LiSA - A Library of Scheduling Algorithms
Handbook for Version 3.0

Michael Andresen, Heidemarie Bräsel, Frank Engelhardt, Frank Werner
Fakultät für Mathematik
Otto-von-Guericke Universität Magdeburg

Abstract

LiSA - A Library of Scheduling Algorithms is a software package for solving deterministic scheduling problems, in particular shop problems described by $\alpha \mid \beta \mid \gamma$, where α characterizes the machine environment, β gives additional constraints for the jobs and γ describes the objective function. The development of LiSA was supported by the projects *Latin Rectangles in Scheduling Theory (1997-1999)* and *LiSA - A Library of Scheduling Algorithms (1999-2001)* by the Ministry of Education and the Arts of the state Saxony-Anhalt. Results of diploma theses and Ph.D. theses in our research team as well as practical courses of students were used for the development of the software package.

The handbook contains all necessary information to work with LiSA: License conditions, technical requirements, mathematical basic knowledge about the models used in LiSA, description of algorithms and used file formats, an example for working with LiSA, instructions for incorporating own algorithms and for the automated call of algorithms.

The handbook is also available as html file at the homepage of LiSA. Additionally, the explanations of the algorithms and the LiSA components are incorporated into the software as help files.

Homepage: http://lisa.math.uni-magdeburg.de

Chapter 1

General Information

1.1 What is LiSA? - Introduction and Overview

LiSA - A Library of Scheduling Algorithms is a software package for solving deterministic shop scheduling problems. In a shop scheduling problem, a set of jobs has to be processed on a set of machines with additional constraints such that a specific objective function becomes optimal. All parameters are known and fixed. In the literature, such problems are usually described by a triplet $\alpha \mid \beta \mid \gamma$, where α denotes the machine environment, β gives additional constraints for the jobs and γ describes the objective function. A feasible solution of a shop problem is denoted as a sequence, and the corresponding starting (or completion) times constitute a schedule. Sequences and schedules are described by matrices and in LiSA also by acyclic digraphs and Gantt charts, respectively. Since most shop problems are hard to solve, LiSA contains a number of constructive and iterative heuristics.

If a user wishes to solve a scheduling problem by means of LiSA, one has to fix the problem type in the $\alpha \mid \beta \mid \gamma$ notation and the number of jobs and machines using the graphical user interface. The processing times and the remaining input parameters can be entered manually or generated randomly. It is also possible to read these data from an XML file. When data input is completed, all algorithms that solve the problem either exactly or approximately can be invoked. After the application of an algorithm, the schedule will be visualized in a Gantt chart which can be job and machine oriented, respectively. If a first solution is available, the iterative algorithms are released for use. Since most algorithms offer a set of parameters, the user has a number of strategies for solving his or her problems at hand. The solution can be stored in an XML file.

LiSA contains some extras, e.g. the check of the complexity status (with a reference) of a problem. Here, the database on the complexity of scheduling problems, developed by the colleagues from the University of Osnabrück, is used. In addition, manipulations of Gantt charts are possible such that the user can influence the construction of a schedule.

LiSA has a modular structure in which every algorithm can also be used externally. So it is possible to call an algorithm by a command line or to incorporate it into an automated call of algorithms. When using such an automated call of algorithms, after fixing a problem type and the parameters, one can solve a specific number of instances using all algorithms available for this type of problems. For all of these instances, LiSA stores a list of solutions generated by the algorithms which can be inspected via the graphical user interface. By means of the generated log file, one can filter the results into a file compatible with Excel so that a quick evaluation is possible. In addition, it is possible to construct hybrid algorithms using this concept.

By the modular structure it is easily possible for the user to incorporate own algorithms into LiSA. In addition to the C++ source code, one needs an XML file which makes algorithm available to the graphical user interface, so that LiSA can automatically check if it can be used for solving a given problem. A help file for the new algorithm makes it possible for other users to understand its parameter set and invoke it correctly.

This handbook has the following structure:

In addition to this introduction, Chapter 1 contains an overview of the LiSA team, the system requirements for the use of LiSA and the license conditions.

The notations used in LiSA, the classification of the problems, the used block matrices model with the basic algorithms and an overview of the algorithms contained in LiSA are given in Chapter 2. Here, we also describe the file format used in LiSA.

The following three chapters describe the input of the problem (Chapter 3), the algorithms used in LiSA (Chapter 4), partitioned into universally usable exact algorithms, universally usable constructive and iterative heuristics, special algorithms and, finally, the output of the results (Chapter 5).

Chapter 6 contains some extras starting with the description of additional internal program modules, e.g. the determination of the complexity status of a problem, the manipulation of a schedule or reducibility algorithms.
Moreover, the incorporation of own algorithms and the automated call of algorithms are described.

Chapter 8 contains some illustrative examples for using LiSA.

The handbook finishes with an XML reference, the GNU license conditions and a literature overview given in an appendix.

1.2 System Requirements and License

LiSA is licensed under GPL (GNU General Public License), the conditions for this license can be found in the appendix.

LiSA was primarily developed for Unix systems. For the compilation, there are a standard C++ compiler and Tcl/Tk 8.0 or higher required, we recommend gcc 3.2 (or higher) and Tcl/Tk 8.4.
Version 2.3 was completely developed under SuSE Linux 8.1. The further development to Version 3 uses SuSE Linux 9.2 and Tcl/Tk 8.4. Successful tests of Version 3.0 have been

made with the following systems:

- *SuSE 9.2 with the use of gcc 3.4.4 und Tcl/Tk 8.4*;
- *SuSE 9.3 with the use of gcc 3.3.5 und Tcl/Tk 8.4*;
- *SuSE 10.3 with the use of gcc 3.3.5 und Tcl/Tk 8.4*;
- *Solaris 9.0 with the use of gcc 3.4.4 und Tcl/Tk 8.4*;
- *Fedora 4.0 with the use of gcc 4.0.2 und Tcl/Tk 8.4*;
- *Debian 5.0 with the use of gcc 4.4.1 und Tcl/Tk 8.5*;

Previous LiSA versions have been successfully compiled on the following systems:

- *SunOS 5.6/5.7 (sparcSTATION and ULTRAsparc)*;
- *IRIX 6.4*;
- *HP-UX 09.05/10.10/10.20*;
- *AIX 4.2*;
- *SuSE 6.1, 8.0, 8.1*;
- *RedHat 5.1.*
- *Solaris 8.*

Under Windows, LiSA can be compiled by means of the Cygwin platform, which offers a complete Unix-comparable environment. In particular for Windows systems, we also offer an automatic installer which installs LiSA on Windows 2000, Windows XP, Windows Vista or Windows 7 without the necessity to use further software.
Hints for compiling LiSA on several platforms can be found in the file `INSTALL`, which is contained in the source package.

1.3 The LiSA Team

Research groups at universities are usually not homogeneously formed over a long period. The following overview contains the names of most scientists and students who were involved in the development of LiSA as well as their major working areas:

Heidemarie Bräsel: Initiator of the project, leader of the LiSA team and supervisor of diploma and Ph.D. sudents (1997-2009)
Thomas Tautenhahn: Responsible for the efficiency of the data structures and algorithms, supervisor of students in the area of programming (1997-2000), habilitation thesis 2002
Per Willenius: Responsible for the user interface of LiSA, the corresponding algorithms and for the coordination of all program modules, supervisor of students in the area of programming (1997-2001), Ph.D. thesis 2000
Martin Harborth: Responsible for the complexity module in the main program of LiSA, supervisor of students in the area of programming (1997-1999), Ph.d. thesis 1999
Lars Dornheim: Responsible for the compatibility of LiSA under various computer configurations and operating systems (1998-1999)
In the first time period, the following students were involved:
Ines Wasmund: Visualization of schedules by Gantt charts
Andreas Winkler: Several neighborhood search procedures

Marc Mörig: Matching algorithms for open shop problems, reducibility algorithms
Christian Schulz: Shifting bottleneck heuristic for the job shop problem
Manuela Vogel: A heuristic for the flow shop problem.

For a short period, the following students were also involved in the project:
Holger Hennes, Birgit Grohe, Christian Tietjen, Carsten Malchau und *Tanka Nath Dhamala* (Sandwich fellow, Ph.D. thesis 2002). In a practical computer course 2001, *Thomas Klemm*, *Andre Herms, Jan Tusch*, *Ivo Rössling*, *Marco Kleber* and *Claudia Isensee* incorporated new algorithms into LiSA.

In 2002, *Lars Dornheim, Sandra Kutz, Marc Mörig und Ivo Rössling* prepared LiSA for a cooperative development. The modularity of the software has been substantially improved. Several errors have been removed so that the software is now more stable. A specific LiSA server has been installed for the communication between the developers and the users of LiSA, and a version management system as well as a bug tracking system have been incorporated. The concept of incorporating own algorithms has been improved and simplified. A new homepage has been created. LiSA Version 2.3 was ready. For the work on LiSA, the students received the special award of the jury at the student conference of the DMV meeting in Halle/S. (2002).

The major work on LiSA Version 3 was done by Marc Mörig, Jan Tusch, Mathias Plauschin and Frank Engelhardt. The main progress in LiSA Version 3 can be described as follows:
- The file format has been changed by *Jan Tusch* to .xml, he also implemented the genetic algorithms;
- The call of algorithms has been automated by *Marc Mörig*, later also by *Mathias Plauschin* and by *Frank Engelhardt*. They also developed further the filtering of the results into a file compatible with Excel, implemented for the first time by *Andre Herms* in Perl;
- For the Windows versions, there is an automatic installer created by *Mathias Plauschin*, improved by *Frank Engelhardt*. For the use of the installer, there is no further software required.

For the last four years, *Frank Werner* also collaborated with the LiSA team, mainly for publications based on the use of algorithms in LiSA. He was responsible for the English version of this handbook.

Chapter 2

Basic Knowledge

2.1 Definitions and Notations

In a *shop scheduling problem*, a set of *jobs* has to be processed on a set of *machines* in a predefined *machine environment* under various *additional constraints* in such a way that an *objective function* becomes optimal. The problem is called *deterministic*, if all parameters are known and fixed. A number of optimization problems where one looks for an optimal sequence of activities under constrained resources can be modeled as a scheduling problem. The following Table 2.1 contains the basic definitions used in LiSA:

Notations	
n, m	Number of jobs and number of machines
$\{A_1, \ldots, A_n\}$	Set of jobs to be processed
$I = \{1, ..., n\}$	Set of indices of the jobs
$\{M_1, \ldots, M_m\}$	Set of machines which process the jobs
$J = \{1, ..., m\}$	Set of indices of the machines
$p_{ij} \geq 0$	Processing time of job A_i on machine M_j
$PT = [p_{ij}]$	Matrix of the processing times
(ij)	Operation, i.e. the processing of job A_i on M_j
SIJ	Set of operations (ij)
u_i, v_j	Number of operations of A_i and on M_j, respectively
c_{ij}	Completion time of operation (ij)
C_i	Completion time of job A_i
$C = [c_{ij}]$	Matrix of the completion times
r_i, d_i	Release date and due date of job A_i, respectively
w_i	weight of job A_i
$L_i = C_i - d_i$	Lateness of job A_i
$T_i = \max\{0, C_i - d_i\}$;	Tardiness of job A_i
$U_i = \begin{cases} 0, & \text{if } C_i \leq d_i \\ 1, & \text{otherwise} \end{cases}$	$\sum U_i$ counts the number of late jobs

Table 2.1: Basic notations for deterministic scheduling problems

For a scheduling problem with more than one machine, we introduce the following sequences: The *machine order of a job A_i* is the sequence of machines which the job has to be processed:

$M^i_{j_1} \to M^i_{j_2} \to \ldots \to M^i_{j_{u_i}}$, where $j_1, \ldots, j_{u_i}$ is a permutation of the numbers $1, \ldots, u_i$.
The *job order on machine* M_j is the sequence of the jobs processed on this machine $A^j_{i_1} \to A^j_{i_2} \to \ldots \to A^j_{i_{v_j}}$, where $i_1, \ldots, i_{v_j}$ is a permutation of the numbers $1, \ldots, v_j$. If not causing confusion, we drop the superscripts.

2.2 Classification of Deterministic Scheduling Problems

LiSA uses the $\alpha \mid \beta \mid \gamma$ classification for deterministic scheduling problems by Graham et al. [15], where

- α describes the machine environment,
- β gives job characteristics and further constraints and
- γ describes the objective function.

LiSA uses this classification not only for describing a problem but also for determining the complexity status of the problem considered. Tables 2.2, 2.3 and 2.4 gives an overview on the possible parameters of a problem $\alpha \mid \beta \mid \gamma$ without guaranteeing completeness. In contrast to the literature, where a repeated processing of a job in a job shop problem is possible, LiSA uses the assumption for all shop problems that any job is processed at most once on any machine (classical case).
The β field may contain no, one or several parameters from $\{\beta_1, \ldots, \beta_5\}$. Here, many other constraints are also possible, e.g.
- *no-wait* : Waiting times between two successive operations of the same job are not allowed.
- *no-idle* : Idle times between two successive jobs on the same machine are not allowed.
- $p_{ij} \in \{1, 2\}$: There are only processing times from $\{1, 2\}$ allowed.

Any criterion $F(C_1, \ldots, C_n)$ from Table 2.4 is *regular*, i.e. if $C^*_i \geq C_i \ \forall \ i \in I$, then $F(C^*_1, \ldots, C^*_n) \geq F(C_1, \ldots, C_n)$.
A non-regular criterion is e.g. die minimization of the penalty costs which occur if a job is completed too early or too late.

There are many other constraints on the processing of the jobs, e.g. a flexible flow-shop problem (FFS) is a combination of a flow-shop and a parallel machine problem. If machine routes are only partly given, it is a mixed shop problem. Moreover, there exist scheduling problems with resource constraints, batching problems and many others.

Machine environment $\boldsymbol{\alpha = \alpha_1\alpha_2}$	
$\alpha_1 \in \{1, P, Q, R\}$	Any job consists of exactly one operation which can be processed on an arbitrary machine.
$\alpha_1 = 1$	There is only one machine, thus: $p_{i1} = p_i$.
$\alpha_1 = P$	Any job has to be processed on exactly one of m identical parallel machines, i.e. we have $p_{ij} = p_i$.
$\alpha_1 = Q$	Each of the m parallel machines has the same given *speed* s_j, i.e. we have $p_{ij} = p_i/s_j$.
$\alpha_1 = R$:	The speeds s_{ij} for the processing of operation (ij) depend both on A_i and on machine M_j, i.e. we have $p_{ij} = p_i/s_{ij}$.
$\alpha_1 \in \{O, F, J\}$	Any job has to be processed on any machine exactly once or at most once (classical case).
$\alpha_1 = O$	open-shop problem: The machine and job orders can be arbitrarily chosen.
$\alpha_1 = J$	job-shop problem: The machine order is fixed and the job order can be arbitrarily chosen.
$\alpha_1 = F$	flow-shop problem: The machine order is fixed and identical for any job, w.l.o.g.: $M_1 \to M_2 \to \ldots \to M_m$. The job order can be arbitrarily chosen.
$\alpha_2 \in \{\circ, c\}$	Characteristics of the number of machines
$\alpha_2 = c$	The number of machines m is constant, $m = c$.
$\alpha_2 = \circ$	The number of machines is variable, i.e. it is part of the input.

Table 2.2: Parameter of the machine environment α

Characteristics of jobs and additional constraints $\{\beta_1, \ldots, \beta_5\}$	
$\beta_1 \in \{\circ, pmtn\}$	Preemption of operations
$\beta_1 = \circ$	Preemption is not allowed.
$\beta_1 = pmtn$	Preemption is allowed, i.e. the processing of a job on a machine can be interrupted and resumed later.
$\beta_2 \in \{\circ, prec, outtree, intree, tree, chain\}$	The precedence constraint $A_i \rightarrow A_k$ means that job A_i must be completed before job A_k starts.
$\beta_2 = \circ$	There are no precedence constraints.
$\beta_2 = chain$	The precedence constraints have a path structure.
$\beta_2 = outtree$	Any job has at most one predecessor.
$\beta_2 = intree$	Any job has at most one successor.
$\beta_2 = tree$	The precedence constraints have a tree structure.
$\beta_2 = prec$	The precedence constraints are given by an acyclic digraph.
$\beta_3 \in \{\circ, r_i \geq 0\}$	Release dates of the jobs
$\beta_3 = \circ$	Any job is available at time 0: $r_i = 0 \quad \forall\, i \in I$.
$\beta_3 = r_i \geq 0$	For any job, a release date $r_i \geq 0$ is given.
$\beta_4 \in \{\circ, d_i\}$	Due dates of the jobs
$\beta_4 = \circ$	There are no due dates.
$\beta_4 = d_i$	Any job must be completed before its due date $d_i \geq 0$.
$\beta_5 \in \{\circ, p_{ij} = 1\}$	Specific processing times
$\beta_5 = \circ$	$p_{ij} \geq 0\ \forall\,(i,j) \in I \times J$, p_{ij}: natural numbers.
$\beta_5 = p_{ij} = 1$ for $\alpha_1 \in \{1, O, F, J\}$	$p_{ij} = 1$ holds for all operations $(ij) \in SIJ$.

Table 2.3: Some additional constraints in β

Objective function $\gamma \in \{f_{max}, \sum f_i\}$	
$f_{max} \in \{C_{max}, L_{max}\}$	Objective function: $f_{max} \to$ min!
$C_{max} = \max_{i \in I}\{C_i\} \to$ min!	Minimize the makespan.
$L_{max} = \max_{i \in I}\{L_i\} \to$ min!	Minimize maximum lateness.
$\sum f_i \in \{\sum C_i, \sum T_i, \sum U_i, \sum w_iC_i, \sum w_iT_i, \sum w_iU_i\}$	Objective Function: $\sum f_i \to$ min!
$\sum C_i \to$ min!	Minimize the sum of the completion times of all jobs.
$\sum T_i \to$ min!	Minimize total tardiness of all jobs.
$\sum U_i \to$ min!	Minimize the number of late jobs.
$\sum w_iC_i \to$ min!	Minimize the weighted sum of the completion times of all jobs.
$\sum w_iT_i \to$ min!	Minimize weighted total tardiness.
$\sum w_iU_i \to$ min!	Minimize the weighted sum of late jobs.

Table 2.4: Regular criteria

2.3 Models for Shop Problems

In order to understand the input and output of LiSA, we briefly explain the models used in LiSA. LiSA has mainly been developed for shop problems, i.e., for flow-, job- and open-shop problems. However, it also contains a number of algorithms for single machine problems. It is assumed that at each time, any job is processed on at most one machine and any machine processes at most one job.

2.3.1 Sequences and Schedules

In order to introduce the notations of a sequence and a schedule, we define the following graphs, where the set of vertices is given by the set of operations SIJ:

- The graph of machine orders $G(MO)$ contains all arcs which correspond to the direct precedence constraints of the jobs.
- The graph of job orders $G(JO)$ contains all arcs which correspond to the direct precedence constraints between jobs on the machines.
- The graph $G(MO, JO) = (SIJ, A)$ contains all arcs from $G(MO)$ and $G(JO)$, i.e.,

$$((ij),(kl)) \in A \Longleftrightarrow \begin{cases} (i = k \ \wedge \text{ after the processing of job } A_i \text{ on} \\ \ M_j, \text{ this job is processed on } M_l) \ \vee \\ (j = l \ \wedge \text{ after machine } M_j \text{ has processed job } A_i, \\ \ \text{this machine processes job } A_k.) \end{cases}$$

A combination of machine and job orders is called *feasible*, if the corresponding graph $G(MO, JO)$ is acyclic. In this case, the graph is called a *sequence graph.*

Example 1 *Three jobs have to be processed on four machines. The matrix of the processing times PT and the job and machine orders are given by*

$$PT = \begin{bmatrix} 2 & 1 & 0 & 1 \\ 2 & 3 & 4 & 3 \\ 1 & 5 & 1 & 2 \end{bmatrix}, \textit{ therefore } SIJ = I \times J \backslash \{(13)\}.$$

$$\begin{array}{llll} A_1: & M_1 \to M_2 \to M_4 & M_1: & A_1 \to A_2 \to A_3 \\ A_2: & M_2 \to M_4 \to M_1 \to M_3 & M_2: & A_2 \to A_3 \to A_1 \\ A_3: & M_4 \to M_1 \to M_2 \to M_3 & M_3: & A_3 \to A_2 \\ & & M_4: & A_3 \to A_1 \to A_2 \end{array}$$

The following figure gives the graphs $G(MO)$, $G(JO)$ and $G(MO, JO)$:

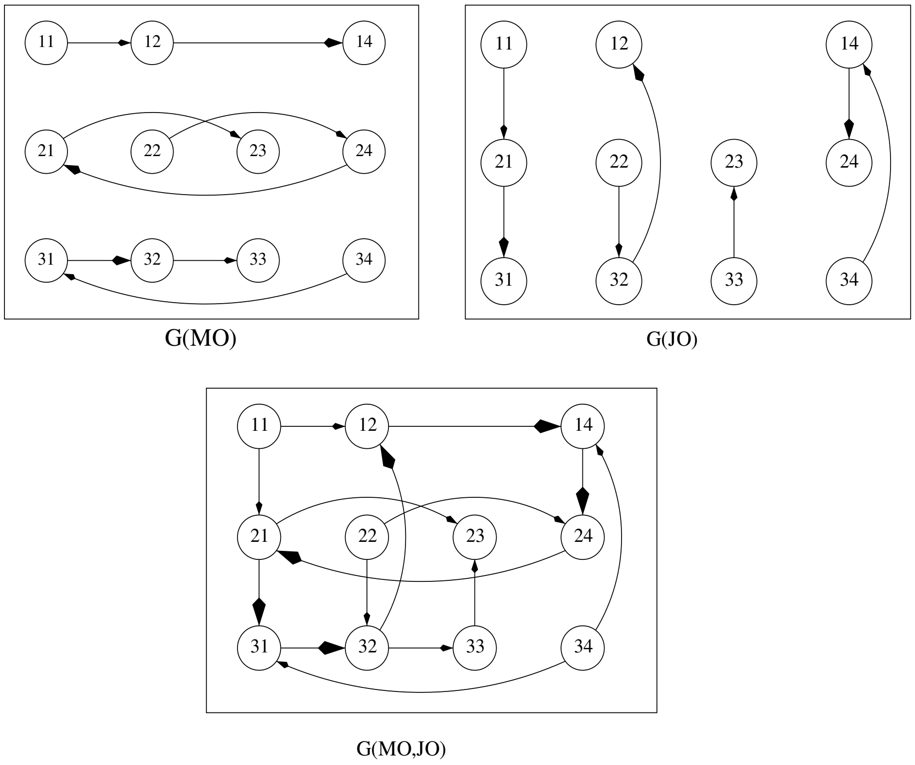


Figure 2.1: $G(MO)$, $G(JO)$ and $G(MO, JO)$

This combination of machine and job orders is not feasible since digraph $G(MO, JO)$ contains the cycle

$$(1,2) \to (1,4) \to (2,4) \to (2,1) \to (3,1) \to (3,2) \to (1,2).$$

This would mean that every operation on the cycle is a predecessor and a successor of itself, which is certainly a contradiction.
If we choose the natural sequence of the jobs and machines, respectively, digraph $G(MO, JO)$

cannot contain cycles since all horizontal arcs are directed from left to right and all vertical arcs are directed from the top to the bottom. In this case, digraph $G(MO, JO)$ is a sequence graph.

Now we assign to each vertex (ij) of the sequence graph $G(MO, JO)$ the processing time p_{ij} as a weight. Then the 'timetable' of processing the operations is denoted as a *schedule*. Usually, schedules are described by the starting or completion times of all operations, and they can be visualized by *Gantt charts*, which can be *machine oriented* oder *job oriented*. There exist the following classes of schedules:

A schedule is called *semi-active*, if no operation can be completed earlier without changing the corresponding sequence.

A schedule is called *active*, if no operation can be completed earlier without delaying some other operation.

A schedule is called *non-delay*, if no machine is idle as long as there is a job which can be processed on this machine.

Any non-delay schedule is active, and any active schedule is semi-active. The opposite is in general not true.

Example 2 *In addition to the matrix of the processing times given in Example 1, let the following acyclic digraph $G(MO, JO)$ be given. In a first step, the corresponding semi-active schedule can be visualized in a job oriented Gantt chart. Now, all operations are processed which are sources (vertices without predecessor) in the sequence graph. Then all sources together with the outgoing arcs from them are dropped and in the next step, all operations are sequenced which are now sources, etc. In addition, this schedule is active, but in the open-shop case not necessarily non-delay since operation* (11) *could start at time 2.*

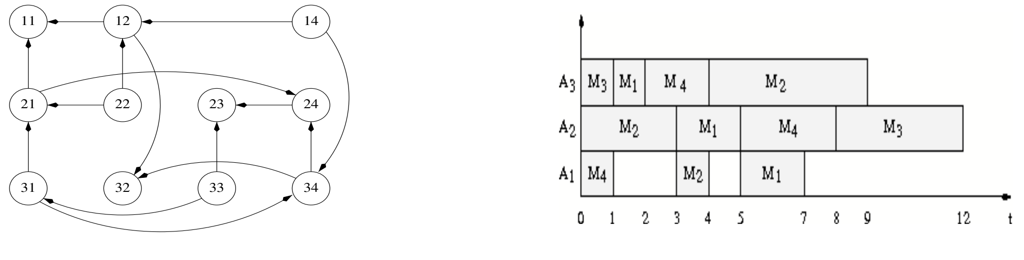


Figure 2.2: Sequence graph $G(MO, JO)$ and schedule (job oriented Gantt chart)

2.3.2 The Block-Matrices Model for Shop Problems

For a shop problem, the machine and job orders, the sequences and schedules can be described by matrices. Any matrix contains an information on operation (i, j) in row i and column j. This is either a structural or a time-wise property of operation (ij). For this reason, we introduce the rank $rg(v)$ of a vertex v in an acyclic digraph. When the number of vertices on a path w is denoted as the length of this path, then rank $rg(v)$ denotes the number of vertices on a longest path ending in vertex v. When the vertices (ij) of the sequence graph $G(MO, JO)$ are weighted by the processing time p_{ij}, the makespan of the corresponding semi-active schedule is determined as the weight of a critical path (path with maximal weight) in the sequence graph.

Thus, for describing the graphs we have the following matrices:

- $G(MO)$ is described by $MO = [mo_{ij}]$, where mo_{ij} is the rank of operation (ij) in $G(MO)$ (matrix of the machine orders).
- $G(JO)$ is described by $JO = [jo_{ij}]$, where jo_{ij} is the rank of operation (ij) in $G(JO)$ (matrix of the job orders).
- $G(MO, JO)$ is described by $LR = [lr_{ij}]$, where lr_{ij} is the rank of operation (ij) in the sequence graph $G(MO, JO)$ (sequence).

In any row i of matrix MO, there is a permutation of the numbers $1, \ldots, u_i$, where u_i is the number of operations of job A_i. In any column j of matrix JO, there is a permutation of the numbers $1, \ldots, v_j$, where v_j is the number of operations which have to be processed on machine M_j. A sequence LR combines the properties of MO and JO. Due to the definition of the rank, any sequence satisfies the *sequence property*: For any number $z = lr_{ij} > 1$, there exists in row i or in column j or in both the number $z - 1$. It can be noted that sequences on complete sets of operations $SIJ = I \times J$ are special latin rectangles. A latin rectangle $LR[n, m, r]$ is is a matrix with n rows, m columns and entries from a set $\{1, \ldots, r\}$, where any number from this set occurs in any row and any column at most once. If a latin rectangle satisfies the sequence property, then it is the rank matrix of a sequence graph $G(MO, JO)$. If the set of operations is incomplete, then we obtain so-called partial latin rectangles as rank matrices.

To describe a semi-active schedule which corresponds one-to-one to a sequence with given matrix of the processing times, we introduce the following matrices:

- Matrix $C = [c_{ij}]$, where c_{ij} is the completion time of operation (ij).
- Matrix $H = [h_{ij}]$, where h_{ij} is the minimal time required for processing all predecessors of operation (ij) (head of (ij): earliest possible starting time of operation (ij)).
- Matrix $T = [t_{ij}]$, where t_{ij} is the minimal time required for processing all successors of operation (ij) (tail of (ij)).
- Matrix $W = H + PT + T = C + T = [w_{ij}]$, where w_{ij} is the maximum weight of a path which contains operation (ij).

Then we obtain the makespan

$$C_{max} = \max_{(ij)\in SIJ} c_{ij} = \max_{(ij)\in SIJ} w_{ij}.$$

All operations (ij) with $w_{ij} = C_{max}$ belong to at least one critical path. By means of matrix W, one can construct all critical paths by breadth first search. Figure 2.3 summarizes the block-matrices model.

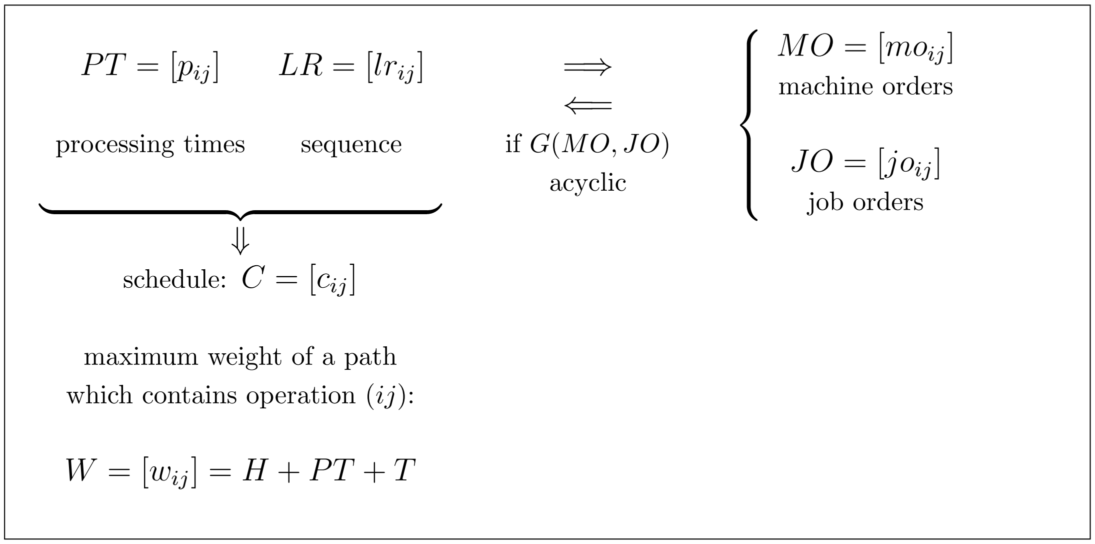


Figure 2.3: Block-Matrices Model

2.3.3 Disjunctive Graph Model and Block-Matrices Model

Shop problems are often modelled in the literature by the *disjunctive graph model*, see e.g. Brucker, [8]. The set of vertices of the disjunctive graph are the operations which are successively numbered and weighted by the processing times. Two vertices are joined by an edge if they cannot be simultaneously processed, i.e. they belong to the same job or to the same machine. There are arcs from a fictitious source to any vertex. From any vertex, there are arcs to a fictitious sink. Precedence constrains for the operations in the job-shop and flow-shop case are modelled by an orientation of the corresponding edge. We look for an orientation of all edges that are still not oriented such that the resulting graph does not contain any cycle and the weight of a critical path becomes minimal (in case of minimizing the makespan).

The model used in LiSA can be derived from the latter model as follows:
- Delete the source and the sink and the incident arcs from the disjunctive graph.
- Determine an acyclic orientation of the disjunctive graph.
If we drop all arcs that are transitive with respect to the machine and job orders from the acyclic orientation of the disjunctive graph, we obtain the sequence graph $G(MO, JO)$ used in LiSA. Moreover, a sequence is a combination of all ranks of an acyclic orientation of a disjunctive graph.

2.3.4 Basic Algorithms for the Block-Matrices Model

There are some basic algorithms belonging to the block-matrices model, which have a linear complexity due to the special structure of the sequence graph. They exclusively use the matrices of the model which describe in one-to-one correspondence the graph-theoretic properties considered.

Algorithm 1 determines the sequence LR by means of the given matrices MO and JO, if graph $G(MO, JO)$ does not contain cycles. The set MQ contains all operations, which are both a source in $G(MO)$ as well as in $G(JO)$.

Algorithm 1: Determination of LR by means of a combination (MO,JO), if $G(MO, JO)$ does not contain any cycle

```
Input: $n, m, SIJ$, $MO$ and $JO$ on the set of operations $SIJ$;
Output: $LR$ on the set of operations $SIJ$, if $G(MO, JO)$ is acyclic;
BEGIN $k := 0$;
  REPEAT
        $k := k + 1$; Determine set $MQ = \{(ij) \in SIJ \mid mo_{ij} = jo_{ij} = 1\}$;
        IF $MQ = \emptyset$ THEN $(MO, JO)$ is infeasible and STOP;
        FORALL$(ij) \in MQ$ DO
            BEGIN
                $lr_{ij} = k$; Mark in $MO$ row $i$ and in $JO$ column $j$;
            END;
        $SIJ := SIJ \setminus MQ$;
        FORALL $(ij) \in SIJ$ in a marked row in $MO$ DO $mo_{ij} := mo_{ij} - 1$;
        FORALL $(ij) \in SIJ$ in a marked column in $JO$ DO $jo_{ij} := jo_{ij} - 1$;
  UNTIL $SIJ = \emptyset$;
END.
```

Algorithm 2 determines MO and JO by means of LR. Here a_i and b_j are the smallest natural numbers which are available for the rank of operation (ij). The maximal element in LR is denoted by r.

Algorithm 2: Determination of MO and JO by means of LR

```
Input: $n, m, r, I, J, SIJ$, $LR$ on the set of operations $SIJ$;
Output: $MO$ and $JO$ on the set of operations $SIJ$;
BEGIN Set $\forall\ i \in I$: $a_i = 1$ and $\forall\ j \in J$: $b_j = 1$;
  FOR $k := 1$ TO $r$ DO
            FORALL $(ij) \in SIJ$ with $lr_{ij} = k$ DO
                BEGIN
                    Set $mo_{ij} = a_i$ and $a_i = a_i + 1$;
                    Set $jo_{ij} = b_i$ and $b_j = b_j + 1$;
                END;
END.
```

Algorithm 3 generates a semi-active schedule, i.e., the matrix $C = [c_{ij}]$ of the completion times of all operations, by means of the matrix of the processing times PT and the sequence LR. Here r_i and $\overline{r}_j$ denote the earliest possible starting time of job A_i and on machine M_j, respectively.

Algorithm 3: Determination of C, provided that PT and LR are given

```
Input: $n, m, r, I, J, SIJ$, $PT$ and $LR$ on the set of operations $SIJ$;
Output: $C$ on the set of operations $SIJ$.
BEGIN
  Set $\forall\ i \in I$: $r_i = 0$ and $\forall\ j \in J$: $\overline{r}_j = 0$;
  FOR $k := 1$ TO $r$ DO
      FORALL $(ij) \in SIJ$ with $lr_{ij} = k$ DO
        BEGIN
          $c_{ij} := \max\{r_i, \overline{r}_j\} + p_{ij}$;
          $r_i := c_{ij}$; $\overline{r}_j := c_{ij}$;
        END;
END.
```

Algorithm 4 determines the matrices $H = [h_{ij}]$ and $T = [t_{ij}]$. h_{ij} is the head of operation (ij), i.e., the smallest time which is necessary for the processing of all preceding operations of (ij) in the sequence graph $G(MO, JO)$. t_{ij} denotes the tail of operation (ij), i.e., the smallest time which is necessary for the processing of all succeeding operations of (ij) in the sequence graph $G(MO, JO)$. Here $r_i, \overline{r}_j$ are again the earliest starting times of job A_i and on machine M_j, respectively. $s_i, \overline{s}_j$ denote the earliest starting times of jobs A_i and on machine M_j, respectively, in a backward calculation.

Algorithm 4: Calculation of H and T

```
Input: $n, m, r, I, J, SIJ$, $PT$ and $LR$ on the set of operations $SIJ$;
Output: $H$ and $T$ on the set of operations $SIJ$.
BEGIN
  Set $\forall\ i \in I$: $r_i = 0$ and $\forall\ j \in J$: $\overline{r}_j = 0$;
  Set $\forall\ i \in I$: $s_i = 0$ and $\forall\ j \in J$: $\overline{s}_j = 0$;
  FOR $k := 1$ TO $r$ DO
    BEGIN
      FORALL $(ij) \in SIJ$ with $lr_{ij} = k$ DO
        BEGIN
        $h_{ij} := \max\{r_i, \overline{r}_j\}$; $r_i := h_{ij} + p_{ij}$; $\overline{r}_j := h_{ij} + p_{ij}$;
        END;
      FORALL $(ij) \in SIJ$ with $lr_{ij} = r - k + 1$ DO
        BEGIN
          $t_{ij} := \max\{s_i, \overline{s}_j\}$; $s_i := t_{ij} + p_{ij}$; $\overline{s}_j := t_{ij} + p_{ij}$;
        END;
    END;
END.
```

Matrix $W = H + PT + T$ contains the weight of a critical path w_{ij} from a source via operation (ij) to a sink of a sequence graph $G(MO, JO)$. Thus, all operations with a maximal weight w_{ij} belong to at least one critical path. Due to the properties of a latin rectangle, we can order all operations in $O(nm)$ time according to non-decreasing ranks. Thus, we can determine the heads and tails as well as all w_{ij} in linear time.

This chapter finishes with an example on the block-matrices model.

Example 3 *Consider the matrix PT of the processing times from Example 1. The due dates of the jobs are given by $d_1 = 6, d_2 = 12, d_3 = 8$. The following combination of machine and job orders is feasible since the graph $G(MO, JO)$ is a sequence graph, i.e., $G(MO, JO)$ does not contain any cycle.*

$$\begin{array}{ll} A_1: & M_4 \rightarrow M_2 \rightarrow M_1 \\ A_2: & M_2 \rightarrow M_1 \rightarrow M_4 \rightarrow M_3 \\ A_3: & M_3 \rightarrow M_1 \rightarrow M_4 \rightarrow M_2 \end{array}$$

$$\begin{array}{ll} M_1: & A_3 \rightarrow A_2 \rightarrow A_1 \\ M_2: & A_2 \rightarrow A_1 \rightarrow A_3 \\ M_3: & A_3 \rightarrow A_2 \\ M_4: & A_1 \rightarrow A_3 \rightarrow A_2 \end{array}$$

Figure 2.4: Machine and job orders and the sequence graph $G(MO, JO)$

Algorithm 1 determines the sequence LR, and Algorithm 3 determines the schedule C:

$$\underbrace{PT = \begin{bmatrix} 2 & 1 & 0 & 1 \\ 2 & 3 & 4 & 3 \\ 1 & 5 & 1 & 2 \end{bmatrix} \quad LR = \begin{bmatrix} 4 & 2 & - & 1 \\ 3 & 1 & 5 & 4 \\ 2 & 4 & 1 & 3 \end{bmatrix}}_{C = \begin{bmatrix} 7 & 4 & - & 1 \\ 5 & 3 & 12 & 8 \\ 2 & 9 & 1 & 4 \end{bmatrix}} \quad \Longleftrightarrow \quad \left\{ \begin{array}{l} MO = \begin{bmatrix} 3 & 2 & - & 1 \\ 2 & 1 & 4 & 3 \\ 2 & 4 & 1 & 3 \end{bmatrix} \\ \\ JO = \begin{bmatrix} 3 & 2 & - & 1 \\ 2 & 1 & 2 & 3 \\ 1 & 3 & 1 & 2 \end{bmatrix} \end{array} \right.$$

Matrices H and T are determined by Algorithm 4, which gives matrix $W = H + PT + T$:

$$W = \begin{bmatrix} 5 & 3 & - & 0 \\ 3 & 0 & 8 & 5 \\ 1 & 4 & 0 & 2 \end{bmatrix} + \begin{bmatrix} 2 & 1 & - & 1 \\ 2 & 3 & 4 & 3 \\ 1 & 5 & 1 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 5 & - & 9 \\ 7 & 9 & 0 & 4 \\ 9 & 0 & 10 & 7 \end{bmatrix} = \begin{bmatrix} 7 & 9 & - & 10 \\ 12 & 12 & 12 & 12 \\ 11 & 9 & 11 & 11 \end{bmatrix}$$

Schedule C yields $C_{max} = 12$ and $C_1 = 7$, $C_2 = 12$, $C_3 = 9$ such that $\sum C_i = 28$, $L_{max} = 1$, $\sum T_i = 2$ and $\sum U_i = 1$ follows. This schedule is optimal in the open-shop case for the objective functions C_{max} and L_{max}, but there are better schedules for $\sum C_i$, $\sum T_i$ and $\sum U_i$.

In a job-shop and a flow-shop problem, respectively, with given matrix MO, all sequences LR are feasible which contain the given machine order.

2.4 Overview of Algorithms

In this chapter we present first an overview of the algorithms available in LiSA. Detailed explanations can be found in Chapter 4.

2.4.1 Universal Algorithms

- Exact Algorithms
A general branch and bound algorithm, which is based on insertion techniques given in Bräsel [5] and Bräsel u.a. [7], can be used for most open-, flow- and job-shop problems with regular criteria. The user can set lower and upper bounds, where an upper bound can be determined by one of the available heuristics. This approach can be used only for small values n, m since the algorithm has a large running time due to its universality. The use of good lower bounds can substantially reduce the running times.

- Constructive Heuristics
Simple dispatching rules are available for a large number of problems, also in case of additional constraints such as given release date of the jobs. A new operation is inserted step by step according to a chosen strategy.

Rule	Choice of the next operation
RAND	randomly
FCFS	first come, first served
EDD	earliest due date first
LQUE	smallest difference between due date and (processing time + tail) first
SPT	shortest processing time first
WSPT	weighted shortest processing time first
ECT	(reachable) earliest completion time first
WI	largest weight first
LPT	largest processing time first

Table 2.5: Dispatching rules

In LiSA Version 3.0, one can find so-called beam-search procedures for many problems. These are restricted branch-and-bound procedures, i.e., the branch-and-bound tree is only partially generated. The algorithm is polynomial since a solution is determined by depth first search and in any step, only a limited number of vertices (beam width) is branched. These procedures can be partitioned into beam-insert and beam-append procedures depending on whether the next operation is inserted or appended.

- Iterative Algorithms
In LiSA, several neighborhood search procedures are available. Each vertex of a neighborhood graph corresponds to a sequence and is weighted by the objective function value of the problem considered. The set of edges is different for the particular neighborhoods. To

start a procedure, an initial solution is required which can be found by a simple constructive heuristic. Then the iterative search on the neighborhood graph starts in order to find a better solution.

Method	Description
Iterative Improvement	transition to a better solution - after enumerating all neighbors or - if the first better neighbor has been found.
Simulated Annealing	allows the transition to a worse solution with a certain probability, which is step by step decreased.
Threshold Accepting	allows the transition to a worse solution by means of a threshold which decreases.
Tabu Search	generates a tabu list in order to avoid a return to a solution already considered.

Table 2.6: Neighborhood search strategies

In Table 2.6, the individual implemented search methods are explained. To get more information, the reader is referred to the books by Brucker [8], Blazewicz u.a. [3] or Pinedo [21]. In LiSA, a number of neighborhoods are available, e.g. the API (adjacent pairwise interchange) neighborhood, where two adjacent operations with respect to a machine or job order are interchanged. In Chapter 4.3, an overview on all neighborhoods available in LiSA can be found.
In LiSA Version 3.0, there are also implemented genetic algorithms. Starting from an initial population (set of starting solutions), a new generation is generated, where specific genetic operators (mutation and crossover) are applied. The fitness measure of a solution (individual) is described by its objective function value. As in the nature, better adapted individuals survive during the course of the evolution.

2.4.2 Specific Algorithms

Some algorithms in LiSA are devoted to specific problem classes.
Table 2.7 contains exact algorithms and heuristics. Most of these algorithms can be found in any monograph on scheduling theory. The exact algorithms for makespan minimization in the job-shop and open-shop case developed by the team of Brucker are available in the internet and are incorporated with permission into LiSA.

In LiSA Version 3.0, there are no algorithms for
- problems with precedence constraints,
- basic algorithms and visualization of problems with allowed preemptions of operations,
- parallel machine problems.

Problem type	Exact procedures
$1\|\| L_{max}$	Branch and bound
$F2\|\| C_{max}$	Johnson rule
$J2\|\| C_{max}$	Jackson rule
$O2\|\| C_{max}$	Algorithm by Gonzalez/Sahni
$J\|\| C_{max}$	Original branch-and-bound algorithm by Brucker
$O\|\| C_{max}$	Original branch-and-bound algorithm by Brucker
O |pmtn | C_{max}	Gonzales/ Sahni (without visualization)
$O2\|\| C_{max}$	LAPT rule, Job with longest alternating processing time first
Problem type	**Heuristics**
$1\|\| L_{max}$	ERD rule, job with earliest release date first
$F \|\| C_{max}$	Beam-Insert with beam width 1 on the set of permutation-flow-shop sequences
$J \|\| C_{max}$	Shifting bottleneck heuristic [1]
$O \|\| C_{max}$ $O \|\| \sum C_i$	Matching heuristics [7]

Table 2.7: Algorithms for specific problems

2.5 The File Format in LiSA

For processing scheduling problems, LiSA uses files that contain XML data. In those files – in the following called *documents* – problem types in the $\alpha \mid \beta \mid \gamma$ notation, problem instances and schedules are stored. In addition, there are other documents which describe algorithms or contain program parameters for LiSA.
While working with the LiSA user interface, there is no knowledge needed about these documents. There, they only appear in the menu entries ***File/Save As*** and ***File/Open***, where they are used to store scheduling problems for later use. However, detailed knowledge about the structure of these documents is crucial when invoking LiSA's algorithms from the command line ($\rightarrow$ 7.1) and for the automated call of algorithms with **auto_alg** ($\rightarrow$ 7.3).

According to its functionality, a *document type* has to be assigned to an XML document. All in all there are five document types:

- `problem` contains a problem type in the $\alpha \mid \beta \mid \gamma$ notation,
- `instance` contains a problem type and an according problem instance,
- `schedule` contains in addition one or several schedules,
- `algorithm` contains a description of an algorithm,
- `controls` contains program parameters for LiSA.

All XML documents used in LiSA start with the following lines:

```
<?xml version="1.0" encoding="ISO-8859-1">
<!DOCTYPE instance PUBLIC "" "LiSA.dtd">
```

The first line is a typical XML file header which has to appear in every document. It describes the XML version and the character encoding (latin-1 in this case). The second line names the document type (`instance`). Additionally, a file named Lisa.dtd is given. It contains a structural description for the document type (a so-called *document type definition*, DTD). It helps the XML interpreter checking if the document structure is valid. This file is located in both subdirectories `bin` and `data` in the LiSA directory.

2.5.1 The Document Type `problem`

An XML file of the document type `problem` only consists of a problem description in the common $\alpha \mid \beta \mid \gamma$ notation. In the following, there is a simple example file with the problem type $F \mid r_i, intree \mid C_{max}$ given.

```
<?xml version="1.0" encoding="ISO-8859-1">
<!DOCTYPE problem PUBLIC "" "LiSA.dtd">
<problem xmlns:LiSA="http://lisa.math.uni-magdeburg.de">
    <alpha env="F" />
    <beta release_times="yes" prec="intree" />
    <gamma objective="Sum_Ci" />
</problem>
```

A complete reference of all possible options that can be put in here is given in appendix A.

2.5.2 The Document Type `instance`

The document type `instance` contains a problem instance, i.e., besides the problem type also parameters like problem size, processing times and due dates. These parameters are summarized in a `<values>` element that follows right after the already introduced `<problem>` element. Giving the problem size, processing times and operation set is mandatory, all other parameters are optional.

Another example, with the problem type $O \mid\mid \sum T_i$, problem size n=5, m=10 and given processing times:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE instance PUBLIC "" "LiSA.dtd">
<instance xmlns:LiSA="http://lisa.math.uni-magdeburg.de">
  <problem>
    <alpha env="O"/>
    <beta/>
    <gamma objective="Sum_Ti"/>
  </problem>
  <values m="10" n="5">
    <processing_times model="lisa_native">
      {
       {  28  32  93  71  42  18  24  10   1   0 }
       {  65  15  68  88  39   0  15  84  39  59 }
       {  15  98  56  59  56  95   0  49  15  25 }
       {  65  80  63  32  62  96  13  20  13  46 }
       {  15  39  43  94  21  25  41  48   3  90 }
      }
    </processing_times>
    <operation_set model="lisa_native">
      {
       {   1   1   1   1   1   1   1   1   1   0 }
       {   1   1   1   1   1   0   1   1   1   1 }
       {   1   1   1   1   1   1   0   1   1   1 }
       {   1   1   1   1   1   1   1   1   1   1 }
       {   1   1   1   1   1   1   1   1   1   1 }
      }
    </operation_set>
    <due_dates>
      {  95  94  39  27  69 }
    </due_dates>
  </values>
</instance>
```

In Appendix A, there are descriptions of all further parameters that belong to a problem instance.

This document type can be passed to the algorithms as input file, that means, a **manual call of an algorithm** ($\rightarrow$ 7.1) can have a document of this type as a parameter. The algorithm will compute one or more solutions for the given instance.

In addition to the parameters that describe a problem instance, an `instance` document may also contain parameters that belong to the algorithm to invoke. These parameters only control the behaviour of the algorithm and do not belong to the problem instance. They are aggregated in a `<controls>` element that directly follows the `<values>` element in the document. The content of this element is individual for each algorithm. Further information is located in the section **Algorithm Modules** (→ 7.1). A reference of the parameters for each algorithm can be found in Chapter 4.
Problem instances can be easily generated with the LiSA GUI. After defining a problem type with ***File/New*** and editing of the parameters via ***Edit/Parameters***, the data can be saved as an `instance` document using ***File/Save As***.

2.5.3 The Document Type `solution`

A `solution` document is structured much like an `instance` document, with the difference that it contains also one or more solutions (sequences). Supplementary data, like completion times or machine and job orders, can also be given. All those data will be stored in a `<schedule>` element.

An example with one solution, which consists of the sequence (`<plan>`) and the matrix of completion times:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE solution PUBLIC "" "LiSA.dtd"> <solution xmlns:LiSA="http://lisa.math.uni-magdeburg.
  <problem>
    <alpha env="O"/>
    <beta/>
    <gamma objective="Sum_Ui"/>
  </problem>
  <values m="5" n="3">
    <processing_times model="lisa_native">
      {
       {  38  25   9  48  23 }
       {  59  25  65  88  82 }
       {   9  48  57  10   5 }
      }
    </processing_times>
    <operation_set>
      {
       { 1  1  1  1  1 }
       { 1  1  1  1  1 }
       { 1  1  1  1  1 }
      }
    </operation_set>
    <due_dates>
      { 107 175  71 }
    </due_dates>
  </values>
  <schedule m="5" n="3" semiactive="yes">
    <plan model="lisa_native">
```

```
          {
           {   4   3   1   5   2 }
           {   5   4   2   7   6 }
           {   2   5   6   3   1 }
          }
        </plan>
        <completion_times model="lisa_native">
          {
           {  95  57   9 143  32 }
           { 158  99  74 328 240 }
           {  14 147 204  24   5 }
          }
        </completion_times>
      </schedule>
    </solution>
```

`solution` documents are created as the output of algorithms. As well as documents of type `instance`, they can also be used as input files for algorithms. In this case, however, all already contained solutions are discarded. Also, the document has to contain algorithm parameters in a `<controls>` element for that purpose. This element has to be placed between the `<values>` and `<schedule>` elements.

2.5.4 The Document Type `algorithm`

This document type is used to include new algorithms into LiSA. In fact, all algorithms are integrated into the GUI using such description files. At program startup, LiSA will look for `algorithm` documents in the directories `data/alg_desc/language/english` or `data/alg_desc/language/german`, respectively, depending on the chosen language and includes the described algorithms into the menu ***Algorithms***.
In order that an algorithm is able to be invoked correctly on a problem instance, several pieces of information have to be specified here. This is the problem type that an algorithm can solve, and parameters that must be passed, among other information.
We leave out an example of such a document because it may never be used during regular work with LiSA. However, an example document is located in the LiSA subdirectory `src/algorithm/sample` (as long as the LiSA source code package is installed). Further reading about `algorithm` documents can be found in the sections **Algorithm Modules** (→ 7.1) and **Inclusion of External Algorithms** (→ 7.2).

2.5.5 The Document Type `controls`

In a `controls`-type document, program options for LiSA are stored. There is one document for every user, which is located in `~/.lisa/default.xml`. The structure of this and all changeable options are described in the appendix A. The most important options can however be changed in LiSA via the menu entry ***Options/General Options***, so that editing this document manually is seldom required.

Chapter 3

Input

An instance of a problem that shall be solved with LiSA consists of the **problem type** ($\rightarrow$ 3.1), the number of jobs n, the number of machines m and the **parameters** ($\rightarrow$ 3.2) that are necessarily related to the problem type, n and m. The input can be done by hand or by opening an input file. In addition, LiSA offers a random generator that generates uniformly distributed data from a given interval. To start the generator, two random integer values, the time seed and the machine seed, have to be given. They can be either given by hand or they are automatically set to the current system time. Entering them manually allows the recreation of an already generated problem.

3.1 Problem Type

In order to treat a problem type with LiSA, first the problem type has to be set in the $\alpha \mid \beta \mid \gamma$ notation according to Graham et al. After that, the machine and the job numbers have to be entered. A problem type therefore is feasible, if at least the machine environment and the objective function are specified. If, additionally, machine and job numbers are given, later the parameter window can be used to input and edit problem specific data.

Invocation

After choosing ***File/New*** from the menu to enter a new problem type, or ***Edit/Problem Type*** to edit an already existing one, the problem type window opens.

Settings

The problem input follows the usual conventions from scheduling textbooks, see also Chapter **Definitions and Notations** 2.1, hence we do not give a description of all possible options for the machine environment, additional constraints or the objective function.

- **machine environment**
- **additional constraints**
- **objective function**
- **(number of) machines**
- **(number of) jobs**

Irregular objective functions

There are some additional objective functions that can be used:
$IRREG1 = \Sigma \mid C_i - d_i \mid$
$IRREG2 = w_{late}L_{max} + w_{early}\max(d_i - C_i)^+ + \Sigma w_i T_i + \Sigma w_{early}(d_i - C_i)^+$
One has to take into account, that for additional weights, which are used by these objective functions, an external problem generator has to be used.

Hints

LiSA always tries to use the last input. If an inconsistent input is made, LiSA will modify former inputs if necessary. For instance, if one chooses multi-machine jobs in a single-machine problem, the problem type will be altered to P.
Since the classification is modified continuously, it can be that certain choices are not implemented yet. LiSA already allows more than ten thousand different problem types to be entered.
Not for every problem type there is a solution method implemented. Appropriate algorithms can be selected from the menus ***Algorithms/Exact Algorithms*** and ***Algorithms/Heuristic Algorithms***, as soon as the problem parameters have been specified.

3.2 Parameters

When LiSA knows the problem type and the number of jobs and machines, all parameters can be entered.

Invocation

After selecting the menu entry ***Edit/Parameters***, the window **Parameters** opens for the parameter input, which is built up differently according to the problem type, n and m. The following choices come up for a problem $J \mid r_i \mid \Sigma w_i T_i$:

Settings

View: The release dates r_i, the due dates d_i and the weights w_i will be shown for all these choices:

- ***Set of Operations:*** The number in row i and column j of this matrix is 1, if operation (ij) exists, 0 otherwise.
- ***Machine Order:*** The number in row i and column j of this matrix is k, if the machine M_j is at position k in the machine order of job J_i.
- ***Processing Times:*** The number in row i and column j in this matrix is the processing time of operation (ij).

Generate: To enter the data by hand, first one has to select the view that contains this data. There, a field can be edited by simply clicking on it. The cursor then jumps to the next field, and so on. After that, according to the problem type, it follows the input of

- **... the processing times**
- **... the set of operations**

- **... the machine order**
- **... the weights**
- **... the due dates**
- **... the release dates**

By choosing Generate, the window **Generate** is opened for randomly generating uniformly distributed numbers. The following parameters can be chosen:

- **Minimum** is the smallest number of the random number interval.
- **Maximum** is the largest number of the random number interval.
- **Time seed** is a random number for the generation of the numerical data.
- **Machine seed** is a random number for the generation of the order of operations on a machine.

With the parameters time seed and machine seed, the generation of a problem instance is guaranteed to be repeatable. Further information about this random generator can be found in Taillard [24].

Adopt Machine Order: The machine order will only be adopted, if every row i is a permutation of the numbers 1 to the number of operations for job J_i.

ATTENTION!

With the menu entry ***Edit/Parameters***, one can also edit the problem instance of the chosen problem type. LiSA always tries to use the last input. If an inconsistent input is made, LiSA will modify former inputs if necessary, so that in the case of a repeated usage of the editing function errors can occur.

3.3 Input File

LiSA has the ability to fetch the problem type and all data of a problem instance from an input file in XML format. Moreover, every file, that was saved with LiSA, can be opened again. In the latter case, a schedule may be loaded, the Gantt chart of which will be shown by default.
Further information about the file format: **The File Format in LiSA**→ 2.5
LiSA is able to operate on a list of sequences in an XML file that comes from an `auto_alg` invocation (see **Automated Call of Algorithms** → 7.3). After opening the file, the list can be viewed via ***Edit/List of Sequences***. In the opening window **List of Sequences**, all sequences in the list are shown in a table in reverse order of their creation in the `auto_alg` file. At the same time, objective function values and some other, self explaining information will be displayed. The list can be sorted by the objective function value or by one of the other shown properties.

By clicking on a field (which is not equal to zero), all information from a sequence will be displayed on the screen that is described in the chapter **Output** (→ 5).

ATTENTION!

Further processing of a sequence in the list will discard all other sequences.

Chapter 4

Algorithms in LiSA

4.1 Universal Exact Algorithm

For solving universal scheduling problems, basically one branch & bound algorithm is available in LiSA.
A branch & bound algorithm differs from a complete enumeration by the fact that it cuts parts of the solution tree. To guarantee that optimal solutions are not excluded, any vertex is evaluated by a lower bound which is valid for the whole branch (i.e., for all solutions belonging to successors of this vertex). If this evaluation is worse than the objective function value of the best known solution, the whole branch can be ignored.
In addition to a **general Branch & Bound Algorithm** (→ 4.1.1), which can be applied to any problem type and any objective function, there are also specific open-shop and job-shop branch & bound algorithms by **Brucker** (→ 4.4.1) available.

4.1.1 Universal Branch & Bound Algorithm

This branch & bound algorithm determines an exact solution for all open-shop, flow-shop, job-shop, or single machine problems with a regular criterion. If an alternative exact procedure is available for the chosen problem type, this should be chosen since the running time for the universal branch & bound algorithm can be possibly rather large.
Using appropriate restrictions (see Settings), this algorithm can also be used for determining a good approximate solution.

Invocation

After the input of the **problem type** and the required **parameters**, this procedure can be reached in the menu item ***Exact Algorithms/Branch & Bound*** in the menu ***Algorithms***.

Settings

Number of solutions: Here an upper bound for the number of output sequences can be settled. The standard setting is 1. The algorithm is able to determine *all* optimal sequences. In this case, one must set a number which is at least as large as the number of optimal solutions.

Lower bound: Here one can set a known lower bound. LiSA considers any sequence, whose objective function does not exceed this value, as optimal. If a value is entered which is larger than the optimal function value, this may lead to the output of sequences which are not optimal. In this way, one can also set a level of the objective function value which is accepted as an approximation.

Upper bound: Here one can set a known upper bound for the objective function value of an optimal solution. Such a bound can be obtained e.g. by the objective function value of a known approximate solution. Then LiSA disregards all partial sequences with a larger objective function value.

Insertion order: By selecting an insertion order it is fixed in which sequence the operations will be appended to the partial sequences.

- **LPT** (longest processing time): The operations are appended in dependence on the processing times, starting with the largest processing time.
- **RANDOM:** The operations are appended in a random order.

Bounding: Here it is fixed which procedure is used by LiSA to calculate lower bounds for the objective function value.

- **NORMAL:** The objective function value of the partial sequence is used as a lower bound.
- **EXTENDED:** (This extended method is not implemented yet.)

Treatment of the problem

This branch & bound algorithm is a universal solver and therefore rather time consuming. If possible, one should use a problem-specific procedure. If the running time becomes too large, the procedure can be stopped at any time. This stopping procedure can also last a few minutes. Then the output gives an approximate solution.
If one looks for several optimal solutions, it is recommended to determine first the optimal function value with the setting 1 for the **number of solutions** and then further optimal solutions in a second run with known bounds.

Invocation in the Autoalg input file

The general branch & bound algorithm is called in an input file for the **Autoalg-function** (→ 7.3) as follows:

```
<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE bb
  string INS_ORDER RANDOM
  long NB_SOLUTIONS 1000000
  double UPPER_BOUND 10000000
  double LOWER_BOUND -10000000
</CONTROLPARAMETERS>
```

The sequence in which the parameters are passed is irrelevant. In case of optional parameters, the default-value is given.
As an executable file, bb must be invoked for the branch & bound procedure. Then the **insertion order**, the **number of solutions** as well as an **upper** and a **lower bound** can be optionally set.

4.2 Universal Constructive Procedures

Heuristic procedures are used for determining quickly an approximate solution. In contrast to **exact procedures** (→ 4.1), they deliver a solution in a substantially smaller running time. However, the quality of these solutions can be far away from an optimal solution.
LiSA contains in addition to simple **constructive heuristics (dispatching rules)** (→ 4.2.1), **matching heuristics** (→ 4.2.2) also beam search procedures for **appending** (→ 4.2.3) and **inserting** (→ 4.2.4) operations, respectively.

4.2.1 Dispatching Rules

Dispatching Rules are very fast heuristics, where step by step new operations are appended. The operation which is appended next is determined by a chosen strategy.
Some problem classes can be even optimally solved by means of such simple procedures.

Invocation

After the input of **problem type** and the required **parameters**, one chooses ***Heuristic Algorithms/Dispatching Rules*** in the menu ***Algorithms***. In dependence on the problem type, particular dispatching rules can also be listed under ***Exact Algorithms***. In this case, usually no settings can be made.

Settings

Generate schedule:

- **SEMI-ACTIVE:** A schedule is called semi-active if no operation can be started earlier without changing the sequence, i.e., a semi-active schedule has no unforced idle times.
- **ACTIVE:** A schedule is called active if it is semi-active and no idle time exists such that a later operation on the corresponding machine can be completely processed within this idle time.
- **NON-DELAY:** A schedule is called non-delay, if at any time where a machine M_j and a job A_i are available, the processing of operation (ij) is started. A non-delay schedule is also active.

Priority: The following definition of priorities are available. The operations to be inserted are ordered according to the chosen strategy and then step by step are appended according to this sequence.

- **RAND** (random): The operations are randomly sequenced.

- **FCFS** (first come - first served): The operation is sequenced next which entered the queue first.
- **EDD** (earliest due date): The operations are ordered according to non-decreasing due dates.
- **LQUE**: The operation is processed next which has the smallest difference 'due date − (processing time + tail)'. In the case of an open shop, instead of the tail the sum of the processing times of all operations of the corresponding job not sequenced yet is taken. As a tie breaking rule, an operation is chosen such that on the corresponding machine the sum of the processing times of the operations not yet sequenced is maximal.
- **SPT** (shortest processing time): The operations are sequenced according to their processing times, starting with the shortest one.
- **WSPT** (weighted shortest processing time): The operation with the smallest quotient of the processing time and the weight is sequenced next.
- **ECT** (earliest completion time): The operation with the earliest completion time is sequenced next.
- **WI**: Jobs are weighted, and the operations are sequenced according to the weights of the jobs, starting with the largest weight.
- **LPT** (longest processing time): The operations are sequenced according to their processing times, starting with the largest time.

Invocation in the Autoalg input file

In the input file for the **Autoalg-function** (→ 7.3), dispatching rules are activated as follows:

```
<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE dispatch
  string SCHEDULE ACTIVE
  string RULE SPT
</CONTROLPARAMETERS>
```

The sequence in which the parameters are passed is irrelevant. In case of optional parameters, the default-value is given.
As an executable file, dispatch must be invoked for all dispatching rules. Optionally, it can be settled which type of a sequence should be **generated** and which **dispatching rule** should be used.

4.2.2 Matching Heuristics

These heuristics generate a sequence for open-shop problems. Here the given problem is reduced to assignment problems which will be solved optimally by searching a maximal matching. This leads to a sequence, where operations with similar processing times are processed in parallel if possible.

Invocation

After the input of **problem type** and the required **parameters**, one chooses ***Heuristic Algorithms/Matching Heuristics*** in the menu ***Algorithms***.

Settings

Kind of algorithm: By this option, one determines how to treat the weights of the operations that are fixed by the setting **Kind of matching**.

- **BOTTLENECK:** The operations are ordered by their weights. Then the largest weight p is determined such that $M = \{p\,(i,j) \mid p\,(i,j) \geq p\}$ contains a perfect matching. The corresponding operations from this matching are appended to the sequence and the weights are deleted. This procedure is repeated until all operations have been appended to the sequence.
- **WEIGHTED:** A maximally weighted matching is determined. The corresponding operations are appended to the sequence and the weights are deleted (they are set on the smallest feasible value). This procedure is repeated until all operations are appended to the sequence. (This is the standard setting.)

Kind of matching: By this option, one fixes how the weights of the particular operations are calculated.

- **HEADS:** The first set to be appended is calculated by the MIN parameter. If an initial set of operations has been appended, the heads are determined for all remaining operations. They are added to the processing times and the results are subtracted from the largest value. This gives the weights for the next matching. In this way, the algorithm minimizes the current makespan value in each step.
- **MIN:** The weight of an operation is calculated as the difference between the processing time of an operation and the largest feasible value. In this way, the average processing time of all operations in one appending step is minimized.
- **MAX:** The weight of an operation directly results from the processing time. In this way, the average processing time of all operations in one appending step is maximized. (This is the standard setting.)

Invocation in the Autoalg input file

In the input file for the **Autoalg-function** ($\rightarrow$ 7.3), matching heuristics are activated as follows:

```
<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE match
  string MINMAX MAX
  string TYPEOF WEIGHTED
</CONTROLPARAMETERS>
```

The sequence in which the parameters are passed is irrelevant. In case of optional parameters, here the default-value is given.
As an executable file, match must be invoked for the matching heuristics. Optionally, the **kind of matching** and the **kind of algorithm** can be settled.

4.2.3 Beam Search Procedures with Appending Technique

Beam Search procedures are a heuristic variant of the **Branch & Bound Procedure** (→ 4.1.1). In contrast to them, the number of partial solutions to be investigated is restricted in any vertex of the solution tree. This leads to an incomplete search in the set of feasible solutions which possibly excludes all optimal solutions. However, such procedures are substantially faster than a complete branch & bound algorithm. The quality of the solution depends both on the chosen problem type and on the chosen settings.
In case of an **appending** procedure, for any partial sequence different positions are considered at which new operations with increasing rank can be appended. Using one of the particular strategies, an operation is searched which fits on this position. In addition to the appending procedure, an **insertion** procedure (→ 4.2.4) is also implemented in LiSA.

Invocation

After the input of the **problem type** and the required **parameters**, one chooses ***Heuristic Algorithms/Beam Search (attach)*** in the menu ***Algorithms***.

Settings

Beam width: Here one fixes how many of the partial sequences generated in each step should be considered further.

Selection method: Here one fixes how the chosen number of partial sequences considered further (**beam width**) should be selected. The partial sequences are evaluated by their objective function values.

- **INSERT1:** From the set of *all* child-sequences in one step, select the best ones.
- **INSERT2:** For any parent-sequence, exactly one (best) child-sequence is chosen, independently whether this is globally promising or not.

Selection criterion: This option is only available in case of $C_{\max}$ problems. It gives an alternative evaluation method for partial sequences.

- **CLAST:** A partial sequence is evaluated by the cost of a longest path through the operation which has been inserted last.
- **LB(Sum_Ci):** A partial sequence is evaluated by the Sum_Ci objective function value.
- **OBJECTIVE:** The objective function value is used for evaluating a partial sequence. This is the standard method.

Tie Breaking: If the selection of an operation to be appended is not uniquely determined, this option fixes by which rule an operation is chosen among the candidate operations.

- **FCFS** (first come - first served): The next sequenced operation is the one which entered the queue first.

- **LPT** (longest processing time): The next sequenced operation is the one which has the largest processing time.
- **SPT** (shortest processing time): The next sequenced operation is the one which has the shortest processing time.
- **RANDOM:** The operation is randomly chosen.

Invocation in the Autoalg input file

In the input file for the **Autoalg-function** (→ 7.3), beam search appending procedures are activated as follows:

```
<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE beam
  string MODE ATTACH
  string ATTACH_WHAT Machines+Jobs
  string BREAK_TIES FCFS
  string INS_METHOD INSERT1
  string CRITERION OBJECTIVE
  long K_BRANCHES 5
</CONTROLPARAMETERS>
```

The sequence in which the parameters are passed is irrelevant. In case of optional parameters, here the default-value is given.
As an executable file, beam must be invoked for all beam search procedures. The information on the beam search mode is mandatory. Optionally, the parameters for the **appending criterion** (ATTACH_WHAT with options machines, jobs, machines+jobs), the **tie breaking**, the **selection method**, the **selection criterion**, and the **beam width** can be set.

4.2.4 Beam Search Procedures with Insertion Technique

Beam Search procedures are a heuristic variant of the **Branch & Bound Procedure** (→ 4.1.1). In contrast to them, the number of investigated partial solutions is limited in any vertex of the solution tree. This leads to an incomplete search in the set of feasible solutions which can possibly exclude all optimal solutions. However, such procedures are substantially faster than a branch & bound algorithm. The quality of the solution depends both on the chosen problem type and on the chosen parameters.
In case of an **insertion** procedure, the sequence in which the operations are added to a partial sequence is settled in advance. For any partial sequence and any new operation, several feasible positions are considered at which the new operation with one of the possible ranks can be inserted. In addition to the insertion procedure, an **appending** procedure (→ 4.2.3) is also implemented in LiSA.

Invocation

After the input of the **problem type** and the required **parameters**, one chooses ***Heuristic Algorithms/Beam Search (insert)*** in the menu ***Algorithms***.

Settings

Beam width: Here one fixes how many of the partial sequences generated in any step are considered further.

Insertion order: Here the sequence is fixed in which the operations are added to the partial sequences.

- **MACHINEWISE:** The operations are sequenced column-wise.
- **DIAGONAL:** This strategy is based on the consideration of the diagonal in a square of the size $M = \min(m, n)$.
- **QUEEN_SWEEP:** This strategy is based on the consideration of independent sets as solutions of a M queens problem on a $M \times M$ chess board ($M = \min(m, n)$).
- **RANDOM:** The operations are inserted in random order.
- **LPT** (longest processing time): The operations are ordered according to decreasing processing times.
- **ECT** (earliest completion time): The operation to be inserted next is chosen in such a way that the completion time for the new partial schedule becomes minimal. In contrast to all other strategies for selecting an insertion order, this selection is not globally made. In any step, for each partial sequence considered, the operation is determined which is inserted next.
- **SPT** (shortest processing time): The operations are ordered according to increasing processing times.

Selection method: Here one fixes how the chosen number of partial sequences investigated further (**beam width**) is selected. The partial sequences are evaluated by their objective function values.

- **INSERT1:** From the set of *all* child-sequences in one step, select the best ones.
- **INSERT2:** For any parent-sequence, exactly one (best) child-sequence is chosen, independently whether this is globally promising or not.

Selection criterion: This option is only available for $C_{\max}$ problems. It gives an alternative evaluation method for partial sequences.

- **OBJECTIVE:** The objective function value is used for evaluating a partial sequence. This is the standard method.
- **CLAST:** A partial sequence is evaluated by the cost of a longest path through the operation which has been inserted last.

Invocation in the Autoalg input file

In the input file for the **Autoalg-function** (→ 7.3), beam search insertion procedures are activated as follows:

```
<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE beam
  string MODE INSERT
  string INS_ORDER LPT
  string INS_METHOD INSERT1
  string CRITERION OBJECTIVE
  long k_BRANCHES 5
</CONTROLPARAMETERS>
```

The sequence in which the parameters are passed is irrelevant. In case of optional parameters, here the default-value is given.
As an executable file, beam must be invoked for all beam search procedures. The information on the beam search mode is mandatory as it is for the **insertion order**. Optionally, the parameters for the **selection method**, the **selection criterion**, and the **beam width** can be set.

4.3 Universal Iterative Improvement Procedures

Iterative improvement procedures try to improve step by step a given starting solution which has been found e.g. by a fast heuristic. To this end, several neighborhoods between feasible solutions are considered, which can be searched by different strategies. Such procedures are often used when looking for excellent approximation solutions in a medium-size running time.
In addition to the simple **iterative search** (→ 4.3.1) and the **tabu search** (→ 4.3.2), in LiSA there are also implemented extended neighborhood search procedures such as **simulated annealing** (→ 4.3.3) or **threshold accepting** (→ 4.3.4). In addition, there are available further metaheuristic algorithms such as **genetic algorithms** (→ 4.3.5) and **ant colony procedures** (→ 4.3.6).
The quality of the solution depends on the used procedures, the quality of the starting solution and the number of generated solutions.

Neighborhoods

In all neighborhood search procedures, the different neighborhoods play a particular role. A neighborhood graph describes which sequences can be generated from a given sequence in one step. The choice of the neighborhood may have a large influence on the quality of the procedure. In LiSA, the following neighborhoods are considered but not every neighborhood is available for any problem type:

- **k_API** (*k*-adjacent pairwise interchange)**:** *k + 1* adjacent operations are randomly re-sequenced.
- **SHIFT:** An operation is shifted in the sequence.
- **PI** (pairwise interchange)**:** Two arbitrary operations are interchanged.
- **TRANS** (transpose)**:** A sub-sequence of operations on a machine is reversed.
- **CR_API** (critical API)**:** A $C_{\max}$ critical operation is interchanged with a directly adjacent operation.

- **SC_API** (semi-critical API)**:** A C_{max} critical operation or with a specific probability also another operation is interchanged with a directly adjacent operation.
- **BL_API** (block API)**:** A $C_{\max}$ critical block-end-operation is interchanged with a directly adjacent operation.
- **CR_SHIFT** (critical SHIFT)**:** A $C_{\max}$ critical operation is shifted in the sequence.
- **BL_SHIFT** (block SHIFT)**:** A $C_{\max}$ critical block-end-operation is shifted in the sequence.
- **CR_TRANS** (critical TRANS) **:** The sequence of jobs between two critical operations on a machine is reversed.
- **SC_TRANS** (semi-critical TRANS) **:** The sequence on a machine is not always reversed between two critical operations but with a small probability also between non-critical operations.
- **3_CR:** A $C_{\max}$ critical operation is interchanged with a directly adjacent operation. In addition, both the predecessor and the successor are interchanged with other operations.
- **k_REINSERTION:** k arbitrary operations are removed from the sequence and then re-inserted.
- **CR_TRANS_MIX:** In 75% of all cases, a neighbor sequence is generated in the CR_TRANS neighborhood, otherwise in the CR_API neighborhood.
- **CR_SHIFT_MIX:** In 75% of all cases, a neighbor is generated in the CR_SHIFT neighborhood, otherwise in the CR_API neighborhood.

4.3.1 Iterative search

In an iterative search procedure, the neighborhood is searched in any step and the first improving solution is accepted. This always ends in a local optimum the quality of which can, however, be rather bad.

Invocation

After the input of the **problem type** and the required **parameters**, one chooses ***Heuristic Algorithms/Iterative Improvement*** in the menu ***Algorithms***.

Settings

Created solutions: This fixes the maximal number of solutions to be generated.

Abort when stuck for (stopping criterion): Here a stopping criterion in form of a maximal number of iterations without an improvement of the best solution found so far can be fixed. If this value is larger than the **number of solutions**, it does not apply.

k for k_API or k_REINSERTION: Here a parameter k is settled, if the **neighborhood** k_API or k_REINSERTION has been chosen.

Abort when reaching objective (stopping criterion): Here one can settle an upper bound for the acceptance of an objective function value. The neighborhood search is finished if the settled objective function value has been reached.

Neighborhood: Here the choice of the neighborhood is fixed, i.e., in which ways neighbors are generated from a given sequence.

- **k_API**
- **SHIFT**
- **PI**
- **CR_API**
- **BL_AP**
- **CR_SHIFT**
- **BL_SHIFT**
- **3_CR**
- **k_REINSERTION.**

Invocation in the Autoalg input file

In the input file for the **Autoalg-function** (→ 7.3), iterative search is activated as follows:

```
<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE nb_iter
  string METHOD IterativeImprovement
  string TYPE RAND
  string NGBH k_API
  long k 1
  long STEPS 5000
  long NUMB_STUCKS 214748000
  double ABORT_BOUND -214748000
</CONTROLPARAMETERS>
```

The sequence in which the parameters are passed is irrelevant. In case of optional parameters, here the default-value is given.
As an executable file, nb_iter must be invoked for all neighborhood search procedures. For iterative search, method IterativeImprovement must be activated. The settlements on the selection of the **neighborhood** are mandatory (possibly in addition with the corresponding parameter for k). Then, optionally, the following parameters can be set: the **number of generated sequences**, the **number of stagnations** after which the procedure is stopped, and possibly a **stopping bound**.

4.3.2 Tabu Search

Tabu search is a simple extension of iterative search. In order to avoid a quick stagnation in a bad local optimum, in any step the neighbor with the best objective function value is accepted. In order to avoid that after the acceptance of a worse solution one goes back to the old solution, recently visited solutions are stored in a tabu list, and they are forbidden to be re-considered in the search. In order to limit the computational time required for the comparisons of any solution with those in the list, the length of the tabu list is restricted.

Invocation

After the input of the **problem type** and the required **parameters**, one chooses ***Heuristic Algorithms/Tabu Search*** in the menu ***Algorithms***.

Settings

Created solutions: Here the maximal number of generated solutions is fixed.

Tabu list length: This parameter fixes how many previous solutions are kept in the memory of the procedure which cannot be chosen again in the current step.

Number of neighbors: Here it is settled how many neighbors should be generated in each iteration.

k for k_API or k_REINSERTION: Here the parameter k is settled, if the **neighborhood** k_API or k_REINSERTION has been chosen.

Abort when reaching objective (stopping criterion): Here one can settle an upper bound for the acceptance of an objective function value. The neighborhood search is finished if the settled objective function value has been reached.

Neighborhood: Here the choice of the neighborhood is fixed, i.e., in which way neighbors are generated from a given sequence.

- **k_API**
- **SHIFT**
- **PI**
- **CR_API**
- **BL_API**
- **CR_SHIFT**
- **BL_SHIFT**
- **3_CR**
- **k_REINSERTION.**

Invocation in the Autoalg input file

In the input file for the **Autoalg-function** (→ 7.3), tabu search is activated as follows:

```
<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE nb_iter
  string METHOD TabuSearch
  string TYPE RAND
  string NGBH k_API
  long k 1
  long STEPS 1
  long TABULENGTH 1
  long NUMB_NGHB 50
  double ABORT_BOUND -214748000
</CONTROLPARAMETERS>
```

The sequence in which the parameters are passed is irrelevant. In case of optional parameters, here the default-value is given.
As an executable file, nb_iter must be invoked for all neighborhood search procedures. For tabu search, method TabuSearch must be activated. The settlements on the selection of the **neighborhood** are mandatory (possibly in addition with the corresponding parameter for k). Then, optionally, the following parameters can be set: the **number of generated sequences**, the **length of tabu list**, the **number of generated neighbors**, and possibly a **stopping bound**.

4.3.3 Simulated Annealing

Simulated annealing is an extended variant of iterative search. Here in any step, the improvement can also be negative with a specific decreasing probability. This should avoid to stagnate quickly in a bad local optimum. The probability for accepting a worse solution depends on the 'temperature' which is decreased during the search process. The whole cooling process may consist of several cooling periods or cycles, where the temperature is re-set to the initial value at the beginning of each period.

Invocation

After the input of the **problem type** and the required **parameters**, one chooses ***Heuristic Algorithms/Simulated Annealing*** in the menu ***Algorithms***.

Settings

Epoch length: Here the length of the so-called epoch can be settled. During an epoch the temperature is constant. At the beginning of the next epoch, the temperature is reduced.

Number of neighbors: Here the number of neighbors is fixed which are generated in each iteration.

Iteration steps: Here the maximal number of iterations is settled.

Abort when stuck for (stopping criterion): Here a stopping criterion in form of a maximal number of iterations without an improvement of the best solution found so far can be fixed. If this value is larger than the **number of solutions**, it does not apply.

k for k_API or k_REINSERTION: Here parameter k can be fixed, if the **neighborhood** k_API or k_REINSERTION has been chosen.

Start temperature: This parameter fixes the initial temperature for the cooling process. If during the cooling process the final temperature is reached, the temperature is re-set to the initial value and a new period starts.

End temperature: This parameter fixes the final temperature for a cooling period. In dependence on the **cooling parameter**, the **epoch length** and the **number of solutions**, this temperature can be reached significantly before the number of generated solutions has been generated or after that. Thus, in the first case, there are several cooling periods while in the second case, there is only one incomplete cooling period.

Cooling parameter: This parameter controls the speed of the cooling process. Depending on the chosen **cooling scheme**, it determines how the temperature is calculated from the current temperature for the next epoch.

Abort when reaching objective (Stopping criterion): Here one can settle an upper bound for the acceptance of an objective function value. The neighborhood search is finished if the settled objective function value has been reached.

Cooling scheme: The cooling process of the temperature from one epoch to the next one can be controlled by several decreasing functions. For each scheme, the **cooling parameter** p $(0 < p < 1)$ can be explicitly settled.

- **LINEAR**: The new temperature is determined from the old one as follows: $T^{new} = T^{old} - p_L \cdot T^{start}$.
- **GEOMETRIC**: The new temperature is determined from the old one as follows: $T^{new} = p_G \cdot T^{old}$.
- **LUNDYANDMEES**: The new temperature is determined from the old one as follows: (according to Lundy-Mees): $T^{new} = T^{old}/(1 + p_{LM} \cdot T^{old})$.

Neighborhood: Here the definition of the neighborhood is fixed, i.e., in which way new sequences are generated from the given sequence.

- **k_API**
- **SHIFT**
- **PI**
- **CR_API**
- **BL_AP**
- **CR_SHIFT**
- **BL_SHIFT**
- **3_CR**
- **k_REINSERTION.**

Invocation in the Autoalg input file

In the input file for the **Autoalg-function** (→ 7.3), Simulated annealing is activated as follows:

```
<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE nb_iter
  string METHOD SimulatedAnnealingNew
  string NGBH k_API
  long k 1
  long EPOCH 100
  long NUMB_NGHB 1
  long STEPS 1
  long NUMB_STUCKS 214748000
  double TSTART 20
  double TEND 0.9
  double COOLPARAM 0.0005
  double ABORT_BOUND -214748000
  string COOLSCHEME LUNDYANDMEES
  string TYPE ENUM
</CONTROLPARAMETERS>
```

The sequence in which the parameters are passed is irrelevant. In case of optional parameters, here the default-value is given.
As an executable file, nb_iter must be invoked for all neighborhood search procedures. For simulated annealing, method SimulatedAnnealingNew must be activated. The information on the choice of the **neighborhood** is mandatory (possibly also with the corresponding parameter for k). Then, optionally, the parameters for the **number of epochs**, the **number of iterations**, and the **number of stagnations** when the procedure is stopped, the **start temperature**, the **end temperature**, the **cooling parameter**, possibly a **stopping bound**, the **cooling scheme** and the way of the **generation of the neighbors** (randomly or enumerative) (variable TYPE with options ENUM and RAND).

4.3.4 Threshold Accepting

Threshold Accepting is also an extended version of iterative search. In contrast to local search, in any step also small deteriorations are accepted. The threshold for accepting worse solutions is decreased during the search and tends to zero. In LiSA, a linear reduction of the threshold is implemented.

Invocation

After the input of the **problem type** and the required **parameters**, one chooses ***Heuristic Algorithms/Threshold Accepting*** in the menu ***Algorithms***.

Settings

Created solutions: Here the maximal number of generated solutions is fixed

Abort when stuck for (stopping criterion): Here a stopping criterion in form of a maximal number of iterations without an improvement of the best solution can be fixed. If this value is larger than the **number of solutions**, it is not applied.

Initial threshold: This parameter gives the maximal allowed deterioration in per mille.

Maximal number of stagnations: In order to avoid trapping into a local optimum in case of a low threshold, the threshold is re-set to the initial value after the given number of iterations without an improvement of the currently best solution.

k for k_API or k_REINSERTION: Here the parameter *k* is fixed, if the **neighborhood** k_API or k_REINSERTION has been chosen.

Abort when reaching objective (stopping criterion): Here one can settle an upper bound for the acceptance of an objective function value. The neighborhood search is finished if the settled objective function value has been reached.

Neighborhood: Here the definition of the neighborhood is settled, i.e., in which way neighbors are generated from the current solution.

- **k_API**
- **SHIFT**
- **PI**
- **CR_API**
- **BL_API**
- **CR_SHIFT**
- **BL_SHIFT**
- **3_CR**
- **k_REINSERTION.**

Invocation in the Autoalg input file

In the input file for the **Autoalg-function** (→ 7.3), Threshold accepting is activated as follows:

```
<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE nb_iter
  string METHOD ThresholdAccepting
  string TYPE RAND
  string NGBH k_API
  long k 1
  long STEPS 1
  long NUMB_STUCKS 214748000
  long PROB 1
  double ABORT_BOUND -214748000
  long MAX_STUCK 30
</CONTROLPARAMETERS>
```

The sequence in which the parameters are passed is irrelevant. In case of optional parameters, here the default-value is given.
As an executable file, nb_iter must be invoked for all neighborhood search procedures. For Threshold accepting, method ThresholdAccepting must be activated. The information on the choice of the **neighborhood** is mandatory (possibly also with the corresponding parameter for k). Then, optionally, the parameters for the **number of generated solutions**, the **number of stagnations** after which the procedure stops, the **initial threshold** and possibly a **stopping bound**.

4.3.5 Genetic Algorithms

Genetic algorithms are metaheuristics which generate from a population of starting solutions new solutions by means of random changes. They imitate the biological concepts of mutation and recombination of two individuals. In case of a *mutation*, the rank of a randomly chosen individual from the current population is changed. This modified sequence is then manipulated such that a new solution for the underlying scheduling problem is obtained.
In case of a *crossover,* a randomly chosen set of ranks of two randomly chosen individuals are interchanged. Then both resulting sequences are transformed into feasible sequences for the problem under consideration. In this way, new individuals are generated. These new individuals are included into the next generation if, according to a fitness function, they are not less fit than their corresponding predecessors.
The initial population is formed by randomly generated sequences (individuals). Using the **Autoalg-function** ($\rightarrow$ 7.3), it is possible to include also already rather good solutions determined by chosen constructive heuristics into the initial population.

Invocation

After the input of the **problem type** and the required **parameters**, one chooses ***Heuristic Algorithms/Genetic Algorithms*** in the menu ***Algorithms***.

Settings

Population size: Here one fixes how many solutions (individuals) should belong to a population.

Number of generations: This parameter settles the number of iterations of this procedure.

Random seed: This parameter serves as input for the random generation of the initial population. In addition, the random decisions in any step are influenced. If the procedure is executed without local search steps, it can be exactly repeated with the same parameter.

Local improvement steps: An individual generated by mutation or crossover can often be easily improved by a simple **local search** ($\rightarrow$ 4.3.1). Here it is settled how many iterations of local search are applied to any new solution.

Mutation probability: This parameter gives the probability with which an individual is modified by a mutation in the current iteration.

Crossover probability: This parameter gives the probability with which two individuals are modified by a crossover in the current iteration.

Population initialization: Here it is settled which type of sequences are randomly generated for the initial population.

- **RANDOM_ORDER:** Sequences are generated the schedules of which can be active or non-delay.
- **ACTIVE_DISPATCH:** Sequences are generated the schedules of which are exclusively active ones.
- **NON_DELAY_DISPATCH:** Sequences with non-delay schedules are exclusively generated.

Local improvement: Here the neighborhood is fixed in which new solutions should be improved. In case of the setting *(disabled)*, no local search is performed.

- **API**
- **SHIFT**
- **PI**
- **CR_API**
- **BL_API**
- **CR_SHIFT**
- **BL_SHIFT**
- **3_CR**
- **(disabled).**

Invocation in the Autoalg input file

In the input file for the **Autoalg-function** (→ 7.3), genetic algorithms are activated as follows:

```
<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE ga
  long POP_SIZE 20
  long NUM_GEN 100
  string INIT RANDOM_ORDER
  string FITNESS OBJECTIVE
  double M_PROB 0.25
  double C_PROB 0.35
  long SEED 1234567890
  long IMPR_STEPS 10
  string L_IMPR (disabled)
</CONTROLPARAMETERS>
```

The sequence in which the parameters are passed is irrelevant. In case of optional parameters, here the default-value is given.
As an executable file, ga must be invoked for all genetic algorithms. Then the information about the **population size**, the **number of generations**, the method for the **population initialization** (INIT), the **mutation probability** (M_PROB) and the **crossover probability** (C_PROB) is mandatory. Then, optionally, the parameters for the **random seed**, the number of **local improvement steps**, and the choice of the neighborhood of **local improvement** can be settled.
In addition, one can influence the choice of the **fitness function** (FITNESS) by the invocation of a genetic algorithm using the Autoalg input file. This function determines which individuals are included into the next generation. Here one can select two options: by OBJECTIVE, one chooses the corresponding objective function value as the fitness function. Alternatively, one can also choose SUM_Ci2.

4.3.6 Beam Ant Colony Procedures

Ant colony optimization is a metaheuristic for optimization problems which are difficult to solve. Based on a probabilistic construction mechanism, a solution is constructed by tree search. Using beam search procedures on this tree, a hybrid algorithm is obtained. The available algorithm follows the ideas by Blum [4].

Invocation

After the input of the **problem type** and the required **parameters**, one chooses ***Heuristic Algorithms/Beam Ant Colony*** in the menu ***Algorithms***.

Settings

Number of beam extensions: Here the number of operations to be inserted in one extension step is settled. This value is only taken into account, if the parameter **Determination of beam extensions** is set to FIXED.

Number of runs: Number of single runs. Since the ants are successively started, this corresponds to the number of ants in the whole population.

Convergence factor for re-start: This value gives a lower bound for the convergence factor from which a new calculation of the pheromone values is done.

Evaporation rate: This parameter characterizes the influence which a new best sequence has on the calculation of the pheromone values. This value should be between 0 and 1.

Influence of earliest completion time: This holds only for the **Strategy of selection of extensions** = SORTED.

Influence of pheromone values: This holds only for the **Strategy of selection of extensions** = SORTED.

Influence of randomness: This holds only for the **Strategy of selection of extensions** = SORTED objective function value.

Determination of beam extensions:

- **MED:** Number of free operations / 2.
- **LDS:** At the beginning as MED, in the second half fixed as 2.
- **FIXED:** Fixed number.

Strategy of selection of extensions:

- **ORIGINAL:** This corresponds to the method described in the literature.
- **SORTED:** Here the particular operations are weighted and sorted according to these values.

Pre-selection of extensions:

- **NONE:**
- **ACTIVE:** Only active schedules are considered.

Invocation in the Autoalg input file

In the input file for the **Autoalg-function** (→ 7.3), Beam Ant Colony procedures are activated as follows:

```
<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE beam_aco
  long BEAM_WIDTH 32
  double UPPER_BOUND 10000000
  double LOWER_BOUND 0
  string EXTENSION_STRATEGY FIXED
  long STEPS 2500
  double CONVERGENCE_FACTOR 0.98
  long FIXED_KEXT 1
  string APPEND_STRATEGY SORTED
  string PRE_SELECTION ACTIVE
  double WEIGHT_EST 1.8
  double WEIGHT_TIJ 2
  double WEIGHT_RAND 1
  double EVAPORATION_RATE 0.3
</CONTROLPARAMETERS>
```

The sequence in which the parameters are passed is irrelevant. In case of optional parameters, here the default-value is given.
As an executable file, beam_aco must be invoked for beam ant colony procedures. Then, optionally, the following parameters can be activated: The information about the **beam width**, the **upper bound**, the **lower bound**, **determination of beam extensions** (EXTENSION_STRATEGY), **number of runs** (STEPS), **convergence factor for re-start**, **number of beam extensions** (FIXED_KEXT), the strategy for the **selection of extensions** (APPEND_STRATEGY), **pre-selection of extensions**, the **influence of earliest completion time** (WEIGHT_EST), the **influence of pheromone values** (WEIGHT_TIJ), and the **influence of randomness**.

4.4 Algorithms for Solving Special Problems

4.4.1 Brucker's Branch & Bound Algorithm

This exact algorithm has been developed and implemented by the team of Peter Brucker at the University of Osnabrück. There exists a variant for **open-shop problems** and a variant for **job-shop problems**.

Invocation

After the input of the **problem type** and the required **parameters**, one chooses ***Exact Algorithms/Brucker's Open-Shop B&B*** and ***Exact Algorithms/Brucker's Job-Shop B&B***, respectively, in the menu ***Algorithms***.

Settings

For the **open-shop** algorithm, the choice of a heuristic for generating sequences must be made. For the job-shop algorithm, no options are available.

Heuristic: The algorithm can use several heuristics for generating or completing sequences. (There exist more than the two heuristics mentioned here. However, they are deactivated since they do not work with incomplete sets of operations.)

- **LB_PREC_RULE:** This heuristic is based on a dispatching rule which results from the calculation of a lower bound.
- **LB_PREC_RULE_VAR:** This heuristic is a variation of the above heuristic.

Invocation in the Autoalg input file

In the input file for the **Autoalg-function** (→ 7.3), the branch & bound algorithm by Brucker is activated as follows:

```
<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE os_bb_wo
  string HEURISTIC MIN_MATCHING
</CONTROLPARAMETERS>
```

As executable file, there must be invoked os_bb_wo for **open-shop** problems and js_bb_br for **job-shop** problems. Then, for open-shop problems, a **heuristic** can be selected. For job-shop problems, there are no further parameters.

4.4.2 Two-Machine Problems

For problems with 2 machines, the Johnson rule yields an optimal solution for the **flow-shop** problem with minimizing the makespan as well as for the extensions to the **job-shop** and **open-shop** case, i.e., it works for the problem type $\alpha \in \{F2, J2, O2\}$.

Invocation

After the input of the **problem type** (choose under α in addition to F, J or O also ***given number of machines*** since it has the initial value 2) and the required **parameters**, one chooses ***Exact Algorithms/Johnson Rule*** in the menu ***Algorithms***.

Invocation in the Autoalg input file

In the input file for the **Autoalg-function** ($\rightarrow$ 7.3), the Johnson rule is activated as follows:

```
<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE johnson
</CONTROLPARAMETERS>
```

For an $O2 \mid\mid C_{max}$ problem, the LAPT (longest alternating processing times) rule also yields an optimal solution.

Invocation

After the input of the **problem type** (note that $\alpha = O2$) and the required **parameters**, one chooses ***Exact Algorithms/LAPT-Rule*** in the menu ***Algorithms***.

Invocation in the Autoalg input file

In the input file for the **Autoalg-function** ($\rightarrow$ 7.3), the LAPT rule is activated as follows:

```
<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE pr
</CONTROLPARAMETERS>
```

4.4.3 Single Machine Problems with Minimizing the Number of Late Jobs

This algorithm minimizes the total number of late jobs in a single machine environment. It finds an exact solution in polynomial time.

Invocation

After the input of the **problem type** and the required **parameters**, one chooses ***Exact Algorithms/Single Machine - Sum Ui*** in the menu ***Algorithms***.

Invocation in the Autoalg input file

In the input file for the **Autoalg-function** ($\rightarrow$ 7.3), the single machine algorithm is activated as follows:

```
<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE single_sum_ui
</CONTROLPARAMETERS>
```

As executable file, there must be invoked single_sum_ui. No parameters are passed to the algorithm.

4.4.4 Open-Shop Problem with Makespan Minimization and pmtn (Allowed Preemptions)

The algorithm by Gonzales/Sahni [14] has been implemented. However, since the LiSA core cannot present Gantt charts with preemptions and in addition, a sequence and a schedule with pmtn is not contained in the graphical interface, the result is written into a file. One uses the LiSA interface to input the **problem type** and the required **parameters**. Then one invokes ***Exact Algorithms/Gonzales & Sahni (Research)***. LiSA stores the results in the file **algo_out.xml** in the directory `~/.lisa/proc`. At each position (ij) of the matrix, one can find an information about the sub-operations into which (ij) is split. Further information about the extension of the block-matrices model to problems with preemptions can be found in Bräsel/Hennes [6].

4.5 Heuristic Algorithms for Special Problems

4.5.1 The Shifting Bottleneck Heuristic

The shifting bottleneck heuristic generates an approximate solution for the **job-shop** and the **flow-shop** problem, respectively, with minimizing the makespan by solving successively single machine problems. In the first step, the weight of a critical path in the corresponding disjunctive graph is determined, where all non-oriented edges have been deleted. This value is a lower bound LB for the makespan C_{max}. Step by step, one now solves for any machine a single machine problem with release dates of the jobs (heads of the operations on the corresponding machine), due dates (LB minus tails of the operations on the considered machine), precedence constraints between the jobs (precedence constraints between the operations on the considered machine from the current disjunctive graph without disjunctive edges) and the minimization of L_{max}. The exact solution of this problem is used as the job order of the operations on the considered machine. In addition, after fixing the job order on a machine, one tries to improve already fixed job orders further, i.e., one shifts the bottleneck.

Invocation

After the input of the **problem type** and the required **parameters**, one chooses ***Heuristic Algorithms/ShiftingBottleneck*** in the menu ***Algorithms***.

Settings

Speed: By this, one can select a fast variant which might generate infeasible schedules or a slower variant which contains the complete heuristic.

- **FAST** The fast variant does not consider the precedence graph of the operations when solving the single machine problems. As a consequence, there can be generated combinations of the machine and job orders which are infeasible, i.e., which contain cycles, what LiSA returns as an error message.
- **SLOW** Here, the shifting bottleneck heuristic is completely implemented.

Invocation in the Autoalg input file

In the input file for the **Autoalg-function** ($\rightarrow$ 7.3), the shifting bottleneck heuristic is activated as follows:

```
<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE bottle
  string SPEED SLOW
</CONTROLPARAMETERS>
```

Computational tests with randomly generated machine orders have created cycles in the FAST variant only for problems with more than 20 machines and 20 jobs.

4.5.2 The Flow-Shop Heuristic

This heuristic determines a heuristic solution for the **flow-shop** problem from the set of all permutation schedules (i.e., the same job order is chosen on all machines). To this end, the insertion order of the jobs according to non-increasing sums of their processing times is used. One starts with the first job of this order and sequences the second job directly before and after the first job. Choosing the partial sequence with the smallest makespan value, the third job is sequenced on position 1, 2 and 3, etc. (beam-insert with width 1 on the set of all permutation schedules).

Invocation

After the input of the **problem type** and the required **parameters**, one chooses ***Heuristic Algorithms/Flow Shop Heuristic*** in the menu ***Algorithms***.

Invocation in the Autoalg input file

In the input file for the **Autoalg-function** ($\rightarrow$ 7.3), the flow-shop heuristic is activated as follows:

```
<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE fsheur
</CONTROLPARAMETERS>
```

As an executable file, there must be invoked fsheur. No further parameters are expected.

Chapter 5

Output

As soon as LiSA has determined a schedule or a file with a schedule has been opened, LiSA always displays the corresponding Gantt chart in the main window. Then in the menu item ***View*** , the following information on the current schedule can be activated:

- Parameters
- Sequence $\rightarrow$ 5.1
- Sequence graph $\rightarrow$ 5.2
- Schedule $\rightarrow$ 5.3
- Gantt chart $\rightarrow$ 5.4

By the call of the menu item ***File/Print***, the current content of the main window is printed. In addition, it is possible to store the current results as an XML file, siehe **output file** $\rightarrow$ 5.5.

5.1 Sequence

A sequence is a feasible combination of the machine orders of the jobs and the job orders on the machines. It is described by the rank matrix of the vertices (operations) of the (acyclic) sequence graph (see 5.2). The rank of a vertex v in an acyclic digraph is defined as the number of vertices on a longest path to vertex v, where v is included.

It can be noted that any sequence is a latin rectangle which satisfies the so-called sequence condition: If there is the number $a > 1$ at position (i, j) in the latin rectangle, then number $a - 1$ exists either in row i or in column j or in both. A latin rectangle $LR[m, n, r]$ is a matrix of order $m \times n$ with entries from a set $B = \{1, ..., r\}$, where any number from B occurs at most once in any row and column.

For further information: see Block-Matrices-Model $\rightarrow$ 2.3.2

Invocation

After the input of the problem instance and the first application of an algorithm, the resulting schedule is always represented by a Gantt chart. After selecting the menu item ***View/Sequence***, the sequence is displayed in the main window.

5.2 Sequence Graph

The sequence graph $G(MO, JO)$ is an acyclic digraph with operations as vertices. The directed edges (arcs) correspond to the direct precedence constraints between operations in the machine and job orders of the operations, i.e., there is an arc from operation (ij) to operation (kl), if ($i = k$ and job J_i has to be processed on machine M_l directly after M_j) or ($j = l$ and on machine M_j, job J_k is processed directly after job J_i) holds.
For further information: see the Block-Matrices-Model → 2.3.2

Invocation

After the input of a problem instance and a first application of an algorithm, the resulting schedule is always represented by a Gantt chart. After the selection of the menu item ***View/Sequence graph***, the sequence graph is displayed in the main window.

Extras

In case of minimizing the makespan, arcs of the critical path(s) are colored in red.

5.3 Schedule

If the operations are weighted by the corresponding processing times in a sequence graph, a schedule (a timetable of processing the operations) can be constructed. The set of schedules belonging to a sequence graph has infinitely many elements, but there is a unique sequence belonging to a schedule. A schedule is called:

- **semi-active**, if all operations are processed as early as possible such that the underlying sequence is not violated.
- **active**, if there does not exist an idle time on the machines, in which an operation sequenced later can be completely processed.
- **non-delay**, if there is no unforced delay of an operation, i.e., if at any time when a machine is available and a job is waiting for the processing, the processing of this operation has to be started.

Any non-delay schedule is active, and any active schedule is semi-active, but the opposite does not necessarily hold.

In LiSA, semi-active schedules are described by the matrix $C = [c_{ij}]$, where c_{ij} denotes the completion time of (ij).
For further information: see Block-Matrices-Model → 2.3.2

Invocation

After the input of a problem instance and the first application of an algorithm, the resulting schedule is always presented by a Gantt chart. After the selection of the menu item ***View/Schedule***, the schedule is displayed in the main window.

Extras

In case of minimizing the makespan, those values c_{ij} are colored in red which belong to operations (ij) which are passed by at least one critical path.

5.4 Gantt Chart

The Gantt chart (named after Henry Laurence Gantt, 1861-1919) visualizes the timetable of the processing of the operations, i.e., it visualizes a schedule.

Invocation

After the input of a problem instance and the first application of an algorithm, the resulting schedule is always represented by a Gantt chart. After the selection of the menu item ***View/Gantt chart***, the schedule is displayed in the main window. Several representations of the Gantt chart can be settled in the menu item ***Options/Gantt chart*** as soon as a schedule is available.

Settings

Orientation:

- **Machine Oriented:** The operations of the jobs appear as a bar over the time axis (x-axis) on the machines.
- **Job Oriented:** The operations on the machines appear as a bar over the time axis (x-axis) for the jobs (y-axis). Here also possibly given release and due dates are displayed.

Special Emphasis:

- **None:** Standard: The operations are colored with up to 23 well distinguishable colors.
- **Critical Path:** In case of minimizing the makespan, the operations belonging to a critical path are colored in red, all other operations are colored in gray.
- **Assign Colors to Some Jobs:** It is possible to choose explicitly colors for at most 5 jobs or machines.

Extras

Manipulations of the Gantt chart:

- **Zoom:** The view can be zoomed. To this end, click on the magnifier symbol or activate the zoom mode in the menu ***Extras***. Open with the mouse a rectangle in order to make this to be the new view. In the chosen zoom mode, one can scroll through the Gantt chart. In order to view again the whole Gantt chart, deactivate the zoom mode.
- **Sequence editor:** A double click on an operation starts the **manipulation of sequences and schedules** (→ 6.2) which can be used to shift the chosen operation in the corresponding machine or job order.

For further information, see: Manipulation of Sequences and Schedules → 6.2.

5.5 Output File

By the call of the menu item ***File/Save as***, the following information is stored in an XML file, where the name of the file and the directory can be arbitrarily chosen:

- Problem type in the $\alpha \mid \beta \mid \gamma$ notation;
- n (number of jobs) and m (number of machines);
- Matrix of the processing times;
- Set of operations (by means of a Boolean matrix);
- Sequence as a latin rectangle with sequence condition;
- Schedule as a matrix of the completion times of all operations.

This file can be used again as input file for LiSA.
For further information on the file format: The File Format in LiSA $\rightarrow$ 2.5

Chapter 6

Extras

LiSA contains several interesting internal tools. Some of them are referred to in this chapter.

6.1 Complexity Status of a Problem

LiSA is able to give the complexity status of a problem in the $\alpha \mid \beta \mid \gamma$ notation. The determination of the complexity is done by analysing a BibTeX-database (file classify.bib) which is based on the following collection of results for scheduling problems: Complexity results of scheduling problems, see: http://www.mathematik.uni-osnabrueck.de/research/OR/class/.

Invocation

As soon as a particular problem type has been settled, one can select the menu item ***Extras/Problem Classification***. In the window problem classification, it is displayed whether the problem is polynomially solvable, pseudo-polynomially solvable, NP-hard or NP-hard in the strong sense.
If the complexity of the problem considered is known, a hint to the corresponding reference is given, where this result can be found. For the output of the corresponding complete reference, the button ***Complete Reference*** must be pressed.

6.2 Manipulation of Sequences and Schedules

LiSA enables one to manipulate later a sequence or schedule which has already been determined by shifting particular operations in the corresponding machine and/or job order.

Invocation

Select an operation to be manipulated by a double click on the corresponding bar in the Gantt chart. The chosen operation (ij) is marked by a black border, and a window opens with the corresponding manipulation possibilities.

Settings

Manipulation possibilities for the selected operation $(i\,j)$

Symbol	Meaning
Source	The operation $(i\,j)$ becomes a source in the sequence graph, i.e., it does not have predecessor operations.
Sink	The operation $(i\,j)$ becomes a sink in the sequence graph, i.e., it does not have successor operations.
MO ⏮	The operation $(i\,j)$ becomes the first operation in the machine order of job i.
MO ◀	The operation $(i\,j)$ is shifted left in the machine order of job i by one position.
MO ⏭	The operation $(i\,j)$ becomes the last operation in the machine order of job i.
MO ▶	The operation $(i\,j)$ is shifted right in the machine order of job i by one position.
JO ⏫	The operation $(i\,j)$ becomes the first operation in the job order on machine j.
JO ▲	The operation $(i\,j)$ is shifted left in the job order on machine j by one position.
JO ⏬	The operation $(i\,j)$ becomes the last operation in the job order on machine j.
JO ▼	The operation $(i\,j)$ is shifted right in the job order on machine j by one position.
↪	The sequence is re-determined which was available before opening the manipulation window.

ATTENTION: If the shift of an operation creates a cycle in the graph $G(MO, JO)$, a window is opened with the error message: *cycle: modification not possible.*
In the lower part of the manipulation window, the immediate predecessors and successors of the chosen operation $(i\,j)$ are displayed. Simultaneously, the processing time p_{ij} and the completion time c_{ij} are displayed.

6.3 Irreducibility Test

This test solves different problems from the irreducibility theory, see Willenius [26]. It is applicable to open-, job- and flow-shop problems with processing times of the jobs and with the regular criteria Cmax, Lmax, SumCi, SumWiCi, SumUi, SumWiUi, SumTi und SumWiTi.
A sequence S^* reduces a sequence S, if S^* does not have a worse objective function value than S for any possible choice of the processing times. Two sequences are called similar if they have the same objective function value for any possible choice of the processing times. A sequence S^* reduces a sequence S strictly, if they are not similar and if S^* reduces the sequence S. A sequence S is called irreducible, if there is no sequence S^* which strictly reduces S. Thus, there exists a globally optimal sequence in the set of all irreducible sequences for any possible choice of the processing times.

Invocation

If LiSA knows the problem type and all parameters and a sequence is already available, the algorithm can be activated in the menu item ***Heuristic Algorithms/Irreducibilitytest (Research)*** in the menu ***Algorithms***.

Settings

Generate sequences: It is settled which sequences should be generated:

- **SIMILAR** All sequences are generated which are similar to the input sequence.
- **ALL_REDUCING** All sequences are generated which strictly reduce the input sequence, namely exactly one from each class of similar sequences. If such a sequence does not exist, LiSA gives an error message: Warning: Sequence is irreducible.
- **ITERATIVE_REDUCING** If a sequence has been found which strictly reduces the input sequence, the algorithm is interrupted and re-started with the new sequence. This is continued until an irreducible sequence has been found.

  If the input sequence is irreducible, LiSA gives an error message: Warning: Sequence is irreducible.

Return following sequences: They are stored in the file *algo_out.xml* in the directory `~/.lisa/proc`. In LiSA, the output is displayed under ***Edit/List of Sequences***.

- **ALL** All generated sequences are returned, where the number can be rather large when using the ALL_REDUCING option.
- **ONLY_IRREDUCIBLE** Only irreducible sequences are returned.

Generate sequences in random order: Selection possibilities:

- **YES** Sequences are generated by the algorithm in random order. This is only interesting when using the ITERATIVE_REDUCING option in order to obtain different results for repeated calls of the algorithm.
- **NO** Sequences are generated by the algorithm in an efficient order.

Chapter 7

On the External Work with LiSA

7.1 Algorithm Modules

All algorithms for solving scheduling problems in LiSA are externalized into modules. They are independent executables which can be invoked in LiSA after the selection in the menu ***Algorithms***. They can also be manually invoked from the command line.

Structure of a module

A module in LiSA basically consists of five particular files:

- an XML file of **document type** (→ 2.5) `algorithm` for describing the algorithm
- a help file in HTML format,
- and finally the algorithm itself in form of an independent executable,

where the first two files are required in a German and an English version.
The XML document is needed for the inclusion of the module into the main program. It can be found in the subdirectories `data/alg_desc/language/german` and `data/alg_desc/language/english`. There it is settled which problem types a particular algorithm can solve, whether an already determined solution is available or whether a problem will be solved exactly or heuristically. Moreover, parameters can be declared which are transferred to the algorithm. The detailed structure of such an XML document is described in Appendix A.
The help file serves for the human-readable description of the algorithm. It should briefly give the solution strategy, the problem types which can be solved and the description of the program parameters. It is contained in the subdirectories `doc/lisa/english/algorithm` and `doc/lisa/german/algorithm`.
The algorithm itself is contained in the subdirectory `bin`.

Command line interface

The program which implements the algorithm accepts exactly two command line parameters: an input file and an output file. The input file must contain a LiSA problem instance in XML format. It can either be of **document type** (→ 2.5) `instance` or of `solution`. The output file is generated by the module of the algorithm and contains the solution determined by the algorithm.

Transfer of parameters

Chapter 4 describes a set of parameters for any algorithm which influence its behavior. The passing of the parameters is done by means of the input file. To this end, the `<controls>` element is available which can be included into documents of the type `instance` and `solution`. It contains particular parameter settings in form of `<parameter>` elements. They arrange the assignment of a value for a parameter by an information on the data type (`integer`, `real` or `string`), the parameter name and the value.
A `<controls>` block for a call of the algorithm **iterative improvement** (→ 4.3.1) can be for instance as follows:

```
<controls>
  <parameter type="string" name="METHOD" value="IterativeImprovement" />
  <parameter type="string" name="NGBH" value="k_API" />
  <parameter type="integer" name="k" value="1" />
  <parameter type="integer" name="STEPS" value="1" />
  <parameter type="integer" name="NUMB_STUCKS" value="214748000" />
  <parameter type="real" name="ABORT_BOUND" value="-214748000" />
</controls>
```

In `instance` documents, `<controls>` blocks must be directly inserted after the `<values>` element. In `solution` documents, they are also after the `<values>` element, but before the first `<schedule>` element.

Output of run time information

Algorithm modules can pass information to the main program by means of the standard output. For that purpose, there are available four key words `PID=`, `OBJECTIVE=`, `WARNING:` and `ERROR:`.

`PID=x` gives the process-ID `x` of the current module to the main program. This information should be passed at the beginning of the execution, if possible.

`OBJECTIVE=x` returns a message about the current computation status. For instance, the current objective function value can be the output value. These values are graphically represented by the LiSA interface during the computations. The first output settles the size of the window, all further information should possibly not exceed this size.

`WARNING:msg` displays a message window with the warning `msg`. The algorithm module continues then its execution.

`ERROR:msg` displays a message window with the error message `msg` and stops the execution of the algorithm.

Manual call of an algorithm

If an algorithm should be executed independently of the LiSA interface, this can be done by the invocation of the corresponding executable file in the subdirectory `bin`. To this end, an input file must be created which contains the problem to be solved. Such files can be easily generated by storing a problem created by LiSA with the menu item ***File/Save as***

(→ 3.3). However, the parameters for the algorithm must be settled by hand (at least an empty `<controls />` tag, if all parameters should be fixed by their standard values).
For further information on the parameters of the particular algorithms, see Chapter 4 and the XML reference in Appendix A.

7.2 Inclusion of External Algorithms

To include a new algorithm module, it suffices to create the files described in the previous section. In principle, the algorithm can be implemented in any programming language which can be compiled into the binary format. However, LiSA provides particularly for C++ classes which simplify the work with scheduling problems and the implementation of algorithms. For instance, the problem description, schedules and parameters can be read from the XML files and written into them. In addition, basic and also specific data types are provided which are helpful for the treatment of scheduling problems. To use this class library, the LiSA source code package must be installed.
In the following, we assume that an algorithm should be implemented in C++.
New algorithms should obtain an own directory in the subdirectory `src/algorithm` to simplify the later creation process. There the following files must be created:

- C++ source code and header files which implement the algorithm.
- `Makefile`
- `Make.List`
- The XML and HTML files mentioned in Section 7.1. They are moved during the creation process into the corresponding directories.
- `Make.Depend` and `Make.Objects`, which are automatically created.

Source code

The algorithm itself is implemented as a usual console program, which takes the inputs described in Chapter 7.1 and writes the determined solution into an output file. As an introductory example we refer to the algorithm in the directory `src/algorithm/sample`, where the basic structure of the implementation of an algorithm is already prefabricated. Mostly it suffices to take over the source code file `sample.cpp` and to replace the marked parts by an own code. Of course, it is also possible to externalize an own code into other files.

Makefile

The following basic structure of the makefile for C++ programs can directly be adopted. Here one must only adapt the name of the program which is the name of the subdirectory, that contains the source code files of the algorithm.

```
# LiSA Sample Algorithm Makefile

# ----------------------------------------------------------------

# LiSA part: sample

PROGRAMNAME=sample

# ----------------------------------------------------------------

TOPPROGRAMPATH=../../..

# ----------------------------------------------------------------

include ../Make.Algorithm
```

Make.List

In this file, all C++- source code files are listed, which are compiled during the creation process and which are linked to an executable module file. In addition to the actual source code files of the module, here also all external dependencies must be given so that the linker can resolve all symbols. In in the source code of the module e.g. class `LISA_Matrix` is used, the corresponding source code file `matrix.cpp` in the subdirectory `src/basics` must be linked. The information on the path is done in relation to the module directory, i.e., the entry `../../basics/matrix.cpp` must be included. The corresponding source code files for the classes can be taken from the documentation on the LiSA class library.

```
CXXSOURCES=\
  sample.cpp \
  ../../basics/list.cpp \
  ../../basics/matrix.cpp \
  ../../basics/pair.cpp \
  ../../lisa/ctrlpara.cpp\
  ../../lisa/lvalues.cpp \
  ../../lisa/ptype.cpp \
  ../../main/global.cpp \
  ../../misc/except.cpp \
  ../../misc/int2str.cpp \
  ../../scheduling/mo_jo.cpp \
  ../../scheduling/schedule.cpp
```

Make.Depend and Make.Object

These files are created by a single call `make depend` in the module directory from the file `Make.List`. If the file `Make.List` is changed, they must be updated by `make depend` again.

XML and HTML documents

These documents must be available in an English and a German version (both versions of the XML document mostly differ only by translation of the name of the algorithm and the parameters). The English versions of the documents have the suffixes `_english.xml` and `_english.html`, respectively, and the German versions have the suffixes `_german.xml` and `_german.html`, respectively. During the creation process, these suffixes are replaced by `.html` and `.xml`, respectively, and the documents are moved to their destinations.

Creation process

The creation process can be started by means of the call `make` in the module directory. During this process, the source code files which are given in `Make.list` are compiled and linked to an executable file. Then all module files are copied to their destination directories (→ 7.1).

7.3 Automated Call of Algorithms

By means of the program `auto_alg`, which is contained in the directory `LiSA/bin`., the call of algorithms can be automated. In order to use this tool, one must first create an input file which serves then as a parameter for the call of the program `auto_alg`.
By means of the random generator contained in LiSA (see Taillard [24]), there are automatically generated instances of scheduling problems for which specific algorithms can be invoked. For instance, a particular instance can be automatically solved by several different algorithms and the results can be compared.
For any instance of the problem, the call `auto_alg` generates an XML file with this instance and the solutions generated by the invoked algorithms. In addition, the initial values used for the generation of the random numbers are given. This XML file can be opened by LiSA (Button: Process/List of sequences). Any solution is callable but a further processing of a particular solution deletes the list.

ATTENTION

The work with the automated call of algorithms must be done very carefully since the input file is not internally checked for correctness. In particular, the user has to take care whether the invoked algorithm is available for the problem type considered and whether the parameters for the generation of the problem instance and the algorithms are correct (take care about capital and small letters as well as space characters). Another error source is the number of algorithms to be executed.

Creation of an input file

The input file is a text file (e.g. `~/test/myproblem.alg`), which defines the problem type to be processed, the parameters for the random number generator and the algorithms to be applied. An illustration is given by the following example. It generates 10 problem instances of the type $O \mid r_i \mid \sum w_iT_i$. To any instance, first the random dispatching rule is applied and then the obtained solution is improved by iterative search.

```
<PROBLEMTYPE>
  Lisa_ProblemType= { O / r_i / SumWiTi }
</PROBLEMTYPE>

<CONTROLPARAMETERS>
  long MINPT 1
  long MAXPT 99
  long MINRI 0
  long MAXRI 100
  long MINDD 200
  long MAXDD 1000
  double MINWI 0.0
  double MAXWI 2.0
  long M 10
  long N 20
  long TIMESEED 658496
  long MACHSEED 2465865
  long DDSEED 1123545
  long WISEED 8885564
  long RISEED 566945
  long NUMBERPROBLEMS 10
  long NUMBERALGORITHMS 3
  long RIDIMODE 0
  double TFACTOR 0.7
</CONTROLPARAMETERS>

<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE lower_bounds
</CONTROLPARAMETERS>

<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE dispatch
  string SCHEDULE ACTIVE
  string RULE RANDOM
</CONTROLPARAMETERS>

<CONTROLPARAMETERS>
  string AUTOALG_EXECUTABLE nb_iter
  string METHOD IterativeImprovement
  string NGBH k_API
  long k 1
  long STEPS 1000
  long NUMB_STUCKS 1000
  double ABORT_BOUND -1
  string TYPE RAND
</CONTROLPARAMETERS>
```

The first block, which is enclosed by `<PROBLEMTYPE> ... </PROBLEMTYPE>`, gives the problem type in the $\alpha \mid \beta \mid \gamma$ notation. The following information can be given (see also **Problem Type** $\rightarrow$ 2.2):

Parameter	**Possible entries**
α	`O`, `J` or `F`
β	`r_i` or nothing (empty)
γ	`Cmax`, `Lmax`, `SumCi`, `SumWiCi`, `SumUi`, `SumWiUi`, `SumTi` or `SumWiTi`

Important: The notation in the input file for `auto_alg` depends on capital and small letters. One should also take care that space characters are correctly given (as in the example).

After the declaration of the problem type, several blocks follow in the input file which are enclosed by `<CONTROLPARAMETERS> ... </CONTROLPARAMETERS>`. The first of these blocks has a particular importance since here the parameters for the random number generator and the number of algorithms are fixed. Parameters, which are not needed, can be dropped. The particular parameters have the following meaning:

Parameter	**Meaning**
`M`, `N`	Gives the problem size $n \times m$, where n is the number of jobs and m is the number of machines.
`NUMBERPROBLEMS`	Gives the number of problem instances which `auto_alg` should generate and solve.
`NUMBERALGORITHMS`	Gives the number of algorithms which should be applied to a problem instance.
`MINPT`, `MAXPT`	Minimal and maximal values for an entry in the matrix of the processing times.
`MINDD`, `MAXDD`	Minimal and maximal values for the due dates of the jobs.
`MINRI`, `MAXRI`	Minimal and maximal values for the release dates of the jobs. These parameters are only needed, if `r_i` has been settled in the problem type.
`MINWI`, `MAXWI`	Minimal and maximal values for the weights of the jobs (is required for the objective functions $\sum w_iC_i$, $\sum w_iU_i$ and $\sum w_iT_i$).
`TIMESEED`	Random seed for the matrix of the processing times.
`MACHSEED`	Random seed for the machine orders.
`DDSEED`	Random seed for the due dates of the jobs.
`RISEED`	Random seed for the release times of the jobs.
`WISEED`	Random seed for the weights of the jobs.
`RIDIMODE`	If this parameter is set to `1`, the due dates d_i are determined by the following formula: $$d_i = r_i + \mathrm{TF} \cdot \sum_{j=1}^{m} p_{ij}, \qquad i = 1, 2, \ldots n \qquad (7.1)$$ The release times r_i of the jobs are uniformly distributed in the interval $[0, r_{\max}]$. $$r_{\max} = \frac{1}{2n} \cdot \sum_{i=1}^{n} \sum_{j=1}^{m} p_{ij} \qquad (7.2)$$
`TFACTOR`	Settles the parameter TF (tightness factor) for the case `RIDIMODE` $= 1$.

After the declaration of the parameters for the random generator, further parameter blocks follow which represent the algorithms successively to be applied. The number of blocks must coincide with parameter `NUMBERALGORITHMS`. The first algorithm should always be `lower_bounds` to avoid later problems with the filtering of the output data. This algorithm calculates lower bounds for C_{max} and $\sum C_i$ and as a third result, it determines the expectation value of $\sum C_i$ under the assumption that idle times do not exist.

The exact notation for the invocation of a particular algorithm is described in the Chapter **Algorithms in LiSA** ($\rightarrow$ 4).

`auto_alg` processes the input file as follows:

- Generation of a problem instance by the random number generator according to the given parameters.
- Application of the algorithms to the generated problem instance according to the order, which is given in the input file. Solutions determined by some algorithm can be used by a subsequent algorithm as starting solution, and they can be iteratively improved further.
- Storage of the problem instance with all solutions found by the algorithms.

These steps are repeated as often as given by the parameter `NUMBERPROBLEMS`.

Additional parameters for the algorithms

The line `AUTOALG_START_FROM {NUMBER1,NUMBER2,...}` determines which outputs are used as input for the current algorithm. `NUMBER1` and `NUMBER2` are the numbers of algorithms already applied. `{0}` means no input for this algorithm. The standard setting for an algorithm with number `i` is `{i-1}`.
By the algorithm `best_of` the best solution for an instance among all applied procedures is chosen. In this way, the best solution found by several constructive procedures for any problem instance can be used as starting solution for an iterative procedure. This selection algorithm works as an independent procedure and has to be counted for the number of algorithms applied.

In the following example, the starting solution should be the best solution among those generated by algorithms 2, 5, and 10:

```
<CONTROLPARAMETERS>
string AUTOALG_EXECUTABLE best_of
string AUTOALG_START_from {2,5,10}
</CONTROLPARAMETERS>
```

By the call `AUTOALG_TIMELIMIT`, a time limit can be set for any algorithm. Then the procedure is stopped by `SIGINT` as soon as the time limit has been reached. The measured time is not the pure time of the process but the real time. The standard setting is `0` and does not correspond to a limit. This parameter should only be used for algorithms which generate an output if they are stopped by `SIGINT`.

Call of Autoalg

The program `auto_alg` can be found in the LiSA main directory in the subdirectory `bin`. It must be invoked exactly from this directory since the algorithms to be invoked must be started from here. Of course, the expected input file can be in an arbitrary directory. In the directory with the input file, `auto_alg` will save all problems solved during the processing. They can be opened and inspected with LiSA.
During the processing, `auto_alg` outputs log messages to the standard output. These messages are rather numerous and might be confusing. It is recommended to transfer this information from the standard output into a file. The written information can be better evaluated, if a filter is applied to this file.

A call of `auto_alg` under Linux/Unix/Cygwin can for instance look as follows:

```
cd ~/LiSA/bin
./auto_alg ~/LiSA-Data/algorithms.alg > ~/LiSA-Data/auto_alg.out
```

Under Windows (with a standard installation of the LiSA package), the command line (Start/Execute → “cmd”) can be used. The corresponding command can look as follows:

```
cd C:\Program Files\LiSA\bin
auto_alg.exe C:\LiSA-Data\algorithms.alg > C:\LiSA-Data\auto_alg.out
```

This activates `auto_alg` with the input file `algorithms.alg` and transfers the output into the file `auto_alg.out`.

Evaluation of the results by filters

The filters are also in the subdirectory `bin`. To execute them, Perl must be installed. When activating them, the output file of `auto_alg` previously generated must be transferred. The filters write their output again to the standard output which can also be transferred by the `>` operator into a file.
The following console command under Linux/Unix/Cygwin activates a filter to the output file previously generated. The result of the filtering is written into the file `filter.out`.

```
./filter_objectives ~/LiSA-Data/auto_alg.out > ~/LiSA-Data/filter.out
```

In the Windows command line, the command looks for instance as follows:

```
perl filter_objectives C:\LiSA-Data\auto_alg.out > C:\LiSA-Data\filter.out
```

Remark: For a call of the filters via the Windows command line one must ensure that Perl is correctly listed in the environment variable `PATH`. However, mostly this is done automatically when installing Perl.

All filters list their results in tables. Each row contains an information on the processed problem instance. It is expected that algorithm `lower_bounds` is executed as the first one. Therefore, the results of the first algorithm are not displayed by the filters. An exception is `filter_runtime`, which also gives the running time of the first algorithm.

filter	**meaning**
`filter_runtime`	Lists columnwise the running time for any algorithm in seconds.
`filter_objectives`	Gives lower bounds for the problem in the first three columns. The following columns list the obtained objective function value for any executed algorithm.
`filter_lmax`	Gives lower bounds for the problem in the first three columns. The following columns list the obtained maximum tardiness of the jobs for any executed algorithm.
`filter_abort`	Gives for any algorithm the execution status in per cent, if given. This is usually 99 % or 100 % if the algorithm has been normally run. However, some algorithms have stopping criteria (a bound for the objective function value has been reached, or too many iterations have been performed without getting an improvement) so that they can additionally terminate. If the algorithm is very fast, here also "n/a" can be written.
`filter_abortStucks`	Gives for iterative algorithms, whether they have been prematurely terminated. This may happen if the objective function value has not been improved over a long time. A "1" means that the algorithm has been prematurely terminated, a "0" means that the given number of iterations has been completely performed. If the algorithm is very fast, here also "n/a" can be written.
`filter_sumti`	Gives for any algorithm the obtained value for $\sum T_i$ (also if this was not the objective function). The first three columns list different lower bounds.

Once again: ATTENTION!

The work with the automated call of algorithms must be done very carefully since the input file is not internally checked for correctness. In particular, the user hast to take care whether the invoked algorithm is available for the problem type considered and whether the parameters for the generation of the problem instance and the algorithms are correct (take care about capital and small letters as well as white spaces). Another error source is the number of algorithms to be executed (please count correctly!)

Chapter 8

Examples

In this chapter it is described how one can invoke an algorithm via the command line and how one can use the graphical interface of LiSA. In the directory **LiSA/data/sample**, some files with examples are contained. These are input files for LiSA and input files for the automated call of algorithms.

8.1 Call of an Algorithm from the Command Line

In this example, we solve a given problem instance without the use of the graphical user interface of LiSA. This is done by coding the problem instance in XML and by the subsequent call of a solution algorithm directly from the command line. This way can be used to automate the solution and optimization of given shop scheduling problems.

Given is a problem instance of the type $J \mid\mid \sum C_i$ with 4 machines and 3 jobs. In addition, the processing times and job orders are given as follows:

$$PT = \begin{bmatrix} 53 & 98 & 0 & 80 \\ 88 & 38 & 89 & 96 \\ 37 & 30 & 0 & 60 \end{bmatrix} \quad \begin{array}{lllllllll} A_1 & : & M_2 & \rightarrow & M_4 & \rightarrow & M_1 & & \\ A_2 & : & M_2 & \rightarrow & M_1 & \rightarrow & M_3 & \rightarrow & M_4 \\ A_3 & : & M_4 & \rightarrow & M_2 & \rightarrow & M_1 & & \end{array}$$

For this problem instance, a schedule should be determined by means of the algorithm ***Beam Search*** ($\rightarrow$ 4.2.3).

Completion of missing information

For the transfer to a solution algorithm, the set of operations and the machine orders must be given in matrix form. From the matrix of the processing times PT, the set of operations can be determined:

$$SIJ = I \times J \backslash \{(13), (33)\} \Longrightarrow \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

Matrix MO is obtained as follows:

$$MO = \begin{bmatrix} 3 & 1 & . & 2 \\ 2 & 1 & 3 & 4 \\ 3 & 2 & . & 1 \end{bmatrix}$$

Generation of the instance document

Now the problem instance has to be transformed into an XML document which can be transferred to the solution algorithm. The information on the parameters of the call for the solution algorithm is important which are also contained in the document – combined in a `<controls>` element. The particular parameters consist of a name, the type (`string`, `integer` or `real`) and a value. In Chapter 4, one can find which parameters are required for a particular algorithm.

Hint: For the correct input of the XML file, the order of the XML elements is important. For instance, the `<controls>` element must follow *directly after* the `<values>` element. The general structure of an `instance` document is described in Appendix A.

For the data given above, the following `instance` document created:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE instance PUBLIC "" "LiSA.dtd">
<instance xmlns:LiSA="http://lisa.math.uni-magdeburg.de">
  <problem>
    <alpha env="J"/>
    <beta/>
    <gamma objective="Sum_Ci"/>
  </problem>
  <values m="4" n="3">
    <processing_times model="lisa_native">
      {
        {  53  98   0  80 }
        {  88  38  89  96 }
        {  37  30   0  60 }
      }
    </processing_times>
    <operation_set model="lisa_native">
      {
        {   1   1   0   1 }
        {   1   1   1   1 }
        {   1   1   0   1 }
      }
    </operation_set>
    <machine_order model="lisa_native">
      {
        {   3   1   0   2 }
        {   2   1   3   4 }
        {   3   2   0   1 }
      }
    </machine_order>
  </values>
  <controls>
    <parameter type="string" name="MODE" value="INSERT" />
    <parameter type="string" name="INS_ORDER" value="LPT" />
    <parameter type="string" name="INS_METHOD" value="INSERT1" />
    <parameter type="string" name="CRITERION" value="OBJECTIVE" />
    <parameter type="integer" name="k_BRANCHES" value="5" />
  </controls>
</instance>
```

Call of an algorithm under Windows

It is assumed that the XML file created is contained under the name `example.xml` in the directory `C:\Scheduling\`. The algorithm to be called (in this case `beam.exe`) is contained in the subdirectory `bin` of the LiSA directory (standard `C:\Program Files\LiSA\bin`). By means of the Windows command line (`cmd`), one has to change into this directory in order to invoke the algorithm from here (the call from another working directory is not possible).

```
cd C:\Program Files\LiSA\bin
beam.exe C:\Schedulng\example.xml C:\Scheduling\example.out.xml
```

This invocation generates in the directory `C:\Scheduling\` the file `example.out.xml` which is a document of the type `solution`. It contains the output of the algorithm and the determined schedule.

Call of the algorithm under UNIX and cygwin

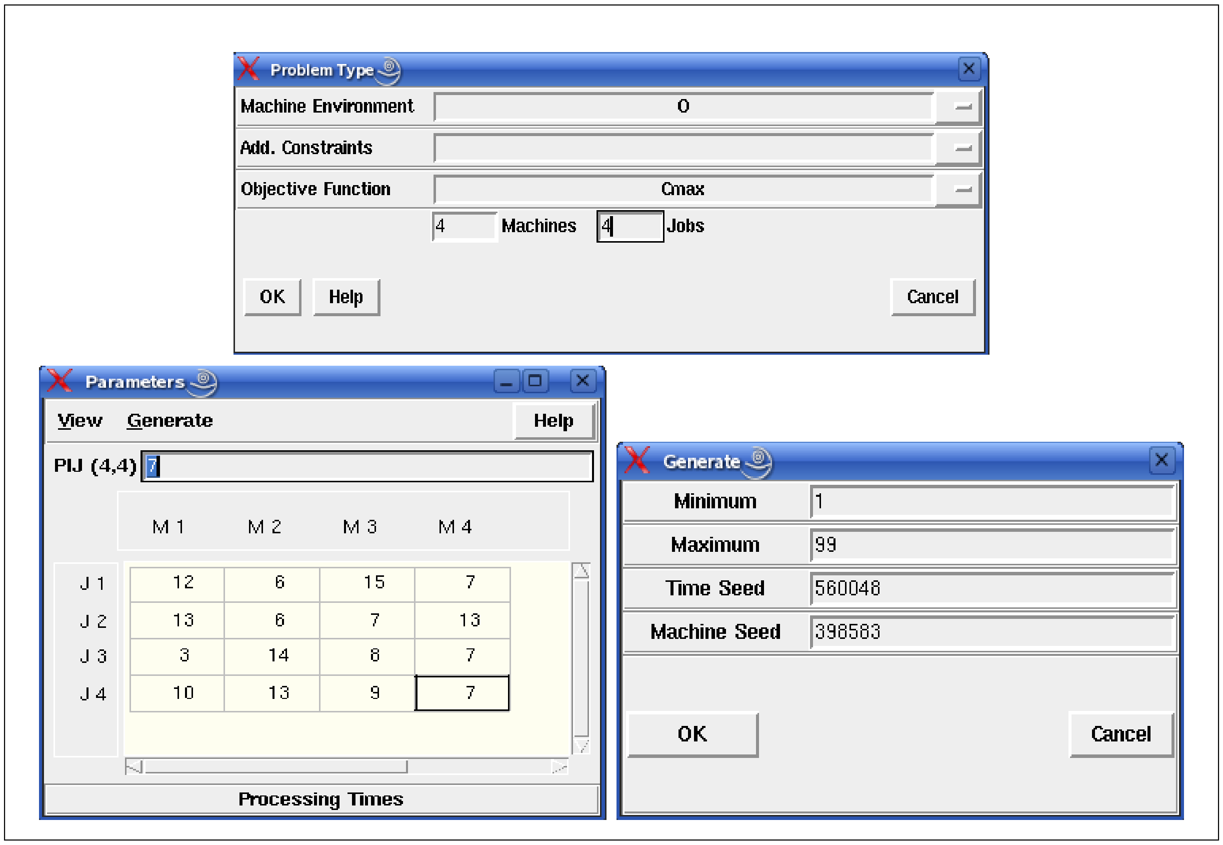
Here it is assumed that the created XML file has been stored in the home directory under the name `example.xml`. The algorithm to be called (in this case `beam`) is contained in the subdirectory `bin` of the LiSA directory (standard `~/LiSA/bin`). By means of a console, the program can be activated in this working directory (a call from another directory is not possible.).

```
cd ~/LiSA/bin
./beam ~/example.xml ~/example.out.xml
```

This invocation generates in the home directory the file `example.out.xml` which is a document of the type `solution`. It contains the output of the algorithm and the determined schedule.

8.2 Example for the Use of LiSA

The user considers an open-shop problem with $m = 4$ machines and $n = 4$ jobs and makespan minimization. No further restrictions are given. How can one use LiSA in this case? After starting LiSA, one clicks under ***File*** the button ***New***. The window with the problem type opens and the problem is entered in the $\alpha \mid \beta \mid \gamma$ description, i.e., one chooses the machine environment O and the objective function Cmax.



Machine Environment: O
Add. Constraints:
Objective Function: Cmax
4 Machines 4 Jobs

PIJ (4,4) 7

	M 1	M 2	M 3	M 4
J 1	12	6	15	7
J 2	13	6	7	13
J 3	3	14	8	7
J 4	10	13	9	7

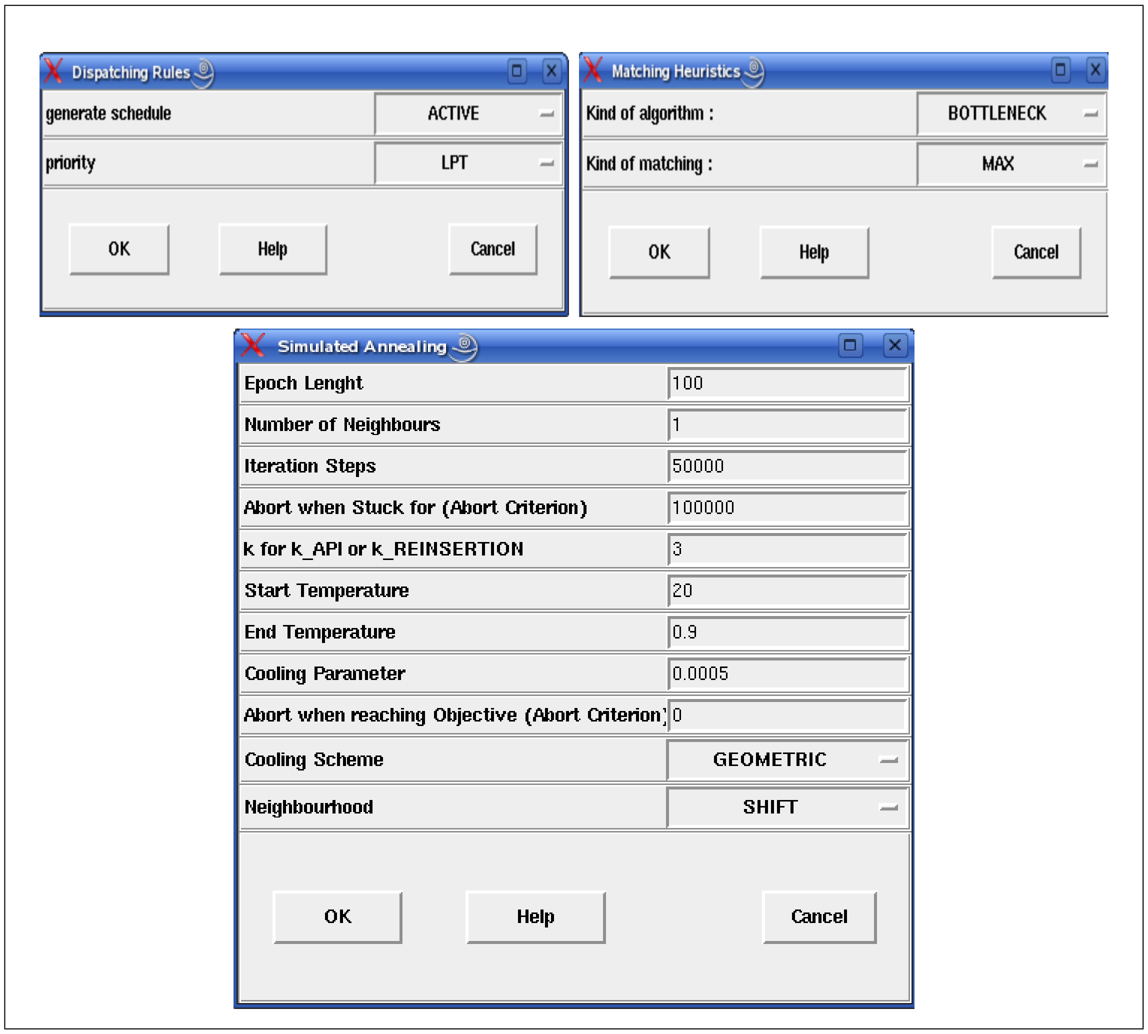
Minimum	1
Maximum	99
Time Seed	560048
Machine Seed	398583

Figure 8.1: Input of a problem instance

Moreover, both the number of machines and the number of jobs are set to 4. Now LiSA makes all modules available which are contained for this problem type. One starts with the input of the processing times (Button ***Edit***, ***Parameter***, ***Generate***, ***... Processing Times***). In addition to the manual input, it is possible to use a random number generator. It generates the processing times uniformly distributed from the interval [Minimum, Maximum]. Time seed and machine seed are arbitrary parameters for starting the generator which determines for the same selection the same numbers.
Figure 8.1 displays the corresponding LiSA windows. The matrix of the processing times PT is given by

$$PT = \begin{bmatrix} 12 & 6 & 15 & 7 \\ 13 & 6 & 7 & 13 \\ 3 & 14 & 8 & 7 \\ 10 & 13 & 9 & 7 \end{bmatrix}$$

In addition, one can also read the data from an XML file the format of which is described in Chapter 2.5. Now LiSA releases the available algorithms. Under ***Algorithms***, there are available both exact and heuristic algorithms.



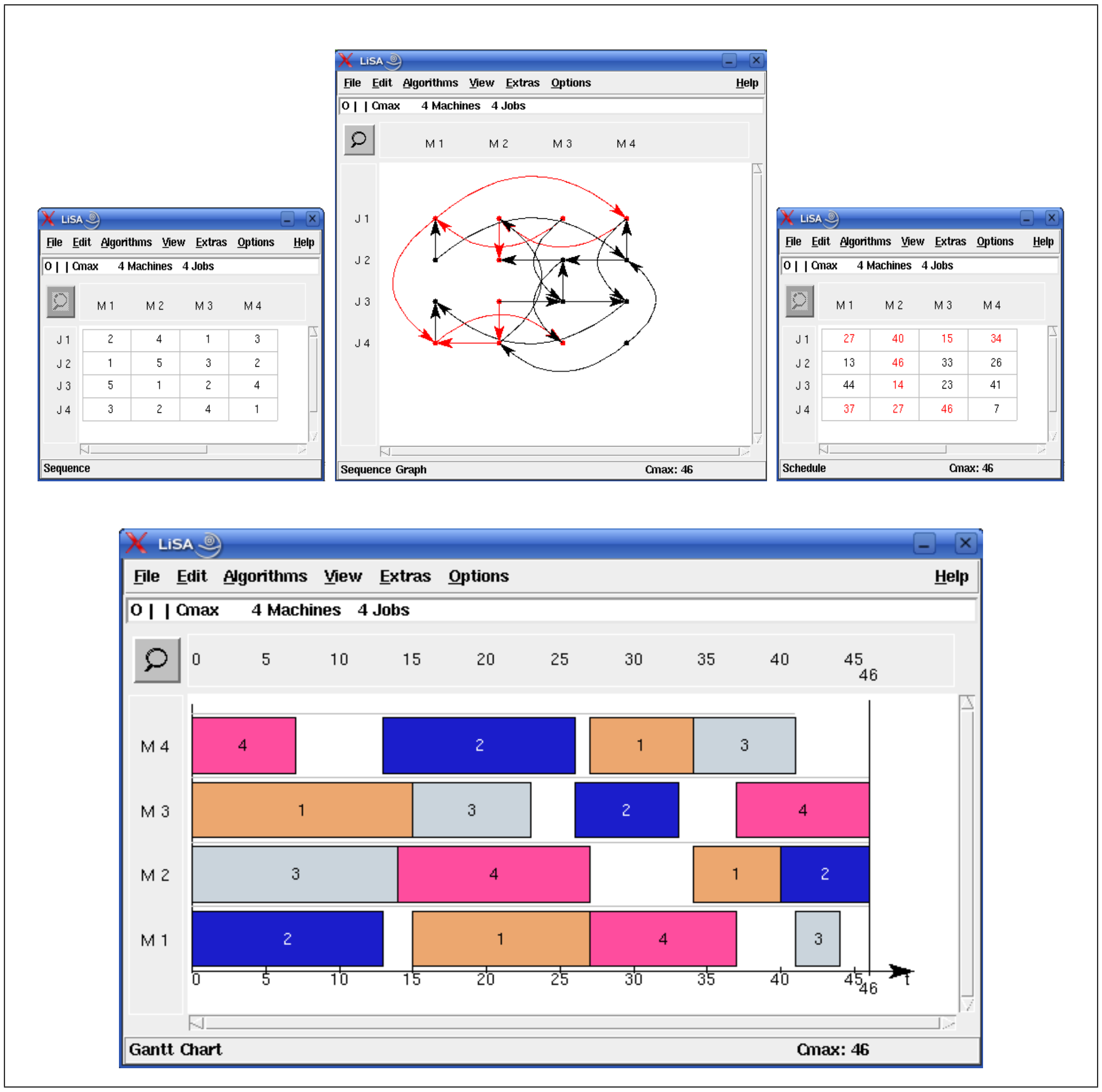
generate schedule	ACTIVE
priority	LPT

Kind of algorithm :	BOTTLENECK
Kind of matching :	MAX

Epoch Lenght	100
Number of Neighbours	1
Iteration Steps	50000
Abort when Stuck for (Abort Criterion)	100000
k for k_API or k_REINSERTION	3
Start Temperature	20
End Temperature	0.9
Cooling Parameter	0.0005
Abort when reaching Objective (Abort Criterion)	0
Cooling Scheme	GEOMETRIC
Neighbourhood	SHIFT

Figure 8.2: Heuristic Algorithms

Figure 8.2 displays some of the available heuristics. Fast dispatching rules like the LPT rule (longest processing time first) generate a first active schedule which can be used for starting several iterative search procedures. Here the window for simulated annealing is displayed. Several parameters such as the neighborhood or the cooling scheme can be chosen. Details for the parameters can be taken from the description of the search procedure. Another constructive procedure is the use of matching algorithms, where step by step operations are appended to the sequence which can be simultaneously processed. Here, the minimization of the maximal processing time of the operations simultaneously processed is used.
When applying an algorithm, LiSA immediately displays the Gantt chart of the generated schedule on the screen. All forms of the output are given in Figure 8.3. So the sequence and the matrix of the completion times of all operations as well as the visualizations in the sequence graph and in the Gantt chart can be selected under ***View***. Gantt charts can be displayed machine and job oriented, the critical path can be highlighted (Button ***Options***). If the number of machines or jobs is too large, i.e., the Gantt chart is too complex, the zoom



Sequence

	M 1	M 2	M 3	M 4
J 1	2	4	1	3
J 2	1	5	3	2
J 3	5	1	2	4
J 4	3	2	4	1

Schedule Cmax: 46

	M 1	M 2	M 3	M 4
J 1	27	40	15	34
J 2	13	46	33	26
J 3	44	14	23	41
J 4	37	27	46	7

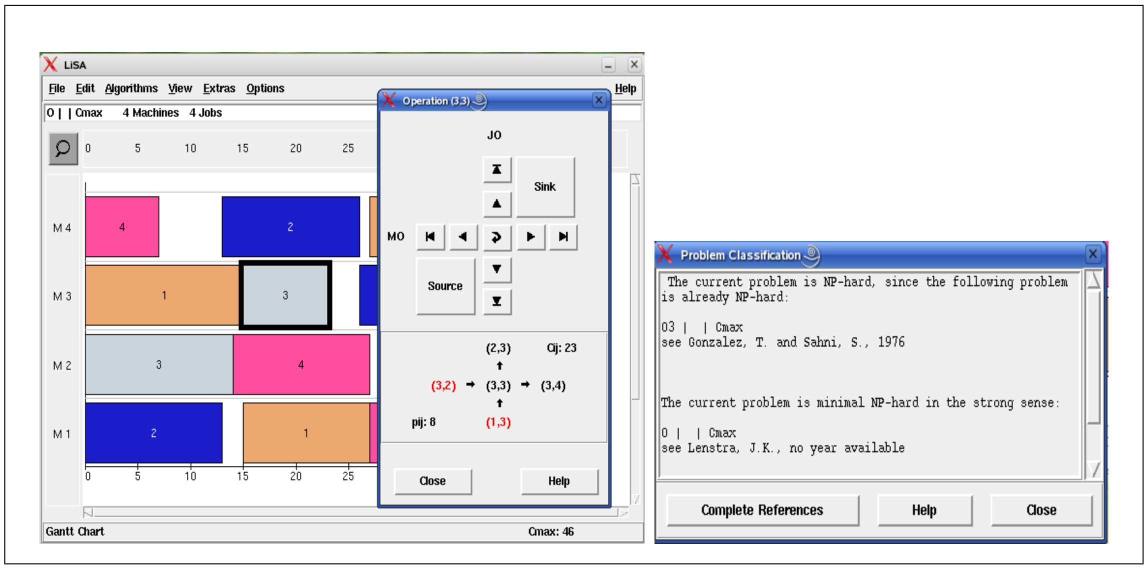
Figure 8.3: Output of the results

function may help. By zooming repeatedly, the Gantt chart can be enlarged such that all information can be viewed when scrolling.

LiSA has some extras, two of them are given in Figure 8.4.

The Gantt charts can be manipulated, i.e., an operation which is chosen by clicking the right mouse button can be shifted both in the machine order (MO) and the job order (JO) one position to the left or right. If this shift does not lead to an acyclic sequence graph, an error massage is given. In addition, it is possible that an operation is chosen as source or sink in the sequence graph. In this case, an acyclic sequence graph is always generated.

As a second extra, LiSA contains a complexity module. As soon as a problem has been entered in the $\alpha \mid \beta \mid \gamma$ notation, the complexity of the problem can be retrieved under ***Extras***, ***Problem classification***. Here LiSA uses the database from Osnabrück on the complexity of deterministic scheduling problems. Moreover, it makes the complete reference available.



```
The current problem is NP-hard, since the following problem
is already NP-hard:

O3 |  | Cmax
see Gonzalez, T. and Sahni, S., 1976


The current problem is minimal NP-hard in the strong sense:

O |  | Cmax
see Lenstra, J.K., no year available
```

Figure 8.4: Manipulation and Complexity

Bibliography

[1] Adams,J., Balas, E., Zawack, D. [1988] *The Shifting Bottleneck Procedure for Job Shop Scheduling*; Management Science 34, 391-401

[2] Baker, K.R. [1984]: *Introduction to Sequencing and Scheduling*; Wiley & Sons, New York

[3] Blazewicz, J., Ecker, K., Pesch, E., Schmidt, G., Weglarz, J. [1996]: *Scheduling Computer and Manufacturing Processes*; Springer Verlag Berlin-Heidelberg-New York

[4] Blum, Ch. [2003]: *Beam-ACO-hybridizing ant colony optimization with beam search: an application to open shop scheduling*; computers & operations research III, Online verfügbar unter www.sciencedirect.com

[5] Bräsel, H.[1990]: *Latin Rectangle in Scheduling Theory*; Professorial Dissertation (in German), University Magdeburg, Germany

[6] Bräsel, H., Hennes, H.[2004]: *On the open-shop problem with preemption and minimizing the average completion time*; European Journal of Operational Research 157, 607-619

[7] Bräsel, H., Tautenhahn,T., Werner,F.[1993]: *Constructive Heuristic Algorithms for the Open Shop Problem*; Computing, 51, 95-110

[8] Brucker, P. [2001]: *Scheduling Algorithms*; Third Edition, Springer Verlag Berlin-Heidelberg-New York

[9] Chretienne, P., Coffman, E.G., Lenstra, J.K. (Editors) [1995]: *Scheduling Theory and its Applications*; John Wiley & Sons, Chichester-New York-Brisbane

[10] Conway, R.W., Maxwell, W.L., Miller, L.W. [1967]: *Theory of Scheduling*; Addison-Wesley Publishing Company, Massachusetts

[11] Dannenbring, D.G. [1977] *An Evaluation of Flowshop Sequencing Heuristics*; Managemant Science 23, 1174-1182

[12] Domschke, W., Scholl, A., Voß, S. [1993]: *Produktionsplanung - Ablauforganisatorische Aspekte*; Springer Verlag Berlin-Heidelberg-New York

[13] French, S. [1982]: *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*; John Wiley & Sons, New York

[14] Gonzales, T., Sahni, S.[1976]: *Open-shop to minimize finish time*; Journal of the Association for Computing Machinery 23 (4), 665-679

[15] Graham, R.E., Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G. [1979]: *Optimization and approximation in deterministic sequencing and scheduling: a survey*; Ann. Discrete Math. 4, 287-326

[16] Graves, S.C., Rinnooy Kan, A.H.G., Zipkin, P.H. (Editors) [1993]: *Handbooks in Operations Research and Management Science (Volume 4): Logistics of Production and Inventory*; Elsevier Science Publishers B.V., North-Holland, Amsterdam-London-NewYork-Tokyo

[17] Lageweg, B.J., Lawlwer, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G. [1981]: *Computer-aided Complexity Classification of Deterministic Scheduling Problems*; Report BM 138, Centre for Mathematics and Computer Science, Amsterdam

[18] Lenstra, J.K. [1977]: *Sequencing by enumerative methods*; Centre Tracts 69, Amsterdam

[19] Morton, T.E., Pentico, D.W. [1993]: *Heuristic Scheduling Systems*; John Wiley & Sons, Inc., New York

[20] Muth, J.F., Thompson, G.L. (Editors) [1963]: *Industrial Scheduling*; Prentice Hall, Englewood Cliffs

[21] Pinedo, M.: Scheduling [1995]: *Theory, Algorithms and Systems*; Prentice Hall, Inc., Englewood Cliffs, New Jersey

[22] Rinnooy Kan, A.H.G. [1976]: *Machine scheduling problems: Classification, Complexity and Computations*; Martinus Nijhoff/ The Hague,

[23] Tanaev, V.S., Sotskov, Y.N., Strusevich, V.A. [1994]: *Scheduling Theory: Multi-Stage Systems*; Kluwer Academic Publishers, Dordrecht

[24] Taillard, E. [1993]: *Benchmarks for basic scheduling problems*; European Journal of Operational Research 64, 278-285

[25] Tanaev, V.S., Sotskov, Y.N., Strusevich, V.A. [1994]: *Scheduling Theory: Multi-Stage Systems*; Kluwer Academic Publishers, Dordrecht

[26] Willenius, P. [2000] *Irreduzibilitẗstheorie bei Shop-Schedulingproblemen*; Shaker Verlag, Aachen, Dissertation

Appendix A

XML-Reference

This chapter gives a detailed overview about the structure of all XML files that are used in LiSA. It rounds up the section **The File Format in LiSA** (→ 2.5), where the five document types have been introduced.

A.1 General

A.1.1 The XML Language

XML (*extensible markup language*) is a markup language, similar to HTML, for storing information in text form. Like in HTML, XML documents consist of elements, that can be nested, such that a tree-like structure is created. These elements may contain text, attributes or other elements. Here a summary about all XML-related notions is given, that are used in this documentation. However, these explanations are kept very simple and are just provided to give a basic understanding of the examples used here. A more detailed description of the language can be viewed on the web site of the World Wide Web Consortium (www.w3.org/xml)

- A *tag* is a string enclosed in sharp brackets (`< >`). If this string begins with a slash (`/`), it is called an *end tag*, otherwise it is a *start tag*. The first word in the string is the name of the tag. After every start tag, there must follow an end tag with the same name.
  Example: `<controls>  </controls>`
- Start tags may contain *attributes*. These are denoted as `[attribute]="[value]"`, with a space character between the tag name and the attribute.
  Example: `<schedule m="5" n="3" semiactive="yes">`
- A pair of start and end tags, as well as the enclosed text, is called an *element*. This text can contain elements itself, which are called *child elements*.
  Example: `<due_dates model="lisa_native"> { 5 8 10 4 2 } </due_dates>`
- If an element does not contain child elements or an enclosed text, the shorter notation `<[tagname] [attribute1]="[value1]" ... />` can be used. The end tag, in this case, is omitted.

A.1.2 Representation of Data

Primitive data types, as integer or real numbers, as well as strings are simply stored in attributes. An example is the `<schedule>` tag:

```
<schedule m="10" n="20" semiactive="yes">
```

Here the problem size is fixed by the integers $m = 10$ and $n = 20$. Moreover, the string `"yes"`, as the value of the attribute `semiactive`, indicates that the following schedule is semi-active. Attention has to be paid to real numbers, where a decimal point must be used instead of a comma.

In addition, vectors and matrices are used as complex data types. A vector of dimension n with the components `a_1` to `a_n` is denoted as follows:

```
{ a_1  a_2  a_3  ...  a_n }
```

As a separator between the values, one or more whitespace characters may be used. An $n \times m$ matrix can be denoted as an n-dimensional vector that itself contains m-dimensional vectors:

```
{
  { a_11  a_12  a_13  ...  a_1m }
  { a_21  a_22  a_23  ...  a_2m }
  ...
  { a_n1  a_n2  a_n3  ...  a_nm }
}
```

A.2 The Document Types `problem`, `instance` and `solution`

These three document types are generally used for the communication of scheduling problems in LiSA. They are built consecutively upon each other. Thus, a `problem` document only stores a problem type in the $\alpha \mid \beta \mid \gamma$ notation; an `instance` document additionally stores an individual problem instance. A `solution` document, besides a problem instance, also stores one or more solutions.

A.2.1 Structure

Representative for the three document types, here only the structure of a `solution` document is shown. In an `instance` document, the `<schedule>` elements are omitted, whereas `problem` documents only consist of the `<problem>` element.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE solution PUBLIC "" "LiSA.dtd">
<solution xmlns:LiSA="http://lisa.math.uni-magdeburg.de">
  <problem>
    <alpha ... />
    <beta ... />
    <gamma ... />
  </problem>
  <values m="..." n="...">
    <processing_times model="lisa_native">
      ...
    </processing_times>
    <operation_set model="lisa_native">
      ...
    </operation_set>
    <due_dates model="lisa_native">
      ...
    </due_dates>
    <release_times model="lisa_native">
      ...
    </release_times>
    <weights model="lisa_native">
      ...
    </weights>
    <weights_2 model="lisa_native">
      ...
    </weights_2>
    <extra model="lisa_native">
      ...
    </extra>
  </values>
  <controls>
    <parameter type="..." name="..." value="..." />
    ...
  </controls>
  <schedule m="..." n="..." semiactive="...">
    <plan model="lisa_native">
      ...
    </plan>
    <machine_sequences model="lisa_native">
      ...
    </machine_sequences>
    <job_sequences model="lisa_native">
      ...
    </job_sequences>
    <completion_times model="lisa_native">
      ...
    </completion_times>
  </schedule>
  ...
</solution>
```

A.2.2 Description

`<solution>`

This element contains one or more solutions for an instance of a scheduling problem.
Parent element: none (root element of the document type `solution`)
Child elements: `<problem>`, `<values>`, `<schedule>`, `<controls>` (*optional*)

This element does not contain attributes.

`<instance>`

This element contains an instance of a scheduling problem.
Parent element: none (root element of the document type `instance`)
Child elements: `<problem>`, `<values>`, `<controls>` (*optional*)

This element does not contain attributes.

`<problem>`

This element represents a problem type. It contains one `<alpha>`, `<beta>` and `<gamma>` tag each, that constitute a problem type in the common $\alpha \mid \beta \mid \gamma$ notation.
Parent element:

- none in the document type `problem` (root element)
- `<instance>` in the document type `instance`
- `<solution>` in the document type `solution`

Child elements: `<alpha>`, `<beta>`, `<gamma>`

This element does not contain attributes.

`<alpha>`

α describes the machine environment.
Parent element: `<problem>`
Child elements: none

Attribute	**Meaning**
`env`	The machine environment of the problem. Possible values are `1`, `O`, `F`, `J`, `G`, `P`, `Q`, `R`
`m`	(*optional*) Fixes the number of machines. If this parameter is omitted, the number of machines is variable and is part of the problem instance.

`<beta>`

β describes job characteristics and additional constraints.
Parent element: `<problem>`
Child elements: none

Attribute	Meaning
`pmtn`	(*optional*) Fixes if preemption is allowed. Possible values are `yes` and `no`.
`prec`	(*optional*) Fixes if precedence constraints between jobs are given. Possible values are `yes`, `no`, `intree`, `outtree`, `tree`, `sp_graph`, `chains`.
`release_times`	(*optional*) Fixes if release dates for the jobs are given. Possible values are `yes`, `no`.
`due_dates`	(*optional*) Fixes if due dates for the jobs are given. Possible values are `yes`, `no`.
`processing_times`	(*optional*) Gives an information about the processing times of the jobs. Possible values are `arbitrary`, `constant`, `uniform`.
`no-wait`	(*optional*) Fixes if waiting times between the operations of one job are forbidden. Possible values are `yes` and `no`.

`<gamma>`

γ describes the objective function.
Parent element: `<problem>`
Child elements: none

Attribute	Meaning
`objective`	The objective function of the problem. Possible values are `Cmax`, `Lmax`, `Sum_Ci`, `Sum_wiCi`, `Sum_Ui`, `Sum_wiUi`, `Sum_Ti`, `Sum_wiTi`, `irreg_1`, `irreg_2`

`<values>`

This element summarizes all information that belongs to a problem instance.
Parent element:

- `<instance>` in the document type `instance`
- `<solution>` in the document type `solution`

Child elements: `<processing_times>`, `<operation_set>`, `<machine_order>` (*optional*), `<release_times>` (*optional*), `<due_dates>` (*optional*), `<weights>` (*optional*), `<weights_2>` (*optional*), `<extra>` (*optional*)

Attribute	**Meaning**
`m`	The number of machines in the problem.
`n`	The number of jobs in the problem.

`<processing_times>`

This element contains the matrix P of the processing times.
Parent element: `<values>`
Child elements: A matrix of the format `n` × `m` with the processing times.

Attribute	**Meaning**
`model`	Reserved, must be `lisa_native`.

`<operation_set>`

This element contains the set of operations as a binary matrix. A one on position (i, j) means that the operation (ij) has to be executed, i.e., job A_i has to be processed on machine M_j. On the other hand, a zero on position (i, j) means that operation (ij) is not executed. p_{ij}, in this case, must also be zero.
Parent element: `<values>`
Child elements: A binary matrix of the format `n` × `m` that represents the set of operations.

Attribute	**Meaning**
`model`	Reserved, must be `lisa_native`.

`<machine_order>`

This element contains the matrix MO that defines the machine order.
Parent element: `<values>`
Child elements: A matrix of the format $\mathtt{n} \times \mathtt{m}$ that represents the machine order.

Attribute	Meaning
`model`	Reserved, must be `lisa_native`.

`<release_times>`

This element contains the release dates of the jobs as an `n`-dimensional vector. The i-th component of this vector contains the release date of job A_i.
Parent element: `<values>`
Child elements: A vector with `n` components.

Attribute	Meaning
`model`	Reserved, must be `lisa_native`.

`<due_dates>`

This element contains the due dates for the jobs as an `n`-dimensional vector. The i-th component of this vector contains the due date of job A_i.
Parent element: `<values>`
Child elements: A vector with `n` components.

Attribute	Meaning
`model`	Reserved, must be `lisa_native`.

`<weights>`

This element contains the weights w_i of the jobs A_i as an `n`-dimensional vector.
Parent element: `<values>`
Child elements: A vector with `n` components.

Attribute	Meaning
`model`	Reserved, must be `lisa_native`.

`<weights_2>`

Parent element: `<values>`
Child elements: A vector with `n` components.

Attribute	**Meaning**
`model`	Reserved, must be `lisa_native`.

`<extra>`

Parent element: `<values>`
Child elements:

Attribute	**Meaning**
`model`	Reserved, must be `lisa_native`.

`<controls>`

This element contains all parameters for an algorithm, for which this document will serve as input file.
Parent element:

- `<instance>` in the document type `instance`
- `<solution>` in the document type `solution`

Child elements: `<parameter>`

This element contains no attributes.

`<parameter>`

This element represents one parameter for an algorithm.
Parent element: `<controls>`
Child elements: none

Attribute	**Meaning**
`type`	The data type of the parameter. Possible values are `string`, `integer`, `real`
`name`	The name of the parameter. For a list of all parameters that have to be specified for a particular algorithm, see Chapter 4.
`value`	The value of the parameter.

`<schedule>`

This element represents one solution of a problem instance. Besides the sequence, it may contain the matrix of the completion times as well as the machine and job orders.
Parent element: `<solution>`
Child elements: `<plan>`, `<machine_sequences>` (*optional*), `<job_sequences>` (*optional*), `<completion_times>` (*optional*)

Attribute	**Meaning**
`m`	The number of machines for this problem.
`n`	The number of jobs for this problem.
`semiactive`	(*optional*) Fixes if a schedule is semi-active.

`<plan>`

This element contains the matrix LR that contains the sequence of the solution.
Parent element: `<schedule>`
Child elements: A latin rectangle of the format `n` $\times$ `m` that represents the sequence of the solution.

Attribute	**Meaning**
`model`	Reserved, must be `lisa_native`.

`<machine_sequences>`

This element contains the matrix MO that represents the machine order of the solution.
Parent element: `<schedule>`
Child elements: A matrix of the format `n` × `m` representing the machine order.

Attribute	Meaning
`model`	Reserved, must be `lisa_native`.

`<job_sequences>`

This element contains the matrix JO that represents the job order of the solution.
Parent element: `<schedule>`
Child elements: A matrix of format `n` × `m` representing the job order.

Attribute	Meaning
`model`	Reserved, must be `lisa_native`.

`<completion_times>`

This element contains the matrix C of the completion times.
Parent element: `<schedule>`
Child elements: A matrix of the format `n` × `m` with the completion times of all operations.

Attribute	Meaning
`model`	Reserved, must be `lisa_native`.

A.3 The Document Type `algorithm`

This document type is used to include algorithms into LiSA. Further information about their use can be found in the section **Algorithm Modules** (→ 7.1).

A.3.1 Structure

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE algorithm PUBLIC "" "LiSA.dtd">
<algorithm
  xmlns:LiSA="http://lisa.math.uni-magdeburg.de"
  name="..."
  type="..."
  call="..."
  code="external"
  help_file="algorithm/...">

  <exact>
    <problem>
        <alpha ... />
        <beta ... />
        <gamma ... />
    </problem>
    ...
  </exact>
  <heuristic>
    <problem>
        <alpha ... />
        <beta ... />
        <gamma ... />
    </problem>
    ...
  </heuristic>
  <alg_controls>
    <integer name="..." caption="..." default="..." />
    ...
    <real name="..." caption="..." default="..." />
    ...
    <choice>
        <item name="..." caption="..." />
        ...
    </choice>
    ...
    <fixed name="..." value="..." />
    ...
  </alg_controls>
</algorithm>
```

Examples of `algorithm` documents can be found in the subdirectory `src/algorithm/sample`.

A.3.2 Description

`<algorithm>`

This element surrounds the whole algorithm description.
Parent element: none (root element of the `algorithm` document).
Child elements: `<heuristic>`, `<exact>`, `<alg_controls>` (*optional*).

Attribute	Meaning
`name`	Name of the algorithm which is shown in the menu **Algorithms**.
`type`	Either `constructive` or `iterative`. Iterative Algorithms require an initial solution to build a new one upon. Constructive algorithms ever build completely new solutions and do not require this.
`code`	Reserved, must be `external`
`call`	Name of the program that implements the algorithm, *without* file extension (`.exe`, for example). This program should have the same name as the directory where the source code is located.
`help_file`	Name of the HTML help file (*with* file extension, like `.html`).

`<exact>` and `<heuristic>`

`<exact>` and `<heuristic>` behave in a similar way. They contain problem descriptions that an algorithm can solve. Problems under `<exact>` are guaranteed to be solved optimally, whereas problems under `<heuristic>` can only be solved heuristically.
Parent element: `<algorithm>`
Child elements: `<problem>`

These elements do not contain attributes.

`<problem>`

This element represents a problem type that can be solved. It is either placed in the `<exact>` or `<heuristic>` elements and contains each `<alpha>`, `<beta>` and `<gamma>` elements that specify the problem type in the common $\alpha \mid \beta \mid \gamma$ notation.
Parent element: `<heuristic>` or `<exact>`
Child elements: `<alpha>`, `<beta>`, `<gamma>`

This element does not contain attributes.

`<alpha>`

α describes the machine environment.
Parent element: `<problem>`
Child elements: none

Attribute	**Meaning**
`env`	The machine environment of the problem. Possible values are `1`, `O`, `F`, `J`, `G`, `P`, `Q`, `R`
`m`	(*optional*) Fixes the number of machines. If this parameter is omitted, the number of machines is variable and is part of the problem instance.

`<beta>`

β describes job characteristics and additional constraints.
Parent element: `<problem>`
Child elements: none

Attribute	**Meaning**
`pmtn`	(*optional*) Fixes if preemption is allowed. Possible values are `yes` and `no`.
`prec`	(*optional*) Fixes if precedence constraints between jobs are given. Possible values are `yes`, `no`, `intree`, `outtree`, `tree`, `sp_graph`, `chains`.
`release_times`	(*optional*) Fixes if release dates for the jobs are given. Possible values are `yes`, `no`.
`due_dates`	(*optional*) Fixes if due dates for the jobs are given. Possible values are `yes`, `no`.
`processing_times`	(*optional*) Gives an information about the processing times of the jobs. Possible values are `arbitrary`, `constant`, `uniform`.
`no-wait`	(*optional*) Fixes if waiting times between the operations of one job are forbidden. Possible values are `yes` and `no`.

`<gamma>`

γ describes the objective function.
Parent element: `<problem>`
Child elements: none

Attribute	Meaning
`objective`	The objective function of the problem. Possible values are `Cmax`, `Lmax`, `Sum_Ci`, `Sum_wiCi`, `Sum_Ui`, `Sum_wiUi`, `Sum_Ti`, `Sum_wiTi`, `irreg_1`, `irreg_2`

`<alg_controls>`

With the element `<alg_controls>` one can declare parameters that can be passed to the algorithm.
Parent element: `<problem>`
Child elements: `<integer>` (*optional*), `<real>` (*optional*), `<choice>` (*optional*), `<fixed>` (*optional*)

This element contains no attributes.

`<integer>`

This declares an integer parameter that can be passed to the algorithm.
Parent element: `<alg_controls>`
Child elements: none

Attribute	Meaning
`name`	Name of the parameter. With this name, the parameter is identified internally.
`caption`	The name of the parameter in a human readable form. This name is shown in the parameter window when the algorithm is called via the menu.
`default`	The standard value for this parameter.

`<real>`

Declares a real-valued parameter that can be passed to the algorithm.
Parent element: `<alg_controls>`
Child elements: none

Attribute	Meaning
`name`	Name of the parameter. With this name, the parameter is identified internally.
`caption`	The name of the parameter in a human readable form. This name is shown in the parameter window when the algorithm is called via the menu.
`default`	The standard value for this parameter.

`<choice>`

`<choice>` parameters offer a selection of a set of predefined values.
Parent element: `<alg_controls>`
Child elements: `<item>`

Attribute	Meaning
`name`	Name of the parameter. With this name, the parameter is identified internally.
`caption`	The name of the parameter in a human readable form. This name is shown in the parameter window when the algorithm is called via the menu.

`<item>`

Items define the values that can be selected with a `<choice>` parameter.
Parent element: `<choice>`
Child elements: none

Attribute	Meaning
`name`	The name of the value.

`<fixed>`

`<fixed>` parameters pass predefined constant values. They are not shown in the LiSA GUI and thus cannot be edited. They can be used to pass some hidden piece of data.
Parent element: `<alg_controls>`
Child elements: none

Attribute	Meaning
`name`	The ID of the parameter.
`value`	The value of the parameter. This cannot be edited by the user.

A.4 The Document Type `controls`

A.4.1 Structure

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE controls PUBLIC "" "LiSA.dtd">
<controls xmlns:LiSA="http://lisa.math.uni-magdeburg.de">
  <parameter type="integer" name="WIDTH" value="600"/>
  <parameter type="integer" name="HEIGHT" value="500"/>
  <parameter type="string" name="LANGUAGE" value="german"/>
  <parameter type="string" name="HTML_VIEWER" value="konqueror"/>
</controls>
```

A.4.2 Description

`<controls>`

The `<controls>` element surrounds all parameter declarations.
Parent element: none (root element of the `controls` document)
Child elements: `<parameter>`
This element does not contain attributes.

`<parameter>`

This element fixes one parameter that is passed to LiSA at program startup.
Parent element: `<controls>`
Child elements: none

Attribute	Meaning
`type`	The type of the parameter. Possible values are `string`, `integer`, `real`
`name`	The name of the parameter.
`value`	The value of the parameter.

A.4.3 LiSA-Program Parameters

Type	Name	Meaning
`integer`	`WIDTH`	Width of the LiSA window at the program startup (in pixels).
`integer`	`HEIGHT`	Height of the LiSA window at the program startup (in pixels).
`string`	`STARTFILE`	File name of a problem in XML format that is initially shown at the program startup.
`string`	`HTML_VIEWER`	The console command that LiSA should use to start the standard browser, for example `iexplore` (Windows), `konqueror` oder `firefox` (Linux)
`string`	`LANGUAGE`	Here the program language can be selected (either `english` or `german`).

Appendix B

GNU License Conditions

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so

Terms and conditions for copying, distribution and modification

itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system n on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation