Algorithmen im Chip-Entwurf 5

Allgemeine Platzierungsverfahren, Partitionierung und Pfad-basierte Timing-Analyse

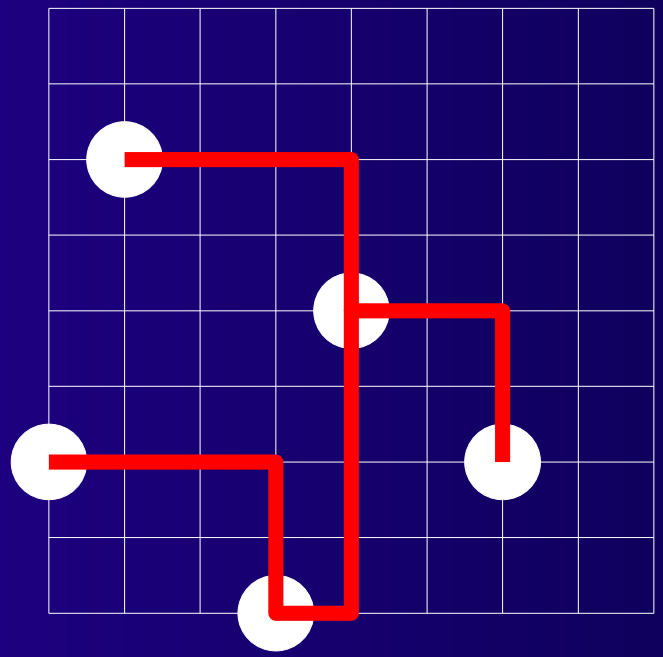
Andreas Koch
FG Eingebettete Systeme
und ihre Anwendungen
TU Darmstadt

Übersicht

- Andere Längenmetriken
  - Half-Perimeter schon kennengelernt
  - Nun Überblick über Alternativen
- Andere Platzierungsprobleme
  - FPGA/MPGA schon kennengelernt, nun auch andere
- Allgemeine Platzierungsverfahren
  - Ein konkretes Verfahren schon kennengelernt: VPR
  - Nun Überblick über Alternativen
- Kernighan-Lin
  - Partitionierung via MinCut
- Timing-Analyse
  - Mehrere kritische Pfade
- Zusammenfassung

Längenmetriken 2

- **Minimaler Rechtwinkliger Überspannender Baum (MRST)**

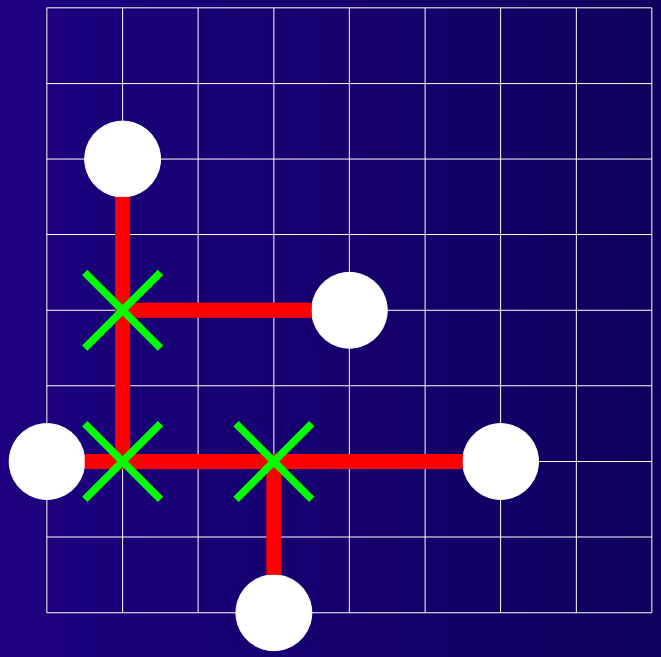


$L_R = 19$

- **Sonderfall von MRST im Raster**
  - In P via Prim's Algorithmus (im Buch 3.4.4)
    - Vollständiger planarer Graph

Längenmetriken 3

- **Rechtw. Steiner-minimaler Baum (RSMT)**



$L_S = 15$

$L_R / L_S = 1{,}26$

- **RSMT-Berechnung ist NP-vollständig**
  - Annäherung durch MRST: max. 1,5x so lang
  - Bessere Näherungen existieren

Längenmetriken 4

- **Quadratischer Euklidischer Abstand**
  - Arbeitet auf Zellen, nicht auf Netzen
    - Für Clique-Modell geeignet

$$\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\gamma_{ij}\left[(x_i-x_j)^2+(y_i-y_j)^2\right]$$

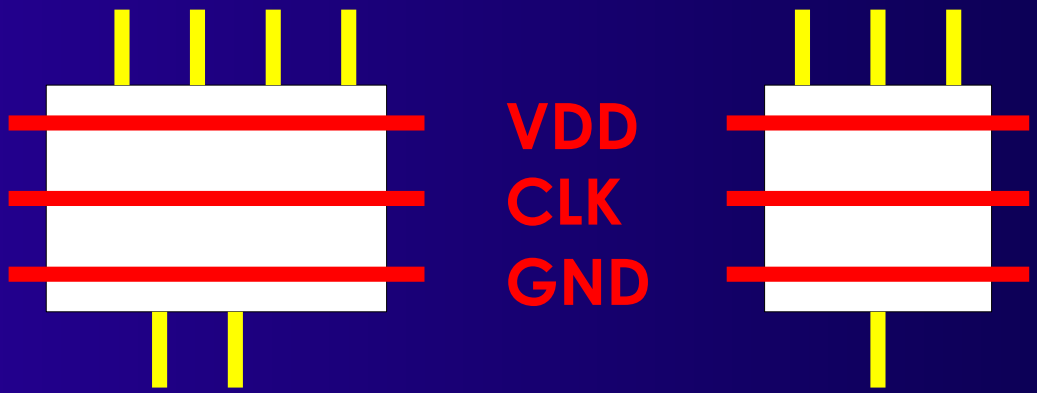
- γ_{ij}
  - =0 wenn $(v_i, v_j) \notin E$
  - $=|(v_i,v_j)|$: Gewichtet nach Anzahl Kanten
  - $< |(v_i,v_j)|$: nicht nur Einzelleitungen

Platzierungsprobleme

- **Standardzellen**
  - Konstantes Layout auf Zell-Ebene
    - Kaum Variationen bei Abmessungen
  - Semi-Custom
    - Anordnen von Zellen fester Funktionalität
    - Bibliothek: NAND4, DFF, ...
- **Building Block**
  - Anordnen von flexiblen Zellen
  - Teilweise Full-Custom möglich
    - Beliebiges Layout auf Zell-Ebene
- **MPGA/FPGA**
  - Auf vorgegebene Strukturen

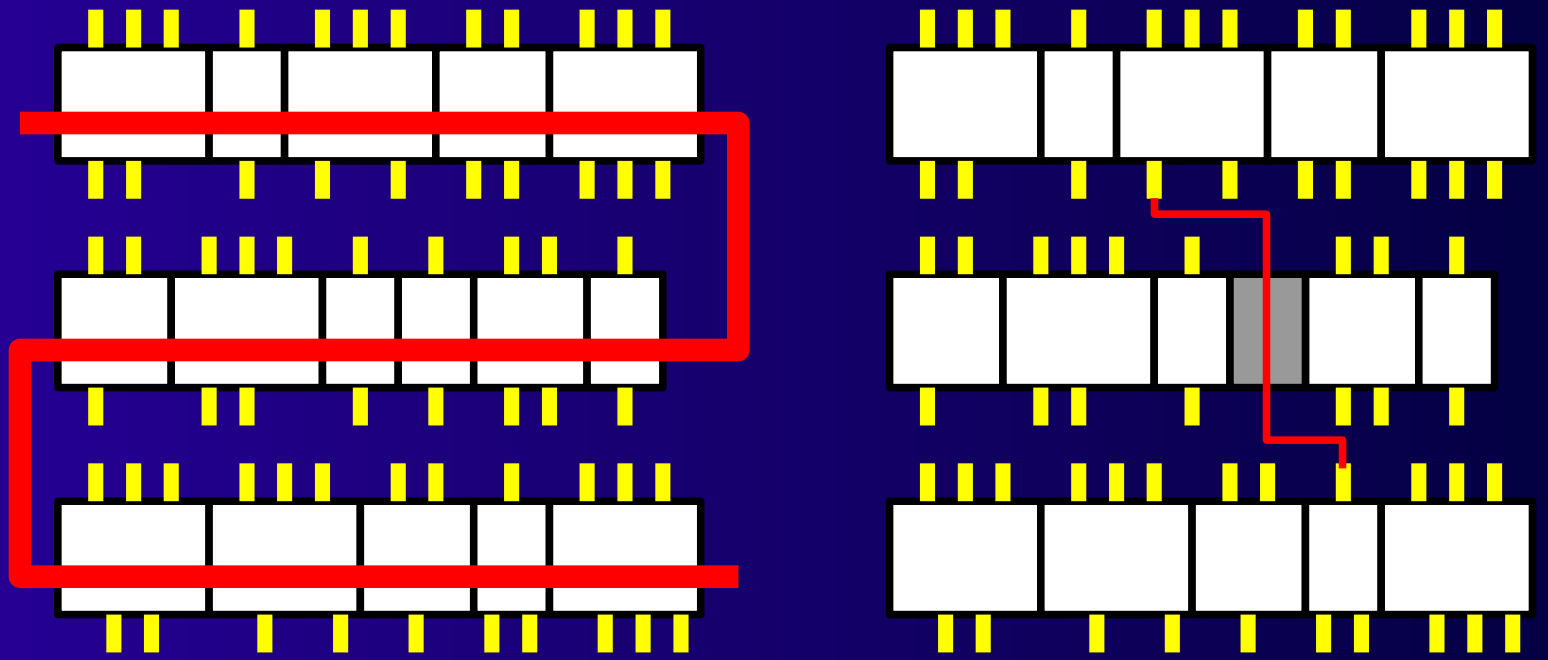
Standardzellen 1

- **Standardzellen (Semi-Custom)**
  - Kleinere Schaltungen (Gatter) aus Bibliothek
  - Festes Layout
    - Grösse
    - Terminal-Anordnung
  - Anreihbar in Zeilen
    - Logistische Signale



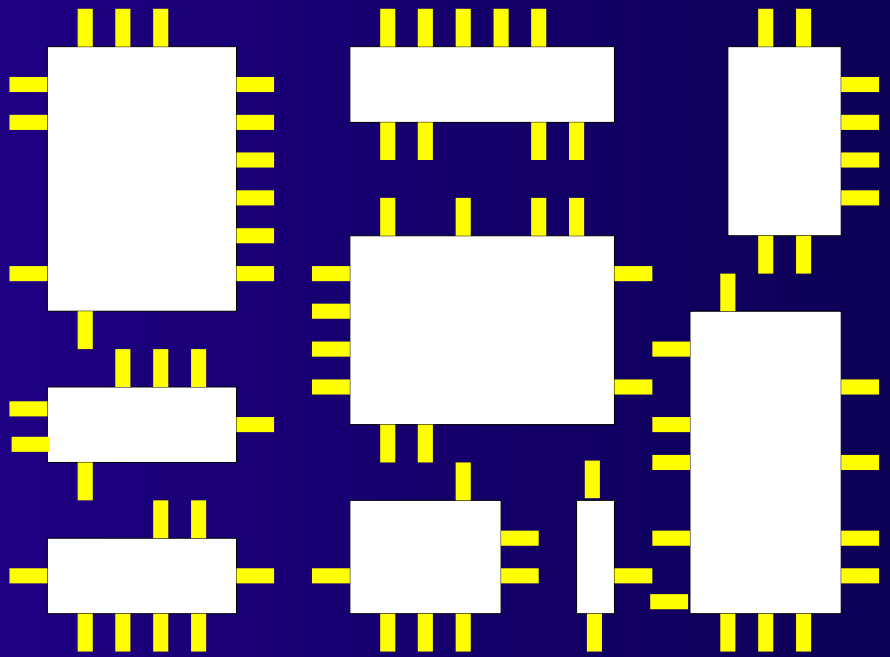
Standardzellen 2

- Zeilenweise Anordnung
- Verdrahtung zwischen Zeilen
- Ausnahmen
  - Angrenzende Verbindungen (abutment)
  - Durchleitungen (feedthroughs)



Building Blocks 1

- **Mehr Flexibilität**
  - Kann auch Full-Custom Teile enthalten
  - Automatisch generierte Blöcke (z.B. RAM)
- **Verdrahtungskanäle an allen Seiten**



Platzierungsverfahren 1

- **Konstruktiv**
  - Zellkoordinaten sind nach einmaligem Plazierungsschritt fest
- **Iterativ**
  - Zellkoordinaten können beliebig oft geändert werden
- **Kombination**
  - Konstruktive Startlösung
  - Dann iterative Verbesserung

Mögliche Optimierungsziele 1

- Minimale Verdrahtungsfläche
- Minimale Verdrahtungslänge
- Schnellste Schaltung
  - Timing-driven
- Anzahl von Leitungen durch Schnittlinie
- Verdrahtbare Schaltung
- Geringes Übersprechen
  - Zwischen Leitungen

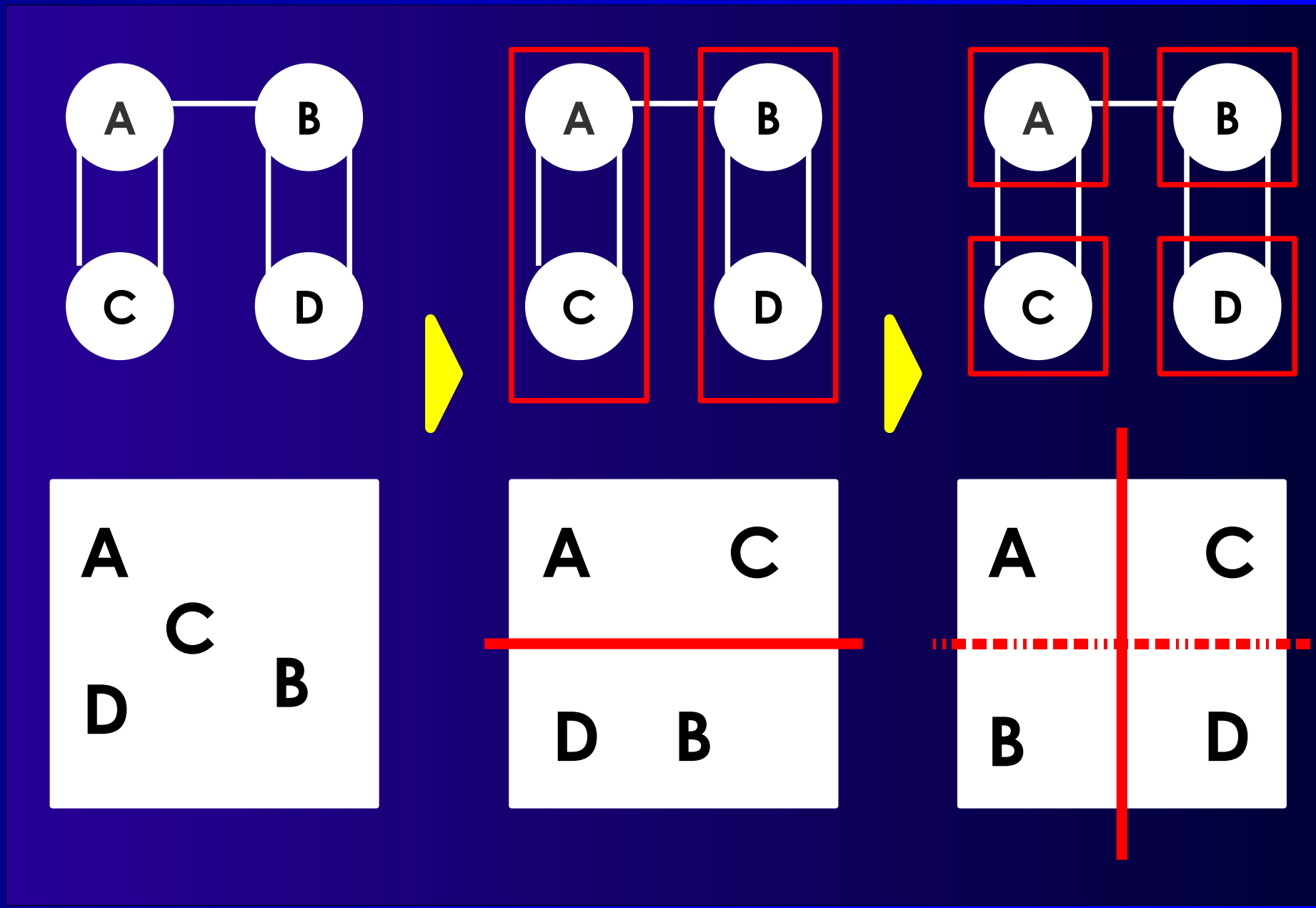
Konstruktive Platzierung 1

- Viele Methoden
- Top-Down Verfahren
  - Starten mit kompletter Schaltung
  - Aufteilen in immer kleinere Probleme
  - Beispiel: Min-Cut
- Bottom-Up Verfahren
  - Beginnen mit einzelnen Zellen
  - Zusammenfügen von Teillösungen
  - Beispiel: Clustering

Min-Cut Platzierung 1

- **Idee**
  - Teile Schaltung in zwei Hälften auf
- **Minimiere die Anzahl der Netze dazwischen**
  - MinCut: Minimiere Gewicht *durchschnittener* Netze
- **Teile auch Layoutfläche nach jedem Schnitt**
- **Ordne Schaltungshälften Layouthälften zu**
  - Horizontal und Vertikal, i.d.R. abwechselnd
- **Wiederhole bis Abbruch**
  - z.B. Nur noch eine Zelle in Partition

Min-Cut Plazierung 2



Min-Cut Plazierung 3

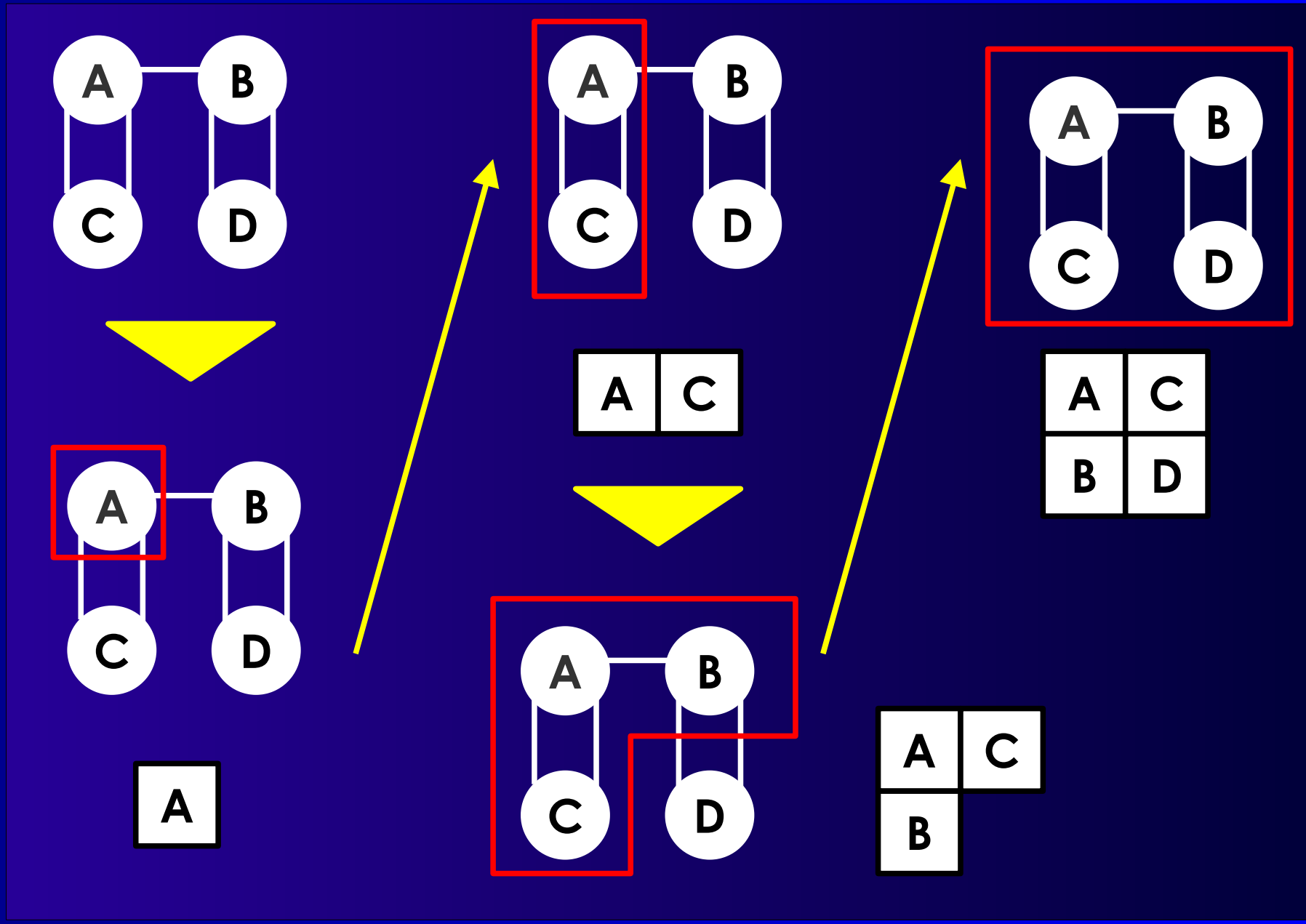
- **Aufteilen des Graphen**
  - Standardalgorithmen
- **Zuweisung der Partitionen an Layout**
  - Einschließlich Richtung der Aufteilung
  - Verschiedene Heuristiken
  - Beispiele:
    - Berücksichtige bereits zugewiesene Partitionen
    - Berücksichtige Chip-I/O-Pads

Platzierung mit Clustering 1

- Beginne mit einer Startzelle als Cluster
- Finde angeschlossene Zelle(n)
- Ordne Zelle(n) „nahe“ um Cluster an
- Füge neue Zellen dem Cluster hinzu

- Entscheidungen:
  - Welche Zellen(n) hinzufügen?
  - Auf welche Art nahegelegen anordnen?

Platzierung mit Clustering 2



Iterative Verbesserung 1

- „Kleine“ Veränderung bestehender Lösung
  - Ändere die Position von Zelle(n)
  - Falls besseres Ergebnis: Immer übernehmen
  - Schlechter: Unter Umständen übernehmen

  ➔ Abhängig von Suchverfahren!

Iterative Verbesserung 2

```
iterative_improvement () {
    s := initial_configuration();
    c := s.cost();
    while (!stop()) {
            s' := s.perturb();
            c':= s'.cost();
            if ( c.accept( c' ) )
                s := s';
    }
}
```

- initial_configuration
- cost
- stop
  - z.B. #Iterationen
  - komplexer möglich
- accept
  - Nachbarsuche
  - Simulated Annealing
  - Tabu-Suche

Iterative Verbesserung 3

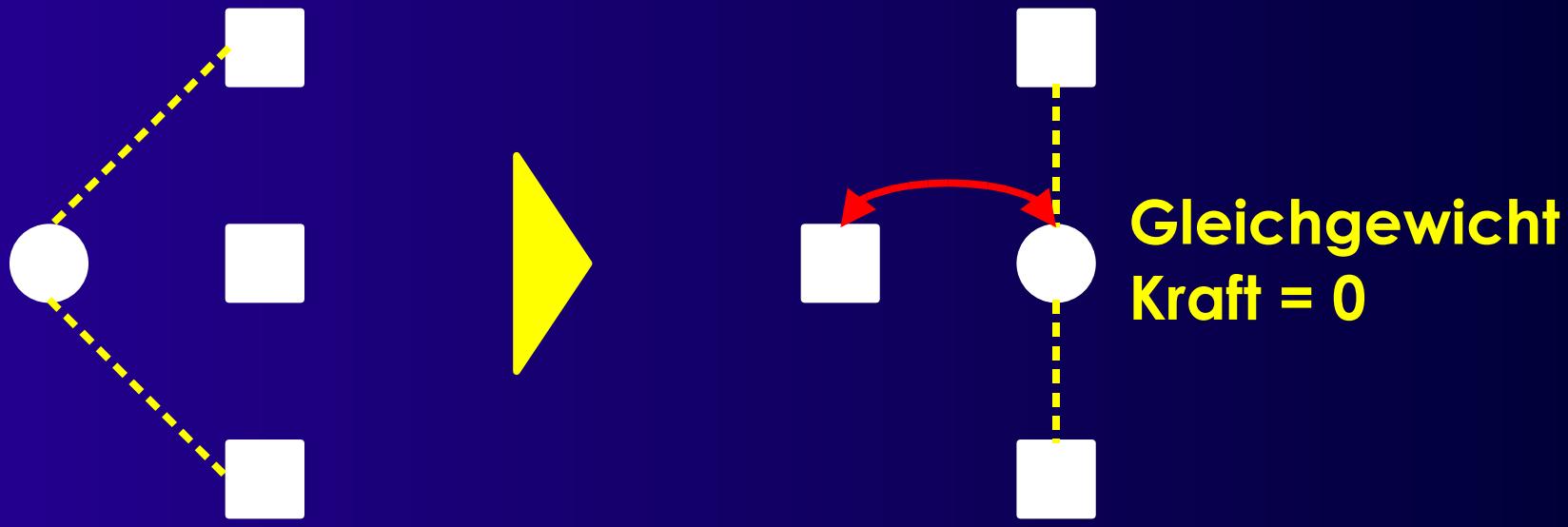
- perturb
- Bei UPP: einfach, z.B. Positionstausch
- Bei Standardzellen oder Building Block:
  - Unterschiedliche Zellgrößen, Überlappung möglich

Iterative Verbesserung 4

- Vorgehensweisen
- Überlappung erlaubt, aber höhere Kosten
  - Bereinige beste gefundene Lösung am Ende
    - Möglicherweise drastische Verschlechterung
  - Beseitige Überlappung direkt nach jedem Zug
    - Bei BB sehr aufwendig, bei SC machbar
    - Aber so genauere Kostenberechnung möglich
- Erzeuge nur überlappungsfreie Lösungen
  - Züge unter Umständen sehr viel aufwendiger

Iterative Verbesserung 5

- **Alternativen zu zufälligem Zellaustausch**
  - Kräfte-gesteuerte Auswahl des Partners
  - Bestimme Idealposition der Zelle
    - Reduziere durch Netze ausgeübte Anziehungskraft
  - Tausche dann mit Zelle auf Idealposition



Iterative Verbesserung 6

- **Berechnung des Schwerpunktes**
  - Verwendet Cliquen-Modell $G(V, E)$
  - γ_{ij}: Gewicht von $(i,j) \in E$, $\gamma_{ij} = 0$ falls $(i, j) \notin E$
  - Bestimme Schwerpunkt (x_i^g, y_i^g) der Zelle i

$$x_i^g = \frac{\sum_j \gamma_{ij} x_j}{\sum_j \gamma_{ij}} \quad \text{Gewichteter Durchschnitt} \quad y_i^g = \frac{\sum_j \gamma_{ij} y_j}{\sum_j \gamma_{ij}}$$

  - Bewege Zelle i dorthin
  - Was tun, wenn dort schon andere Zelle liegt?
    - Bewege andere Zelle auf ihren Schwerpunkt
    - Erzeugt Folge von Zügen, ggf. Tabu-Mechanismus

Partitionierung

- Aufteilen eines Graphen
- Hier motiviert durch Plazierung
  - Min-Cut
- Andere Anwendungen
  - Aufteilen einer Schaltung auf mehrere Chips
  - Verkleinern der Problemgröße
    - Vorbearbeitung vor anderem Algorithmus
- Viele Verfahren
  - Beispiel: Kernighan-Lin

Kernighan-Lin Partitionierung 1

- **Problem**
  - Gewichteter, ungerichteter Graph $G(V,E)$
  - $|V| = 2n$
  - γ_{ab}: Gewicht von $(a,b) \in E$, $\gamma_{ab}=0$ bei $(a,b) \notin E$
  - Finde Mengen A und B mit
    - $A \cup B = V$, $A \cap B = \emptyset$, $|A| = |B| = n$
  - Minimiere

$$\sum_{(a,b)\in A\times B} \gamma_{ab}$$

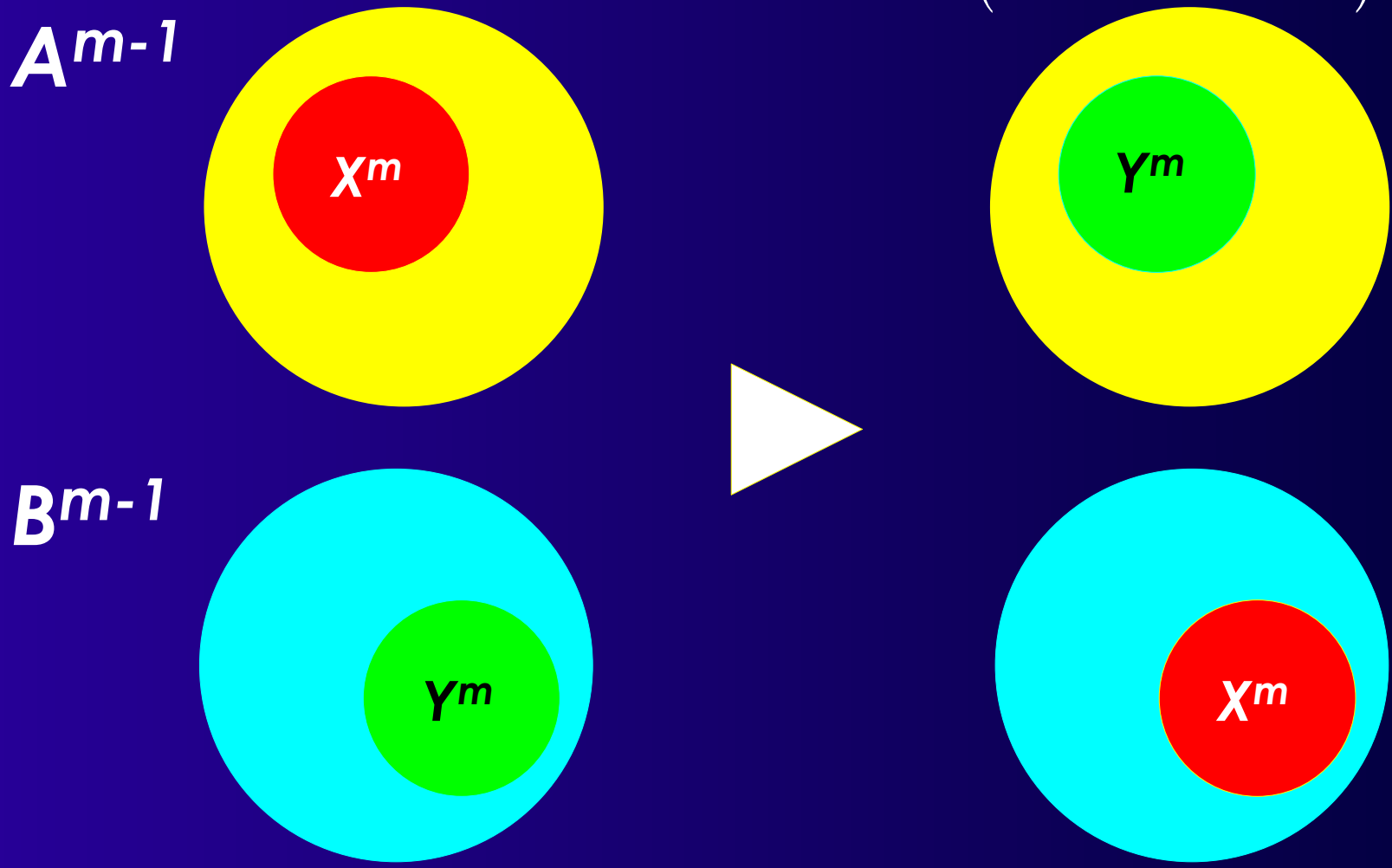
- **Arbeitet auf Cliquen-Modell**

Kernighan-Lin Partitionierung 2

- **Partitionierungsproblem ist NP-vollständig**
- **KL ist eine Heuristik**
  - Im praktischen Einsatz bewährt
- **Vorgehensweise**
  - Anfangslösung bestehend aus A^0 und B^0
    - I.d.R. nicht optimal
  - Isoliere Untermengen von A^{m-1} und B^{m-1}
  - Tausche diese aus um A^m und B^m zu bestimmen
  - Wiederhole, solange Verbesserung erreichbar

Kernighan-Lin Partitionierung 3

$$A^m = (A^{m-1} \setminus X^m) \cup Y^m$$



$$B^m = (B^{m-1} \setminus Y^m) \cup X^m$$

Kernighan-Lin Partitionierung 4

- Optimum immer in einem Schritt erzielbar
  - Bei geeignetem X^m und Y^m
- Problem: Wie X^m und Y^m bestimmen?
  - Schwer zu finden
- ➔ Suche Lösung in mehreren Schritten
  - Wiederhole, bis keine Verbesserung mehr
- Anzahl Schritte unabhängig von *n*
  - In der Praxis <= 4.

Kernighan-Lin Partitionierung 5

- Konstruktion von X^m und Y^m
- Externe Kosten

$$E_a = \sum_{y \in B^{m-1}} \gamma_{ay} \quad , a \in A^{m-1}$$

- Interne Kosten

$$I_a = \sum_{x \in A^{m-1}} \gamma_{ax} \quad , a \in A^{m-1}$$

- Analog für B

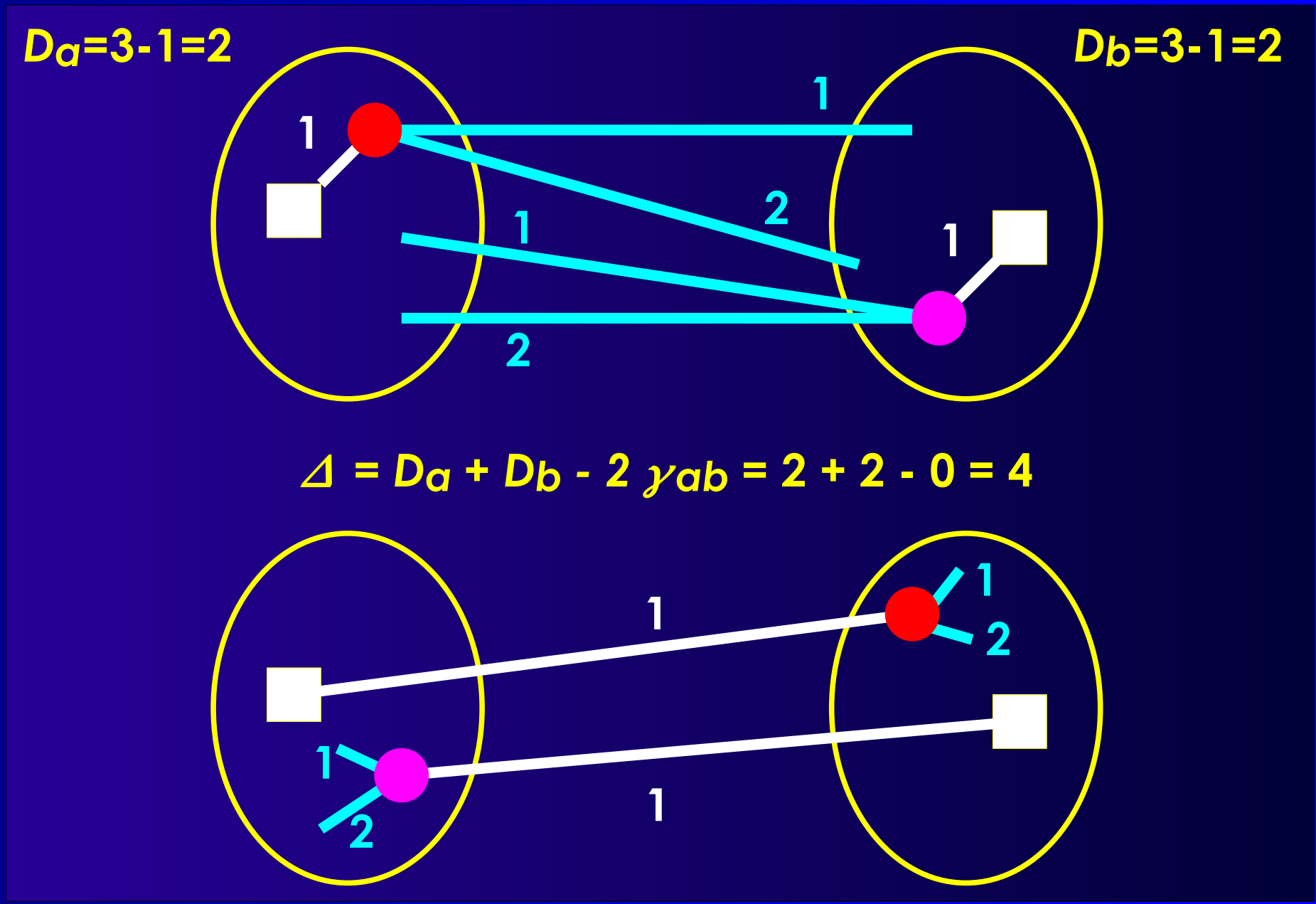
Kernighan-Lin Partitionierung 6

- $D_a = E_a - I_a$ für $a \in A^{m-1}$ (desirability)
  - >0: Knoten sollte nach *B* getauscht werden
  - <0: Knoten sollte in *A* bleiben
- Verbesserung Δ der Schnittkosten
  - Bei Austausch von $a \in A^{m-1}$ und $b \in B^{m-1}$

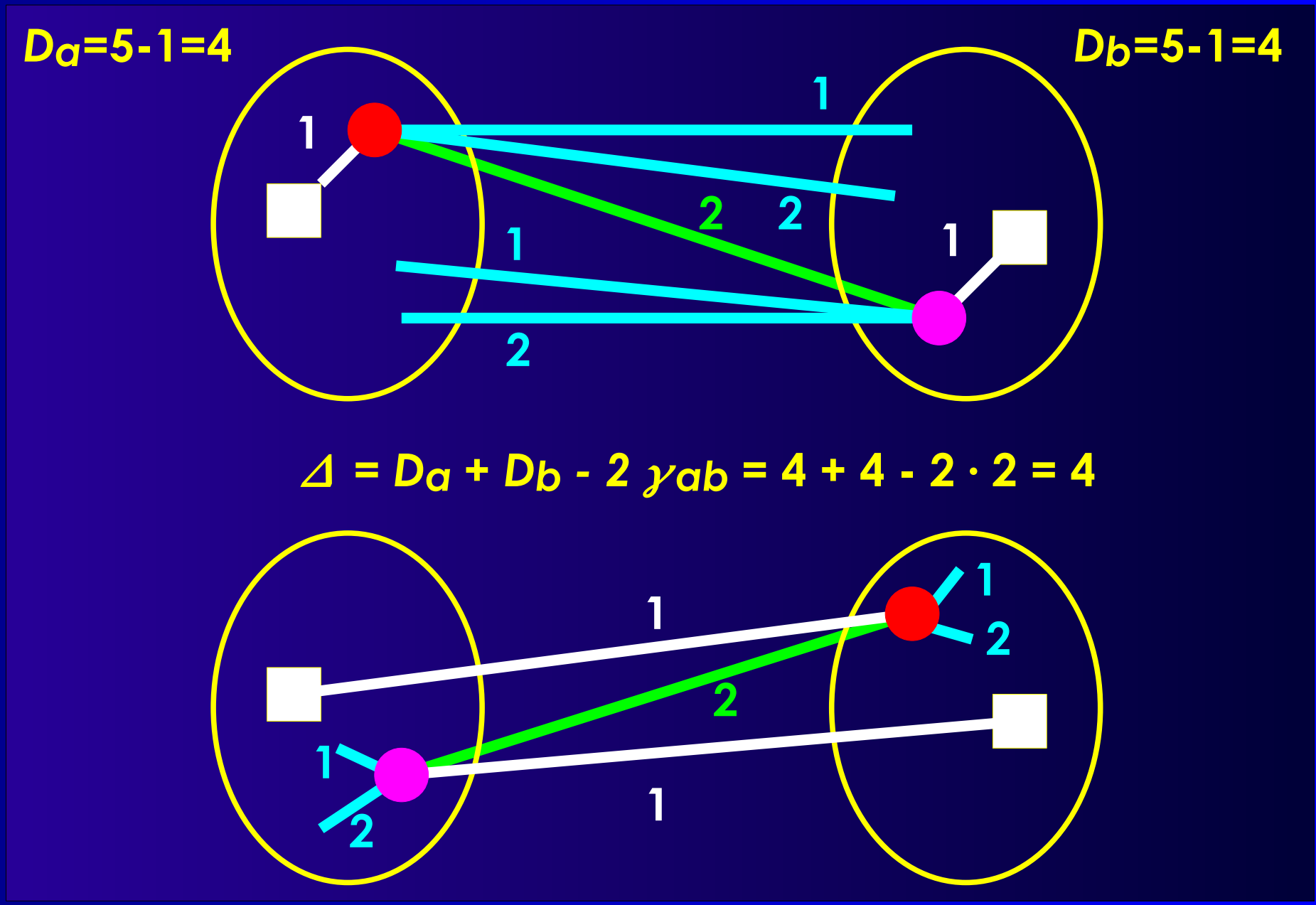
$$\Delta = D_a + D_b - 2\gamma_{ab}$$

  - Δ kann negativ sein!

Kernighan-Lin Partitionierung 7



Kernighan-Lin Partitionierung 8



Kernighan-Lin Partitionierung 9

initialize(A^0, B^0);
m := 1;
do {
  foreach $a \in A^{m-1}$
    "berechne D_a";
  foreach $b \in B^{m-1}$
    "berechne D_b"
  for (i:=1; i <= n; ++i) {
    "finde freie $a_i \in A^{m-1}$, $b_i \in B^{m-1}$ mit
      $\Delta_i := D_{ai} + D_{bi} - 2\,\gamma_{aibi}$ maximal"
    "sperre a_i und b_i"
    foreach "freies" $x \in A^{m-1}$
      $D_x := D_x + 2\,\gamma_{xai} - 2\,\gamma_{xbi}$;
    foreach "freies" $y \in B^{m-1}$
      $D_y := D_y - 2\,\gamma_{yai} + 2\,\gamma_{ybi}$;
  }
  "finde ein k mit $\sum_{i=1}^{k} \Delta_i$ ist max."
  $G := \sum_{i=1}^{k} \Delta_i$

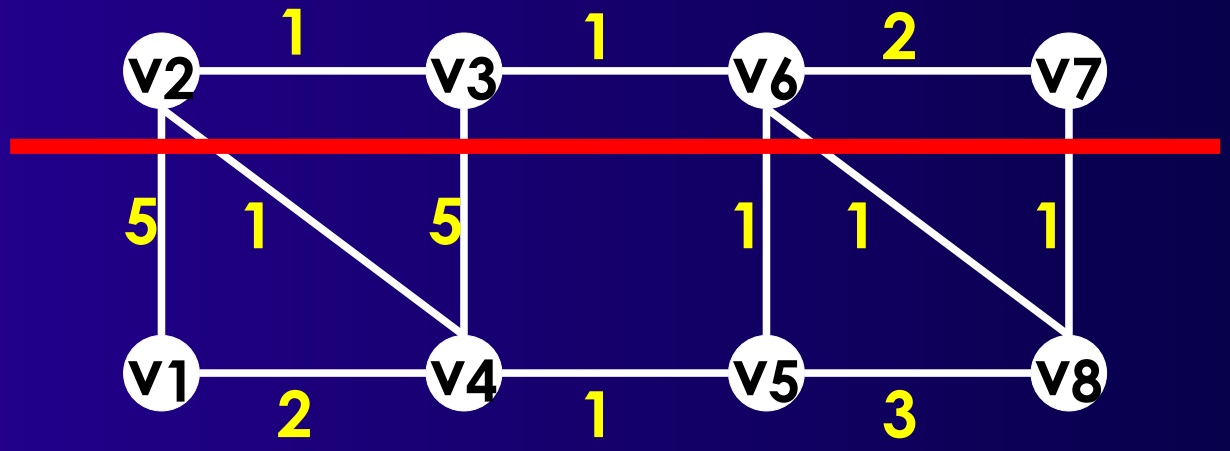
$$D_x = E_x - I_x$$
$$= E_x^{old} + \gamma_{ax} - \gamma_{bx} - (I_x^{old} - \gamma_{ax} + \gamma_{bx})$$
$$= D_x^{old} + 2\gamma_{ax} - 2\gamma_{bx}$$

  if (G > 0) {
    $X^m := \{a_1, \ldots a_k\}$;
    $Y^m := \{b_1, \ldots, b_k\}$;
    $A^m := (A^{m-1} \setminus X^m) \cup Y^m$;
    $B^m := (B^{m-1} \setminus Y^m) \cup X^m$;
    "entsperre alle Knoten in A^m and B^m"
    m := m + 1;
  }
} **while** (G > 0);

Kernighan-Lin Partitionierung 10

- Δ_i kann negativ werden
- $\sum \Delta_i$ kann zeitweise auch negativ sein
  - Bei dicht verbundenen Teilmengen
  - Keine Verbesserung bei Austausch von Einzelknoten
  - Erst bei Austauch der gesamten Teilmenge

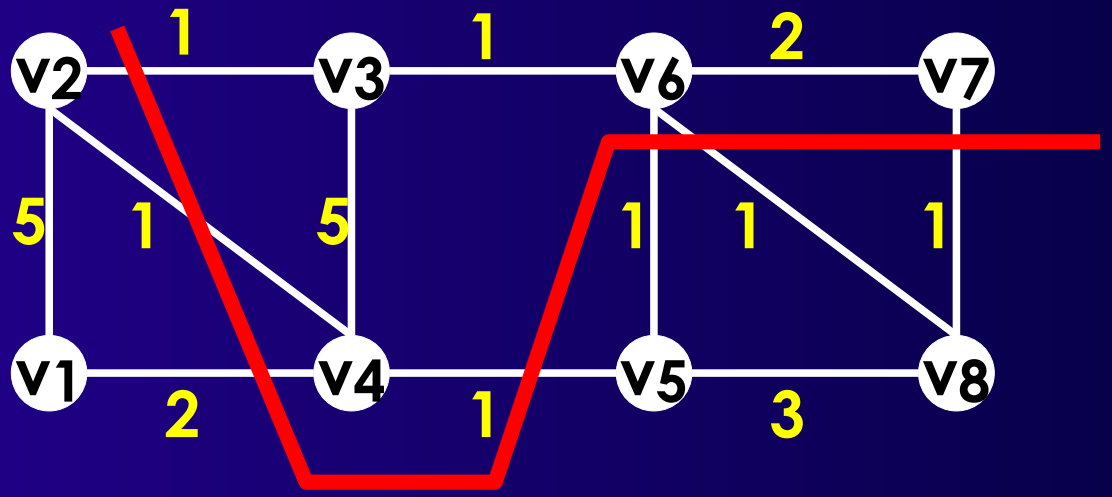
Kernighan-Lin Partitionierung 11



i	A^0 V2	V3	V6	V7	B^0 V1	V4	V5	V8	Δ_i
1	**5**	3	-1	-1	3	**3**	-3	-1	6
2		-5	-1	**-1**	-3		**-1**	-1	-2
3		-5	**1**		-3			**3**	2
4		**-3**			**-3**				-6

$\Sigma \Delta_i$: 6, 4, 6, 0

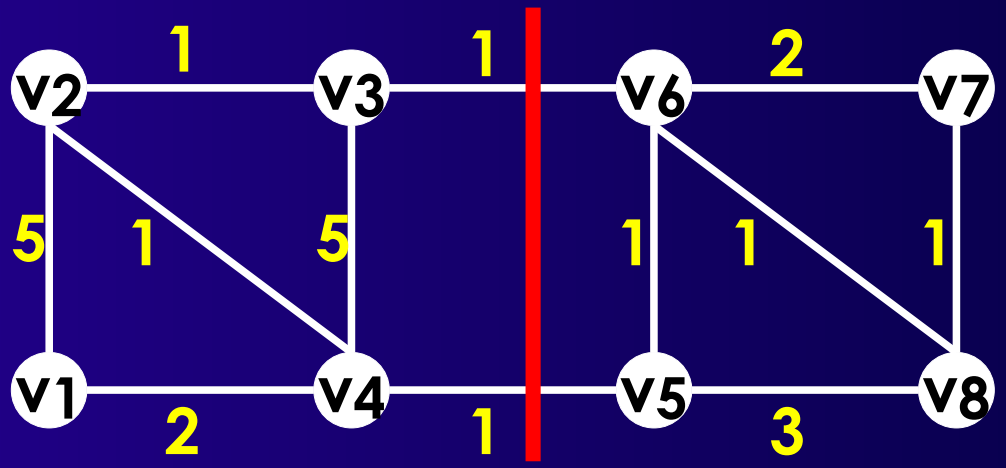
Kernighan-Lin Partitionierung 12



i	A^1				B^1				Δ_i
	v3	v4	v6	v7	v1	v2	v5	v8	
1	-5	-1	-1	-1	-3	-3	-1	-1	-2
2	5		-3	-3	-7	-5	3		8
3			-3	-3	-7	-7			-10
4				1		3			4

$\Sigma \Delta_i$: -2, 6, -4, 0

Kernighan-Lin Partitionierung 13



- Danach keine Verbesserung mehr in G
- Innere Schleife: *n* Iterationen
  - Finden des Paares mit bestem Δ: $O(n^2)$
  - Nach Δ sortiert: O(n log n)

→ $O(n^3)$ oder $O(n^2 \log n)$

Kernighan-Lin Partitionierung 14

- KL: Lokale Suche mit variabler Nachbarschaft
- Schnellere Verfahren
  - Fiduccia-Mattheyses (FM)
    - Wesentlich schneller: O(*n*)
    - Aber schlechtere Qualität der Lösungen
  - QuickCut (QC): avg. O(|E| log *n*)
    - Gleiche Qualität wie KL
- Diverse Alternativen
  - Spectral Partitioning, Multi-Level-FM, ...

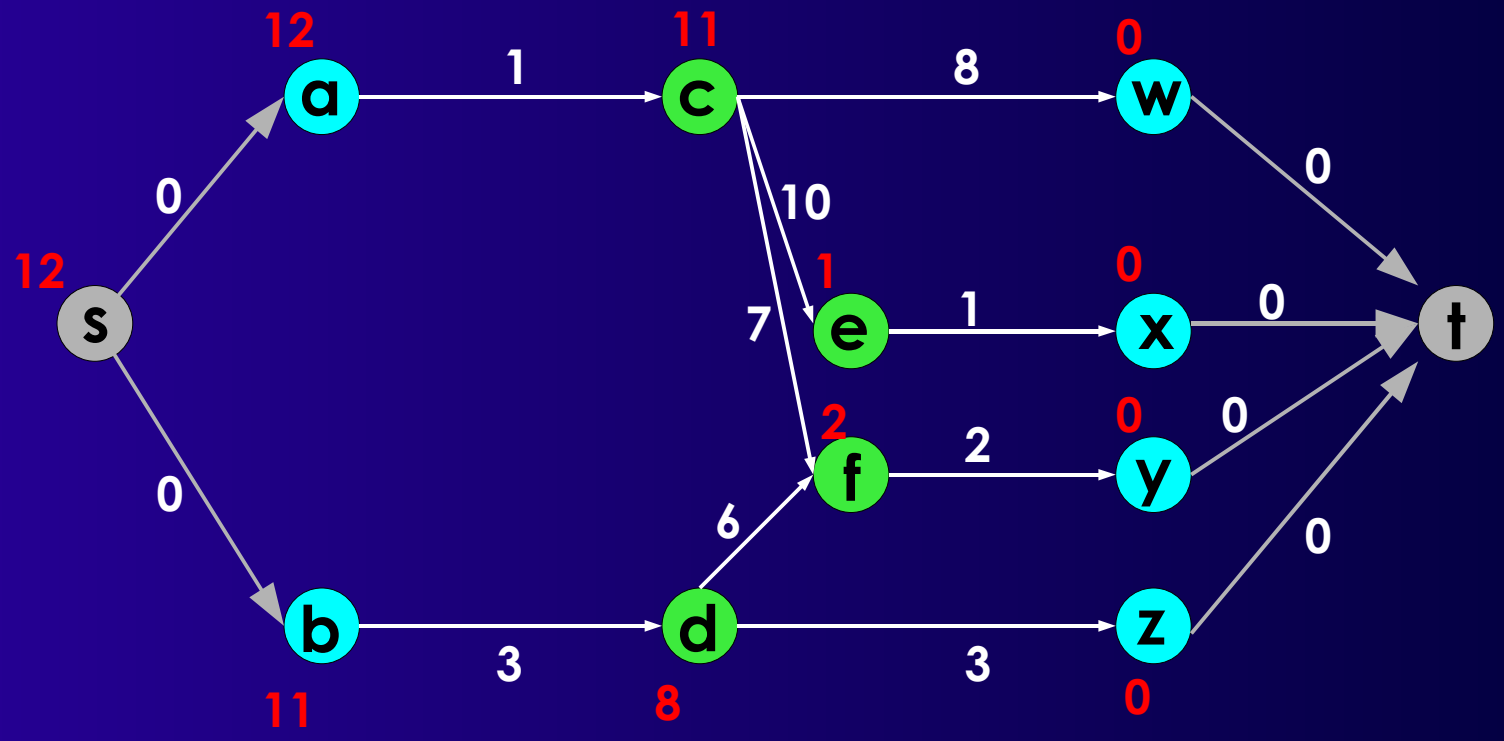
Timing-Analyse

- **Kritischer Pfad**
  - Einfach (slack=0)
  - Nächstkritischerer Pfad?
- **Vorgehensweisen**
  - Alle Pfade berechnen
    - Rechenzeit- und Speicherbedarf
  - *k* längste Pfade „en bloque“ berechnen
    - Wenig flexibel: *k* bei Start der Berechnung fest
  - Pfade inkrementell berechnen
    - Flexibel: Rechen- und Speicheraufwand reduziert
- **Idee**
  - Timing-Graph annotieren
  - Pfade aufzählen (enumerate)

Verfahren nach Ju und Saleh

- **Design Automation Conference 1991**
  - Paper auf Web-Seite
  - Details in Abschnitt 3
- **Graphannotation**
  - Längste Verzögerung zur Senke bestimmen
  - Aber auch an jeder Abzweigung merken
    - Wieviel schneller würde die Alternative sein?
- **Pfadaufzählung**
  - Beginne mit längstem Pfad
  - Wähle minimal schnellere Abzweigung
  - Erzeuge von dort ausgehend längsten Pfad
- **Vorteil**
  - Erzeugung beliebig vieler/weniger Pfade
    - Exakt an Anforderungen anpassbar

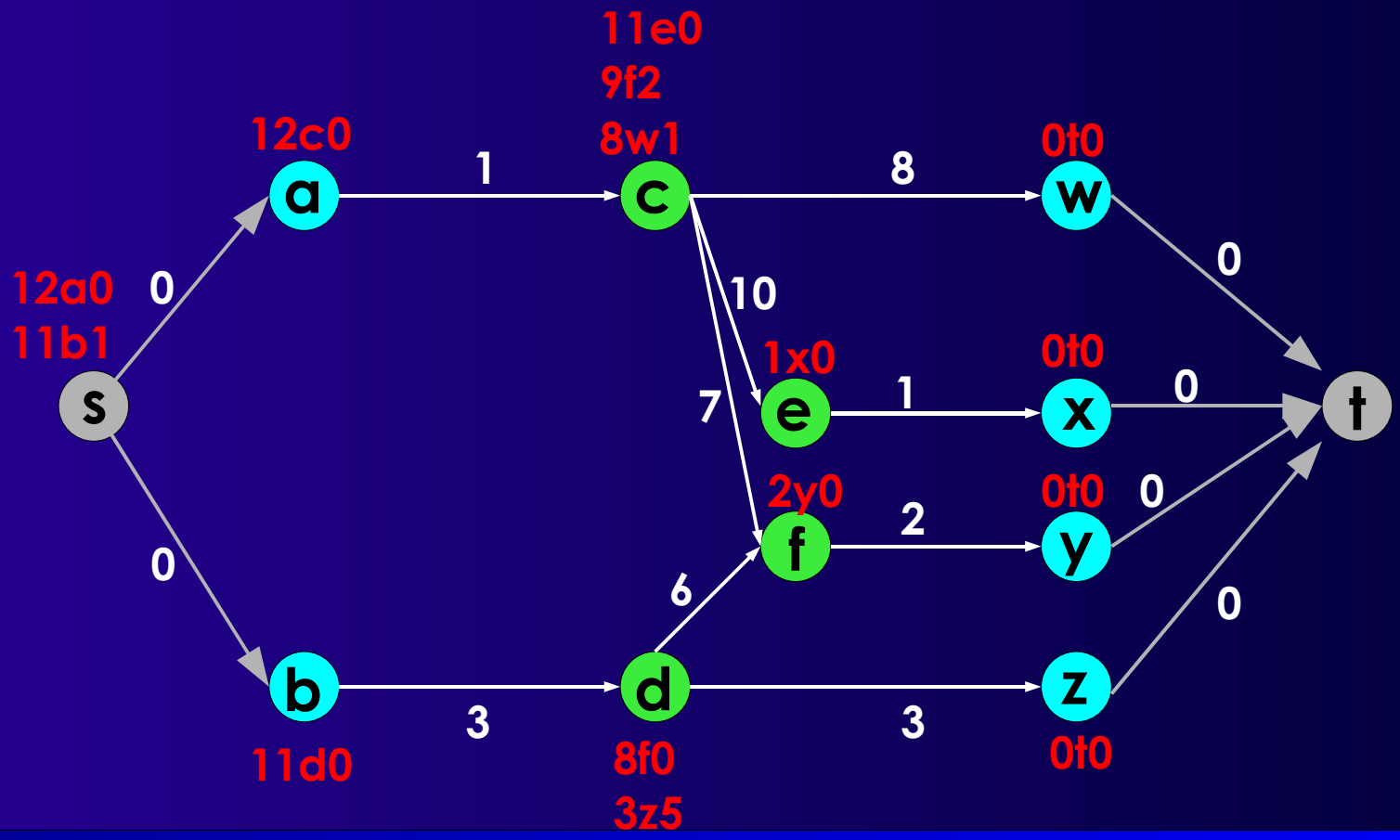
Annotation des Timing-Graphen



$$T_{sink}(u) = \underset{(u,v)\in E}{Max}\left(T_{sink}(v) + w(u,v)\right)$$

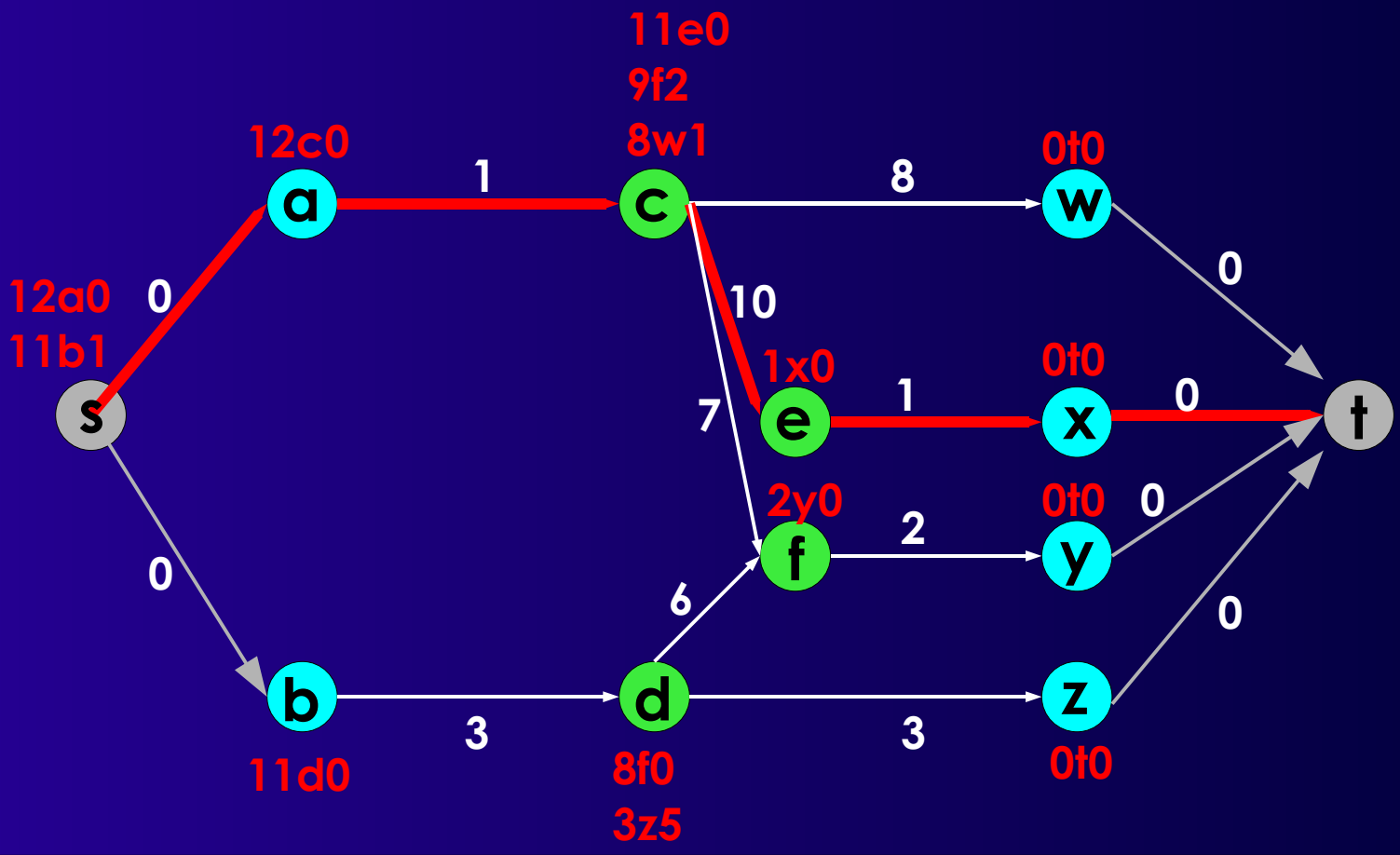
Erweiterung

- Aber zusätzlich je *u*
  - $T_{sink}(v) + w((u,v))$ von allen direkten Nachbarn *v* absteigend sortiert merken
  - benachbarte Differenzen berechnen: *branch slacks*



Längster Pfad

- Beginn bei *s*
- Dann jeweils Kante mit maximaler T_{sink}
  - Bis *t* erreicht



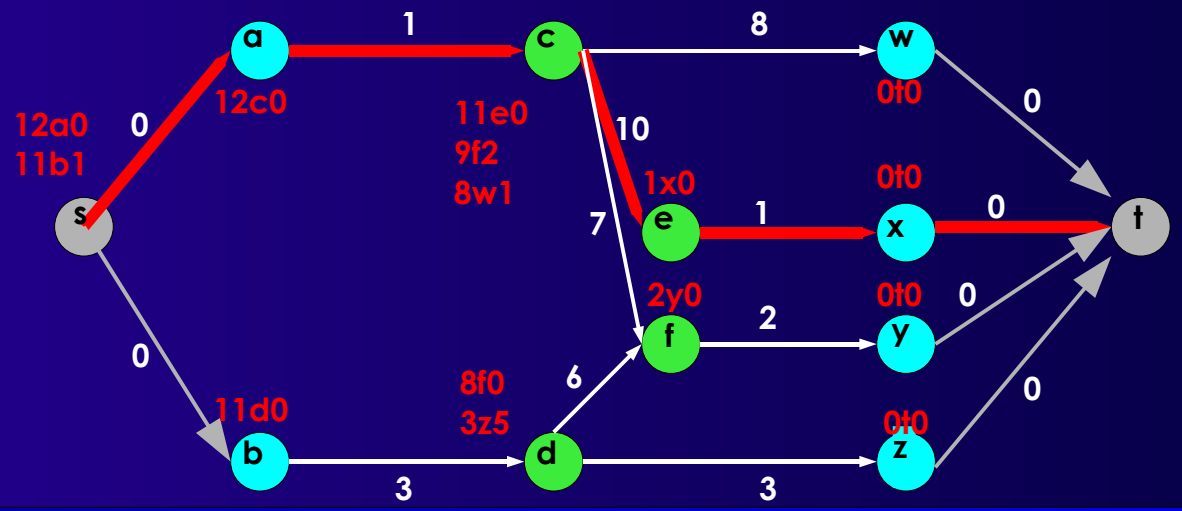
Datenstruktur

```
struct branch {
        edge            e;
        unsigned int    slack;
}

struct path {
        list<vertex>                                vertices;
        unsigned int                                totaldelay;
        ordered<branch,increasing branch.slack>     branches;
        unsigned int                                nextdelay;
}
```

ordered<branch, decreasing Tsink> succ(v)

p0=(<sacext>, 12, <(sb,1),(cf,2)>, 11)



Vorgehen 1

- longest_path(list<vertex> head)
  - Verlängert *head* zu längstmöglichem Pfad
    - Wählt dazu jeweils Nachbar mit max. T*sink*
  - Merkt sich Nachbarn mit nächstkleinerer T*sink*
    - Also: Den mit kleinstem branch slack
  - Berechnet aktuelles und nächstkleineres Delay

- branch_path(path p)
  - Zweigt an Stelle *v* mit min. branch slack von *p* ab
  - Markiert Abzweigung in *p* als „genommen“
    - Berechne nächstkleineres Delay von *p* neu
  - Berechnet nun longest_path(p.vertices+<*v*>)

Vorgehen 2

- **Kernalgorithmus**
  - Annotiere Graph mit T_{sink} und branch slacks
  - Berechne längsten Pfad p_0=longest_path(<s>)
  - Merkt sich p_0 in P
  - Wiederholt, bis genug Pfade oder Delay=0:
    - Finde p aus P mit nächstkleinerem Delay = Max > 0
    - Generiere neuen Pfad p' = branch_path(p)
      - Verwende langsamste Abzweigung (min. branch slack)
    - Nimm p' in P auf

- **P enthält danach die gesuchten Pfade**

Beispiel

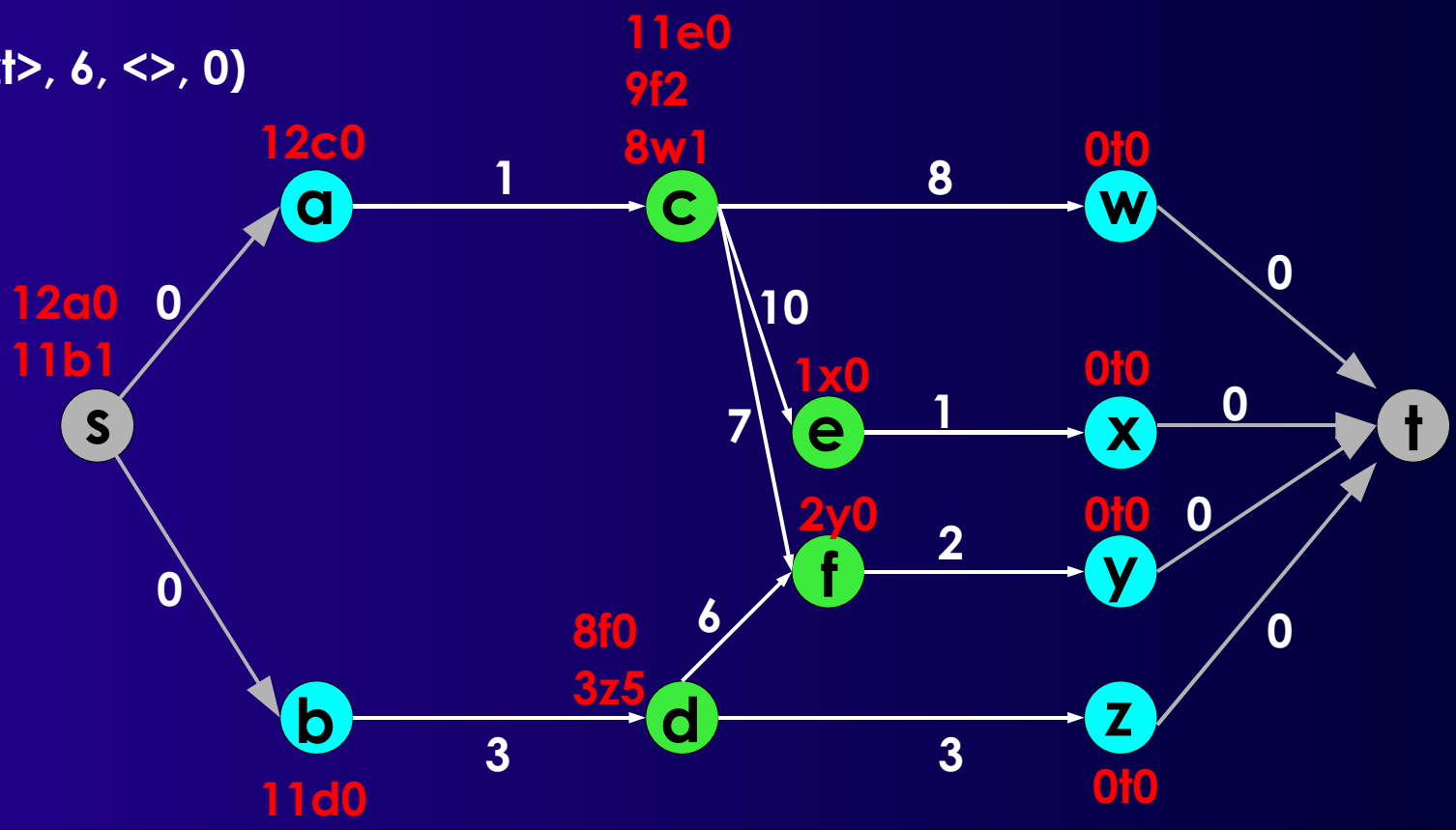
p_0 = (<sacext>, 12, <(~~sb,1~~), (~~cf,2~~)>, ~~11~~) ~~10~~ 0

p_1 = (<s|bdfyt>, 11, <(~~dz, 5~~)>, ~~4~~) 0

p_2 = (<sac|fyt>, 10, <(~~cw,1~~)>, ~~9~~) 0

p_3 = (<sac|wt>, 9, <>, 0)

p_4 = (<sbd|zt>, 6, <>, 0)



Zusammenfassung

- Neue Längenmetriken
- Platzierungsprobleme
- Kernighan-Lin MinCut-Partitionierung
- Schnelle pfadorientierte Timing-Analyse

- Papers auf Web-Seite
  - Ju & Saleh 1991: Kritische Pfadaufzählung
    - Nur Abschnitt 3 relevant