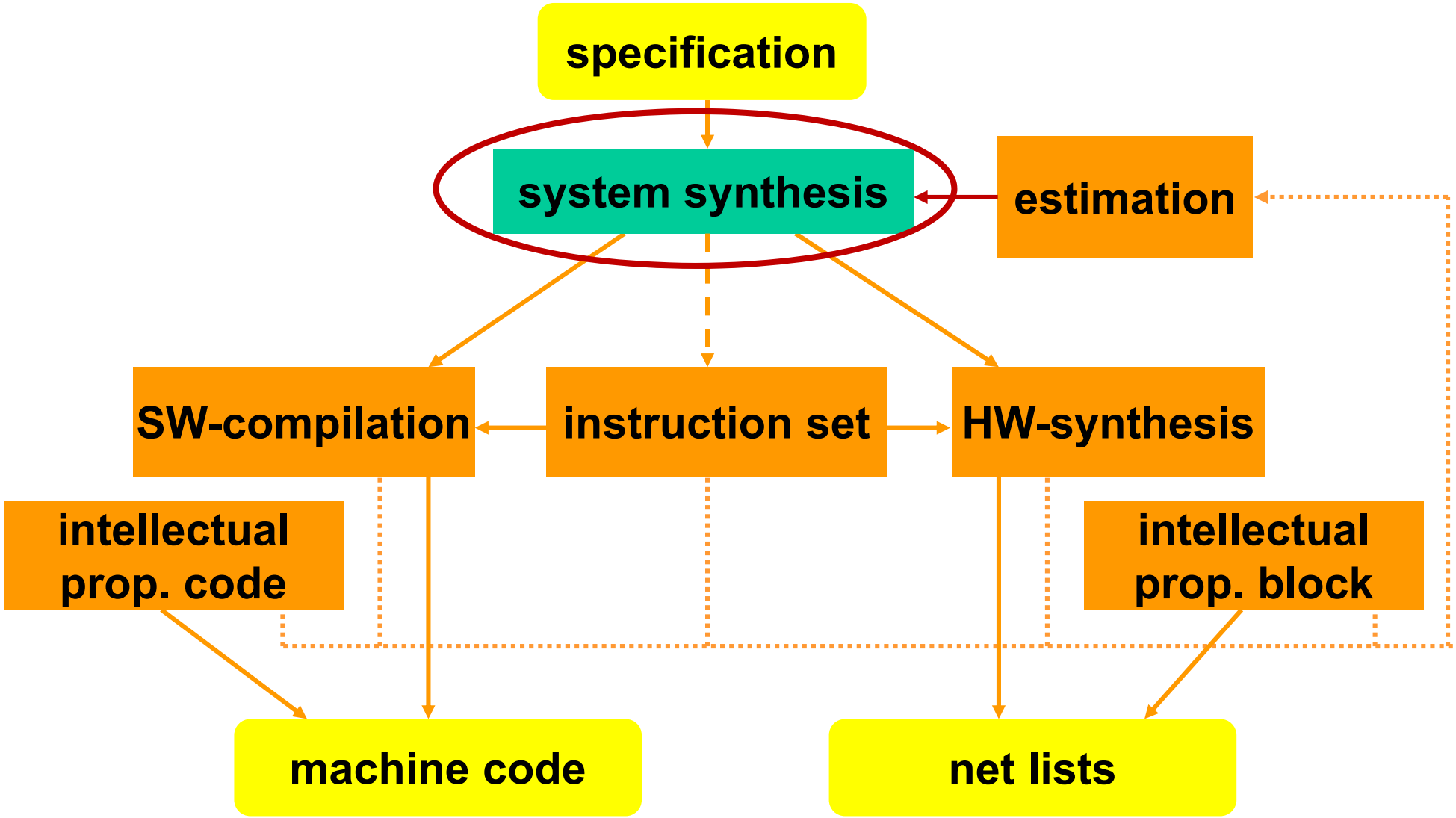
Hardware-Software Codesign

4. System Partitioning

Lothar Thiele

System Design



Mapping

Mapping transforms behavior into structure and execution:

- **allocation**: select components
- **binding**: assign functions to components
- **scheduling**: determine execution order

partitioning

mapping

- … finally, synthesis results into **implementation**

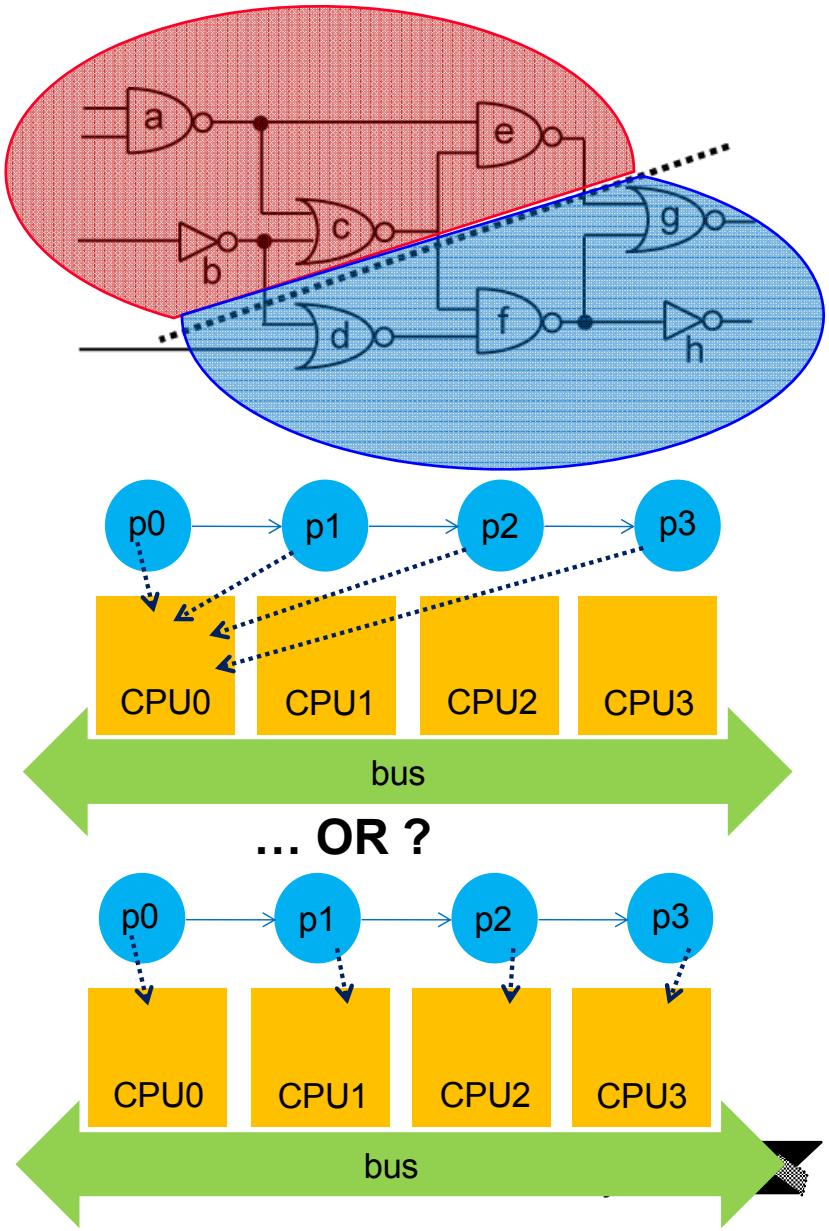
Levels of Abstractions

Mapping can be done

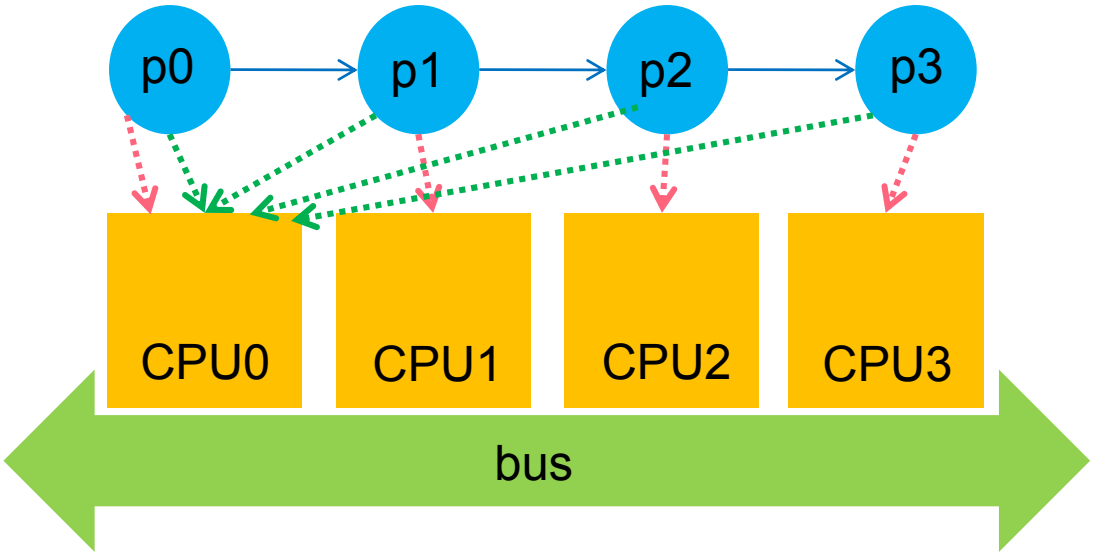
- at low level: register transfer level (RTL) or netlist level
  - e.g., split a digital circuit and map it to several devices (FPGAs, ASICs)
  - system parameters (e.g., area, delay) relatively easy to determine

- at high level: system level
  - comparison of design alternatives for optimality (design space exploration)
  - system parameters are unknown and difficult to determine

    → to be estimated via analysis, simulation, (rapid) prototyping



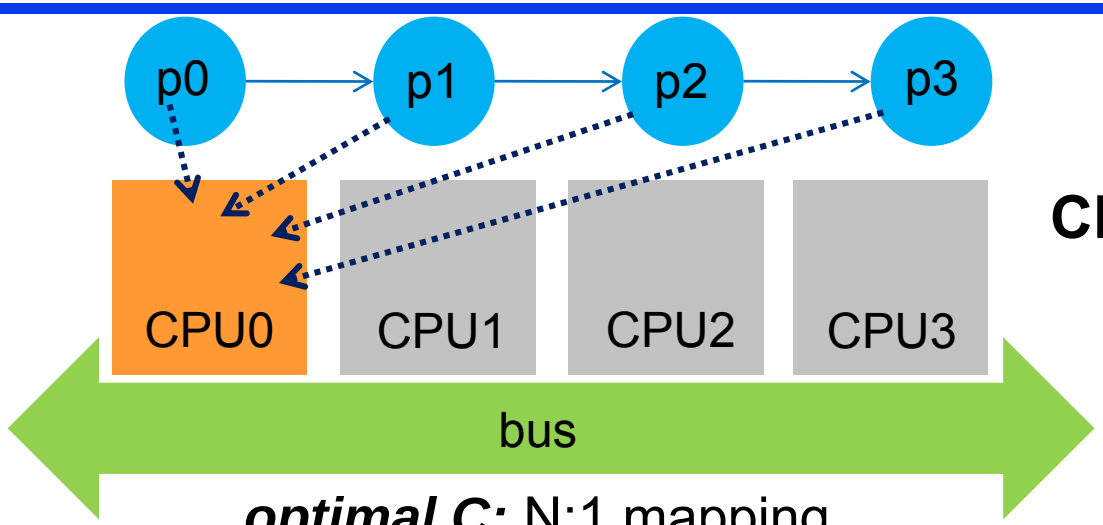
Model-Based Synthesis – Example

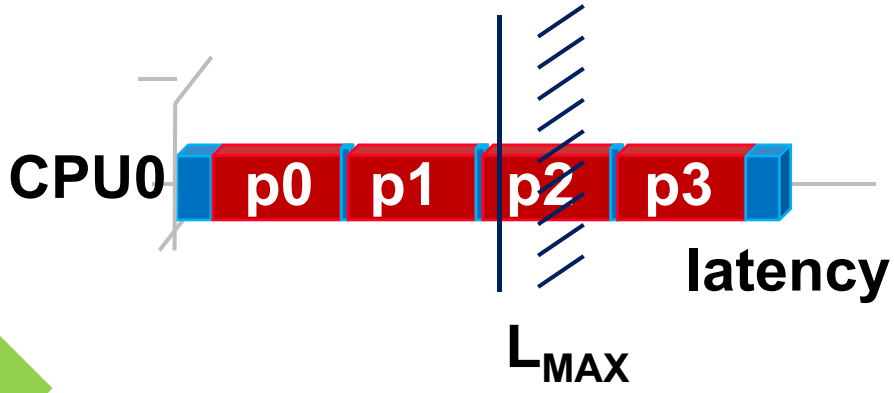


optimal C: N:1 mapping
optimal L: 1:1 mapping

- **considered performance**
  - **cost C:** cost of allocated components, e.g., sum
  - **latency L:** due to scheduling (resource sharing)
- **conflicting design goals**
  - feasible schedule **L ≤ Lmax**
  - feasible allocation **C ≤ Cmax**

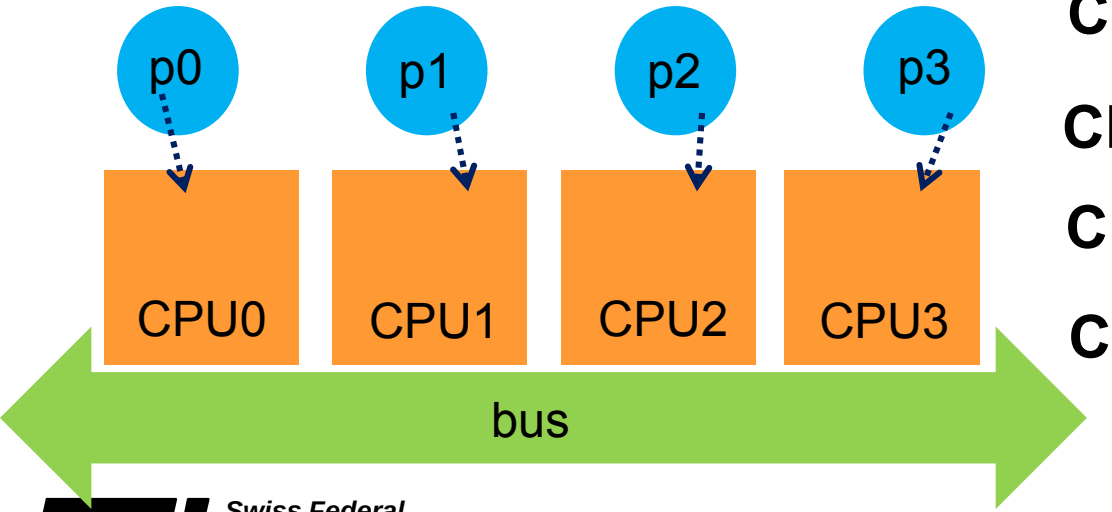
Example – Alternatives

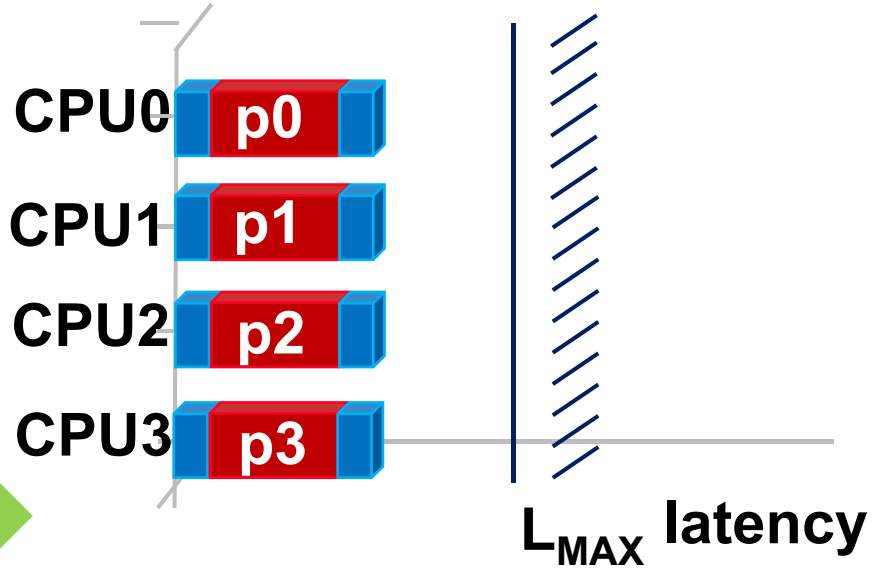




optimal C: N:1 mapping

optimal L: 1:1 mapping





Cost Functions

Quantitatively measure performance of a design point

- system cost **C[$]**
- latency **L[sec]**
- power consumption **P[W]**
- ...

Estimation is required to find **C, L, P** values, for each design point

- *example:* linear cost (preference) function with penalty

$$f(C,L,P) = k_1 \cdot h_C(C, C_{max}) + k_2 \cdot h_L(L, L_{max}) + k_3 \cdot h_P(P, P_{max})$$

- h_C, h_L, h_P … denote how strong **C**, **L**, **P** violate design constraints C_{max}, L_{max}, P_{max}
- k_1, k_2, k_3 … weighting and normalization

The Formal Partitioning Problem

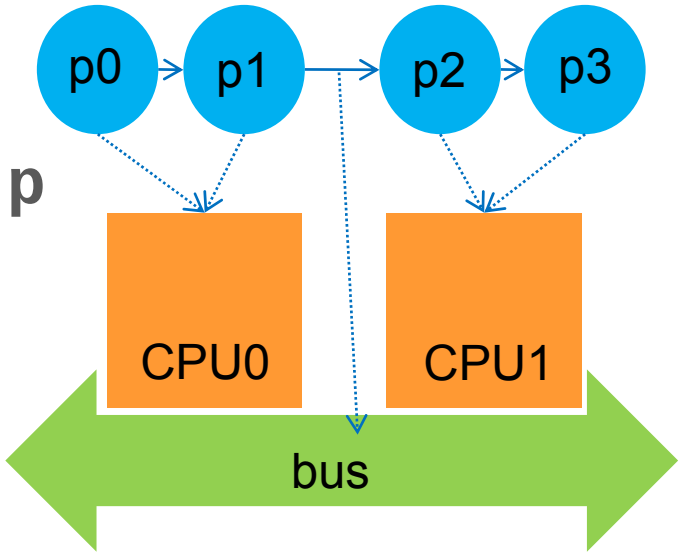
assign **n** objects $O=\{o_1,...,o_n\}$ to **m** blocks (also called partitions) $P=\{p_1,...,p_m\}$, such that

- $p_1 \cup p_2 \cup ... \cup p_m = O$ (all objects are assigned –mapped)
- $p_i \cap p_j = \{\} \ \forall i,j: i \neq j$ (an object is not assigned or "mapped" twice)
- and costs $c(P)$ are minimized

objects **O**

partitions **p**

blocks **m**



note: in ***system synthesis*** (simple model)

- objects = process network graph nodes
- blocks = architecture graph nodes
- cost = measured/estimated with dedicated cost functions (e.g., latency, power, hardware cost)

Partitioning Methods

- **Exact methods**
  - enumeration
  - ***integer linear programs (ILP) (→see next slides)***

- **Heuristic methods**
  - constructive methods
    - random mapping
    - hierarchical clustering
  - iterative methods
    - Kernighan-Lin algorithm
    - simulated annealing
    - evolutionary algorithms

Integer Programming Model

Ingredients:

- objective function (cost)
- constraints

} involving linear expressions of integer variables from a set **X**

objective $$C = \sum_{x_i \in X} a_i x_i \text{ with } a_i \in R, x_i \in \mathbb{N} \quad (1)$$

constraints $$\forall j \in J : \sum_{x_i \in X} b_{i,j} x_i \geq c_j \text{ with } b_{i,j}, c_j \in \mathbb{R} \quad (2)$$

Integer programming (IP) problem:

minimize objective function (1) subject to constraints (2)

note: if all x_i are constrained to be either `0` or `1`, the IP problem is said to be a 0/1 integer programming problem

Small Example of 0/1 IP

minimize: $C = 5x_1 + 6x_2 + 4x_3$

subject to: $x_1 + x_2 + x_3 \geq 2$

$x_1, x_2, x_3 \in \{0,1\}$

x_1	x_2	x_3	C	
0	1	1	10	
1	0	1	9	← optimal (minimal)
1	1	0	11	
1	1	1	15	

Integer Linear Program for Partitioning

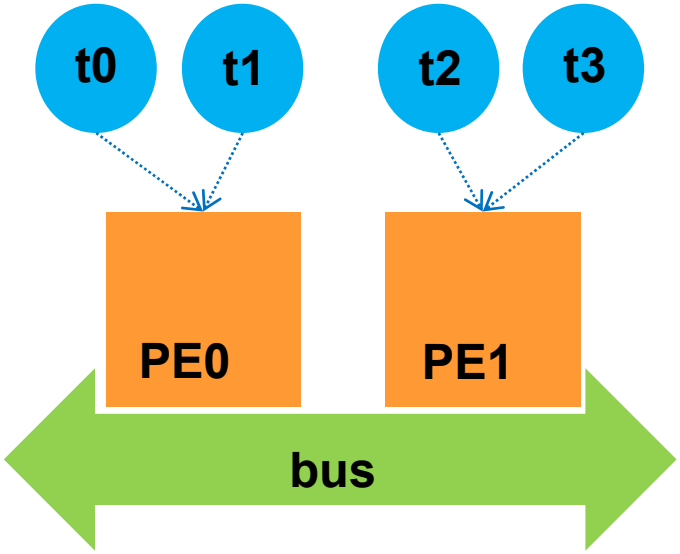
- Binary variables $x_{i,k}$
  - $x_{i,k}$ = 1: object o_i in block p_k
  - $x_{i,k}$ = 0: object o_i not in block p_k
- Cost $c_{i,k}$, if object o_i is in block p_k
- Integer linear program:

$$\boldsymbol{x_{i,k}} \in \{0,1\} \quad 1 \le \boldsymbol{i} \le \boldsymbol{n}, 1 \le \boldsymbol{k} \le \boldsymbol{m}$$

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

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

Example – Partitioning



task / PE	t0	t1	t2	t3
PE0	1	1	0	0
PE1	0	0	1	1

e.g., optimized for a load balanced system

exe. time	t0	t1	t2	t3
PE0	5	15	10	30
PE1	10	20	10	10

load balancing:

$\mathbf{load_{PE0} = 5+15}$

$\mathbf{load_{PE1} = 10+10}$

Variations in ILP

Additional constraints:

- e.g., maximum h_k objects in block k

$$\sum_{i=1}^{n} x_{i,k} \leq h_k \quad 1 \leq k \leq m$$

Maximizing the cost function:

- can be done by setting C'= -C in a minimization problem

ILP for synthesis

Solving the synthesis problem with ILP is very popular:

- If not solving to optimality, runtimes are acceptable and a solution with guaranteed quality can be determined.
- Scheduling can be integrated.
- Various additional constraints can be added.
- *However, finding the right equations to model the constraints is an art.*

Remarks on Integer Programming

Integer programming is NP-complete

- In practice, *runtimes can increase exponentially* with the size of the problem.
- But problems of some thousands of variables can still be solved with *commercial solvers* (depending on the size/structure of the problem) or *approximation algorithms (heuristics).*
- IP models can be a *good starting point* for designing heuristic optimization methods.

Partitioning Methods

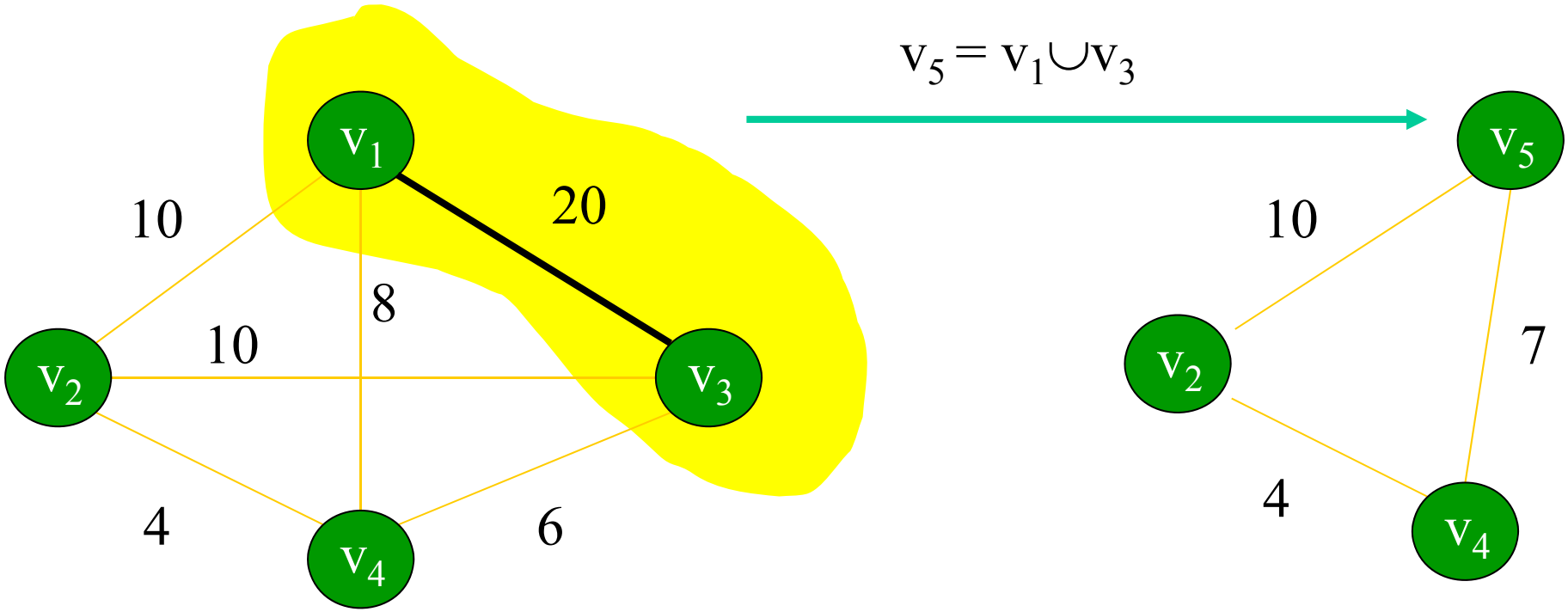
- **exact methods**
  - enumeration
  - integer linear programs (ILP)

- **heuristic methods**
  - ***constructive methods (→see next slides)***
    - random mapping
    - hierarchical clustering
  - iterative methods
    - Kernighan-Lin algorithm
    - simulated annealing
    - evolutionary algorithms

Constructive Methods

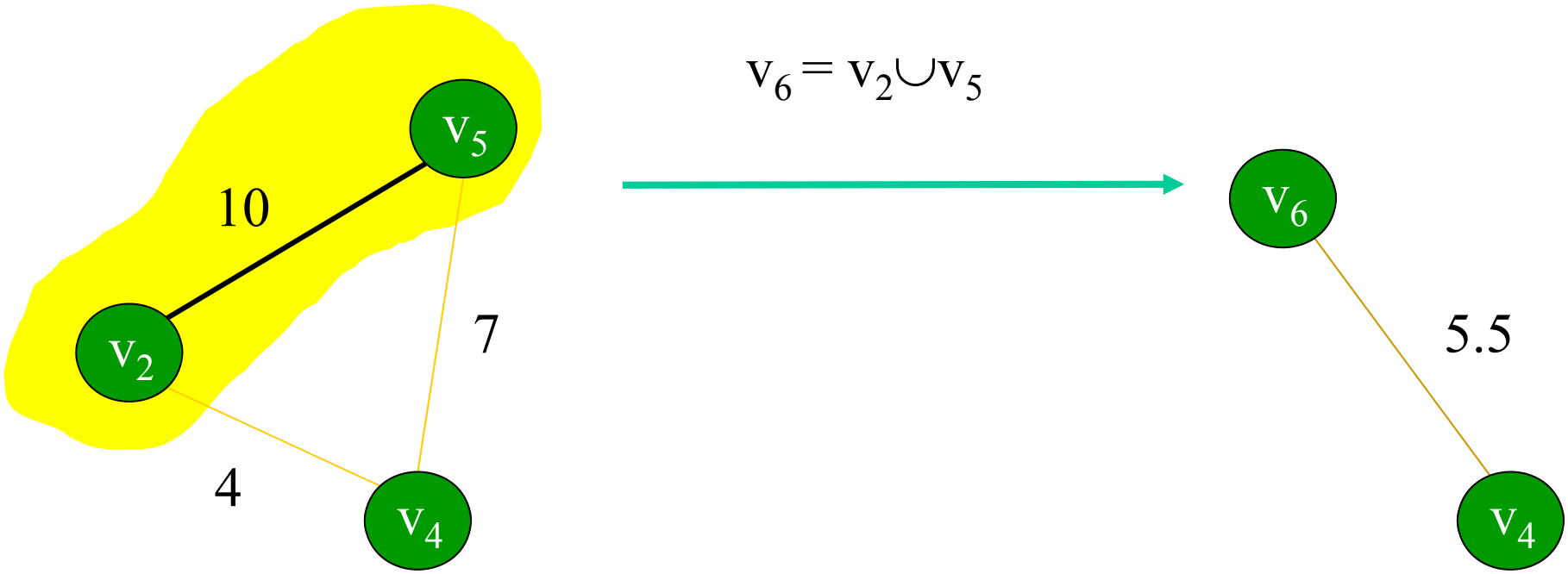
- ***Examples***
  - random mapping
    - each object is assigned to a block randomly
  - hierarchical clustering
    - stepwise grouping of (e.g., two) objects
    - and evaluate closeness function (how desirable it is to group objects)
- Constructive methods are often used to generate a starting partition for iterative methods

Hierarchical Clustering Example (1)

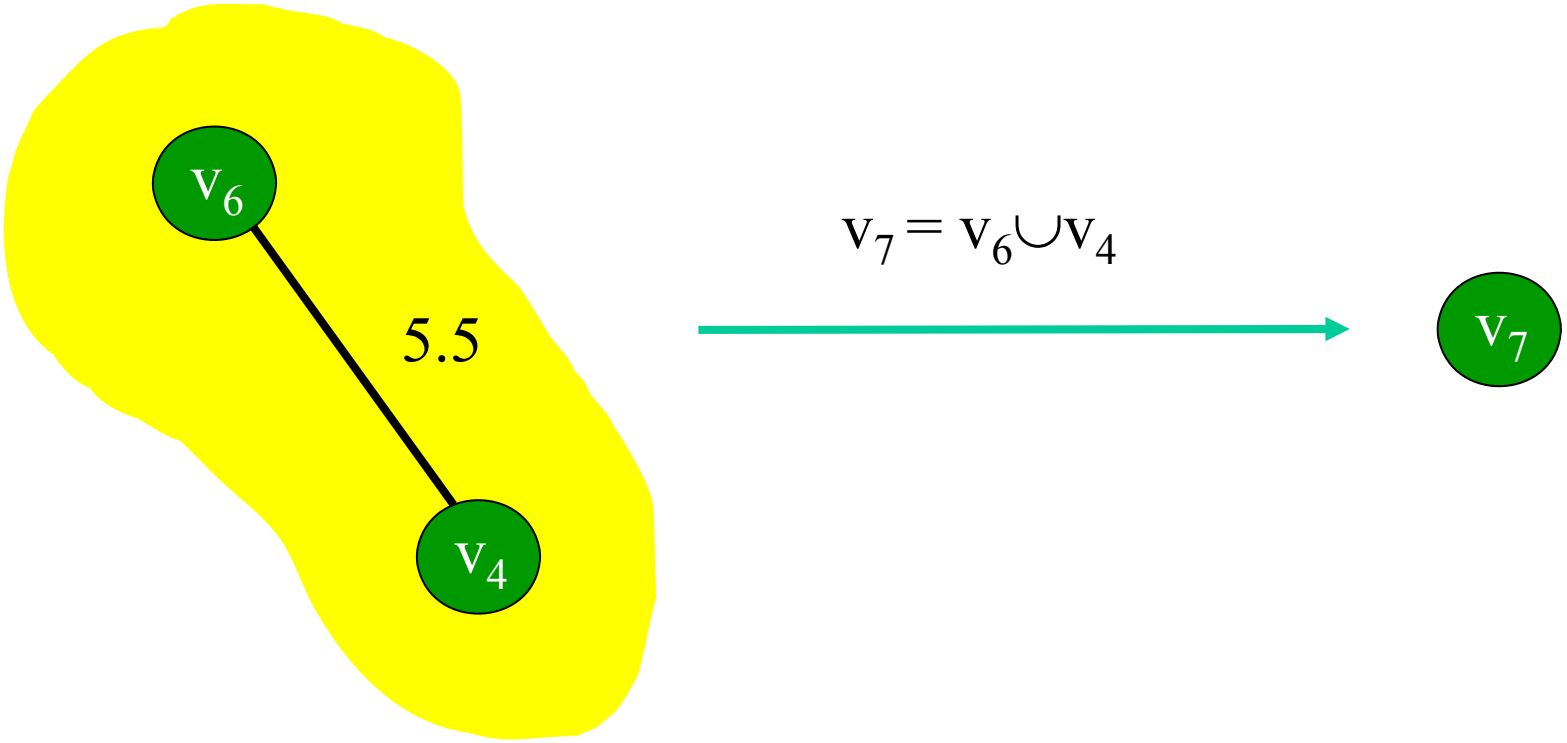


closeness function: arithmetic mean of weights

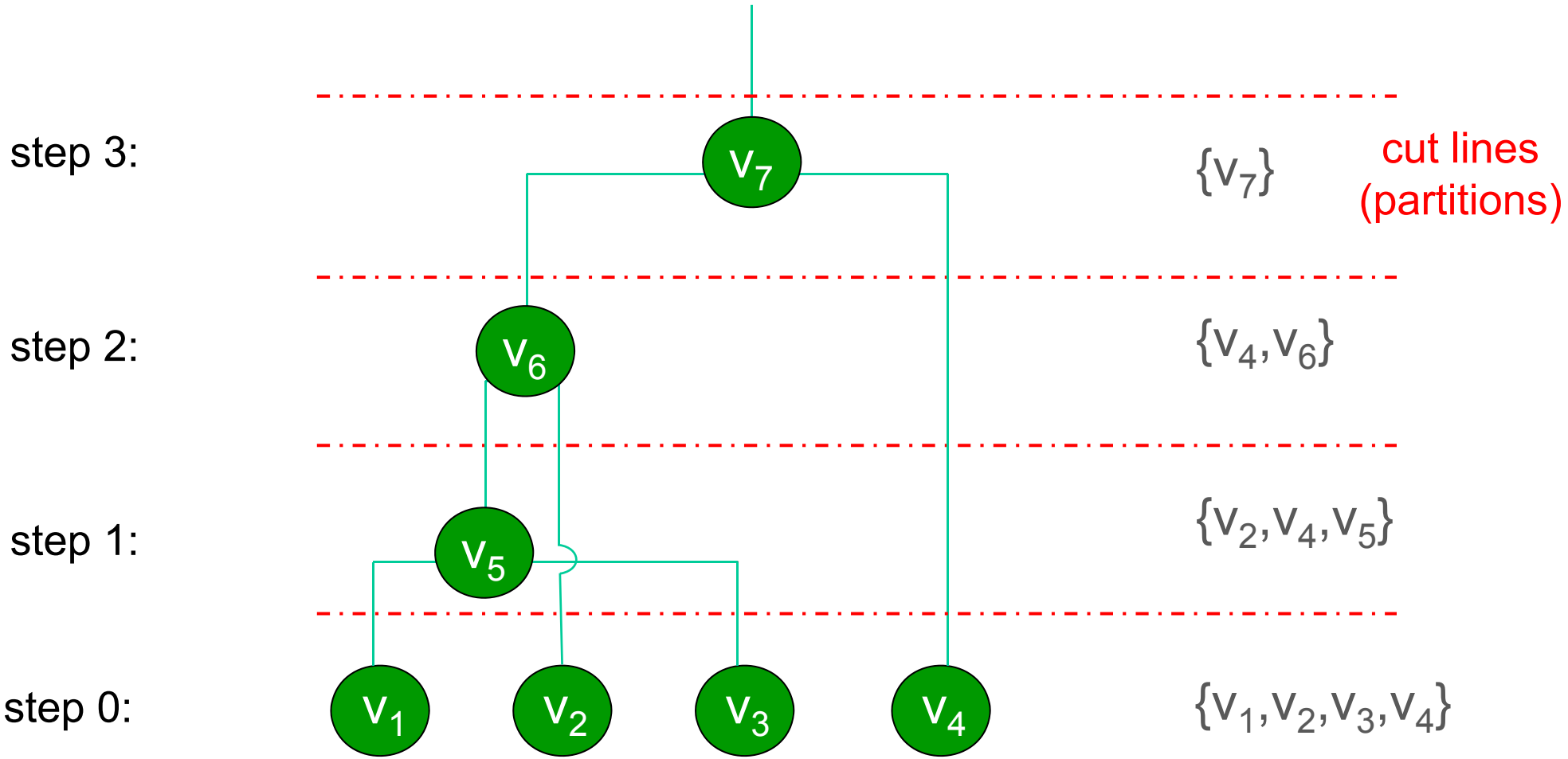
Hierarchical Clustering Example (2)



Hierarchical Clustering Example (3)



Hierarchical Clustering – Summary



Partitioning Methods

- **exact methods**
  - enumeration
  - integer linear programs (ILP)

- **heuristic methods**
  - constructive methods
    - random mapping
    - hierarchical clustering
  - **iterative methods** ***(→see next slides)***
    - Kernighan-Lin algorithm
    - simulated annealing
    - evolutionary algorithms

Iterative Methods (1)

Often used principle for iterative methods:

- start with some initial configuration (partitioning)
- search *neighborhood* (similar partitions) and *select a neighbor* as candidate
- evaluate *fitness (cost) function of candidate*
  - accept candidate using acceptance rule
  - if not, select another neighbor
- stop if quality is sufficiently high, if no improvement can be found, or after some fixed time

Ingrediences:

- initial configuration, function to find a *neighbor* as next candidate, cost function, acceptance rule, stop criterion

Iterative Methods (2)

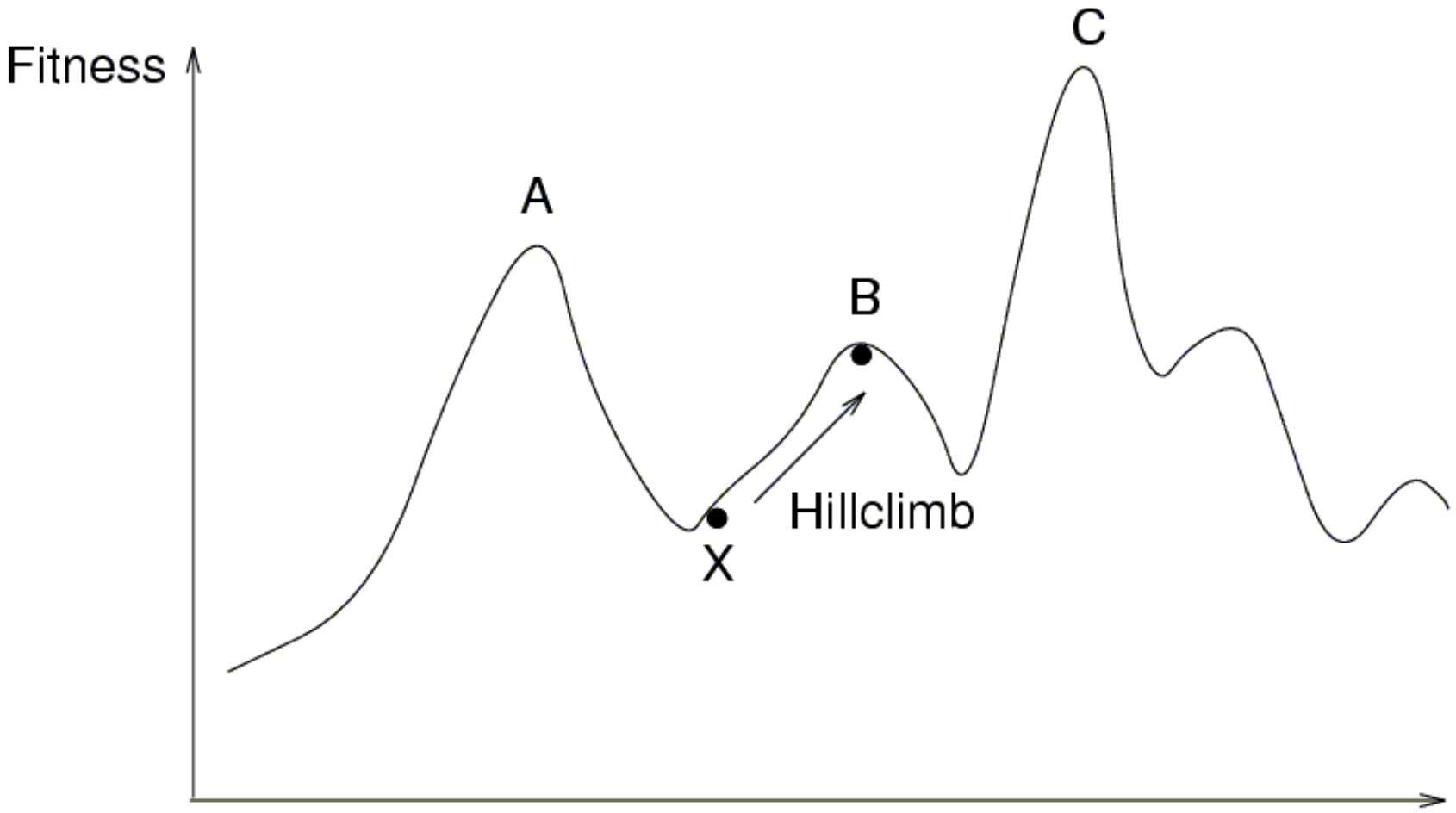
Simple iterative improvement or *"hill climbing":*

- candidate is always and only accepted if cost is lower (or fitness is higher) than current configuration
- stop when no neighbor with lower cost (higher fitness) can be found

Disadvantages:

- local optimum as best result
- local optimum depends on initial configuration
- generally no upper bound on iteration length

Iterative Methods – Illustration



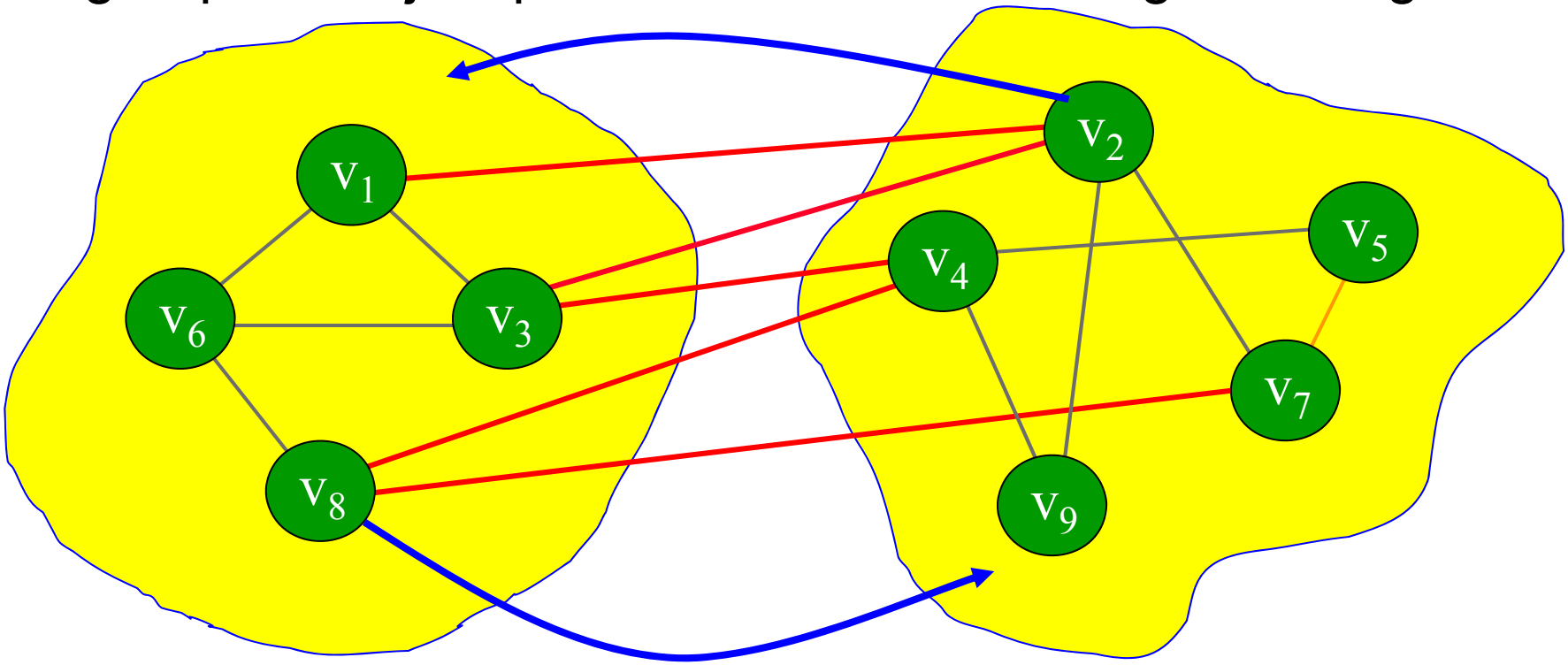
How to Cope with Disadvantages?

- Repeat algorithm many times with different initial configurations
- Use information gathered in previous runs
- Use a more complex “acceptance rule” to jump out of local optimum
- Use a more complex strategy that accepts sometimes randomly generated solutions

Iterative Methods – Simple Greedy Heuristic

- Iterate until no improvement in cost:

  re-group the object pairs that leads to the largest cost gain



example: cost = number of edges crossing the partitions

before re-group: 5 ; after re-group: 4 ; gain = 1

Iterative Methods – Kernighan-Lin

Improved algorithm: Kernighan-Lin:

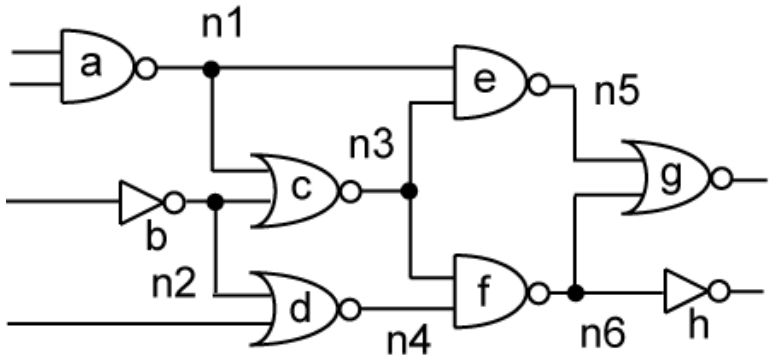
- as long as a better partition is found
  - from all possible pairs of objects

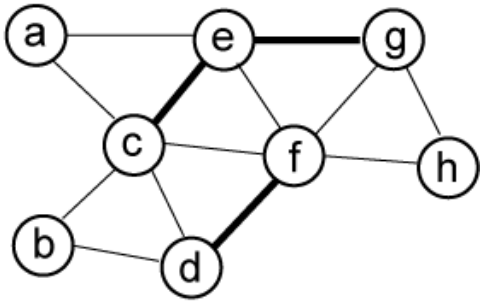
    → *virtually* re-group the “best” (lowest cost of resulting partition)
  - from the remaining (not yet touched) objects

    → *virtually* re-group the “best” pair
  - continue until all objects have been re-grouped
  - from these *n/2* partitions, take the one with smallest cost and *actually* perform the corresponding re-group operations

Illustration of KL Algorithm (1)

Example: partitioning of digital circuit





communication cost
from node x to node y

cost matrix c(x,y)

c(x,y)	a	b	c	d	e	f	g	h
a	0	0	.5	0	.5	0	0	0
b	0	0	.5	.5	0	0	0	0
c	.5	.5	0	.5	1	.5	0	0
d	0	.5	.5	0	0	1	0	0
e	.5	0	1	0	0	.5	1	0
f	0	0	.5	1	.5	0	.5	.5
g	0	0	0	0	1	.5	0	.5
h	0	0	0	0	0	.5	.5	0

Illustration of KL Algorithm (2)

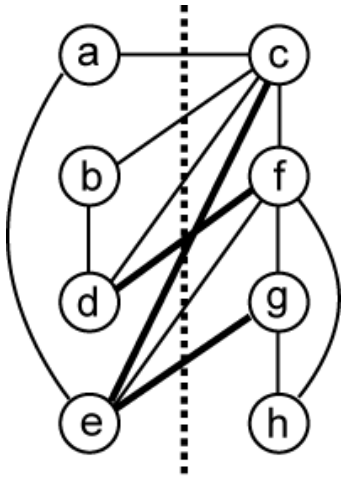
first re-group

pair	$E_x - I_x$	$E_y - I_y$	$c(x,y)$	gain
(a,c)	$0.5 - 0.5$	$2.5 - 0.5$	0.5	1
(a,f)	$0.5 - 0.5$	$1.5 - 1.5$	0	0
(a,g)	$0.5 - 0.5$	$1 - 1$	0	0
(a,h)	$0.5 - 0.5$	$0 - 1$	0	-1
(b,c)	$0.5 - 0.5$	$2.5 - 0.5$	0.5	1
(b,f)	$0.5 - 0.5$	$1.5 - 1.5$	0	0
(b,g)	$0.5 - 0.5$	$1 - 1$	0	0
(b,h)	$0.5 - 0.5$	$0 - 1$	0	-1
$\mathbf{(d,c)}$	$\mathbf{1.5 - 0.5}$	$\mathbf{2.5 - 0.5}$	**0.5**	**2**
(d,f)	$1.5 - 0.5$	$1.5 - 1.5$	1	-1
(d,g)	$1.5 - 0.5$	$1 - 1$	0	1
(d,h)	$1.5 - 0.5$	$0 - 1$	0	0
(e,c)	$2.5 - 0.5$	$2.5 - 0.5$	1	2
(e,f)	$2.5 - 0.5$	$1.5 - 1.5$	0.5	1
(e,g)	$2.5 - 0.5$	$1 - 1$	1	0
(e,h)	$2.5 - 0.5$	$0 - 1$	0	1

some definitions

- E_i = external costs of vertex **i**
- I_i = internal costs of vertex **i**

- $D_i = E_i - I_i$ = desirability to move a vertex (**x** or **y**)
- **gain** $= D_x + D_y - 2 \cdot c(x,y)$ = gain due to change in cut costs



initial partitioning

Illustration of KL Algorithm (3)

second re-group

pair	$E_x - I_x$	$E_y - I_y$	$c(x,y)$	gain
(a,f)	$0-1$	$1-2$	0	-2
(a,g)	$0-1$	$1-1$	0	-1
(a,h)	$0-1$	$0-1$	0	-2
(b,f)	$0.5-0.5$	$1-2$	0	-1
$\mathbf{(b,g)}$	$\mathbf{0.5-0.5}$	$\mathbf{1-1}$	**0**	**0**
(b,h)	$0.5-0.5$	$0-1$	0	-1
(e,f)	$1.5-1.5$	$1-2$	0.5	-2
(e,g)	$1.5-1.5$	$1-1$	1	-2
(e,h)	$1.5-1.5$	$0-1$	0	-1

some definitions

- E_i = external costs of vertex **i**
- I_i = internal costs of vertex **i**

- $D_i = E_i - I_i$ = desirability to move a vertex (**x** or **y**)
- **gain** $= D_x + D_y - 2 \cdot c(x,y) =$ gain due to change in cut costs

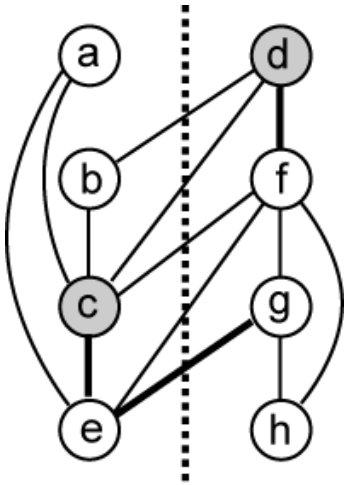


Illustration of KL Algorithm (4)

third re-group

pair	$E_x - I_x$	$E_y - I_y$	$c(x,y)$	gain
(a,f)	$\mathbf{0-1}$	$\mathbf{1.5-1.5}$	$\mathbf{0}$	**-1**
(a,h)	$0-1$	$0.5-0.5$	0	-1
(e,f)	$0.5-2.5$	$1.5-1.5$	0.5	-3
(e,h)	$0.5-2.5$	$0.5-0.5$	0	-2

some definitions

- E_i = external costs of vertex **i**
- I_i = internal costs of vertex **i**

- $D_i = E_i - I_i$ = desirability to move a vertex (**x** or **y**)
- $gain = D_x + D_y - 2*c(x,y)$ = gain due to change in cut costs

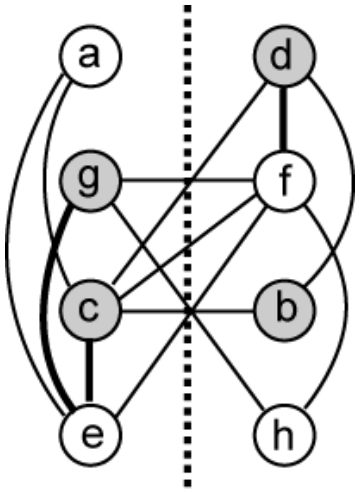


Illustration of KL Algorithm (5)

… and final re-group

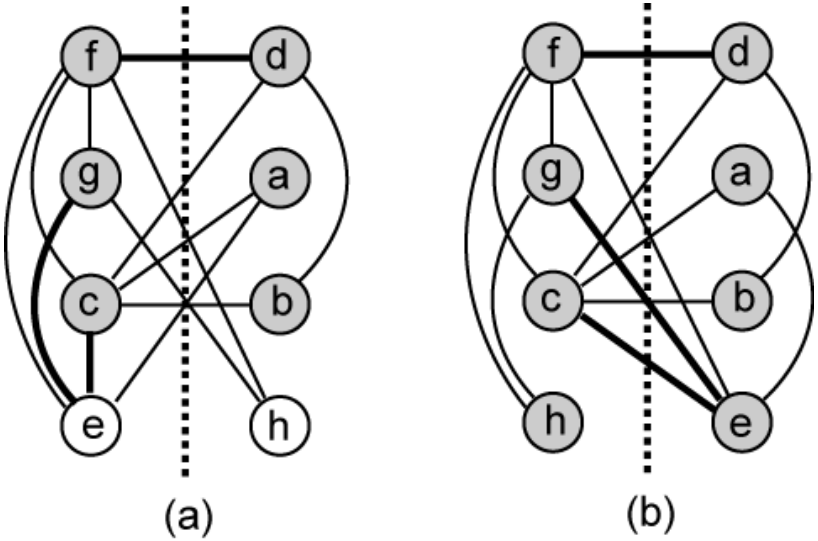
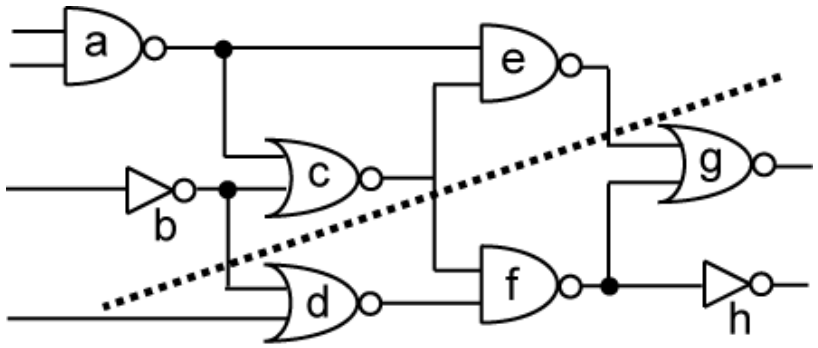


Illustration of KL Algorithm (6)

- Two best solutions found:

i	pair	$gain(i)$	$\sum gain(i)$	cutsize
0	-	-	-	5
1	(d, c)	**2**	**2**	**3**
2	(b, g)	**0**	**2**	**3**
3	(a, f)	-1	1	4
4	(e, h)	-1	0	5



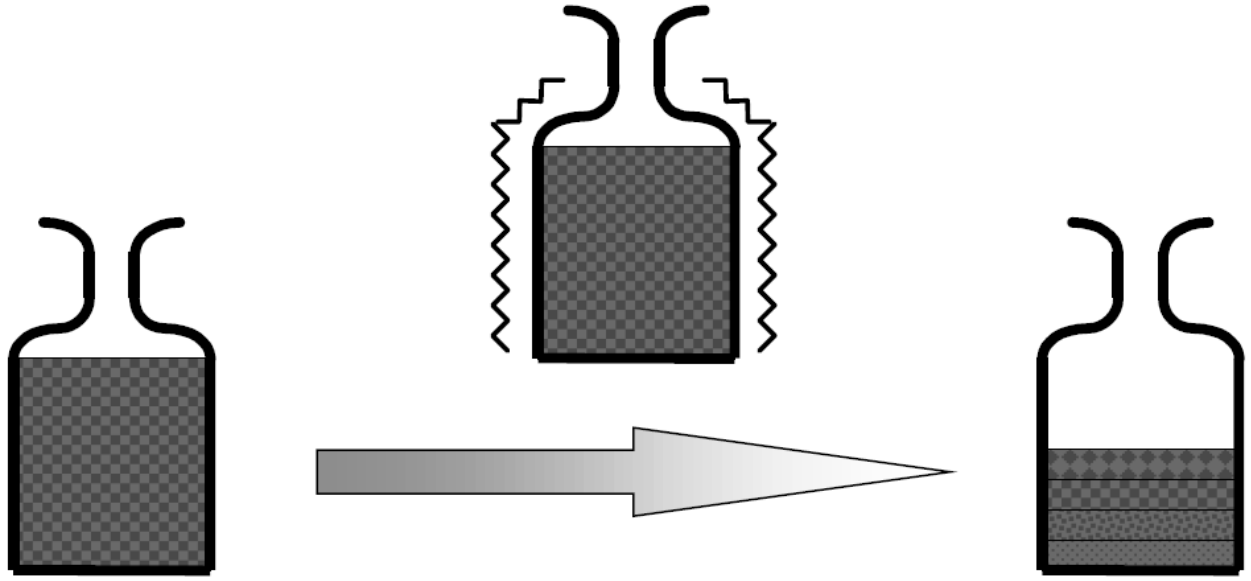
- Start from one of these solutions the whole process again … .

Simulated Annealing – Underlying Philosophy

- Inspired from the physical process of annealing (from metallurgy),

  where a "structured" lattice structure of a solid is achieved by

  1. *heating up* the solid to its melting point
  2. … and then *slowly cooling down* until it solidifies to a low-energy state



Simulated Annealing – Underlying Philosophy (2)

- Solids take on a **minimal-energy state** during cooling down
  if the temperature is decreased sufficiently slowly

- There is a non-zero probability that a particle "jumps" to a higher-energy state ($\mathbf{e_{i+1} > e_i}$):

$$P(e_i, e_{i+1}, T) = e^{\frac{e_i - e_{i+1}}{k_B T}}$$

k_B = Boltzmann constant

T = temperature

e_i = current energy state

e_{i+1} = next energy state

Simulated Annealing Applications

Application to combinatorial optimization:

- energy = cost of a solution (partition)
- cost decreases with temperature (a global parameter)
- increases in cost are accepted with a certain *probability* (that depends both on the *difference between cost values* and also on "*temperature*")

Simulated Annealing Algorithm

By analogy with the physical process:

- replace existing solutions by (randomly generated) new feasible solutions from a neighborhood
- improve a solution by always accepting better-cost neighbors (if selected) but allow for a (*stochastically*) guided acceptance of worse-cost neighbors
- gradual cooling: gradually decrease the probability of accepting worse-cost solutions
  - selecting solutions is almost random when **T** is large
  - … but increasingly selects the better cost solution as **T** goes to **`zero`**

Advantage

- allowance for “uphill” moves potentially avoids local optima

Simulated Annealing – Possible Coding

```
temp = temp_start;
cost = c(P);
while (Frozen() == FALSE) {
    while (Equilibrium() == FALSE) {
       P' = RandomMove(P);
       cost' = c(P');
       deltacost = cost' - cost;
        if (Accept(deltacost, temp) > random[0,1)) {
           P = P';
           cost = cost';
       }
    }
   temp = DecreaseTemp(temp);
}
```

initial solution

$$\text{Accept}(deltacost, temp) = e^{-\frac{deltacost}{k \cdot temp}}$$

Simulated Annealing – Possible Coding (contn.)

- **`RandomMove(P)`**
  - choose a random solution in the neighborhood of P
- **`DecreaseTemp(), Frozen()`**
  - cooling down; there are many different choices, for example:
    - initially: **`temp:=1.0;`**
      in any iteration: **`temp := α*temp`** (typ.: **$0.8 \leq \alpha \leq 0.99$**)
  - frozen after a certain time or if there is no further improvement
- **`Equilibrium()`**
  - usually after a defined number of iterations
- ***Complexity***
  - from exponential to constant, depending on the choice of the functions **`Equilibrium(), DecreaseTemp(),`** and **`Frozen()`**