



Vorlesung
WS 2013/2014

Florian Stock, Andreas Koch

Eingebette Systeme und Anwendungen
Technische Universität Darmstadt



- ▶ Bisher: Positionen für Objekte bestimmt?
- ▶ Wie werden diese Verbunden?
 - ▶ Verdrahtung bisher immer nur abgeschätzt
- ▶ Üblicherweise wird Verdrahtung zweigeteilt
 - ▶ Ähnliche dem hierarchischem Ansatz bei Partitionierung und Floorplanning
 - ▶ Globale Verdrahtung
 - ▶ Bestimmt die Lage ganzer *Verdrahtungskanäle*
 - ▶ Auf dem ganzen Chip
 - ▶ Lokale Verdrahtung
 - ▶ Bestimmt den Verlauf einzelner *Leitungen*
 - ▶ Innerhalb eines Verdrahtungskanales



- ▶ 2 Ansätze:
 - ▶ Sequentiell – ein Netz nach dem anderen
 - ▶ Reihenfolge der Netze sehr wichtig
 - ▶ Üblicherweise sortiert nach
 - ▶ Kritikalität (Criticality)
 - ▶ Anzahl der Terminals
 - ▶ Gleichzeitig (Concurrently) – alle Netze Gleichzeitig
 - ▶ Sehr Berechnungsaufwendig
 - ▶ Üblicherweise mittels hierarchischen Ansätzen

Globale Verdrahtung

Problem

- Eingabe:**
 - ▶ Lage der Terminals (aus Platzierung)
 - ▶ Zu Verbindene Terminals (Netzliste)
 - ▶ Kanalkapazitäten
- Ausgabe:**
 - ▶ Verdrahtungstopology
- Optmierung:**
 - ▶ Anzahl der gerouteten Netze
 - ▶ Standardzellen & ASICs können Reihen/Zellen Kanälen Platz machen
 - ▶ Bei FPGAs nicht möglich
 - ▶ Routing Fläche
 - ▶ Verdrahtungslänge



- ▶ Allgemeine Verfahren
 - ▶ Generische Graphen/Weg-Such-Algorithmen
- ▶ Eingeschränkte Verfahren
 - ▶ Kanalverdrahtung
 - ▶ Switchbox-Verdrahtung



- ▶ Verdrahtungslagen (Layer)
 - ▶ Anzahl technologieabhängig (28 nm bis zu 10)
 - ▶ Erlaubte Ausrichtung
Horizontal, Vertikal, beides, 45°
 - ▶ Optional: Über Logik verdrahten (over-cell routing)
 - ▶ Lagenverbindung mittels *Vias*
- ▶ Freie Verdrahtung oder auf Raster (grid-based/gridless oder shaped-based)
 - ▶ Frei viel flexibler, aber entsprechend langsamer
- ▶ Beschränkungen: Design und/oder Performanz

Multilayer Beispiele

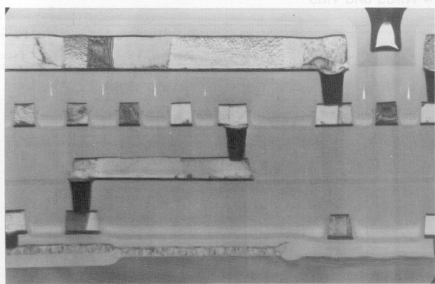
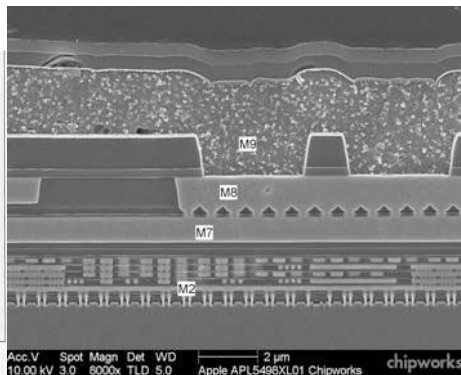


Figure 2-15: Cross-section of 4-level metal/1-level poly interconnect (courtesy UMC).





- ▶ Modellierung der Ressourcen als Graph
⇒ Routing-Ressource-Graph
- ▶ Alle Ressourcen werden als Knoten behandelt
- ▶ Kantengewichte entsprechen der Verzögerung
(Verbreitetes Verzögerungsmodell: Elmore Delay)



- ▶ 3 allgemeine Algorithmen
 - ▶ Maze
 - ▶ Line-Search
 - ▶ A*-Suche
- ▶ 2 globalen Routing-Algorithmen
 - ▶ Pathfinder
 - ▶ ILP-Formulierung
- ▶ Spezielle lokale Routing-Algorithmen
 - ▶ Kanal-Verdrahtung
 - ▶ Left-Edge Algorithmus
 - ▶ Robuster Kanalrouter

Maze Algorithmus

Lees Algorithmus



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ Lee, 1961
- ▶ BFS
 - 2 Phasen: Wellenpropagation und Rückverfolgung
- ▶ Arbeitet auf einfachem Raster
- ▶ Findet garantiert den kürzesten Pfad
- ▶ Sehr langsam und speicherintensiv (beides $\mathcal{O}(n^2)$)

Maze Algorithmus

Wellenausbreitung

MazeSearchPropagation(int xdim, int ydim, Vertex q, Vertex s) **begin**

```
int cells[xdim][ydim] ;  
InitCells(cells) ;  
Set schonBesucht :=  $\emptyset$  ;  
int dist := 0 ;  
repeat  
  Set neighbours :=  
  neighbours(schonBesucht) ;  
  foreach Zelle  $c \in$   
  neighbours \ schonBesucht do  
     $c := dist$  ;  
  dist := dist + 1 ;  
until  $s \in schonBesucht$  ;
```

- ▶ 1. Phase: Wellenausbreitung von Quelle aus
- ▶ In jedem Schritt nächste unbesuchte Nachbarn besuchen
- ▶ Bis die Senke erreicht ist

Maze Algorithmus

Rückverfolgung

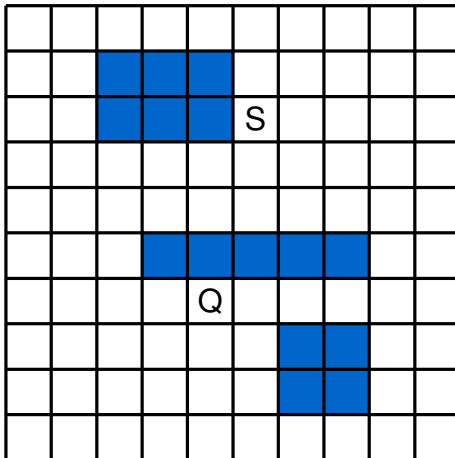
MazeSearchBacktrace(int cells[xdim][ydim],
Vertex q, Vertex s) **begin**

```
List<Zellen> route ;  
Zelle next = s ;  
repeat  
    route.append(next) ;  
    Set<Zellen> kandidaten :=  
    neighbours(next) ;  
    next:=selectRandom(kandidaten) ;  
until next = q;  
route.append(q) ;  
return route;
```

- ▶ 2. Phase: Rückverfolgung von Senke aus
- ▶ In jedem Schritt eine Zelle mit kleinere Nummer wählen
- ▶ Bis die Quelle erreicht ist

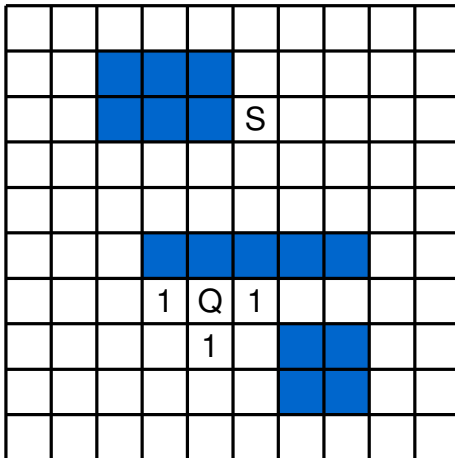
Maze Algorithmus

Beispiel



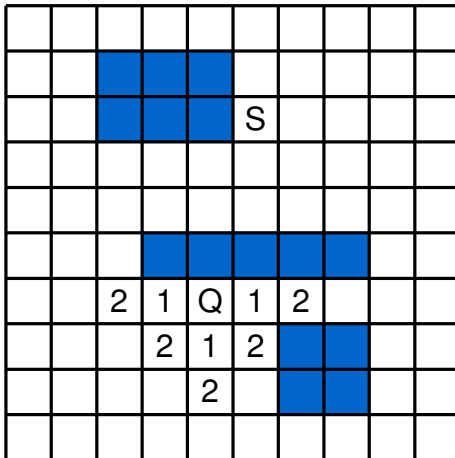
Maze Algorithmus

Beispiel



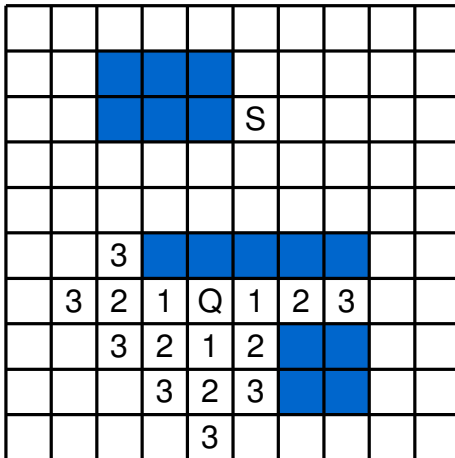
Maze Algorithmus

Beispiel



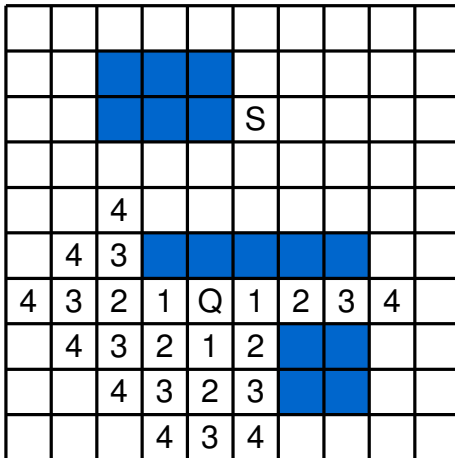
Maze Algorithmus

Beispiel



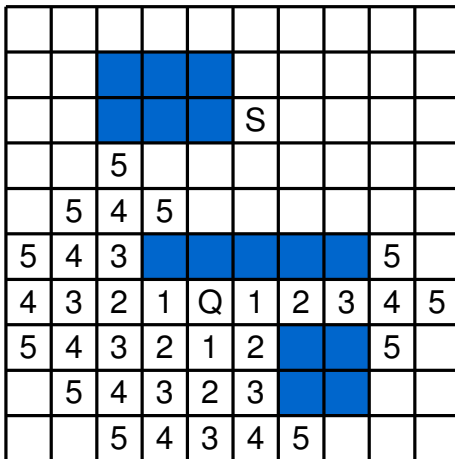
Maze Algorithmus

Beispiel



Maze Algorithmus

Beispiel



Maze Algorithmus

Beispiel

					S				
	6	5	6						
6	5	4	5	6				6	
5	4	3						5	6
4	3	2	1	Q	1	2	3	4	5
5	4	3	2	1	2			5	6
6	5	4	3	2	3			6	
	6	5	4	3	4	5	6		

Maze Algorithmus

Beispiel

	7			S					
7	6	5	6	7				7	
6	5	4	5	6	7		7	6	7
5	4	3						5	6
4	3	2	1	Q	1	2	3	4	5
5	4	3	2	1	2			5	6
6	5	4	3	2	3			6	7
7	6	5	4	3	4	5	6	7	

Maze Algorithmus

Beispiel

	8								
8	7				S			8	
7	6	5	6	7	8		8	7	8
6	5	4	5	6	7	8	7	6	7
5	4	3						5	6
4	3	2	1	Q	1	2	3	4	5
5	4	3	2	1	2			5	6
6	5	4	3	2	3			6	7
7	6	5	4	3	4	5	6	7	8

Maze Algorithmus

Beispiel

	9								
9	8							9	
8	7				S		9	8	9
7	6	5	6	7	8	9	8	7	8
6	5	4	5	6	7	8	7	6	7
5	4	3						5	6
4	3	2	1	Q	1	2	3	4	5
5	4	3	2	1	2			5	6
6	5	4	3	2	3			6	7
7	6	5	4	3	4	5	6	7	8

Maze Algorithmus

Verbesserungen

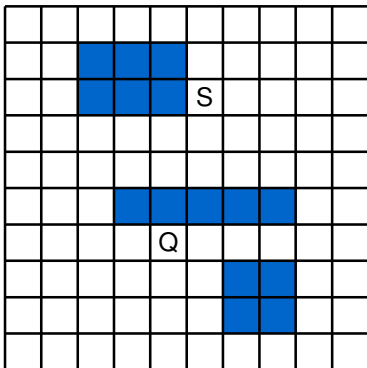
- ▶ Verbesserter Speicherbedarf
(Es reichen 2 Bit pro Zelle)
- ▶ Beschleunigung (bis zu $10-50 \times$ schneller):
 1. In Richtung Senke DFS
 2. Bei Hinderniss zum BFS umschalten
 3. Findet Weg wenn möglich, aber nicht Kürzesten
- ▶ Suchraum einschränken:
 - ▶ Von Quelle und Senke gleichzeitig propagieren
 - ▶ Bounding Box (Quelle und Senke) + 10% Sicherheit als Rahmen
- ▶ Mehrere Ebenen
Höhere Kosten für Vias
- ▶ Multi-Terminal Netze
 - ▶ Verdrahte erst zwei Terminals
 - ▶ Den *gesamten* Pfad als Quelle für Wellenpropagation
 - ▶ Kürzester Pfad nicht mehr garantiert! (MRST)



- ▶ Linien-Suche (Line Search, Line Probing)
- ▶ Mikami, Tabuchi (1968); ähnlich Hightower (1969)
- ▶ Findet Pfad wenn er existiert
aber nicht unbedingt den kürzesten
- ▶ Idee:
 - ▶ Quelle und Senke haben eine Liste von Liniensegmenten
 - ▶ Im Ersten Schritt werden horizontale und vertikale Liniensegmente von Start und Senke ausgewählt
 - ▶ Linien enden an Hindernissen oder Grenzen
 - ▶ Schneiden sich Liniensegmente von Quelle und Senke ist ein Pfad gefunden
 - ▶ Wenn nicht, werden alle (Hightower: einer) Gitterpunkte der bisherigen Segmente Ausgangspunkt für neue (senkrechte) Liniensegmente (Fluchtpunkt/Escape Point)

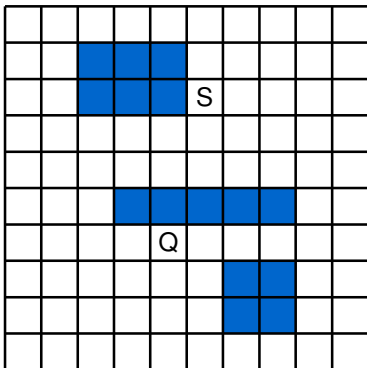
Linien-Suche

Beispiel



Linien-Suche

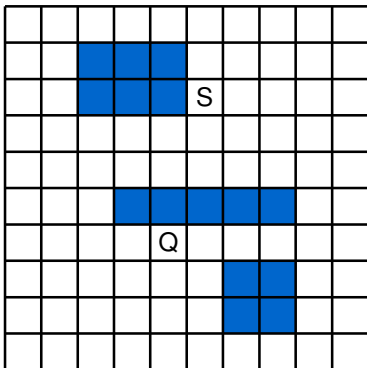
Beispiel



1. Iteration

Linien-Suche

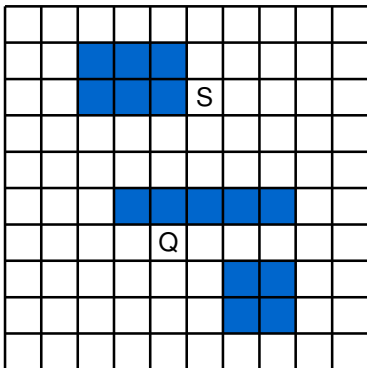
Beispiel



1. Iteration
2. Iteration

Linien-Suche

Beispiel



1. Iteration
2. Iteration

Komplexität: $\mathcal{O}(L)$,
 L = Anzahl Liniensegmente

- ▶ Hart, 1968
- ▶ Verallgemeinerung der BFS
- ▶ Praktisch weit verbreitet
- ▶ Nicht in alle Richtungen *gleichzeitig* suchen
Die *richtige* bevorzugen
- ▶ Kostenfunktion $c(x) = s(x) + g(x)$
 - $s(x)$: Kosten von Quelle bis zu Knoten x
 - $g(x)$: Geschätzte Kosten von x bis zur Senke
 g darf nicht überschätzen (zulässig/admissable) \Rightarrow optimal
- ▶ Bei normalen BFS ist $b(x) = 0$
Einfache Abschätzung wäre z.B. Manhattan-Distanz



- ▶ Übliche Implementierung: Prioritätswarteschlange

```
PriorityQueue q ;
```

```
q.push(quelle, 0) ;
```

```
repeat
```

```
    next := q.pop() ;
```

```
    if next  $\neq$  senke then
```

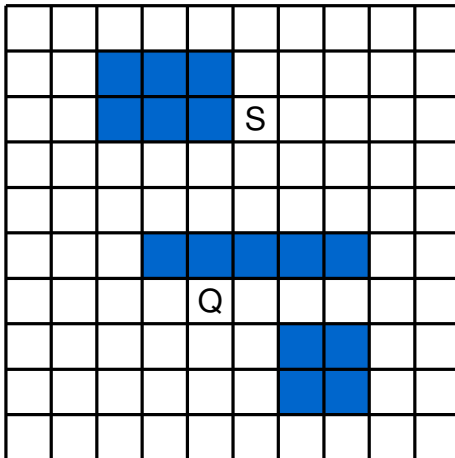
```
        foreach Nachbarzelle z von next do
```

```
            q.push(z, c(z)) ;
```

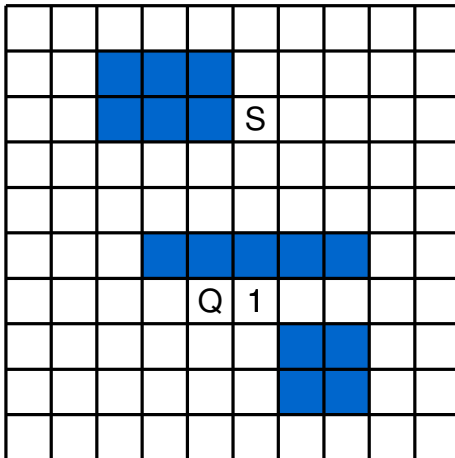
```
until next = senke;
```

- ▶ Komplexität: Worst Case wie Maze-Algorithmus

A*-Suche Beispiel

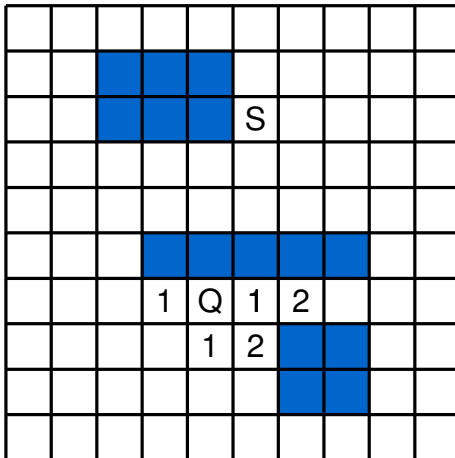


A*-Suche Beispiel



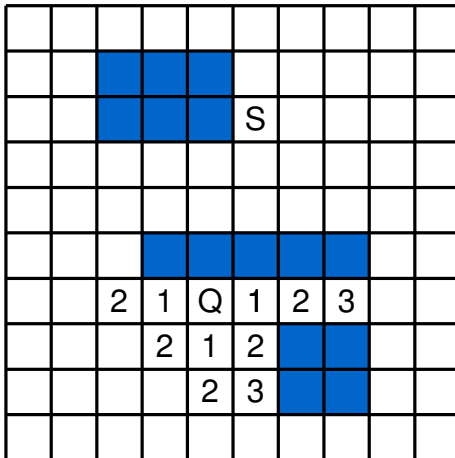
A*-Suche

Beispiel

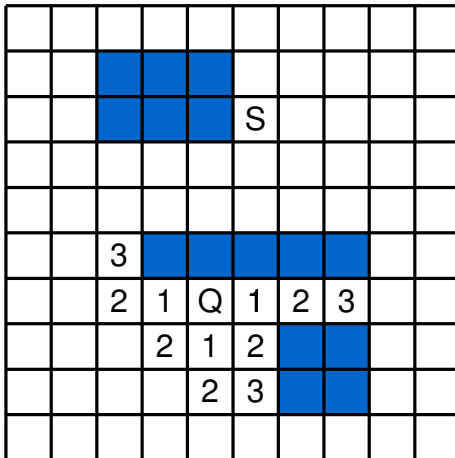


A*-Suche

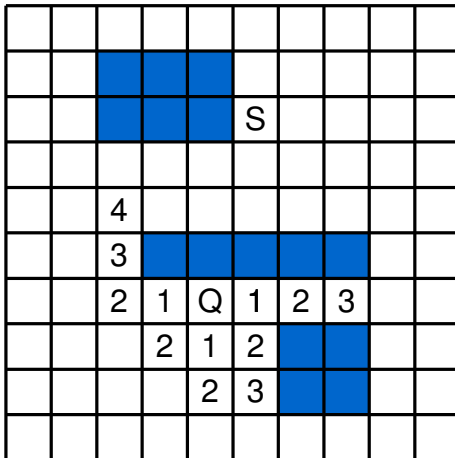
Beispiel



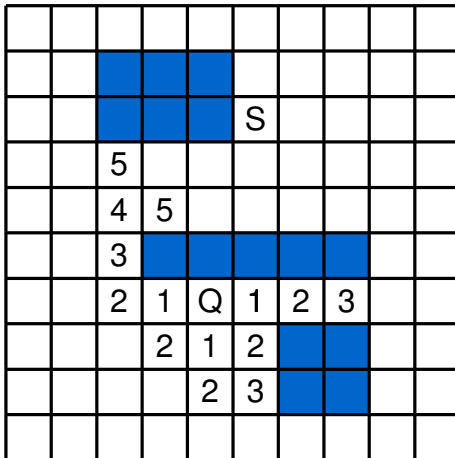
A*-Suche Beispiel



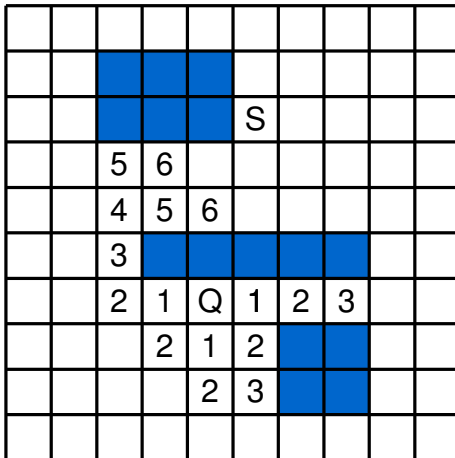
A*-Suche Beispiel



A*-Suche Beispiel

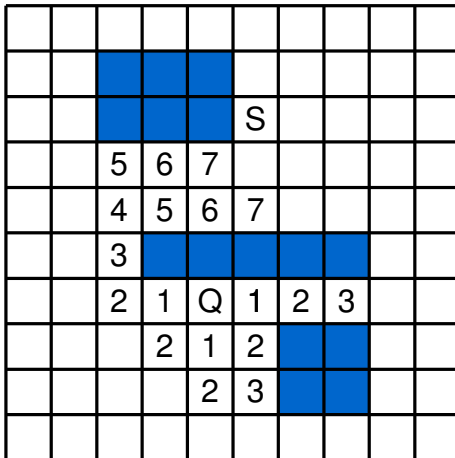


A*-Suche Beispiel



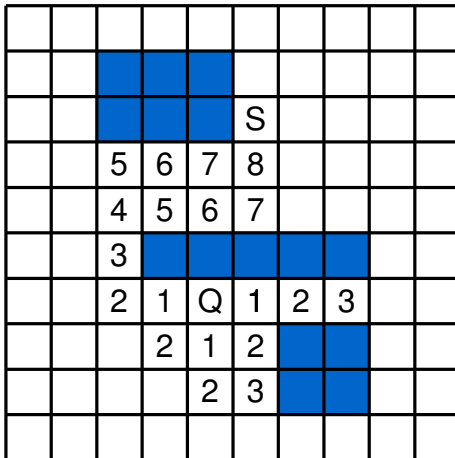
A*-Suche

Beispiel



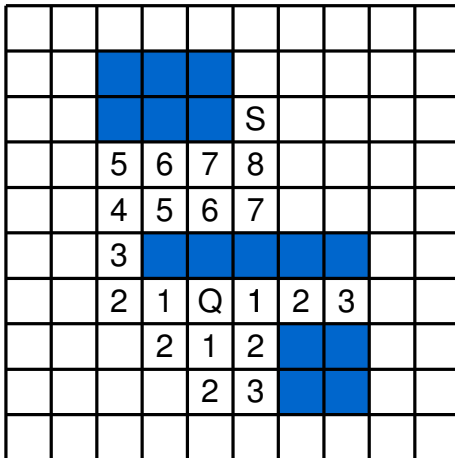
A*-Suche

Beispiel



A*-Suche

Beispiel



- ▶ Verdrahten aller Netze gleichzeitig
- ▶ Umgeht Problem: Welches zuerst verdrahten?
- ▶ Formulierung:

$$\begin{array}{ll} \text{Min} & \sum_{i=1}^n \sum_{j=1}^{k_i} L_{ij} \times x_{ij} \\ \text{s.t.} & \end{array}$$

$$\sum_{j=1}^{k_i} x_{ij} = 1 \quad i = 1, \dots, n$$

$$\sum_{i,j \text{ mit } e \in T_{ij}} x_{ij} \leq C_e \quad \forall \text{ Kanten } e$$

$$x_{ij} \in \{0, 1\}$$

- ▶ Alle Routingbäume T_{ij} für Netz i aufzählen
- ▶ x_{ij} entscheidet, ob T_{ij} gewählt wird

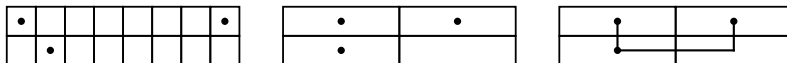


- ▶ Problem: Sehr, sehr viele Möglichkeiten
⇒ Sehr, sehr lange Laufzeit
- ▶ Ansatz zur Beschleunigung:
 - ▶ Hierarchischer Ansatz
 - ▶ Burstein, Pelavin (1983)
 - ▶ Unterteilung Rekursiv in kleinere Gebiete
⇒ bis nur noch 2×2 -Gebiete übrig sind
⇒ Kleine Anzahl (11) an möglichen Verdrahtungsbäumen
- ▶ Alle Varianten haben gemeinsam:
Ergebnis ist potenziell nicht mehr optimal

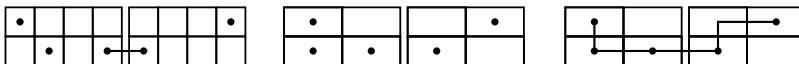
Hierarchisches ILP

Beispiel

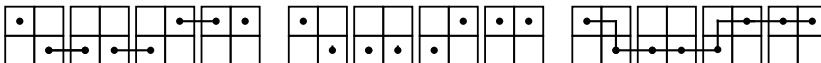
Level 1



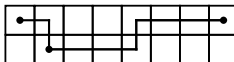
Level 2



Level 3



Lösung:





- ▶ McMurchie, Ebeling (1995); Verbesserung: Swartz, Betz, Rose (1998)
- ▶ Zielarchitektur: FPGA
 - ⇒ Begrenzte Routingressourcen
 - ⇒ Feste Kanalbreite
 - ⇒ Verdrahtbarkeit höchste Priorität
Geschwindigkeit nachrangig
- ▶ Ideen:
 - ▶ Sequentiell
 - ▶ 2 Phasen: Signal Routing, Globales Routing
 - ▶ *Rip-up and re-route*

- ▶ *Marktwirtschaft*: Angebot und Nachfrage
 - ▶ Stark nachgefragte Ressource (Metallsegmente, Pins, ...) sind teuer
 - ▶ Wenig nachgefragte sind billig
 - Haben oft Nachteile (sind z.B. langsamer)
 - ▶ Verschiedene *Verbraucher* (=Netze) akzeptieren unterschiedliche Preise
- ▶ Versuche Gesamtbedarf zu decken (d.h. alle Netze werden verdrahtet)



- ▶ Verdrahte jedes Netz für sich alleine
 - ▶ Mit den aktuellen Ressourcenkosten
 - ▶ Ignoriere die Ressourcenbegrenzungen
- ▶ Zähle Mehrfachbelegungen
- ▶ Grundlage für Nachfrageberechnungen
- ▶ Solange Mehrfachbelegungen
 - ▶ Erhöhe Kosten für stark nachgefragte Ressourcen
 - ▶ Verwerfe gesamte Verdrahtung
 - ▶ Verdrahte nochmal mit den neuen Kosten
- ▶ Sollte nach 30-45 Iterationen konvergieren



Signal Router:

- ▶ Verdrahtet einzelne Netze
- ▶ Herkömmlicher Maze Router
 - ▶ Verbesserungen/Alternativen möglich

Globaler Router:

- ▶ Verdrahtet gesamte Schaltung

```
globalrouter() begin
    count := 0 ;
    repeat
        | foreach Netz n do
            | signalrouter(n) ;
            count++ ;
    until (!sharedressources ||
        count > limit);
    if count > limit then
        | return "unrouteable" ;
```


Gerade konstruiertes Routing für
Netz n

```
Tree<RtgRsrc> signalrouter(Net n) begin
```

```
  Tree<RtgRsrc> RT ;
```

```
  RtgRsrc i, j, w, v := nil ;
```

```
  PriorityQueue<int,RtgRsrc> PQ ;
```

```
  HashMap<RtgRsrc,int> PathCost ;
```

```
  i := n.source() ;
```

```
  RT.add(i, ()) ;
```

```
  PathCost[*] :=  $\infty$  ;
```

```
  PathCost[i] := 0 ;
```

```
  foreach SinkTerminal j in n.sinks() do
```

```
    _ /* route Verbindung zur Senke j */
```

```
  return (RT) ;
```

Wellenausbreitung

```
Tree<RtgRsrc> signalrouter(Net n) begin
```

```
    Tree<RtgRsrc> RT ;
```

```
    RtgRsrc i, j, w, v := nil ;
```

```
    PriorityQueue<int,RtgRsrc> PQ ;
```

```
    HashMap<RtgRsrc,int> PathCost ;
```

```
    i := n.source() ;
```

```
    RT.add(i, ()) ;
```

```
    PathCost[*] :=  $\infty$  ;
```

```
    PathCost[i] := 0 ;
```

```
    foreach SinkTerminal j in n.sinks() do
```

```
        _ /* route Verbindung zur Senke j */
```

```
    return (RT) ;
```

Quelle ist Bestandteil der
Verdrahtung

```
Tree<RtgRsrc> signalrouter(Net n) begin  
  Tree<RtgRsrc> RT ;  
  RtgRsrc i, j, w, v := nil ;  
  PriorityQueue<int,RtgRsrc> PQ ;  
  HashMap<RtgRsrc,int> PathCost ;  
  i := n.source() ;  
  RT.add(i, ()) ;  
  PathCost[*] := ∞ ;  
  PathCost[i] := 0 ;  
  foreach SinkTerminal j in n.sinks() do  
    | /* route Verbindung zur Senke j */  
  return (RT) ;
```

Zunächst alles unerreichbar

```
Tree<RtgRsrc> signalrouter(Net n) begin
```

```
    Tree<RtgRsrc> RT ;
```

```
    RtgRsrc i, j, w, v := nil ;
```

```
    PriorityQueue<int,RtgRsrc> PQ ;
```

```
    HashMap<RtgRsrc,int> PathCost ;
```

```
    i := n.source() ;
```

```
    RT.add(i, ()) ;
```

```
    PathCost[*] :=  $\infty$  ;
```

```
    PathCost[i] := 0 ;
```

```
    foreach SinkTerminal j in n.sinks() do
```

```
        _ /* route Verbindung zur Senke j */
```

```
    return (RT) ;
```

Kosten von Quelle zu Quelle
sind 0

```
Tree<RtgRsrc> signalrouter(Net n) begin  
  Tree<RtgRsrc> RT ;  
  RtgRsrc i, j, w, v := nil ;  
  PriorityQueue<int,RtgRsrc> PQ ;  
  HashMap<RtgRsrc,int> PathCost ;  
  i := n.source() ;  
  RT.add(i, ()) ;  
  PathCost[*] := ∞ ;  
  PathCost[i] := 0 ;  
  foreach SinkTerminal j in n.sinks() do  
    | /* route Verbindung zur Senke j */  
  return (RT) ;
```

```
Tree<RtgRsrc> signalrouter(Net n) begin
```

```
    Tree<RtgRsrc> RT ;  
    RtgRsrc i, j, w, v := nil ;  
    PriorityQueue<int,RtgRsrc> PQ ;  
    HashMap<RtgRsrc,int> PathCost ;  
    i := n.source() ;  
    RT.add(i, ()) ;  
    PathCost[*] :=  $\infty$  ;  
    PathCost[i] := 0 ;  
    foreach SinkTerminal j in n.sinks() do  
        | /* route Verbindung zur Senke j */  
    return (RT) ;
```

PathFinder Signal Router

Ganze bisherige Route ist
Ausgangspunkt

```
foreach SinkTerminal j in n.sinks() do
  PQ.clear();
  foreach v in RT.nodes() do
    PQ.add(0, v);
  repeat
    v := PQ.removeLowestCostNode();
    if v ≠ j then
      foreach w in v.neighbors() do
        if PathCost[w] > (PathCost[v] + w.cost()) then
          PathCost[w] := PathCost[v] + w.cost();
          PQ.add(PathCost[w], w);
      until v = j;
    while !(v in RT.nodes()) do
      w := v.findCheapestNeighbor(PathCost);
      RT.add(v,(w,v));
      v.updateCost();
      v := w;
```

PathFinder Signal Router



- ▶ Kostenbasierte Wellenausbreitung
- ▶ Kosten \neq Distanz
- ▶ Verdrahtungsressourcen sind persistent (z.B. globale Variablen)
- ▶ $w.cost()$ wird über alle Netze berechnet
 - ▶ Über mehrere Aufrufe vom Signal Router
 - ▶ Auch über mehrere Iterationen vom Global Router

```
foreach SinkTerminal j in n.sinks() do
    PQ.clear();
    foreach v in RT.nodes() do
        PQ.add(0, v);
    repeat
        v := PQ.removeLowestCostNode();
        if v  $\neq$  j then
            foreach w in v.neighbors() do
                if PathCost[w] > (PathCost[v] + w.cost()) then
                    PathCost[w] := PathCost[v] + w.cost();
                    PQ.add(PathCost[w], w);
    until v = j;
    while !(v in RT.nodes()) do
        w := v.findCheapestNeighbor(PathCost);
        RT.add(v, w, v);
        v.updateCost();
        v := w;
```




Pfadrückverfolgung

```
foreach SinkTerminal j in n.sinks() do
  PQ.clear();
  foreach v in RT.nodes() do
    PQ.add(0, v);
  repeat
    v := PQ.removeLowestCostNode();
    if v ≠ j then
      foreach w in v.neighbors() do
        if PathCost[w] > (PathCost[v] + w.cost()) then
          PathCost[w] := PathCost[v] + w.cost();
          PQ.add(PathCost[w], w);
      until v = j;
    while !(v in RT.nodes()) do
      w := v.findCheapestNeighbor(PathCost);
      RT.add(v,(w,v));
      v.updateCost();
      v := w;
```

- ▶ `v.updateCost()` aktualisiert die Daten
- ▶ Ressource jetzt benutzt \Rightarrow für Nachfolger teurer

```
foreach SinkTerminal j in n.sinks() do
  PQ.clear();
  foreach v in RT.nodes() do
    PQ.add(0, v);
  repeat
    v := PQ.removeLowestCostNode();
    if v  $\neq$  j then
      foreach w in v.neighbors() do
        if PathCost[w] > (PathCost[v] + w.cost()) then
          PathCost[w] := PathCost[v] + w.cost();
          PQ.add(PathCost[w], w);
    until v = j;
  while !(v in RT.nodes()) do
    w := v.findCheapestNeighbor(PathCost);
    RT.add(v,(w,v));
    v.updateCost();
    v := w;
```



- ▶ Fließt via Kostenfunktion $v.cost()$ ein

$$c_v = b_v \cdot p_v$$

b_v : Basiskosten eines Knotens v

- ▶ Zunächst $b_v = 1$

p_v : Verteuerungsfaktor (penalty factor)

- ▶ Aktuelle Kosten für v
- ▶ Erfasst hohe Nachfrage
- ▶ Beginnt klein, wächst mit der Zeit

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

$\text{occupancy}(v)$: Belegungszahl der Ressource v

$\text{capacity}(v)$: Belegungskapazität der Ressource v

p_{fak} : Ist = 0.5. Nach jeder Iteration des globalen Router $\times 2$

- ▶ Bei jeder Netzänderung $\text{occupancy}(v)$ aktualisieren
(passiert in $v.updateCost()$)

PathFinder

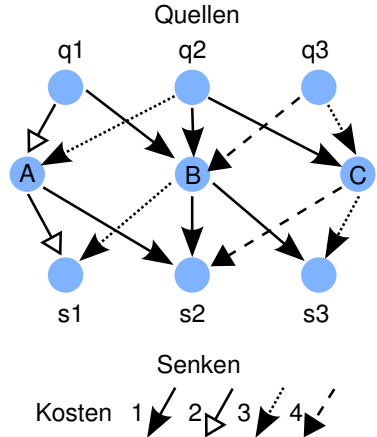
Beispiel p_v

- ▶ Mit Maze-Router abhängig von Reihenfolge der Netze

1,2,3 Gesamtkosten = 17

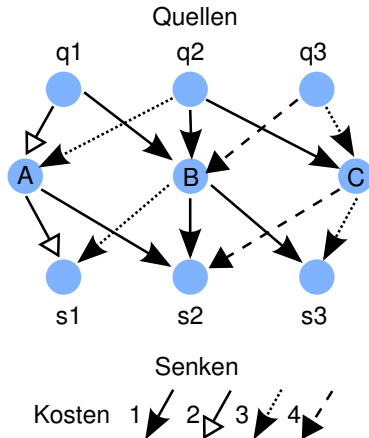
2,1,3 Gesamtkosten = 15

3,2,1 Unroutbar!



PathFinder

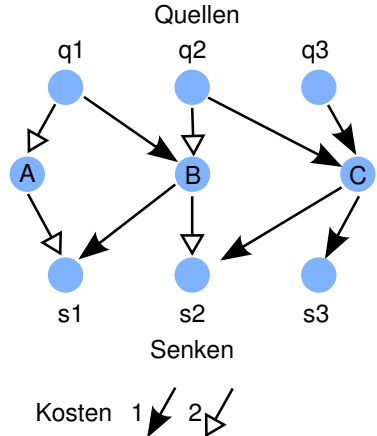
Beispiel p_v



PathFinder

Beispiel h_v

- ▶ Routingreihenfolge 1,2,3
- ▶ C doppelt belegt
- ▶ Lösung: Netz 1 muss für Netz 2 platz machen
- ▶ Problem hierbei: 1 ist gar nicht behindert, geht also selbst mit p_v nicht freiwillig



PathFinder

Historische Überbelegung h_v

- ▶ p_v reicht alleine nicht aus
- ▶ Besseres *Gedächtnis* einführen
 - ▶ Historische Überbelegungen (historic congestion) erhöhen aktuellen Preis
 - ▶ h_v **akkumuliert** alle Mehrfachbelegungen
 - ▶ p_v sieht nur aktuelle
 - ▶ Kostenfunktion erweitern:

$$c_v = b_v \cdot p_v \cdot h_v$$

- ▶ Aktualisiere einmal pro Global Router Iteration (Startwert $h(v)^1 = 1$):

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

PathFinder

Basiskosten b_v

- ▶ Idee: Verzögerung der Routing Ressourcen einfließen lassen
 - ▶ Führt aber zu Verschlechterung!
- ▶ Besser: Feste Kosten
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
- ▶ Vorschlag:
 - ▶ Input Pins $b_v = 0.95$
 - ▶ Alle anderen Elemente $b_v = 1$

PathFinder

Finaler Globaler Router



Graph<RtgRsrc> RRG ;

globalrouter(Set<Nets> N) **begin**

 HashMap<Net, Tree<RtgRsrc> NRT ;

 count := 0 ;

 pv := 0.5 ;

repeat

foreach Netz $n \in N$ **do**

 NRT[n].unroute() ;

 NRT[n] := signalrouter(n) ;

 pv := 2 * pv ;

 count++ ;

foreach r in RRG.Edges **do**

 r.updateHistory() ;

 r.updateWith(pv) ;

until (!sharedresources || count > limit);

if count > limit **then**

 return "unrouteable" ;

Routing Ressource Graph

PathFinder

Finaler Globaler Router

```
Graph<RtgRsrc> RRG ;
globalrouter(Set<Nets> N) begin
  HashMap<Net, Tree<RtgRsrc> NRT ;
  count := 0 ;
  pv := 0.5 ;
  repeat
    foreach Netz n ∈ N do
      NRT[n].unroute() ;
      NRT[n] := signalrouter(n) ;
    pv := 2 * pv ;
    count++ ;
    foreach r in RRG.Edges do
      r.updateHistory() ;
      r.updateWith(pv) ;
  until (!sharedresources || count > limit);
  if count > limit then
    return "unrouteable" ;
```

rip up-Funktion muß p_v
aktualisieren

PathFinder

Finaler Globaler Router



```
Graph<RtgRsrc> RRG ;
globalrouter(Set<Nets> N) begin
  HashMap<Net, Tree<RtgRsrc> NRT ;
  count := 0 ;
  pv := 0.5 ;
  repeat
    foreach Netz n ∈ N do
      NRT[n].unroute() ;
      NRT[n] := signalrouter(n) ;
    pv := 2 * pv ;
    count++ ;
    foreach r in RRG.Edges do
      r.updateHistory() ;
      r.updateWith(pv) ;
  until (!sharedresources || count > limit);
  if count > limit then
    return "unrouteable" ;
```

h_v aktualisieren

PathFinder

Finaler Globaler Router



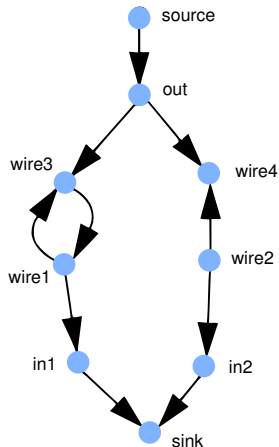
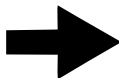
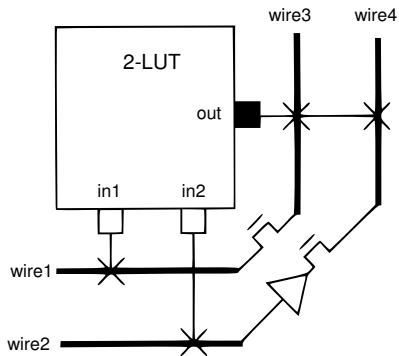
```
Graph<RtgRsrc> RRG ;
globalrouter(Set<Nets> N) begin
  HashMap<Net, Tree<RtgRsrc> NRT ;
  count := 0 ;
  pv := 0.5 ;
  repeat
    foreach Netz n ∈ N do
      NRT[n].unroute() ;
      NRT[n] := signalrouter(n) ;
    pv := 2 * pv ;
    count++ ;
    foreach r in RRG.Edges do
      r.updateHistory() ;
      r.updateWith(pv) ;
  until (!sharedresources || count > limit);
  if count > limit then
    return "unrouteable" ;
```

Gesamtkosten aktualisieren

- ▶ Fundamentale Datenstruktur
- ▶ Modelliert vorhandene Ressourcen und Verbindungsnetzwerk
- ▶ Knoten: Konkrete Verdrahtungs Ressourcen
 - ▶ Leitungen (Verdrahtungssegmente)
 - ▶ Pins
- ▶ Kanten: Möglichen Verbindungen dazwischen
 - ▶ Schalter (Pass-Transistoren, bidirektional)
 - ▶ Buffer (unidekretional)
- ▶ Äquivalente Pins:
 - ▶ Outputs: Source-Knoten
 - ▶ Inputs: Sink-Knoten
 - ▶ Fassungsvermögen (capacity): Anzahl der In/Out-Pins

Routing-Ressource-Graph

Beispiel



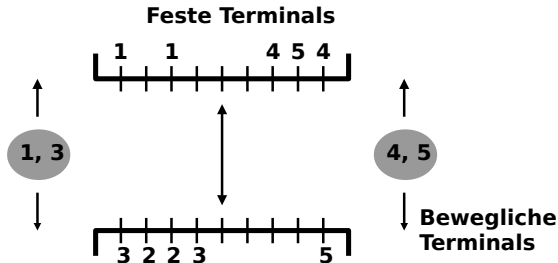
- ▶ Ausbau auf Verzögerung:
 - ▶ Erweitern der Kostenfunktion um Criticality Abart

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

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

- ▶ $d_{u,v}$: Verzögerung von u nach v
- ▶ Verbesserung der Suche (ohne Verzögerungsorientierung)
 - ▶ Alte Wellenfront in PQ nicht verwerfen
funktioniert **nur** bei reiner Verdrahtungsorientierung
 - ▶ Aufteilung der Fläche in *Bins*, und nur Punkte im Bin des Ziels expandieren
 - ▶ Suche nur auf Bounding-Box (+ etwas *Luft*) beschränken
 - ▶ Gerichtete Tiefensuche auf das Ziel zu

- ▶ Verdrahtung von Netzen in rechteckigem Kanal



- ▶ Oben und unten feste Terminals, links und rechts bewegliche
- ▶ Ziel: Minimiere Fläche (min. Länge, min. Vias)

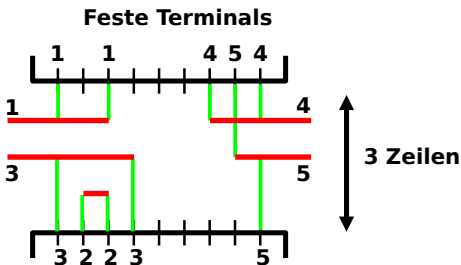


- ▶ Variante: Switchbox-Verdrahtung
 - ▶ Alle Terminals und Abmessung an allen Seiten fest
- ▶ Klassisches Modell:
 - ▶ Verdrahtung läuft auf Einheitsraster
 - ▶ Zwei Verdrahtungsebenen
 - ▶ Getrennt für horizontale und vertikale Segmente
 - ▶ Vermindert übersprechen
 - ▶ Kleinerer Lösungsraum (schnellere aber potentiell schlechtere Lösung)
 - ▶ Ein (1) horizontales Segment pro Netz
Ausnahme: Bei Konfliktauflösung 2 H-Segmente
- ▶ Mögliche Erweiterungen:
Rasterlos, 45°-Verbindungen, mehr als 2 Ebenen, Doglegs, ...

Kanalverdrahtung

Beispiel im klassischen Modell

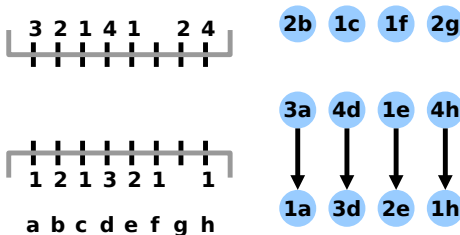
- ▶ Verdrahtung von Netzen in rechteckigem Kanal



Kanalverdrahtung

Vertikale Einschränkungen

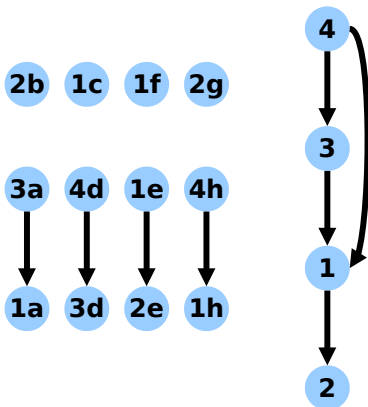
- ▶ Zwei *gegenüberliegende* Terminals
 - ▶ Oberes Segment in den Kanal **muß** über unterem Segment im Kanal liegen (sonst Kurzschluß)
 - ▶ Vertikaler Einschränkungsgraph (Vertical Constraint Graph/VCG)
 - ▶ VCG alleine: Wenig aussagekräftig
- Ein verbundenes Knotenpaar pro gegenüberliegenden Terminals



Kanalverdrahtung

Vertikale Einschränkungen

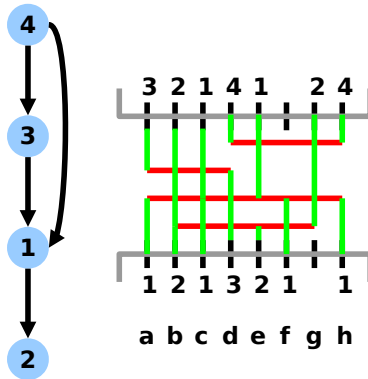
- ▶ Zusätzliche Forderung im klassischen Modell
 - ▶ Alle Terminals eines Netzes laufen auf *einem* horizontalem Segment
- ⇒ Alle Segmente eines Netzen enden in *einer* Zeile
- ▶ Darstellung im VCG
 - ▶ Verschmelzen der Terminal-Knoten zu einem Knoten pro Netz



Kanalverdrahtung

Vertikale Einschränkungen

- ▶ Eindeutige Lösung des Beispiels



Kanalverdrahtung

Vertikale Einschränkungen



- ▶ Hier gezeigt: Extremformen von VCGs
 - ▶ Vollständig verschmolzen/getrennt
- ▶ Auch möglich: Zwischenstufen
 - ▶ Ein Knoten pro horizontalem Segment
- ▶ Was tun bei Zyklen im VCG?
 - ▶ Mehr als ein H-Segment pro Netz notwendig
 - ▶ Lösung: Knoten auftrennen!
 - ▶ Führt zu VCG-Zwischenformen



- ▶ Einschränkungen

 - Vertikale: VCG

 - Horizontale: Im klassischen Modell:

 - ▶ Keine Überlappung zwischen H-Segmenten verschiedener Netze in gleicher Zeile (sonst Kurzschluß)

- ▶ Falls nur vertikale Einschränkungen:

 - ▶ Berechnung des längsten Pfades

- ▶ Falls keine vertikalen Einschränkungen:

 - ▶ Keine gegenüberliegenden Terminals
⇒ Left-Edge Algorithmus (1971)

- ▶ Falls horiz. und vertikale existieren

 - ⇒ NP-Vollständig!

Kanalverdrahtung

Left-Edge Algorithmus

- ▶ Modelliere Netz i als Intervall

$$[X_{i_{min}}, X_{i_{max}}]$$

- ▶ Begrenzt durch Position der linken und rechten Terminals
 - ▶ Ausreichend, da keine vertikalen Einschränkungen
Zeile des H-Segments kann *überall* erreicht werden
 - ▶ Lokale Dichte in Spalte x : $d(x)$
Anzahl von Intervallen, die Spalte x enthalten
 - ▶ Maximale lokale Dichte $d_{max} = \max_x d(x)$ ist untere Schranke für Anzahl Zeilen
 - ▶ Optimale Lösung:
 - ▶ Packe nicht-überlappende Intervalle in eine Zeile
 - ▶ Greedy
- ⇒ Minimale Anzahl von Zeilen

Left-Edge Algorithmus

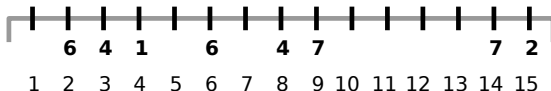
Algorithmus

```
leftEdge(list<interval> i_list) begin
```

```
    /* intervalle in i_list nach aufsteigender linker Koordinate sortiert */ set<set<interval > > solution ;  
    set<interval> row ;  
    interval f ;  
    solution :=  $\emptyset$  ;  
    while !i_list.empty() do  
        f := i_list.head() ;  
        i_list := i_list.tail() ;  
        row :=  $\emptyset$  ;  
        repeat  
            row := row  $\cup$  {f} ;  
            f := erstes Element in i_list ohne Überlappung mit f ;  
            i_list.remove(f) ;  
        until f = nil ;  
        solution := solution  $\cup$  {row} ;  
    return (solution) ;
```

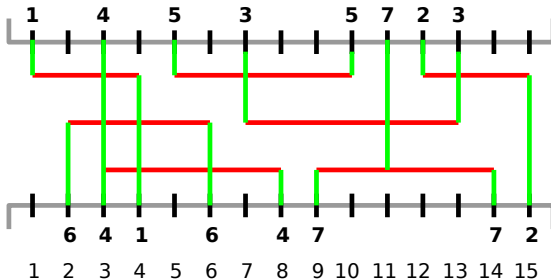
Left-Edge Algorithmus

Beispiel



Left-Edge Algorithmus

Beispiel



Left-Edge Algorithmus

Komplexität

- ▶ Entspricht als graphentheoretisches Problem dem Färbeproblem:
Bestimme minimale Anzahl von Farben, so dass keine benachbarten Knoten die gleiche Farbe haben (Farben \leftrightarrow Zeilen)
 - ▶ Klassisches NP-vollständiges Problem
 - ▶ Für Intervallgraphen aber in P
 - ▶ n Intervalle
 - ▶ d Zeilen
 - ▶ Sortieren nach linker Koordinate $\mathcal{O}(n \log n)$
 - ▶ Äußere Schleife: d Durchläufe
 - ▶ Innere Schleife: max. n Intervalle betrachten
- $\Rightarrow \mathcal{O}(n \log n + d n)$
Kann noch verbessert werden: $\Theta(n)$

Kanalverdrahtung

Robuster Kanalrouter (Yoeli)



- ▶ Heuristik (Yoeli, 1991)
- ▶ Iteriert über alle Zeilen im Kanal
- ▶ Verkleinert Problem mit jeder Iteration
- ▶ Wechselt zwischen oberster/unterster Zeile
Arbeitet sich zur Kanalmitte vor
- ▶ Zwei Phasen
 1. Berechnen von Gewichten für Netze
Wie gut wäre aktuelle Zeile für Netz?
 2. Selektion von Untermenge mit maximalem Gewicht
Heuristik bei Verletzung vertikaler Einschränkungen

Robuster Kanal-Router

Berechnung der Gewichte w_i für Netz i

1. Falls i Spalten der maximalen Dichte überspannt:

$$w_i = w_i + B \quad (B \text{ groß})$$

Hoffe auf Verringerung der max. Dichte, unabhängig von Seite (steepest descent)

2. Falls i ein Terminal auf der *aktuellen* Seite (oben/unten) auf Spalte x hat:

$$w_i = w_i + d(x) \quad (\forall \text{ Spalten } x)$$

Bevorzuge Netze mit Terminals auf aktueller Seite

3. Für alle Spalten x bei denen eine vertikale Einschränkung verletzt würde:

$$w_i = w_i - K d(x) \quad (5 \leq K \leq 10)$$

Bestrafe verletzte Einschränkung



- ▶ Regeln typisch für Heuristiken
 - ▶ Robust
 - ▶ Unempfindlich gegen kleine Änderungen
 - ▶ Nach Bestimmung der Gewichte
 - ▶ Finde Netz-Untermenge mit maximalem Gewicht, die in dieselbe Zeile passen
 - ▶ Ohne Verletzung horizontaler Einschränkungen
 - ▶ Verwende Intervallgraphen
 - ▶ Kante zwischen Knoten überlappender Intervalle
 - ▶ Unabhängige Menge (Menge unverbundener Knoten)
 - ▶ Unabhängige Menge maximalen Gewichts
 - Für Intervallgraphen in P, allgemein NP-vollständig
 - ▶ Vorgehensweise: Dynamisches Programmieren
 - ▶ Konstruiere optimale Lösungen aus Teillösungen
- Komplexitätsparameter $\gamma : 1 \leq \gamma \leq \text{Kanallänge}$

Robuster Kanal-Router

Dynamisches Programmieren



- ▶ $\gamma =$ Spalte c
Betrachte nur Netze mit **rechtem** Ende $\leq c$
- ▶ Bestimme Lösung $\gamma = c$ aus Lösung $\gamma < c$
 - ▶ Altes Maximalgewicht plus Netz n mit rechtem Ende in Spalte c
Es exist. max. zwei solcher Netze (Terminals oben und unten)
 - ▶ n Teil der optimalen Lösung, falls
Gewicht von n plus Gewicht bestehender Netze **ohne Überlappung** mit $n \geq$ max. Gewicht ohne n



- ▶ Für Spalte c ausgewähltes Netz merken
 - ▶ In `selected_net[c]`
 - ▶ Kann leer sein (= 0, kein neues dazugekommen)
 - ▶ Letztes (=rechtes) Netz immer in Lösung
 - ▶ Dann nach links suchen
 - Nach **nicht-überlappendem** Netz
 - ▶ Wiederhole bis linker Rand erreicht



- ▶ Annahme: d_{max} (untere Schranke) Durchgänge reichen
⇒ Wäre optimale Lösung
- ▶ Nur der *Versuch* der Vermeidung von V-Konflikten, keine Garantie
- ▶ Falls V-Konflikte unvermeidbar:
 - ▶ Entferne ein oder mehrere Netze (Auswahl heuristisch)
 - ▶ Verdrahte Netze mit Maze-Routing
 - ▶ Rip-up and re-route Ansatz
 - ▶ Garantiert alle keine Lösung!
- ▶ Falls sich herausstellt, dass keine Lösung:
 - ▶ Erneuter Durchlauf mit zusätzlicher Zeile
Ggf. auch zusätzlicher Spalte

Robuster Kanalrouter Algorithmus



```
RobustRouter(placed_netlist N) begin
  set<int> row ;
  seq<set<int> > S ;
  int[channel_width+1] totalwght, selected_net ;
  bool top ;
  int height, c, r, i ;
  top := true ;
  height := N.dmax() ;
  for (r := 1 ; r ≤ height ; ++r) do
    *SIEHE RECHTE SPALTE* ;
    while c > 0 do
      if (selected_net[c] != 0) then
        n := selected_net[c];
        row := row ∪ {n} ;
        c :=  $x_{n_{min}} - 1$  ;
      else
        c := c-1 ;
      end if
    end while
    S.append(row);
    top := !top;
    N := N ohne Netze in row;
  *Im Falle von V-Konflikten Maze-Routing anwenden*
```

```
foreach Netze  $i \in$  netlist N do
   $w_i := i.compute\_weight(N, top)$  ;
  totalwght[0] := 0 ;
  for (c:=1; c ≤ channel_width; ++c) do
    selected_net[c] := 0 ;
    totalwght[c] := totalwght[c-1] ;
    if ( $\exists$  Netz n:rechtes Terminal Spalte c, oben) then
      if ( $w_n + totalwght[x_{n_{min}} - 1] > totalwght[c]$ ) then
        totalwght[c] :=  $w_n + totalwght[x_{n_{min}} - 1]$ ;
        selected_net[c] := n;
      end if
    end if
    if ( $\exists$  Netz n:rechtes Terminal Spalte c, unten) then
      if ( $w_n + totalwght[x_{n_{min}} - 1] > totalwght[c]$ ) then
        totalwght[c] :=  $w_n + totalwght[x_{n_{min}} - 1]$  ;
        selected_net[c] := n ;
      end if
    end if
  end for
  row :=  $\emptyset$  ;
  c := channel_width ;
```

► Ggf. Wiederholung mit erhöhter Breite/Länge



- ▶ Wo sind die Kanäle? Channel Definition Problem (CDP)
- ▶ Reihenfolge für die Verdrahtung der Kanäle?
Channel Ordering Problem (COP)
- ▶ Standzellen-Entwurf
 - ▶ Bei rein lokaler Verdrahtung
⇒ Reihenfolge egal, Kanäle sind zwischen den Zellreihen
 - ▶ Bei nicht lokaler: Globale Verdrahtung
 - ▶ Trennt Netze auf einzelne Kanäle auf
 - ▶ Übergang zwischen den Kanälen
(Feedthroughs, reservierte Verdrahtungsebenen)
 - ▶ Idee: RSMT
- ▶ Bei Slicing Floorplans einfache Lösung:
 - ▶ Schnittlinien bilden Kanäle
 - ▶ Form/Größe abhängig von Reihenfolge
 - ▶ Reihenfolge definiert durch Schnittbaum (DFS mit Post-Order Durchlauf)



- ▶ Verdrahtung
- ▶ Globale, lokale und Kanalverdrahtung
- ▶ Lösungsalgorithmen:
 - ▶ Maze-Algorithmus (Lees Algorithmus)
 - ▶ Liniensuche
 - ▶ A*-Suche
 - ▶ Pathfinder
 - ▶ ILP-Formulierung
 - ▶ Left-Edge Algorithmus
 - ▶ Robuster Kanalrouter