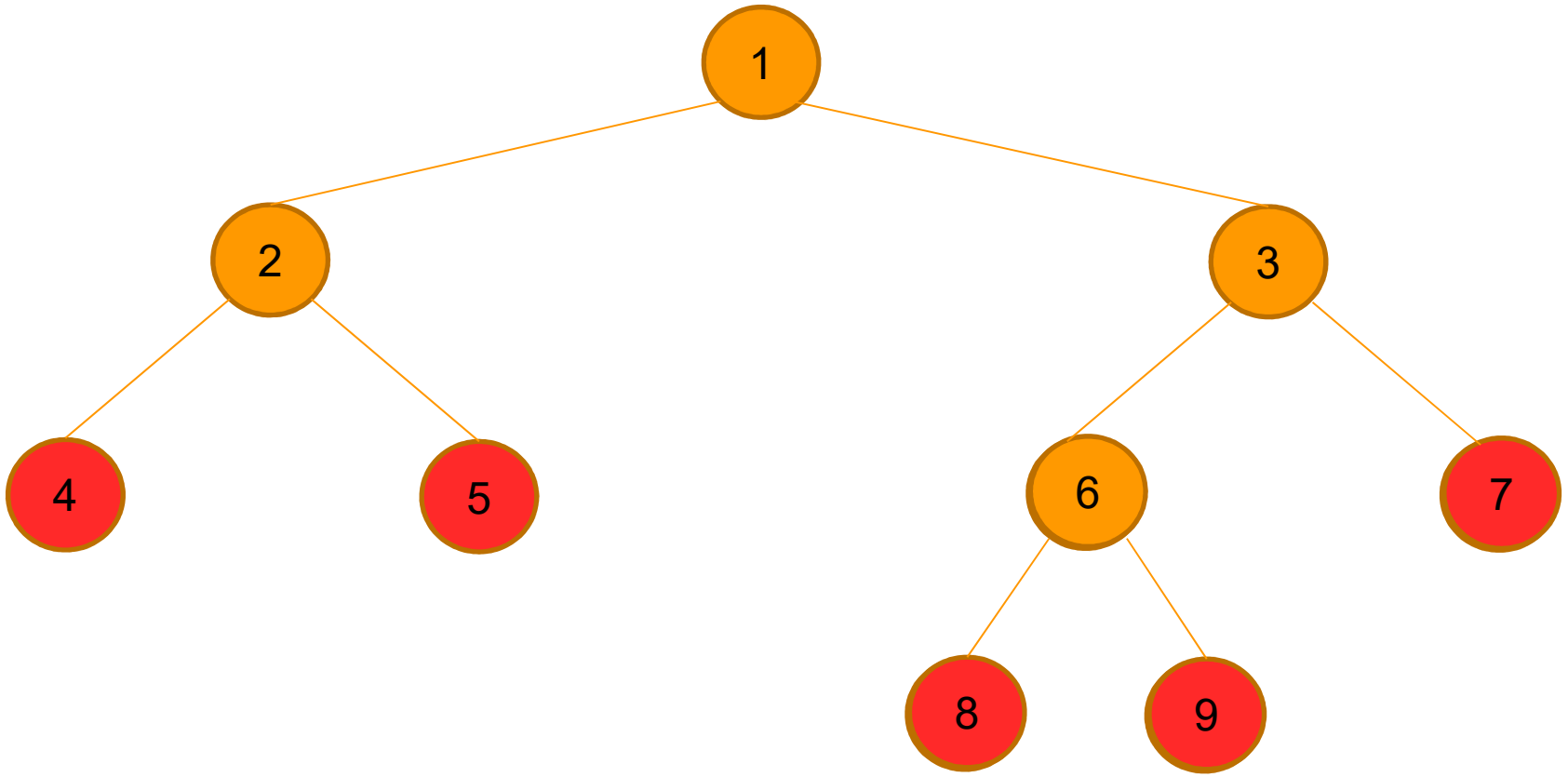
3 Branch&Bound approaches (B&B)

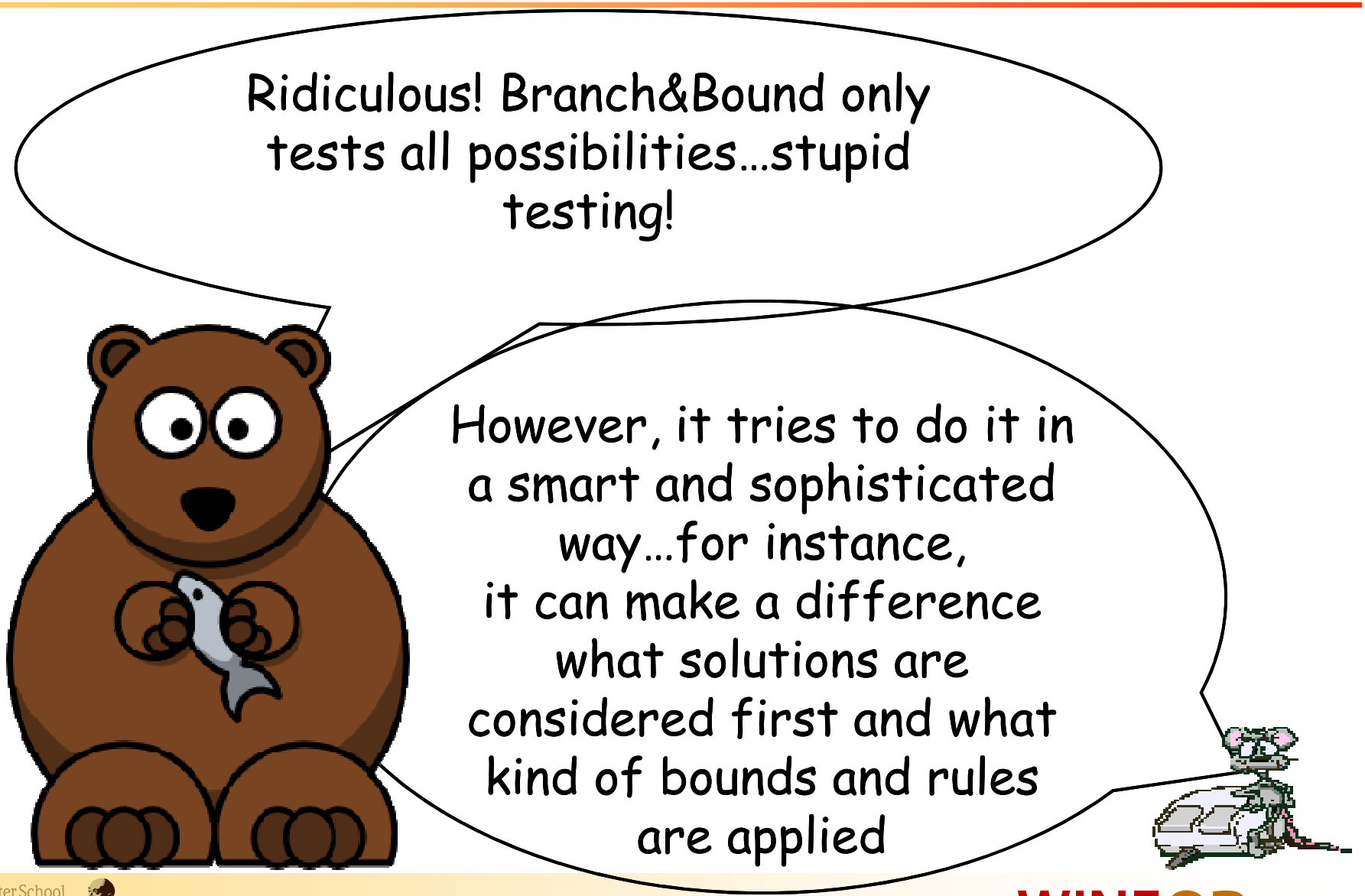
- A Branch&Bound algorithm explores the solution space by iteratively fixing variables with respective valid values
- This is conducted in parallel for numerous partial solutions that are stored in a specific data structure
- During the process, branching is conducted for a chosen partial solution by assigning alternative values to a selected variable
- If a complete solution is generated, we obtain an upper bound (minimization problem) by its objective function value
- Afterwards, the largest lower bound is calculated in order to examine the quality of the generated partial solution
- If there remains no gap between lower bound and upper bound, the currently considered partial solution is fathomed; otherwise it is stored in a priority list
- Hence, we can illustrate the enumeration process by an enumeration tree

Scheme of the B&B enumeration process



Active nodes (i.e., a living leaf) that are currently stored in the priority lists

Consequences of findings



3.1 Basic principles of B&B

- Basically, Branch&Bound (B&B) enumerates the solution space by systematically testing all possible values for the decision variables
- Therefore, in each step of the enumeration process, a B&B procedure branches over one or more variables
- Hence, different partial solutions are stored in a priority list. This list…
  - …sorts all generated partial solutions according to some criteria (depth, quality,…)
  - …decides about the next partial solution to be chosen (always proceeds with the first entry of this list)
  - …is usually organized as a priority heap (i.e., an efficient data structure that allows the identification of the element with highest priority in O(1))

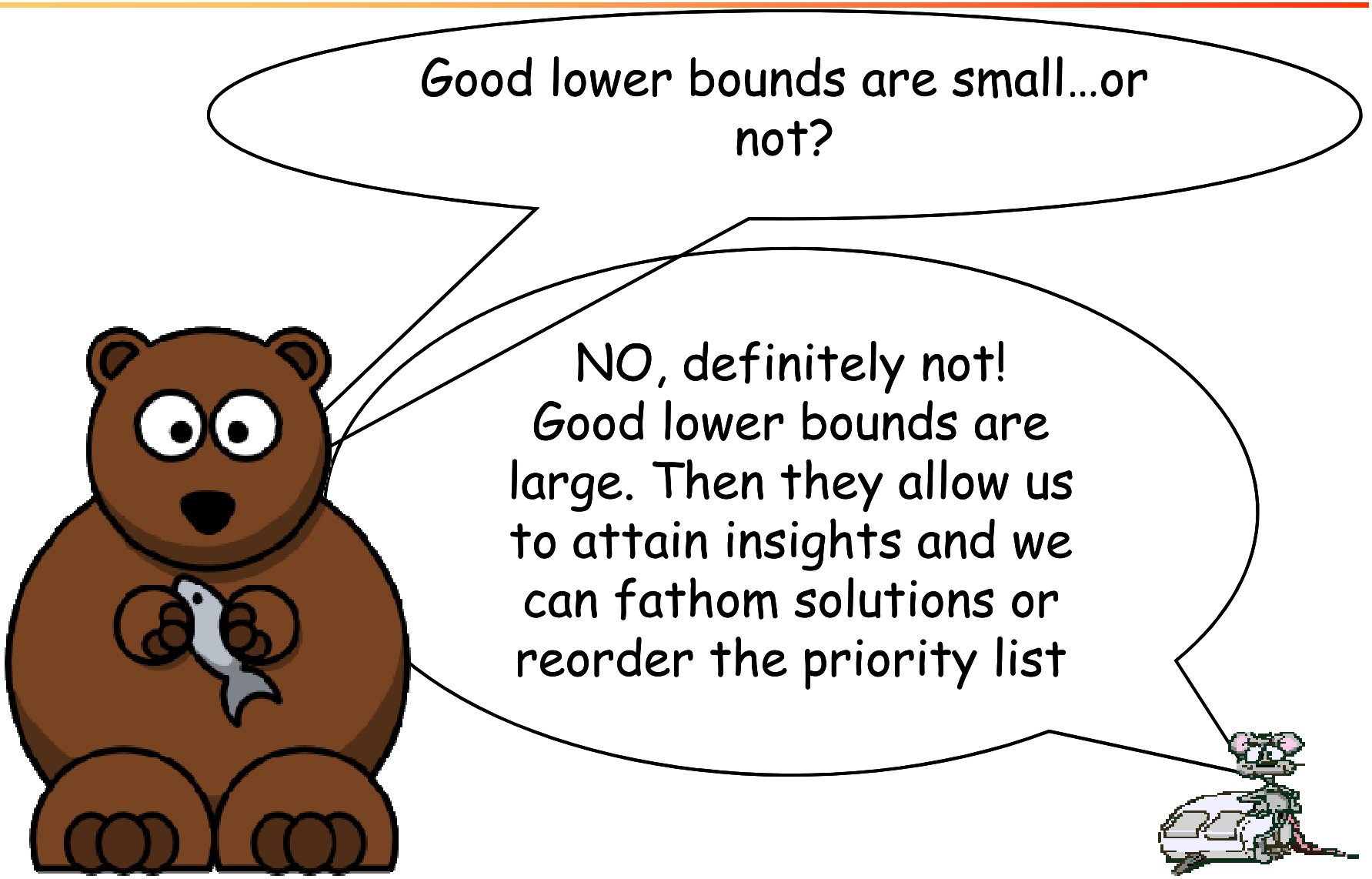
Branch&Bound elements

- Consequently, a B&B procedure has to unambiguously define rules or criteria by which
  - the priority list is sorted (Enumeration scheme)
  - a set of currently unfixed variables (in the chosen partial solution) is selected for being branched (Choice of branching variable/variables)
  - bounding and dominance rules are applied (Bound and Dominating)
- Since all these elements have significant influence on the overall performance of the algorithm, researchers have spent substantial effort in deriving and designing best performing meta-strategies
- Clearly, since the majority of considered problems is NP-hard, performance is assessed for average but not for worst cases. Worst cases are annoying and would result in exponential effort

Bounding

- In what follows, we assume a minimization problem
- Bounding allows us to fathom generated partial solutions early, i.e., before they have produced numerous predecessors
  - Specifically, in each node, a lower bound is derived that informs us about the solution quality maximally attainable by starting from the currently considered partial solution
  - Therefore, such a node can be fathomed if this lower bound is larger than or equal to an already known feasible solution
  - Upper bounds are either generated by the additional application of heuristics or derived from generated lower bounds during the enumeration process

Consequences of findings



Lower bounds

- LBs have significant impact on the overall efficiency of the enumeration process
- Hence, significant effort should be spent to derive tight bounds (…efficiently computable tight bounds are worth the effort)
- For many applications, it could be shown that tighter bounds lead to
  - Better sorted priority lists (in particular in case of BFS-B&B approaches)
  - Significant reductions of the number of explored solutions
- Clearly, the computational effort for deriving the bound in each node of the enumeration tree has to be kept limited
  - Tradeoff between positive impact and computational effort has to be addressed
  - Slight problem relaxations that lead to modified strong polynomial problems are frequently most promising

Dominance rules

- Besides bounding, B&B procedures should apply sophisticated dominance rules to eliminate non-promising alternative solutions
- These rules are frequently logically derived from attributes of the given problem
- For instance, a quite simple rule is to avoid unnecessary repetitions that are caused by testing an identical set of assigned elements in a modified sequence (if this leads to identical side effects)
  - In this case, we have to store already explored partial solutions (be careful: exponential number is possible)
  - while we delete dominated repetitions
- As known from the generation of efficient LBs, the definition of powerful dominance rules requires substantial insights into the problem structure

Enumeration scheme

- Clearly, the decision about the partial solution and their variables that are chosen for the next branching step have considerable impact on the resulting computational effort of the enumeration process
- For instance, if the search is conducted in less efficient parts of the solution space first, the entire process suffers from…
  - …less tightened upper and lower bounds
  - …partial solutions that are too inefficient to fathom substantial parts of the remaining unexplored solution space

Observation

- Therefore, in this case, substantial running time is spent on non-promising parts of the solution space
- Consequently, instances are not solved to optimality in reasonable time
- Hence, significant research effort is spent on deriving best enumeration schemes

Basic enumeration schemes

- Breadth-FS
  - Partial solutions are prioritized that are positioned at lower levels, i.e., solution with a minimum number of fixed variables
  - Consequently, the levels of the enumeration tree are iteratively generated
  - No preference for specific regions of the solution space
  - Memory consumption may become exhaustive
- Depth-FS
  - Partial solutions are prioritized that are positioned at deeper levels, i.e., solution with a maximum number of fixed variables
  - Consequently, solutions are completed much earlier
  - However, this scheme frequently does not guide the searching process to the most promising regions
  - Only a single current partial solution and the best known feasible solution are stored

Basic enumeration schemes

- Best-FS
  - Enumerates the partial solutions with lowest lower bound values first (preference of most promising partial solutions)
  - Memory consumption may become exhaustive
  - Therefore, in order to reduce this consumption, sophisticated techniques have to be additionally applied
  - Frequently, this method works quite efficiently (total average enumeration effort is minimized compared to the other listed methods)
  - However, the efficiency depends mainly on the significance of the applied lower bounds
  - First complete solution that is selected from the priority list for branching is proven to be optimal
  - Without applying additional upper bound generation methods, this procedure can terminate without finding any solution

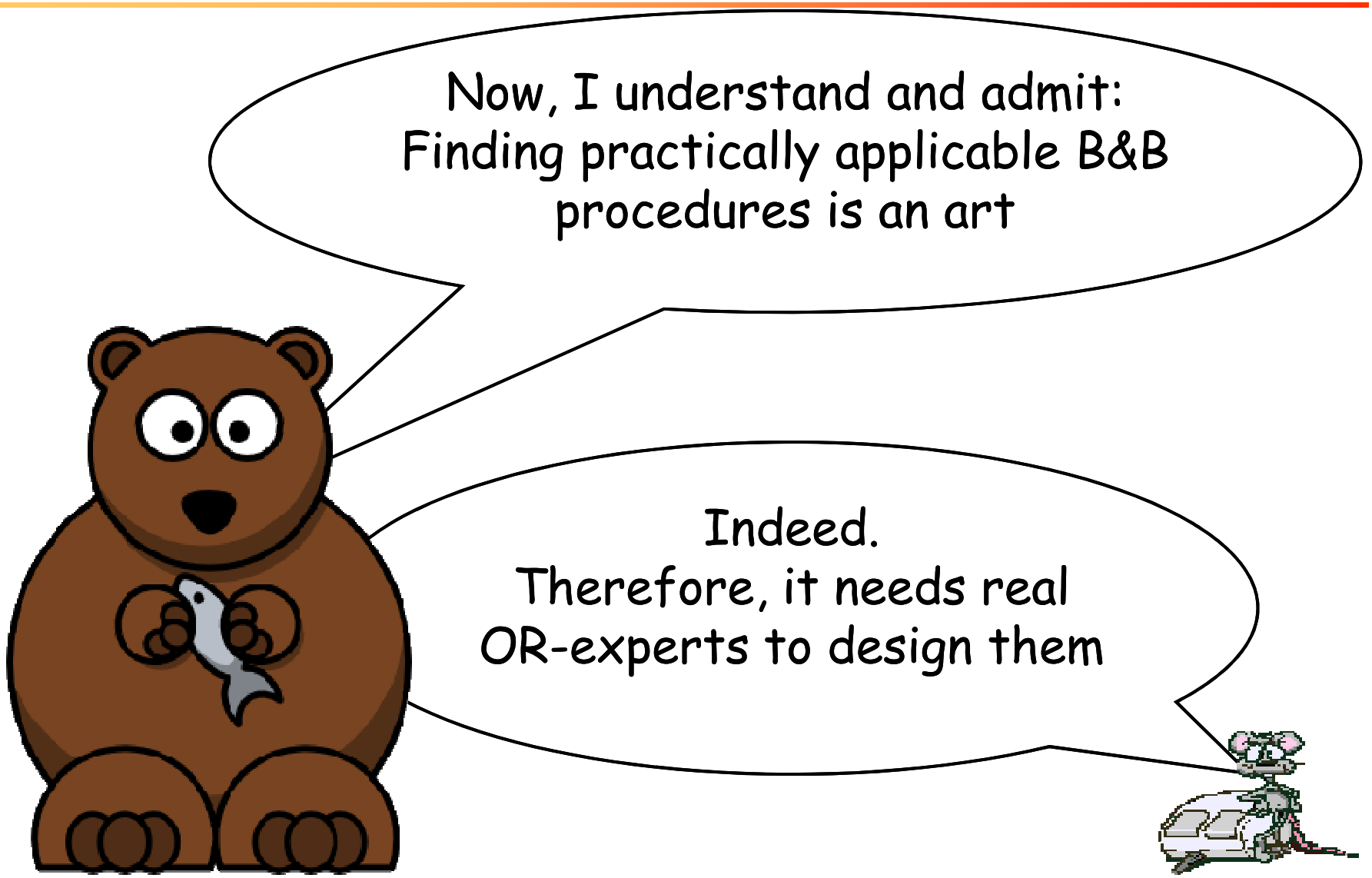
Basic enumeration schemes

- IDA*
  - BFS-adaption of DFS
  - Basic idea is to apply DFS, but branching is allowed only for partial solutions with a lower bound not exceeding a global lower bound
  - If no solution can be generated, the global bound is increased and the enumeration process restarted
  - Therefore, as known from BFS, most promising parts of the solution space are enumerated first
  - However, parts of the solution space have to be explored several times since the enumeration process has to be restarted several times with different global lower bounds
  - Repetition can be avoided to a certain amount if discarded solutions are stored in a second list to be potentially reactivated after increasing the global lower bound

Basic enumeration schemes

- LLB
  - ="Local Lower Bound"
  - Is "a mixture" of DFS and IDA*
  - It expands partial solutions in non-decreasing sequence of their lower bound values; but if a partial solution is chosen, its subtree is enumerated completely (however, in sequence of the lower bounds)
  - Therefore, the LLB works only locally and no repetition occurs during the enumeration process
  - However, the enumeration efficiency of BFS is not attained if, for instance, the applied lower bounds do not provide useful information on the first levels of the enumeration tree

Consequences of findings



3.2 Solving the Knapsack Problem with B&B

- In what follows, we exemplarily consider again the Knapsack Problem (KP) in order to initially illustrate two very simple Branch&Bound algorithms
- Attention:
  - The KP is a maximization problem
  - Therefore, the roles of lower and upper bounds are exchanged
  - New solutions may provide an increased lower bound (guaranteed profit)
  - Problem relaxations and/or logical conclusions may provide decreased upper bounds (maximum solution quality of a considered partial solution)

The lower bound of the LP-relaxation

- First of all, we introduce a modified numbering of the goods according to their efficiencies

$$e_1 = \frac{p_1}{w_1} \geq e_2 = \frac{p_2}{w_2} \geq \ldots \geq e_n = \frac{p_n}{w_n}$$

- Thus, we first determine the critical item s*
- After allocating the goods in increasing sequence, s* is the first job that does not fit into the knapsack
- Thus, we obtain a feasible solution by excluding s*
- This solution has the following total weight and price

$$p^* = \sum_{i=1}^{s^*-1} p_i \wedge w^* = \sum_{i=1}^{s^*-1} w_i$$

A first very simple upper bound

- Since items are sorted according to their efficiency in non-increasing order, we know that each capacity unit of the knapsack cannot be used more efficiently than e_1
- Consequently, if we have a partial solution with a current weight w and a current price *p* and let *b* be the item with lowest index whose status (in or out) has not been determined yet, we may obtain directly the following upper bound on the objective function value

$$U_0 = p + \left\lfloor (C - w) \cdot \frac{p_b}{w_b} \right\rfloor$$

- Thus, if no element is assigned, we obtain the following value

$$U_0 = \left\lfloor C \cdot \frac{p_1}{w_1} \right\rfloor$$

Applied to the solution of the LP-relaxation

- If we generate the solution of the LP-relaxation and erase the critical good from the knapsack, we can apply the simple upper bound
- Hence, the objective function value of the optimal solution of the LP-relaxation coincides with the resulting bound since the critical good is partially assigned
- This bound will be applied in the two simple B&B procedures that are introduced next

A more sophisticated upper bound

- Martello and Toth proposed the following upper bound
- It is based on the decision whether the current critical good b^* (see definition above) is assigned to the knapsack or not
- If b^* is not additionally assigned to the knapsack, the remaining capacity cannot be filled more efficiently than e_{b^*+1} price units per capacity unit
- If b^* is additionally assigned to the knapsack, the remaining **negative** capacity (i.e., knapsack is overfilled) "cannot be lost more efficiently" than e_{b^*-1} price units per capacity unit
- Thus, we obtain

$$U = \max\left\{\left\lfloor p + (C - w)\cdot\frac{p_{b^*+1}}{w_{b^*+1}}\right\rfloor, \left\lfloor p + p_{b^*} + (C - w - w_{b^*})\cdot\frac{p_{b^*-1}}{w_{b^*-1}}\right\rfloor\right\}$$

Two simple Branch&Bound versions

- In what follows, we exemplarily apply two different Branch&Bound algorithms
- First, a simple DFS procedure (Depth-first-search) that assigns the elements in an arbitrary sequence (i.e., just in sequence of the given element numbers)
- Moreover, a simple but much more efficient Best FS procedure (Best-first-search version) is illustrated that always branches the variable that decides about the assignment of the current critical element
- In both procedures, the simple upper bound (introduced before) is applied in each node

A DFS Branch&Bound algorithm

- This approach tries to allocate the items in a predetermined sequence
- Whenever a partial solution cannot be completed to a new optimal one, it is fathomed, i.e., backtracking is conducted
  - Generation of an upper bound of the maximal attainable weight by applying the LP-relaxation
  - Sum of current weight and this upper bound provides an upper bound
- Otherwise, if the solution is finally completed, a new temporary best solution improves the current lower bound
- A solution is denoted as completed whenever, due to capacity constraints, there is no assignable item available anymore

Example

- Let us consider again our small example

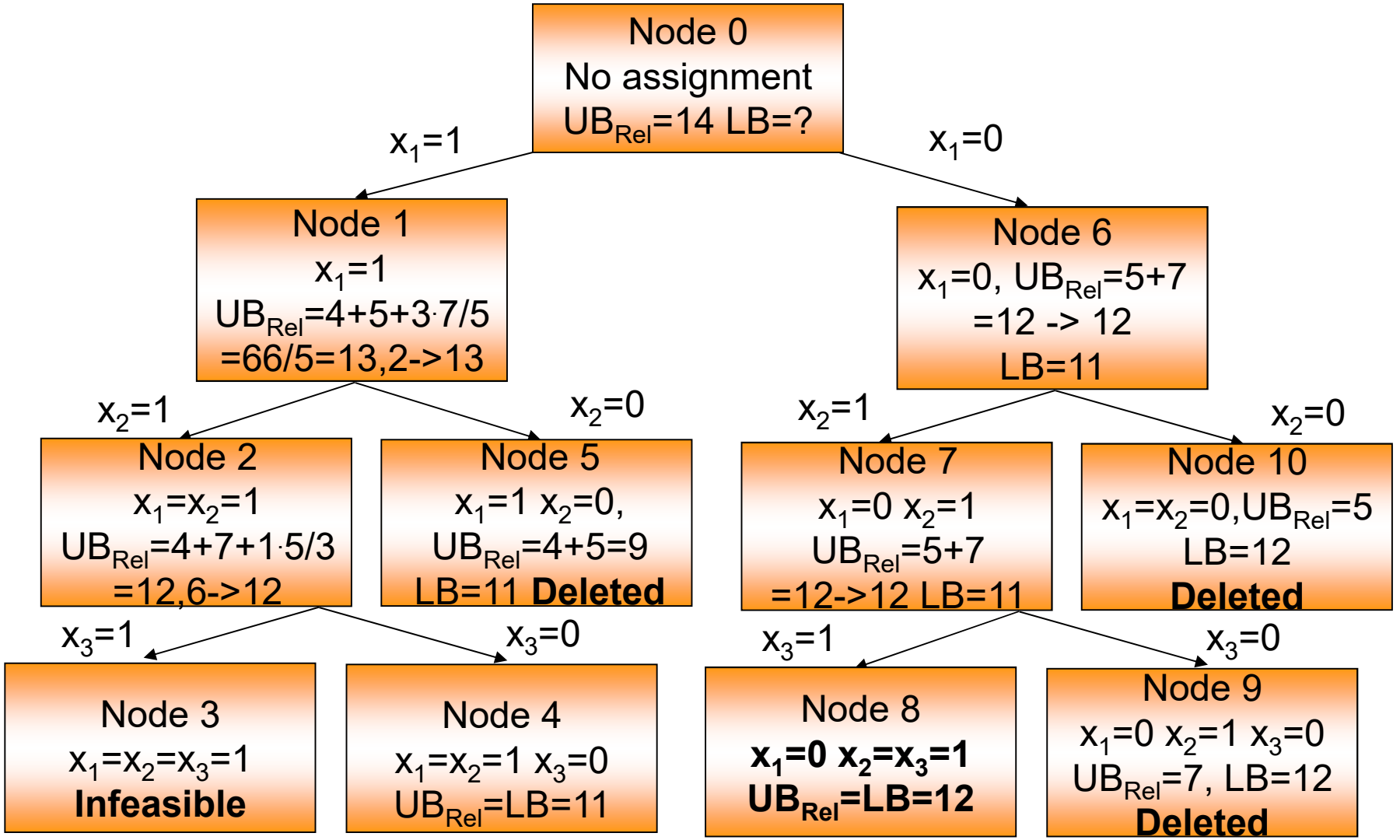
$$\text{Maximize } Z = 4 \cdot x_1 + 7 \cdot x_2 + 5 \cdot x_3$$

$$\text{s.t. } 4 \cdot x_1 + 5 \cdot x_2 + 3 \cdot x_3 \leq 10 \wedge x \in \{0,1\}^n$$

- We sort the items according to their efficiency
- Thus, we obtain the following table

Index of item	Price	Weight	Efficiency	Efficiency position
1	4	4	1	3
2	7	5	7/5	2
3	5	3	5/3	1

Applying the DFS Branch&Bound procedure



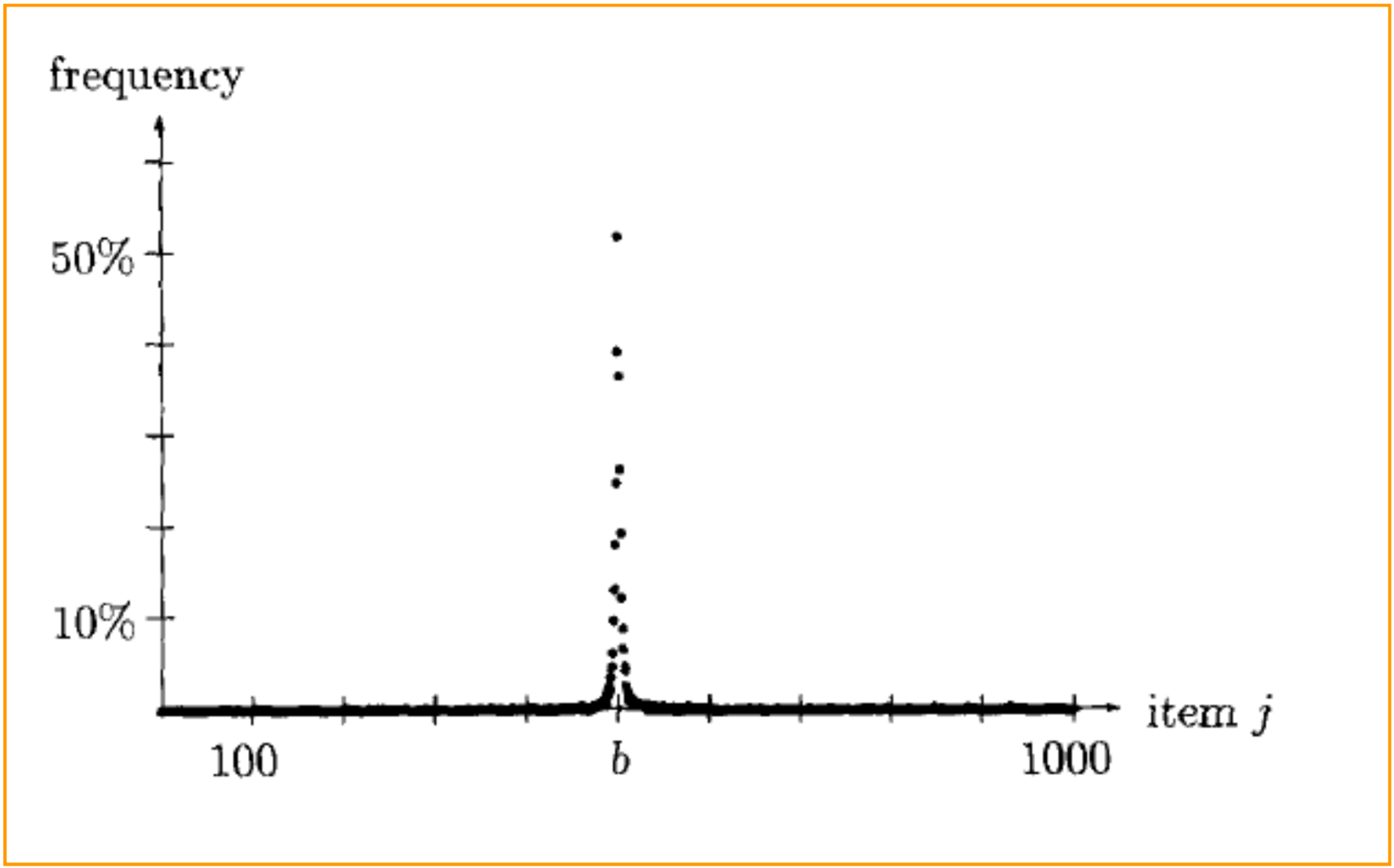
Observation

- Clearly, the computational effort can be significantly reduced by a smarter ordering of the job numbers
- Specifically, decisions of assigning or not assigning a critical job have much higher impact on derivable bounds
- This cognition is directly exploited in the following algorithm

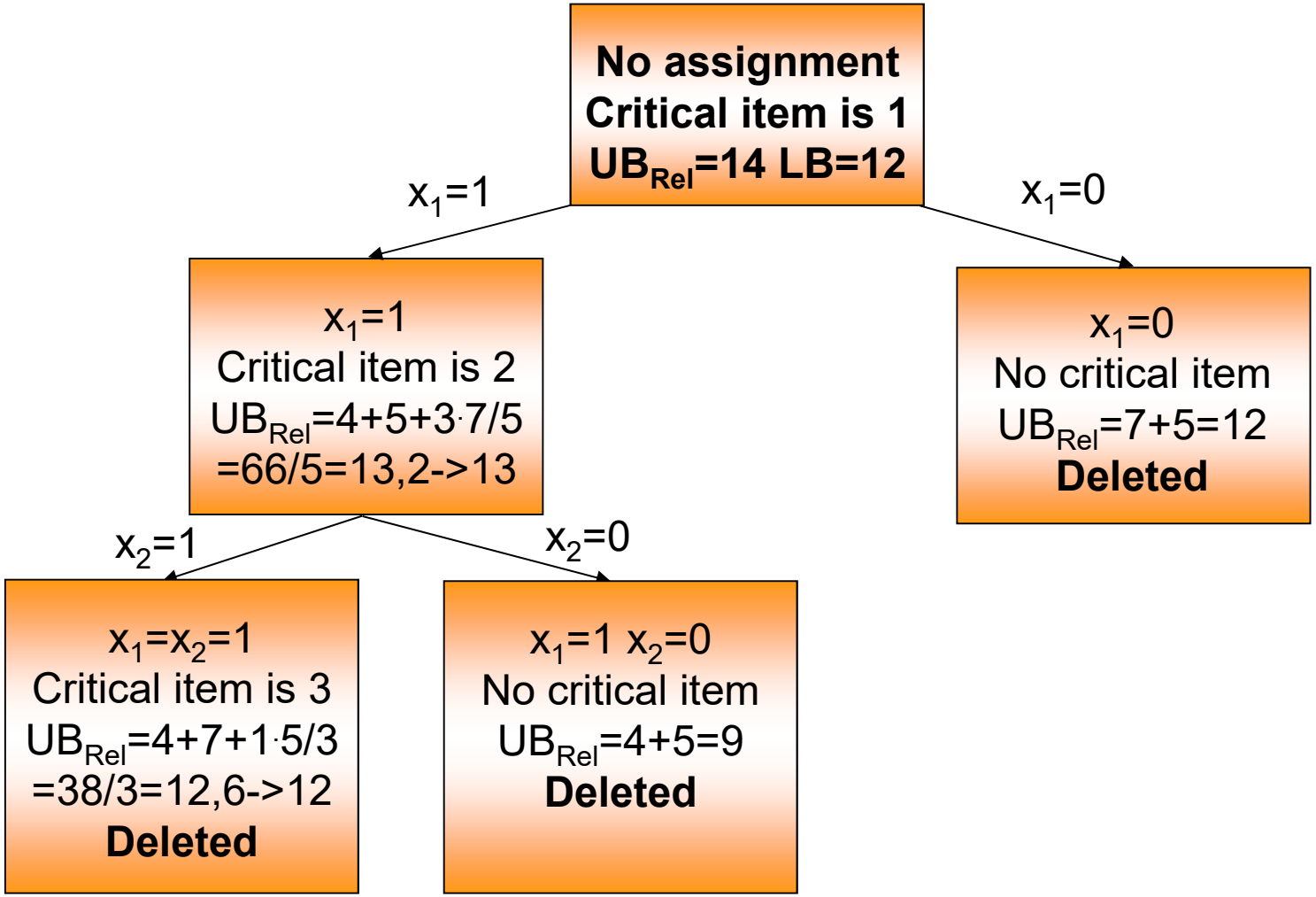
A Best FS Branch&Bound algorithm

- This kind of algorithm is characterized by the following attributes
  - Always proceeds with the node with the largest upper bound value
  - Again, a complete solution is kept throughout the calculations
  - This solution is derived from the LP-relaxation
  - Therefore, the critical element b* of this solution is considered for the next branching step. Alternative, states in a branching step are
    - b* is assigned
    - b* is not assigned
  - In order to derive an upper bound, the LP-relaxation of the Knapsack Problem is used again
  - Additionally, the solution of the relaxation without the assignment of the critical good is applied in order to provide a new solution, i.e., may be there is a new lower bound
- Usually, by applying this Best FS Branch&Bound algorithm, the resulting enumeration tree comprises a significantly lower number of generated nodes

Critical set of items



Enumeration process



3.3 A B&B approach to the aTSP

- In what follows, we consider again the aTSP
- We introduce a very illustrative but simple Branch&Bound approach exactly solving this problem
- It was originally proposed by Little et al. (1963) while these authors introduce the notation “Branch&Bound” for the first time

3.3.1 A problem relaxation to the aTSP

- In what follows, we introduce a somewhat helpful relaxation of the Traveling Salesman Problem
- Detailed analyses of the TSP provides us with the cognition that the subcycle restriction complicates the problem significantly
- To be more precise, if we drop these restrictions completely, we obtain a well-solvable problem, namely the **Linear Assignment Problem (LAP)**

Excursion: The LAP

- In what follows, we introduce a new problem frequently applied to layout planning constellations, the so-called Linear Assignment Problem (LAP)
- Basically, this model can be interpreted as an allocation problem of N elements to be placed on altogether N positions
- If an element is assigned to a specific location, predefined costs occur
- Every element has to be allocated to one definitely defined location
- The objective of the model is to allocate the N elements in a way that minimizes the resulting total costs

Mathematical definition of the LAP

- Parameters $\forall i \in \{1,\ldots,N\} : \forall j \in \{1,\ldots,N\} : C_{i,j}$

  Costs that occur if the ith element is placed on the jth location;

- Variables $\forall i \in \{1,\ldots,N\} : \forall j \in \{1,\ldots,N\} : x_{i,j}$

  Binary decision variables indicating whether the respective element is placed to the defined location;

Mathematical definition of the LAP

- **Definition of $x_{i,j}$:**

$$x_{i,j} = \begin{cases} 1 \text{ if the element i is located on position j} \\ 0 \text{ otherwise} \end{cases}$$

- **Restrictions:**

$$\forall j \in \{1,\ldots,N\} : \sum_{i=1}^{N} x_{i,j} = 1 \qquad (1)$$

Every location is occupied by exactly one element

$$\forall i \in \{1,\ldots,N\} : \sum_{j=1}^{N} x_{i,j} = 1 \qquad (2)$$

Every element is placed on exactly one location

Mathematical definition of the LAP

- Objective function

$$Minimize\, Z = \sum_{i=1}^{N} \sum_{j=1}^{N} x_{i,j} \cdot c_{i,j}$$

Minimize the total sum of allocation costs

Observations

- The model is equivalent to the aTSP without respecting its subcycle restrictions
- Why?
  - Note that each node i obtains an element j that is assigned to it
  - Consequently, we have a mapping succ(i)
  - Therefore, cyclical paths arise
  - Unfortunately, subcycles are possible
- Therefore, we can conclude that each optimal solution to the LAP is a lower bound to the optimal solution of the aTSP
- Moreover, we can conclude that each lower bound to the LAP is also a lower bound to the aTSP

Fortunately, the LAP is well-solvable

- Specifically, the LAP can be solved in polynomial time $O(N^3)$
- This is done by the application of the so-called well-known **Hungarian method**
- In order to understand this nice algorithm, we have to get some specific insights into the Linear Assignment Problem
- Therefore, in what follows, we take a closer look at some specific LAP attributes

LAP as a Linear Problem

- In what follows, we consider the LAP as a Linear Program

$$Ax = b \text{ with } x = \begin{pmatrix} x_{1,1} \\ x_{1,2} \\ \dots \\ x_{1,N} \\ x_{2,1} \\ \dots \\ x_{2,N} \\ \dots \\ x_{N,N} \end{pmatrix} \in IR^{N^2}; b = \begin{pmatrix} 1 \\ \dots \\ 1 \end{pmatrix} \in IR^{2 \cdot N}; c = \begin{pmatrix} c_{1,1} \\ c_{1,2} \\ \dots \\ c_{1,N} \\ c_{2,1} \\ \dots \\ c_{2,N} \\ \dots \\ c_{N,N} \end{pmatrix} \in IR^{N^2}$$

LAP as a Linear Problem

and matrix $A =$

$$\begin{pmatrix}
1 & 0 & \dots & 0 & 1 & 0 & \dots & 0 & 1 & 0 & \dots & 0 & 1 & 0 & \dots & 0 & \dots 1 & 0 & \dots & 0 \\
0 & 1 & 0\dots & 0 & 0 & 1 & 0\dots & 0 & 0 & 1 & 0\dots & 0 & 0 & 1 & 0\dots & 0 & \dots 0 & 1 & 0\dots & 0 \\
\dots & \dots & \dots & \dots & \dots & \dots & & & & & & & & & & & & & & \\
\dots & & & & & & & & & & & & & & & & & & & \\
 & & & & & & & & & & & & & & & & & & & \\
 & & & & & & & & & & & & & & & & & & & \\
 & & & & & & & & & & & & & & & & & & & \\
 & & & & & & & & & & & & & & & & & & & \\
 & & & & & & & & & & & & & & & & & & & \\
 & & & & & & & & & & & & & & & & & & & \\
 & & & & & & & & & & & & & & & & & & & \\
\dots & & & \dots & & & & & & & & & & & & & & & & \\
0 & 0 & \dots 0 & 1 & 0 & \dots & \dots 0 & 1 & 0 & \dots & \dots 0 & 1 & 0 & \dots & \dots 0 & 1 & 0 & \dots & \dots 0 & 1 \\
1 & 1 & \dots & 1 & 0 & \dots & 0 & 0 & 0 & \dots & \dots & 0 & 0 & \dots & \dots & 0 & & & & \\
 & & & & 1 & 1 & \dots & 1 & 0 & \dots & \dots & 0 & 0 & \dots & \dots & 0 & & & & \\
 & & & & & & & & 1 & 1 & \dots & 1 & 0 & \dots & \dots & 0 & & & & \\
 & & & & & & & & & & & & \dots & & & \dots & \dots & \dots & \dots & \dots \\
 & & & & & & & & & & & & \dots & & & \dots & 0 & 0 & \dots & 0 \\
 & & & & & & & & & & & & \dots & & & \dots & \dots & \dots & \dots & \dots \\
 & & & & & & & & & & & & \dots & & & \dots & 0 & 0 & \dots & 0 \\
 & & & & & & & & & & & & \dots & & & \dots & 0 & 0 & \dots & 0 \\
 & & & & & & & & & & & & \dots & & & \dots & \dots & \dots & \dots & \dots \\
 & & & & & & & & & & & & & & & & 1 & 1 & \dots & 1
\end{pmatrix}$$

LAP as a Linear Problem - example for N = 4

$$Ax = b \text{ with } x = \begin{pmatrix} x_{1.1} \\ x_{1.2} \\ x_{1.3} \\ x_{1.4} \\ x_{2.1} \\ x_{2.2} \\ x_{2.3} \\ x_{2.4} \\ x_{3.1} \\ x_{3.2} \\ x_{3.3} \\ x_{3.4} \\ x_{4.1} \\ x_{4.2} \\ x_{4.3} \\ x_{4.4} \end{pmatrix} \in IR^{N^2}; \; b = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \in IR^{2 \cdot N}; \; c = \begin{pmatrix} c_{1.1} \\ c_{1.2} \\ c_{1.3} \\ c_{1.4} \\ c_{2.1} \\ c_{2.2} \\ c_{2.3} \\ c_{2.4} \\ c_{3.1} \\ c_{3.2} \\ c_{3.3} \\ c_{3.4} \\ c_{4.1} \\ c_{4.2} \\ c_{4.3} \\ c_{4.4} \end{pmatrix} \in IR^{N^2}$$

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

- The first four lines of matrix A represent the first group of restrictions (1); the last four lines represent the second group of restrictions (2)

The dual of the LP_{LAP}

- In order to obtain some insights into the problem structure, we consider the dual of the LAP

$$A^T \pi \leq c \; with: \; \pi = \begin{pmatrix} v_1 \\ \dots \\ v_N \\ u_1 \\ \dots \\ u_N \end{pmatrix} \in IR^{2N}; \; b = \begin{pmatrix} 1 \\ \dots \\ 1 \end{pmatrix} \in IR^{2N}; \; c = \begin{pmatrix} c_{1,1} \\ c_{1,2} \\ \dots \\ c_{1,N} \\ c_{2,1} \\ \dots \\ c_{2,N} \\ \dots \\ c_{N,N} \end{pmatrix} \in IR^{N^2}.$$

Matrix A^T

$$
\text{and } A^T = \begin{pmatrix}
1 & 0 & \dots & \dots & 0 & 1 & 0 & \dots & \dots & 0 \\
 & 1 & & & & 1 & \dots & & & \\
 & & 1 & & & \dots & \dots & & & \\
 & & & \dots & & \dots & \dots & & & \\
 & & & & 1 & 1 & 0 & & & 0 \\
1 & 0 & & & 0 & & 1 & & & 0 \\
 & 1 & 0 & \dots & 0 & & 1 & & & 0 \\
 & & 1 & 0\dots & 0 & & \dots & & & \dots \\
 & & & \dots & 0 & & \dots & & & \dots \\
 & & & & 1 & & 1 & & & 0 \\
\dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\
1 & 0 & \dots & \dots & 0 & & & & 0 & 1 \\
 & 1 & & & & & & & 0 & 1 \\
 & & \dots & \dots & & & & & \dots & \dots \\
 & & \dots & 0 & 1 & & & & 0 & 1
\end{pmatrix}
$$

Observations – The dual of the LAP

- Obviously, owing to the simple structure of matrix A, the dual of the LAP has a nice structure
- By analyzing the dual program, we get:

$$c_{i,j} \geq u_i + v_j$$

- And we know that if x and π are optimal, it holds that:

$$b^T \pi = c^T x$$

$$\Leftrightarrow \sum_{i=1}^{N} u_i + \sum_{i=1}^{N} v_i = \sum_{i=1}^{N} \sum_{j=1}^{N} c_{i,j} x_{i,j}$$

Observations – The dual of the LAP

- Consider the following solution:
  - Take a feasible solution of the LAP
  - For every element $x_{i,j}=1$ define $u_i+v_j=c_{i,j}$
  - Consider the resulting objective function value

$$\sum_{i=1}^{N}\sum_{j=1}^{N} c_{i,j}x_{i,j} = \sum_{i=1}^{N} \sum_{j\in\{k\,|\,x_{i,k}=1,k\in\{1,\ldots,N\}\}} c_{i,j}x_{i,j} = \sum_{(i,j)\in\{(k,l)\,|\,x_{k,l}=1,(k,l)\in\{1,\ldots,N\}\times\{1,\ldots,N\}\}} c_{i,j}x_{i,j}$$

$$= \sum_{(i,j)\in\{(k,l)\,|\,x_{k,l}=1,(k,l)\in\{1,\ldots,N\}\times\{1,\ldots,N\}\}} \left(u_i + v_j\right)\cdot x_{i,j}$$

$$= \sum_{(i,j)\in\{(k,l)\,|\,x_{k,l}=1,(k,l)\in\{1,\ldots,N\}\times\{1,\ldots,N\}\}} \left(u_i + v_j\right) = \sum_{i=1}^{N} u_i + \sum_{j=1}^{N} v_j$$

Observations – The dual of the LAP

- Therefore, we can conclude that if the constructed solution is feasible for the dual problem it would be optimal for our relaxed problem
- Question: Is it always feasible?
- Answer: Unfortunately, no!
- Note that for all combinations *(i,j)*: $u_i+v_j \leq c_{i,j}$ has to be fulfilled
- We have to define an algorithm generating the respective u_i and v_j values without violating any restriction of the dual problem
- This solution is proven to be an optimal one

Reduction of the matrix

Theorem:

If a constant is added to (or subtracted from) all entries in a row or column of an assignment matrix, the optimal assignment is kept unchanged

The correctness of the Theorem follows directly from the following facts

- In each row/column exactly one assignment is done
- Thus, the objective function value is modified by the constant for all solutions, but the relative ordering is kept

The Hungarian method

1. Initialization
   - Subtract the smallest entry in each row from every entry in this row
   - Subtract the smallest entry in each column from every entry in this column
   - Here we stop the execution of the algorithm and use the computed reduction of the matrix as a valid lower bound for the LAP
2. Try to find a feasible assignment by only using zero entries. If a feasible assignment is already possible, stop the algorithm. Otherwise, proceed with the next step
3. Cover the zeros with a minimum number of (vertical and horizontal) lines
4. Subtract the minimum uncovered entry from every uncovered entry and add this selected value to every twice covered entry; Go to step 2

Correctness of the procedure

- The correctness of the first three steps follows directly from the theorem itemized above
- However, what about step four?
  - First of all, it can be stated that only uncovered elements in the matrix remain to be used as new positions for an element
  - Additionally, if a twice covered element is zero, we have alternative zeros in the respective row and column
    - This results from the fact that there is at least one additional element equal to zero either in the respective row or column
    - Therefore, we keep a cost-equivalent alternative

Correctness of the procedure

- In this step, we subtract the smallest value from all uncovered entries, i.e., we subtract this value from all rows (or alternatively all columns)
- Clearly, owing to the Theorem, this step results in an equivalent problem
- Subsequently, we add this value to all covered rows and columns. Thus, the twice covered ones are increased twice
- Again, by making use of the Theorem, we know that we obtain an equivalent problem

- Additionally, we know that in each step of the procedure, at least one new zero element is generated
- Thus, the procedure terminates with an optimal solution

Complexity of the procedure

- Clearly, steps 1 up to 4 can be executed in time $O(N^2)$
- Moreover, we have at most O(N) iterations of the steps 2-4

- Thus, we obtain an overall complexity of $O(N^3)$ steps

Simple examples

- We consider the following two examples

$$C_1 = \begin{pmatrix} 9 & 12 & 16 \\ 12 & 8 & 10 \\ 15 & 11 & 12 \end{pmatrix}; \; C_2 = \begin{pmatrix} 1 & 4 & 6 & 3 \\ 9 & 7 & 10 & 9 \\ 4 & 5 & 11 & 7 \\ 8 & 7 & 8 & 5 \end{pmatrix}$$

Example C_1

- We commence with the following cost matrix

$$C_1 = \begin{pmatrix} 9 & 12 & 16 \\ 12 & 8 & 10 \\ 15 & 11 & 12 \end{pmatrix}$$

Example 1

$$C_1 = \begin{pmatrix} 9 & 12 & 16 \\ 12 & 8 & 10 \\ 15 & 11 & 12 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 3 & 7 \\ 4 & 0 & 2 \\ 4 & 0 & 1 \end{pmatrix}$$

$$\Rightarrow \begin{pmatrix} 0 & 3 & 6 \\ 4 & 0 & 1 \\ 4 & 0 & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} \boldsymbol{9} & 12 & 16 \\ 12 & \boldsymbol{8} & 10 \\ 15 & 11 & \boldsymbol{12} \end{pmatrix}$$ optimal assignment

Example C_2

- The second example is somewhat more complex

$$C_2 = \begin{pmatrix} 1 & 4 & 6 & 3 \\ 9 & 7 & 10 & 9 \\ 4 & 5 & 11 & 7 \\ 8 & 7 & 8 & 5 \end{pmatrix}$$

Example 2

- We apply the first step of the algorithm

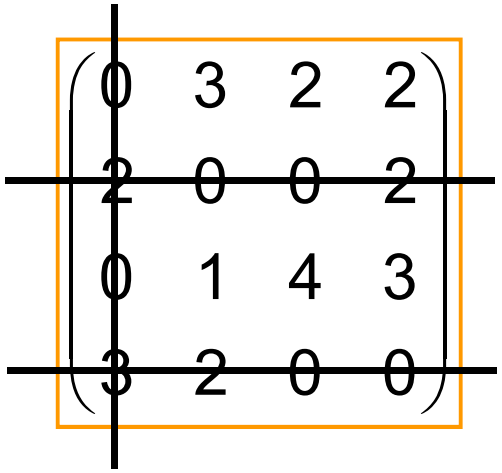
$$C_2 = \begin{pmatrix} 1 & 4 & 6 & 3 \\ 9 & 7 & 10 & 9 \\ 4 & 5 & 11 & 7 \\ 8 & 7 & 8 & 5 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 3 & 5 & 2 \\ 2 & 0 & 3 & 2 \\ 0 & 1 & 7 & 3 \\ 3 & 2 & 3 & 0 \end{pmatrix}$$

$$\Rightarrow \begin{pmatrix} 0 & 3 & 2 & 2 \\ 2 & 0 & 0 & 2 \\ 0 & 1 & 4 & 3 \\ 3 & 2 & 0 & 0 \end{pmatrix}$$

$\Rightarrow$ No solution since the rows 1 and 3 are blocking each other

Example 2 – step 3

- Now, we have to cover the zeros with a minimal number of horizontal or vertical lines



- Minimal value of uncovered elements is 1
- Thus, we now obtain

Thus, we obtain by conducting step 4

- … the following equivalent problem

$$\begin{pmatrix} 0 & 2 & 1 & 1 \\ 3 & 0 & 0 & 2 \\ 0 & 0 & 3 & 2 \\ 4 & 2 & 0 & 0 \end{pmatrix}$$

- And obtain the optimal solution

$$C_2 = \begin{pmatrix} \boldsymbol{1} & 4 & 6 & 3 \\ 9 & 7 & \boldsymbol{10} & 9 \\ 4 & \boldsymbol{5} & 11 & 7 \\ 8 & 7 & 8 & \boldsymbol{5} \end{pmatrix}$$

Consequences

- Clearly, the optimal solution to the LAP directly provides us with a lower bound to the considered TSP
- Additionally, each feasible dual solution (for instance the dual result for the first step) also determines a lower bound to the TSP

3.3.2 The B&B procedure of Little et al.

- In what follows, we briefly introduce the well-known Branch&Bound procedure of Little et al. (1963)
- As already shown, we know that each optimal solution to the LAP is a lower bound of the optimal solution to the aTSP
- Moreover, we can conclude that each lower bound for the LAP is also a lower bound for the aTSP
- As proven above, the LAP can be solved in polynomial time $O(n^3)$ by the **Hungarian method**
- However, during the time-consuming enumeration process of a Branch&Bound procedure, even this efficient computation may be too costly since it is applied in each node
- Therefore, we need a fast computable lower bound for the LAP to be applied in each node of the resulting B&B tree

Bounding

- Consequently, only the first step of the Hungarian method is applied in order to derive a first feasible dual solution
- Note that the objective function of the dual solution is a lower bound for the Linear Assignment Problem (LAP)
- This bound is applied in each node

Branching scheme I

- By using the LAP bound, the algorithm works in a best-first manner which branches always the node with the lowest bound value
- In each branching step, a specific variable $x_{i,j}$ is taken, wherefore two subsequent nodes arise by the possible values $x_{i,j}=0$ and $x_{i,j}=1$
  - For the 0-case, the respective costs $c_{i,j}$ are set to infinity
  - For the 1-case, the respective costs $c_{i,j}$ are kept unchanged and all other alternative entries get the infinity value
- After reducing the matrix, the new nodes get the lower bound values

$$F_v = \sum_{i=1}^{N} u_i + \sum_{j=1}^{N} v_j$$

Branching scheme II

- But, how can we select the respective $x_{i,j}$ variables to generate the branching step?
  - The variable is chosen which maximizes the resulting lower bound
  - By fixing $x_{i,j}$ to 0, we forbid this transport, and therefore we get at least the additional costs $k_{i,j}=\min\{c_{i,p} \mid p=1,\ldots,N\} + \min\{c_{p,j} \mid p=1,\ldots,N\}$ (detailed computation is given on the next slide) while the c-values are from the cost matrix of the considered node and $c_{i,j}$ is set to infinity
  - Therefore, in each node, we chose $x_{a,b}$ with $k_{a,b}=\max\{k_{i,j} \mid i,j=1,\ldots,N\}$ to maximize the resulting lower bounds of the subsequent nodes
  - After adding a tour element by using the 1-value, we can eliminate all edges that would lead to a subcycle

Computation of matrix *K*

Let $K = (k_{i,j})_{1 \le i,j \le N} = \begin{cases} \min\limits_{k \in \{1,\dots,N\}} \{c_{k,j}\} + \min\limits_{l \in \{1,\dots,N\}} \{c_{i,l}\} & if\ x_{i,j}\ was\ not\ branched\ before \\ & and\ x_{i,j}\ was\ not\ excluded \\ -\infty & otherwise \end{cases}$

and $k_{a,b} = \max\{k_{i,j} \mid x_{i,j}\ wasn't\ branched\ and\ x_{i,j}\ wasn't\ excluded\}$.

We chose $x_{a,b}$ as the branching variable.

Example

$$C = (c_{i,j})_{1 \le i,j \le N} = \begin{pmatrix} \infty & 7 & 4 & 2 & 1 & 3 \\ 3 & \infty & 3 & 2 & 4 & 6 \\ 2 & 3 & \infty & 4 & 5 & 3 \\ 7 & 1 & 5 & \infty & 4 & 4 \\ 4 & 4 & 3 & 5 & \infty & 3 \\ 4 & 3 & 3 & 6 & 2 & \infty \end{pmatrix}$$

Reducing the matrix root node 0

$$C = (c_{i,j})_{1 \le i,j \le N} = \begin{pmatrix} \infty & 6 & 3 & 1 & 0 & 2 \\ 1 & \infty & 1 & 0 & 2 & 4 \\ 0 & 1 & \infty & 2 & 3 & 1 \\ 6 & 0 & 4 & \infty & 3 & 3 \\ 1 & 1 & 0 & 2 & \infty & 0 \\ 2 & 1 & 1 & 4 & 0 & \infty \end{pmatrix} \begin{matrix} 1 \\ 2 \\ 2 \\ 1 \\ 3 \\ 2 \end{matrix}$$

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

Lower bound is $1+2+2+1+3+2=11$

Search for the branching variable

$$K = \left(k_{i,j}\right)_{1\le i,j\le N} = \begin{pmatrix} \infty & 0 & 0 & 0 & 1 & 0 \\ 0 & \infty & 0 & 2 & 0 & 0 \\ 2 & 0 & \infty & 0 & 0 & 0 \\ 0 & \mathbf{4} & 0 & \infty & 0 & 0 \\ 0 & 0 & 1 & 0 & \infty & 1 \\ 0 & 0 & 0 & 0 & 1 & \infty \end{pmatrix}$$

We chose $x_{4,2}$ as the branching variable

$x_{4,2}=1$ (node 2)

$$C = \left(c_{i,j}\right)_{1\le i,j\le N} = \begin{pmatrix} \infty & \infty & 3 & 1 & 0 & 2 \\ 1 & \infty & 1 & \infty & 2 & 4 \\ 0 & \infty & \infty & 2 & 3 & 1 \\ \infty & \mathbf{0} & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & 2 & \infty & 0 \\ 2 & \infty & 1 & 4 & 0 & \infty \end{pmatrix}$$

$x_{4,2}=1$ and reducing

$$C=\left(c_{i,j}\right)_{1\le i,j\le N}=\begin{pmatrix}\infty & \infty & 3 & 0 & 0 & 2 & 0\\ 0 & \infty & 0 & \infty & 1 & 3 & 1\\ 0 & \infty & \infty & 1 & 3 & 1 & 0\\ \infty & \mathbf{0} & \infty & \infty & \infty & \infty & 0\\ 1 & \infty & 0 & 1 & \infty & 0 & 0\\ 2 & \infty & 1 & 3 & 0 & \infty & 0\\ 0 & 0 & 0 & 1 & 0 & 0 & \end{pmatrix}$$

Lower bound is $11+1+1=13$

$x_{4,2}=0$ (node 1)

$$C = \left(c_{i,j}\right)_{1\le i,j\le N} = \begin{pmatrix} \infty & 6 & 3 & 1 & 0 & 2 \\ 1 & \infty & 1 & 0 & 2 & 4 \\ 0 & 1 & \infty & 2 & 3 & 1 \\ 6 & \infty & 4 & \infty & 3 & 3 \\ 1 & 1 & 0 & 2 & \infty & 0 \\ 2 & 1 & 1 & 4 & 0 & \infty \end{pmatrix}$$

$x_{4,2}=0$ and reducing

$$C=\left(c_{i,j}\right)_{1\le i,j\le N}=\begin{pmatrix}\infty & 5 & 3 & 1 & 0 & 2 & 0\\ 1 & \infty & 1 & 0 & 2 & 4 & 0\\ 0 & 0 & \infty & 2 & 3 & 1 & 0\\ 3 & \infty & 1 & \infty & 0 & 0 & 3\\ 1 & 0 & 0 & 2 & \infty & 0 & 0\\ 2 & 0 & 1 & 4 & 0 & \infty & 0\\ 0 & 1 & 0 & 0 & 0 & 0 & \end{pmatrix}$$

Lower bound $11+3+1=15$

The process is continued with node 2

$$K = \left(k_{i,j}\right)_{1\le i,j\le N} = \begin{pmatrix} \infty & \infty & 0 & 1 & 0 & 0 \\ 0 & \infty & 0 & \infty & 0 & 0 \\ \mathbf{1} & \infty & \infty & 0 & 0 & 0 \\ \infty & 0 & \infty & \infty & \infty & \infty \\ 0 & \infty & 0 & 0 & \infty & 1 \\ 0 & \infty & 0 & 0 & 1 & \infty \end{pmatrix}$$

$x_{3,1}$ is chosen

$x_{3,1}=0$

$$C = \left(c_{i,j}\right)_{1\le i,j\le N} = \begin{pmatrix} \infty & \infty & 3 & 0 & 0 & 2 \\ 0 & \infty & 0 & \infty & 1 & 3 \\ \infty & \infty & \infty & 1 & 3 & 1 \\ \infty & 0 & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & 1 & \infty & 0 \\ 2 & \infty & 1 & 3 & 0 & \infty \end{pmatrix}$$

$x_{3,1}=0$ and reducing

$$C=\left(c_{i,j}\right)_{1\le i,j\le N}=\begin{pmatrix} \infty & \infty & 3 & 0 & 0 & 2 & 0\\ 0 & \infty & 0 & \infty & 1 & 3 & 0\\ \infty & \infty & \infty & 0 & 2 & 0 & 1\\ \infty & 0 & \infty & \infty & \infty & \infty & 0\\ 1 & \infty & 0 & 1 & \infty & 0 & 0\\ 2 & \infty & 1 & 3 & 0 & \infty & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & \end{pmatrix}$$

Lower bound $13+1=14$

$x_{3,1}=1$

$$K = \left(k_{i,j}\right)_{1\le i,j\le N} = \begin{pmatrix} \infty & \infty & \infty & 0 & 0 & 2 \\ \infty & \infty & 0 & \infty & 1 & 3 \\ 0 & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 1 & \infty & 0 \\ \infty & \infty & 1 & 3 & 0 & \infty \end{pmatrix}$$

$x_{3,1}$=1 and reducing

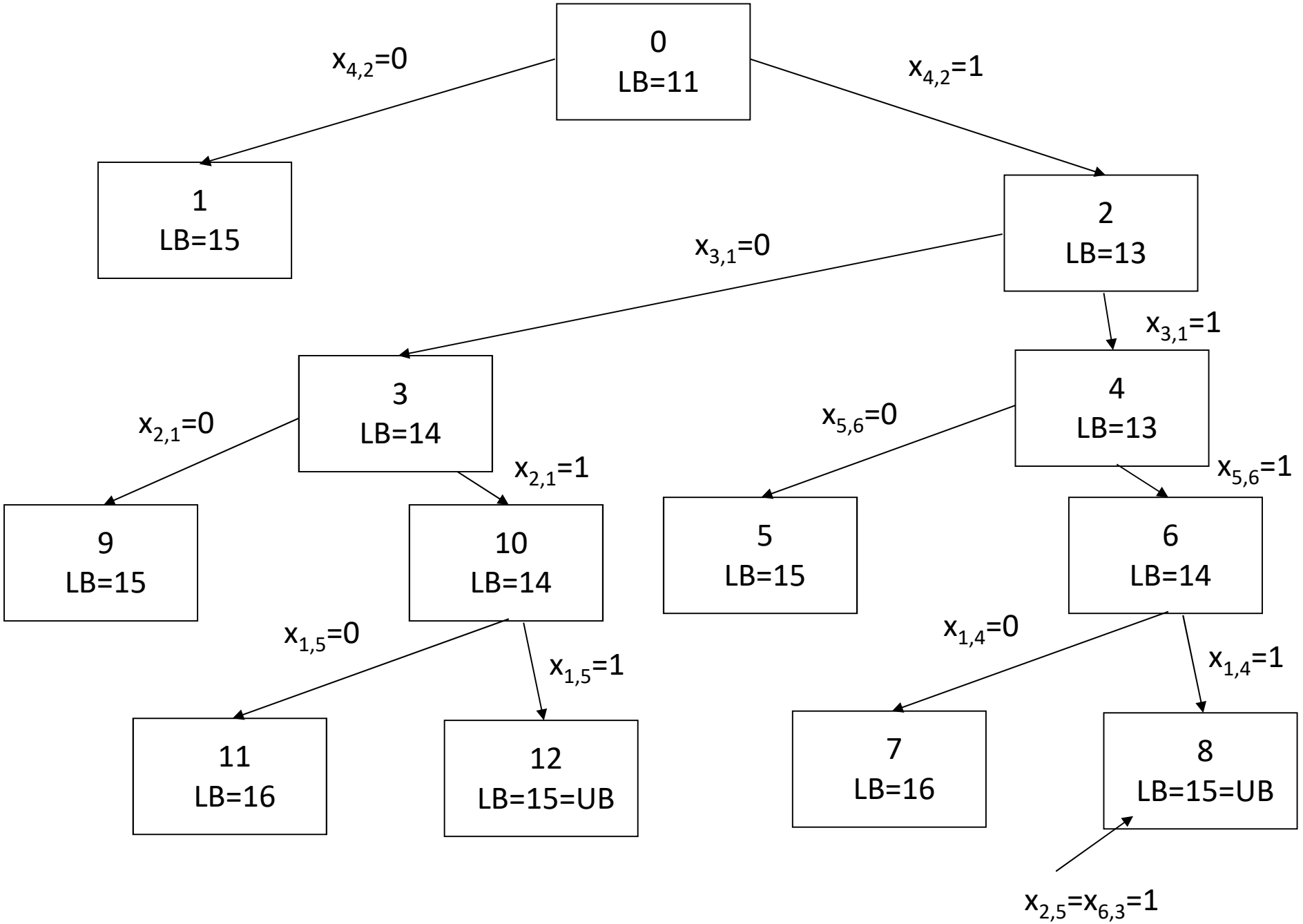
$$C = (c_{i,j})_{1 \le i,j \le N} = \begin{pmatrix} \infty & \infty & \infty & 0 & 0 & 2 \\ \infty & \infty & 0 & \infty & 1 & 3 \\ 0 & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 1 & \infty & 0 \\ \infty & \infty & 1 & 3 & 0 & \infty \end{pmatrix} \begin{matrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{matrix}$$

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

Lower bound 13

to be continued …

Complete enumeration tree of the B&B process



Computational results

- Tests were conducted on an IBM 7090
- Two types of problems were studied
  - Asymmetric distance matrices with elements consisting of uniformly distributed 3-digit random numbers
  - Various published problems and subproblems constructed therefrom by deleting cities. Most of these problems have been made up from road atlases or maps, and are symmetric
- The following slide provides us with the measured results

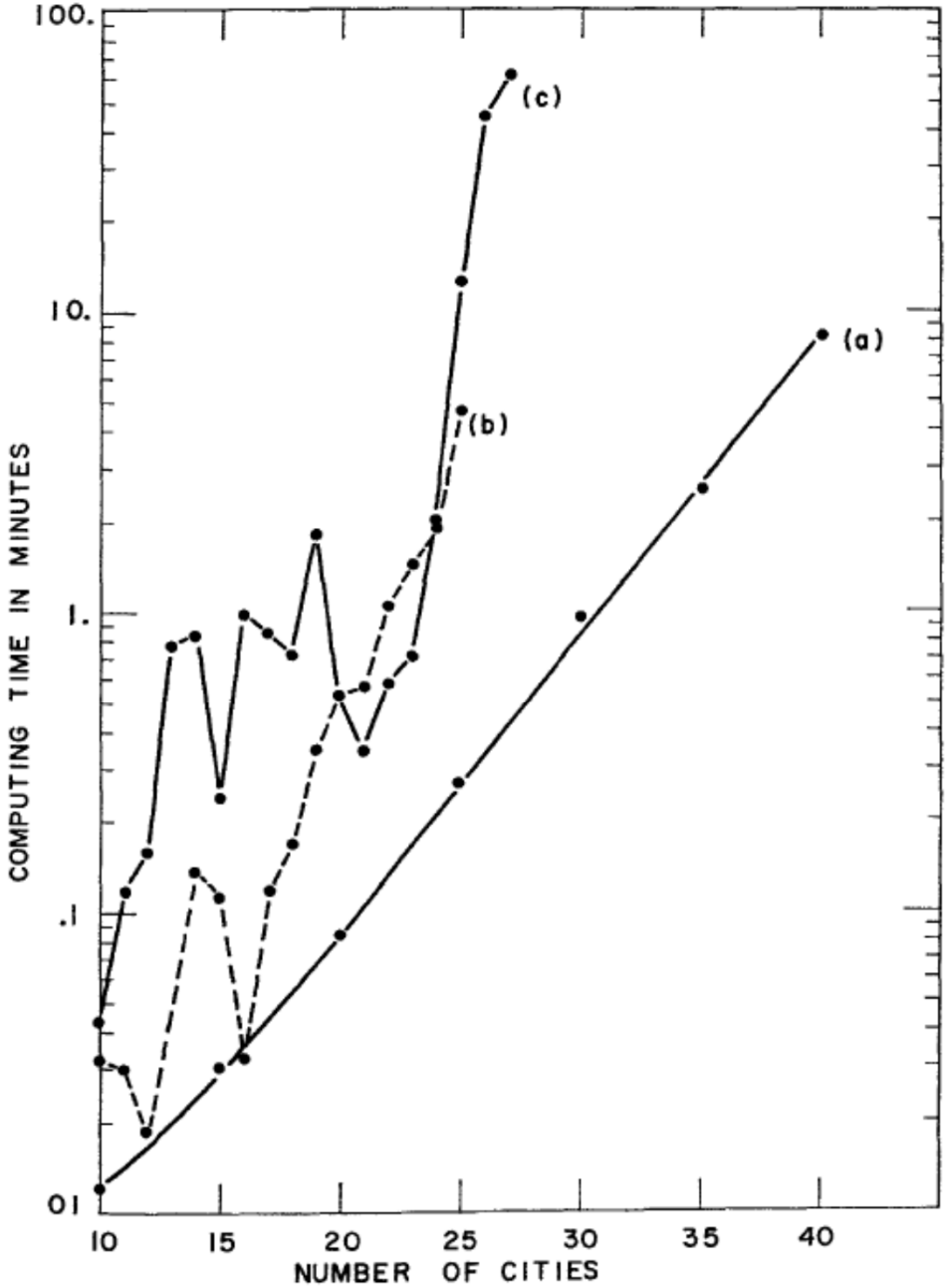


Fig 7. Computing times on IBM 7090 (a) Average times for 3 digit random number distance matrices (b) Subproblems derived from Held and Karp's 25-city problem (c) Subproblems derived from Held and Karp's 48-city problem

Conclusions

- Clearly, the average computational time grows exponentially with increasing problem size
- As a rule of thumb, adding 10 cities to the problem multiplies the necessary computational time by a factor of 10

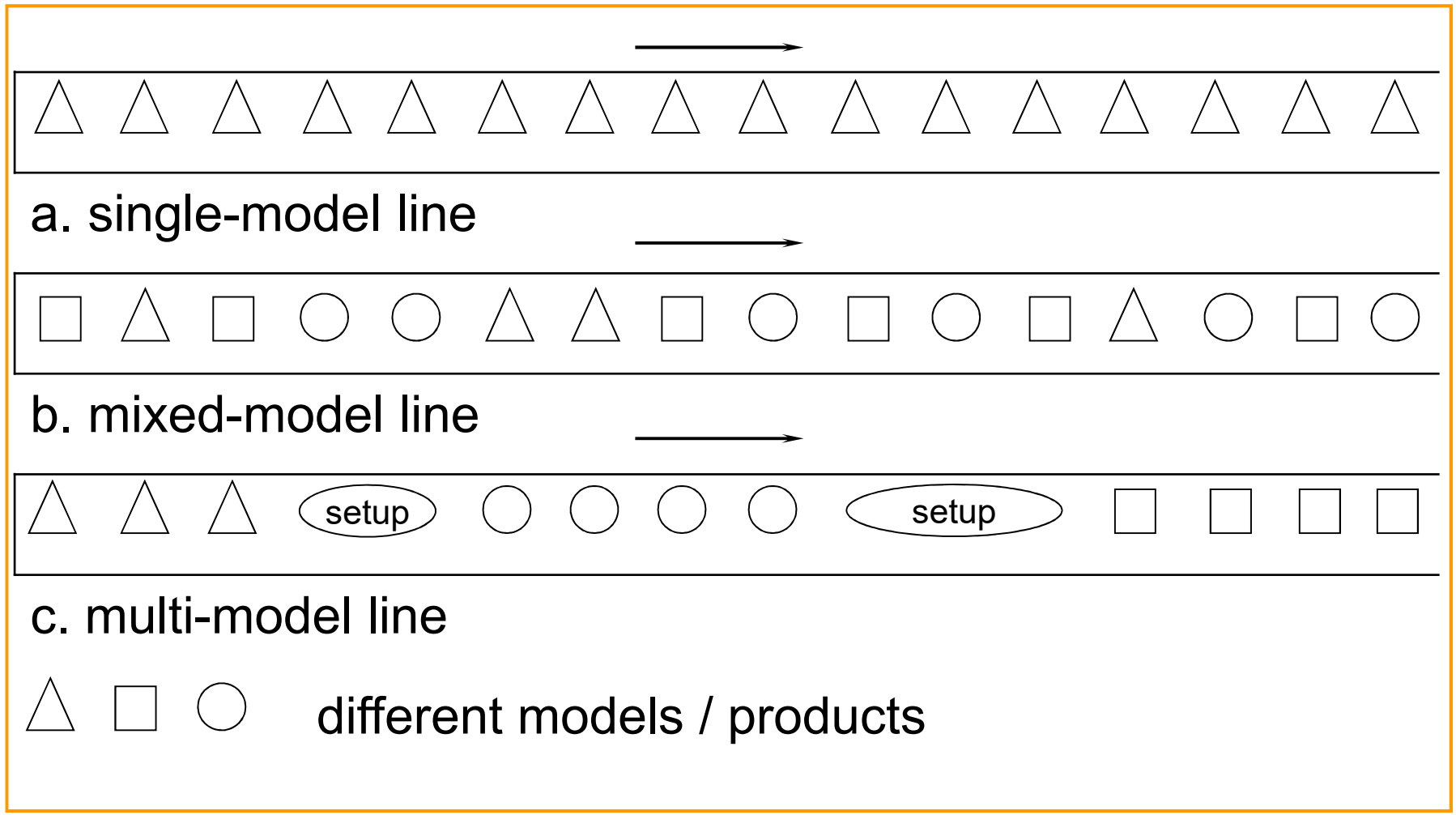
3.4 SALOME 1 – A B&B approach for SALBP-I

- In what follows, we introduce a considerably more sophisticated Branch&Bound procedure: the B&B procedure SALOME-1
- It solves SALBP-1 instances of moderate size to optimality in reasonable time
- SALBP-1 means “Simple Assembly Line Balancing Problem of Type 1”
- Therefore, it considers the balancing process of assembly lines
- Consequently, in what follows, we briefly introduce
  - the layout of assembly lines,
  - existing basic types of assembly lines, and finally,
  - the two basic planning levels that have to be considered in order to efficiently use assembly lines.

3.4.1 Attributes of assembly lines

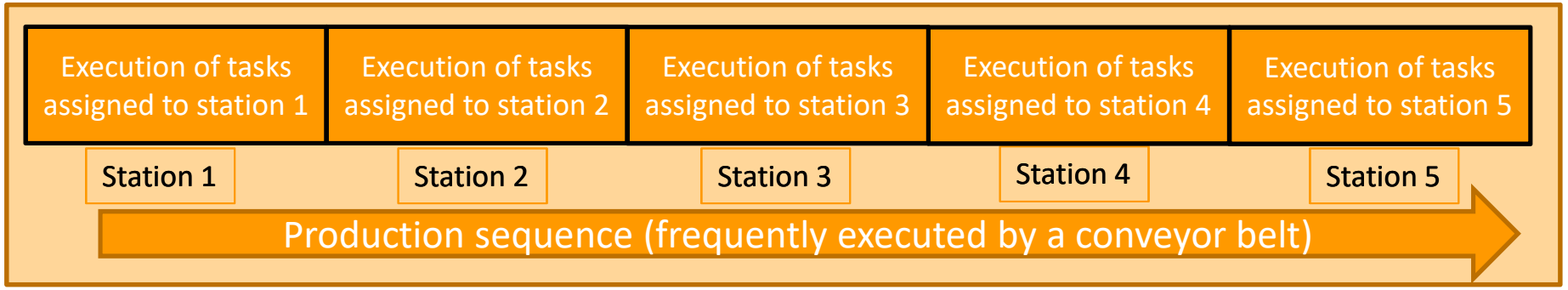
- Assembly lines arrange facilities solely according to the flow principle, i.e., the production process determines the layout
- Assembly lines are production systems without buffers, i.e., no intermediate storage systems between the different stages
- There is a predetermined global cycle or takt time c, i.e., in each takt time interval, a new product item is launched at the line
- Originally, assembly lines were designed for the mass production of absolutely homogeneous products
- Here, these systems attain smallest variable costs for the following reasons
  - Minimization of the number of conducted transports
  - Maximum training of employed workers
    - Executing identical tasks in an extremely high frequency
    - Workers become highly skilled for the respective tasks

Types of assembly lines



Single-model assembly line

- Originally, assembly lines were designed for a single variant or product only
- If only a single variant has to be produced on an assembly line, the production process (aside from potentially occurring disturbances) acts quite stationary in each station along the elapse of the different takts
- Specifically, the work load in each station stays constant from product unit to product unit
- At each station, the same set of tasks is performed for each product unit



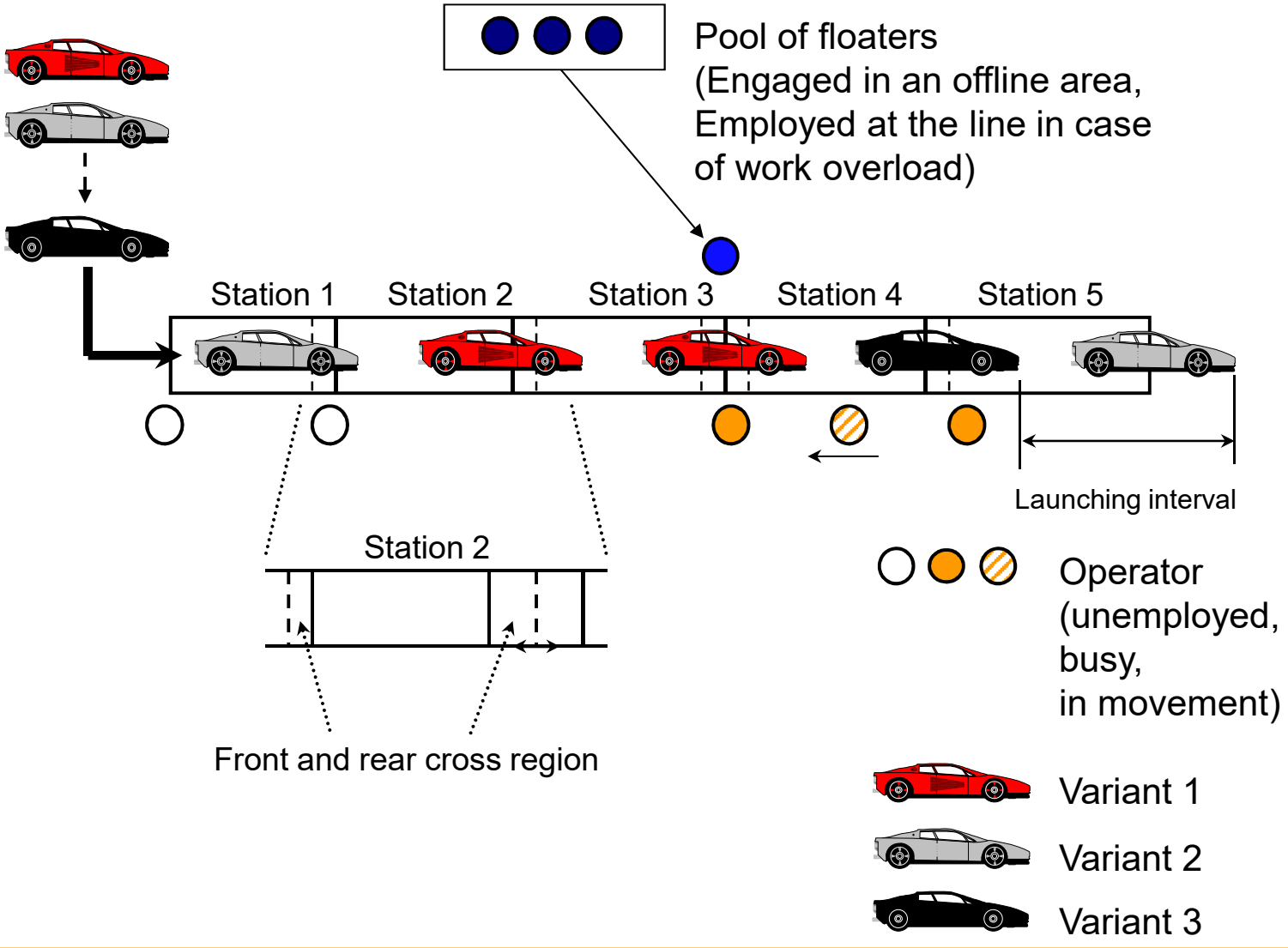
Balancing process

- The assignment of the tasks to the stations (balancing of the line) has substantial impact on the overall performance of the line
- The maximum workload of all stations determines the minimum feasible takt time of the single-model assembly line
- The takt time defines the production speed of the assembly line
- Moreover, the number of stations at the line corresponds to the resulting investment costs that have to be spent for erecting the assembly line

Mixed-model assembly lines

- In times of Mass Customization, customers demand to order an individual variant (of a mass product)
- However, if there are various variants to be produced at the same assembly line, the production process becomes more complex
- When we produce various variants, the work load in each station may change from takt to takt
- Therefore, in order to allow an efficient use of capacities, stations have to become more adaptable in size and structure
- For this purpose, the layout of mixed-model assembly lines integrates (among others) for instance
  - overlapping areas between stations
  - different kinds of workers (floaters and operators)
  - offline areas for not feasibly completed product units

Mixed-model assembly line



Basic terms

- **Assembly**
  - Process of collecting and fitting together various parts
  - A finished product is created
  - Relationship of parts and the flow of material can be visualized by assembly charts (Gozinto charts)
  - Workpieces are unfinished units of the product
- **Operation or task**
  - Is a portion of the total work content in an assembly process
  - A task time is predetermined for each operation
  - Indivisible operations, i.e., task cannot be divided
- **Station**
  - In a station, a certain amount of work (some tasks) is performed
  - Workers and floater are assigned firmly or loosely, respectively

Basic terms

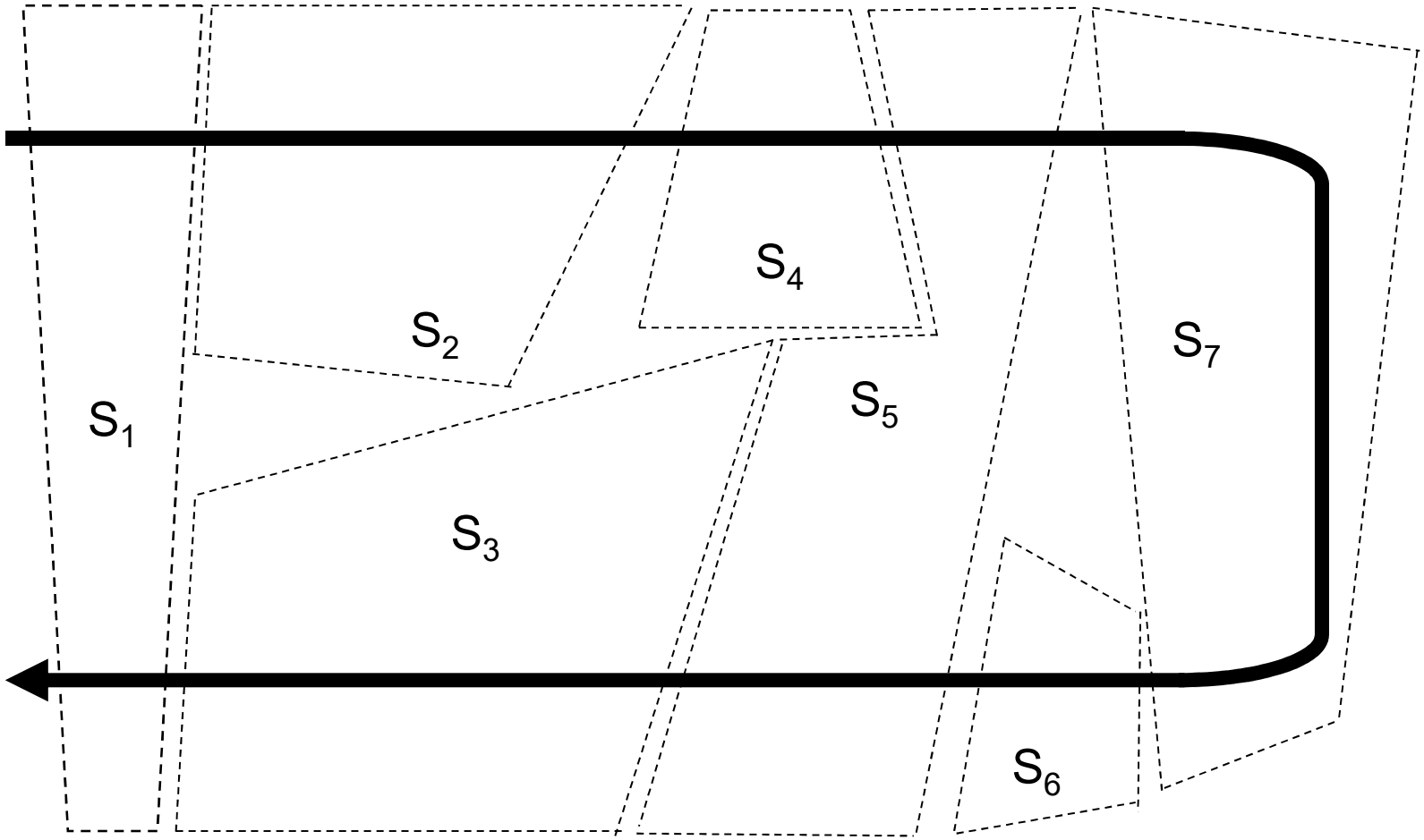
- **Cycle time or Takt time**
  - Maximal amount of time a workpiece is allowed to consume during its processing in a station
  - Equals the launching time in the production process
  - Consequently, cycle time cannot be smaller than the maximal duration of a task to be executed at the line

- **Precedence constraints**
  - Owing to technological restrictions, the ordering in which operations are performed at the line may be partially predetermined
  - Illustration may be given by so-called precedence graphs

Layout of assembly lines

- Serial lines
  - Traditional layout of assembly lines
  - I.e., single stations are arranged in a straight line along the line
- U-shaped lines
  - The layout of U-lines is defined according to this letter
  - I.e., stations contain tasks coming from different points of the production process
- Parallel lines
  - Here, lines are arranged parallel to each other
  - Thus, items may be produced on each line alternatively
  - Consequently, a more flexible production system is established
  - Specifically, product items, whose successive production may cause substantial work overload, may be produced on different lines

Illustration – U-shaped assembly line



Paced and unpaced assembly lines

- Paced assembly lines
  - Either the workpieces are steadily moved from station to station by a conveyor belt at constant speed, or
  - Workpieces are intermittently transferred to the subsequent station after being processed
  - Stations are coupled in an inflexible way
  - I.e., there are no intermediate buffers between stations
  - Inflexible structure, but accelerates the production
- Unpaced assembly lines
  - Here, stations are decoupled by intermediate buffers
  - I.e., workpieces are hold if the subsequent station is still busy
  - Blocking occurs if intermediate buffers are filled to capacity
  - More flexible structure
  - Allows setup activities

Types of stations

- Parallel stations
  - Resources are installed several times at the same station
  - I.e., different work pieces are processed simultaneously
  - Consequently, by making use of parallel stations, better or feasible constellations may become possible
  - For instance, if there are tasks with extremely large processing times (>takt time), parallel stations become necessary
  - However, the installation of parallel stations is costly
    - Specific arrangements at the conveyor belt are necessary
    - Multiple investments in resources

Open and closed stations

- Closed stations
  - The extent of closed stations is predetermined, i.e., fixed throughout the production process
  - Consequently, there are strict boundaries not crossable by the deployed workers or floaters
- Open stations
  - Here, boundaries may be crossed by personnel in order to complete a currently processed workpiece
  - But, workers crossing the boundaries do usually not interfere with each other
  - Specifically, it has to be distinguished between
    - Open-to-the-right stations
    - Open-to-the-left stations

Observations

- Open-to-the-right stations
  - Positive effect: Items can be completed in time
  - Negative effect: Less time available for the subsequent item in this station
  - I.e., Timely completion versus worker drift

- Open-to-the-left stations
  - No tradeoff
  - Only positive effects
  - Necessary consideration of technical restrictions (e.g., wire length)

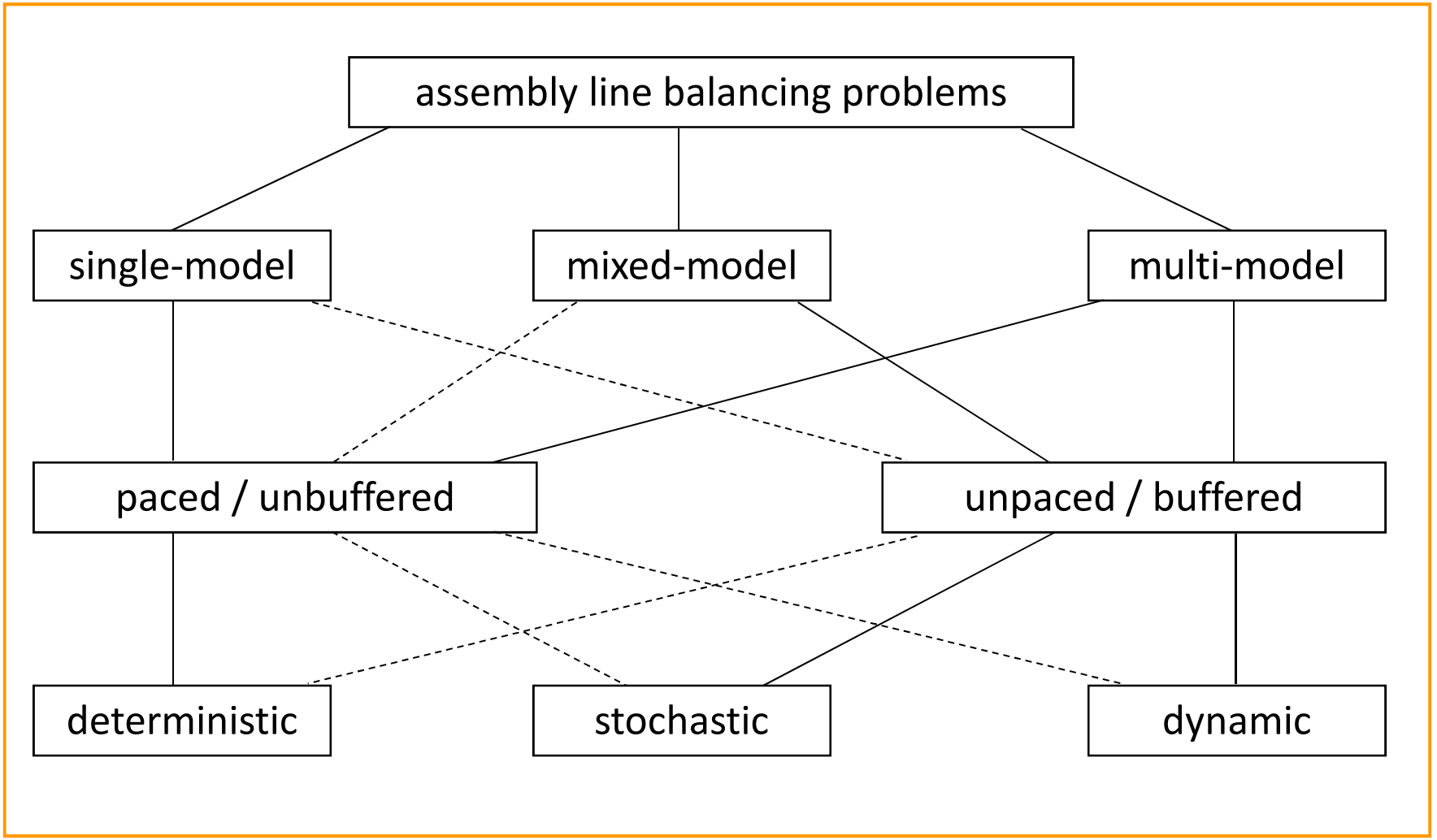
3.4.2 Assembly Line Balancing

- Determination of the line layout and capacities
  - Structure of the lines
  - Assignment of tasks, machines and personnel resources
- Balancing problems:
  - Assembly lines are inflexible
  - Small adaptation range for the production control
- Consequence:

  Line layout must be robust enough in order to smooth the varying capacity demand

➢ **Anticipation of possible scenarios and constellations in order to evaluate a generated assembly line layout**

Classification of balancing problems



3.4.3 Simple Assembly Line Balancing

Assumptions of the model:

- There are *N* predetermined tasks (set *V*) to be performed in order to produce the single product
- Each task j∈V has a predetermined processing time t_j
- There are precedence constraints between the different tasks resulting from technological reasons and defined by the precedence graph G=(V,E)
- All stations are uniformly equipped, may be able to perform all tasks (if assigned), and possess the uniform time capacity C
- No buffers between stations exist. Therefore, products are directly or continuously transported from station to station (potentially by a conveyor belt)

SALBP-F model – Mathematical definition

Feasibility variant of SALBP:

- N: Number of tasks to be executed at the assembly line
- C: Takt time of the assembly line
- M: Number of stations
- t_i: Processing time of task i
- Binary variable $x_{i,s}$=1 if and only if task i is assigned to station s, otherwise =0

Restrictions:

$$(1)\quad \text{Feasible task assignment: } \forall i \in \{1,\ldots,N\} : \sum_{s=1}^{M} x_{i,s} = 1$$

SALBP-F model – Mathematical definition

$$(2)\ \text{Takt time restriction: } \forall s \in \{1,\ldots,M\} : \sum_{i=1}^{N} x_{i,s} \cdot t_i \leq C$$

$$(3)\ \text{Precendence constraints: } \forall (i,j) \in E : \sum_{s=1}^{M} x_{i,s} \cdot s - \sum_{s=1}^{M} x_{j,s} \cdot s \leq 0$$

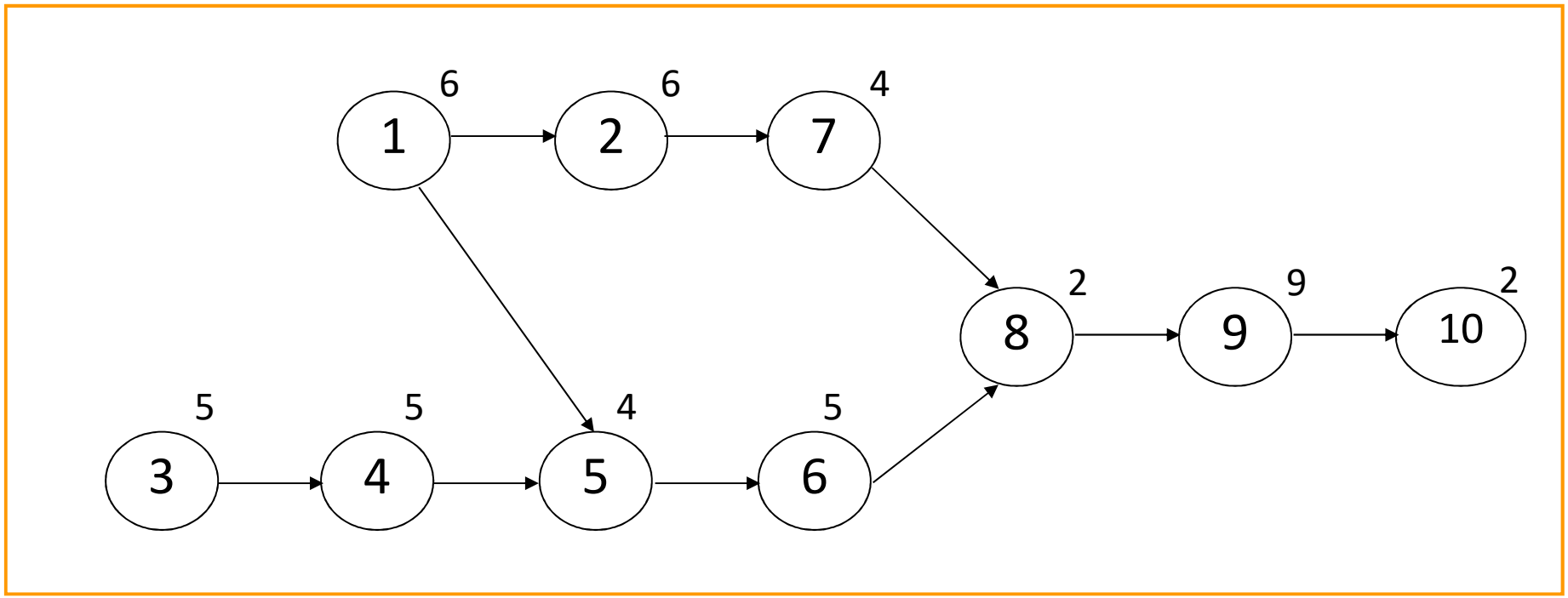
$$(4)\ \text{Domain constraint: } \forall i \in \{1,\ldots,N\} : \forall s \in \{1,\ldots,M\} : x_{i,s} \in \{0,1\}$$

- Unfortunately, already the feasibility variant of SALBP is NP-hard
- Therefore, all optimization variants, which are considered in what follows, are also NP-hard

3.4.3 SALBP – Optimization models

- In what follows, we distinguish between three different optimization variants altogether
- Depending on the objective function, we obtain the models
  - **SALBP-1: Minimizing the number of necessary stations *M*, i.e., minimizing the necessary investments**
  - SALBP-2: Minimizing the takt time *C*, i.e., maximizing the output rate *1/C*
  - SALBP-E: Maximizing the efficiency, i.e., minimizing the product $C \cdot M$

SALBP 1 example – precedence graph



See Scholl (1999), p.118

Basic terms

- Line Balance
  - Found feasible task assignment, i.e., an assignment of the given tasks to the stations of the line
  - It is feasible if all precedence constraints are fulfilled
- In our example, we obtain determining cycle time 11 the following line balance
  - Station 1: 1,3; i.e., resulting idle time amounts to 0
  - Station 2: 2,4; i.e., resulting idle time amounts to 0
  - Station 3: 5,6; i.e., resulting idle time amounts to 2
  - Station 4: 7,8; i.e., resulting idle time amounts to 5
  - Station 5: 9,10; i.e., resulting idle time amounts to 0

Exact solution approaches for SALBP-1

- Besides some heuristics, the literature provides various exact solution approaches that guarantee an optimal solution
- The most famous ones are
  - **FABLE (Johnson(1988))**
  - **OPTPACK (Nourie and Venta (1991))**
  - **EUREKA (Hoffmann(1992))**
  - **SALOME (Scholl and Klein(1997))**
  - SPEZSAL (sequential and, in particular, parallel version) (Bock (2000))

FABLE

- Proposed by Johnson in 1988
- It is a depth-first-search Branch&Bound approach
- It works task-oriented, i.e., in each node, a single task is appended to a current partial solution with a last opened station k
  - If an assignable task exists, it is appended, otherwise, the station is closed and station k+1 is opened
  - Before starting, tasks are renumbered such that
    - no task has a larger numbered predecessor
    - if two tasks are not successors or predecessors to each other, the task with the larger processing time gets a smaller number
  - After assigning a task to a station, only larger numbered tasks can be assigned to this station as well (avoiding the repeated enumeration of identical sets of tasks in one station)
- After completing a solution or fathoming a solution the procedure tracks back, i.e., it tests an alternative task

FABLE

- FABLE applies various bounding rules. Specifically, it applies LB1, LB2, LB3, and LB4 (all these rules are introduced later)
- Furthermore, FABLE applies the following logical tests
  - Maximum load rule
  - Jackson dominance rule
  - Labeling dominance rule (due to memory limitations, it becomes dynamically inactive when a larger unlabeled task is assigned)
  - First station dominance rule
  - Task time incrementing rule

EUREKA

- Works station-oriented, i.e., a node represents always a completed station load that is appended to a partial solution
- Forward and backward enumeration
  - First, for a specific time, a layout is searched in forward direction
  - Second, the same process is conducted in backward direction
  - If no solution is found within a given time limit, the heuristic originally proposed by Hoffmann (1963) is applied
- IDA*-Branch&Bound procedure that works with the simple idle time rule only (LB1, see below)
  - This rule derives the smallest number of stations M that would have a total idle time larger or equal to the idle time of the currently considered partial layout
  - Total idle time of a solution with M station is $M \cdot C - t^{total}$, while t^{total} gives the total processing time of all tasks

EUREKA

- No further logical rule or dominance criterion is applied during the enumeration process of this procedure

OPTPACK

- Proposed by Nourie and Venta in 1991
- Uses the branching scheme of FABLE
- Applies heuristics in order to find feasible upper bounds before starting the enumeration process
- Applies only the first LB (idle-time argument)
- Furthermore, OPTPACK applies the following two logical tests
  - Maximum load rule
  - Tree dominance rule

The Branch&Bound algorithm SALOME

- In what follows, we consider a Branch&Bound procedure that is specifically designed to tackle SALBP-1 instances
- A modified version of this approach, however, is proposed in order to solve SALBP-2 (this is denoted as SALOME-2)
- This approach is a combination of two previously known Branch&Bound algorithms (EUREKA and FABLE)
- SALOME is station-oriented, i.e., each node represents a complete load of a station that has been opened either in forward or in backward direction
- This results from the fact that after closing a station, the lower bound of a current solution is affected more substantially
- SALOME uses a Local Lower Bound Technique
- This will be depicted next

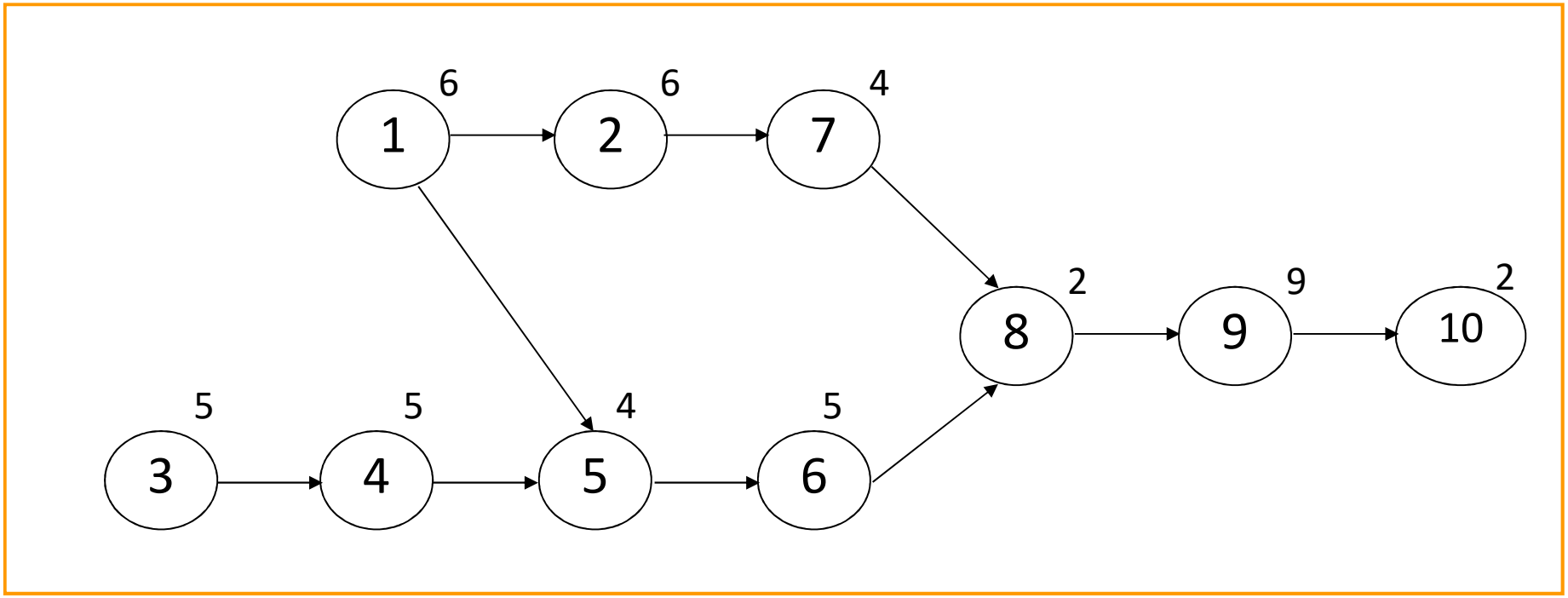
Local Lower Bound Technique (LLB)

- LLB is a mixture of IDA* and DFS (mixture of FABLE and EUREKA)
- I.e., there is only a local clustering of loads, but in order to avoid multiple explorations, LLB is allowed to be increased without starting from scratch again
- The bound is applied to each node, i.e., the root obtains the Global Lower Bound as the initial “Local Lower Bound” (LLB)
- It is increased after exploring all station loads that meet this bound
- LLB means that the examination process considers only station loads whose current lower bound is equal to LLB
- In subsequence of exploring all these loads, the LLB is increased to the minimum larger value of a load explored before

LLB – simple example

- We again consider our simple example

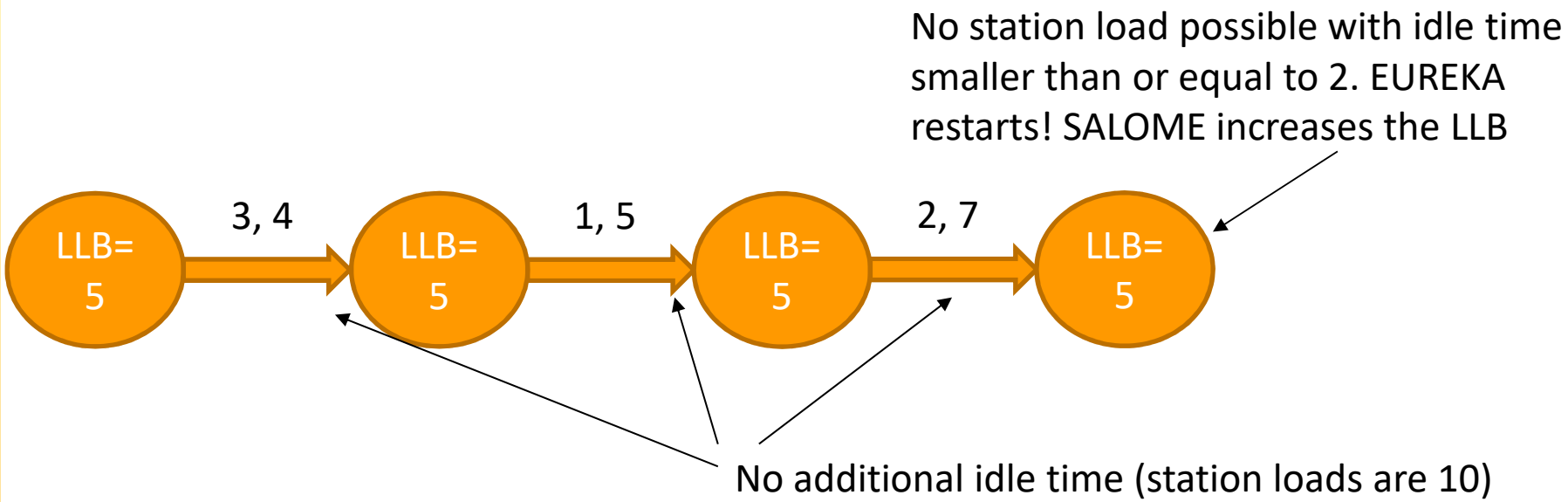
Predetermined takt time of the assembly line is C=10



See Scholl (1999), p.118

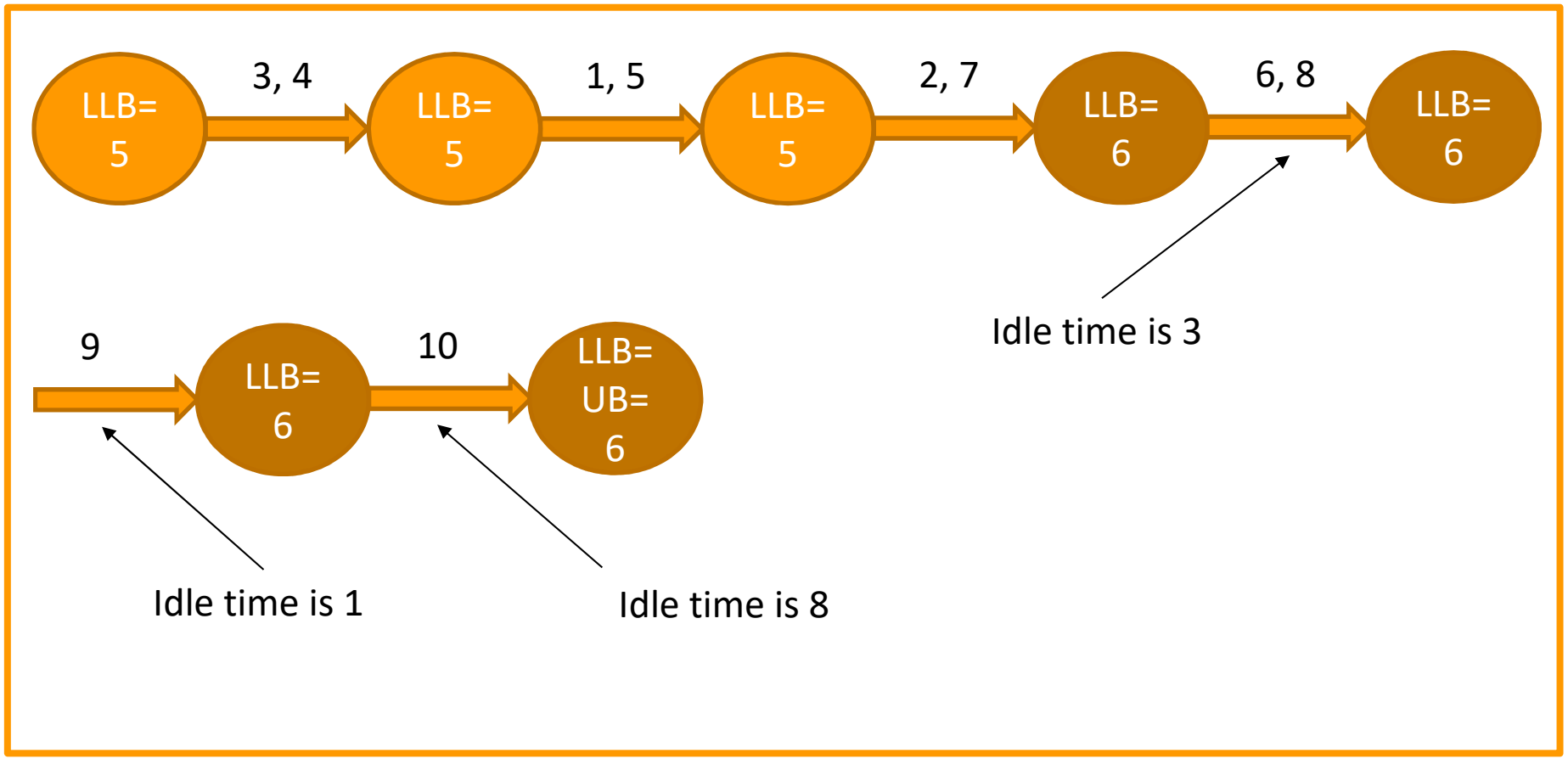
LLB – simple example

- We obtain a simple lower bound by adding all task processing times 6+6+5+5+4+5+4+2+9+2=48
- Thus, by dividing through C=10, we obtain at least 5 stations
- Hence, in order to attain a feasible solution, the total allowed delay time or idle time is 2
- Therefore, we start with LB=5



LLB – simple example

- Hence, SALOME resumes as follows:



LLB

- Acts still locally in a depth-first manner
- Best FS still acts much more globally
- However, best loads in a locality are enumerated first
- Avoidance of repeated enumerations of identical parts of the solution space

Generating a current station load

- In what follows, tasks are numbered according to the precedence graph, i.e., each task may posses lower numbered predecessors only
- I.e., in order to determine a station load, SALOME assigns tasks in ascending order only
- After assigning a task *x* in a previous step, only larger numbered tasks *y>x* may follow for a given currently considered partial station load
- Consequently, SALOME considers sets of tasks as station loads, but no sequences of tasks in these stations

Maximum Station Load Rule

- Clearly, if a station load is completed (and therefore the station under consideration is closed), it has to be checked whether it exists an available task *i* that is feasibly assignable, i.e.,
  - All predecessors of this task are already assigned and
  - the remaining processing time in the station is not smaller than the processing time of the task *i*
- If such a task exists, the rule allows to fathom the currently considered partial station load
- This rule is always applied in each node of the enumeration tree of SALOME

Dynamically selecting enumeration direction

- SALOME dynamically decides whether the subsequent station is opened either in forward or in backward direction
- This decision is based on the following rule

  Open the **next station in forward direction**

  - if $T(k_f)>T(k_b)$ applies, or
  - if it holds $T(k_f)=T(k_b)$ and not $|B(k_f)|>|B(k_b)|$
- The following abbreviations are used

$$T(k)=\frac{\sum_{j\in B_k}\left(\frac{t_j}{L_{j,LLB}-E_j+1}\right)}{|B_k|}$$

Additional abbreviations

- k_f: Index of subsequent station to be opened in forward direction
- k_b: Index of subsequent station to be opened in backward direction
- $L_{j,LLB}$: Index of station that is the latest where task j can be assigned to if LLB stations are maximally available
- E_j: Index of station that is the earliest where task j can be assigned to
- Note that L and E are calculated by heads and tails of the respective tasks (c.f., Lower Bound 4)
- B_k: Set of available tasks that can be theoretically assigned to station k (i.e., set of assignable tasks)

E_j and L_j

- Simple bounds for the earliest and latest stations a task can be assigned to are obtainable by

$$E_j := \left\lceil \frac{\left(t_j + \sum_{h \in P_j^*} t_h \right)}{C} \right\rceil \qquad \text{for } j = 1,\ldots,N$$

$$L_{j,M} := M + 1 - \left\lceil \frac{\left(t_j + \sum_{h \in F_j^*} t_h \right)}{C} \right\rceil \qquad \text{for } j = 1,\ldots,N$$

The idea behind the rule

- In order to minimize the computational effort in the examination process, we branch complex nodes as late as possible
- A node, i.e., a station, causes high computational effort if
  - a large number of tasks are assignable (i.e., large B values)
  - a smaller portion of task execution time is firmly assigned to the station in question (i.e., we have an increased computational effort due to a larger number of assignable tasks)
- Thus, stations are preferred if the expected computational effort is smaller

Preprocessing – Task incrementing rule

- Prior to the exploration process, the minimum processing time t_{min} of all tasks is generated
- All tasks i with $C-t_i<t_{min}$ obtain the maximum execution time C in order to avoid unnecessary computations

Logical rules

- After completing a load in the currently considered station, the following rules are applied
  - Maximum station load
    - A load is fathomed if an available task is still assignable, i.e., the remaining processing time in the station is large enough to assign this task
    - Consequently, the current load is dominated
  - Jackson dominance rule
    - In a preprocessing step, all tasks are considered according to a possible dominance
    - Task i potentially dominated task j if
      - all successors of j are also successors of i and
      - processing time of i is not smaller than the processing time of j or
      - if successors and processing times are identical, but it holds that i<j

Jackson dominance rule

- Let us consider a pair of tasks i and j that can be alternatively assigned to a current (partial) station load
- Assuming i potentially dominates j, while there is a current station load comprising task j but not i
- That means, task i is available and can be assigned to the current station instead of j, i.e., the remaining available time capacity of the station is sufficient to assign task i
- Then, this load can be fathomed since it is dominated by the station load arising by replacing j with i
- Note that this is true due to the fact that the set of tasks that become assignable after this assignment is not reduced since all successors of task j are also successors of task i

Modified task numbering

- All tasks are renumbered in order to obtain complete solutions earlier
- This is done according to the following rules
  - if a task potentially dominates another task, it gets a smaller number
  - if both tasks are not comparable, the one with more successors is preferred in the new numbering
  - Finally, ties are broken by lower initial task numbers
- Consequently, all tasks are renumbered according to the rules listed above. Subsequently, the exploration process starts

Dynamic Prefixing

- Reduction rule that makes directly use of the concept of earliest and latest stations
- Specifically, if a station k is considered with current Local Lower Bound LLB, all tasks i with $L_i(LLB)=k$ are directly assigned to it
- Since the given LLB should be reached, these tasks have to be directly assigned to station k
- This reduces the complexity of the subsequent enumeration

Tree dominance rule

- Extension of the well-known **labeling dominance rule** originally proposed by Johnson in his algorithm FABLE
- The idea behind labeling dominance rule
  - Each partial solution is an assignment of tasks to stations
  - Such a constellation can be unambiguously identified by the labeling scheme of Schrage and Baker (1978)
  - Task labels L(j) are calculated as follows

$$L(j) := \sum_{k \in \left\{ h \,\middle|\, h < j \wedge h \notin P_j^* \right\}} L(k) + 1,$$

Labeling dominance rule

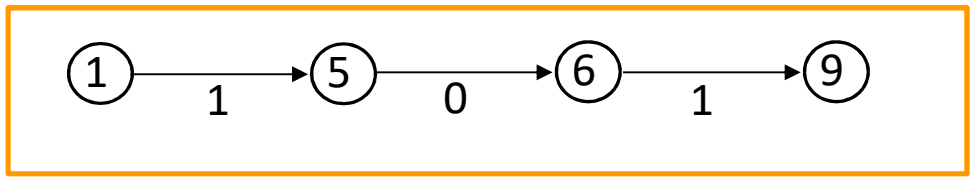
- Uniqueness of addresses follows directly from the following observations
- Consider a task h without predecessor
- Its label l(h) is larger than all label combinations (i.e., the sum of these labels) of lower numbered tasks
- Therefore, this value is unambiguously defined
- Note that predecessors of a task are out of interest since – due to the precedence constraints – this assignment sequence is predetermined
- In a worst case scenario, there are no precedence constraints
- Here, the task n=1,…,N obtains the value 2^{n-1}

Labeling dominance rule

- Hence, each generated partial solution obtains a sum of labels that unambiguously defines the set of tasks already assigned to some station
- This sum of labels unambiguously identifies a partial solution and gets a minimum number of stations (not necessarily integral) that was reached up to this point of time
- If the same partial solution is generated later without improving this value, the current constellation is fathomed
- This leads to considerable complexity reductions but requires extreme memory consumptions in worst case scenarios
- However, this considerable memory consumption can be significantly reduced by integrating this rule into a specifically designed hash-list based data structure

Tree dominance rule

- Partial solutions are stored as binary out trees
- Nodes are tasks being currently assigned or not
- Arcs represent assignments of sets of tasks
  - Nodes are inserted in ascending order
  - Outgoing arcs represent the state of the respective task and all larger numbered ones up to the next task, i.e., there is an interval of tasks whose current state is determined by the arc
  - E.g., if we have



    we actually got the assignment of tasks 1,2,3,4,6,7,8
- Each path starting at the root node, represents a partial solution
- I.e., the minimum number of stations used is stored in each terminal node of the complete tree

Observations

- As empirically shown by Nourie and Venta (1991), the tree labeling technique consumes significantly less storage than the labeling scheme proposed by Schrage and Baker (1978)
- Its application considerably reduces the size of the enumeration tree

SALBP-notations

In what follows, we make use of the following abbreviations

N	Number of tasks
V	set of all tasks $\left(=\{1,...,N\}\right)$
j	index for the tasks $\left(j=1,...,N\right)$
C	Cycle time (=Takt time)
p	production rate $\left(=\frac{1}{C}\right)$
M	Number of stations
k	index for the stations $\left(k=1,...,M\right)$
t_j	operation time (task time) of task j
p_j	station requirement of task j $\left(=\frac{t_j}{C}=t_j \cdot p\right)$

SALBP-notations II

t_{max} maximal task time $\left(= \max\left\{t_j \mid j = 1,\ldots,N\right\}\right)$

t_{min} minimal task time $\left(= \min\left\{t_j \mid j = 1,\ldots,N\right\}\right)$

t_{sum} sum of task times $\left(= \sum_j t_j\right)$

P_j set of immediate predecessors of task j

F_j set of immediate successors (followers) of task j

A set of direct precedence relations $\left(= \left\{(i,j) \mid i \in V \text{ and } j \in F_j\right\}\right)$

P_j^* set of immediate and transitive predecessors of task j

F_j^* set of immediate and transitive successors (followers) of task j

A^* set of all precedence relations $\left(= \left\{(i,j) \mid i \in V \text{ and } j \in F_j^*\right\}\right)$

SALBP-notations III

S_k station load, set of tasks assigned to station k

$t(S_k)$ station time of station $k \left(= \sum_{j \in S_k} t_j \right)$

c_r Implemented (operating) cycle time $\left(= \max \left\{ t(S_k) \mid k = 1, \ldots, M \right\} \right)$

E_j earliest station of task j

L_j latest station of task j

Applied lower bounds

- In order to obtain lower bounds during the exploration process, altogether seven different bounds are applied
- In what follows, we introduce all these bounds separately
- In SALOME, all these bounds are applied
  - In the root node all seven bounds are calculated and applied
  - During the computations, however, owing to its complexity, bound 4 is not adjusted
  - In each node, the maximum value potentially corrects the current LLB

Bound LB_1

- Most simple bound
- It bases solely on the cognition that the total workload has to be divided into pieces of size C
- Sum of operating times has to be shared between stations
- Consequently, we need at least the following number of stations

$$LB_1 := \lceil t_{sum} / C \rceil = \left\lceil \sum_{j=1}^{N} p_j \right\rceil \qquad (1)$$

Bound LB_2

- Basic idea results from the cognition that there are tasks that cannot be combined in a single station
- Much more, they have to be assigned to different stations
- Bound 2 commences this examination by identifying tasks whose processing time is larger than half of the takt time C
- These tasks belong to the set J(1/2,1]
- Since all these tasks need a station of their own, we can derive the following lower bound

$$LB_2 := \left| J\left(\frac{1}{2},1\right] \right| + \left\lceil \frac{1}{2} \cdot \left| J\left[\frac{1}{2},\frac{1}{2}\right] \right| \right\rceil \qquad (2)$$

Bound LB_3

- The basic principle of Lower Bound 2 is extended in the third lower bound
- Here, task time is clustered into three groups instead of only two
- This is done by building the sets
  - J(1/3,2/3]
  - J(2/3,1]
- We can define the Lower Bound 3 by

$$LB_3 := \left\lceil \left| J\left(\frac{2}{3},1\right] \right| + \frac{2}{3}\cdot\left| J\left[\frac{2}{3},\frac{2}{3}\right] \right| + \frac{1}{2}\cdot\left| J\left(\frac{1}{3},\frac{2}{3}\right) \right| + \frac{1}{3}\cdot\left| J\left[\frac{1}{3},\frac{1}{3}\right] \right| \right\rceil \quad (3)$$

Bound LB_4

- The Assembly Line Balancing Problem can be interpreted as a single-stage scheduling problem
- Assigning a task k to a station is interpreted as to schedule the job k with processing time t_k on a considered single machine
- After being executed, there are several tasks coming behind the job
- Therefore, these tasks (or jobs) define a job-dependent tail n_x for a considered job x that has to be processed subsequently
- By deciding for a sequence $h_1, h_2, \ldots, h_N$, we obtain the following total makespan

$$\max\left\{p_{h_1} + n_{h_1},\ p_{h_1} + p_{h_2} + n_{h_2}, \ldots, p_{h_1} + \ldots + p_{h_N} + n_{h_N}\right\} \qquad (4)$$

Bound LB_4

- The respective tails are calculated by making use of the bounds 1, 2, and 3
- In each new iteration, we adjust the tails of a considered task j according to the results achieved by the previous iteration
- In what follows, the set J denotes the set of tasks that are successors of the currently considered task

$$LB_1(J) := \sum_{h \in J} p_h \qquad (5)$$

$$LB_2(J) := \left| J\left(\frac{1}{2}, 1\right] \right| + \frac{1}{2} \cdot \left| J\left[\frac{1}{2}, \frac{1}{2}\right] \right| \qquad (6)$$

$$LB_3(J) := \left| J\left(\frac{2}{3}, 1\right] \right| + \frac{2}{3} \cdot \left| J\left[\frac{2}{3}, \frac{2}{3}\right] \right| + \frac{1}{2} \cdot \left| J\left(\frac{1}{3}, \frac{2}{3}\right) \right| + \frac{1}{3} \cdot \left| J\left[\frac{1}{3}, \frac{1}{3}\right] \right| \qquad (7)$$

Bound LB_4

- Specifically, we have to adjust the generated bounds slightly
- I.e., if the processing time of the task j to be assigned is too small, the tail has to be reduced accordingly
- This has to be ascribed to the fact that it may be possible that the task j can be additionally assigned to the last station that is closed during the computation of the respective tail
- Clearly, this is not the case for the *first LB*
- For the *second LB,* the mistake is at most 1/2 if t_j is smaller than C/2 or the bound is not an integer
- For the *third LB,* the mistake is at most 1/3 if t_j is smaller than $2/3 \cdot C$

Adjusting the second LB by 1/2

- We consider the formula

$$LB_2(J) := \left| J\left(\frac{1}{2}, 1\right] \right| + \frac{1}{2} \cdot \left| J\left[\frac{1}{2}, \frac{1}{2}\right] \right|$$

- In the first part of the formula we have a maximum slack time strictly less than ½
- However, task *i* as a predecessor does not fit in this gap if it holds that $p_i \geq ½$, otherwise we have to check whether $LB_2(J)$ is integer
  - If so, we have to subtract ½
  - If not, no problem occurs since, in this case it holds that

$$p_i + LB_2(J) \leq \lceil LB_2(J) \rceil, \text{ due to } p_i < \tfrac{1}{2} \text{ and } LB_2(J) + \tfrac{1}{2} = \lceil LB_2(J) \rceil$$

  - I.e., the assignment of task i would not open a new station

Adjusting the second LB by 1/2

- Therefore, if the $LB_2(J)$ is integer and it holds that p_i<½ the "closing" of all opened stations of the bound (weighting them with 1) may overestimate the station requirement by at most ½
- This results from

$$p_i + LB_2(J) > \lceil LB_2(J) \rceil, \text{ due to}$$
$$0 < p_i < \tfrac{1}{2} \text{ and } LB_2(J) = \lceil LB_2(J) \rceil$$

- Subtracting ½ eliminates the possible overestimation

Adjusting the third LB by 1/3

- We consider the formula

$$LB_3(J) := \left|J\left(\frac{2}{3},1\right]\right| + \frac{2}{3}\cdot\left|J\left[\frac{2}{3},\frac{2}{3}\right]\right| + \frac{1}{2}\cdot\left|J\left(\frac{1}{3},\frac{2}{3}\right)\right| + \frac{1}{3}\cdot\left|J\left[\frac{1}{3},\frac{1}{3}\right]\right|$$

- In all parts of the formula the maximum slack is bounded by 1/3
- However, if $p_i \geq 2/3$ it cannot be combined with the elements of the sets J(2/3,1] and J(1/3,2/3)
- If it holds $p_i = 2/3 - \epsilon$ we have

$$p_i + LB_3(J) > \lceil LB_3(J) \rceil, \text{ if it holds that } 0 < p_i = \tfrac{2}{3} - \varepsilon \text{ and}$$

$$J = \{j\}, \text{ with } p_j = \tfrac{1}{3} + \varepsilon \Rightarrow LB_3(J) = \tfrac{1}{2}, \text{ but we have}$$

$$p_i + LB_3(J) > \lceil LB_3(J) \rceil = 1. \text{ BUT: i and j fit in one station!}$$

Bound LB_4

Consequently, due to the cognitions derived above, we obtain the following formulas

$$n_{j1} := LB_1\left(F_j^*\right) \tag{8}$$

$$n_{j2} := \begin{cases} LB_2\left(F_j^*\right) & \text{if } p_j \geq 1/2 \text{ or } LB_2\left(F_j^*\right) \notin IN \\ LB_2\left(F_j^*\right) - 1/2 & \text{otherwise} \end{cases} \tag{9}$$

$$n_{j3} := \begin{cases} LB_3\left(F_j^*\right) & \text{if } p_j \geq 2/3 \\ LB_3\left(F_j^*\right) - 1/3 & \text{otherwise} \end{cases} \tag{10}$$

In this computations, F_j^* defines the set of (direct and indirect) successors of task j in the precedence graph

Bound LB_4

- In an optimal solution (that minimizes the total makespan of the schedule), we schedule all jobs in sequence of non-increasing tails
- I.e., this minimal makespan, and therefore a valid lower bound for SALBP-1, is generated by

$$n_{j4} := \max\left\{p_{h_1} + n_{h_1}, p_{h_1} + p_{h_2} + n_{h_2}, ..., p_{h_1} + p_{h_2} + ... + p_{h_r} + n_{h_r}\right\} \quad (11)$$

Since n_{j1} through n_{j4} provide lower bounds on the station requirement of F_j^*, the tail of task j can be determined by

$$n_j := \max\left\{n_{j1}, n_{j2}, n_{j3}, n_{j4}\right\} \quad (12)$$

Bound LB_4

Procedure tails of tasks

for $j := N$ downto 0 do

if $F_j = \phi$ then $n_j := 0$

else begin

sort tasks of F_j^* according to non-increasing tails to obtain the list $[h_1, h_2, ..., h_r]$ with $r := |F_j^*|$ elements;

compute n_{j1}, n_{j2}, n_{j3}, and n_{j4} by applying (8) through (11);

$n_j := \max\{n_{j1}, n_{j2}, n_{j3}, n_{j4}\}$;

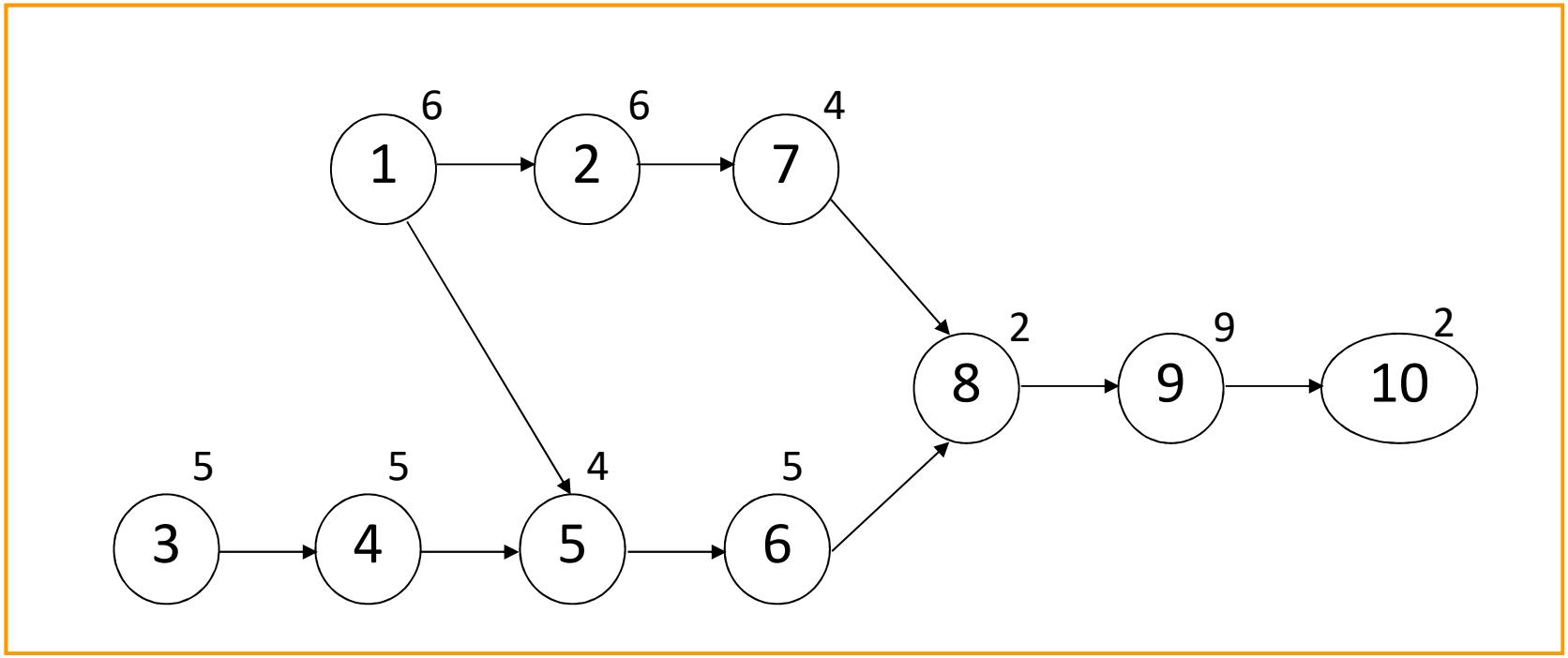
if $n_j < \lceil n_j \rceil$ and $p_j + n_j > \lceil n_j \rceil$ then $n_j := \lceil n_j \rceil$

end;

$LB_4 := \lceil n_0 \rceil$;

Example

- We consider again the following simple example
- The takt time is C=10



See Scholl (1999), p.118

Bound LB_4

j	10	9	8	7	6	5	4	3	2	1	0
p_j	0.2	0.9	0.2	0.4	0.5	0.4	0.5	0.5	0.6	0.6	0
n_{j1}	0	0.2	1.1	1.3	1.3	1.8	2.2	2.7	1.7	3.2	4.8
n_{j2}	0	0	0.5	0.5	1	1.5	1.5	2	1	2.5	4.5
n_{j3}	0	0	0.67	0.67	0.67	1.17	1.67	2.17	1.17	2.67	4.17
n_{j4}	0	0.2	1.9	2.2	2.2	2.7	3.4	3.9	2.6	4.1	5.7
n_j	0	0.2	1.9	2.2	2.2	2.7	3.4	3.9	2.6	4.1	5.7
Rounded	0	1	2	2.2	2.2	3	3.4	4	3	4.1	(6)

Bound LB_4

We explain the computation of n_{14} and n_{04}. Since the successors of task 1 build the ordered list $[2,5,6,7,8,9,10]$, we obtain

$$n_{14}:=\max\{0.6+3,1+3,1.5+2.2,1.9+2.2,2.1+2,3+1,3.2+0\}=4.1.$$

Owing to the ordered list $[1,3,4,2,5,6,7,8,9,10]$ of all following tasks, the tail of task 0 is

$$n_{04}:=\max\{0.6+4.1,1.1+4,1.6+3.4,2.2+3,2.6+3,3.1+2.2,3.5+2.2,3.7+2,$$
$$4.6+1,4.8+0\}=5.7.$$

By rounding up n_0, we obtain the lower bound $LB_4=6$ for the entire problem

Bound LB_4

- Besides the computed tails, we obtain also heads, i.e., station required by preceding tasks
- Consequently, we can derive heads analogously by applying the above computation in reverse sequence
- Eventually, we obtain the bound

$$LB_4^1 := \max\left\{\lceil a_{n+1} \rceil, \lceil n_0 \rceil, \lceil Z \rceil\right\} \qquad (13),$$

with

$$Z = \max\left\{a_j + p_j + n_j \,\middle|\, 1 \le j \le N\right\}$$

In order to derive a_{n+1}, we consider all predecessors of a considered task j.

These tasks are in the set P_j^*

Bound LB_4 – Calculating the heads

- Heads are generated analogously in reverse order, i.e., starting at the beginning and interpreting each arc in opposite direction
- Doing so, we obtain

j	0	1	2	3	4	5	6	7	8	9	10	11
p_j	0	0.6	0.6	0.5	0.5	0.4	0.5	0.4	0.2	0.9	0.2	0
a_j	-	0	1	0	0.5	1.6	2	1.6	3.5	4	5	5.2
n_j	5.7	4.1	3	4	3.4	3	2.2	2.2	2	1	0	-

Considering heads and tails simultaneously

- Unfortunately, this improved bound leads to a new scheduling problem known to be NP-hard in the strong sense
- The algorithm of Carlier (1982) solves it to optimality
- However, this worst case complexity may be exponential, and therefore may be too time consuming

Bound LB_5

- The idea behind this bound deals with head and tails again
- Specifically, a minimum number of stations can be derived by the fact that the earliest station where a task is assignable to must be lower or equal to the latest station
- Therefore, we obtain

$$LB_5 := \min\left\{M \;\middle|\; L_j(M) \geq E_j \;\; \forall\, j = 1,\ldots,N\right\} \qquad (14)$$

Bound LB_5 – applied to the example

j	1	2	3	4	5	6	7	8	9	10
t_j	6	6	5	5	4	5	4	2	9	2
E_j	1	2	1	1	2	3	2	4	5	5
$L_j(5)$	2	3	2	3	3	4	4	4	4	5
$L_j(6)$	3	4	3	4	4	5	5	5	5	6

- Complexity of the bounds depends on the applied method to generate heads and tails
- In order to obtain the results listed above, we have applied the following computation:

$$E_j := \left\lceil \frac{\left(t_j + \sum_{h \in P_j^*} t_h \right)}{C} \right\rceil \quad \wedge \quad L_j(M) := M + 1 - \left\lceil \frac{\left(t_j + \sum_{h \in F_j^*} t_h \right)}{C} \right\rceil$$

Using improved bounds for LB_5

- Clearly, the fifth bound can be significantly tightened by making use of improved head and tail computations
- This is done by using the fourth bound instead of LB_1
- Generally, we obtain the modified computations for head and tail, respectively

$$E_j' := \lceil a_j + p_j \rceil \qquad \text{for } j = 1,\ldots,N \quad (15)$$

$$L_j'(M) := M + 1 - \lceil p_j + n_j \rceil \qquad \text{for } j = 1,\ldots,N \quad (16)$$

Bound LB_5

- Consequently, we obtain (improved) values for earliest and latest stations
- In our example, we obtain (see Scholl (1999) p.49):

j	1	2	3	4	5	6	7	8	9	10
E_j	1	2	1	1	2	3	2	4	5	6
$L_j(6)$	2	3	2	3	3	4	4	4	5	6

Procedure for computing LB_6

Step 1. Assign the tasks in $J\left(\frac{1}{2},1\right]$ to the stations $1,\ldots,d_1 := \left|J\left(\frac{1}{2},1\right]\right|$ in the order of non-increasing operation times.

Step 2. Consider the tasks in $J\left(\frac{1}{3},\frac{1}{2}\right]$ in the order of non-decreasing operation times. Successively, assign these tasks to the earliest of the first d_1 stations which shows enough idle time until all tasks are assigned or none of the d_1 stations remains.

Procedure LB_6

Step 3. Let d be the number of tasks not assigned in Step 2. Then, at least

$$d_2 := \left\lceil \frac{d}{2} \right\rceil$$ stations are additionally needed for these tasks.

Step 4. Compute a bound d_3 on the station requirement for the tasks in

$$J\left[0, \frac{1}{3}\right] :$$

$$d_3 := \max\left\{0, \max\left\{ d_3(q) \,\middle|\, q \in \left\{ p_j \,\middle|\, j \in J\left[0, \frac{1}{3}\right] \right\} \right\}\right\}$$

Result. $LB_6 := \lceil d_1 + d_2 + d_3 \rceil$

Bound LB_6

Step 4 considers all tasks with $p_j \leq 1/3$. For each value $q \in \left[0, 1/3\right]$ given by some p_j, a bound on the (additional) station requirement of $J\left[q, 1/3\right]$ arises by:

$$d_3(q) := \sum_{j \in J[q,1-q]} p_j - \left| J\left(\frac{1}{2}, 1-q\right] \right| - d_2, \qquad (17)$$

due to the equation $J\left[q, 1/3\right] = J[q,1-q] - J\left(\frac{1}{2}, 1-q\right] - J\left(\frac{1}{3}, \frac{1}{2}\right]$

Therefore, we can conclude: $J[q,1-q] = J\left[q, 1/3\right] + J\left(\frac{1}{2}, 1-q\right] + J\left(\frac{1}{3}, \frac{1}{2}\right]$.

Bound LB_6 – additional observations

- *Step 3* computes a lower bound on the station requirement of those tasks that are not assigned in step 2.
- Clearly, at most two of them may share a station
- *Step 4* finally considers the smaller tasks of set J[0,1/3]. Since the station requirement of these smaller tasks is finally added, a repeated adding is avoided by subtracting the results of previous steps

Bound LB_7

- This bound exploits the strong relationship between SALBP-1 and SALBP-2
- Clearly, if we reduce the number of allowed stations *M,* the possible minimum takt time *C* may be increased
- Therefore, the idea is to lower bound the number of stations by applying the predetermined takt time as an upper bound that has to be guaranteed by the number of implemented stations
- For this purpose, the following lower bound of the takt time is applied:
  - If we have M stations and at least M+1 tasks, at least the total processing time of the two smallest ones provides us with a lower bound of the resulting takt time
  - Therefore, all M+1 tasks are sorted and renumbered in sequence of non-increasing processing times

Bound LB_7

- This idea is generalized by the following lower bound
- Specifically, the total processing time of the *k* smallest tasks is calculated if (k-1)·M+1 tasks have to be assigned

$$C(M) := \max\left\{\sum_{i=0}^{k} t_{k\cdot M+1-i} \,\middle|\, k = 1,\ldots,\left\lfloor (N-1)/M \right\rfloor\right\} \qquad (18)$$

By means of (18), the bound LB_7 for SALBP-1 can be formulated as

$$LB_7 := \min\left\{M \,\middle|\, C(M) \le C\right\} \qquad (19)$$

Selected computational results

- In order to empirically prove the efficiency of the proposed Branch&Bound algorithm SALOME, Scholl (1999) presents and analyzes several test results
- For this purpose, already known as well as newly generated test instances are solved
- Maximum computation time is set to 500 seconds per experiment, i.e., calculations are instantly stopped by this time and best found solution is taken
- All in all, a combined data set of 269 instances is solved. Note that optimal solutions of altogether 263 instances are known

Tested approaches

- FABLE – for details see above
- EUREKA – for details see above
- OPTPACK – for details see above
- SALOME
  - Altogether four different versions are implemented
  - SL1 (not using prefixing, simple permutation rule, LB_5, LB_6, and LB_7)
  - SL2
  - BiSL (SL2 with bidirectional branching), and
  - BiSLt (BiSL with tree labeling)

Some results for the most complex data set

	Combined Data Set						
	OPT-PACK	**FABLE**	**EUREKA**	**SL1**	**SL2**	**BiSL**	**BiSLt**
# opt	172	179	165	191	199	224	245
Av. rel. deviation	0.6	0.95	1.1	0.75	0.67	0.46	0.22
Max. rel. deviation	7.69	7.69	14.29	7.89	7.69	7.69	7.69
Av. CPU time	188.8	175.9	226.4	153.9	138.1	98.6	56.1

Observations

- SALOME clearly outperforms the former approaches FABLE and EUREKA
- Bidirectional branching as well as using tree labeling is of significant importance
- Obviously, the more instruments SALOME uses, the better the results become

References of Section 3

- Bock, S.: Modelle und verteilte Algorithmen zur Planung getakteter Fließlinien. Gabler DUV, Wiesbaden, 2000.
- Bock, S.: Using Distributed Search Methods for Balancing Mixed-Model Assembly Lines in the Automotive Industry. OR Spectrum 30: 551-578, 2008.
- Hoffmann, T.R.: Assembly Line Balancing with a Precedence Matrix. Management Science 9: 551-562, 1963.
- Hoffmann, T.R.: EUREKA. A Hybrid System for Assembly Line Balancing. Management Science 38: 39-47, 1992.
- Hoffmann, T.R.: Response to Note on Microcomputer Performance of "FABLE" on Hoffmann's Data Sets. Management Science 39: 1192-1193, 1993.
- Johnson, R.V.: Optimally Balancing Large Assembly Lines with "FABLE". Management Science 34: 240-253, 1988.
- Land, A.H.; Doig, A.G.: An automatic method of solving discrete programming problems. In: Econometrica, 28 (3): 497–520, 1960. (doi:10.2307/1910129)
- Little, J.D.C; Murty, K.D.; Sweeney, D.W.; Karel, C.: An algorithm for traveling salesman problem. Operations Research, 11: 972-989, 1963.

References of Section 3

- Martello, S.; Toth, P.: Knapsack Problems: Algorithms and Computer Implementations. Wiley Series in Discrete Mathematics and Optimization, 1990. (ISBN-10: 0-4719-2420-2), (ISBN-13: 978-0471924203)
- Nourie, F.J.; Venta, E.R.: Finding optimal line balances with OptPack. Operations Research Letters 10:165-171, 1991.
- Scholl, A.: Balancing and Sequencing of Assembly Lines. 2nd edition. Physica, 1999. (ISBN-10: 3-7908-1180-7) , (ISBN-13: 978-3790811803)
- Scholl, A.; Klein, R.: SALOME: a bidirectional branch and bound procedure for assembly line balancing. INFORMS: Journal on Computing 9:319-334, 1997.
- Vahrenkamp, R.; Mattfeld, D.C.: Logistiknetzwerke - Modelle für Standortwahl und Tourenplanung. 2nd edition. Springer Gabler, Wiesbaden, 2014.