Karlsruher Institut für Technologie

Institut für Prozessdatenverarbeitung und Elektronik

Institut für Technik der Informationsverarbeitung

4. Hardware/Software Partitionierungsverfahren

Dr.-Ing. Oliver Sander
Dipl.-Inform. Leonard Masing

Hardware/Software Co-Design

Inhalt

4.1 Einführung (I)

- Implementierung einer komplexen Systemspezifikation erfordert oftmals eine Aufteilung in eine Hardware- und Softwarepartition:
  - **Reine Hardwareimplementierung:** geringe Flexibilität, hohe Leistung
  - **Reine Softwareimplementierung:** hohe Flexibilität, geringe Leistung
- Trade- off zwischen Aufteilung in Hardware und Software nötig.
- Kostenfunktionen als Optimierungskriterien:
  - Kommunikationsaufwand zwischen den Systemkomponenten.
  - Flächenbedarf der Hardware (Logikzellen, Register, Verdrahtung).
  - Flexibilität bezüglich späterer Änderungen der Spezifikation.
  - Datendurchsatz.
  - Leistungsverbrauch.
  - Minimierung der benötigten Speicherbandbreite.

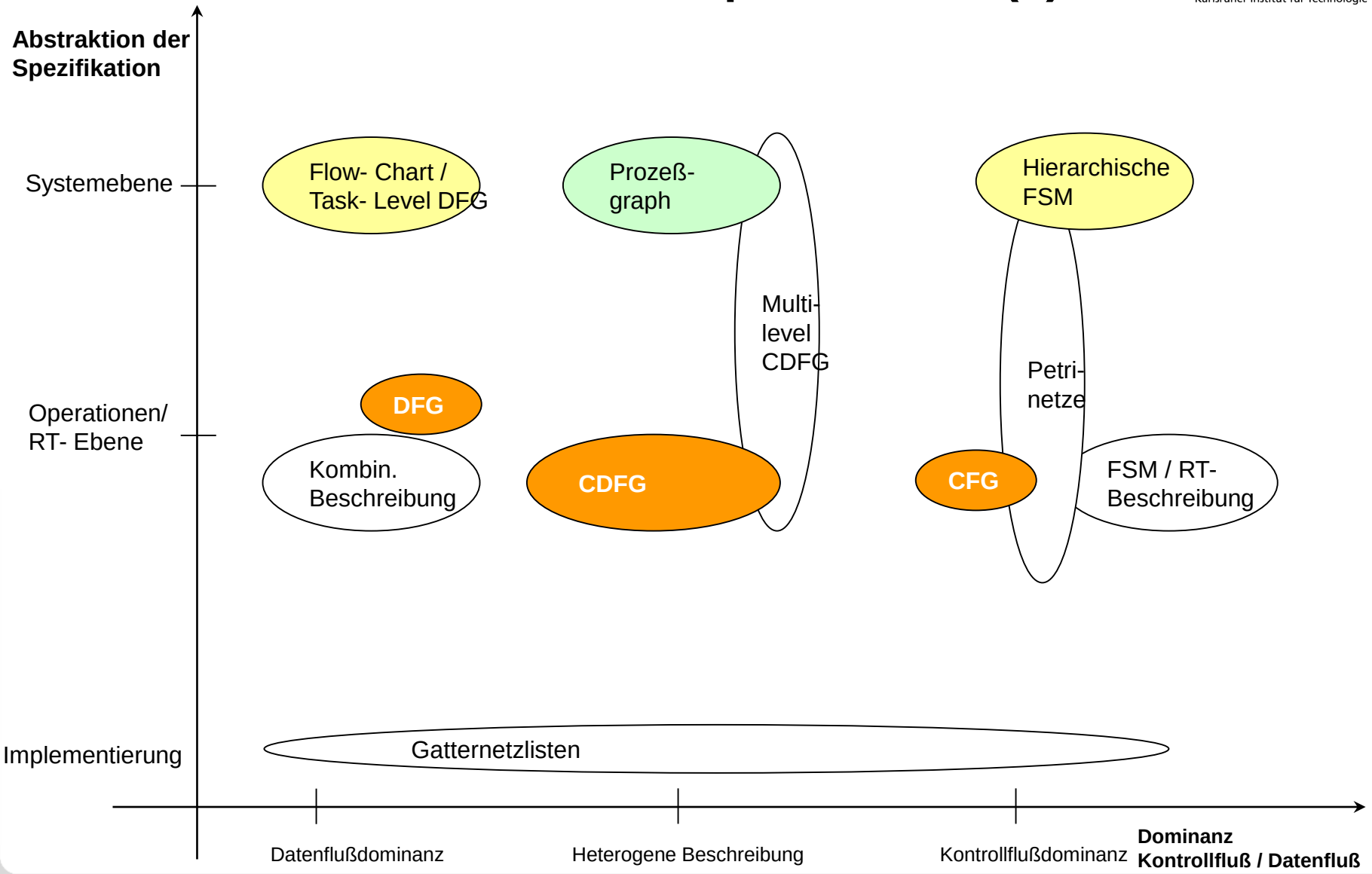
4.1 Einführung (II)

- Die Gesamtaufgabe der Hardware/Software Partitionierung kann in folgende Teilprobleme untergliedert werden:
  - Festlegung der Abstraktionsebene der Spezifikation.
  - Wahl der Granularität (Trade-off Komplexität vs. Genauigkeit).
  - Allokation von Systemkomponenten.
  - Auswahl realistischer/möglicher Metriken und Schätzungen.
  - Entwurf einer geeigneten Zielfunktion.
  - Auswahl geeigneter Partitionierungsalgorithmen.
  - Entwurfsablauf und Designinteraktion.

4.1 Abstraktionsebenen der Spezifikation (I)

- Systemebene
  - Hierarchische Taskgraphen:
    - datenflußdominierte Flowcharts
    - kontrollflußdominierte Statecharts
  - Graphen mit Kontrollfluß- und Datenabhängigkeiten zwischen den Prozeßknoten -> Sequenzgraphen (mit Hierarchie!)
- Operationsebene
  - Pendants zur Systemebene:
    - Kontrollflußgraphen (CFG)
    - Datenflußgraphen (DFG)
    - Kontroll-/ Datenflußgraphen (CDFG)
  - Petri- Netze, geeignet für kontrollflußdominierte Anwendungen..
- Netzlisten auf niedriger struktureller Ebene
  - Enthalten ggf. auch Blöcke mit komplexen Verhaltensbeschreibungen:
    - IP-Blöcke
    - Analog-Blöcke

4.1 Abstraktionsebenen der Spezifikation (II)



4.1 Komplexitätsbetrachtungen

- Partitionierungsproblem ist das Zuordnen von n Objekten O = {o_1, ..., o_n} zu m Blöcken (Partitionen) P = {p_1, ..., p_m}, so daß
  - $p_1 \cup p_2 \cup ... \cup p_m = O$; $p_i \cap p_j = \{\}$ $\forall$ i, j: $i \neq j$ und die Kosten c(P) minimal sind
- Das Partitionierungsproblem eines Graphen ist i.d.R. NP-vollständig
- Beispiel: Ein Graph mit n Knoten ist in k disjunkte Partitionen gleicher Größe p mit k•p=n zu zerlegen.
  - Für die Wahl der ersten Partition hat man $\binom{n}{p}$ Möglichkeiten, für die zweite sind es nur noch $\binom{n-p}{p}$ Kombinationen

    entfernt Doubletten, da alle Partitionen gleichwertig sind.

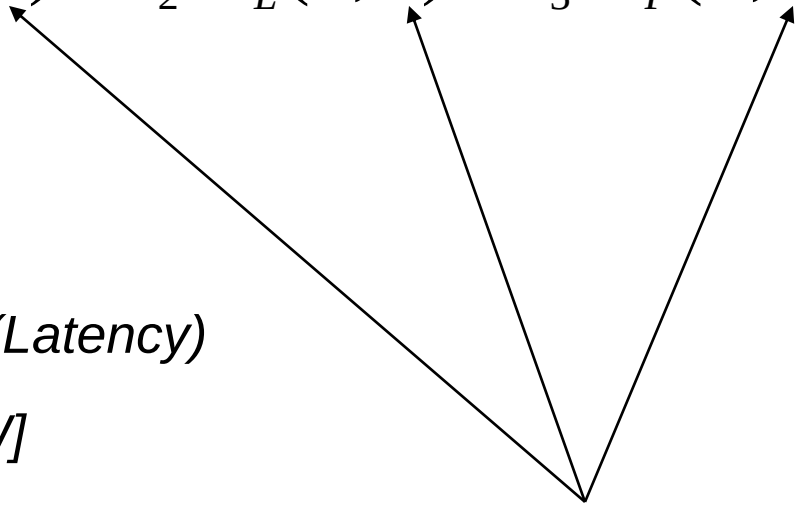
  - Insgesamt erhält man $\frac{1}{k!}\binom{n}{p}\cdot\binom{n-p}{p}\cdots\binom{2p}{p}\cdot\binom{p}{p} = \frac{1}{k!}\cdot\frac{n!}{(p!)^k}$ Kombinationen.
  - Wie man an der folgenden Tabelle sieht, steigt der Aufwand exponentiell:

n	K=2	K=5	K=10
10	126	945	1
20	92378	$2{,}55 \cdot 10^9$	$6{,}54 \cdot 10^8$
50	$6{,}32 \cdot 10^{13}$	$4{,}03 \cdot 10^{29}$	$1{,}35 \cdot 10^{37}$
100	$5{,}04 \cdot 10^{28}$	$9{,}12 \cdot 10^{63}$	$6{,}45 \cdot 10^{85}$

4.1 Kostenfunktionen zur Hardware/Software-Partitionierung

- **Beispiel für eine Kostenfunktion:**

$$f(C,L,P) = k_1 \cdot h_c(C,\overline{C}) + k_2 \cdot h_L(L,\overline{L}) + k_3 \cdot h_P(P,\overline{P})$$

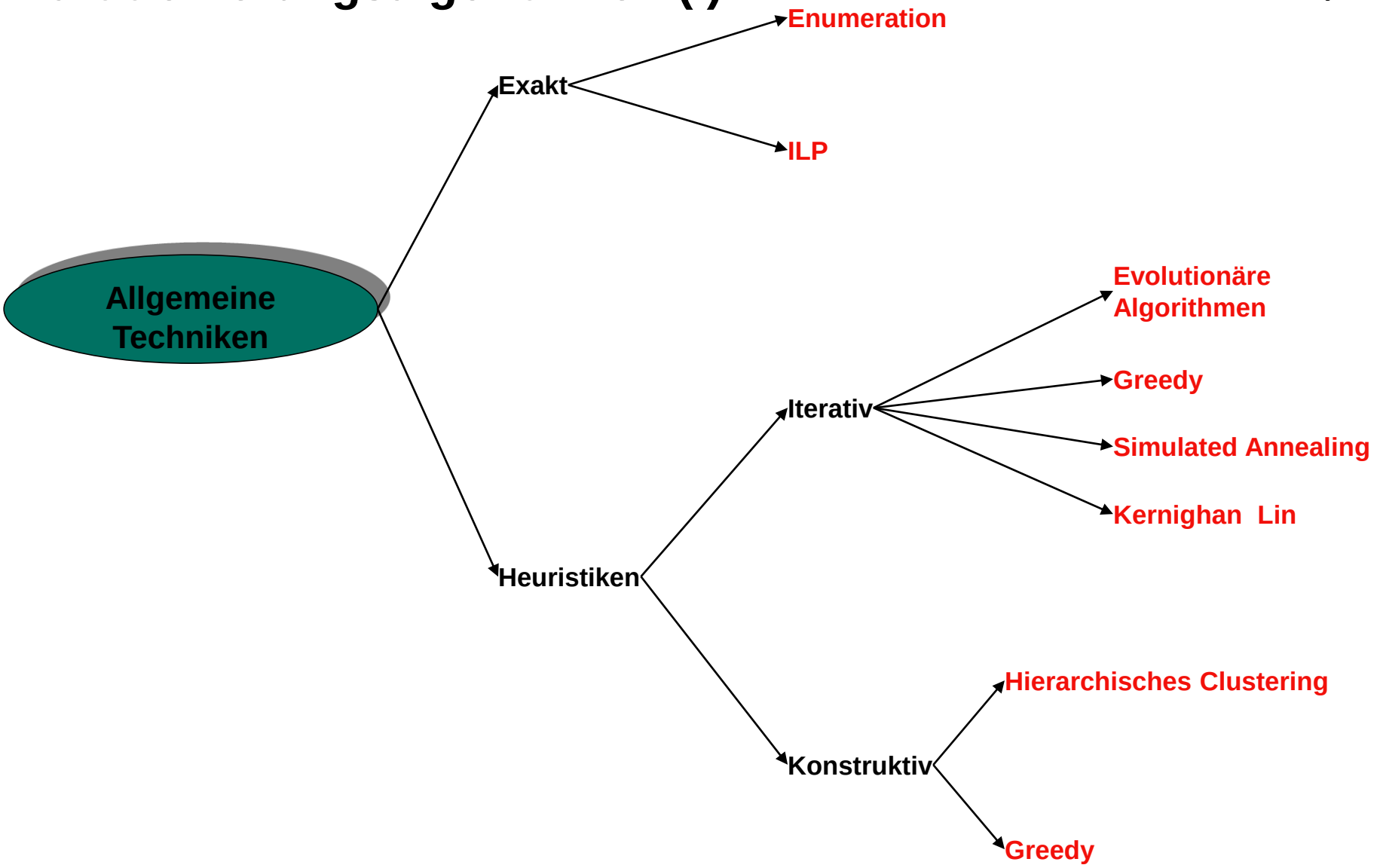


- *C ≙ Systemkosten in [Euro]*
- *L ≙ Ausführungszeit in [sec] (Latency)*
- *P ≙ Leistungsaufnahme in [W]*
- *h_C, h_L, h_P, geben an, wie stark C, L, P die Entwurfsbedingungen (Constraints C, L, P überstrichen) verletzen.*
- *k_1, k_2, k_3 ≙ Gewichtung und Normierung*

- Warum ist eine Hardware/Software Partitionierung notwendig?
- Welche Randbedingungen bzw. Optimierungskriterien können für Trade-offs herangezogen werden?
- Warum können welche Abstraktionsebenen gewählt werden?
- Welche Ansätze können für die Partitionierung angewendet werden?

Inhalt

4.2 Klassifikation von Partitionierungsalgorithmen (I)



4.2 Klassifikation von Partitionierungsalgorithmen (II)

- **Exakte Lösungsverfahren** (können aufwendig werden):
  - finden die global beste Lösung
  - Beispiele:
    - Enumeration der Lösungen
    - Integer Lineare Programmierung (ILP)

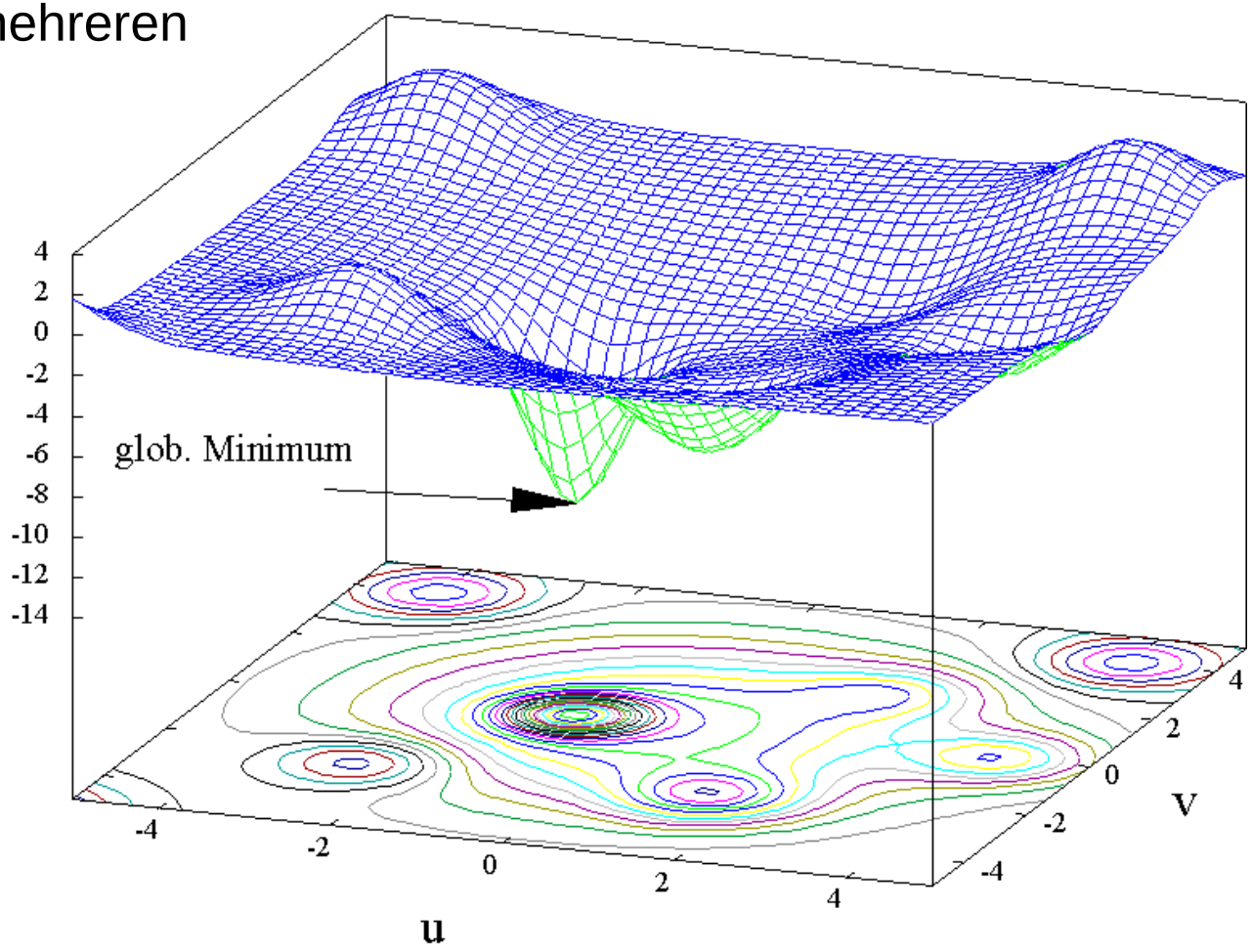
- **Heuristische Lösungsverfahren:**
  - **Findet hinreichend gute Lösung auf Basis unvollständiger Information**
  - **Klassen:**
    - **Konstruktive Verfahren**:
      - Random Mapping
      - Hierarchical Clustering
    - **Iterative Verfahren:**
      - Kernighan Lin Algorithmus (robust und erprobt)
      - Simulated Annealing "
      - effiziente globale Entwurfsraumexploration (genet. Alg.)

4.2 Klassifikation von Partitionierungsalgorithmen (III)

- Wesentliche Vorgehensweisen bei Heuristiken:
  - **Konstruktive Algorithmen:**
    - Konstruktive Verfahren gruppieren funktionale Objekte in eine geringe Anzahl von Partitionen.
    - Die Objekte werden dabei in Cluster anhand einer zwischen den Objekten definierten Nähefunktion (-> *Closeness*) zusammengefaßt.
  - **Iterative Algorithmen:**
    - Iterative Verfahren gehen von einer Anfangspartitionierung aus und verbessern diese solange, bis eine Güte-/Kostenfunktion konvergiert oder ein Abbruchkriterium erfüllt ist.
    - Jede Partitionierung wird mit der Güte-/ Kostenfunktion bewertet:
      - Die Ergebnisse sind i.a. besser als bei konstruktiven Verfahren.
      - Können auch aus lokalen Minima einer Kostenfunktion wieder herauskommen (je nach Optimierungstechnik der Heuristik): Hill-Climbing Eigenschaft
        -> ggf. werden temporäre Verschlechterungen der Kostenfunktionswerte akzeptiert
  - **Heterogene Verfahren:**
    - Sind eine Kombination von konstruktiven und iterativen Verfahren.

4.2 Klassifikation von Partitionierungsalgorithmen (IV)

- Beispiel für eine Kostenfunktion *K(u,v)* mit mehreren lokalen Minima



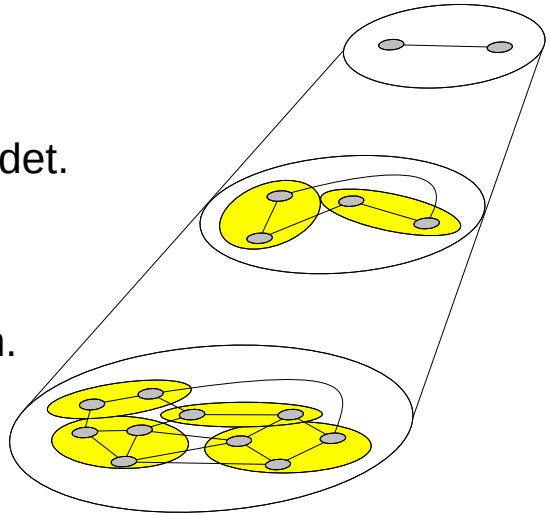
Oliver Sander
Hardware/Software Co-Design

- Wie können Partitionierungsalgorithmen klassifiziert werden?
- Welche Lösungsverfahren gibt es?
- Warum sind iterative Verfahren ‘besser’ als konstruktive Verfahren?
- Wie kann eine Startlösung aussehen?
- Wird immer eine bessere Lösung gefunden?

Inhalt

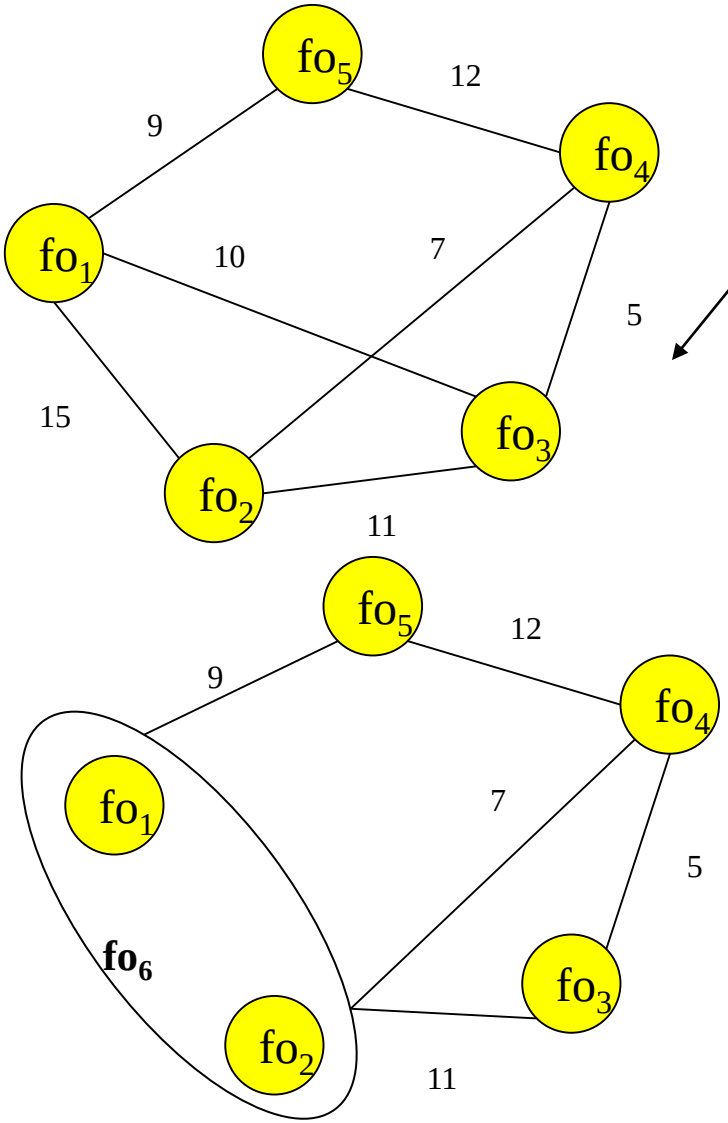
4.3 Konstruktive Verfahren

- **Auswahlverfahren:**
  - *Random Mapping*:
    - Jedes Objekt wird zufällig auf einen Block abgebildet.
  - *Hierarchical Clustering*:
    - Schrittweises Zusammengruppieren von Objekten.
    - ***Nähefunktion*** gibt an, wie gewinnbringend die Gruppierung zweier Objekte ist.

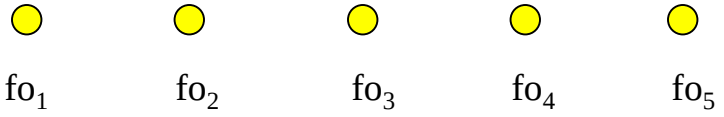


- **Konstruktive Verfahren:**
  - Werden oft verwendet, um eine Anfangspartition für iterative Verfahren zu erzeugen.
  - Haben das Problem, daß es sehr schwierig sein kann, eine geeignete *Nähefunktion* zu definieren.

4.3.1 Hierarchisches Clustering: Beispiel (I)

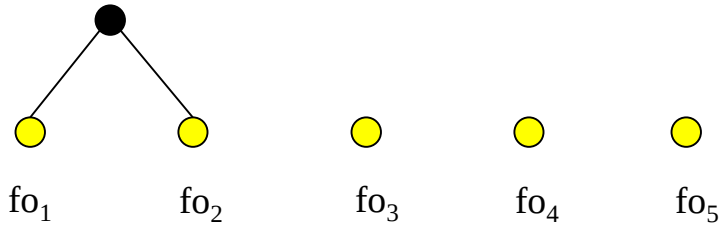


Betrachte jeden Knoten anfänglich als „trivialen“ Cluster.

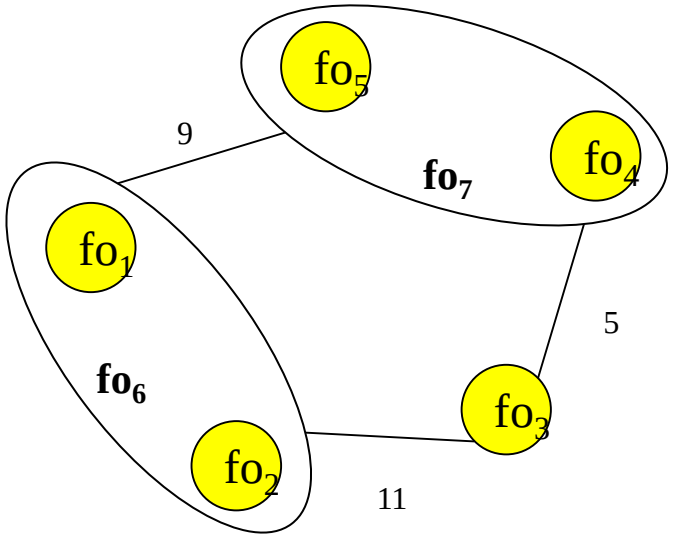


- Verschmelze Knoten f_{o1} und f_{o2}, Kante hat größten Wert 15
- Lösche Kante (f_{o1}, f_{o3}), denn Kante (f_{o2}, f_{o3}) führt auch in den gleichen Cluster und hat einen größeren Wert 11

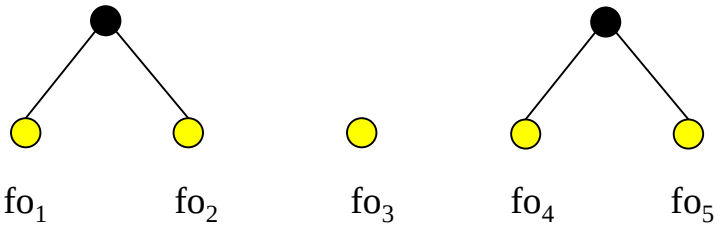
  (Variante besteht darin, alle Kanten, die vom gleichen Knoten in den Cluster führen, durch einzelne Kante zu ersetzen und diese zu mitteln)



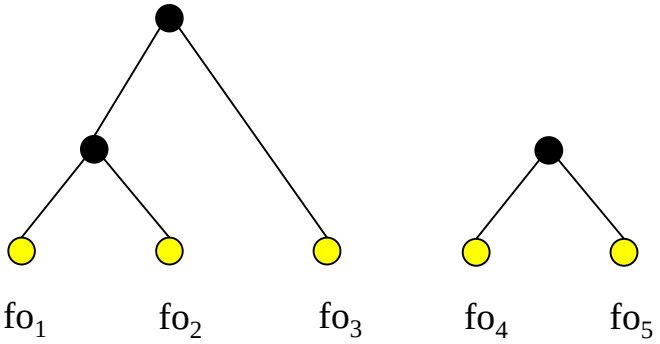
4.3.1 Hierarchisches Clustering: Beispiel (II)

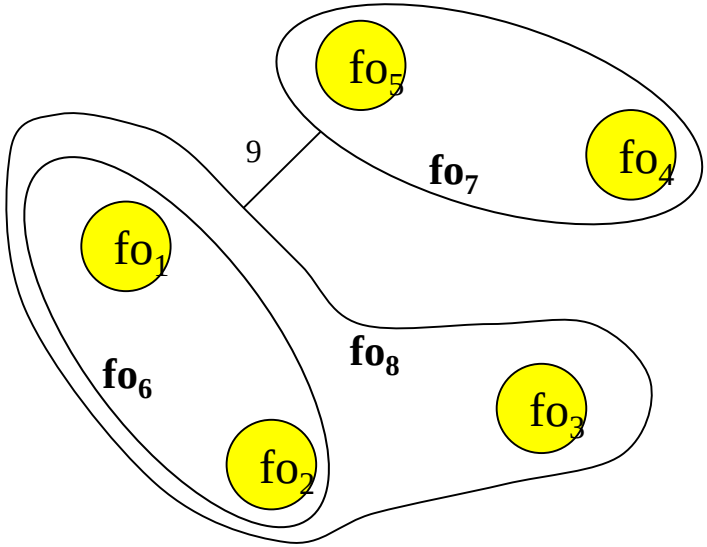


- Verschmelze Knoten f_{o5} und f_{o4}, Kante hat größten Wert 12. Lösche Kante ($[f_{o1}, f_{o2}]$, f_{o4})

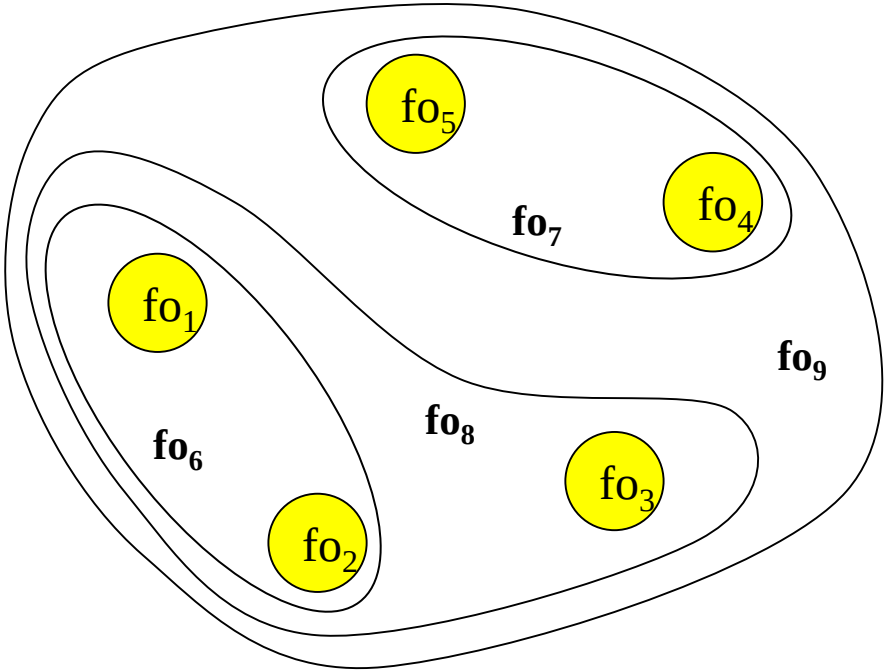


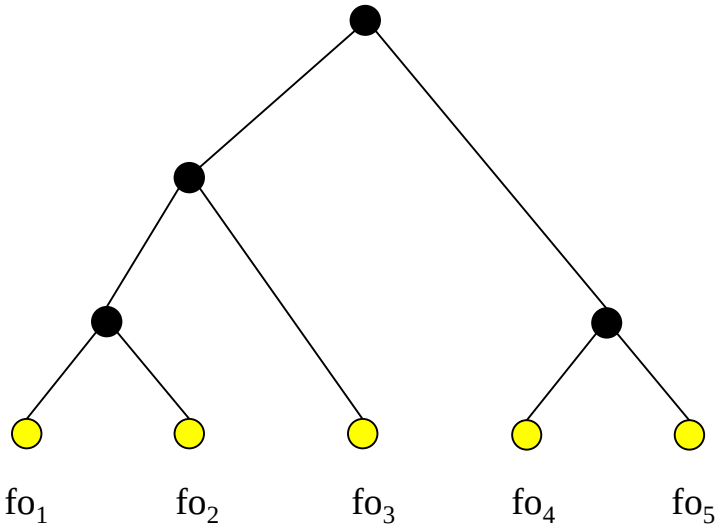
- Verschmelze Superknoten $[f_{o1}, f_{o2}]$ mit f_{o3}, Kante hat größten Wert 11
- Lösche Kante ($[f_{o4}, f_{o5}]$, f_{o3})





4.3.1 Hierarchisches Clustering: Beispiel (III)





- Ende der Partitionierung durch hierarchisches Clustering.
- Die Hierarchie des erzeugten Baums spiegelt die Clusteringschritte der Partitionierung wieder.

4.3.1 Hierarchisches Clustering: Algorithmus

```
PROCEDURE HIERARCHICAL CLUSTERING(O) {
    P := { };
    FOR i = 1 TO n DO
        p_i := {o_i };
        P := P ∪ p_i;
    ENDFOR
    FOR i = 1 TO n DO
        FOR j = 1 TO n DO
            ComputeCloseness(p_i, p_j);
        ENDFOR
    ENDFOR
    numblocks = n;
    k := n + 1
    WHILE (Terminate(P) == FALSE)
        p_i, p_j := FindClosestObjects(P);
        p_k := {p_i, p_j};
        P := P \ p_i \ p_j ∪ p_k;
        numblocks := numblocks -1;
        FOREACH Block p_l ∈ P \ p_k DO
            ComputeCloseness(p_l, p_k);
        ENDFOR
        k := k + 1;
    ENDWHILE
    RETURN(P);
} END PROCEDURE
```

O = Menge der Objekte;
P = noch leere Menge der Partitionen p_i

Initialisiere jedes Objekt als eine Gruppe.

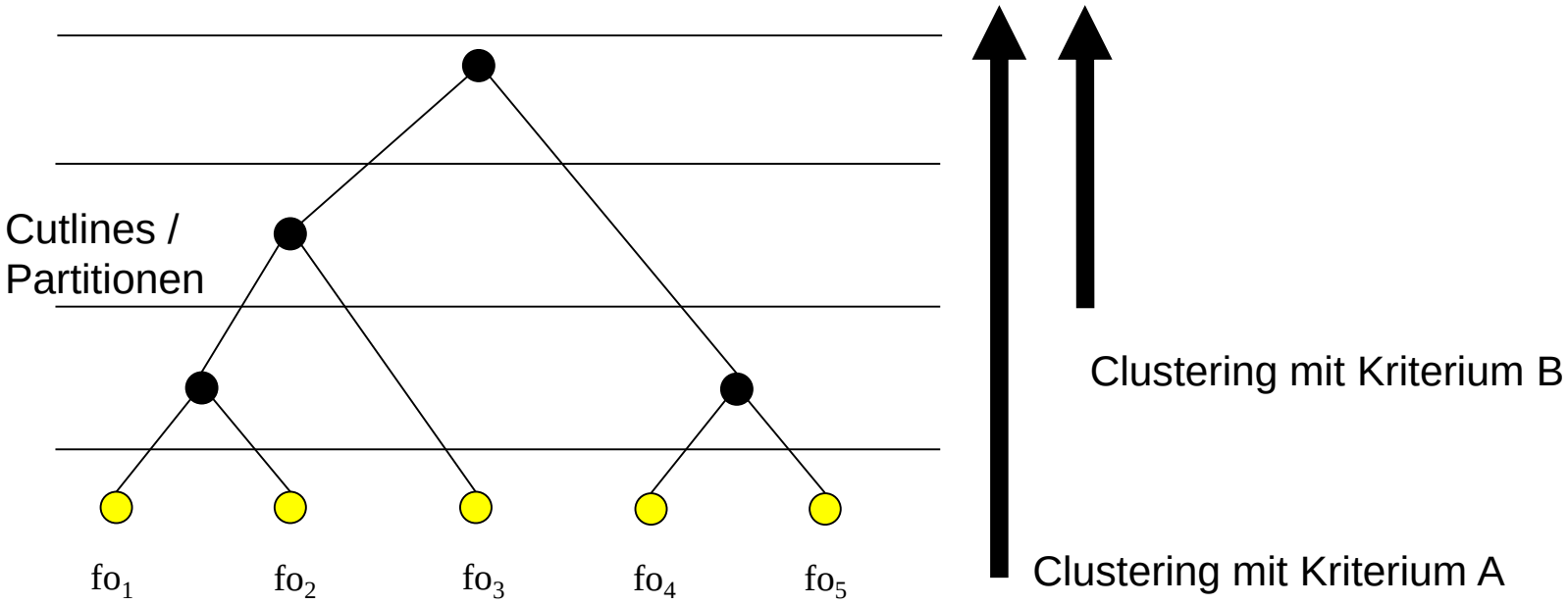
Berechne *Closeness-Function* zwischen den Objekten.

Gruppiere die beiden Objekte, und bereinige / aktualisiere die Partitionsmenge.

Closeness-Update der anderen Partitionen.
Berücksichtigt auch Mehrfachkanten von $p_i \rightarrow p_k$

4.3.1 Hierarchisches Clustering

- **Variante:** Multistage Clustering
  - *Clustering* Vorgang in mehrere aufeinander folgende Phasen aufgeteilt.
  - In jeder Phase wird andere Nähefunktion / Kriterium verwendet, wobei jeweils der in der letzten Clustering-Phase erzeugte Graph benutzt wird.
  - Schneiden problematisch, wenn Baum nicht ausbalanciert.
  - Zu starke Ausbalancierung ggf. auch nachteilig, wenn stark zusammenhängende Knoten unterschiedlichen Clustern zugeordnet werden.



4.3.1 Hierarchisches Clustering: Clustering-Metriken

- **Datenabhängigkeiten:**
  - Ziel ist die Reduktion des Kommunikationsaufkommens zwischen Knoten durch Zusammenlegen dieser Knoten in einer Partition.

- ***Sharing* von Operatoren:**
  - Komplexe Operatoren mit aufwendiger HW-Implementierung werden in einer Partition untergebracht. Erlaubt Mehrfachnutzung der HW, vorausgesetzt ein Schedule für zeitkritische Teile der Spezifikation wird nicht behindert.

- **Kontrollfluß:**
  - Ziel: Reduzierung der Übergabe von Kontrollflüssen zwischen Partitionen, wenn Operationen aus einem Kontrollflußzweig zusammengehalten werden:
    - Operationen liegen in disjunkten Kontrollflußzweigen $\Rightarrow$ kein Aufwand bei Übergabe
    - Operationen liegen im gleichen Kontrollflußzweig ohne dazwischenliegende Verzweigungen.
      $\Rightarrow$ Zuweisung zum gleichen Cluster anstreben
    - Kontrollflußverzweigung zwischen zwei Operationen
      $\Rightarrow$ Verzweigungswahrscheinlichkeit als Gewichtung für Nähefunktion

4.3.1 Hierarchisches Clustering: Eigenschaften

- Bedingte Eignung der *Clustering* Verfahren für die Partitionierung eines Systems.
- Ist bezüglich der Gesamtbewertung nicht in der Lage, ein lokales Extremum (bspw. Minimum) zu überwinden.
- Gute Anwendbarkeit für große Knotenzahlen.
- Anwendung zur Erzeugung einer initialen Partitionierung, auf die dann andere Verfahren aufsetzen können.
- Komplexität eines Partitionierungsproblems kann exponentiell mit der Knotenanzahl wachsen, weshalb durch Vorschaltung eines Clusterings die Komplexität reduziert werden kann.

- Wie geht der Algorithmus vor?
- Was ist die Closeness-Funktion?
- Wie muss der Baum geschnitten werden, um bestimmte Partitionierungen zu erreichen?
- Welche Komplexität hat der Algorithmus?

Inhalt

- 4.1 Einführung, Partitionierungsansätze, Komplexität
- 4.2 Klassifikation von Partitionierungsalgorithmen
- 4.3 Konstruktive Algorithmen
  - 4.3.1 Hierarchisches Clustering
- 4.4 Iterative Algorithmen
  - 4.4.1 **Tabu-Search**
  - 4.4.2 Kernighan Lin
  - 4.4.3 Fiduccia Mattheyses
  - 4.4.4 Integer Linear Programming - ILP
  - 4.4.5 Simulated Annealing
  - 4.4.6 Genetische Algorithmen
- 4.5 Hardware/Software Partitionierungsverfahren und Co-Design-Systeme

4.4.1 Tabu-Search Algorithmus (I)

- **Flexible Suchheuristik** für schnelles und nahezu optimales Lösen von Optimierungsaufgaben -> sehr gutes Verfahren für kombinatorische Optimierung.

- Verfahren bestimmt ausgehend von einer Startlösung die beste Nachbarlösung, ansonsten diejenige Nachbarlösung, die das Ergebnis am wenigsten verschlechtert.

- Zur **Vermeidung von Zyklen wird immer der beste Suchschritt ausgeführt, der nicht zu einer bereits betrachteten Lösung** führt.

4.4.1 Tabu-Search Algorithmus (II)

- Dies erfordert die **temporäre Speicherung und Sperrung** der letzten n bereits gefundenen Lösungen/Suchschritte (*Tabu-Liste*):
  - **Überwindung lokaler Extrema** der Kostenfunktion möglich.
  - Verhindert vorübergehend die Umkehrung eines Suchschrittes.
  - Tabu- Liste als FIFO mit Länge n ausgelegt. ($n \approx |V| / 10$).
  - Bei **Akzeptanz** einer neuen Lösung wird der **älteste Eintrag aus der Liste** entfernt.
  - Akzeptanz einer Lösung, wenn sie keinen Eintrag (Prädikat) in der Tabu- Liste verletzt oder wenn sie ein globales Minimum im Vergleich zu den bisher untersuchten Lösungen liefert.
  - **Länge der Liste** hat **Einfluß auf die Wirksamkeit** des Verfahrens. Ist n zu klein, so können Zyklen entstehen, anderenfalls werden ggf. nach einer bestimmten Zeit keine Nachbarlösungen gefunden.
- Verfahren durch Vorgehensweise **deterministisch**.

4.4.1 Tabu-Search Algorithmus (III)

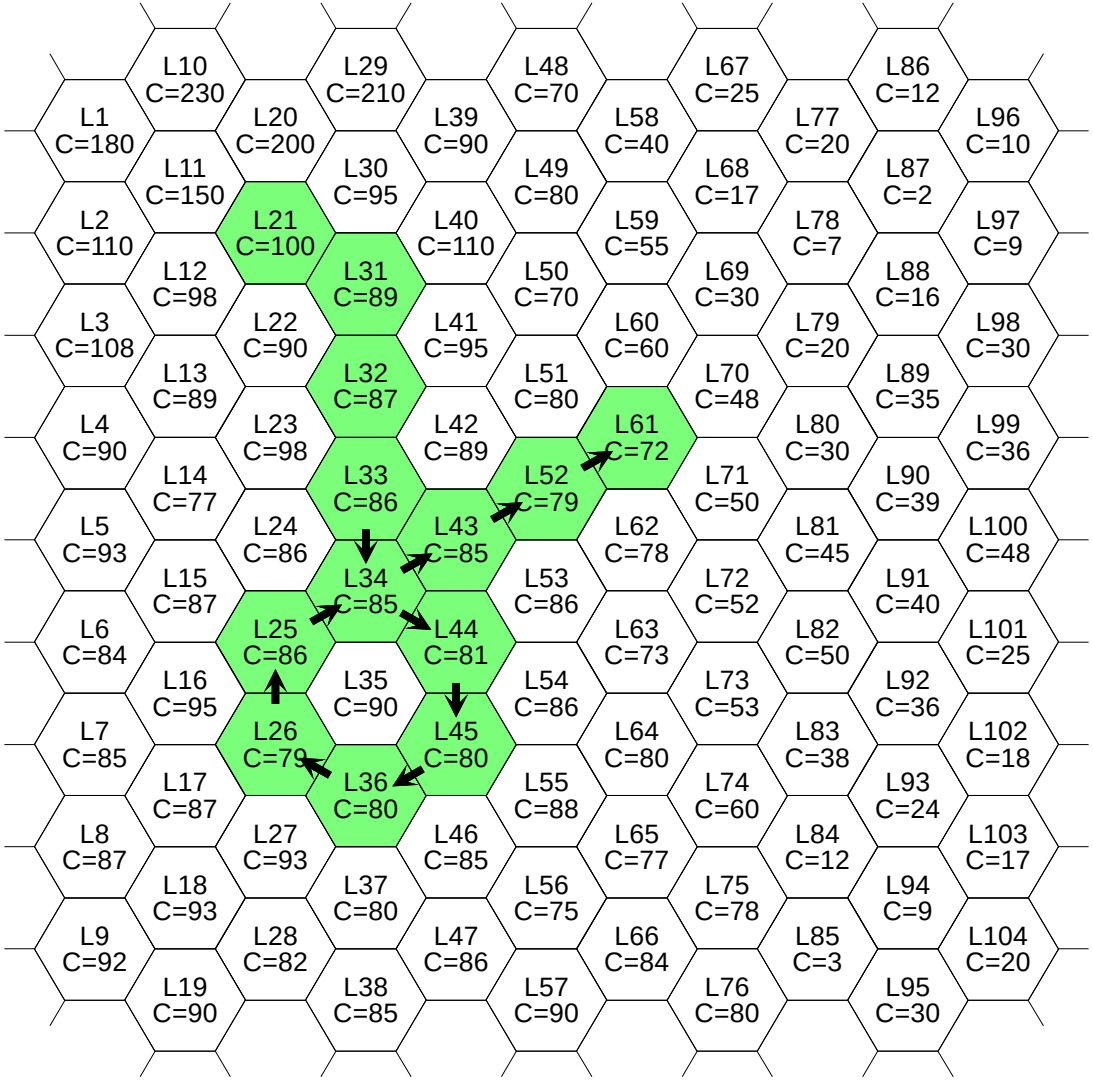
```
CurrentSolution:=InitialSolution
BestSolution:= CurrentSolution
TabuList:={ }
repeat
        N:=k_BestNeighbours(CurrentSolution)
        NewSolution:=none
        for i:=1 to k do

                X:= FirstElement(N)
                if cost(X)<cost(BestSolution) ∨ (∀ p ∈ TabuList: ¬ p(CurrentSolution, X) ) then
                        NewSolution:=Select_MinCost(NewSolution, X)
                endif
                N:= N \ {X}

        endfor

        TabuList := Take_N_First(n, TabuList ∪ Tabu(CurrentSolution, NewSolution))
        CurrentSolution := NewSolution
        BestSolution:=Select_MinCost(BestSolution, CurrentSolution )

until stopcondition
```

Initialisierung mit einer Vorgabelösung

Akzeptiere Nachbarlösung, wenn diese besser als die beste Lösung ist oder Lösungsschritt nicht in der Tabuliste gespeichert ist.
Erlaubt auch Akzeptanz von Verschlechterungen.

Aktualisiere Tabu- Liste, Beste Lösung

4.4.1 Tabu-Search Algorithmus: Beispiel (I)



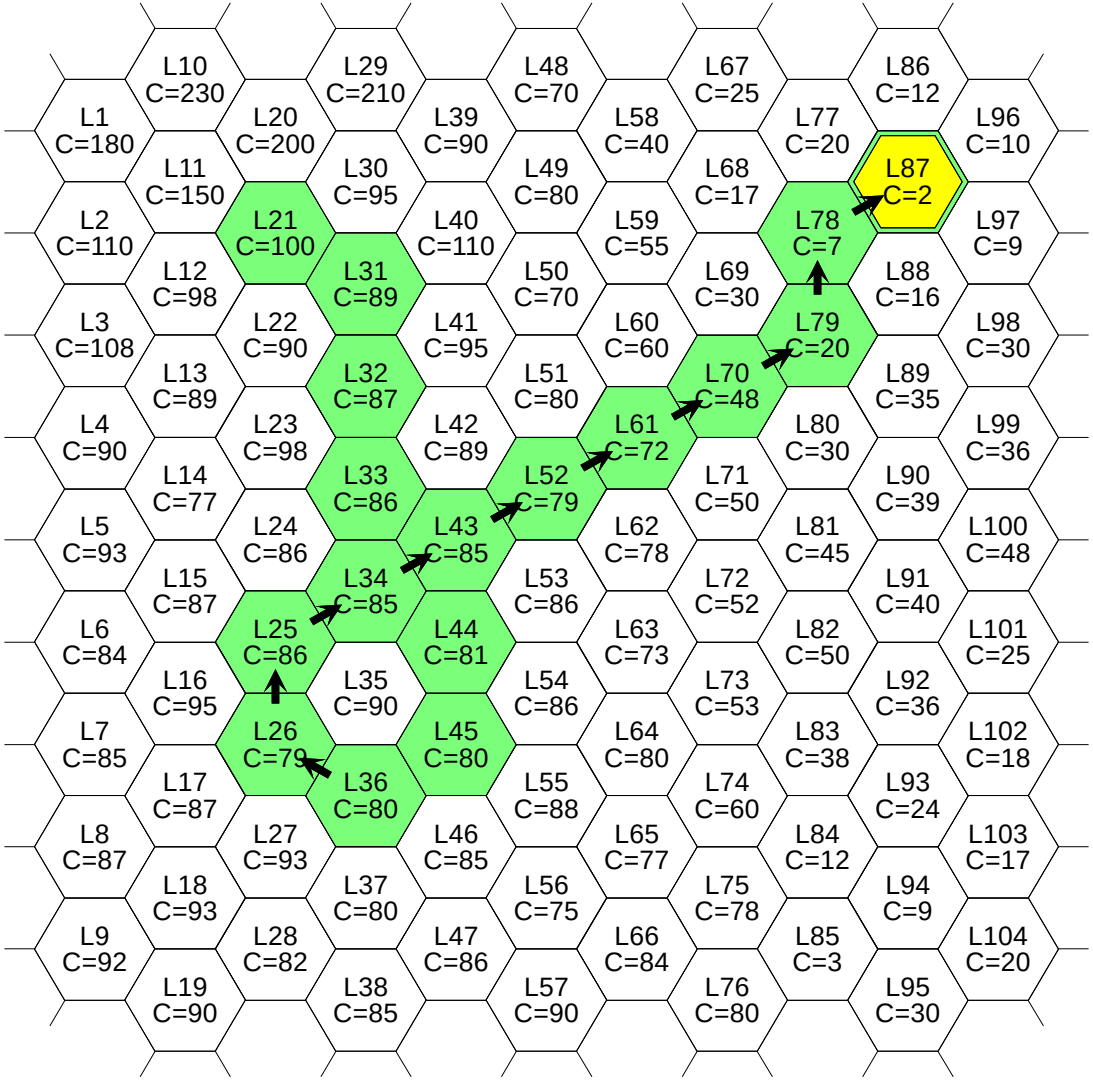
n_{Tabu_Liste}=10 k=6

Stop_Cnd: BS <5

BS:	CS:	Step:
100	100	**Start:** L21
89	89	L21 ⇒L31
87	87	L31 ⇒L32
86	86	L32 ⇒L33
85	85	L33 ⇒L34
81	81	L34 ⇒L44
80	80	L44 ⇒L45
80	80	L45 ⇒L36
79	79	L36 ⇒L26
79	86	L26 ⇒L25
79	85	L25 ⇒L34
79	85	L34 ⇒L43
79	79	L43 ⇒L52
72	72	L52 ⇒L61

Oliver Sander
Hardware/Software Co-Design

4.4.1 Tabu-Search Algorithmus: Beispiel (II)



n_{Tabu_Liste}=10 k=6

Stop_Cnd: BS <5

BS:	CS:	Step:
72	72	L52 ⇒L61
48	48	L61 ⇒L70
20	20	L70 ⇒L79
7	7	L79 ⇒L78
2	2	L78 ⇒L87

Oliver Sander
Hardware/Software Co-Design

4.4.1 Tabu-Search Algorithmus: Beispiel (III)

- Anmerkungen zur Partitionierung mit Tabu-Search:
  - Das auf den vorhergehenden Folien dargestellte Beispiel ist als allgemeines abstrahiertes Beispiel zur Darstellung von Tabu-Search.
  - Beim Partitionierungsproblem im Besonderen ist an dieser Stelle zu beachten, dass ein Lösungsschritt mit einer Variation der die Lösung beschreibenden Parameter einhergeht.
    - Hier: das Verschieben eines Knotens in eine andere Partition.
    - Folge: Änderung der Partitionierung als Lösung.
  - Das Verbieten eines Lösungsschritts beinhaltet damit auch das Verbot, die Variation dieser Parameter rückgängig zu machen.
  - Im Falle der Partitionierung kann es also nicht sein, dass eine Lösung (d.h. Partitionierung) kurz- oder mittelfristig nochmals besucht wird, denn dann hätten folglich die bisherigen Parameter-variationen zurückgenommen werden müssen, was Tabu-Search innerhalb des Gültigkeitsbereichs der Tabu-Liste aber so verbietet.

- Für was kann TS gut eingesetzt werden?
- Wie geht der Algorithmus vor?
- Was ist die Tabu-List?
- Wie wird diese upgedated?
- Welche Komplexität hat der Algorithmus?