ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

INSTITUT FÜR INFORMATIK

Algorithmentheorie
WS 2003/2004

Anwendung der gelernten Konzepte

im Schaltkreisentwurf:

eine Sammlung von Beispielen

Alejandro Czutro

Februar 2004

Abstract

In diesem Semester wurden in der Vorlesung zur Algorithmentheorie mehrere Konzepte vorgestellt, die bei der Suche nach effizienten Algorithmen sehr oft hilfreich sind, wie z.B Divide&Conquer, Randomisierung und dynamische Programmierung. Außerdem wurden mehrere Strukturen zur Implemetierung von Vorrangwarteschlangen vorgestellt, die bei der Verwaltung mehrerer Objekte des gleichen Typs große Effizienz versprechen. Beispiele sind Binomial Queues, Treaps und Fibonacci-Heaps. Es wurde über Greedy-Verfahren geredet und es wurden mehrere Graphenalgorithmen vorgestellt, die minimale Spannbäume, maximale Flüsse oder minimale Schnitte in Graphen berechnen. Bei so vielen verschiedenen behandelten Themen fragt man sich oft, ob diese ganze Sachen denn überhaupt nützlich sind. Der Zweck dieser Präsentation ist nicht, diese Themen noch ein Mal explizit zu behandeln, da dies in der Vorlesung und durch Übungen bereits zur Genüge geschehen ist, sondern anhand einiger ausgewählter Problemen aus dem Gebiet der Rechnerarchitektur zu zeigen, daß mehrere dieser Methoden in der Praxis durchaus eine Verwendung finden.

Keywords

Platzierung, Bin Packing, Routing, minimale Schnitte, dynamische Programmierung, Greedy-Verfahren, Kernighan-Lin, Stuck-at-Fehler, Entdeckende Testmuster, Kompatibilität von Testmustern, Event Driven Fault Simulation, Vorrangwarteschlangen

Kurze Übersicht

Teilprozesse des Hardwareentwurfs

Hardwaresynthese: besteht aus mehreren Teilprozessen:

- ◇ Algorithmische Ebene
- ◇ Register-Transfer Ebene
- ◇ Gatterebene
- ◇ Transistorebene
- ◇ Layoutebene

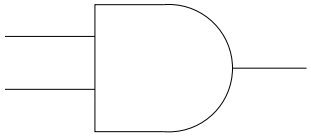
Hardwaretest: notwendig am Schluß des gesamten Syntheseprozesses, um sicher zu sein, daß die gebauten Teile das tun, was sie sollten.

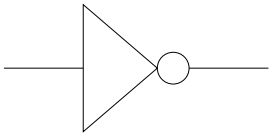
Die betrachteten Probleme

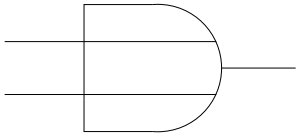
Hardwaresynthese: Platzierung und Routing

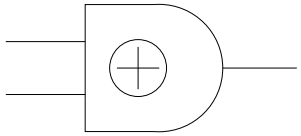
Hardwaretest: Kompatibilität von Testmustern und
Ereignisgesteuerte Fehlersimulation

Symbole für die Gatter

AND: 

NOT: 

OR: 

XOR: 

Platzierungs-

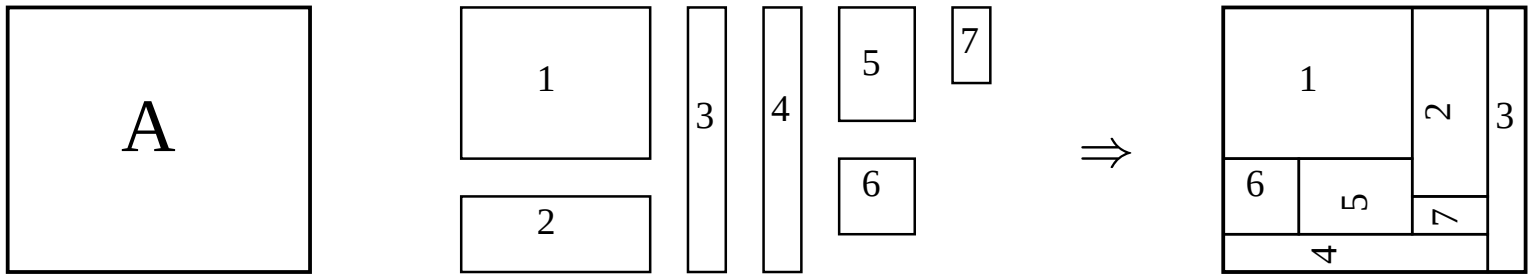
und Routingprobleme

Allgemeine Problemstellung

Gegeben: Eine Menge von fertigen Modulen $M_1, \ldots, M_n$, gegeben durch geometrische Figuren; und eine Fläche $A = a \times b$.

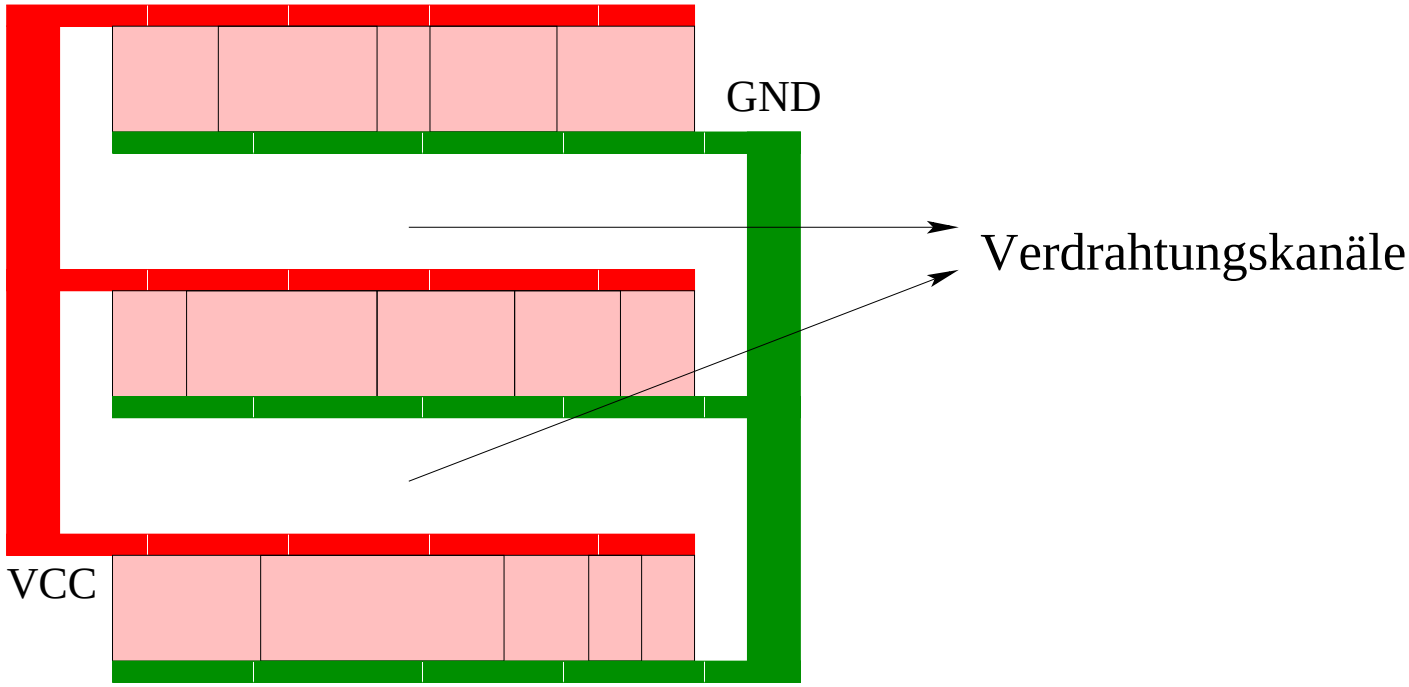
Gesucht: Die beste Möglichkeit, die M_i auf die Fläche zu legen, so daß alle M_i Platz finden und mehrere wichtige Randbedingungen eingehalten werden.

Beispiel:



Ein einfaches Platzierungsproblem

Problemstellung: Platzierung von Standardzellen, alle gleich hoch, verschiedene Breiten; beliebige Verdrahtung



Lösungsansatz

Lösung:

- ◇ Betrachte, statt der n Modulen $M_1, \ldots, M_n$, deren Breiten $m_1, \ldots, m_n$.
- ◇ Sei B die Breite jeder Zellenschiene.
- ◇ Arbeite mit normierten Breiten

$$\frac{m_1}{B}, \ldots, \frac{m_n}{B}.$$

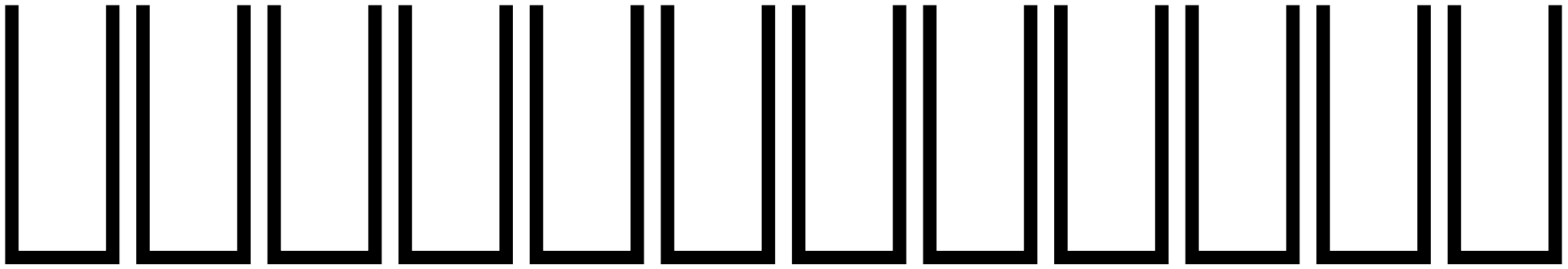
- ◇ Da die normierte Breite der Schienen 1 ist, liegt hier nichts anderes als ein `Bin-Packing`-Problem vor.

Bin Packing

Zur Erinnerung: Das `Bin-Packing`-Problem sucht die minimale Anzahl von Bins der Größe 1, die man braucht, um eine Reihe von Objekten der Größe ≤ 1 darin zu unterbringen.

Strategien: `Online`-Strategien: NF, FF, BF;
`Offline`-Strategien: FFD, BFD
Diese Verfahren sind `Greedy`-Algorithmen.

Beispiel: Gegeben: $0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8, 0.8, 0.3, 0.5, 0.4, 0.2$

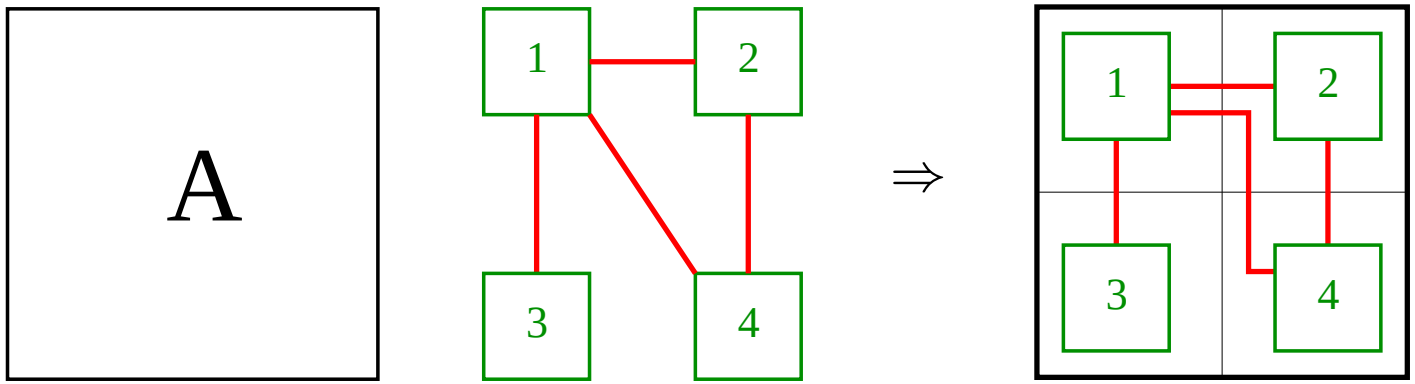


Ein einfaches Routingproblem

Gegeben: Eine Menge von fertigen Modulen $M_1, \ldots, M_n$, alle ungefähr gleich groß; Kosten $K_1, \ldots, K_r$ für die Verbindungen zwischen den Modulen und eine Fläche $A = a \times b$.

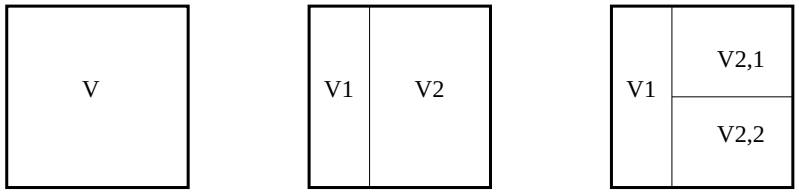
Gesucht: Unter der Annahme, daß eine Anordnung der M_i auf der Fläche gefunden wurde, ein Raster, so daß die Verdrahtungskosten minimal sind.

Beispiel:



Kosten

Wie sollen die entstandenen Kosten definiert werden?

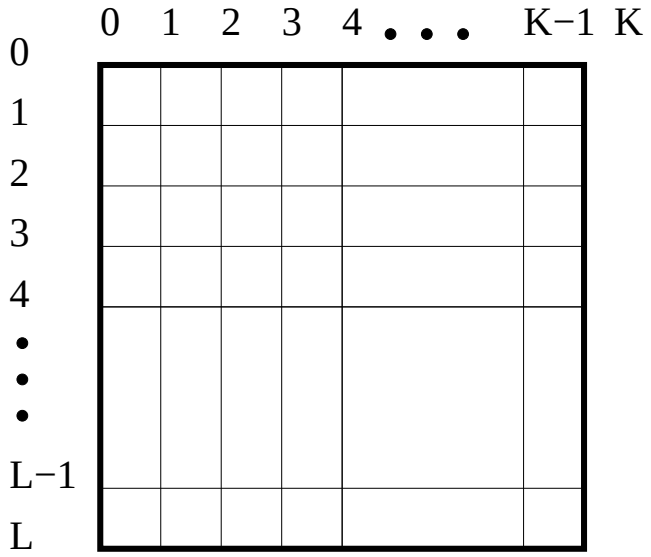


- ◇ Jede Verbindung zwischen zwei Modulen hat ein Gewicht (z.B. Breite des Verbindungsbuses).
- ◇ Seien V eine Menge von Modulen und E die Menge der Kanten zwischen diesen Modulen.
- ◇ Erzeugt der Raster eine Partition $V_1 \dot{\bigcup} V_2 = V$, so sind die Routingkosten bzgl. V gegeben durch

$$\mathrm{COST}(V) = \mathrm{COST}(V1) + \mathrm{COST}(V2) + \underbrace{\sum_{\substack{e := \{v_1, v_2\} \in E, \\ v_1 \in V_1, \\ v_2 \in V_2}} w(v_1, v_2)}_{=: \mathrm{W}(V_1, V_2)}$$

eine mögliche Lösung

◇ Führe Unterteilungspunkte ein:



Weiter sei $V_{i=i_1..i_2}^{j=j_1..j_2}$ die Fläche, die von den Unterteilungspunkten i_1 und i_2 in waagrechter Richtung und von den Unterteilungspunkten j_1 und j_2 in senkrechter Richtung begrenzt wird.

◇ Wähle zwei Parameter $M < K$ und $N < L$.

eine mögliche Lösung *ff*

- Die optimalen Kosten $C^*\left(V_{i=i_1..i_2}^{j=j_1..j_2}\right)$ der Partitionierung von $V_{i=i_1..i_2}^{j=j_1..j_2}$ sind gegeben durch $min\{\Phi, \Psi\}$ mit

$$\Phi := \min_{\mu=i_1+M,\ldots,i_2-M} \left\{ C^*\left(V_{i=i_1..\mu}^{j=j_1..j_2}\right) + C^*\left(V_{i=\mu..i_2}^{j=j_1..j_2}\right) + W\left(V_{i=i_1..\mu}^{j=j_1..j_2}, V_{i=\mu..i_2}^{j=j_1..j_2}\right) \right\}$$

$$\Psi := \min_{\nu=j_1+N,\ldots,j_2-N} \left\{ C^*\left(V_{i=i_1..i_2}^{j=j_1..\nu}\right) + C^*\left(V_{i=i_1..i_2}^{j=\nu..j_2}\right) + W\left(V_{i=i_1..i_2}^{j=j_1..\nu}, V_{i=i_1..i_2}^{j=\nu..j_2}\right) \right\}$$

Basisfall: Hat $V_{i=i_1..i_2}^{j=j_1..j_2}$ Breite M und Höhe N, so liegt das Problem des minimalen Schnittes vor.

Effiziente Berechnung der Rekursion: Dynamische Programmierung!

Dynamische Programmierung

Szenario: Die Lösung eines Problems setzt sich aus den Lösungen mehrerer kleineren Teilprobleme zusammen. Dabei muß auf einige dieser Teillösungen mehrmals zugegriffen werden.

Steigerung der Effizienz: Einmal berechnete Lösungen werden in einer Tabelle gespeichert.

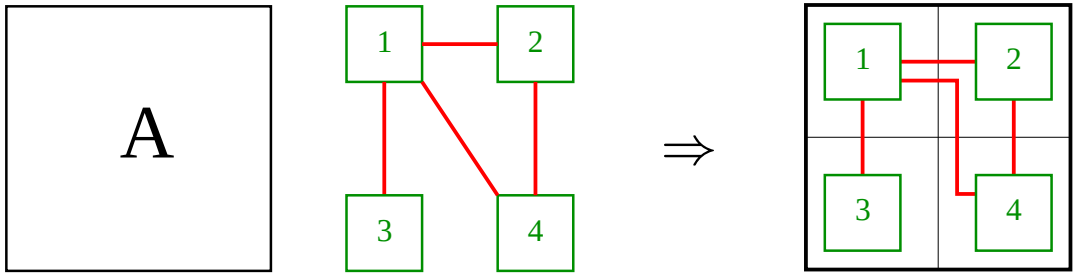
Genauer: Bestimme eine Menge T, die alle Teilprobleme enthält; finde eine Reihenfolge der Probleme $T_1, \ldots, T_l \in T$, so daß ein Teilproblem T_j nur von Teilproblemen T_i abhängt, wenn $i < j$.
Anschließend werden die Lösungen der Probleme $T_1, \ldots, T_l \in T$ der Reihe nach berechnet und jeweils gespeichert.

Ein echtes Platzierungsproblem

Gegeben: Eine Menge von fertigen Modulen $M_1, \ldots, M_n$, alle ungefähr gleich groß; Kosten $K_1, \ldots, K_r$ für die Verbindungen zwischen den Modulen und eine Fläche $A = a \times b$.

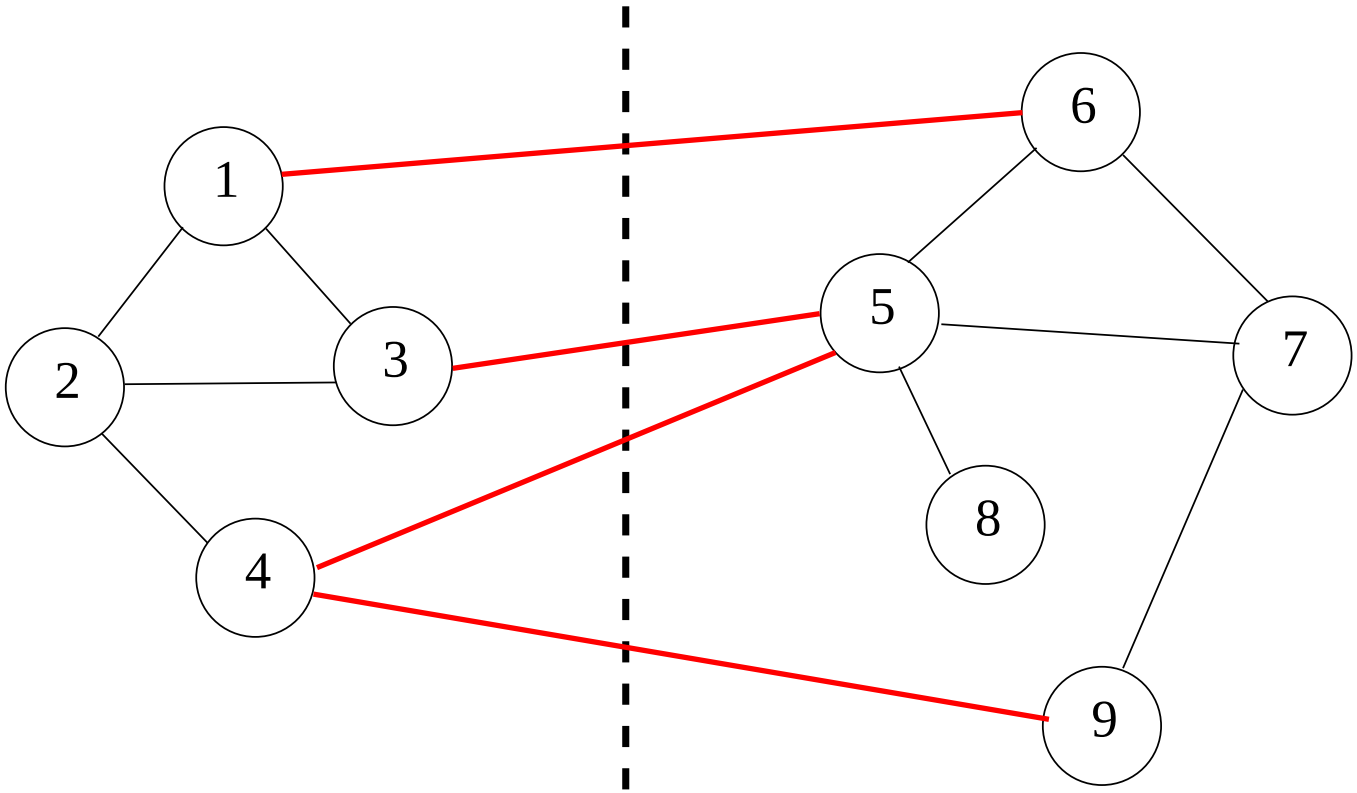
Gesucht: Eine Anordnung der M_i auf der Fläche, so daß alle M_i Platz finden; zusätzlich ein Raster, so daß die Verdrahtungskosten minimal sind.

Beispiel:



ein echtes Platzierungsproblem *ff*

Was ist der Unterschied zum vorigen Problem? Partition muß „symmetrisch“ sein:



Die Kernighan-Lin-Heuristik

Diese Heuristik wird verwendet, um einen minimalen Schnitt $V_1 \dot{\cup} V_2 = V$ in einem Graphen $\mathcal{G} := (V, E)$ zu finden, so daß $\left|\left|V_1\right| - \left|V_2\right|\right| \in \{0, 1\}$. Dieses Problem ist auch als `KL-Mincut`-Problem bekannt.

Eigenschaften: Die Heuristik ist deshalb interessant, weil das Problem schärfer ist als das `Mincut`-Problem aus der Vorlesung.

- ◇ Greedy-Verfahren
- ◇ Vertauscht zwei Knotenteilmengen „gleicher“ Mächtigkeit in jedem Durchlauf.
- ◇ Jeder Durchlauf besteht aus $\frac{|V|}{2}$ Schritten.

Die Kernighan-Lin-Heuristik *ff*

Ein Durchlauf: Gegeben ist bereits eine Partition $V_1 \dot{\cup} V_2 = V$, so daß V_1 und V_2 „gleich“ groß sind.
Es werden $\frac{|V|}{2}$ Schritte durchgeführt:

i) Ein Knoten aus V_1 und ein Knoten aus V_2 werden vertauscht, so daß die Kosten mit dieser Vertauschung minimiert werden.

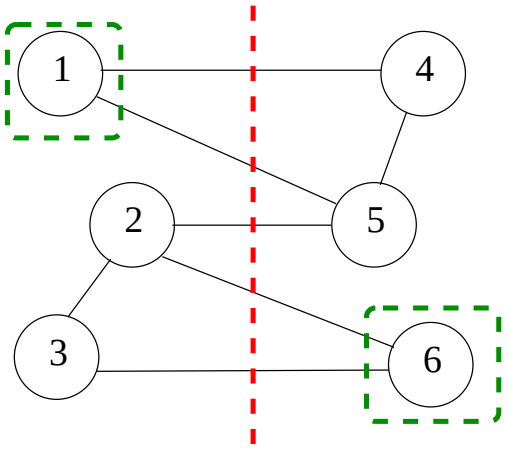
ii) Sperre die vertauschten Knoten, so daß sie in diesem Durchlauf nicht mehr vertauscht werden können.

iii) Für jedes Knotenpaar, das sich mit den übrigen Knoten bilden läßt, berechne den Gewinn, falls diese Knoten vertauscht werden.

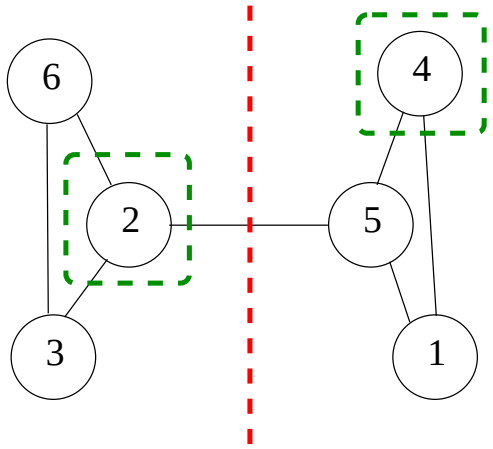
iv) Zurück zu i)

Die Kernighan-Lin-Heuristik *ff*

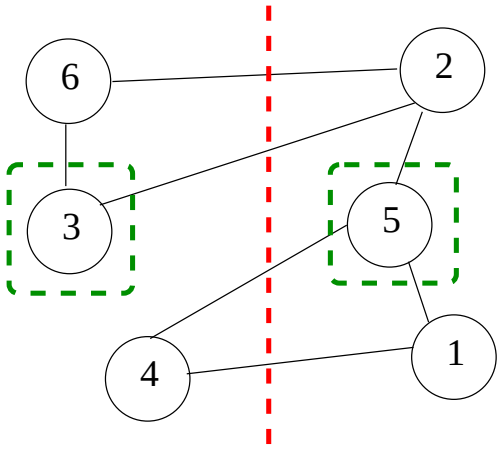
Beispiel eines Durchlaufs:



C = 5



C = 1



C = 4

Zwischen zwei Durchläufen:

- ◇ Den Schritt i bestimmen, ab dem keine Verbesserung der globalen Kosten mehr vorkommt.
- ◇ Die Schritte $1, \ldots, i-1$ dauerhaft machen.

Die Kernighan-Lin-Heuristik *ff*

Iterativer Ansatz:

- ◇ Start in einem beliebigen Zustand.
- ◇ Lösung des k-ten Durchlaufs ist der Startzustand des $(k+1)$-ten Durchlaufs.
- ◇ Abbruchkriterium: Keine Verbesserung der Kosten mehr möglich.

Hardwaretest:

zwei Probleme

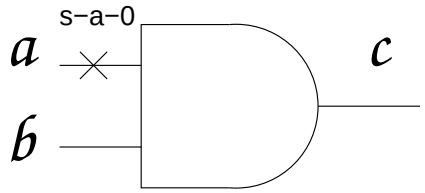
Testen: kurze Einführung

Stuck-at-Fehler

- Es gibt viele Modelle, um physikalische Defekte formal zu beschreiben.

- Das `Single-stuck-at-fault`-Modell ist das meist verwendete:
  - Gute Algorithmen sind bekannt.
  - Bei n Gattern gibt es nur $c \cdot n$ mögliche Fehlersituationen.

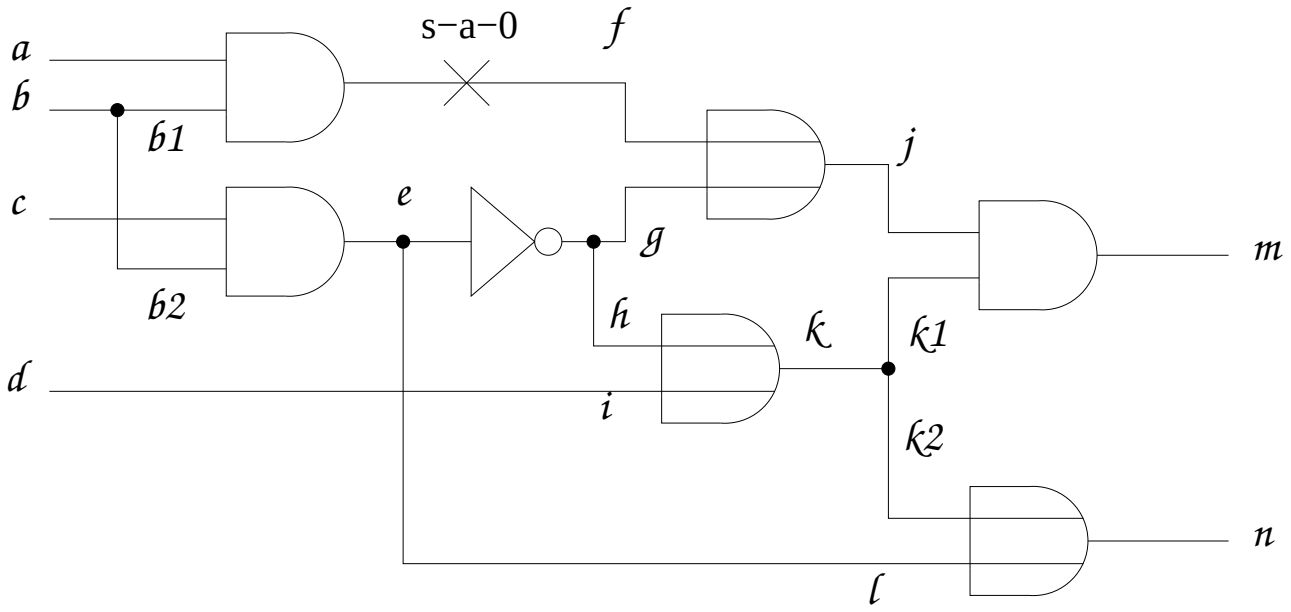
- Eine Leitung l hat einen `stuck-at-c`-Fehler $\left(\text{i.Z. } {}^{l}/_{c}\right)$, wenn die Leitung l konstant auf c gesetzt ist:



Entdeckung von Fehlern

entdeckendes Testmuster: Ein Testmuster (eine Belegung der Inputs des Schaltkreises) $\mathfrak{t} := t_1 t_2 \ldots t_n$ entdeckt einen Fehler $\mathfrak{f}$, wenn durch die Anwendung dieses Musters ein durch $\mathfrak{f}$ verursachter Effekt an einem Ausgang des Schaltkreises zu beobachten ist.

Beispiel: Der Test $\mathfrak{t} = 1111$ entdeckt den Fehler $^{f}/_{0}$:



Probleme beim Testen

High Fault Coverage: muß gewährleistet sein $\Rightarrow$ Für jeden Fehler muß man ein entdeckendes Muster finden und anwenden.

Aber: Bei 50 000 Leitungen sind das 100 000 Fehler bzw. anzuwendende Muster.

Die Testhardware ist i.d.R. langsamer als das `CUT` und jede Minute, die man am Testen verbringt ist sehr teuer.

Was soll man dann tun? Man müßte Muster finden, die in der Lage sind, mehrere Fehler `gleichzeitig` zu entdecken.

Problem der Testkompaktierung

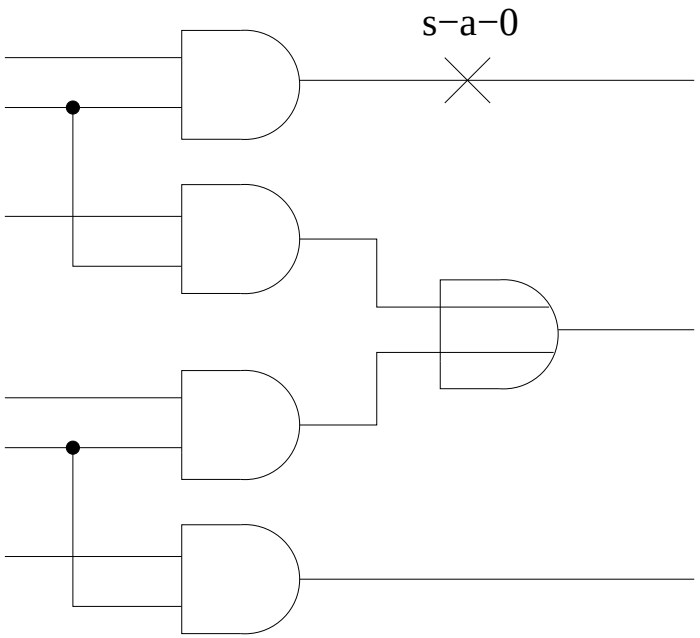
Gegeben: Eine Fehlermenge F und eine Testmustermenge T, so daß mit den Mustern aus T alle Fehler aus F entdeckt werden können.

Gesucht: Eine Testmenge T' mit $\left|T'\right| < \left|T\right|$, so daß mit T' alle Fehler aus F entdeckt werden.

Lösungsansatz

Tatsache: Die Ausgänge eines Schaltkreises hängen meistens von (viel) weniger Eingängen ab, als es insgesamt gibt:

Beispiel:



Das Testmuster $t = 11xxxx$ entdeckt den eingezeichneten `stuck-at-0`-Fehler.

Kompatible Testmuster

Definition: Zwei Testmuster $\mathfrak{t} := t_1 \ldots t_n$ und $\mathfrak{s} := s_1 \ldots s_n$ heißen

`kompatibel` $:\Leftrightarrow \quad \forall\, i \in \{1, \ldots, n\} \quad s_i \neq \bar{t}_i.$

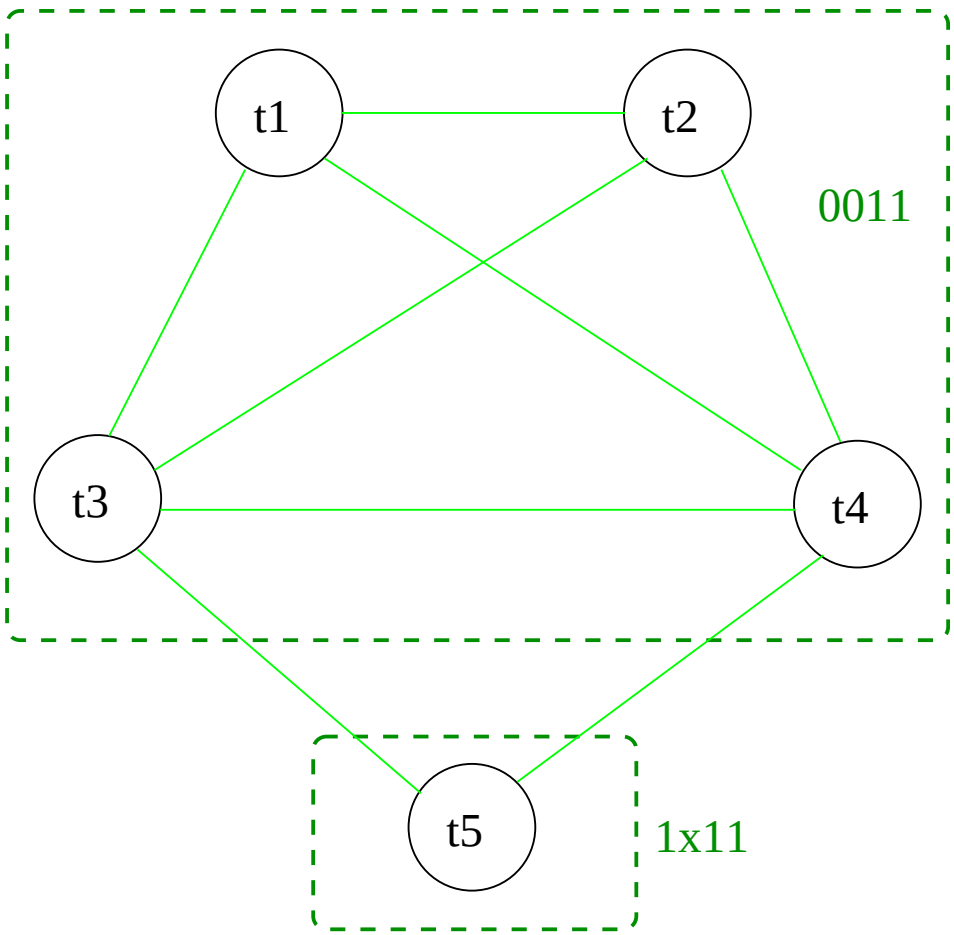
Beispiel: $10x$ ist kompatibel zu $1x0$, aber nicht zu $x1x$.

Bemerkung: Zu zwei kompatiblen Mustern $\mathfrak{t}_1$ und $\mathfrak{t}_2$ gibt es ein Muster $\mathfrak{t} := \mathfrak{t}_1 \bigcap \mathfrak{t}_2$, das alle Stellen spezifiziert, die auch $\mathfrak{t}_1$ oder $\mathfrak{t}_2$ spezifizieren, und zu beiden $\mathfrak{t}_i$ kompatibel ist.
Beispiel: Zu $1010xxx$ und $1xx0x10$ gibt es $1010x10$.

Korollar: Durch iterative Anwendung lassen sich auch (sehr) viele Testmuster durch ein einziges ersetzen, das zu ihnen allen kompatibel ist.

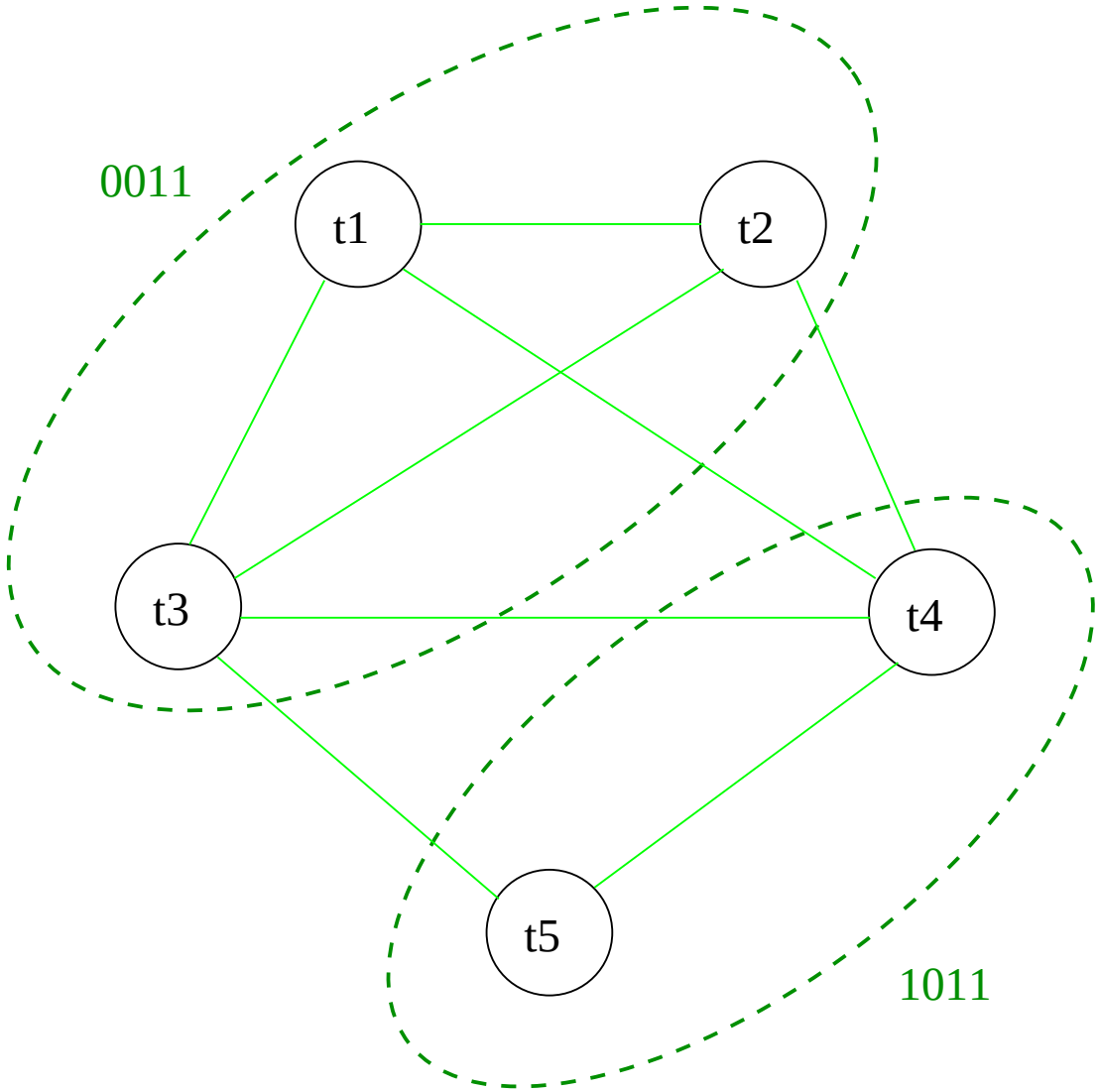
Kompatibilitätsgraph

$$
\begin{aligned}
t_1 &= 00xx \\
t_2 &= 0x1x \\
t_3 &= x0x1 \\
t_4 &= x011 \\
t_5 &= 1x11
\end{aligned}
$$



Kompatibilitätsgraph

$$
\begin{aligned}
t_1 &= 00xx \\
t_2 &= 0x1x \\
t_3 &= x0x1 \\
t_4 &= x011 \\
t_5 &= 1x11
\end{aligned}
$$



Lösungsansatz

Satz: Testmuster, dessen zugehörigen Knoten eine Clique bilden, können (optimal) zu einem einzigen Testmuster kompaktiert werden.

Lösung des Kompaktierungsproblems: Man braucht bloß den Kompatibilitätsgraphen mit einer minimalen Anzahl von Cliquen zu überdecken.

Der Haken an der Sache: Der Kompatibilitätsgraph enthält in der Praxis Zehntausende von Knoten und die Cliquenberechnung ist NP-vollständig.

Lösung mit einem Greedy-Verfahren

Input: Testmuster $\mathfrak{t}_1, \ldots, \mathfrak{t}_n$ (t[1],...,t[n])

Output: Muster $\mathfrak{s}_1, \ldots, \mathfrak{s}_m$; $m < n$ mit gleicher FC wie $\mathfrak{t}_1, \ldots, \mathfrak{t}_n$

Procedure:

```
for(i = 1 to n){
    for(j = (i + 1) to n){
        if(t[i] und t[j] nicht markiert und kompatibel){
            ersetze t[i] durch t[i]⋂t[j] und markiere t[j];
        }
    }
}
entferne die markierten t[i];
```

Laufzeit: $\mathcal{O}(n^2)$

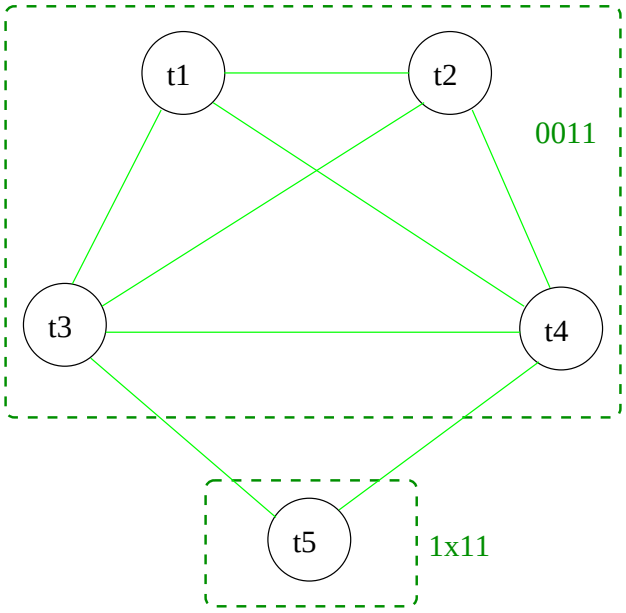
Lösung mit einem Greedy-Verfahren *ff*

Beispiel:

$$
\begin{aligned}
t_1 &= 00xx \\
t_2 &= 0x1x \\
t_3 &= x0x1 \\
t_4 &= x011 \\
t_5 &= 1x11
\end{aligned}
$$

- ◇ Kombiniere t_1 und t_2 zu $001x$.
- ◇ Kombiniere $001x$ und t_3 zu 0011.
- ◇ Kombiniere 0011 und t_4 zu 0011.

$\Rightarrow$ 0011, $1x11$



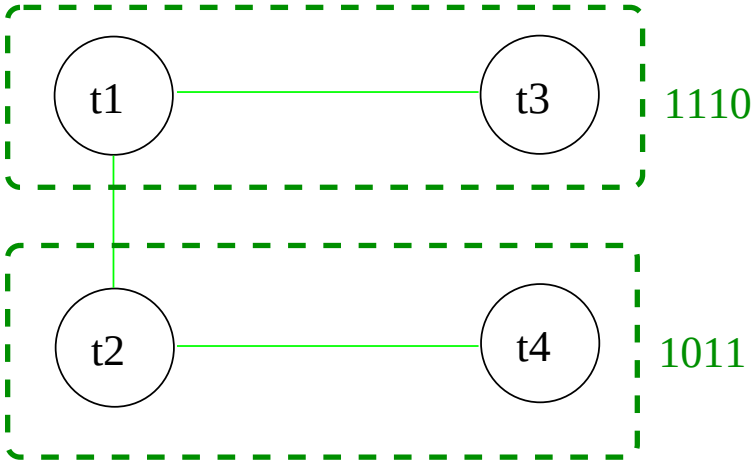
Lösung mit einem Greedy-Verfahren *ff*

Beispiel 2:

$$\begin{aligned} t_1 &= 1x10 \\ t_2 &= x01x \\ t_3 &= 1110 \\ t_4 &= 10x1 \end{aligned}$$

- ◇ Kombiniere t_1 und t_2 zu 1010.
- ◇ Keine weitere Kombinationen möglich.

$\Rightarrow$ 1010, 1110, $10x1$



Frage: Warum begnügt man sich mit einem Verfahren, das suboptimale Lösungen liefert?

Ereignisgesteuerte Fehlersimulation

Vorrangwarteschlangen

Definition: Abstrakter Datentyp zur Verwaltung von Objekten des gleichen Typs, auf die eine totale Ordnung (`Priorität`) definiert ist.

Operationen: Q, Priority Queue

`Q.initialize()`, `Q.isEmpty()`, `Q.insert(e)`, `Q.deletemin()`, `Q.min()`, `Q.decreasekey(v,k)`

Implementationen: Liste, Heap, Treap, binary Queue, Fibonacci-Heaps

Problemstellung

Gegeben:

- ◇ Schaltkreis mit n Inputs und einem Output
- ◇ $m > n$ Signalknoten, die topologisch sortiert sind
- ◇ Array T (Testmuster) und Array S (Werte der Signalknoten)
- ◇ Fehler $^{\text{line}}/_{\text{value}}$

Gesucht: einfacher Algorithmus, der den Fehler durch ereignisgesteuerte Simulation entdeckt.

Algorithmus

Input: Testmuster T, Signalknotenarray S

Procedure:

```
int i; gate G; priority_queue<int> fc.initialize();
for(i = 1 to n) S[i] = T[i];                    /* Belegung der primären Inputs */
for(i = (n + 1) to m){                          /* SK in top. Reihenfolge durchlaufen */
    G = pred(i);                                /* Vorgängergatter des Signalknotens i */
    if(type(G) == INV) S[i] = S[in1(G)]';
    if(type(G) == AND) S[i] = S[in1(G)] ∧ S[in2(G)];
}                                               /* fehlerfreie Simulation fertig */

fc.insert(line);
while(!fc.isEmpty()){
    int f = fc.deletemin();
    if(f == m) return DETECTED;
    G = succ(f);                                /* Nachgängergatter des Signalknotens f */
    if(type(G) == INV){
        fc.insert(out(G)); S[out(G)] = S[out(G)]';
    }
    else{if((f == in1(G) && S[in2] == 1) || (f == in2(G) && S[in1] == 1)){
        fc.insert(out(G)); S[out(G)] = S[out(G)]';
    }}
}

return UNDETECTED;
```

Schlußwort

Schlußwort

Wir haben ein paar Probleme aus dem Bereich der Rechnerarchitektur vorgestellt, die zwar ziemlich fachspezifisch sind, die aber trotzdem an entscheidenden Stellen auf elementare Konzepte zugreifen, die in der Algorithmentheorie vermittelt wurden.

Dies läßt die Hoffnung zu, daß die Plackerei für die Algorithmentheorie nicht umsonst war ;-)

Minipräsentation
Algorithmentheorie
WS 2003/2004

Alejandro Czutro
Matrikelnummer 1133719

Februar 2004

Typesetting with TEX