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

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.

4.1 Einführung (II)

- 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 (III)

- 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)

- Die Spezifikationsmethode ist im wesentlichen von den Anforderungen der Anwendung abhängig:

- Systemebene:
  - Hierarchische Taskgraphen:
    - datenflußdominierte Flowcharts
    - kontrollflußdominierte Statecharts

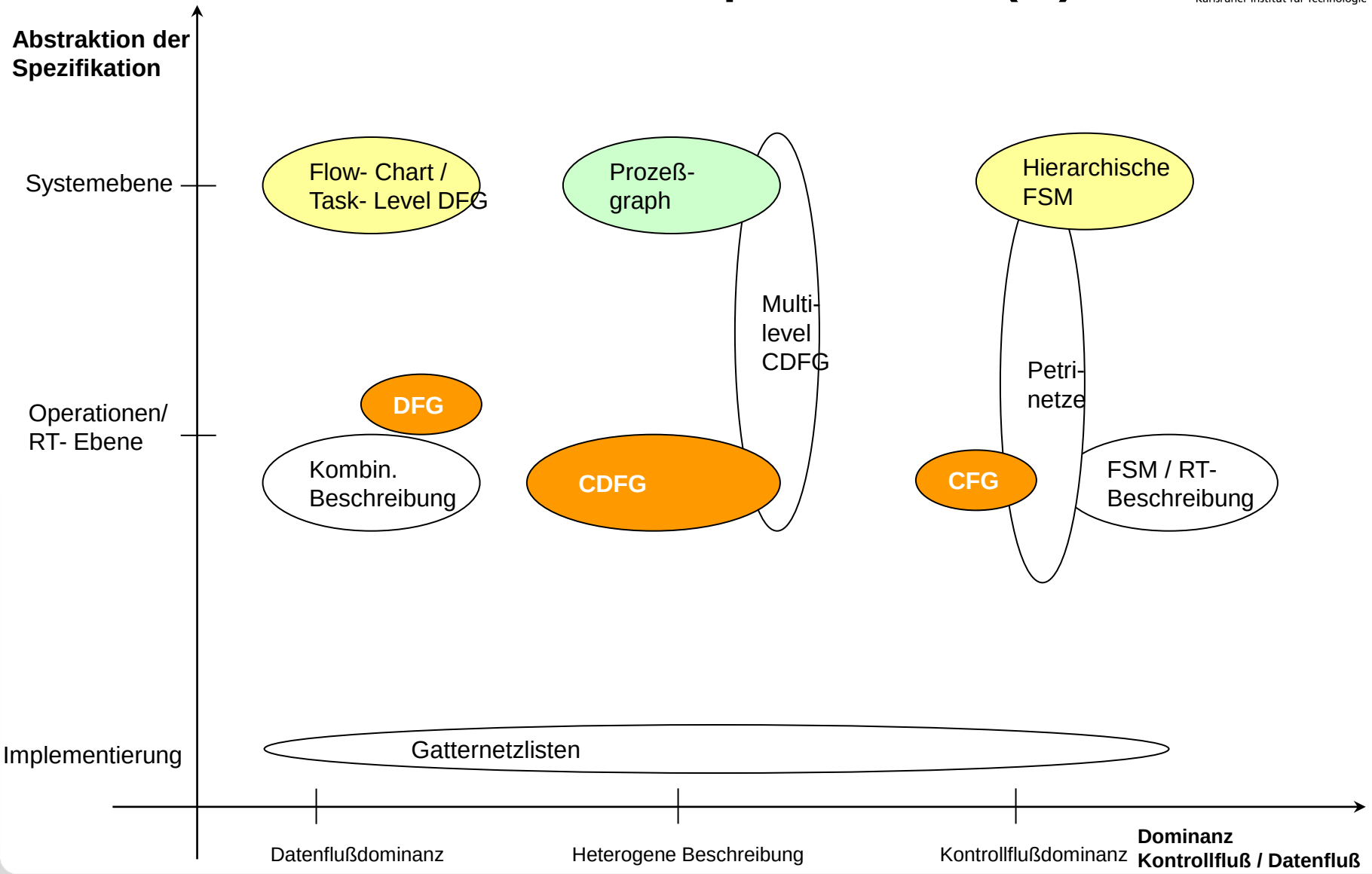
  - Graphen mit Kontrollfluß- und Datenabhängigkeiten zwischen den Prozeßknoten -> Sequenzgraphen (mit Hierarchie!)

4.1 Abstraktionsebenen der Spezifikation (II)

- Operationsebene :
  - Pendants zur Systemebene:
    - Kontrollflußgraphen (CFG)
    - Datenflußgraphen (DFG)
    - Kontroll-/ Datenflußgraphen (CDFG)
  - Petri- Netze, geeignet für kontrollflußdominierte Anwendungen.
  - Spezifikationssprachen für FSM-, Datenpfadbeschreibung, Partitionierung durch Abbildung auf CDFGs und Sequenzgraphen.

- Netzlisten auf niedriger struktureller Ebene:
  - Enthalten ggf. auch Blöcke mit komplexen Verhaltensbeschreibungen:
    - IP-Blöcke
    - Analog-Blöcke

4.1 Abstraktionsebenen der Spezifikation (III)



4.1 Partitionierungsansätze (I)

- **Partitionierung auf verschiedenen Abstraktionsebenen:**
  - Festlegung der Abstraktionsebene, auf der die Zerlegung stattfindet.
  - Spezifikation legt dazu den Ansatzpunkt fest.
  - Vor der Partitionierung sind Transformationen auf eine niedrigere Abstraktionsebene durch Syntheseschritte möglich:
    - Umkehrung der Richtung ist i.a. nach der Partitionierung nicht möglich.

- **Partitionierung abstrakter anwendungsorientierter Spezifikationen:**
  - Partitionierung durch Zerlegung eines Spezifikationsgraphen.
  - Für **kontrollflußdominierte** Anwendungen erfolgt dies in Form einer Dekomposition von Zustandsautomaten -> mehrere FSMs.
  - Für **datenflußdominierte** Anwendungen als Zerlegung von Datenflußgraphen.
  - Die Abschätzung der Entwurfsqualität ist für eine effektive Partitionierung entscheidend.

4.1 Partitionierungsansätze (II)

- **Funktionale Partitionierung verhaltensorientierter Spezifikationen**
  - **Spezifikation als parallele ausführbare Prozesse in Sprachnotation:**
    - VHDL mit Hardwareschwerpunkt.
    - C / C++ mit Softwareschwerpunkt.
  - **Partitionierung auf funktionaler Ebene durch Zerlegung der Prozesse in Untermenge von Prozessen und Abbildung auf HW/SW-Partitionen:**
    - Ausnutzung der Parallelisierung in der HW-Implementierung.
    - Codemigration HDL $\Leftrightarrow$ C unter Berücksichtigung von spezifikationsbedingten Einschränkungen möglich (migrationsfähige Sprachkonstrukte).
    - Vorteil: verringerte Problemkomplexität im Vergleich mit Partitionierung auf niedrigerer struktureller Ebene.
    - Sinnvolle Abstraktionsebene für HW/SW-Partitionierung wegen Codemigration und Schätzbarkeit.
    - Vergleich der Entwurfsalternativen
      -> erlaubt Exploration des Entwurfsraums

4.1 Partitionierungsansätze (III)

- **Implementierungsnahe Spezifikationen für HW/HW-Partitionierungen:**
  - Strukturelle Beschreibung der Hardware in Form von Netzlisten mit nebenläufigen Operationen -> wenige Syntheseschritte nötig (Gatteroptimierung, Mapping, Retiming).
  - Partitionierung notwendig, wenn Netzliste nicht vollständig auf einer HW-Komponente realisierbar:
    -> Anwendung bei Rapid-Prototyping von ASICs auf FPGA-Strukturen.
  - Einsatz von Bi- bzw. K-Wege-Partitionierungsverfahren.
  - Einfache Abschätzung des Flächenbedarfs durch Summation der Partitionsflächen, da alle Komponenten (Register, MUX, Multiplizierer, Bitbreiten, etc.) bekannt sind.
  - Große Objektzahl nachteilig -> einfache Heuristiken notwendig.
  - Hierarchische Strukturen von Algorithmen nicht mehr erkennbar:
    -> Nur suboptimale Lösungen bei Verwendung von lokal arbeitenden Heuristiken.

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.

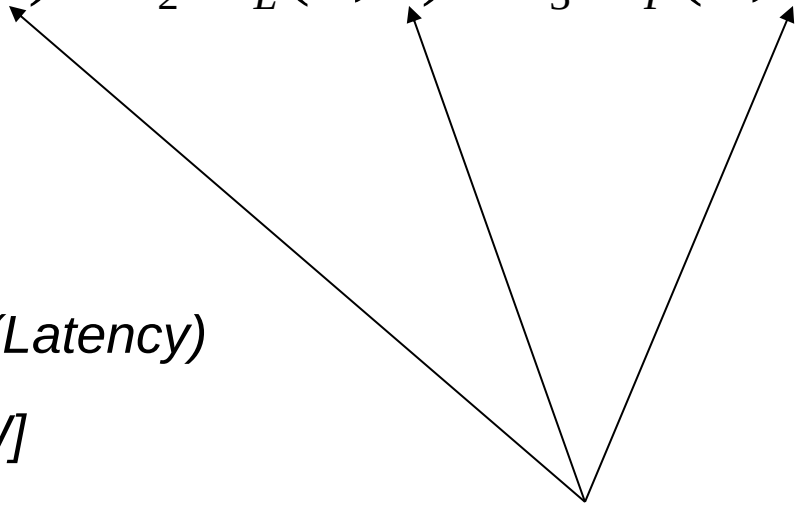
  - Insgesamt erhält man $\frac{1}{k!}\binom{n}{p}\cdot\binom{n-p}{p}\cdots\binom{2p}{p}\cdot\binom{p}{p} = \frac{1}{k!}\cdot\frac{n!}{(p!)^k}$ Kombinationen.
  - Wie man an der folgenden Tabelle sieht, steigt der Aufwand exponentiell:

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:**

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



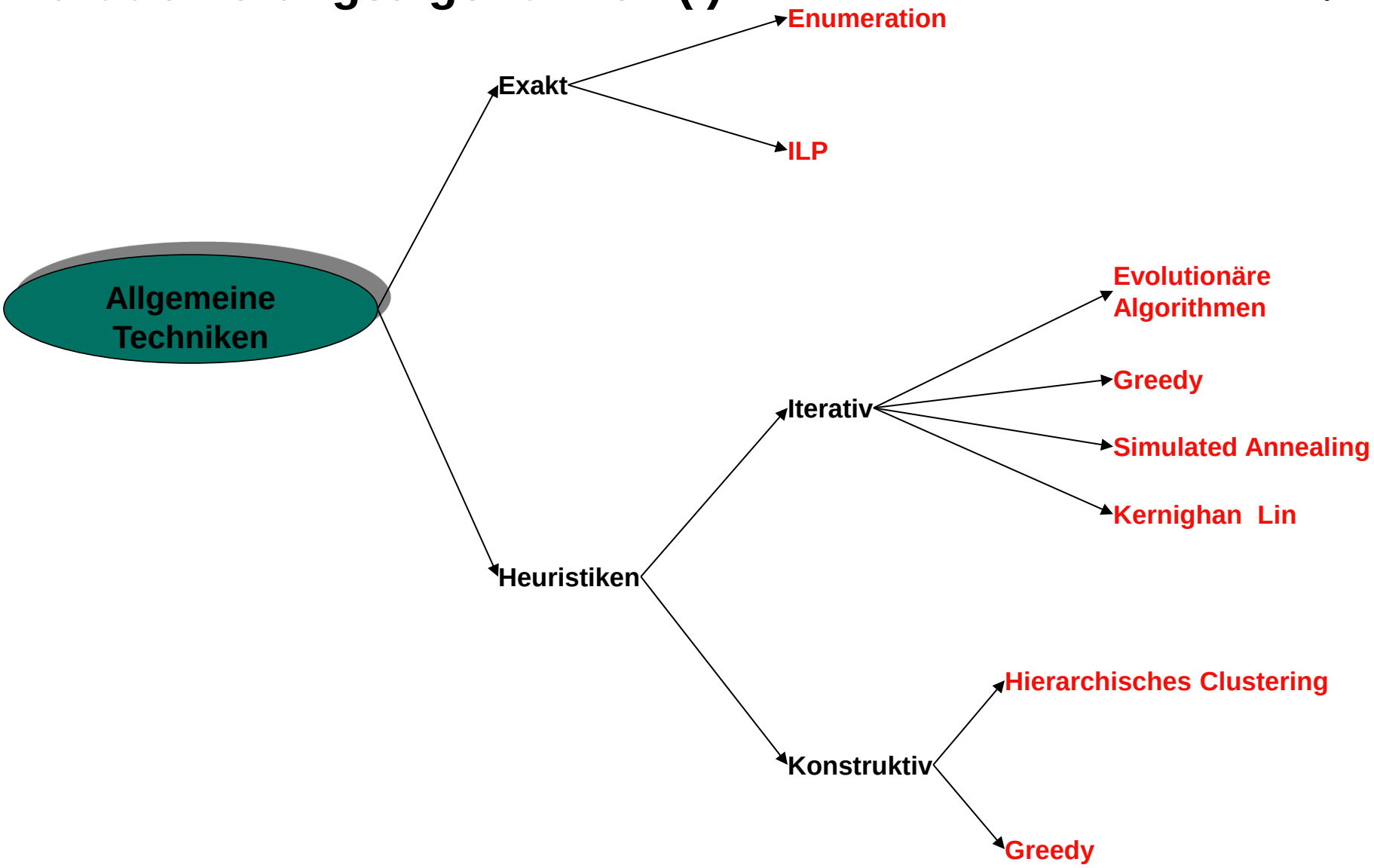
- *C ≙ Systemkosten in [Euro]*
- *L ≙ Ausführungszeit in [sec] (Latency)*
- *P ≙ Leistungsaufnahme in [W]*
- *h_C, h_L, h_P, geben an, wie stark C, L, P die Entwurfsbedingungen (Constraints C, L, P überstrichen) verletzen.*
- *k_1, k_2, k_3 ≙ Gewichtung und Normierung*

- 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

- 4.1 Einführung, Partitionierungsansätze, Komplexität
- **4.2 Klassifikation von Partitionierungsalgorithmen**
- 4.3 Konstruktive Algorithmen
  - 4.3.1 Hierarchisches Clustering
- 4.4 Iterative Algorithmen
  - 4.4.1 Kernighan Lin
  - 4.4.2 Fiduccia Mattheyses
  - 4.4.3 Tabu-Search
  - 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

4.2 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)

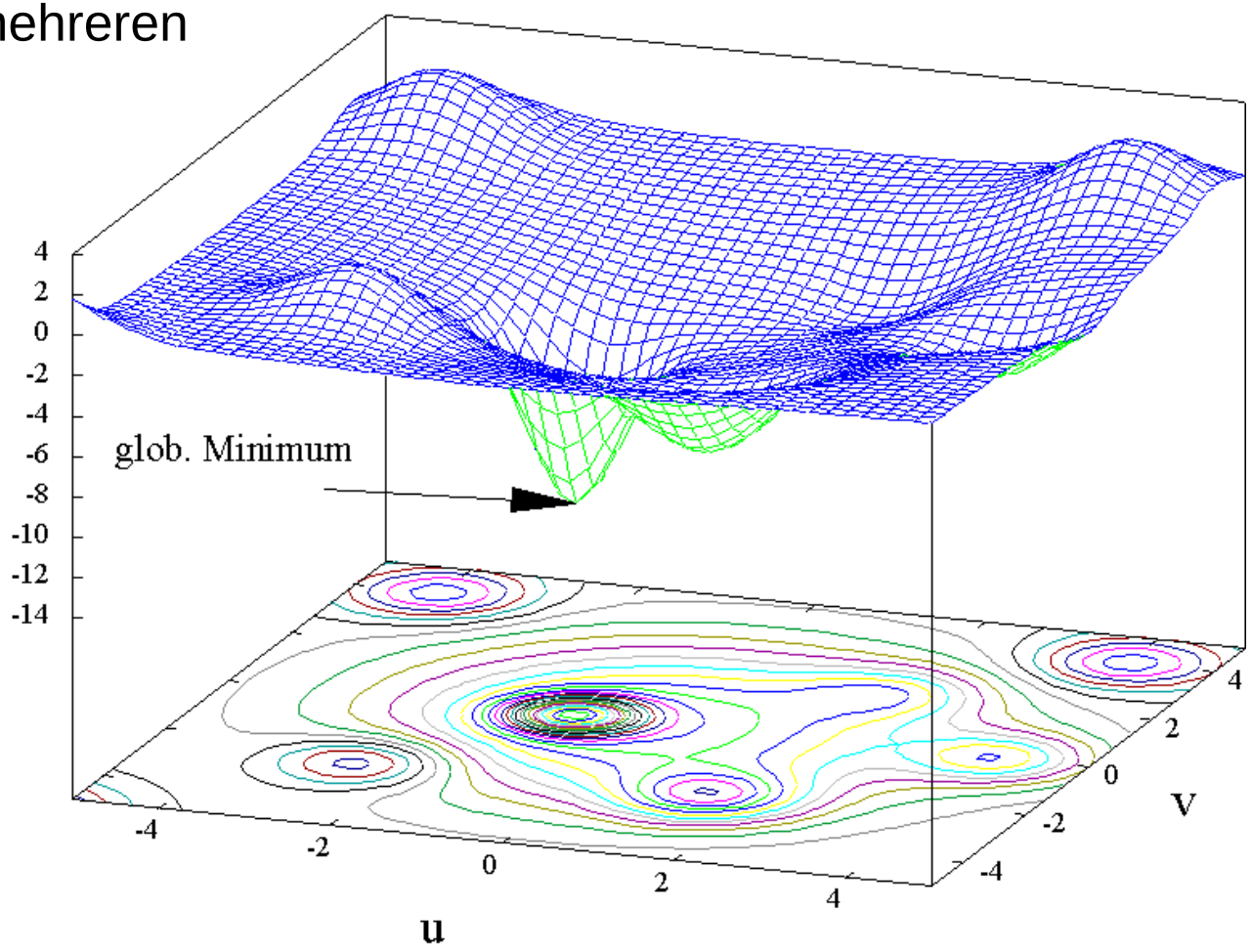
- **Heuristische Lösungsverfahren:**
  - **Findet hinreichend gute Lösung auf Basis unvollständiger Information**
  - **Klassen:**
    - **Konstruktive Verfahren**:
      - Random Mapping
      - Hierarchical Clustering
    - **Iterative Verfahren:**
      - Kernighan Lin Algorithmus (robust und erprobt)
      - 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

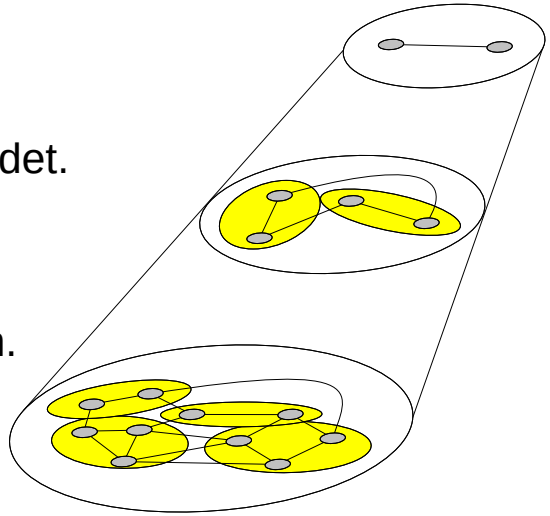
- Wie können Partitionierungsalgorithmen klassifiziert werden?
- Welche Lösungsverfahren gibt es?
- Warum sind iterative Verfahren ‘besser’ als konstruktive Verfahren?
- Wie kann eine Startlösung aussehen?
- Wird immer eine bessere Lösung gefunden?

Inhalt

- 4.1 Einführung, Partitionierungsansätze, Komplexität
- 4.2 Klassifikation von Partitionierungsalgorithmen
- 4.3 Konstruktive Algorithmen
  - **4.3.1 Hierarchisches Clustering**
- 4.4 Iterative Algorithmen
  - 4.4.1 Kernighan Lin
  - 4.4.2 Fiduccia Mattheyses
  - 4.4.3 Tabu-Search
  - 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

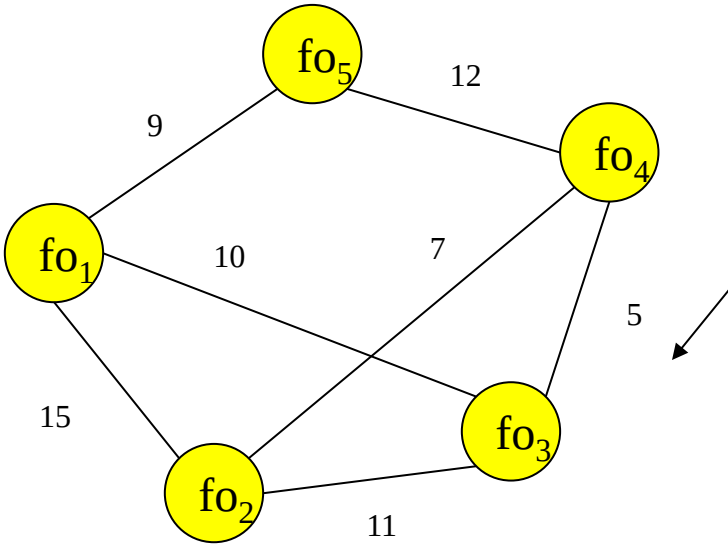
4.3 Konstruktive Verfahren

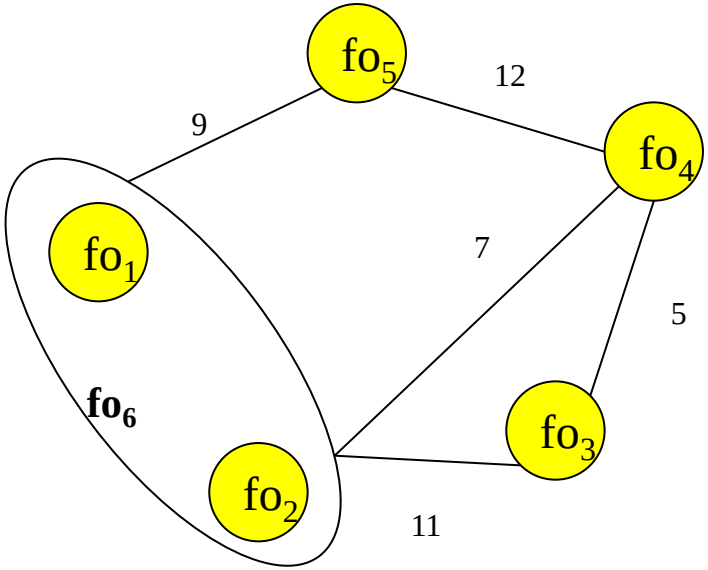
- **Auswahlverfahren:**
  - *Random Mapping*:
    - Jedes Objekt wird zufällig auf einen Block abgebildet.
  - *Hierarchical Clustering*:
    - Schrittweises Zusammengruppieren von Objekten.
    - ***Nähefunktion*** gibt an, wie gewinnbringend die Gruppierung zweier Objekte ist.



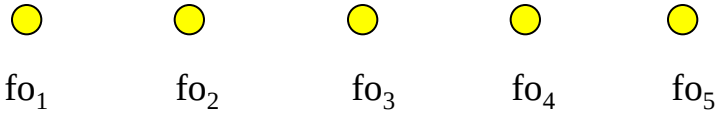
- **Konstruktive Verfahren:**
  - Werden oft verwendet, um eine Anfangspartition für iterative Verfahren zu erzeugen.
  - Haben das Problem, daß es sehr schwierig sein kann, eine geeignete *Nähefunktion* zu definieren.

4.3.1 Hierarchisches Clustering: Beispiel (I)



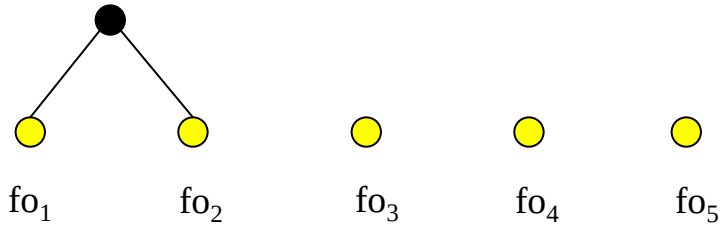


Betrachte jeden Knoten anfänglich als „trivialen“ Cluster.

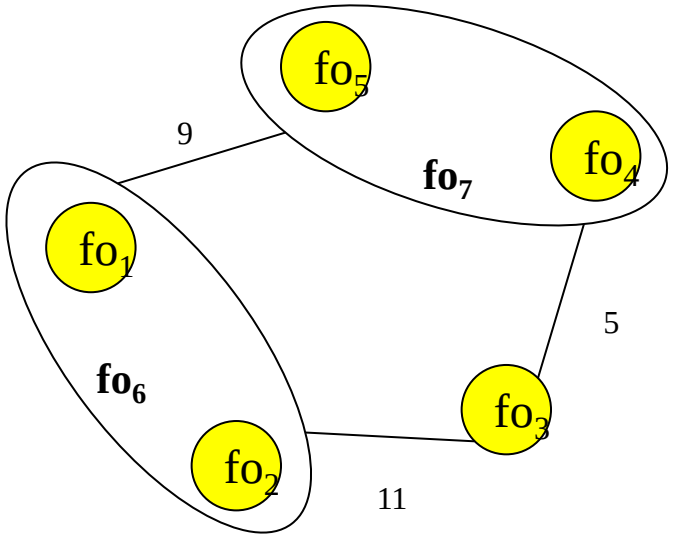


- Verschmelze Knoten f_{o1} und f_{o2}, Kante hat größten Wert 15
- Lösche Kante (f_{o1}, f_{o3}), denn Kante (f_{o2}, f_{o3}) führt auch in den gleichen Cluster und hat einen größeren Wert 11

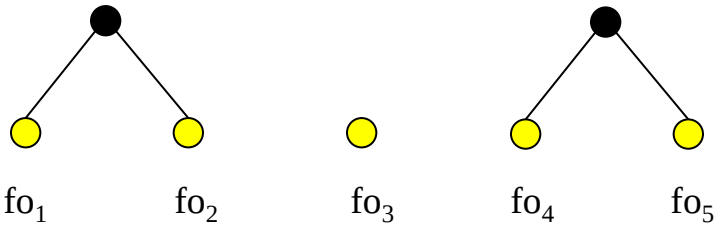
(Variante besteht darin, alle Kanten, die vom gleichen Knoten in den Cluster führen, durch einzelne Kante zu ersetzen und diese zu mitteln)



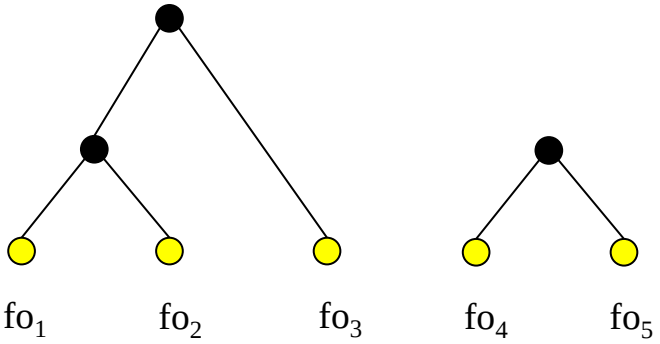
4.3.1 Hierarchisches Clustering: Beispiel (II)

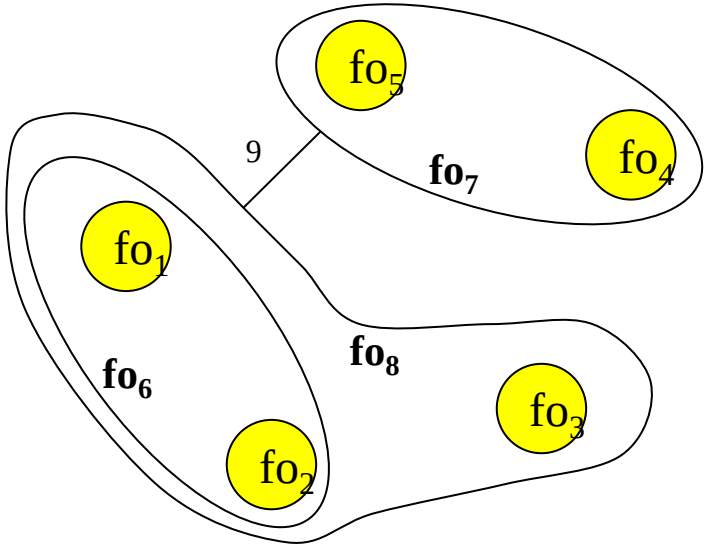


- Verschmelze Knoten f_{o5} und f_{o4}, Kante hat größten Wert 12. Lösche Kante ($[f_{o1}, f_{o2}]$, f_{o4})

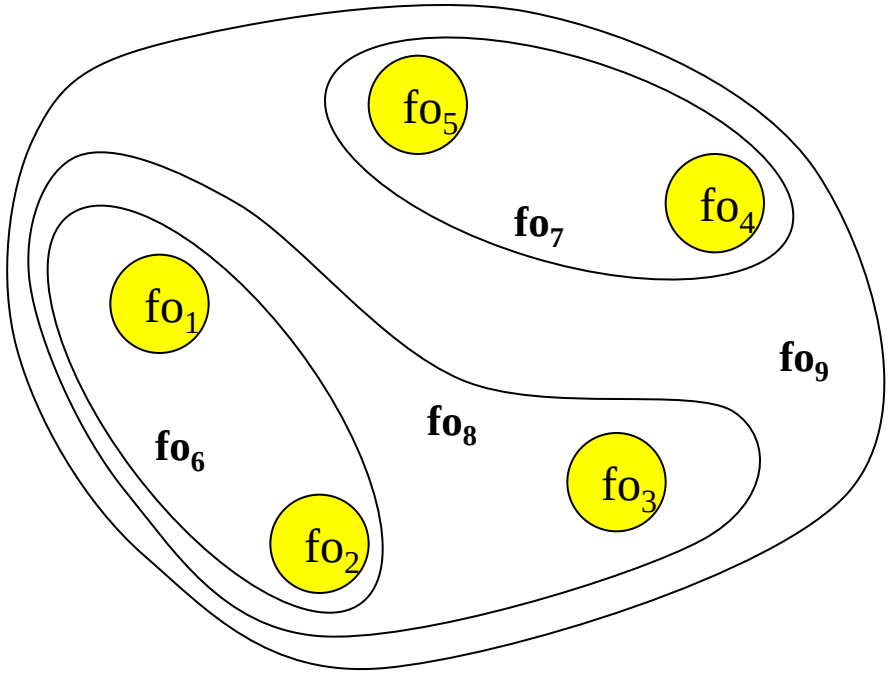


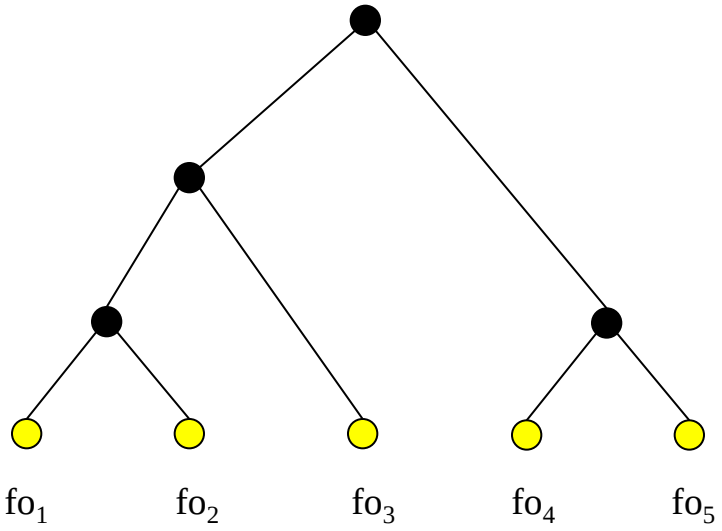
- Verschmelze Superknoten $[f_{o1}, f_{o2}]$ mit f_{o3}, Kante hat größten Wert 11
- Lösche Kante ($[f_{o4}, f_{o5}]$, f_{o3})





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.

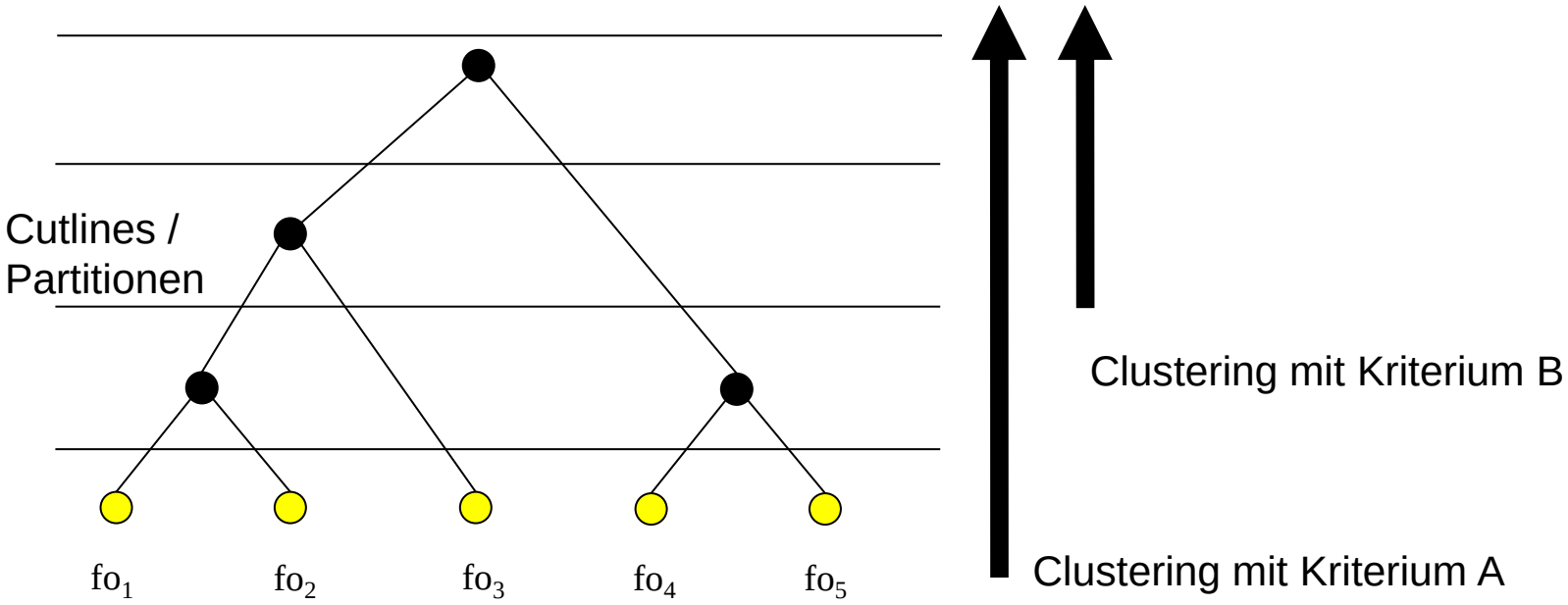
Berechne *Closeness-Function* zwischen den Objekten.

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.

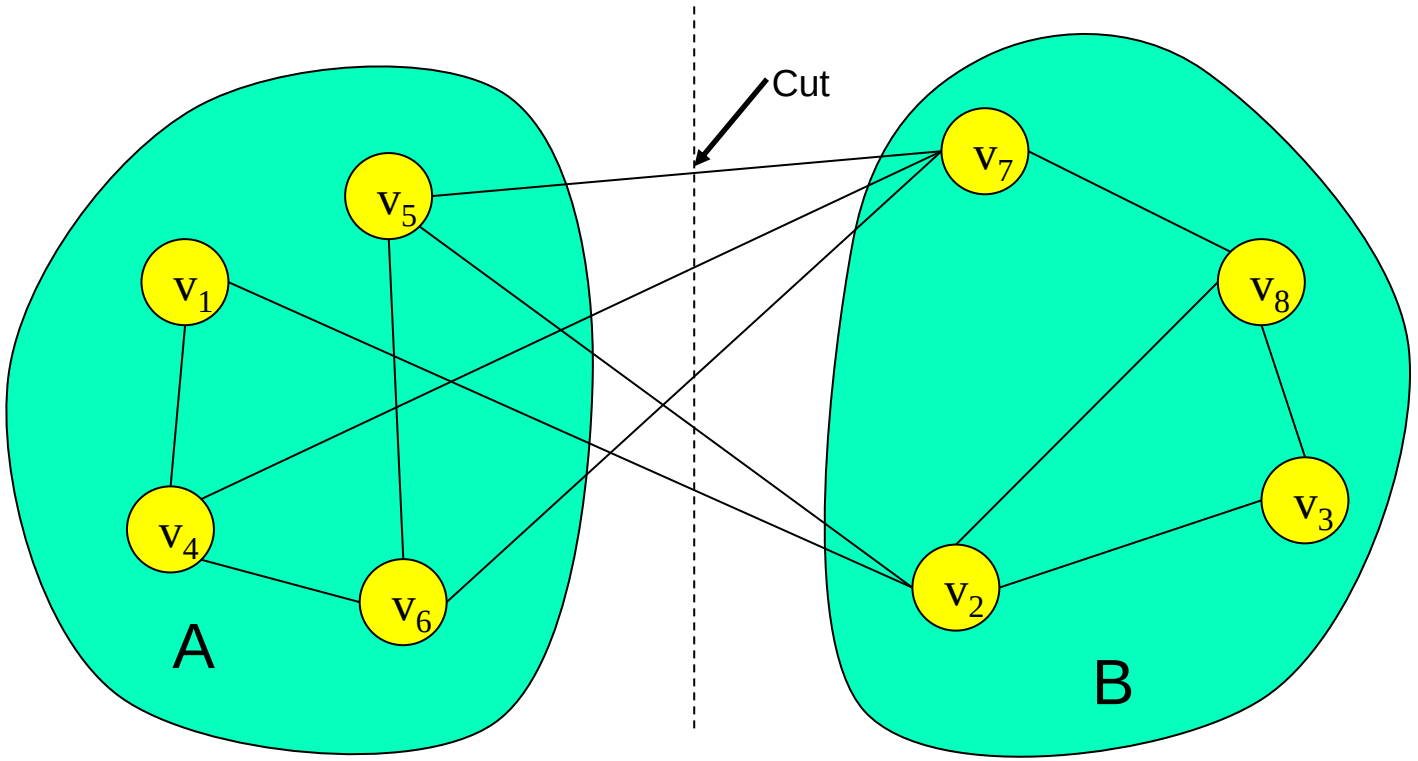
- Wie geht der Algorithmus vor?
- Was ist die Closeness-Funktion?
- Wie muss der Baum geschnitten werden, um bestimmte Partitionierungen zu erreichen?
- Welche Komplexität hat der Algorithmus?

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 Kernighan Lin**
  - 4.4.2 Fiduccia Mattheyses
  - 4.4.3 Tabu-Search
  - 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 Kernighan Lin (KL) Algorithmus (I)

- Anwendung des KL-Algorithmus bei Bipartitionen:
  - Es werden Objekte/Knoten aus den beiden Partitionen paarweise (virtuell) vertauscht und die Vertauschung (reell) für das Paar mit dem größten Kostengewinn durchgeführt. (Min-Cut-Algorithmus)



4.4.1 Kernighan Lin Algorithmus (II)

- **Ziel:** geeignete Vertauschung von Knotenpaaren, so daß die Anzahl der Schnitte zwischen den beiden Partitionen A, B minimal wird.
- Der Algorithmus akzeptiert während der Iterationen vorübergehend auch Kostenverschlechterungen, wodurch die Überwindung lokaler Minima der Kostenfunktion möglich wird.

- Jeder Kante e=(v_i, v_j) wird ein Kostenfaktor c(e_{ij}) zugeordnet.

- **Definition**: Externe Kosten für Knoten $v_i \in A$

$$c_{ext}(v_i) = \sum_{e=(v_i,v_j)\in E,\ v_j \in B} c(e_{ij})$$

- **Definition**: Interne Kosten für Knoten $v_i \in A$

$$c_{\text{int}}(v_i) = \sum_{e=(v_i,v_j)\in E,\ \ v_i,v_j \in A} c(e_{ij})$$

- **Definition**: Gewinn (gain) für Knoten $v_i \in A$, wenn dieser in die Partition B verschoben wird. Der gain entspricht dabei der Verringerung des Kantenschnitts (= Einsparung).

$$gain(v_i) = c_{ext}(v_i) - c_{\text{int}}(v_i)$$

- Analog für Knoten aus Partition B.

4.4.1 Kernighan Lin Algorithmus (III)

- Ein Verschiebung eines Knotens in die andere Partition bewirkt, daß alle bislang externen Kanten zu internen werden und alle internen Kanten jetzt zur Menge der geschnittenen Kanten gehören.
- Wird nun ein Knotenpaar $\{v_i, v_j\}$, mit $v_i \in A$ und $v_j \in B$ gegenseitig vertauscht, so entsteht als **neuer Gewinn**: $gain(v_i) + gain(v_j) - 2c(e_{ij})$
  Die Kosten $c(e_{ij})$ müssen **zweimal abgezogen werden**, weil v_i und v_j **gleichzeitig vertauscht** werden und damit die Kante c_{ij} zweimal zur internen Kante gemacht wird, obwohl sie trotzdem weiterhin geschnitten bleibt.
- Komplexität $O(n^3)$; n = Anzahl der Knoten.

4.4.1 Kernighan Lin Algorithmus: Ablauf

for all v_i **do**

Berechne $c_{ext}(v_i)$, $c_{int}(v_i)$ und $gain(v_i)$

end for

SetOfLockedNodes:=∅

BestCost:=CostPrevStep:=CutSize(P_O)

BestLastStep:=0

Initialisierung
Graph G_P mit n Knoten v und einer Anfangs-partitionierung P_O={P_A, P_B}.

for s= 1 **to** n/2 **do**

Kern- Block

end for

Hauptschleife O(n^3).

if BestLastStep > 0 **then**

for s:= 1 to BestLastStep **do**

„Tausche Knoten, die in Bestpair[s] im jeweiligen Schritt gespeichert wurden.“

end for

end if

Aktualisierung der Vertauschungen.

4.4.1 Kernighan Lin Algorithmus: Kern-Block

BestGain:= -∞

for all pairs $\{v_i, v_j\}$ with $v_i \in P_a$, $v_j \in P_B$, $v_i \notin$ SetOfLockedNodes **and** $v_j \notin$ SetOfLockedNodes **do**

(a) Finde das lokal beste Knotenpaar,
welches bis jetzt noch nicht bewegt wurde.

end for

Blockiere das gefundene Knotenpaar.

(i_{min}, j_{min}):=BestPair[s]

SetOfLockedNodes:= SetOfLockedNodes $\cup \{v_{i_{min}}\} \cup \{v_{j_{min}}\}$

for all v_i **with** $v_i \notin$ SetOfLockedNodes **do**

(b) Aktualisiere die *Gains* von allen Knoten, die mit BestPair[s] verbunden sind.

end for

CostCurrStep :=CostPrevStep – BestGain
CostPrevStep:= CostCurrStep
if CostCurrStep < BestCost **then**
    BestLastStep:=s
    BestCost := CostCurrStep
end if

Beachte: im folgenden Schritt ist eine lokale Kostenerhöhung erlaubt, falls BestGain<0. Gain = Einsparung.

4.4.1 Kernighan Lin Algorithmus: Ablauf

(a)

if $gain(v_i)+gain(v_j)-2c(e_{ij}) > BestGain$ **then**

$BestPair[s]:=(v_i, v_j)$

$BestGain:= gain(v_i)+gain(v_j)-2c(e_{ij})$

end if

Prüfe, ob das aktuelle Knotenpaar besser ist als vorhergehende Kandidaten.

Feldvariable mit besten Knotenpaar für Iteration s.

(b)

if $v_i \in P_a$ **then**

$$gain(v_i)_{new}:=(c_{ext}(v_i)-c(e_{ij_{min}})+c(e_{ii_{min}}))-(c_{int}(v_i)+c(e_{ij_{min}})-c(e_{ii_{min}})) = gain(v_i)_{old} -2c(e_{ij_{min}}) +2c(e_{ii_{min}})$$

else

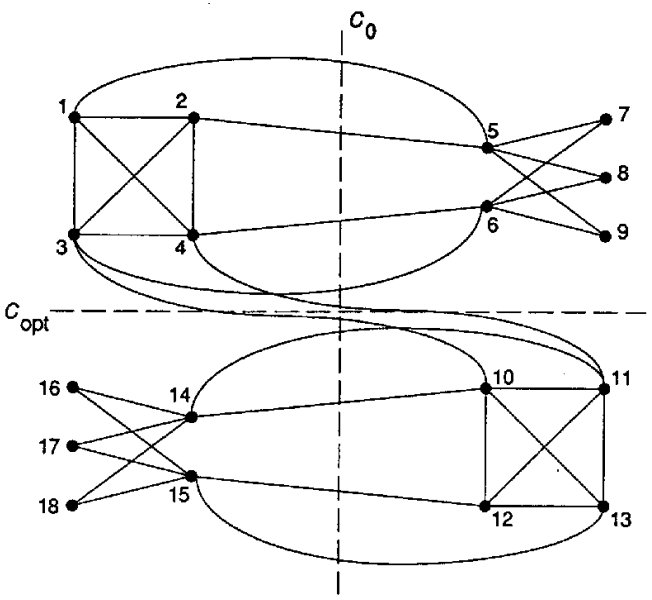
$$gain(v_i)_{new}:=(c_{ext}(v_i)-c(e_{ii_{min}})+c(e_{ij_{min}}))-(c_{int}(v_i)+c(e_{ii_{min}})-c(e_{ij_{min}})) = gain(v_i)_{old} -2c(e_{ii_{min}}) +2c(e_{ij_{min}})$$

end if

Berechne die Gain-Aktualisierung für die Vertauschung.

4.4.1 Kernighan Lin Algorithmus: Beispiel

- Beispiel: 18 Knoten → Bisektionierungen . $\binom{18}{9}/2 = 24310$



Schritt	Knotenpaar	Change	Cutsize
0	-	0	10
1	{4,10}	+2	12
2	{2,12}	0	12
3	{1,13}	-4	8
4	{3,11}	-6	2
5	{7,18}	+4	6
6	{8,17}	+4	10
7	{5,15}	+2	12
8	{9,16}	0	12
9	{6,14}	-2	10

- Eine *Erweiterung des Kernighan Lin Algorithmus*, arbeitet *direkt mit Hyperkanten*.
- Einzelne Knoten werden bewegt (kein paarweises Vertauschen).
- Unbalancierte Partitionen sind möglich → **Fiduccia Mattheyses Algorithmus**
- Effiziente Auswahl zu bewegender Knoten und *Gain*-Update (*kritisches Netz*).
- Komplexität für einen Durchlauf des Basisalgorithmus O(n), n=Anzahl aller Netze bzw. Hyperkanten; (Vergleich dazu Kernighan Lin: $O(n^3)$ n = Anzahl der Knoten).

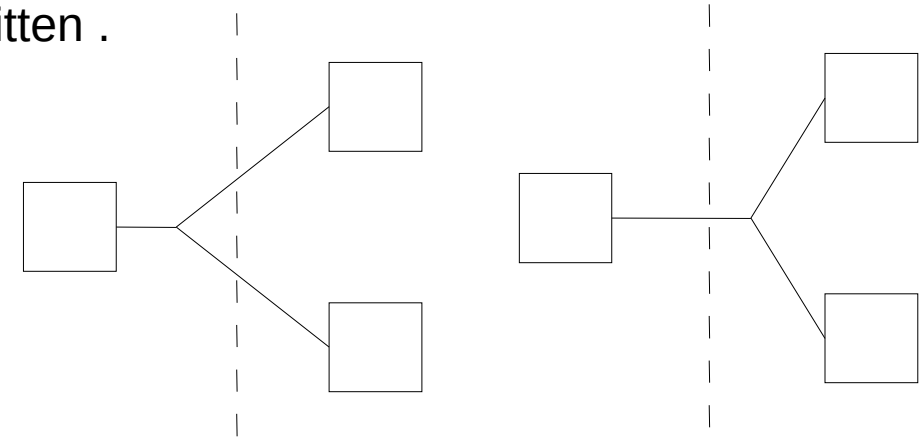
- Welche Voraussetzungen müssen für KL eingehalten werden?
- Wie geht der Algorithmus vor?
- Was ist der gain?
- Warum muss für den BestGain wieder $2c(e_{ij})$ abgezogen werden?
- Welche Komplexität hat der Algorithmus?

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 Kernighan Lin
  - **4.4.2 Fiduccia Mattheyses**
  - 4.4.3 Tabu-Search
  - 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

4.4.2 Fiduccia Mattheyses (FM) Algorithmus (I)

- **Begriffe zum Algorithmus:**
  - Ein **Netz** läßt sich mit der Kante eines **Hypergraphen** darstellen, dessen Knoten gerade den Zellen entsprechen, die über das Netz miteinander verbunden sind. Alle verbundenen Zellen befinden sich dabei auf gleichem „Potential“ .
  - Ein Netz erhält den **Status „Cut“,** wenn es durch eine Bipartitionierung geschnitten wird und seine Zellen damit in unterschiedlichen Bereichen liegen. Dabei spielt es keine Rolle, ob die gezeichnete Schnittlinie im Beispiel unten zwei oder nur eine Teilkante schneidet. Das Netz wird dann nur einmal geschnitten .



4.4.2 Fiduccia Mattheyses Algorithmus (II)

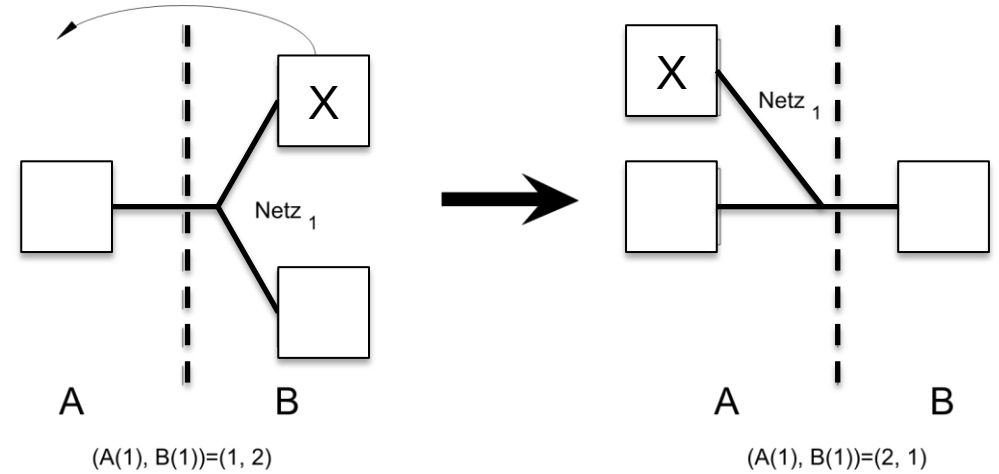
- Als **Cutset** bezeichnet man die Menge der Netze mit dem Status „Cut“, oftmals ist damit auch die Anzahl der geschnittenen Netze gemeint.
- **Gain g(i)** einer Zelle i : Wenn die Zelle i in die andere Partition verschoben wird, so kann sich dadurch die Anzahl der Netze
  mit Status „Cut“ ändern.

  g(i)= „# alter Cutset“ – „# neuer Cutset“

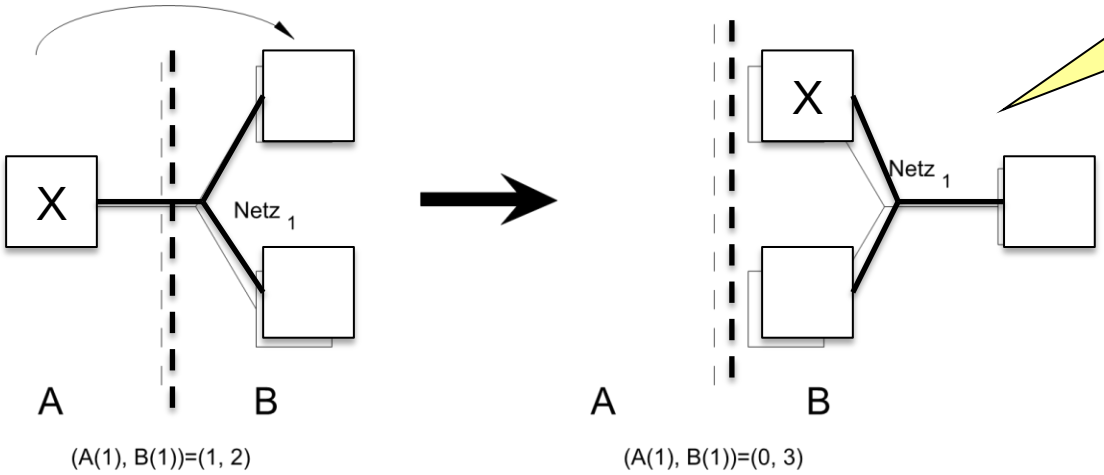
- Wenn g(i) > 0, dann gab es eine Verringerung der Cuts bzw. der Cutset wurde kleiner (= Einsparung).
- Eine **Basiszelle** ist eine Zelle, die für eine Verschiebung in die andere Partition ausgewählt wurde. Sie verletzt nicht das **Balance-Kriterium** und maximiert den Gain g(i), d.h. der Gain g(i) erlangt Zuwachs.
- Man bezeichnet dann ein Netz als **kritisches Netz**, wenn an dem Netz eine Zelle daran hängt, die ‚wenn sie in die andere Partition verschoben wird, den Cut-Zustand des Netzes ändert. Das ist nur der Fall, wenn das Netz in einer der beiden Partitionen (A/B) keine oder nur eine Zelle hat.
  ⇒ **Nur kritische Netze tragen zu einer Änderung des Cutset bei.**

4.4.2 Fiduccia Mattheyses Algorithmus (III)

- Beispiele:



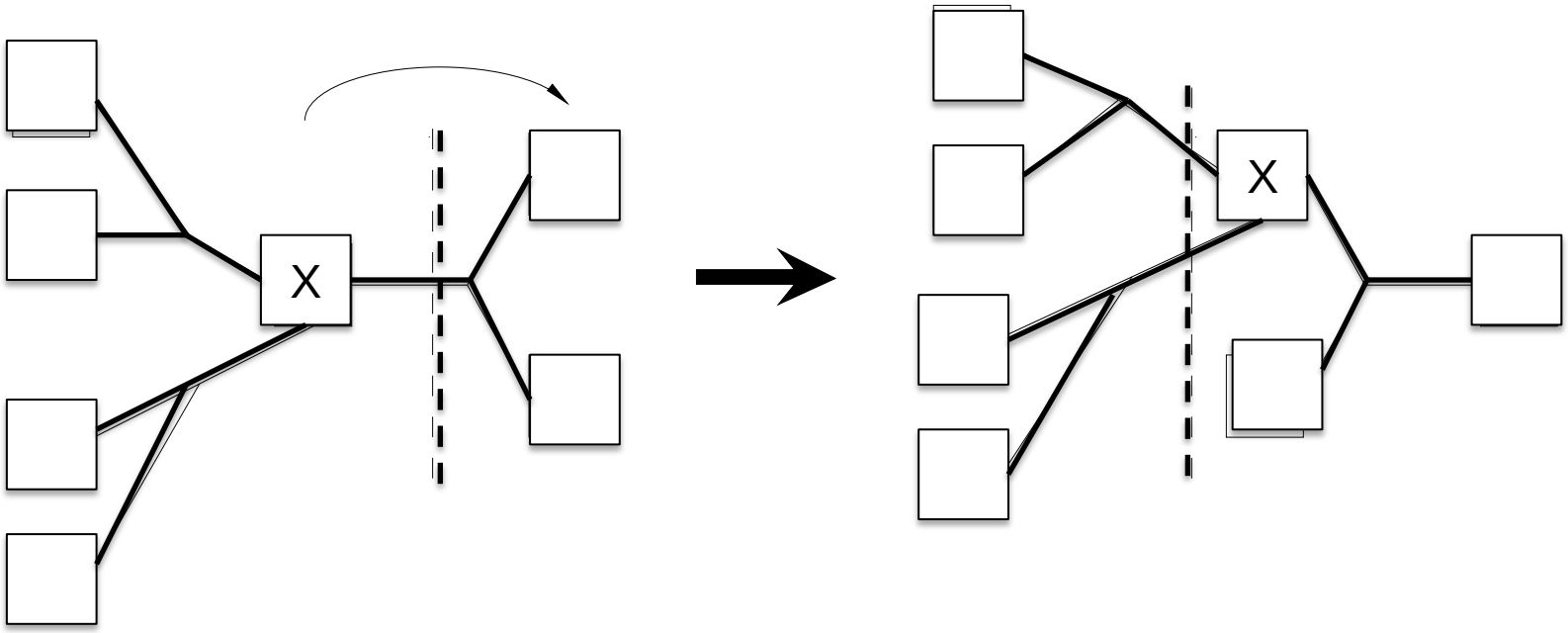
Keine Änderung des Cut-Zustands, ist aber trotzdem ein kritisches Netz



Änderung des Cut-Zustands

4.4.2 Fiduccia Mattheyses Algorithmus (IV)

- Beispiele:



- Die Cut- Zustände von Netz 1 sowie auch Netz 2 und Netz 3 werden durch die Verschiebung der Zelle X in die Partition B geändert.

- **Anmerkung:** (A(n), B(n)) bezeichnet die Distribution des Netzes n in den Partitionen A und B, was der Anzahl der zum gleichen Netz gehörenden Zellen in der jeweiligen Partition entspricht.

4.4.2 Fiduccia Mattheyses Algorithmus (V)

- **Balance- Kriterium**: Jeder Zelle i wird eine Größe (= Fläche) s(i) zugeordnet. Die Größen der Partitionen A, B ergeben sich damit zu:

$$Größe von\ A = |A| = \sum_{Zelle\, i \in A} s(i) \qquad \text{bzw.} \qquad Größe von\ B = |B| = \sum_{Zelle\, i \in B} s(i)$$

- Die Partitionen sind dann balanciert wenn: $\frac{|A|}{|A|+|B|} \approx r$ für ein r : $0 \le r \le 1$

- Der Wert von r ist von der Wahl des Nutzers abhängig. Idealerweise ist $r \approx ½$, aber man kann auch eine Bereich vorgeben z.B. r: [0,3 … 0,7].

- Es gibt auch noch andere mögliche Balance- Kriterien.
  Der **Ratio-Cut** ist ein solches Kriterium (als sekundäres Kriterium).

$$R_{V_A,V_B} = \frac{Cut(V_A,V_B)}{|V_A| \cdot |V_B|} \text{ (zu minimieren)}$$

  Gibt es mehrere Partitionierungen mit gleichem maximalen Gain g(i), so liefert eine besser ausbalancierte Partition ein größeres Produkt im Nenner, wodurch der Ratio-Cut minimiert wird. Eine solche Partition ist dann auszuwählen.

4.4.2 Fiduccia Mattheyses Algorithmus: Ablauf

1. Berechne den ***Gain* gain(i)** für jede Zelle. (Initialisiere j:=1)
2. Suche die Zelle, die Balancierung erfüllt, die nicht markiert ist und maximalen *Gain* hat.
3. Markiere diese Zelle und aktualisiere den Gain der betroffenen kritischen Netze.
4. Wenn es noch unmarkierte Zellen gibt, dann j:=j+1. Merke die Markierungsabfolge.
   Suche dann eine neue Basiszelle (gemäß 2.) und fahre in **Schritt 3** fort.
5. Suche die Sequenz von Verschiebungen, so daß der Gain maximal wird. Wenn es mehrere Verschiebungen mit demselben Gain gibt, dann wähle die Sequenz mit der besten Balancierung aus (nutze dazu den Ratio-Cut).
   Wenn Gain ≤ 0, dann beende den Algorithmus und **gehe nach 8**.
6. Führe die angenommenen Verschiebungen tatsächlich durch und hebe alle Markierungen auf.
7. Fahre bei **Schritt 1** solange fort, bis das Optimum erreicht wurde.
8. Fertig.

- **Beachte:** Bei der Kostenberechnung werden nur kritische Netze betrachtet. Der Cut kann sich nur dann verändern, wenn eine Zelle eines kritischen Netzes verschoben wird, die entweder alleine in einer Partition ist oder wenn die andere Partition keine Zellen enthält, die am gleichen Netz hängen.
- **Vorteil:** Gegenüber dem Kernighan Lin Algorithmus hat der Algorithmus eine wesentlich geringere Komplexität. (O(n); n = Anzahl Hyperkanten/Netze; für einen vollständigen Durchlauf von Schritt 1 bis 6.)

- Was ist der Unterschied zu KL?
- Wie ist der Gain definiert?
- Was ist der Ratio-Cut?
- Wie geht der Algorithmus vor?
- 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 Kernighan Lin
  - 4.4.2 Fiduccia Mattheyses
  - **4.4.3 Tabu-Search**
  - 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

4.4.3 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.

4.4.3 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**.

4.4.3 Tabu-Search Algorithmus (III)

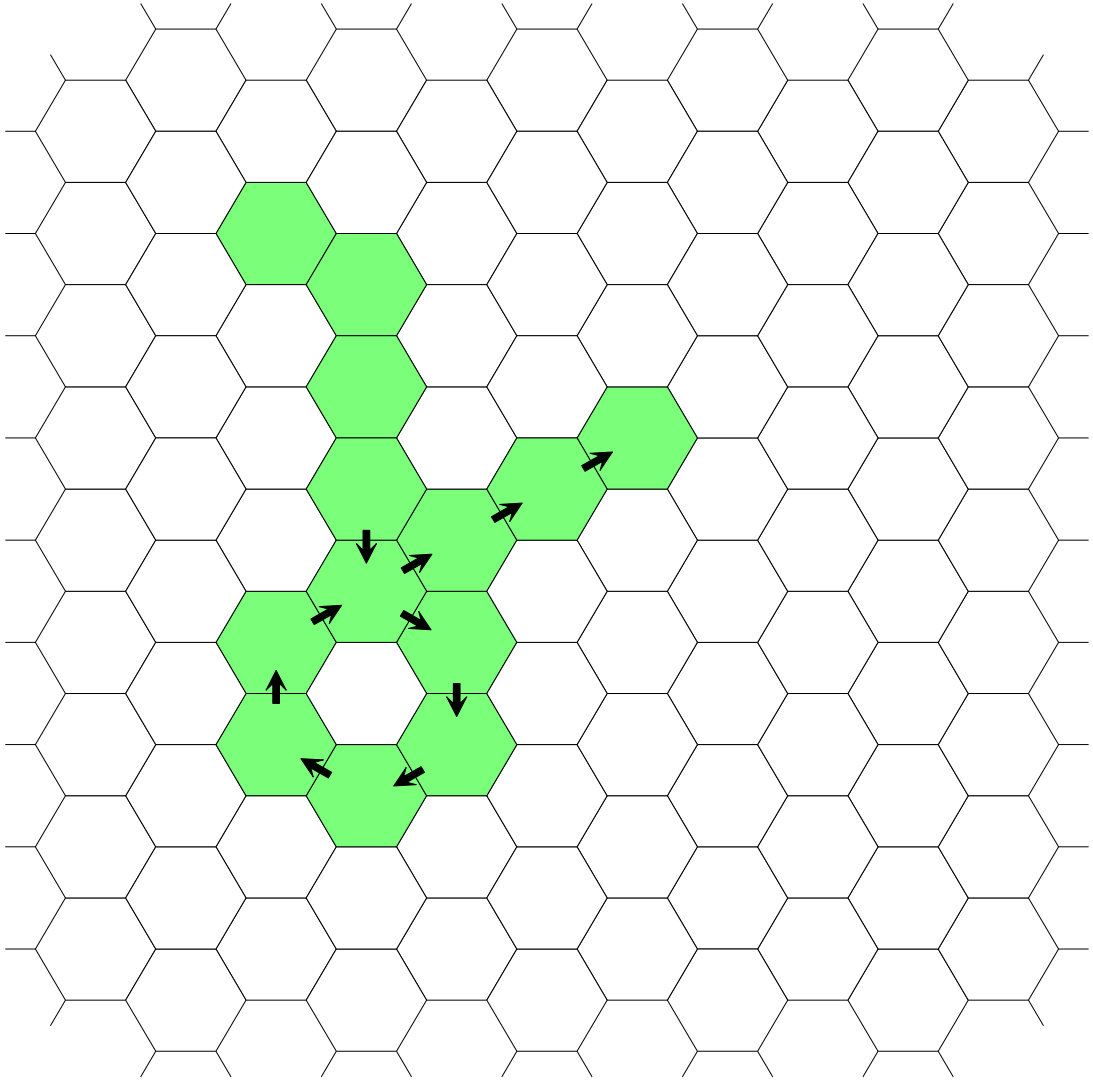
```
CurrentSolution:=InitialSolution
BestSolution:= CurrentSolution
TabuList:={ }
repeat
        N:=k_BestNeighbours(CurrentSolution)
        NewSolution:=none
        for i:=1 to k do

                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

4.4.3 Tabu-Search Algorithmus: Beispiel (I)

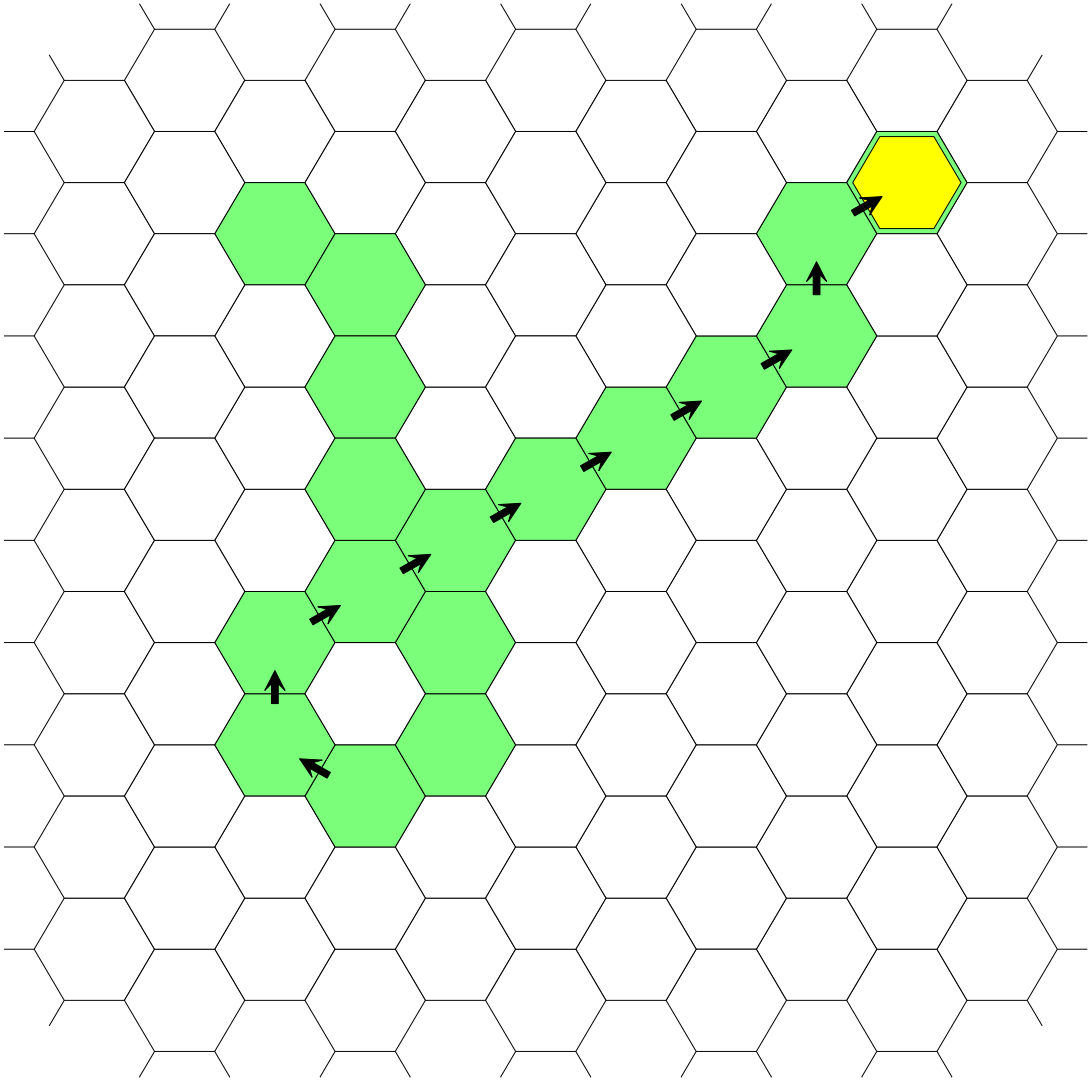


n_{Tabu_Liste}=10 k=6

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

4.4.3 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

4.4.3 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?

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 Kernighan Lin
  - 4.4.2 Fiduccia Mattheyses
  - 4.4.3 Tabu-Search
  - **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

4.4.4 Integer Linear Programming (ILP)

- **Anwendung auf das Partitionierungsproblem:**
  - **binäre Entscheidungsvariable** $x_{i,k}$ = 1: wenn Objekt o_i im Block p_k
  - Kosten $c_{i,k}$, wenn Objekt o_i im Block p_k ist
  - ganzzahliges lineares Programm für m- Partitionen, n Objekte:
    - $x_{i,k} \in \{0, 1\}$ $1 \le i \le n$, $1 \le k \le m$

$$\sum_{k=1}^{m} x_{i,k} = 1 \qquad 1 \le i \le n$$

$$\text{minimiere} \quad \sum_{k=1}^{m} \sum_{i=1}^{n} x_{i,k} \cdot c_{i,k} \qquad 1 \le k \le m,\ 1 \le i \le n$$

  - Die Beschränkungen werden durch Nebenbedingungen modelliert:
    - z.B. maximale Anzahl von h_k Objekten im Block k

$$\sum_{i=1}^{n} x_{i,k} \le h_k \qquad 1 \le k \le m$$

    - schwierig, wenn Nebenbedingungen nicht linear sind
- Anm: Detaillierte Betrachtung von ILP erfolgt in der Vorlesung HSO

- Weitere Informationen zu ILP in der Übung und in HSO

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 Kernighan Lin
  - 4.4.2 Fiduccia Mattheyses
  - 4.4.3 Tabu-Search
  - 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

4.4.5 Simulated Annealing

- **Probabilistisches kombinatorisches Optimierungsverfahren:**
  - Robustes stochastisches sowie problemunabhängiges Verfahren.
  - Gesteuerte Enumeration, Anleihen aus der Natur (Festkörperphysik)
  - Metalle und Glas nehmen beim Abkühlen unter bestimmten Bedingungen einen Zustand minimaler Energie ein:
    - bei jeder Temperatur wird ein thermodynamisches Gleichgewicht erreicht;
    - die Temperatur wird beliebig langsam erniedrigt.
  - Wahrscheinlichkeit, daß ein Teilchen in einen Zustand höherer Energie springt:

$$P(e_i, e_j, T) = e^{\frac{e_i - e_j}{k_B \cdot T}}$$

  - Anwendung auf kombinatorische Optimierung:
    - Energie ≅ Kosten der Lösung.
    - Verringerung der Kosten mit simulierter Temperatur, aber manchmal auch Akzeptieren von Kostenerhöhungen.

4.4.5 Simulated Annealing: Algorithmus (I)

- **Probabilistisches Suchverfahren:**
  - Die äußere Schleife wird solange iteriert, bis ein vordefiniertes Abbruchkriterium erfüllt ist.
  - In jedem Durchlauf der äußeren Schleife wird der Wert einer Temperatur T, ausgehend von einer Starttemperatur kontinuierlich reduziert. In einer inneren Schleife wird ein neuer Zustand s_{new} generiert und mit einer Kostenfunktion c(s) bewertet.
  - Wird der neu generierte Zustand mit geringeren Kosten bewertet als der bisher eingenommene Zustand s, so wird ein Zustandsübergang nach s_{new} bedingungslos akzeptiert. Ist $c(s_{new}) > c(s)$, dann wird der neue Zustand nur akzeptiert, wenn die Akzeptanzfunktion einen größeren Wert liefert als eine Zufallszahl aus dem Intervall [0, 1].
  - Dieses Verfahren erlaubt es, auch aus einem lokalen Minimum zu entweichen, indem es auch Verschlechterungen akzeptieren kann.
    ⇒ Bei unendlich kleinen Abkühlschritten und unendlich vielen Iterationen wird ein globales Optimum erreicht .

Zum nachlesen

4.4.5 Simulated Annealing: Algorithmus (II)

```
procedure simulated_annealing(S,c)
begin
    s:=start_state(S)
    T:=start_temperature(S)
    repeat
        while not stationary_state(S,T) do
            s_new:=random_modify(s)
            if random([0,1]) ≤ e^(-(c(s_new)-c(s))/T) then
                s:=s_new
        end
        T:=update(T)
    until convergence
end
```

Initiallösung aus Lösungsraum S

hohe Anfangstemperatur

*Abbruchkriterium **siehe (1)***

*Geringfügige Änderung der aktuellen Lösung **siehe (2)***

*Metropolis- Kriterium **siehe (3)***

Akzeptanz der neuen Lösung

*kühle Temperatur weiter ab **(4)***

*Konvergenz / Abbruchkriterium **siehe (5)***

4.4.5 Simulated Annealing

(1) Abbruch, wenn keine Akzeptanz während einer vorgegebenen Anzahl von Iterationen.

(2) Verändere s geringfügig. Bietet verschiedene Möglichkeiten z.B. Verschiebung eines Moduls oder Vertauschung zweier Module.

(3) Metropolis- Kriterium steuert die Akzeptanz neuer Lösungen:

> Je höher die Temperatur, desto höher ist auch die Wahrscheinlichkeit, dass schlechtere Ergebnisse akzeptiert werden. Bei großem T geht der Exponent auch bei einer Verschlechterung gegen 0, wodurch: $e^x \rightarrow 1$, so dass die Wahrscheinlichkeit für eine Akzeptanz steigt .

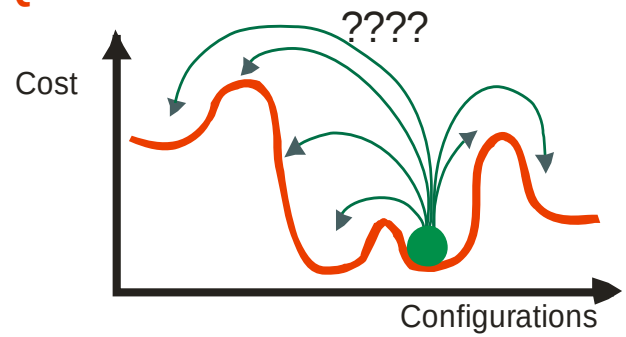
(4) Die Temperatur ist i.d.R. eine Funktion $T_{neu}=\alpha \cdot T_{alt}$, wobei α nicht konstant sein muß.

(5) Konvergenz, falls über eine vorgegebene Anzahl von Iterationen keine Verbesserung der Kostenfunktion erfolgte oder wenn $T < T_{min}$

4.4.5 Simulated Annealing: Einfluss der „Temperatur“

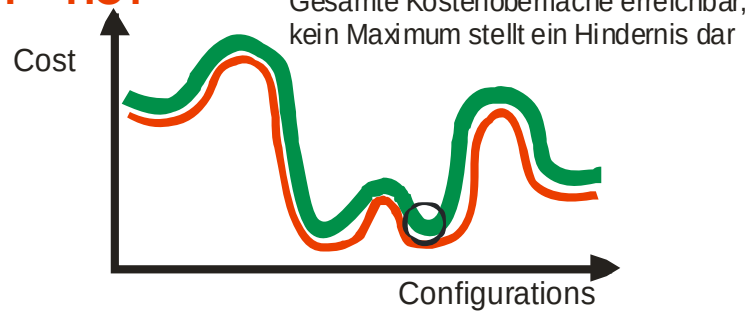
Karlsruher Institut für Technologie

Question

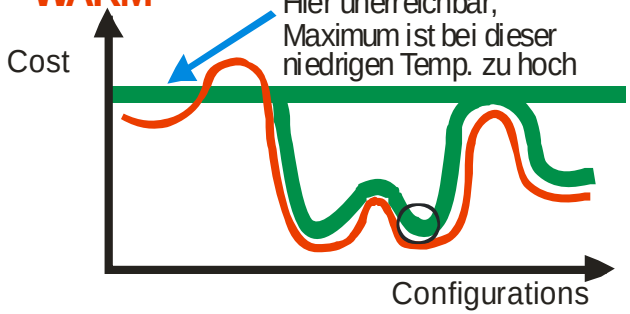


- **Erreichbarkeit benachbarter bzw. entfernter Lösungen in Abhängigkeit von der Temperatur T**
- **Hill-Climbing Eigenschaft**

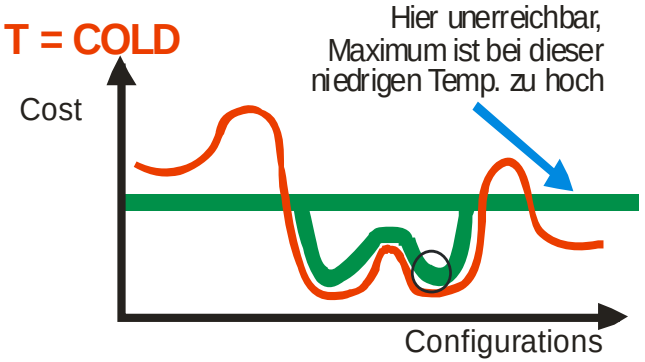
T = HOT



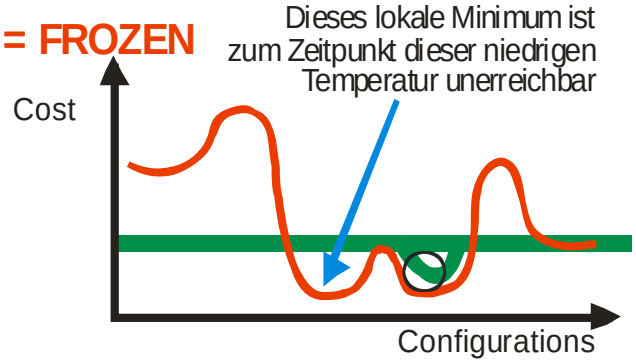
T = WARM



T = COLD

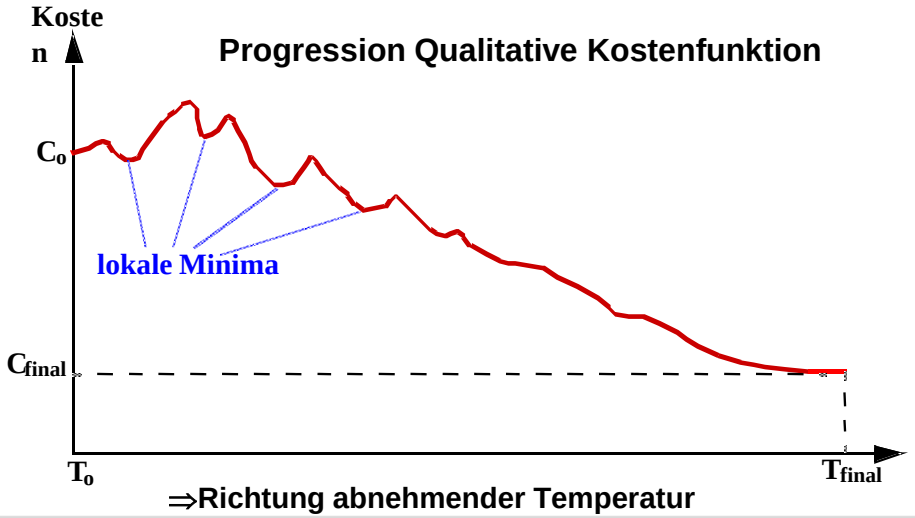


T = FROZEN



4.4.5 Simulated Annealing

- Zeitkomplexität:
  - von exponentiell bis konstant, je nach Implementierung der Funktionen *update(T), stationary_state(S,T), convergence*
  - je länger die Laufzeit, desto besser die Ergebnisse
  - bei unendlicher Laufzeit und $\Delta T \rightarrow 0$ kann globales Optimum erreicht werden.
  - üblich: Funktionen so konstruiert, dass polynomielle Laufzeit erreicht wird
- Gleichgewicht: *stationary_state*
  - nach bestimmter Anzahl von Iterationen
  - oder wenn sich keine Verbesserung mehr ergibt



- An welches Fachgebiet ist der Algorithmus angelehnt?
- Wie geht der Algorithmus vor?
- Was ist das Metropolis-Kriterium?
- Was ist Hill-Climbing?
- Welche Komplexität hat der Algorithmus?

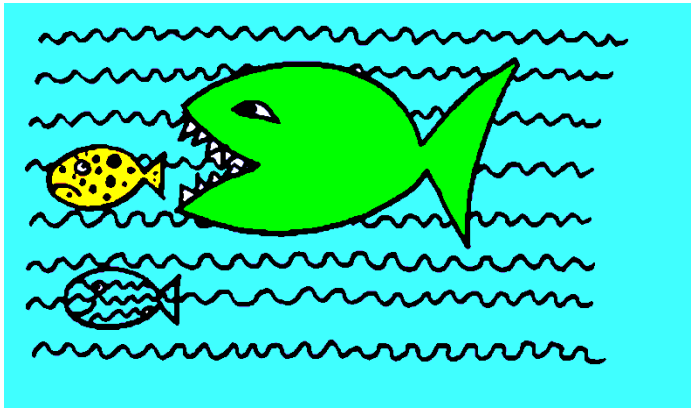
Inhalt

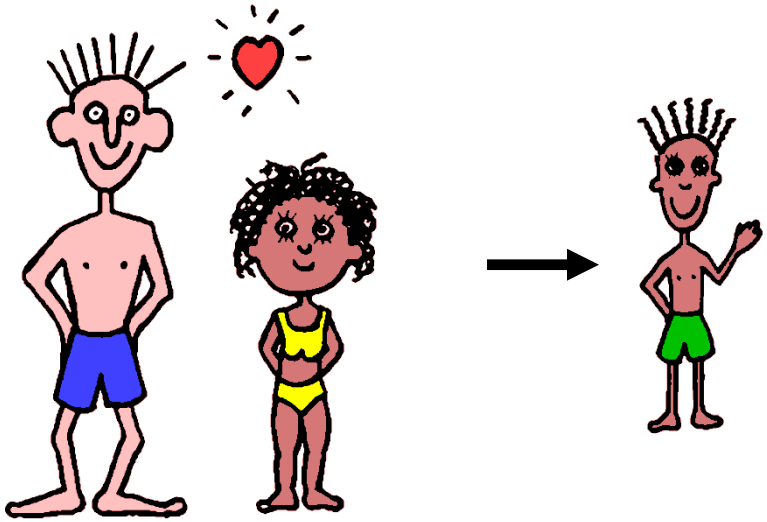
- 4.1 Einführung, Partitionierungsansätze, Komplexität
- 4.2 Klassifikation von Partitionierungsalgorithmen
- 4.3 Konstruktive Algorithmen
  - 4.3.1 Hierarchisches Clustering
- 4.4 Iterative Algorithmen
  - 4.4.1 Kernighan Lin
  - 4.4.2 Fiduccia Mattheyses
  - 4.4.3 Tabu-Search
  - 4.4.4 Integer Linear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

4.4.6 Genetische Algorithmen (GA): Prinzipien der Evolution

Selektion

- Individuen einer Population mit „ungünstigen“ Merkmalen (hier z.B. fehlende Tarnung) werden selektiert (hier gefressen).





Mutation

- Individuen einer Population können ihre Merkmale ändern.

Kreuzung

- Individuen einer Population können ihre Merkmale kombinieren, wobei ein Merkmal (hier z.B. Hautfarbe) ein anderes gleichen Typs dominieren kann.

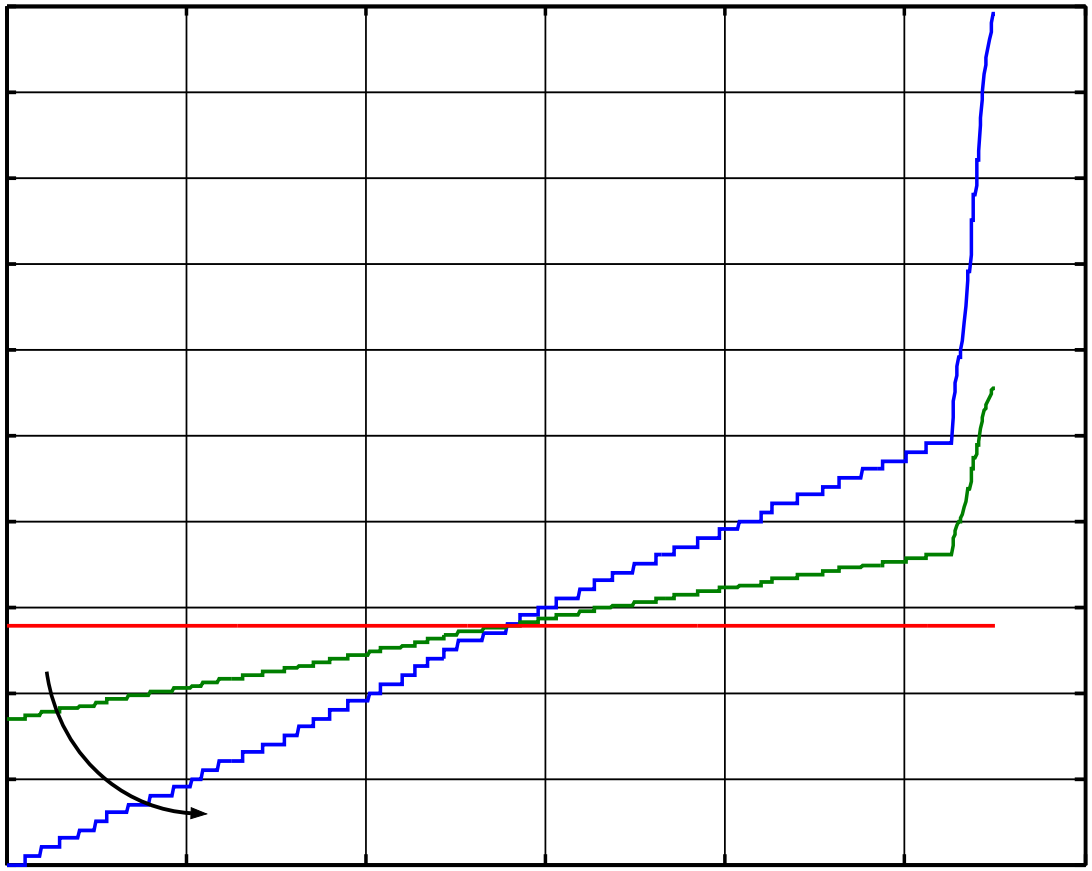
4.4.6 Genetische Algorithmen: Parameter

- GA‘s sind **evolutionäre Verfahren** zur Lösung von Optimierungsaufgaben.
- **Grundprinzip:** Erzeugung von Lösungsmengen, die von den evolutionären Grundprinzipien **Kreuzung, Mutation und Selektion** in neue Populationen überführt werden.
- Relevante Parameter sind:
  - die Größe einer Population, sowie die Anzahl der Nachkommen einer Generation;
  - die Kreuzungswahrscheinlichkeit p_c und Mutationswahrscheinlichkeit p_m
- Der Auswahl der Kandidaten der Nachfolgegeneration liegt das Prinzip des Überlebens der besten Individuen zugrunde.
- Ein Qualitätsmaß *q(Ind)* wird zur Auswahl der besten Individuen für die Nachfolgegeneration verwendet.
- Bei Anwendung auf das Partitionierungsproblem ist das Qualitätsmaß bestimmt durch die Kostenfunktion des Partitionierungsansatzes.
- Skalierung des Qualitätsmaßes notwendig, damit sehr dominante Individuen nicht die ganze Population dominant machen ($c \approx 2$).

$$q_s(Ind) = \frac{c \cdot q_{average} - q_{average}}{q_{\max} - q_{average}} \cdot \left(q(Ind) - q_{average}\right) + q_{average}$$

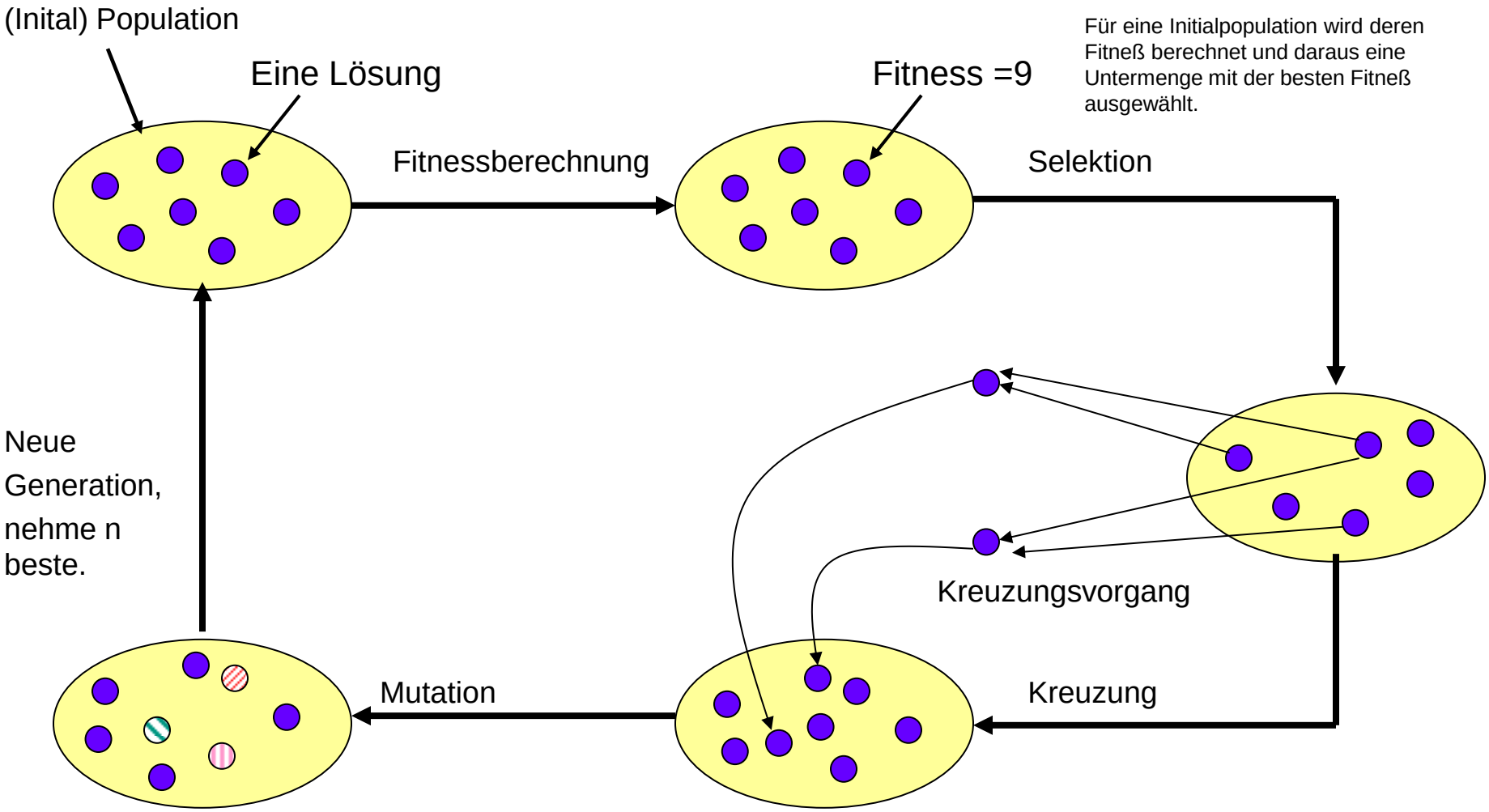
- Differenzmaß für die mittlere Qualität der Population, welches über ein Kontrollparameter c beeinflußt werden kann.

4.4.6 Genetische Algorithmen: Parameter - Beispiel



- Einfluss der Skalierungsfunktion auf das Qualitätsmaß der Individuen
- Bewirkt für $c \approx 2$ eine Anhebung zu gering bewerteter Individuen in Richtung qaverage
- Individuen, die viel zu dominant sind, werden weniger stark bewertet (d.h. Absenkung).

4.4.6 Genetische Algorithmen: Evolutionsprinzip



4.4.6 Genetische Algorithmen: Algorithmus

Pop:=Initial_Population

REPEAT

FOR i:= 1 **TO** n_Children **DO**

{Father, Mother}:=SelectParents(Pop)

Child:=GenerativeCrossover(Father, Mother, p_c)

Pop:=Pop $\cup$ Child

ENDFOR

FOR_ALL Individual $\in$ Pop **DO**

Individual :=Mutate(Individual , p_m)

ENDFOR

Pop:=SelectFittest(Pop, n)

UNTIL StoppingCriterion = true

RETURN Best Configuration out of Pop

Kreuzung von zwei Elternteilen mit Crossover- Wahrscheinlichkeit p_c. Bedingt durch p_c kann Child auch leer sein.

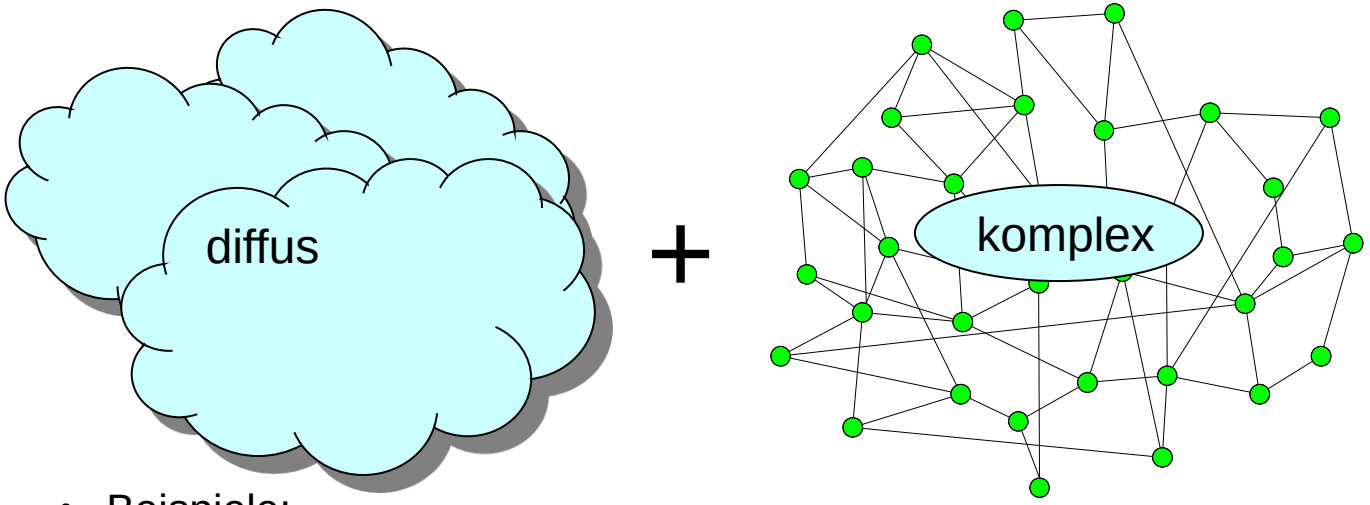
Mutation jedes Individuums mit Wahrscheinlichkeit p_m .

Auswahl der n besten Individuen für die Nachfolgegeneration.

Nach einer bestimmten Anzahl von Generationen wird die beste Konfiguration ausgewählt und zurückgeliefert.

4.4.6 Genetische Algorithmen: Einsatzbereiche

- Das Problem ist oft



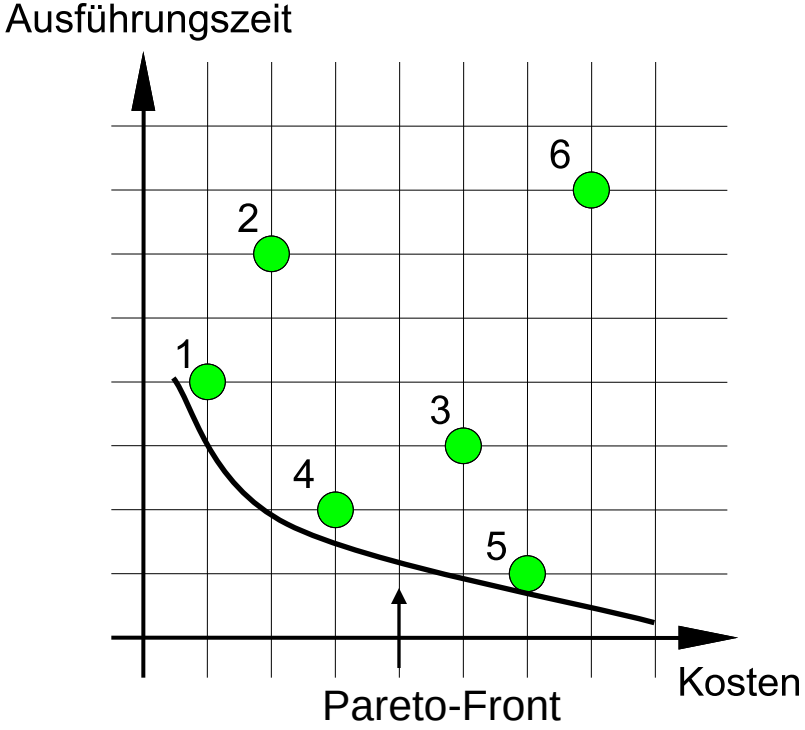
  - Beispiele:
    - Systemsynthese
    - Wegelaufplanung in der Robotik
    - Containerbeladung, Stundenplanerstellung

- **Mehrzieloptimierung**
  - mehrere Kriterien, die miteinander in Konflikt stehen:
    - z.B. Performance vs. Kosten vs. Leistungsaufnahme
  - Genetische Algorithmen finden **Pareto-Fronten** (Menge von **Pareto**- Punkten)

4.4.6 Mehrzieloptimierung: Dominanz, Paretopunkte, Pareto-Ranking

- **Definition:** Ein (Entwurfs)punkt J_i wird vom Punkt J_k dominiert, wenn J_k in jedem Kriterium **gleich oder besser** als J_i ist: $J_i \succ J_k$
- **Definition:** Ein Punkt ist **Pareto-optimal** bzw. ein **Pareto-Punkt**, wenn er von keinem anderen Punkt dominiert wird.

Fitnessfunktion: $$F'(J) = \sum_{i=1, J \neq J_i}^{N} \begin{cases} 1: & J \succ J_i \\ 0: & sonst \end{cases}$$



F´(J) liefert die Anzahl der Lösungspunkte aus der Gesamtmenge, welche den Punkt J dominieren, d.h. in ihren Eigenschaften gleich oder besser sind.

F´(1)=0 F´(2)=1

F´(3)=1 F´(4)=0

F´(5)=0 F´(6)=5

(hier kleine Werte besser!)

4.4.6 Genetische Algorithmen: Mehrzieloptimierung

- Abbildung der möglichen Lösungen / Implementierungen des Entscheidungsraums mittels einer Zielfunktion (Kostenfunktion) auf einen Zielraum von ggf. niedrigerer Dimensionalität.

Entscheidungsraum

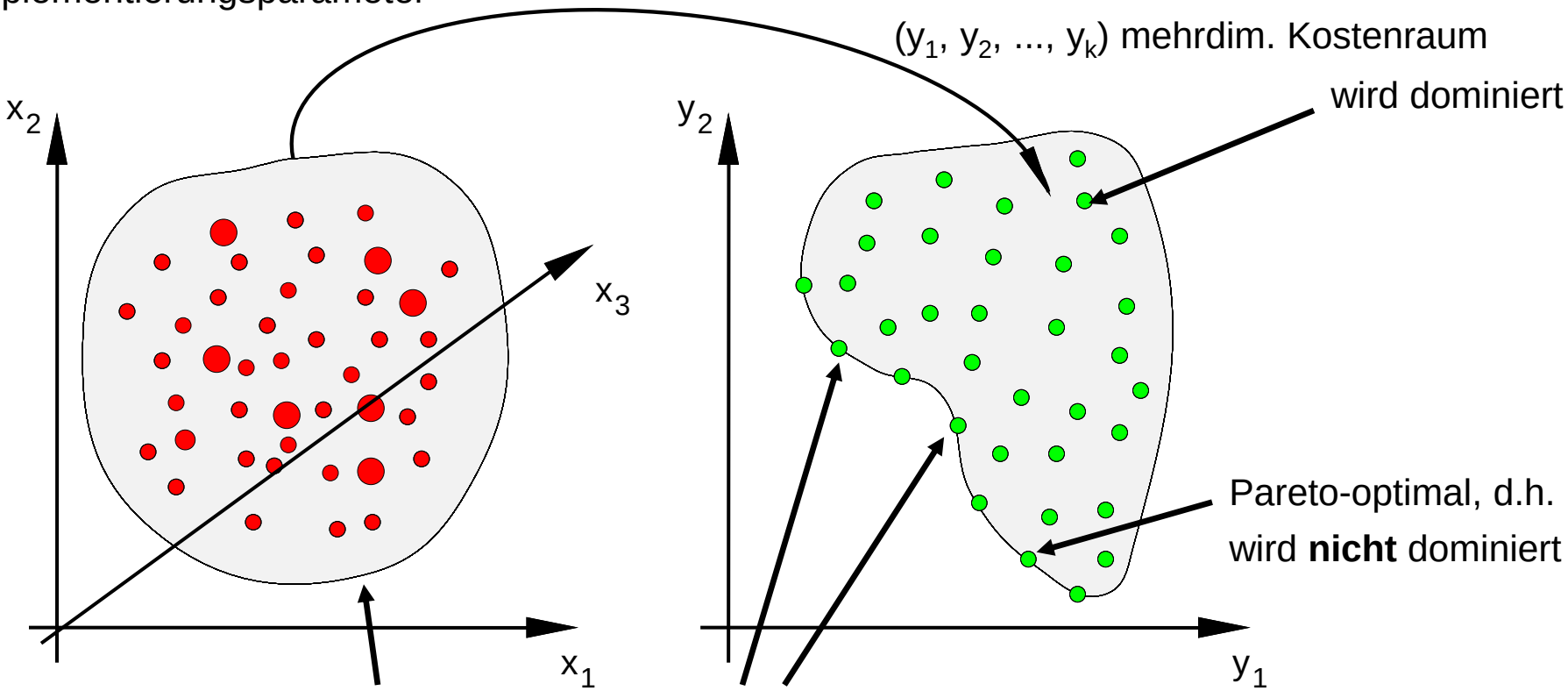
$(x_1, x_2, ..., x_n)$

Implementierungsparameter

minimiere f

Objekt- /Zielraum

$(y_1, y_2, ..., y_k)$ mehrdim. Kostenraum



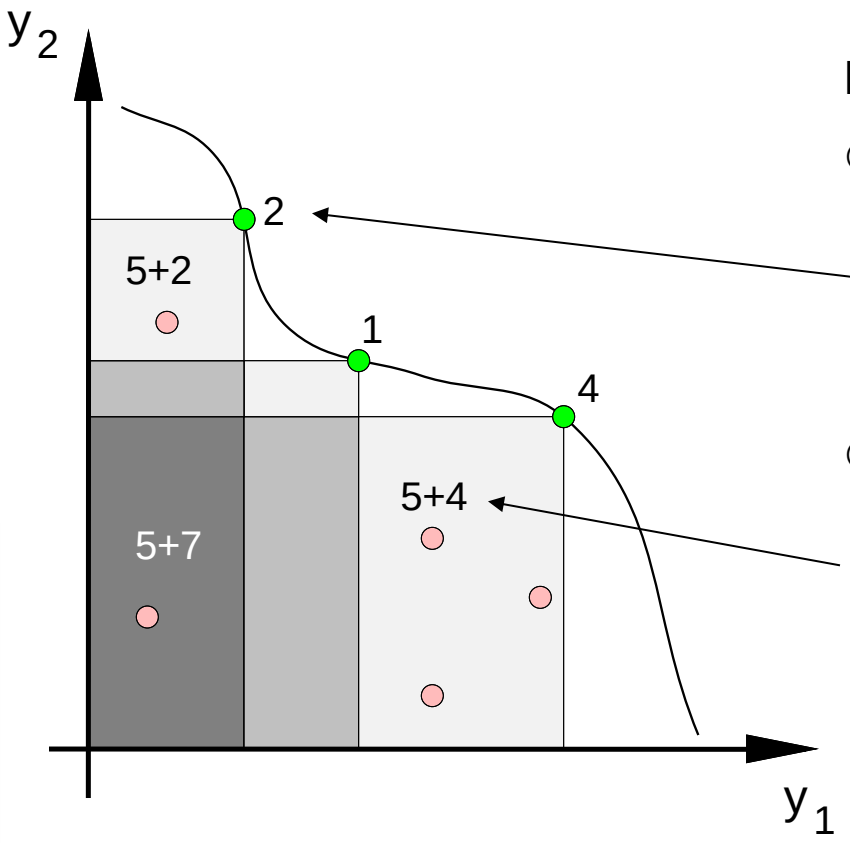
Schwierigkeiten: *großer* Suchraum und *vielfache* Optima

4.4.6 Genetische Algorithmen: Mehrzieloptimierung

- Klassische Einziel-Optimierungsmethoden:
  - Beispiele: Simulated Annealing, ILP, Kernighan Lin usw.
  - Die Entscheidungsfindung (decision making) erfolgt durch Gewichtung der einzelnen Ziele in der Kostenfunktion
    vor der Optimierung.

- Populations-basierte Optimierungsmethoden:
  - Evolutionäre Algorithmen
  - Entscheidungsfindung **nach der Optimierung**

4.4.6 Mehrzieloptimierung: Strength Pareto mit GA‘s (I)



Fitness- Zuweisungsschema:

① Für nicht-dominierte Lösungen:

$$\text{fitness} = \left(\begin{array}{c}\text{Anzahl der}\\ \text{dominierten Lösungen}\end{array}\right)$$

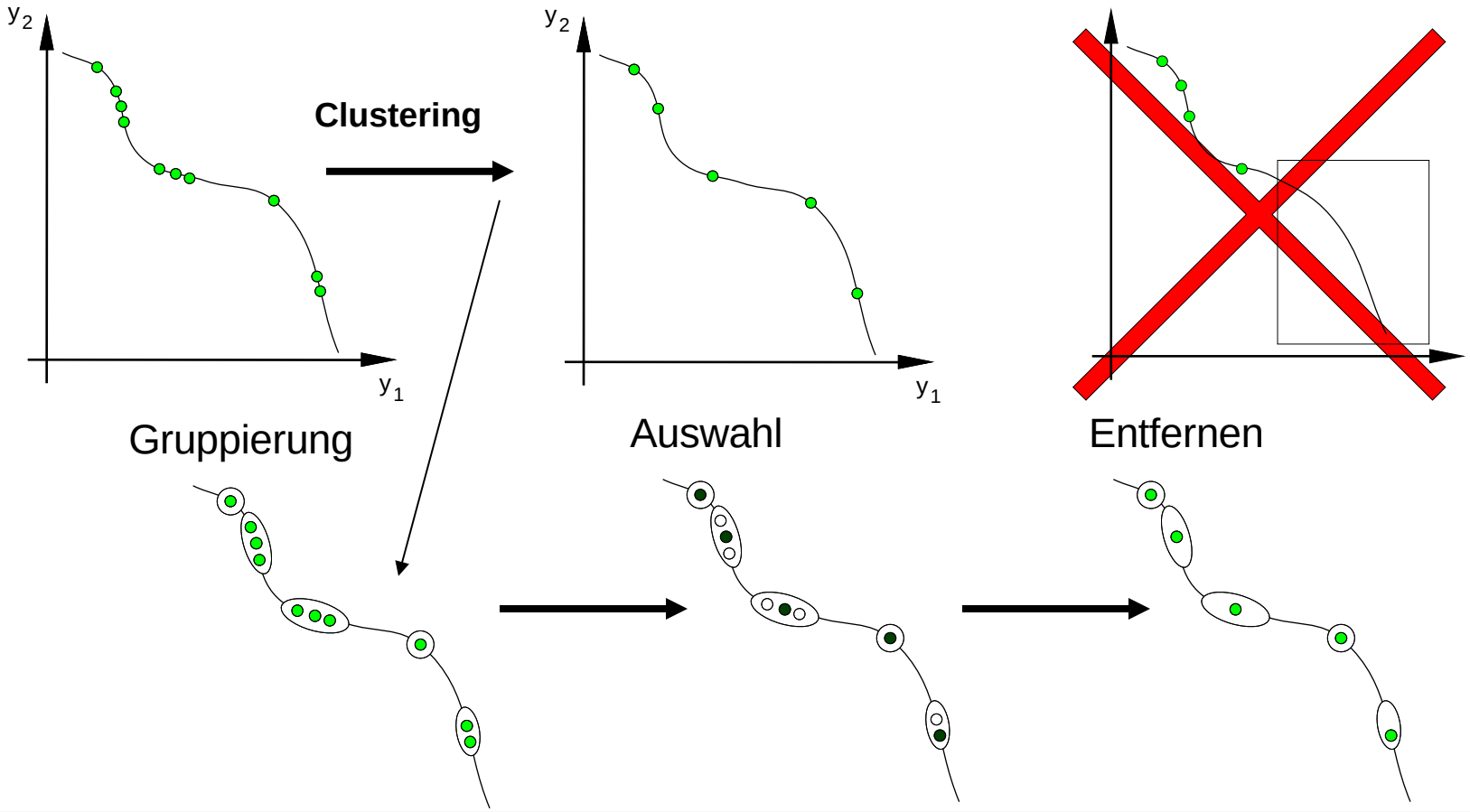
② Für dominierte Lösungen:

$$\text{fitness} = \left(\begin{array}{c}\text{Anzahl der}\\ \text{nicht}-\text{Pareto Lösungen}\end{array}\right) + \sum\left(\begin{array}{c}\text{Fitness der}\\ \text{dominierenden Lösungen}\end{array}\right)$$

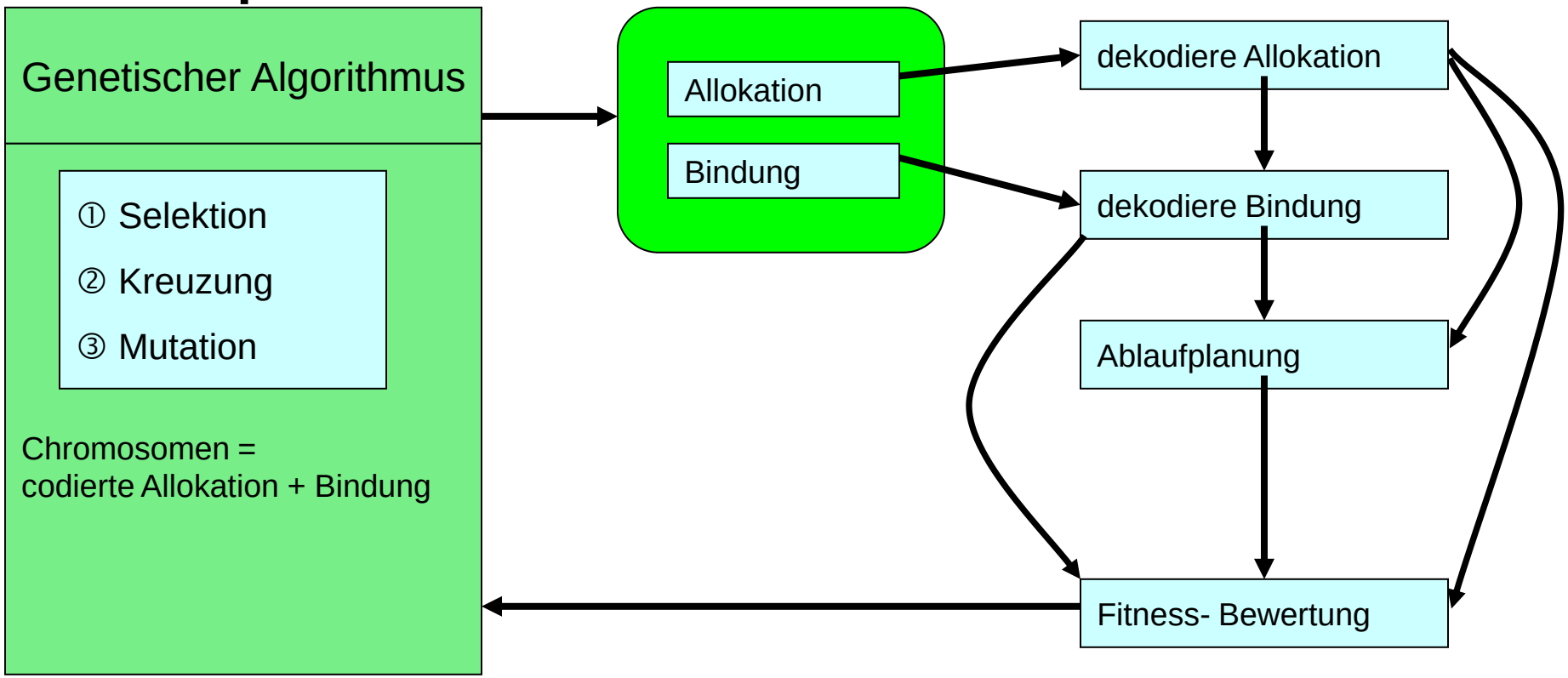
- Heller ist besser als dunkler → Lenkung in Richtung einer Pareto-optimalen Menge
- Weniger ist besser als viele → Trotzdem ist die Diversität des Lösungsraums zu erhalten.

4.4.6 Mehrzieloptimierung: Strength Pareto mit GA‘s (II)

- **Clustering:** Reduziere die Menge der nicht-dominierten Lösungen, ohne dabei die Charakteristik des Ganzen zu zerstören.
  Erhaltung der Diversität zur Vermeidung einer lokal beschränkten Suche mit suboptimalen Ergebnissen.



4.4.6 Exploration des Entwurfsraums mit GAs



- Anwendung eines genetischen Algorithmus auf das Entwurfsraumproblem.
- Die Repräsentation eines Individuums erfolgt mittels eines **Chromosoms**, das in einzelne **Gene unterteilt** ist. Durch die Gene werden die Variablen des Optimierungsproblems dargestellt, für die eine optimale Belegung gefunden werden soll.
- Die Kodierung der Individuen wird so gewählt, daß jedes Individuum, dargestellt durch ein Chromosom, eine **potentielle Lösung** repräsentiert.

- Was sind genetische Algorithmen?
- Wo werden diese eingesetzt?
- Wie gehen diese Algorithmen vor?
- Was ist Mehrzieloptimierung?
- Was ist eine Kostenfunktion?
- Was ist Entwurfsraum-Exploration?

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 Kernighan Lin
  - 4.4.2 Fiduccia Mattheyses
  - 4.4.3 Tabu-Search
  - 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**

4.5 HW/SW Partitionierung

Das Bipartitionierungsproblem als einfachster Fall:

- $P=\{P_{SW}, P_{HW}\}$; mit **O** als Menge aller Objekte.

Rein software-orientierter Ansatz: $P=\{\mathbf{O}, \varnothing\}$

- Alle Funktionen sind zunächst in Software realisiert.
- Die Performanz ist, abhängig von der Funktion, in SW oft unzureichend.
  - Folge: Migration von Objekten aus **O** in die HW- Partition.

Rein hardware-orientierter Ansatz: $P=\{\varnothing, \mathbf{O}\}$

- Die Performanz ist in der Hardwarerealisierung erfüllt.
- Die Kosten und der Implementierungsaufwand sind dann oftmals zu hoch.
  - Folge: Migration von Objekten aus **O** in die SW- Partition.

4.5 HW/SW Partitionierung

Karlsruher Institut für Technologie

Greedy Algorithmen:

- Es werden Objekte in die jeweils andere Partition verschoben, bis keine Verbesserung mehr eintritt.

```
REPEAT {
    P_old = P;
    FOR i = 1 TO n  DO{
        IF (f(Move(P, o_i)) < f(P)) THEN {
            P = Move(P, o_i);
        }
    }
UNTIL (P == P_old)
```

Kostenfunktion

Keine weiteren Änderungen

4.5 HW/SW Partitionierung: HW-orientiertes Greedy-Verfahren

```
PROCEDURE HW ORIENTED GREEDY
    P = {{ }, O}
    REPEAT
        P_old=P;
            FOREACH (o_i ∈ p_HW)
                TryMove(P,o_i);
            ENDFOR
    UNTIL (P == P_old)
END PROCEDURE

PROCEDURE TryMove(P,o_i)
    IF SatisfiesPerformance(Move(P,o_i)) AND f(Move(P,o_i)) < f(P)) DO
        P = Move(P, o_i);
        FOREACH (o_j ∈ Successors(o_i))
            TryMove(P,o_j);
        ENDFOR
    ENDIF
END PROCEDURE
```

Reine HW- Partitionierung für den Anfang.

Sicherung der letzten Partition.

Beginne Rekursion für jedes Objekt aus der HW- Menge.

Wiederholung bis keine Änderung mehr.

Prüfe auf Verbesserung und Erfüllung der Performanceforderungen.

Rekursion

Rekursiver Aufruf für jeden Nachfolger.

Nachteil: Algorithmus kann aus lokalem Minimum nicht mehr entweichen.

- Warum ist Partitionierung wichtig?
- Wie kann Hardware/Software Partitionierung aussehen?
- Welche Typen von Hardware/Software Co-Design gibt es?
- Wo findet HW/SW Co-Design heute in der Wissenschaft oder auch in der Industrie Anwendung?
- Ende… geschafft ;)

Inhalt der Vorlesung (I)

- Einführung und Motivation

- Zielarchitekturen
  - Allgemeiner Aufbau
  - Klassifikation
  - General-Purpose Prozessoren (GPP)
    - Architekturen
      - Akkumulatormaschine, Stackmaschine, Registersatzmaschine
    - Performanzsteigerung
      - Pipelining, Superskalarität / Out-of-Order Execution, Very Large Instruction Word (VLIW), Single Instruction Multiple Data (SIMD), Caches, Multiple Instruction Multiple Data (MIMD)
  - Spezialprozessoren
    - Microcontroller (µC), Digitale Signalprozessoren (DSP), Grafik Prozessoren (GPU)
  - Bus, Multicore & Network-on-Chip (NoC)
  - Anwendungs-spezifische Prozessoren (ASIP)
  - Field Programmable Gate Arrays (FPGA)
  - System-on-Chip (SoC)

Inhalt der Vorlesung (II)

- Abschätzung der Entwurfsqualität
  - Abstraktionsebenen
  - Systemsynthese
  - Graphenmodelle für Kontroll- /Datenfluss
  - Parameter von Schätzverfahren
    - Exaktheit, Treue
  - Schätzung von Hardwaremetriken
    - Taktperiode, Taktschlupf
  - Schätzung von Softwaremetriken
    - MIPS, FLOPS, WCET, Cachekonfliktgraph, Profiling, Tracing

Inhalt der Vorlesung (III)

- Hardware/Software Partitionierungsverfahren
  - Einführung, Partitionierungsansätze, Komplexität
  - Klassifikation von Partitionierungsalgorithmen
  - Konstruktive Algorithmen
    - Clustering-Verfahren
  - Iterative Algorithmen
    - Kernighan Lin (KL), Fiduccia Mattheyses (FM), Tabu-Search (TS), Simulated Annealing (SA), Genetische Algorithmen (GA)
  - Hw/Sw Partitionierungsverfahren und Co-Design-Systeme