rau. *A Petri Net Approach for the Performance Analysis of Business Processes.* Mai 1995.

[117] K. Verbarg. *Approximate Center Points in Dense Point Sets.* Mai 1995.

[118] K. Tutschku. *Recurrent Multilayer Perceptrons for Identification and Control: The Road to Applications.* Juni 1995.

[119] U. Rhein-Desel. *Eine „Übersicht" über medizinische Informationssysteme: Krankenhausinformationssysteme, Patientenaktensysteme und Kritiksysteme.* Juli 1995.

[120] O. Rose. *Simple and Efficient Models for Variable Bit Rate MPEG Video Traffic.* Juli 1995.

[121] A. Schömig. *On Transfer Blocking and Minimal Blocking in Serial Manufacturing Systems — The Impact of Buffer Allocation.* Juli 1995.

[122] Th. Fritsch, K. Tutschku und K. Leibnitz. *Field Strength Prediction by Ray-Tracing for Adaptive Base Station Positioning in Mobile Communication Networks.* August 1995.

[123] R. V. Book, H. Vollmer und K. W. Wagner. *On Type-2 Probabilistic Quantifiers.* August 1995.

[124] M. Mittler, N. Gerlich, A. Schömig. *On Cycle Times and Interdeparture Times in Semiconductor Manufacturing.* September 1995.

[125] J. Wolff von Gudenberg. *Hardware Support for Interval Arithmetic - Extended Version.* Oktober 1995.

[126] M. Mittler, T. Ono-Tesfaye, A. Schömig. *On the Approximation of Higher Moments in Open and Closed Fork/Join Primitives with Limited Buffers.* November 1995.

[127] M. Mittler, C. Kern. *Discrete-Time Approximation of the Machine Repairman Model with Generally Distributed Failure, Repair, and Walking Times.* November 1995.

[128] N. Gerlich. *A Toolkit of Octave Functions for Discrete-Time Analysis of Queuing Systems.* Dezember 1995.

[129] M. Ritter. *Network Buffer Requirements of the Rate-Based Control Mechanism for ABR Services.* Dezember 1995.

[130] M. Wolfrath. *Results on Fat Objects with a Low Intersection Proportion.* Dezember 1995.

[131] S. O. Krumke and J. Valenta. *Finding Tree–2–Spanners.* Dezember 1995.

[132] U. Hafner. *Asymmetric Coding in (m)-WFA Image Compression.* Dezember 1995.

[133] M. Ritter. *Analysis of a Rate-Based Control Policy with Delayed Feedback and Variable Bandwidth Availability.* January 1996.

[134] K. Tutschku and K. Leibnitz. *Fast Ray-Tracing for Field Strength Prediction in Cellular Mobile Network Planning.* January 1996.

[135] K. Verbarg and A. Hensel. *Hierarchical Motion Planning Using a Spatial Index.* January 1996.

[136] Y. Luo. *Distributed Implementation of PROLOG on Workstation Clusters.* February 1996.

[137] O. Rose. *Estimation of the Hurst Parameter of Long-Range Dependent Time Series.* February 1996.

[138] J. Albert, F. Räther, K. Patzner, J. Schoof, J. Zimmer. *Concepts For Optimizing Sinter Processes Using Evolutionary Algorithms.* February 1996.

[139] O. Karch. *A Sharper Complexity Bound for the Robot Localization Problem.* June 1996.

[140] H. Vollmer. *A Note on the Power of Quasipolynomial Size Circuits.* June 1996.

[141] M. Mittler. *Two-Moment Analysis of Alternative Tool Models with Random Breakdowns.* July 1996.

[142] P. Tran-Gia, M. Mandjes. *Modeling of customer retrial phenomenon in cellular mobile networks.* July 1996.

[143] P. Tran-Gia, N. Gerlich. *Impact of Customer Clustering on Mobile Network Performance.* July 1996.

[144] M. Mandjes, K. Tutschku. *Efficient call handling procedures in cellular mobile networks.* July 1996.

[145] N. Gerlich, P. Tran-Gia, K. Elsayed. *Performance Analysis of Link Carrying Capacity in CDMA Systems.* July 1996.

[146] K. Leibnitz, K. Tutschku, U. Rothaug. *Künstliche Neuronale Netze für die Wegoptimierung in ATG Leiterplattentestern.* Juli 1996.