

Algorithmen im Chip-Entwurf 9

Reale FPGA-Router: PathFinder/VPR

Andreas Koch
FG Eingebettete Systeme
und ihre Anwendungen
TU Darmstadt

Übersicht

- **Problem**
- **Ideen**
- **Modellierung**
- **Algorithmus**
- **Details**
- **Verbesserungsmöglichkeiten**

Problem

- **Verdrahtung auf FPGAs**
- **Begrenzte Anzahl von Ressourcen**
 - Verbindungssegmente
- **Feste Kanalbreite**
 - Unterschied zu vielen ASICs
- **Verdrahtbarkeit ausschlaggebend**
 - Geschwindigkeit zweitrangig

Idee

- **Berücksichtige Verdrahtbarkeit**
 - Bei Lösung des gesamten Verdrahtungsproblems
- **Bestimme**
 - Nachfrage nach Ressourcen
 - ◆ Metallsegmente, Pins, etc.
- **Nachfrage bestimmt Preise**
 - Verschiedene „Verbraucher“ akzeptieren unterschiedliche Preise
 - ◆ „Verbraucher“ = Netze
 - „Billige“ Lösungen haben Nachteile
 - ◆ Sind z.B. langsamer
- **Versuche Gesamtbedarf zu decken**

Vorgehen

- **Verdrahte jedes Netz für sich alleine**
 - Mit den aktuellen Ressourcenkosten
 - Optimal
 - ◆ ... für gegebenen Algorithmus
 - Ignoriere Ressourcenbegrenzungen
- **Zähle Mehrfachbelegungen**
- **Grundlage für Nachfrageberechnung**
- **Solange Mehrfachbelegungen ...**
 - Erhöhe Kosten für stark nachgefragte Ressourcen
 - Verwerfe gesamte Verdrahtung
 - Verdrahte nochmal mit den neuen Kosten
- **Sollte nach 30-45 Iterationen konvergieren**

Algorithmus

- **PathFinder**
 - „A Negotiation-Based Performance-Driven Router for FPGAs“
 - Larry McMurchie, Carl Ebeling
 - Paper auf FPGA 1995
- **Später verbessert von Swartz, Betz und Rose**
 - „A Fast Routability-Driven Router for FPGAs“
 - Paper auf FPGA 1998
- **Details in Buch 1999**
 - „Architecture and CAD for Deep-Submicron FPGAs“

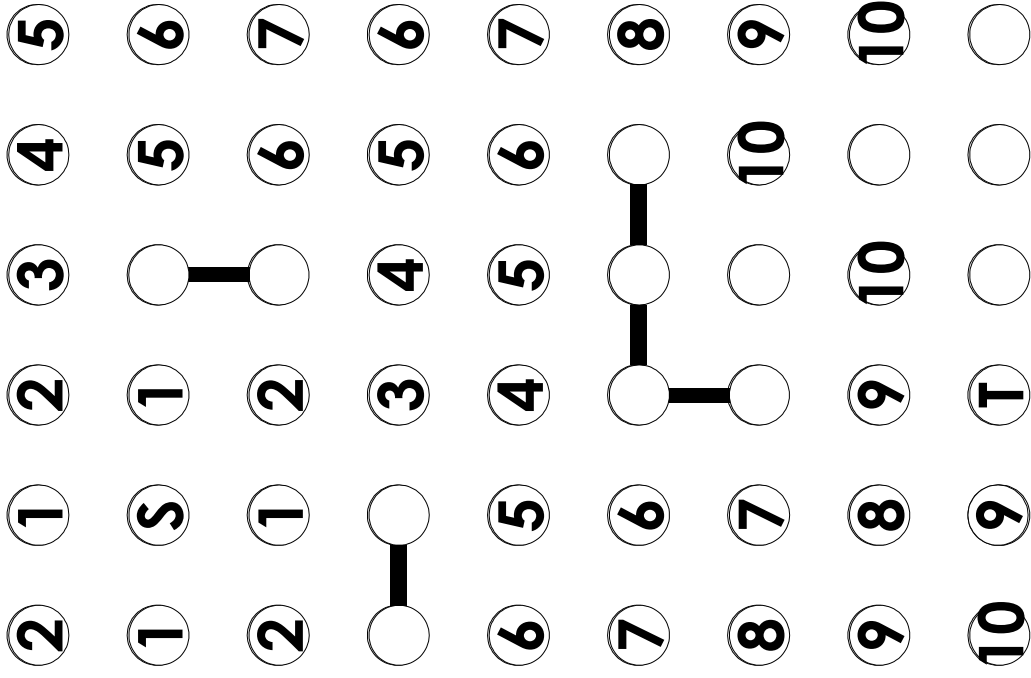
Zwei Stufen

- **Signal Router**
 - Verdrahtet einzelne Netze
 - Maze Router (Lee)
 - ◆ Aber Verbesserungen möglich
- **Global Router**
 - Verdrahtet gesamte Schaltung

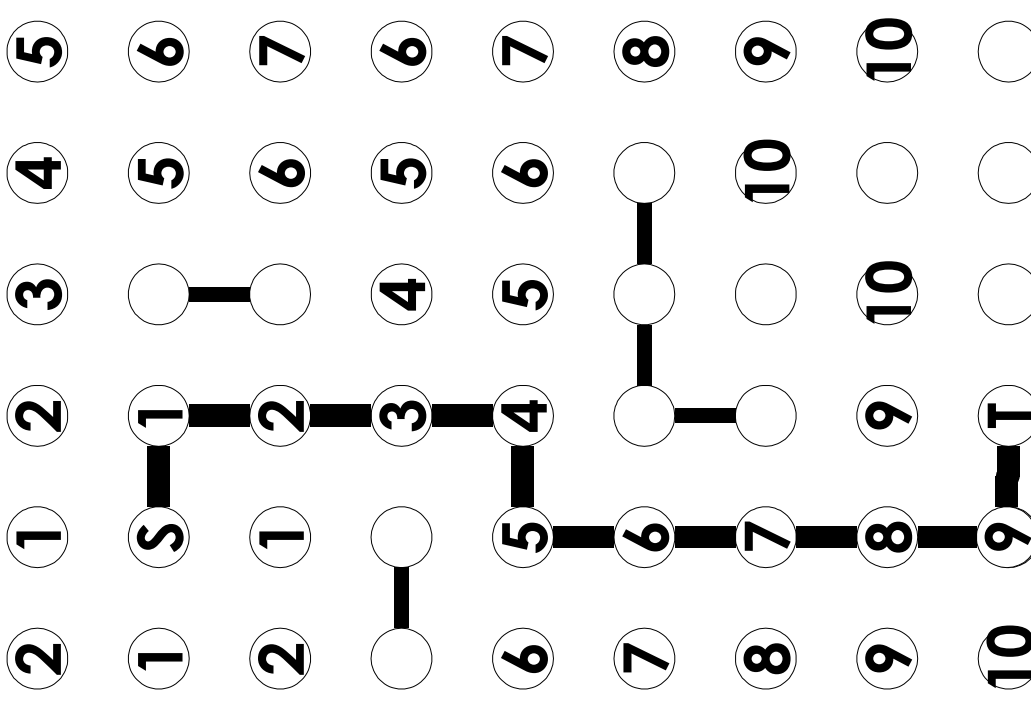
```
globalrouter() {  
    count = 0;  
    while (sharedresources() && count < limit) {  
        foreach (n in Nets)  
            signalrouter(n);  
        count++;  
    }  
    if (count == limit)  
        return „unroutable“  
}
```

Maze Router

Wellenausbreitung



Pfadrückverfolgung



Vorgehen und Kosten

- **Beim Maze-Router**
 - **Breitensuche**
 - ◆ Wellenfront
 - **Kosten: Manhattan-Distanz**
 - ◆ $D = |x1-x2| + |y1-y2|$
 - **Kosten nur bei Rückverfolgung berücksichtigt**
 - ◆ Nicht bei Wellenausbreitung
 - ❖ In alle Richtungen

- **Variation für Signal Router**
 - **Hohe Nachfrage verursacht hohe Kosten**
 - **Bevorzugt in billige Richtungen ausbreiten**
 - ◆ Später Verfeinerung
 - ❖ Zeitkritische Netze dürfen höhere Kosten verursachen

Datenstrukturen

- **Ausbreitung nicht geometrisch auf Fläche**
- **Sondern auf Graph von Routing-Ressourcen**
 - **Routing Resource Graph (RRG)**
- **RtgRsrc**
 - **Einzelne Routing-Ressource**
 - **Z.B. Segment und CLB-Pin**
 - **Wichtig**
 - ◆ **Konkreter Pin, nicht nur logisches Terminal**
- **RtgRsrc sind Knoten im RRG**
- **Kanten: Mögliche Verbindungen dazwischen**
 - **Später genauer ...**

Signal Router 1

```
Tree<RtgRsrc>
signalrouter(Net n) {
    Tree<RtgRsrc> RT; // Gerade konstruiertes Routing für Netz n
    RtgRsrc i, j, v = nil, w;
    PriorityQueue<int,RtgRsrc> PQ;
    HashMap<RtgRsrc,int> PathCost; // Wellenausbreitung

    i = n.source();
    RT.add(i, ()); // Quelle ist Bestandteil der Verdrahtung
    PathCost[*] = +Inf; // Zunächst alles unerreichbar
    PathCost[i] = 0; // Kosten von Quelle zu Quelle sind 0

    foreach (SinkTerminal j in n.sinks()) {
        /* route Verbindung zur Senke j*/
    };

    return (RT);
}
```

Signal Router 2

```
foreach (SinkTerminal j in n.sinks) {  
    PQ.clear();  
    foreach (v in RT.nodes())  
        PQ.add(0, v)  
    do {  
        v = PQ.removeLowestCostNode();  
        if (v != j)  
            foreach (w in v.neighbors()) { /* Kosten ≠ Distanz ! */  
                if (PathCost[w] > PathCost[v] + w.cost()) {  
                    PathCost[w] = PathCost[v] + w.cost();  
                    PQ.add(PathCost[w], w);  
                }  
            }  
        } while (v != j)  
  
        while (!(v in RT.nodes())) {  
            w = v.findCheapestNeighbor(PathCost);  
            RT.add(v, (w, v));  
            v.updateCost(); /* Rsrc. jetzt benutzt, für Nachfolger teurer */  
            v = w;  
        }  
    }  
}
```

Ganze bisherige Route ist Ausgangs,,punkt“

Kostenbasierte Wellenausbreitung

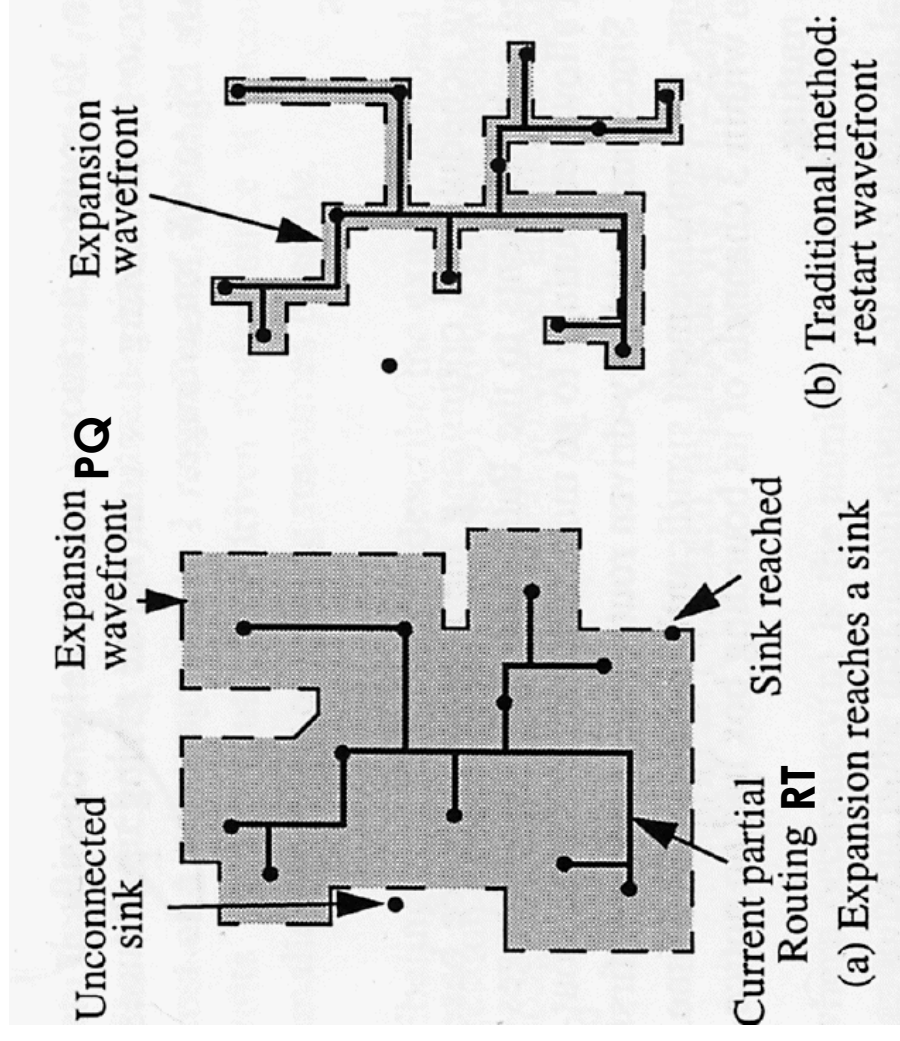
Pfadrückverfolgung

Signal Router Details 1

- **Verdrahtungsressourcen sind persistent**
 - Z.B. Globale Variablen
- **v.cost() über alle Netze berechnet**
 - Mehrere Aufrufe von Signal Router
 - Auch mehrere Iteration vom Global Router (später!)
- **v.updateCost() aktualisiert die Daten**
- **v.neighbors() definiert Verdrahtungsarch.**
 - Routing Resource Graph
 - Sinnvolle Begrenzung:
 - ◆ Nicht mehr als 3 Kanäle ausserhalb des umschliessenden Rechtecks suchen
 - ◆ Verkleinert Suchraum ein
 - ❖ Minimale Qualitätsminderung

Signal Router Details 2

■ Pfadrückverfolgung und Anschluss



Verdrahtbarkeit

■ Fließt via Kostenfunktion $v.cost()$ ein

$$c_v = b_v \cdot p_v$$

■ Idee

- Basiskosten b_v eines Knotens v
 - ◆ Zunächst annehmen $b_v = 1$ (wird später verfeinert)

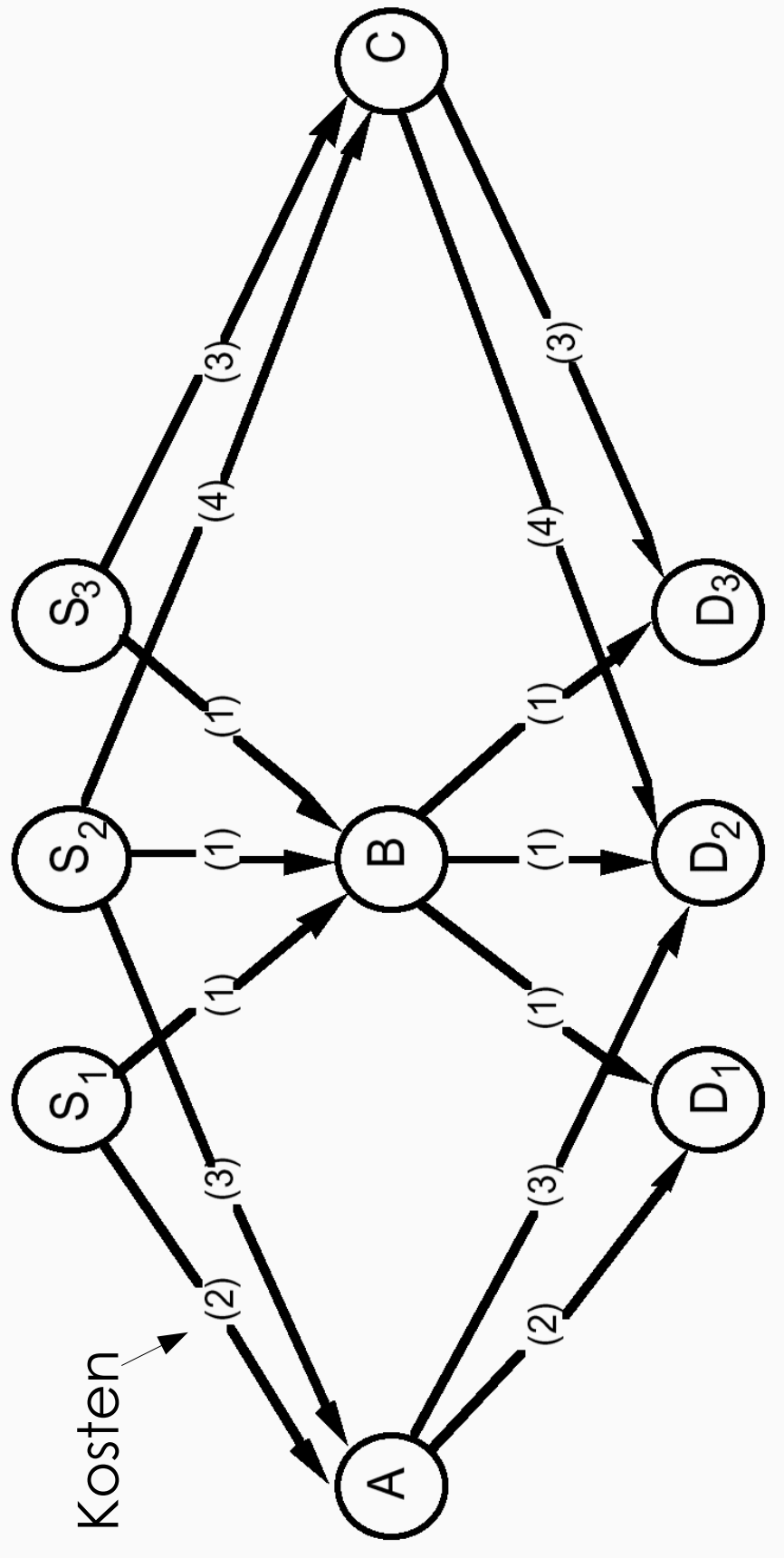
■ Verteuerungsfaktor p_v

- ◆ Erfasst hohe Nachfrage
- ◆ Beginnt klein, wächst im Laufe der Zeit an

$$p(v) = 1 + \max(0, [\text{occupancy}(v) + 1 - \text{capacity}(v)] \cdot p_{fac})$$

- ◆ $\text{Occupancy}(v)$: Belegungsanzahl der Ressource v
- ◆ $\text{Capacity}(v)$: Belegungskapazität der -"-
- ◆ $p_{fac0} = 0.5$, $p'_{fac} = 2 p_{fac}$ nach Iteration vom Global Router
- ◆ Bei jeder Netzänderung $\text{occupancy}(v)$ aktualisieren
 - ❖ Passiert in $v.updateCost$

Beispiel p_v



Maze Routing Reihenfolge: 1,2,3 = 14 2,1,3=12 3,2,1=Fehlschlag

Alternative: Pathfinder-Algorithmus

1. Iteration Global Router: B dreifach belegt, Kosten $v.cost()$ merken

M. Iteration Global Router: 1 nun über A billiger, Kosten merken

N. Iteration Global Router: 3 nun über C billiger, keine Überbelegung mehr

Algorithmus: 1. Versuch

```
globalrouter(Set<Nets> N) {
    HashMap<Net,Tree<RtgRsrc>> NRT;
    count = 0;
    pfac = 0.5;
    while (shareresources() && count < limit) {
        foreach (n in N) {
            // dieses Netz wegnehmen
            NRT[n].unroute();           // muss pv aktualisieren!

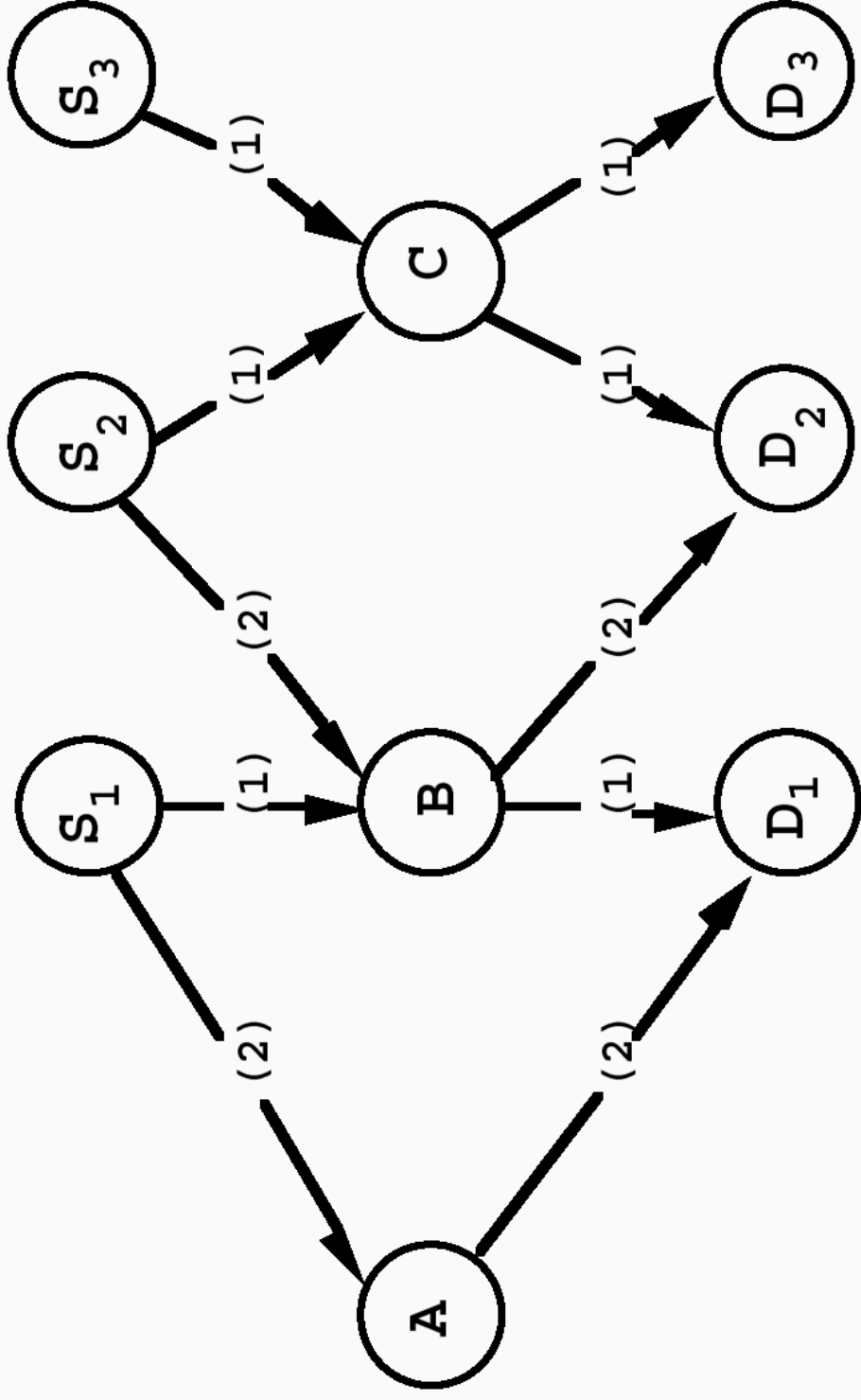
            // und neu verdrahten
            NRT[n] = signalrouter(n);

        }
        // Überbelegungen in nächstem Durchlauf verteuern
        pfac = 2 * pfac;
        updateAllRtgRsrcCosts(pfac);    // wird noch verfeinert!
        count++;
    }
    if (count == limit)
        return „unroutable“
}
}
```

Beispiel

- **An Tafel**
 - **Folien im Web**

Weitergehendes Beispiel



Mit einfachem Maze Router in Reihenfolge 1,2,3: C doppelt belegt

Lösung: 1 aus dem Weg schaffen, 2 neu verdrahten
Aber: 1 ist gar nicht behindert, geht also nicht freiwillig

Lösung

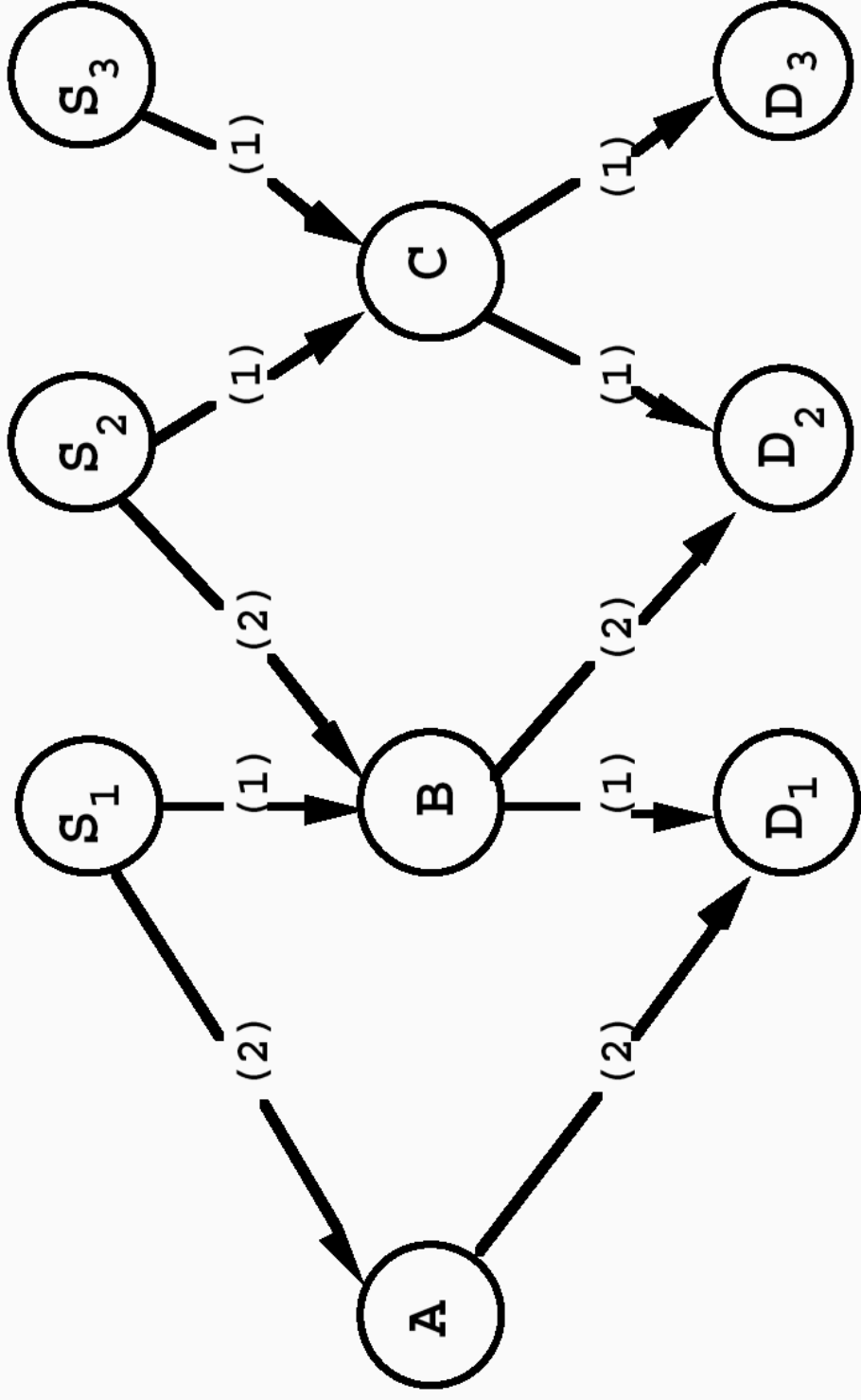
- p_v reicht alleine nicht aus
- Besseres „Gedächtnis“ einführen
 - Historische Überbelegungen erhöhen den akt. Preis
 - h_v akkumuliert alle Mehrfachbelegungen
 - ◆ p_v sieht nur aktuelle Belegung
 - Kostenfunktion erweitern $c_v = b_v \cdot p_v \cdot h_v$
- Aktualisiere einmal pro Global Router Iteration i

$$h(v)^i = \begin{cases} 1, & i=1 \\ h(v)^{i-1} + \max(0, \text{occupancy}(v) - \text{capacity}(v)), & i > 1 \end{cases}$$

Beispiel

- **An Tafel**
 - **Folien auch auf Web-Seite**

Wirkung von h_v



1,2,3: C doppelt belegt

Weitere Iterationen: C wird immer teurer durch Akkumulieren der h_c

2 weicht dann auf B aus, Doppelbelegung via p_B, h_B , 1 weicht auf A aus

Basiskosten b_v

- **Idee: RtgRsrc-Verzögerung einfließen lassen**
 - Bei uns nur via T_{switch}
 - Führt aber zu Verschlechterung!
- **Besser: Feste Kosten (U Toronto)**
 - Benötigt 10% weniger Tracks als mit variablen b_v
- **Idee zur Beschleunigung:**
 - Bevorzuge Input Pins
 - ◆ Niedrigere Kosten
 - ◆ „Lockt“ Maze Router via PriorityQueue PQ schneller zu Sinks
 - ❖ Werden eher abgearbeitet
- **Vorschlag**
 - Input Pins $b_v = 0.95$
 - Alle anderen Elemente $b_v = 1$

Vervollständige globalrouter()

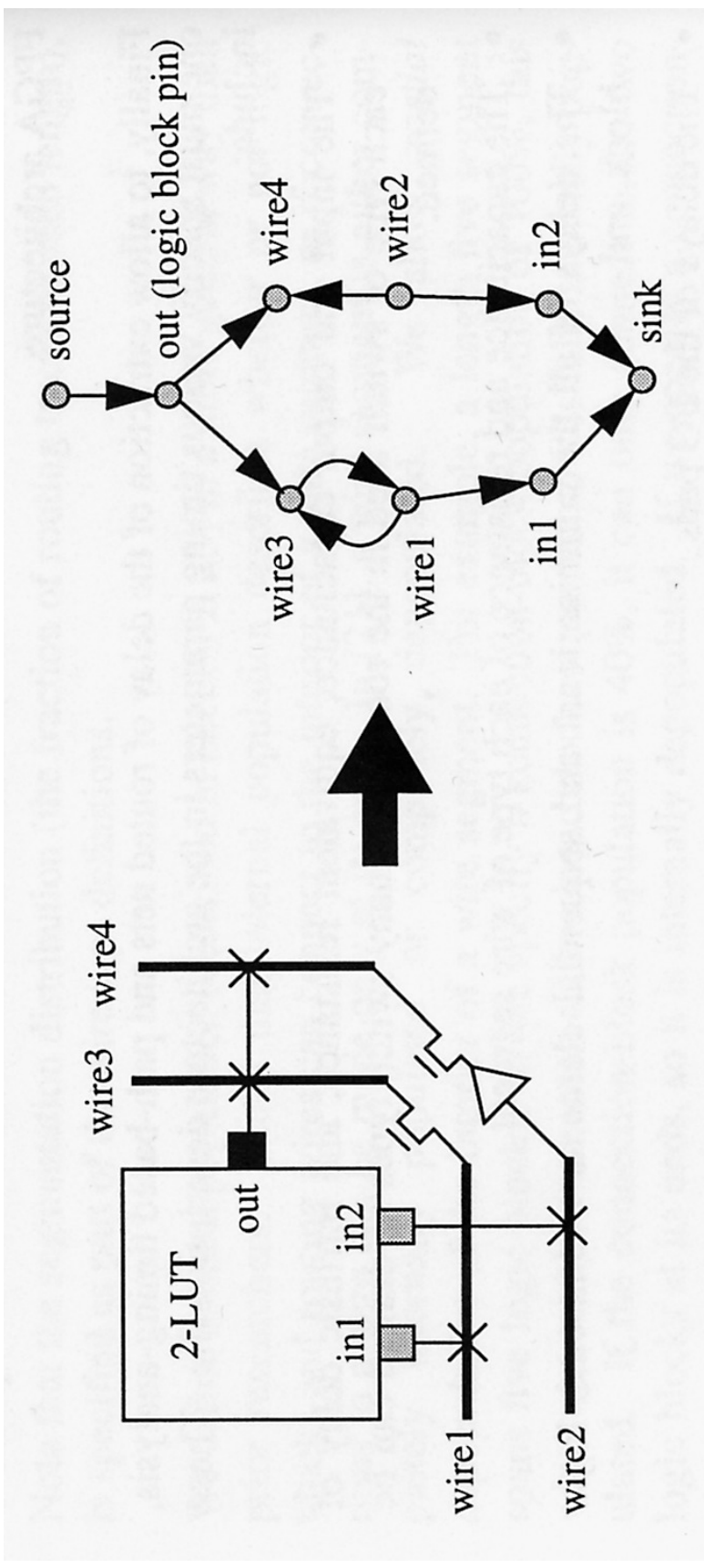
```
Graph<RtgRsrc> Interconnect; // Kanten (RtgRsrc,RtgRsrc)

globalrouter(Set<Nets> N) {
    HashMap<Net,Tree<RtgRsrc>> NRT;
    count = 0;
    pfac = 0.5
    while (sharedresources() && count < limit) {
        foreach (n in N) {
            NRT[n].unroute(); // muss pv aktualisieren!
            NRT[n] = signalrouter(n);
        }
        pfac = 2 * pfac;
        count++;
        foreach (r in Interconnect.nodes()) {
            r.updateHistory(); // hv aktualisieren
            r.updateWithNewPfac(); // Gesamtkosten aktualisieren
        }
    }
    if (count == limit)
        return „unroutable“
}
}
```

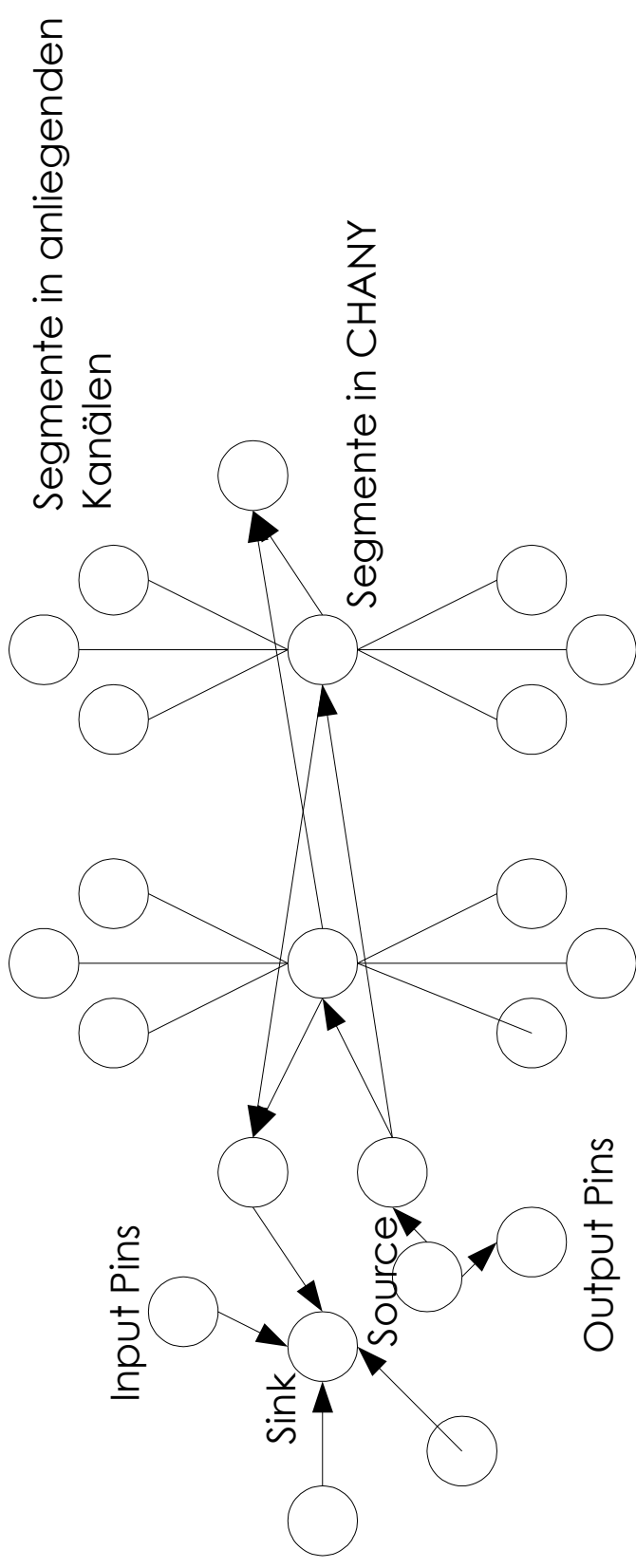

Routing Resource Graph RRG

- **Fundamentale Datenstruktur**
- **Modelliert Verbindungsnetzwerk**
- **Knoten**
 - Leitungen (Verdrahtungssegmente)
 - Pins
- **Kanten**
 - Schalter (Pass-Transistoren, bidirektional)
 - Buffer (unidirektional)
- **Äquivalente Pins**
 - Outputs: Source-Knoten
 - Inputs: Sink-Knoten
- **Fassungsvermögen (capacity)**
 - Bei Source/Sink-Knoten: Anzahl der Out/In-Pins

RRG Beispiel 1



RRG Beispiel 2



- Verzögerung $d_{u,v}$
- T_{switch} zwischen Metallsegment-Knoten u, v

Ausbau auf Verzögerung

- **Optimiere auch noch Verzögerung**
 - Zwischen Terminals i und j eines Netzes
- **Erweiterung der Kostenfunktion $v.cost(u)$**

$$C_{u,v} = Crit(i, j) \cdot d_{u,v} + [1 - Crit(i, j)] \cdot b_v \cdot h_v \cdot p_v$$

- $d_{u,v}$: Verzögerung von u nach v
- **Crit(i,j): Abart der Criticality(i,j)**

$$Crit(i, j) = \max(0.99 - \frac{slack(i, j)}{D_{max}}, 0)$$

- **Idee: Auch kritische Netze achten etwas auf Verdrahtbarkeit**

Änderung signalrouter()

```
foreach (SinkTerminal j in n.sinks ordered decreasing Crit(i,j)) {
    PQ.clear();
    foreach (v in RT.nodes())
        PQ.add(0, v)
    do {
        v = PQ.removeLowestCostNode();
        if (v != j)
            foreach (w in v.neighbors()) {
                if (PathCost[w] > PathCost[v] + w.cost(v)) {
                    PathCost[w] = PathCost[v] + w.cost(v);
                    PQ.add(PathCost[w], w);
                }
            }
        } while (v != j)

        while (!(v in RT.nodes())) {
            w = v.findCheapestNeighbor(PathCost);
            RT.add(v, (w, v));
            v.updateCost();
            v = w;
        }
    }
}
```

Änderung globalrouter()

```
Graph<RtgRsrc> Interconnect;

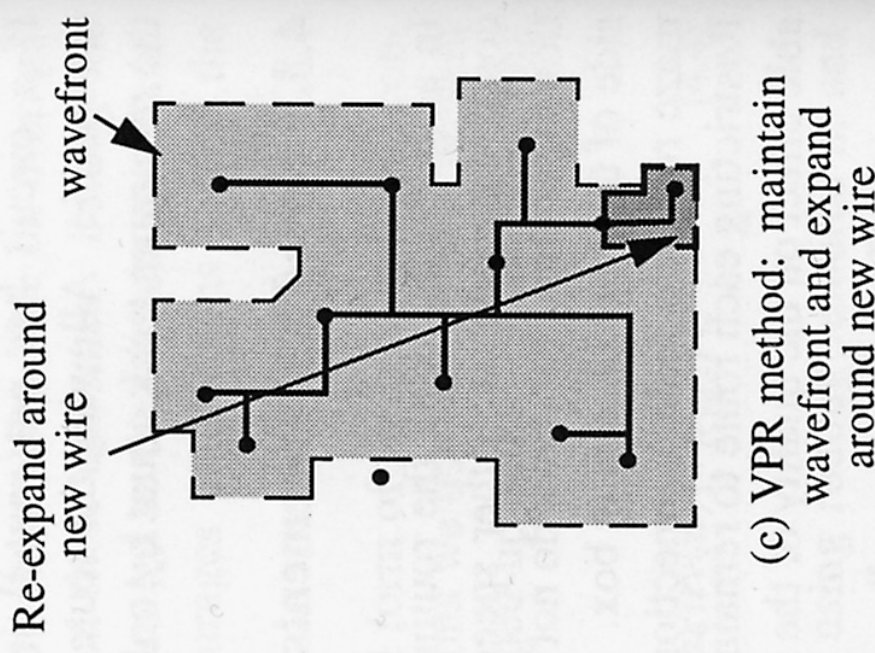
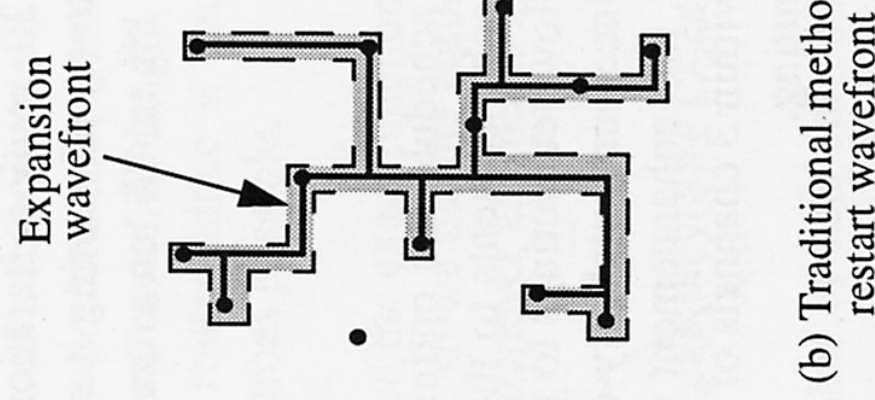
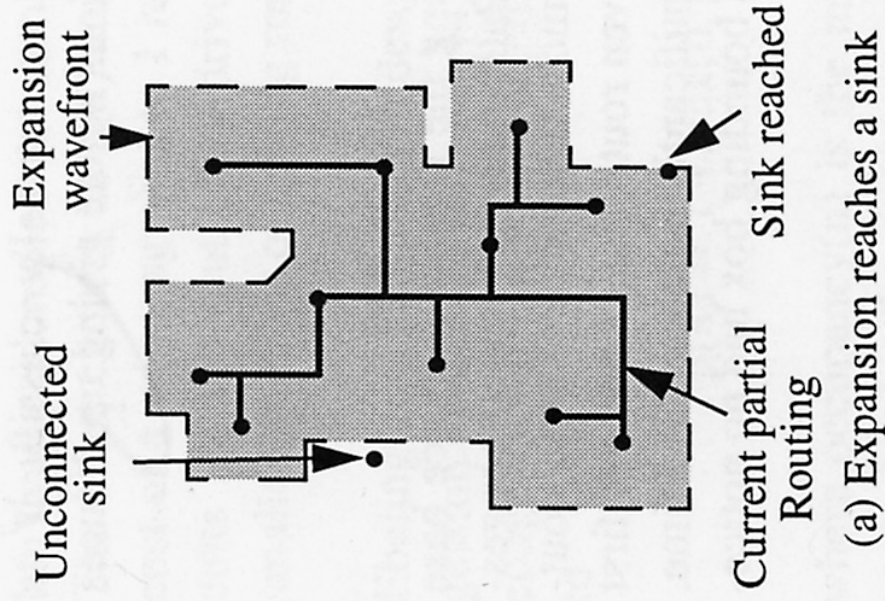
globalrouter(Set<Nets> N) {
    HashMap<Net,Tree<RtgRsrc>> NRT;
    count = 0;
    foreach (n in N)
        foreach (j in n.sinks())
            Crit[n.source(), j] = 1
    while (sharedresources() && count < limit) {
        foreach (n in N) {
            NRT[n].unroute(); // muss pv aktualisieren!
            NRT[n] = signalrouter(n);
        }
        count++;
        foreach (r in Interconnect)
            r.updateHistoryAndPfac();// hv und pfac aktualisieren
        N.timingAnalysis(); // Crit[i,j] aller Netze aktualisieren
    }
    if (count == limit)
        return „unroutable“
}
```

**Im ersten Durchgang
Nachfrage bei minimaler
Verzögerung bestimmen**

Vergleich

- **PathFinder [McMurchie&Ebeling 1995]**
 - Zunächst nur verdrahtungsorientiert
 - Keine vorgegebene Sink-Reihenfolge
 - Wellenausbreitung
 - ◆ Bis alle Sinks erreicht
- **Verbesserbar**
 - Alte Wellenfront in PQ nicht verwerfen
 - ◆ Einfach neue Sink an RT anschliessen
 - ◆ Neue Segmente in PQ übernehmen (VPR 1997)
 - ◆ Nur bei reiner Verdrahtungsorientierung
 - ◆ Klappt *nicht* bei Optimierung auf Verzögerung!
 - ❖ Unterschiedliche Kosten c_v bei anderen Terminals i, j
 - Bei Verzögerungsorientierung
 - ◆ Jetzt steht Sink-Reihenfolge fest
 - ❖ Im Paper: Absteigende A_{ij} (vergleichbar Criticality)

Schnellere Wellenausbreitung



- Nur bei reiner Verdrahtungsorientierung
- Nicht bei Einbeziehungen von Timing!

Verbesserungen

- **Swartz, Betz, Rose 1998 (U Toronto)**
- **Optimierung auf Geschwindigkeit**
 - **Qualitätsverlust?**
- **Zwei Kernideen**
 - **Gezielte Ausbreitung statt breiter Wellenfront**
 - **Beschränkung auf sinnvolle Startpunkte**
- **Diverse Detailverbesserungen**

Ausbreitung 1

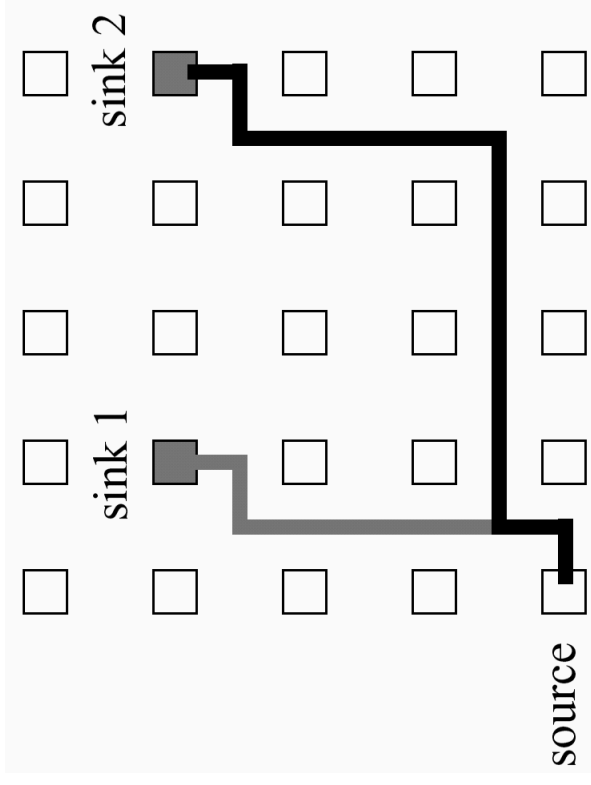
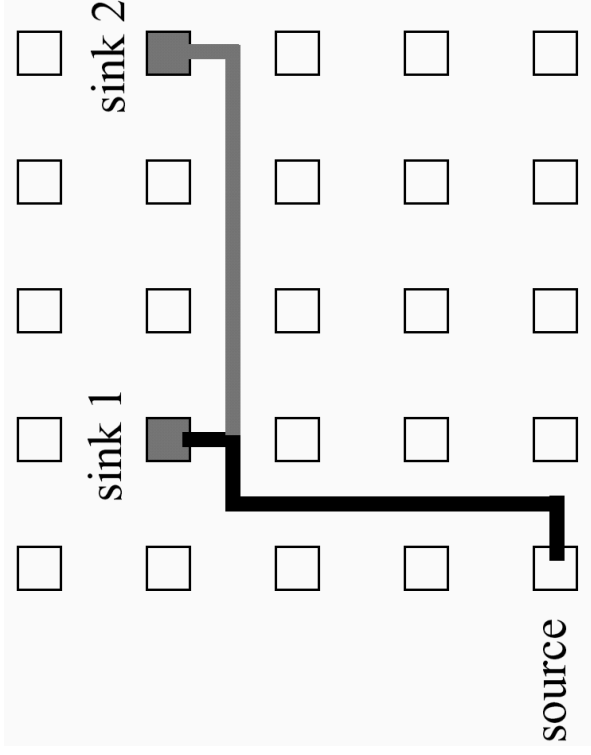
■ Gerichtete Tiefensuche DDFS statt BFS

- Suche bevorzugt in Richtung auf Ziel j zu

$$\text{Cost}(i, v) = \text{PathCost}(i, u) + C_0 + \alpha \cdot \Delta D$$

- ◆ $\text{PathCost}(i, u)$: Kosten bis zum Vorgänger u
- ◆ C_0 : Verdrahtungsabhängige Basiskosten
 - ❖ Vergleichbar c_v , wächst aber viel stärker
 - ❖ Weniger Iterationen
- ◆ ΔD : Manhattan-Distanz von v zum Ziel j
 - ❖ <0 : v liegt näher an j als u (= billiger)
 - ❖ >0 : v liegt weiter von j als u (= teurer)
- ◆ α : Richtungsfaktor
 - ❖ $=0$: BFS, keine richtungsabhängigen Komponenten
 - ❖ >0 : Nicht mehr verdrahtungsorientiert, Greedy
 - ❖ $=1.5$: Empfohlen, hohe Beschleunigung, gute Qualität

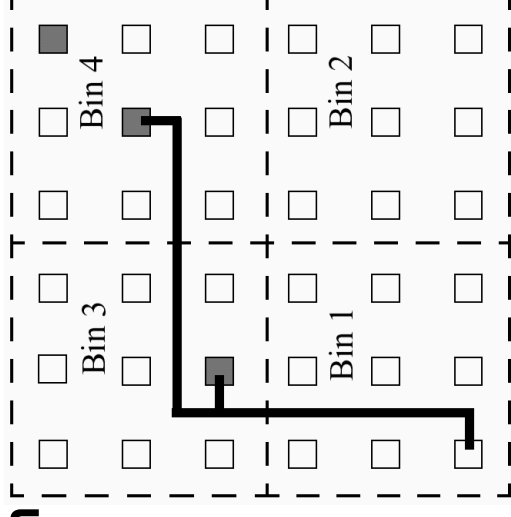
Ausbreitung 2



- Reihenfolge der Sinks
 - Nächstgelegene zuerst
 - ◆ Bessere Anschließbarkeit der folgenden Sinks
- Reihenfolge der Netze
 - Die mit vielen Terminals zuerst
 - ◆ Vermeidung von Blockaden

Sinnvolle Startpunkte

- **PathFinder/VPR**
 - **Ausbreitung von gesamten RT aus**
 - ◆ Übernahme in PQ mit Kosten 0
 - **Ineffizient, gerade bei vielen Terminals**
- **Idee**
 - **Nur Segmente aus RT „nahe“ beim Ziel in PQ**
 - **Aufteilen der gesamten Fläche in Bins**
 - ◆ Hier:
 - ❖ Nur Segmente in Bin 4 expandieren
- **Lohnend bei**
 - **Netzen mit vielen Terminals**



Binning Details

- **Bin-Größe**
 - Sollte passen
 - Berechnung pro Netz n
 - ◆ Durchschnittliche Fläche pro Sink $A_s = \text{bbox}(n) / |\text{sinks}(n)|$
 - ◆ Bewährt: Bin-Größe $4 \times A_s$
 - Expandiere
 - ◆ Nur Segmente im gleichen Bin wie nächstes Ziel
 - ❖ Einfache Entfernungsberechnung, kein Bin-Raster

- **Leere Bins**
 - Bin mit Ziel enthält noch keine RT-Segmente
 - Erweitere Suchradius auf 8 Nachbar-Bins
 - Falls immer noch leer
 - ◆ Suche von ganzem RT aus

Auswirkungen

- **Low-Stress Routing**
 - >10% mehr Tracks als minimal erforderlich
- **15 Beispielschaltungen**
- **Durchschnittliche Rechenzeit**
 - BFS in VPR: 731s
 - DDFS: 14s
 - DDFS+Bins: 7s
- **Durchschnittlicher Qualitätsverlust**
 - BFS in VPR: 15.5 Tracks
 - DDFS: 15.5 Tracks
 - DDFS+Bins: 15.8 Tracks

Programmierprojekt

- **Algorithmus nicht genau nachprogrammieren**
 - Viele Details nicht gezeigt!
- **Konzepte verstehen**
- **Inspiration für eigene Ideen**
- **Sinnvoll**
 - Routing Graph
 - Darin nach Verdrahtungen suchen
- **Papers auf Web-Site**
 - PathFinder, McMurchie & Ebeling 1995
 - Verbesserungen von Swartz et al., 1998
 - Auszüge aus VPR Beschreibung, 1999 [19MB!]

Zusammenfassung

- **Verdrahtungsproblem auf FPGAs**
- **Verdrahtbarkeitsorientierte Verdrahtung**
- **PathFinder-Algorithmus**
 - Gewichteter Maze-Router
 - p_v, h_v
- **Erweiterung auf Verzögerung**
 - Durch Criticality
- **Verbesserungen**
 - Bessere Suchalgorithmen
- **Vorlesung am Dienstag**
 - Verdrahtung 2 (ASIC)