TORSCHE Scheduling Toolbox for Matlab

User's Guide

(Release 0.4.0)

TORSCHE Scheduling Toolbox for Matlab

User's Guide

(Release 0.4.0; Rev. 1926)

Michal Kutil, Přemysl Šůcha, Michal Sojka and Zdeněk Hanzálek

Centre for Applied Cybernetics, Department of Control Engineering
Czech Technical University in Prague
{kutilm,suchap,sojkam1,hanzalek}@fel.cvut.cz
http://rtime.felk.cvut.cz/scheduling-toolbox/

Toolbox contributors: Roman Čapek, Miroslav Hájek, Jindřich Jindra, Jan Martinský, David Matějíček, Pavel Mezera, Josef Mrázik, Vojtěch Navrátil, Miloš Němec, Ondřej Nývlt, Martin Panáček Rostislav Prikner, Zdeněk Prokůpek, Milan Šilar and Miloslav Stibor.

Prague, October 12, 2007

TORSCHE Scheduling Toolbox for Matlab is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

TORSCHE Scheduling Toolbox for Matlab is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with Scheduling Toolbox; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Licences of external software packages are stated below:

- GLPK (GNU Linear Programming Kit) Version 4.6 is free software under GNU General Public License as published by the Free Software Foundation.
- MATLAB MEX INTERFACE FOR CPLEX is free software under the terms of the GNU Lesser General Public License as published by the Free Software Foundation.
- Utility "unzip.exe" is licenced by Info-ZIP (for more details see file *\scheduling\contrib\unzip\LICENSE*).
- Patch 2.5.9-6, libiconv-2.dll and libintl-2.dll is software under GNU General Public License as published by the Free Software Foundation.
- gzip (GNU zip) 1.2.4 is software under GNU General Public License as published by the Free Software Foundation.

Contents

List of Figures

2 Quick Start

3 Tasks

4 Sets of Tasks

6 Graphs

7 Scheduling Algorithms

8 Real-Time Scheduling

9 Graph Algorithms

10 Other Algorithms

11 Case Studies

List of Tables

6 Graphs

7 Scheduling Algorithms

9 Graph Algorithms

10 Other Algorithms

11 Case Studies

Chapter 1

Introduction

TORSCHE (Time Optimisation, Resources, SCHEduling) Scheduling Toolbox for Matlab is a freely (GNU GPL) available toolbox developed at the Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Control Engineering. The toolbox is designed to undergraduate courses and to researches in operations research or industrial engineering.

The current version of the toolbox covers following areas of scheduling: scheduling on monoprocessor/dedicated processors/parallel processors, cyclic scheduling and real-time scheduling. Furthermore, particular attention is dedicated to graphs and graph algorithms due to their large interconnection with scheduling theory. The toolbox offers transparent representation of scheduling/graph problems, various scheduling/graph algorithms, a useful graphical editor of graphs, an interface for Integer Linear Programming and an interface to TrueTime (MATLAB/Simulink based simulator of the temporal behaviour).

The scheduling problems and algorithms are categorized by notation $(\alpha \mid \beta \mid \gamma)$ proposed by [Graham79] and [Błażewicz83]. This notation, widely used in scheduling community, greatly facilitates the presentation and discussion of scheduling problems.

The toolbox is supplemented by several examples of real applications. The first one is scheduling of DSP algorithms on a HW architecture with pipelined arithmetic units. Further, there is an application of response-time analysis in real-time systems. The toolbox is equipped with sets of benchmarks from research community (e.g. DSP algorithms, Quadratic Assignment Problem).

We are pleased with growing number of users and we are very glad, that this toolbox will be cited in the third edition of the book 'Scheduling: Theory, Algorithms and Systems' by Michael Pinedo [Pinedo02].

This user's guide is organized as follows: Chapter 3, "Tasks", Chapter 4, "Sets of Tasks", Chapter 5, "Classification in Scheduling" and Chapter 6, "Graphs" presents the tool architecture and basic notation. The most interesting part is Chapter 7, "Scheduling Algorithms" describing implemented off-line scheduling algorithms demonstrated on various examples. Section Chapter 8, "Real-Time Scheduling" is dedicated to on-line scheduling and on-line scheduling algorithms. Graph algorithms are discussed in Chapter 9, "Graph Algorithms". Supplementary algorithms are described in Chapter 10, "Other Algorithms". The text is supplemented with case studies, presented in Chapter 11, "Case Studies", showing practical applications of the toolbox.

Chapter 2

Quick Start

2.1 Software Requirements

TORSCHE Scheduling Toolbox for Matlab (0.4.0) currently supports MATLAB 6.5 (R13) and higher versions. If you want to use the toolbox on different platforms than MS-Windows or Linux on PC (32bit) compatible, some algorithms must be compiled by a C/C++ compiler. We recommend to use Microsoft Visual C/C++ 7.0 and higher under Windows or gcc under Linux.

2.2 Installation

Download the toolbox from web <http://rtime.felk.cvut.cz/scheduling-toolbox/download.php> and unpack Scheduling toolbox into the directory where Matlab toolboxes are installed (most often in `<Matlab root>\toolbox` on Windows systems and on Linux systems in `<Matlab root>/toolbox`). Run Matlab and add two new paths into directories with Scheduling toolbox and demos, e.g.:

```
>> addpath(path,'c:\matlab\toolbox\scheduling')
>> addpath(path,'c:\matlab\toolbox\scheduling\stdemos')
```

Several algorithms in the toolbox are implemented as Matlab MEX-files (compiled C/C++ files). Compiled MEX-files for MS-Windows and Linux on PC (32bit) compatible are part of this distribution. If you use the toolbox on a different platform, please compile these algorithms using command `make` from `\scheduling` directory (in Matlab environment). Before that, please specify the compiler using command `mex -setup` from (also in Matlab environment). We suggest to use Microsoft Visual C/C++ or gcc compilers.

2.3 Help

To display a list of all available commands and functions please type

```
>> help scheduling
```

To get help on any of the toolbox commands (e.g. `task`) type

```
>> help task
```

To get help on overloaded commands, i.e. commands that do exist somewhere in Matlab path (e.g. `plot`) type

```
>> help task/plot
```

Or alternatively type `help plot` and then select `task/plot` at the bottom line line of the help text.

2.4 How to Solve Your Scheduling Problems

Solving procedure of your scheduling problem can be divided into four basic steps:

1. Define a set of tasks.
2. Define the scheduling problem.
3. Run the scheduling algorithm.

Task is defined by command `task`, for example:

```
>> t1 = task('task1', 5, 1, inf, 12)
Task "task1"
 Processing time: 5
 Release time:    1
 Due date:        12
```

This command defines task with name “task1”, processing time 5, release time 1, and duedate at time 12. In the same way we can define next tasks:

```
>> t2 = task('task2', 2, 0, inf, 11);
>> t3 = task('task3', 3, 5, inf, 9);
```

To create a set of tasks use command `taskset`:

```
>> T = taskset([t1 t2 t3])
Set of 3 tasks
```

For short:

```
>> T = [t1 t2 t3]
Set of 3 tasks
```

Due to great variety of scheduling problems, it is not easy to choose a proper algorithm. For easier selection of the proper algorithm, the toolbox uses a notation, proposed by [Graham79] and [Błażewicz83], to classify scheduling problems. Those classifications are created by command `problem`:

```
>> p=problem('1|pmtn,rj|Lmax')
1|pmtn,rj|Lmax
```

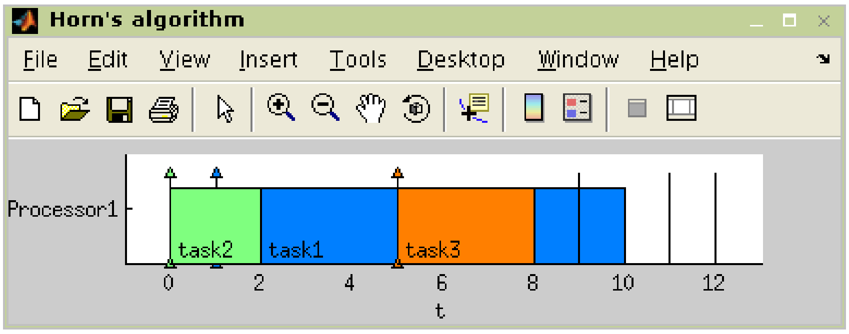
Now we can execute the scheduling algorithm, for example Horn’s algorithm:

```
>> TS = horn(T,p)
Set of 3 tasks
 There is schedule: Horn's algorithm
   Solving time: 0.29s
```

The final schedule, given by Gantt chart ,is shown in Figure 2.1. The figure is plotted by:

```
>> plot(TS)
```

Figure 2.1 The taskset schedule



2.5 Save and Load Functions

Data from the Matlab workspace can be saved and loaded by standard commands `save` and `load`. For example:

```
>> save file1
>> save file2 t1 t2

>> load file2
```

Chapter 3

Tasks

3.1 Introduction

Task is a basic term in scheduling problems describing a unit of work to be scheduled. The terminology is adopted from the following publications: I – [Błażewicz01], II – [Butazo97], III – [Liu00]. Graphic representation of task parameters is shown in Figure 3.1. Task `T`$_j$ in the toolbox is described by the following properties:

Name (`Name`)
label of the task

Processing time[I] **p_j** (`ProcTime`)
is the time necessary to execute task `T`$_j$ on the processor without interruption

(Computation time [II])

Release time[III] **r_j** (`ReleaseTime`)
is the time at which a task becomes ready for execution

(Arrival time [I,II], Ready time [I], Request time [II])

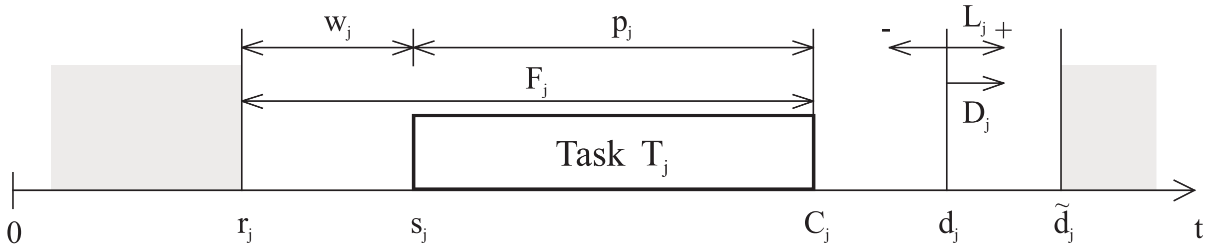
Deadline[I] **d_j** (`Deadline`)
specifies a time limit by which the task has to be completed, otherwise the scheduling is assumed to fail

Due date[I] **$\tilde{d}_j$** (`DueDate`)
specifies a time limit by which the task should be completed, otherwise the criterion function is charged by penalty

Weight[I] (`Weight`)
expresses the priority of the task with respect to other tasks (Priority [II])

Processor (`Processor`)
specifies dedicated processor on which the task must be executed

Figure 3.1 Graphics representation of task parameters



Rest of the task properties shown in Figure 3.1 are related to *start time* of task s_j, i.e. result of scheduling (see sections Section 3.4.1 and Section 4.3.2). Properties *completion time* C_j ($C_j=s_j+p_j$), *waiting time* w_j ($w_j=s_j+r_j$), *flow time* F_j ($F_j=C_j-r_j$), *lateness* L_j ($L_j=C_j+d_j$) and *tardiness* D_j ($D_j=\max\{C_j-d_j,0\}$) can be derived from start time s_j.

3.2 Creating the `task` Object

In the toolbox, task is represented by the object `task`. This object is created by the command with the following syntax rule (properties contained inside the square brackets are optional):

```
t1 = task([Name,]ProcTime[,ReleaseTime[,Deadline[,DueDate
          [,Weight[,Processor]]]]])
```

Command `task` is a constructor of object `task` and returns the object. In the syntax rule above the object is the variable `t1`. Examples of the object task creating are shown in Figure 3.2.

Figure 3.2 Creating task objects

```
>> t1 = task(5)
Task ""
 Processing time: 5
 Release time:    0
>> t2 = task('task2',5,3,12)
Task "task2"
 Processing time: 5
 Release time:    3
 Deadline:        12
>> t3 = task('task3',2,6,18,15,2,2)
Task "task3"
 Processing time: 2
 Release time:    6
 Deadline:        18
 Due date:        15
 Weight:          2
 Processor:       2
```

3.3 Graphical Representation of the `task` Object

Parameters of a task can be graphically displayed using command `plot`. For example parameters of task `t3`, created above, can be displayed by command:

```
>> plot(t3)
```

For more details see Reference Guide @task/plot.m.

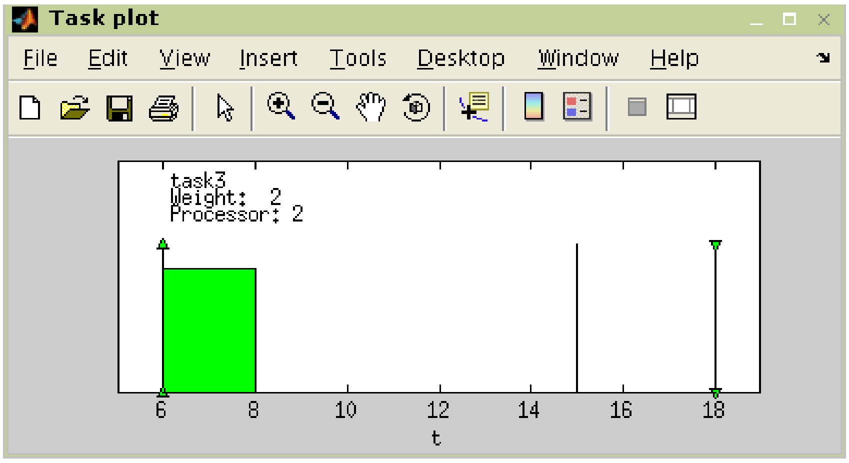
3.4 Object `task` Modifications

Command `get` returns the value of the specified property or values of all properties. Command `set` sets the value of the specified property. These two commands has the same syntax as is described in Matlab user's guide. Property access is allowed using the . (dot) operator too.

NOTE

To obtain a list of all accessible properties use command get. Note that some private and virtual properties aren't accessible using the . (dot) operator, although they are presented when the automatic completion by Tab key is used.

Figure 3.3 Plot example



An example of task modification:

```
>> get(t3)
           Name: 'task3'
       ProcTime: 2
    ReleaseTime: 6
       Deadline: 18
        DueDate: 15
         Weight: 2
      Processor: 2
      UserParam:
          Notes: ''
>> set(t3,'ProcTime',4)
>> get(t3,'ProcTime')

ans =

     4
```

3.4.1 Start Time of Task

Command `add_scht` adds the start time into an object task. Schedule of a task is described by three arrays (`start`, `length`, `processor`). The length of array is equal to number of task preemptions minus one. Opposite command to `get_scht is appointed for getting a schedule from the task object.`For more details see Reference Guide @task/add_scht.m , @task/get_scht.m.

3.4.2 Color Modification

Commands `set_graphic_param` and `get_graphic_param` can be used to define color of tasks. If color of task is set, command `plot` will use it. Use of these commands is shoved on the following example:

```
>> t = task('task',5);
>> set_graphic_param(t,'color','red')
>> get_graphic_param(t,'color')

ans =

red
```

3.5 Periodic Tasks

Periodic tasks are tasks, which are released periodically with a fixed period. There is a `ptask` object in TORSCHE that allows users to work with periodic tasks. Periodic tasks are mainly used in real-time scheduling area (see Chapter 8, "Real-Time Scheduling").

3.5.1 Creating the `ptask` Object

The syntax of `ptask` constructor is:

```
pt = ptask([Name,]ProcTime,Period[,ReleaseTime[,Deadline[,Duedate[,Weight[,Processor]]]]])
```

Almost all parameters are the same as for `task` object except for `Period`, which specifies the period of the task.

3.5.2 Working with `ptask` Objects

The way of manipulating `ptask` objects is the same as for `task` objects. It is possible to change their properties using `set` and `get` methods as well as by dot notation. In addition, there is `util` method which returns CPU utilization factor of the task.

Chapter 4

Sets of Tasks

4.1 Creating the `taskset` Object

Objects of the type task can be grouped into a set of tasks. A set of tasks is an object of the type `taskset` which can be created by the command `taskset`. Syntax for this command is:

```
T = taskset(tasks[,prec])
```

where variable `tasks` is an array of objects of the type `task`. Furthermore, relations between tasks can be defined by *precedence constraints* in parameter `prec`. Parameter `prec` is an adjacency matrix (see Chapter 6, “Graphs”) defining a graph where nodes correspond to tasks and edges are precedence constraints between these tasks. If there is an edge from $\mathtt{T}_\mathrm{i}$ to $\mathtt{T}_\mathrm{j}$ in the graph, it means that $\mathtt{T}_\mathrm{i}$ must be completed before $\mathtt{T}_\mathrm{j}$ can be started.

If there are not precedence constraints between the tasks, we can use a shorter form of creating a set of tasks using square brackets (see the first line in Figure 4.1).

Figure 4.1 Creating a set of tasks and adding precedence constraints

```
>> T1 = [t1 t2 t3]
Set of 3 tasks

>> T1 = taskset(T1,[0 1 1; 0 0 1; 0 0 0])
Set of 3 tasks
 There are precedence constraints

>> T2 = taskset([3 4 2 4 4 2 5 4 8])
Set of 9 tasks
```

You can also create a set of tasks directly from a vector of processing times. Call the command `taskset` as shown in Figure 4.1. Tasks with those processing times will be automatically created inside the set of tasks. Precedence constraints can be added in the same way as in case of taskset `T1` (see Figure 4.1).

4.2 Graphical Representation of the Set of Tasks

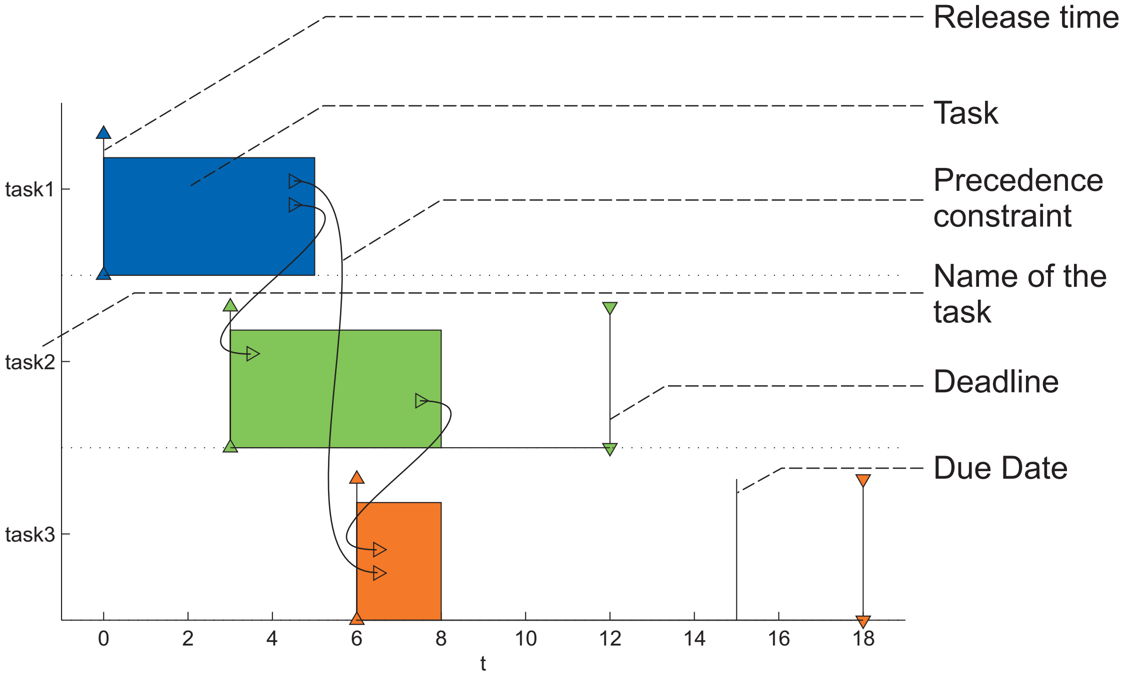
As for single tasks, command `plot` can be used to draw parameters of set of tasks graphically. An example of plot output with explanation of used marks is shown in Figure 4.2. For more details see Reference Guide @taskset/plot.m.

4.3 Set of Tasks Modification

Commands changing parameters of tasksets are the same as for task object. Command `get` returns the value of the specified property or values of all properties. Command `set` sets the value of the specified property. These two commands has got a standard syntax, which is described in Matlab user manual. Property access is allowed over the . (dot) operator too.

Figure 4.2 Gantt chart for a set of scheduled tasks

```
>> plot(T1)
```



NOTE

To obtain a list of all accessible properties use command `get`. Note that some private and virtual properties aren't accessible over the . (dot) operator, although they are displayed when the automatic completion by Tab key is used.

4.3.1 Modification of Tasks Parameters Inside the Set of Tasks

Tasks parameters may be modified via virtual properties of object taskset. The list of virtual properties are: `Name`, `ProcTime`, `ReleaseTime`, `Deadline`, `DueDate`, `Weight`, `Processor`, `UserParam`. All parameters are arrays data type. Items order in the arrays is the same as tasks order in the set of the tasks.

Figure 4.3 Access to the virtual property examples

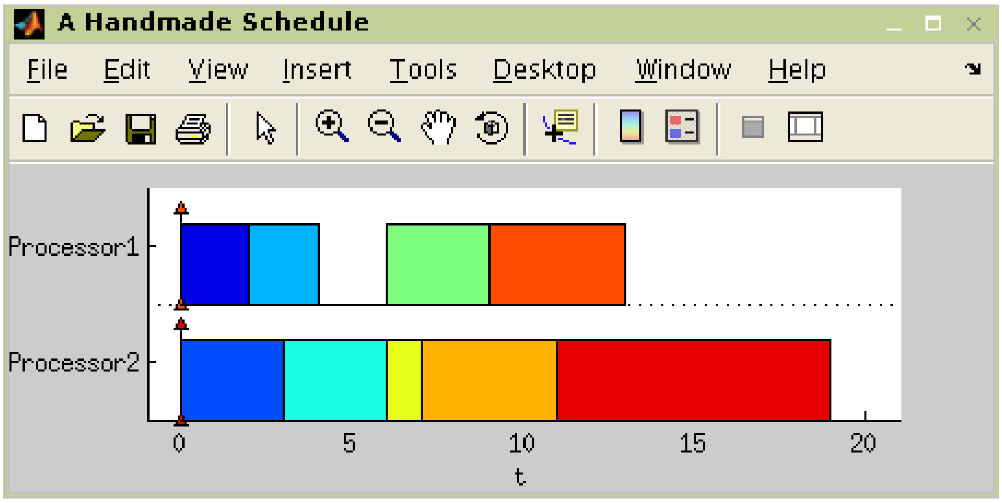
```
>> T2.ProcTime
ans =
     3     4     2     4     4     2     5     4     8
>> T2.ProcTime(3) = 5;
>> T2.ProcTime
ans =
     3     4     5     4     4     2     5     4     8
>> T2.ProcTime = T2.ProcTime -1;
>> T2.ProcTime
ans =
     2     3     4     3     3     1     4     3     7
```

4.3.2 Schedule

The only way how to operate with schedule of tasks is through commands `add_schedule` and `get_schedule`. Command `add_schedule` inserts a schedule (i.e. start time s_j, number of assigned processor, ...) into taskset object. Its syntax is described in Reference Guide @taskset/add_schedule.m. An example of `add_schedule` command use is shown in Figure 4.4. Vector `start` is vector of start times (i.e. first task starts at 0), vector processor is vector of assigned processors (i.e. first task is assigned to the firs processor) and string description is a brief note on used scheduling algorithm.

Figure 4.4 Schedule inserting example

```
>> start = [0 0 2 3 6 6 7 9 11];
>> processor = [1 2 1 2 1 2 2 1 2];
>> description = 'a handmade schedule';
>> add_schedule(T2,description,start,T2.ProcTime,processor);
>>
>> get_schedule(T2)
ans =
     0     0     2     3     6     6     7     9    11
>> plot(T2);
```



On the other hand, the schedule can be obtained from a taskset using command `get_schedule` (e.g. as is shown in Figure 4.4). For more details about this function see Reference Guide @taskset/get_schedule.m. Graphical schedule interpretation (Gantt chart) can be obtained using function `plot`.

Parameters of a given schedule (e.g. value of optimality criteria, solving time, ...) can be obtained using function `schparam`. It returns information about schedule inside the taskset and its syntax is described in Reference Guide @taskset/schparam.m. An example of use is shown in Figure 4.5.

Figure 4.5 Schedule parameters

```
>> param = schparam(T2,'cmax')
param =
    19

>> param = schparam(T2)
param =
          cmax: 19
         sumcj: 80
        sumwcj: 80
```

4.4 Other Functions

4.4.1 Count and Size

Commands `count(T)` and `size(T)` return number of tasks in the set of tasks `T`. At this moment they return the same value. Returned value will be different after implementing the general shop problems into the toolbox. Now it is recommended to use command `count`.

4.4.2 Sort

The function returns sorted set of tasks inside taskset over selected parameter. Its syntax is described in Reference Guide @taskset/sort.m. An example is shown in Figure 4.6.

Figure 4.6 Taskset sort example

```
>> T2.ProcTime
ans =
     2     3     4     3     3     1     4     3     7
>> T3 = sort(T2,'ProcTime','dec');
>> T3.ProcTime
ans =
     7     4     4     3     3     3     3     2     1
```

4.4.3 Random taskset

Random taskset `T` can be created by the command `randtaskset`. Tasks parameters in the taskset are generated with a uniform distribution. The syntax is described in Reference Guide randtaskset.m. Example of its application is shown in Figure 4.7.

Figure 4.7 Example of random taskset use

```
>> T = randtaskset(8,[8 15],[3 6]);

>> T.ProcTime
ans =
    15    12    14    11    14    12    14     9

>> T.ReleaseTime
ans =
     4     4     5     3     4     5     5     4
```

Note

Random task can be created by command `randtask`.

Chapter 5

Classification in Scheduling

5.1 The `problem` Object

The object `problem` is a structure describing the classification of deterministic scheduling problems in the notation proposed by [Graham79] and [Błażewicz83]. An example of its usage is shown in the following code.

```
>> prob = problem('P|prec|Cmax')
P|prec|Cmax
```

This notation consists of three parts ($\alpha \mid \beta \mid \gamma$). The first part (`alpha`) describes the processor environment, the second part (`beta`) describes the task characteristics of the scheduling problem as precedence constraints, or release times. The last part (`gamma`) denotes an optimality criterion.

Special problems, not specified by the notation, can be identified by one-word name, e.g. `CSCH`. For more information see Reference Guide @problem/problem.m.

Command `is` is used to test whether a notation includes specific description. A simple problem test should be included in each scheduling algorithm of the toolbox. An example is shown below.

```
if ~is(prob,'alpha','P') |  ~is(prob,'betha','rj') | ~is(prob,'gamma','Lmax')
     error('Can not solve this scheduling problem.');
end
```

Chapter 6

Graphs

6.1 Introduction

Graphs and graph algorithms are often used in scheduling algorithms, therefore operations with graphs are supported in the toolbox. A graph is data structure including a set of nodes, a set of edges and information on their relations. As it is known from definition of graph from graph theory: G = (V,E,ε). If ε is a binary relation of E over V, then G is called a direct graph. When there is no concern about the direction of an edge, the graph is called undirected. Object Graph in the toolbox is described as directed graph. Undirected graph can be created by addition of another identical edge in opposite direction.

6.2 Creating Object `graph`

There is a few of different ways of creating a graph because there are several methods how to express it. The object graph is generally described by an adjacency matrix[1]. Graph object is created by the command with the following syntax:

```
g = graph('adj',A)
```

where variable `A` is an adjacency matrix. It is also possible to describe a graph by an incidency matrix[2]. The syntax is:

```
g = graph('inc',I)
```

where vaiable `I` is an incidency matrix. Another way of creating Graph object is based upon a matrix of edges weights[3]. It is obvious, that just simple graphs can be created by this way. The syntax in this case is:

```
g = graph(B)
```

Any key-word is not required here. This method is considered to be default one because it makes easy setting of weight of an edge. The value of weight is automatically saved as user parameter of the edge. The most complex way of graph creating is definition by a list of edges (or/and nodes). The list of edges (nodes) in the form of cell type is ordered as an argument of the graph function. The cell contains information about initial and terminal node (or number of the node) and arbitrary count of user parameters (e.g. weight of an edge/node). The syntaxe is:

```
g = graph('edl',edgeList,'ndl',nodeList)
```

where `edgeList` is list of edges and `nodeList` is list of nodes. An example of graphs creation is shown in Figure 6.1.

Another possibility to create the object graph is to use a tool [Graphedit], or transform an object taskset to the graph (see Section 6.5).

---

[1] The adjacency matrix $A = (a_{ij})_{n\times n}$ of G is defined by $a_{ij} := \left\{\begin{array}{l} 1,2,\dots \quad \text{if } v_i v_j \in E \\ 0 \quad \text{otherwise.} \end{array}\right.$.

[2] The incidency matrix $I = (i_{jk})_{n\times n}$ of G is defined by $a_{ij} := \left\{\begin{array}{l} 1,2,\dots \quad \text{if } v_i v_j \in E \\ 0 \quad \text{otherwise.} \end{array}\right.$.

[3] The matrix of weights B = (b_{ij})_{n \times n} of G is defined by b_{ij}:=\left{\begin{array}{l} weight_{k} \quad \mbox{ if } v_{i} v_{j} \in E \\ Inf \quad \mbox{otherwise.} \end{array} \right, where weight_{k} matches weight of k-th edge

Figure 6.1 Creating a graphs from adjacency matrix

```
>> g1 = graph('adj',[0 2 0; 0 1 0; 0 0 0])

 adjacency matrix:
     0     2     0
     0     1     0
     0     0     0

>> g2 = graph([0 2; 1 0])

 adjacency matrix:
     0     1
     1     0

>> g3 = graph('inc',[0 1 0 -1 -1; 1 0 -1 1 1; -1 -1 1 0 0])

 adjacency matrix:
     0     0     1
     2     0     1
     0     1     0

>> g4 = graph('edl',{1,2, 35,[5 8]; 2,3, 68,[2 7]})

 adjacency matrix:
     0     1     0
     0     0     1
     0     0     0
```

6.3 Object `graph` Modification

Command `get` returns a value of the specified property or values of all properties. Command `set` sets the value of the specified property. These two commands have the same syntax as is described in Matlab user's guide. Property access is allowed over the . (dot) operator too.

To obtain list of parameters which can by modified use Command `set`, as is shown bellow.

Figure 6.2 Command set for graph

```
>> set(g2)
                 Name: Name of the GRAPHS
                    N: Array of nodes
                    E: Array of edges
            UserParam: User parameters
            DataTypes: Type of UserParam's data
                Color: Color of area of the GRAPH
             GridFreq: Grid frequency
                Notes: An arbitrary string
                  inc: Incidency matrix
                  adj: Adjacency matrix
                  edl: List of edges (cell)
                  ndl: List of nodes (cell)
edgeUserparamDatatype: Cell of data types of edges' UserParam
nodeUserparamDatatype: Cell of data types of nodes' UserParam
```

Property `Name` is a graph name and property `N` is an array of node objects. Object node includes all information about node such as name, user parameters ... Property `E` is an array of edge objects where object edge carries all information about edge. Edge order in array is determined by command `between`. This command returns edge indexes for all edges which interconnect two nodes.

6.3.1 User Parameters on Edges

Many algorithms (e.g. for cyclic scheduling in Section 9.4) consider an edge-weighted graph, i.e. edges are weighted by one or more parameters. In object `graph` these parameters are stored in user parameters of corresponding edges. To facilitate access to user parameters, the toolbox contains two couples of functions for geting/seting data from/to user parameters.

Table 6.1 List of functions

function	description
`UserParam = edge2param(g)`	Returns user parameters of edges in graph `g` as an `n`-by-`n` matrix if the graph is simple and the value of user parameters is numeric, cell array otherwise.
`g = param2edge(g,UserParam)`	Adds data in an `n`-by-`n` matrix or cell array to user parameters of edges in graph `g`.
`UserParam = node2param(g)`	Returns user parameters of nodes in graph `g` as a numeric array or cell array.
`g = param2node(g,UserParam)`	Adds data in a numeric array or cell array to user parameters of n in graph `g`.

In addition, functions `edge2param` and `param2edge` are further extended by optional parameters. The `I`-th user parameter can be accessed using

```
userParam = edge2param(g,I)
```

and

```
g = param2edge(g,userParam,I)
```

Analogical syntax is valid for functions `node2param` and `param2node`.

If there is not edge between two nodes, the corresponding user parameter is considered to be `Inf`. If there are parallel edges or matrix UserParam does not match with graph `g`, the algorithm returns cell array. The different value indicating that there is not an edge can be defined as a parameter `notEdgeParam`.

```
UserParam = edge2param(g,I,notEdgeParam)
```

and

```
g = param2edge(g,UserParam,I,notEdgeParam)
```

An example of practical usage is shown in example of [Critical circuit ratio] computation.

6.4 Graphedit

The toolbox is equipped with a simple but useful edi-tor of graphs called Graphedit based on System Handle Graphics of Matlab. It allows construct directed graphs with various user parameters on nodes and edges by simple and intuitive way. The constructed graph can be easily used in the toolbox as instance of object Graph described in the previous subsections, which can be exported to workspace or saved to binary mat-file.

The Graphedit is depicted Figure 6.3 . As you can see, drawing canvas is the dominant item in the main window. One canvas presents one edited graph. In the bottom of the window are tabs for switching canvases. So there is possibility to work independently with several graphs in one Graphedit. User can find all functions of Graphedit in the main menu. The most used ones are accessible via icons in the toolbar. Properties of graph and its edges and nodes (name, user pa-rameters, color...) may be edited in property editor which is a part of Graphedit.

Graphedit has the following syntax

```
graphedit(g)
```

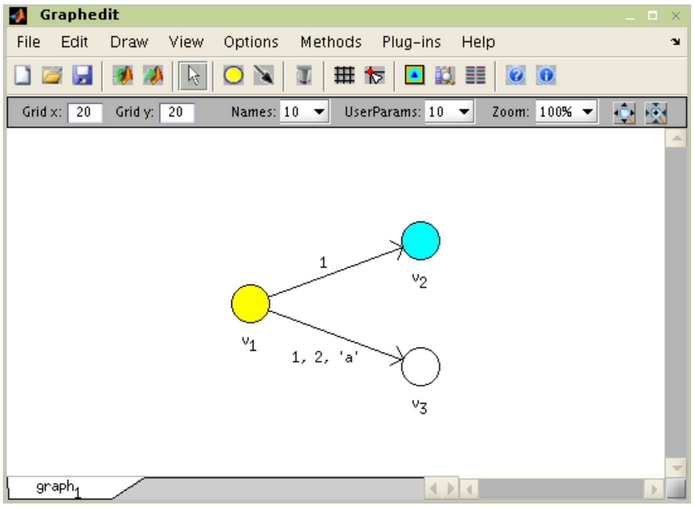
where `g` is an object graph. To open graphedit with an empty plot call Graphedit without parameters.

6.4.1 The Graph Construction

Graphedit operates in four editing modes (Add node, Add edge, Delete and Edit). Selection of mode can be made by four buttons (depicted in Figure 6.3) in the toolbar or in main menu. Mode **Add node** is used for creating and placing of node, mode **Add edge** for connecting nodes by edge, mode **Delete** for deleting nodes or edges by clicking on it. And properties of nodes and edges can be edited in **Edit** mode.

Graphedit also offers a possibility to choice appear-ance of a node and contains tool for design your own node-picture which can be formed by bitmapped im-age or any geometric pattern or their arbitrary combi-nation. This function has nothing to do with graph theory; however it is useful for presentation purposes. The system of designing own nodes is displayed in Fig 6.

Figure 6.3 Graphedit



Grid x: 20 Grid y: 20 Names: 10 UserParams: 10 Zoom: 100%

6.4.1.1 Placing of Nodes and Edges

Placing of nodes or edges is accomplished just by selecting aporiate drawing mode and clicking of the mouse. Each node can be moved by dragging it to a new location. Because the change of shape of edge is often required, every edge is represented by Bézier curves. The way of editing its curve is very similar to way known from common drawing tools dealing with vector graphics. By right click in the edge you display context menu and in it choose 'Edit'. Shape of the edge can be changed by draging little square which has appeared.

6.4.2 Plug-ins

Graphedit contains system of plug-ins. It is very helpful tool which allows execution of almost arbitrary function right from GUI of Graphedit via main menu. The function may be some algorithm from toolbox or code implemented by user. The only one condition is, that the function must have object graph as first argument. Other input arguments may be arbitrary; output of the function can be anything – Graph objects will be automatically drown, other data types will be saved into workspace.

By selecting 'Add New Plug-in' in main menu of the Graphedit you can plug in chosen function. Its removing is possible by 'Remove Plugin'.

6.4.3 Property editor

You can displey Property Editor window by main menu 'View' - 'Property Editor' or by apropriate icon in the toolbar. When graph, node or edge is selected its parameters will be shown in Editeable fields. Values of properties will be changed by enter new data to these fields.

6.4.4 Export/Import to/from Matlab workspace

Data between Graphedit and Matlab workspace are mostly exchanged in the form of graph object. Exporting and importing is possible by main menu or by icons in Graphedit's toolbar. Before exporting, user is asked to order a name of variable to which will be current graph saved. The same pays for importing a graph object from the workspace.

6.4.5 Saving/Loading to/from Binary File

Saving and loading proceeds by way similar to exporting and importing. The only change of it is in necessity to select a file to save graph to or loading graph from.

6.4.6 Change of Appearance of Nodes

Graphedit also offers a possibility to choice appearance of a node and contains tool for design your own node-picture which can be formed by bitmapped image or any geometric pattern or their arbitrary combination. This function has nothing to do with graph theory; however it is useful for presentation purposes. The system of designing own nodes is called Node Designer and it accessible by main menu or icon in the toolbar.

6.5 Transformations Between Objects `taskset` and `graph`

Object graph can be transformed to the object taskset and taskset can be transformed back to the object graph. Obviously, the nodes from graph are transformed to the tasks in taskset and edges are transformed to the precedence constrains and vice versa.

6.5.1 Transformations from `graph` to `taskset`

The object graph `g` can be transformed to the taskset as follows:

```
T = taskset(g)
```

Each node from the graph G will be converted to a task. Tasks properties (e.g. Processing Time, Deadline …), are taken from node `UserParam` attribute. The assignment of the attributes of the nodes can be specified in optional parameters of function `taskset`. For example, whet the first element of the node `UserParam` attribute contains processing time of the task and the second one contains the name of the task, the conversion can be specified as follows

```
T = taskset(g,'n2t',@node2task,'proctime','name')
```

Default order of `UserParam` attribute is:

```
{'ProcTime','ReleaseTime','Deadline','DueDate','Weight','Processor',
 'UserParam'}
```

All edges are automatically transformed to the task precedence constrains. Their parameters are saved to the cell array in:

```
T.TSUserParam.EdgesParam
```

For more details please see Reference Guide @taskset/taskset.m.

6.5.2 Transformations from `taskset` to `graph`

It is possible to transform taskset to the graph object. The command for transformation is

```
g = graph(T)
```

All parameters from taskset are transformed into the graph variables in the opposite direction than was described above.

For more information about parametrization of tasks to/from node and precedence constrains to/from edge transformations see `taskset` and `graph` help or Reference Guide @graph/graph.m and @taskset/taskset.m.

Chapter 7

Scheduling Algorithms

Scheduling algorithms are the most interesting part of the toolbox. This section deal with scheduling on monoprocessor/dedicated processors/parallel processors and with cyclic scheduling. The scheduling algorithms are categorized by notation $(\alpha \mid \beta \mid \gamma)$ proposed by [Graham79] and [Błażewicz83].

7.1 Structure of Scheduling Algorithms

Scheduling algorithm in TORSCHE is a Matlab function with at least two input parameters and at least one output parameter. The first input parameter must be taskset, with tasks to be scheduled. The second one must be an instance of problem object describing the reguired scheduling problem in $(\alpha \mid \beta \mid \gamma)$ notation. Taskset containing resulting schedule must be the first output parameter. Common syntax of the scheduling algorithms calling is:

```
TS = name(T,problem[,processors[,parameters]])
```

name
command name of algorithm

TS
set of tasks with schedule inside

T
set of tasks to be scheduled

problem
object of type problem describing the classification of deterministic scheduling problems

processors
number of processors for which schedule is computed

parameters
additional information for algorithms, e.g. parameters of mathematical solvers etc.

The common structure of scheduling algorithms is depicted in Figure 7.1. First of all the algorithm must check whether the reguired scheduling problem can be solved by himself. In this case the function `is` is used as is shown in part "scheduling problem check". Further, algorithm should perform initialization of variables like `n` (number of tasks), `p` (vector of processing times), ... Then a scheduling algorithm calculates start time of tasks (`starts`) and processor assignemen (`processor`) - if required. Finaly the resulting schedule is derived from the original taskset using function `add_schedule`.

7.2 List of Algorithms

Table 7.1 shows reference for all the scheduling algorithms available in the current version of the toolbox. Each algorithm is described by its full name, command name, problem clasification and reference to literature where the problem is described.

Figure 7.1 Structure of scheduling algorithms in the toolbox.

```
function [TS] = schalg(T,problem)
%function description

%scheduling problem check
if ~(is(prob,'alpha','P2') && is(prob,'betha','rj,prec') && ...
     is(prob,'gamma','Cmax'))
     error('Can not solve this problem.');
end

%initialization of variables
n = count(T);              %number of tasks
p = T.ProcTime             %vector of processing time

%scheduling algorithm
...
starts = ...               %assignemen of resulting start times
processor = ...            %processor assignemen

%output schedule construction
description = 'a scheduling algorithm';
TS = T;
add_schedule(TS, description, starts, p, processor);

%end of file
```

Table 7.1 List of algorithms

algorithm	command	problem	reference
[Algorithm for 1|rj|Cmax]	alg1rjcmax	1|rj|Cmax	[Błażewicz01]
[Bratley's Algorithm]	bratley	1|rj,~dj|Cmax	[Błażewicz01]
[Hodgson's Algorithm]	alg1sumuj	1||$\sum$Uj	[Błażewicz01]
[Algorithm for P||Cmax]	algpcmax	P||Cmax	[Błażewicz01]
[McNaughton's Algorithm]	mcnaughtonrule	P|pmtn|Cmax	[Błażewicz01]
[Algorithm for P|rj,prec,~dj|Cmax]	algprjdeadlinepreccmax	P|rj,prec,~dj|Cmax	
[Hu's Algorithm]	hu	P|in-tree,pj=1|Cmax	[Błażewicz01]
[Brucker's algorithm]	brucker76	P|in-tree,pj=1|Lmax	[Bru76], [Błażewicz01]
[Horn's Algorithm]	horn	1|pmtn,rj|Lmax	[Horn74], [Błażewicz01]
[List Scheduling]	listsch	P|prec|Cmax	[Graham66], [Błażewicz01]
[Coffman's and Graham's Algorithm]	coffmangraham	P2|prec,pj=1|Cmax	[Błażewicz01]
[Scheduling with Positive and Negative Time-Lags]	spntl	SPNTL	[Brucker99], [Hanzalek04]
[Cyclic scheduling (General)]	cycsch	CSCH	[Hanen95], [Sucha04]
[SAT Scheduling]	satsch	P|prec|Cmax	[TORSCHE06]

7.3 Algorithm for Problem 1|rj|Cmax

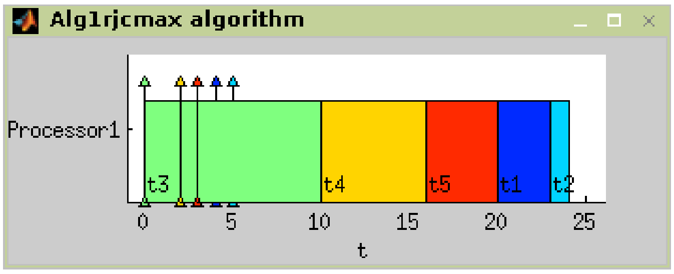
This algorithm solves 1|r_j|Cmax scheduling problem. Tha basic idea of the algorithm is to arrange and schedule the tasks in order of nondecreasing release time r_j. It is equivalent to the First Come First Served rule (FCFS). The algorithm usage is outlined in Figure 7.1 and the correspondin schedule is displayed in Figure 7.2 as a Gantt chart.

```
TS = alg1rjcmax(T,problem)
```

Figure 7.2 Scheduling problem 1|r_j|Cmax solving.

```
>> T = taskset([3 1 10 6 4]);
>> T.ReleaseTime = ([4 5 0 2 3]);
>> p = problem('1$|$rj$|$Cmax');
>> TS = alg1rjcmax(T,p);
>> plot(TS);
```

Figure 7.3 Alg1rjcmax algorithm - problem 1|r_j|Cmax



7.4 Bratley's Algorithm

Bratley's algorithm, proposed to solve 1|r_j,˜d_j|Cmax problem, is algorithm which uses branch and bound method. Problem is from class NP-hard and finding best solution is based on backtracking in the tree of all solutions. Number of solutions is reduced by testing availabilty of schedule after adding each task. For more details about Bratley's algorithm see [Błażewicz01].

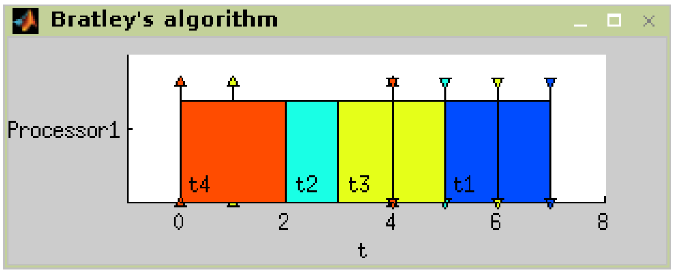
In Figure 7.3 the algorithm usage is shown. The resulting schedule is shown in Figure 7.4.

```
TS = bratley(T,problem)
```

Figure 7.4 Scheduling problem 1|r_j,˜d_j|Cmax solving.

```
>> T = taskset([2 1 2 2]);
>> T.ReleaseTime = ([4 1 1 0]);
>> T.Deadline = ([7 5 6 4]);
>> p = problem('1|rj,~dj|Cmax');
>> TS = bratley(T,p);
>> plot(TS);
```

Figure 7.5 Bratley's algorithm - problem 1|r_j,˜d_j|Cmax



7.5 Hodgson’s Algorithm

Hodgson’s algorithm is proposed to solve 1||$\sum$Uj problem, that means it minimalize number of delayed tasks. Algorithm operates in two steps:

1. The subset Ts of taskset T, that can be processed on time, is determined.
2. A schedule is determined from the subsets Ts and Tn = T − Ts (tasks, that can not be processed on time).

Implementation: Apply EDD (Earliest Due Date First) rule on taskset T. If each task can be processed on time, then this is the final schedule. Else move as much tasks with the longest processing time from Ts to Tn as is needed to process each task from Ts on time. Then schedule subset Tn in an arbitrary order. Final schedule is [Ts Tn]. For more details about Hodgson’s algorithm see [Błażewicz01].

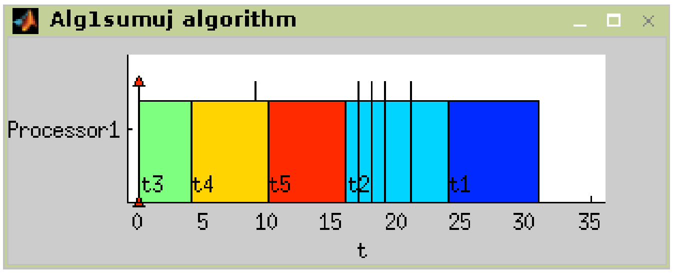
In Figure 7.5 the algorithm usage is outlined. The resulting schedule is displayed in Figure 7.6.

```
TS = alg1sumuj(T,problem)
```

Figure 7.6 Scheduling problem 1||$\sum$Uj solving.

```
>> T = taskset([7 8 4 6 6]);
>> T.DueDate = ([9 17 18 19 21]);
>> p = problem(’1||sumUj’);
>> TS = alg1sumuj(T,p);
>> plot(TS);
```

Figure 7.7 Hodgson’s algorithm - problem 1||$\sum$Uj



7.6 Algorithm for Problem P||Cmax

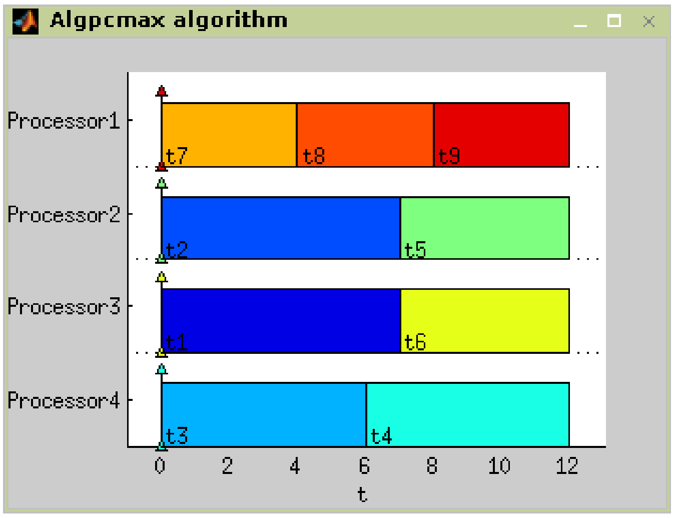
This algorithm solves problem P||Cmax, where a set of independent tasks has to be assigned to parallel identical processors in order to minimize schedule length. Preemption is not allowed. Algorithm finds optimal schedule using Integer Linear Programming (ILP). The algorithm usage is outlined in Figure 7.7 and resulting schedule is displayed in Figure 7.8.

```
TS = algpcmax(T,problem,processors)
```

Figure 7.8 Scheduling problem P||Cmax solving.

```
>> T=taskset([7 7 6 6 5 5 4 4 4]);
>> T.Name={’t1’ ’t2’ ’t3’ ’t4’ ’t5’ ’t6’ ’t7’ ’t8’ ’t9’};
>> p = problem(’P$||$Cmax’);
>> TS = algpcmax(T,p,4);
>> plot(TS);
```

Figure 7.9 Algpcmax algorithm - problem P||Cmax



7.7 McNaughton's Algorithm

McNaughton's algorithm solves problem P|pmtn|Cmax, where a set of independent tasks has to be scheduled on identical processors in order to minimize schedule length. This algorithm consider preemption of the task and the resulting schedule is optimal. The maximum length of task schedule can be defined as maximum of this two values: max(p_j); (Σp_j)/m, where m means number of processors. For more details about Hodgson's algorithm see [Błażewicz01].

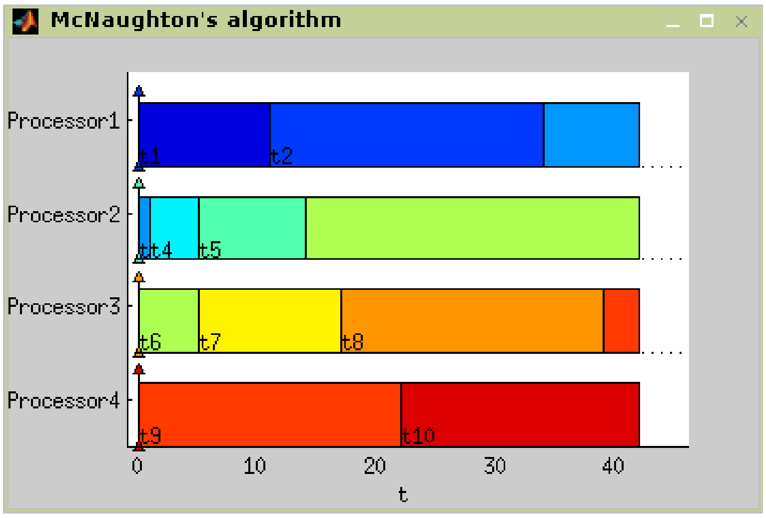
The algorithm use is outlined in Figure 7.9. The resulting Gantt chart is shown in Figure 7.10.

```
TS = mcnaughtonrule(T,problem,processors)
```

Figure 7.10 Scheduling problem P|pmtn|Cmax solving.

```
>> T = taskset([11 23 9 4 9 33 12 22 25 20]);
>> T.Name = {'t1' 't2' 't3' 't4' 't5' 't6' 't7' 't8' 't9' 't10' };
>> p = problem('P|pmtn|Cmax');
>> TS = mcnaughtonrule(T,p,4);
>> plot(TS);
```

Figure 7.11 McNaughton's algorithm - problem P|pmtn|Cmax



7.8 Algorithm for Problem P|rj,prec,˜dj|Cmax

This algorithm is designed for solving P|r_j,prec,˜d_j|Cmax problem. The algorithm uses modified List Scheduling algorithm [List Scheduling] to determine an upper bound of the criterion Cmax. The optimal schedule is found using ILP(integer linear programming).

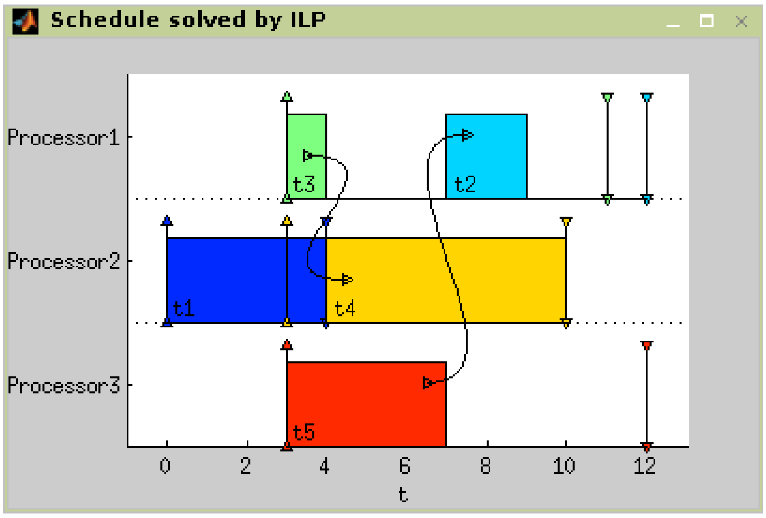
In Figure 7.11 the algorithm usage is shown. The resulting Gantt chart is displayed in Figure 7.12.

```
TS = algprjdeadlinepreccmax(T,problem,processors)
```

Figure 7.12 Scheduling problem P|p_j,prec,˜d_j|Cmax solving.

```
>> t1 = task('t1',4,0,4);
>> t2 = task('t2',2,3,12);
>> t3 = task('t3',1,3,11);
>> t4 = task('t4',6,3,10);
>> t5 = task('t5',4,3,12);
>> prec = [0 0 0 0 0;...
           0 0 0 0 0;...
           0 0 0 1 0;...
           0 0 0 0 0;...
           0 1 0 0 0];
>> T = taskset([t1 t2 t3 t4 t5],prec);
>> prob = problem('P$|$rj,prec,\textasciitilde{}dj$|$Cmax');
>> TS = algprjdeadlinepreccmax(T,prob,3);
>> plot(TS);
```

Figure 7.13 Algprjdeadlinepreccmax algorithm - problem P|p_j,prec,˜d_j|Cmax



7.9 List Scheduling

List Scheduling (LS) is a heuristic algorithm in which tasks are taken from a pre-specified list. Whenever a machine becomes idle, the first available task on the list is scheduled and consequently removed from the list. The availability of a task means that the task has been released. If there are precedence constraints, all its predecessors have already been processed. [Leung04] The algorithm terminates when all the tasks from the list are scheduled. In multiprocessor case, the processor with minimal actual time is taken in each iteration of the algorithm.

Heuristic (suboptimal) algorithms do not guarantee finding the optimal. A subset of heuristic algorithms constitute approximation algorithms . It is a group of heuristic algorithms with analytically evaluated accuracy. The accuracy is measured by *absolute performance ratio*. For example when the objective of scheduling is to minimize $\mathtt{C}_{\mathrm{max}}$, absolute performance ratio is defined as $R_A = inf\,\{r \geq 1 | C_{max}(A(I))/C_{max}(OPT(I)) \forall I \in \Pi\}$, where $\mathtt{C}_{\mathrm{max}}$`(A(I))` is $\mathtt{C}_{\mathrm{max}}$obtained by approximation algorithm `A`, $\mathtt{C}_{\mathrm{max}}$(OPT(I)) is $\mathtt{C}_{\mathrm{max}}$obtained by an optimal algorithm [Błażewicz01] and Π is a set of all instances of the given scheduling problem. For an arbitrary List Scheduling algorithm is proved that $\mathtt{R}_{\mathrm{LS}}$`=2-1/m`, where `m` is the number of processors. Time complexity of the LS algorithm is `O(n)`.

List Scheduling algorithm is implemented in Scheduling Toolbox as function:

```
TS = listsch(T,problem,processors [,strategy])

TS = listsch(T,problem,processors [,schoptions])
```

T
set of tasks

problem
object problem

processors
number of processors

strategy
strategy for LS algorithm

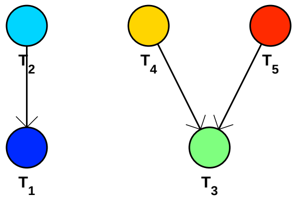
schoptions
optimization options (see Section [Scheduling Toolbox Options])

The algorithm is able to solve R|prec|Cmax or any easier problem. For more details about List Scheduling algorithm see [Błażewicz01].

Example 7.9.1 List Scheduling - problem P|prec|Cmax.

The set of tasks contains five tasks named {‘t1’, ‘t2’, ‘t3’, ‘t4’, ‘t5’} with processing times [2 3 1 2 4]. The tasks are constrained by precedence constraints as shown in Figure 7.14.

Figure 7.14 An example of P|prec|Cmax scheduling problem.



The solution of the example is shown in Figure 7.16. The LS algorithm found a schedule with $\mathtt{C}_{\mathrm{max}}$`=` 7.

7.9.1 LPT

Longest Processing Time first (LPT), intended to solve P||Cmax problem, is a strategy for LS algorithm in which the tasks are arranged in order of non increasing processing time $\mathtt{p}_{\mathrm{j}}$ before the application of List Scheduling algorithm. The time complexity of LPT is $O(n \cdot log(n))$. The absolute performance ratio of LPT for problem P||Cmax is $R_{LPT} = 4/3 - 1/(3 \cdot m)$ [Błażewicz01]

LPT is implemented as optional parameter of List Scheduling algorithm and it is able to solve R|prec| Cmax or any easier problem.

```
RS = listsch(T,problem,processors,'LPT')
```

LS algorithm with LPT strategy demonstrated on the example from previous paragraph is shown in Figure 7.17. The resulting schedule with $\mathtt{C}_{\mathrm{max}}$`= 7` is in Figure 7.18.

Figure 7.15 Scheduling problem P|prec|Cmax solving.

```
>> t1=task('t1',2);
>> t2=task('t2',3);
>> t3=task('t3',1);
>> t4=task('t4',2);
>> t5=task('t5',4);

>> prec = [0 0 0 0 0;...
           1 0 0 0 0;...
           0 0 0 0 0;...
           0 0 1 0 0;...
           0 0 1 0 0];

>> T = taskset([t1 t2 t3 t4 t5],prec);
>> p = problem('P|prec|Cmax');
>> TS = listsch(T,p,2);
>> plot(TS);
```

Figure 7.16 Result of List Scheduling.

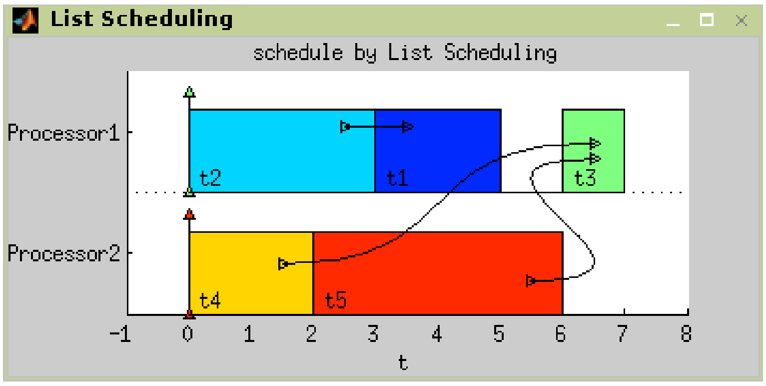


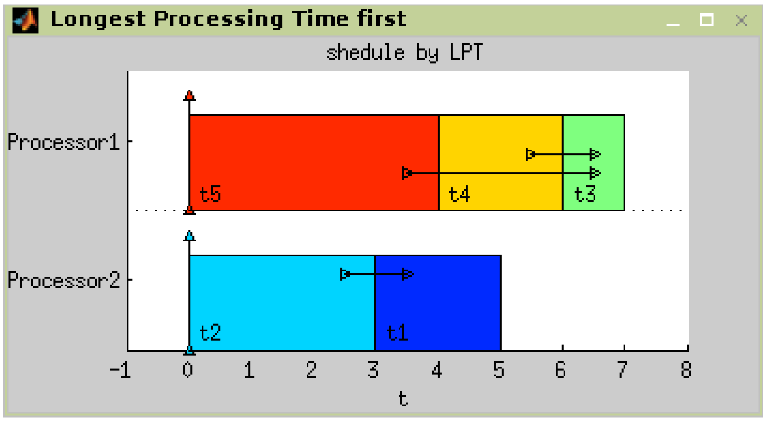
Figure 7.17 Problem P|prec|Cmax by LS algorithm with LPT strategy solving.

```
>> t1=task('t1',2);
>> t2=task('t2',3);
>> t3=task('t3',1);
>> t4=task('t4',2);
>> t5=task('t5',4);

>> prec = [0 0 0 0 0;...
           1 0 0 0 0;...
           0 0 0 0 0;...
           0 0 1 0 0;...
           0 0 1 0 0];

>> T = taskset([t1 t2 t3 t4 t5],prec);
>> p = problem('P|prec|Cmax');
>> TS = listsch(T,p,2,'LPT');
>> plot(TS);
```

7.9.2 SPT

Shortest Processing Time first (SPT), intended to solve P||Cmax problem, is a strategy for LS algorithm in which the tasks are arranged in order of nondecreasing processing time $\mathrm{p_j}$ before the application of List Scheduling algorithm. The time complexity of SPT is also $O(n \cdot log(n))$ [Błażewicz01]

Figure 7.18 Result of LS algorithm with LPT strategy.



SPT is implemented as optional parameter of List Scheduling algorithm and it is able to solve R|prec| Cmax or any easier problem .

```
TS = listsch(T,problem,processors,'SPT')
```

LS algorithm with SPT strategy demonstrated on the example from Figure 7.14 is shown in Figure 7.19. The resulting schedule with C_{max}= 7 is in Figure 7.20.

Figure 7.19 Solving P|prec|Cmax by LS algorithm with SPT strategy.

```
>> t1=task('t1',2);
>> t2=task('t2',3);
>> t3=task('t3',1);
>> t4=task('t4',2);
>> t5=task('t5',4);

>> prec = [0 0 0 0 0;...
           1 0 0 0 0;...
           0 0 0 0 0;...
           0 0 1 0 0;...
           0 0 1 0 0];

>> T = taskset([t1 t2 t3 t4 t5],prec);
>> p = problem('P|prec|Cmax');
>> TS = listsch(T,p,2,'SPT');
>> plot(TS);
```

7.9.3 ECT

Earliest Completion Time first (ECT), intended to solve P||ΣCj problem, is a strategy for LS algorithm in which the tasks are arranged in order of nondecreasing completion time C_j in each iteration of List Scheduling algorithm. The time complexity of ECT is equal or better than $O\left(n^2 \cdot log(n)\right)$.

ECT is implemented as optional parameter of List Scheduling algorithm and it is able to solve R|rj, prec|ΣwjCj or any easier problem.

```
TS = listsch(T,problem,processors,'ECT')
```

An example of P|rj|ΣwjCj scheduling problem given with set of five tasks with names, processing time and release time is shown in Table 7.2. The schedule obtained by ECT strategy with $\Sigma C_j = 58$ is shown in Figure 7.24.

Figure 7.20 Result of LS algorithm with SPT strategy.

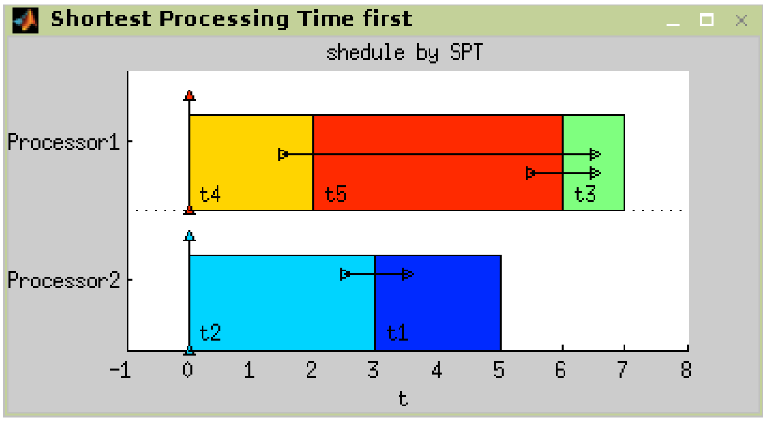


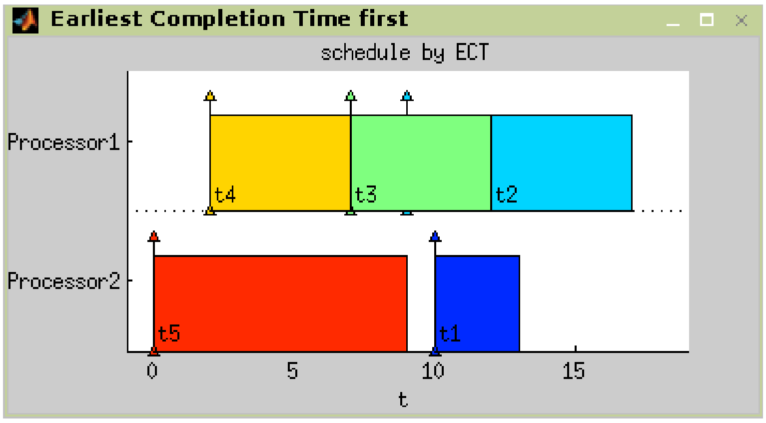
Table 7.2 An example of P|rj|ΣwjCj scheduling problem.

name	processing time	release time
t1	3	10
t2	5	9
t3	5	7
t4	5	2
t5	9	0

Figure 7.21 Solving P|rj|ΣCj by ECT

```
>> t1=task('t1',3,10);
>> t2=task('t2',5,9);
>> t3=task('t3',5,7);
>> t4=task('t4',5,2);
>> t5=task('t5',9,0);

>> T = taskset([t1 t2 t3 t4 t5]);

>> p = problem('P|rj|sumCj');
>> TS = listsch(T,p,2,'ECT');
>> plot(TS);
```

Figure 7.22 Result of LS algorithm with ECT strategy.



7.9.4 EST

Earliest Starting Time first (EST), intended to solve P||ΣCj problem, is a strategy for LS algorithm in which the tasks are arranged in order of nondecreasing starting time $\mathtt{r_j}$ before the application of List

Scheduling algorithm. The time complexity of EST is $O(n \cdot log(n))$.

EST is implemented as an optional parameter to List Scheduling algorithm and it is able to solve R|rj, prec|ΣwjCj or any easier problem.

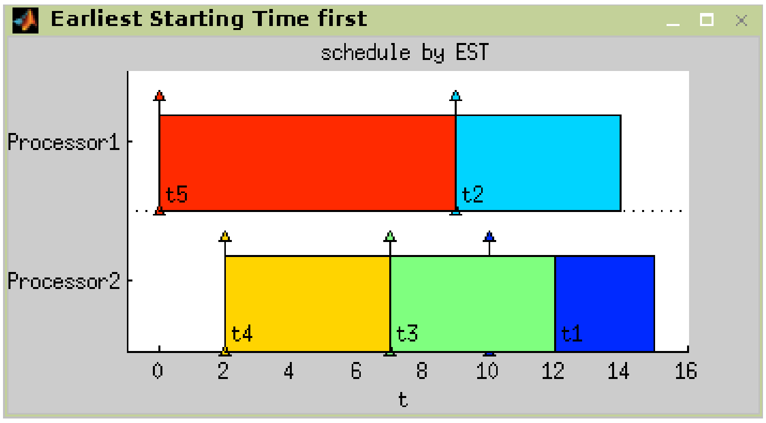
```
TS = listsch(T,problem,processors,'EST')
```

LS algorithm with EST strategy demonstrated on the example from Figure 7.14 is shown in Figure 7.23. The resulting schedule with $\Sigma C_j = 57$ is in Figure 7.24.

Figure 7.23 Problem P|rj|ΣCj by LS algorithm with EST strategy solving.

```
>> t1=task('t1',3,10);
>> t2=task('t2',5,9);
>> t3=task('t3',5,7);
>> t4=task('t4',5,2);
>> t5=task('t5',9,0);

>> T = taskset([t1 t2 t3 t4 t5]);

>> p = problem('P|rj|sumCj');
>> TS = listsch(T,p,2,'EST');
>> plot(TS);
```

Figure 7.24 Result of LS algorithm with EST strategy.



7.9.5 Own Strategy Algorithm

It's possible to define own strategy for LS algorithm according to the following model of function. Function with the same name as the optional parameter (name of strategy function) is called from List Scheduling algorithm:

```
TS = listsch(T,problem,processors,'OwnStrategy')
```

In this case, strategy algorithm is called in each iteration of List Scheduling algorithm upon the set of unscheduled task. Strategy algorithm is a standalone function with following parameters:

```
[TS, order] = OwnStrategy(T[,iteration,processor]);
```

T
set of tasks

order
index vector representing new order of tasks

iteration
actual iteration of List Scheduling algorithm

processor
selected processor

The internal structure of the function can be similar to implementation of EST strategy in private directory of scheduling toolbox.

Figure 7.25 An example of OwnStrategy function.

```
function [TS, order] = OwnStrategy(T, varargin) % head

% body
if nargin>1
    if varargin{1}>1
        order = 1:length(T.tasks);
        return
    end
end

wreltime = T.releasetime./taskset.weight;
[TS order] = sort(T,wreltime,'inc'); % sort taskset
% end of body
```

Standard variable `varargin` represents optional parameters `iteration` and `processor`. The definition of this variable is required in the head of function when it is used with `listsch`.

7.10 Brucker's Algorithm

Brucker's algorithm, proposed to solve 1|`in-tree`,p_j=1|Lmax problem, is an algorithm which can be implemented in O(n.log n) time [Bru76][Błażewicz01]. Implementation in the toolbox use listscheduling algorithm while tasks are sorted in non-increasing order of theyr modified due dates subject to precedence constraints. The algorithm returns an optimal schedule with respect to criterion L_{max}. Parameters of the function solving this scheduling problem are described in the Reference Guide brucker76.m.

Examples in Figure 7.26 and Figure 7.27 show, how an instance of the scheduling problem [Błażewicz01] can be solved by the Brucker's algorithm. For more details see `brucker76_demo` in `\scheduling\stdemos`.

Figure 7.26 Scheduling problem 1|`in-tree`,p_j=1|Lmax solving.

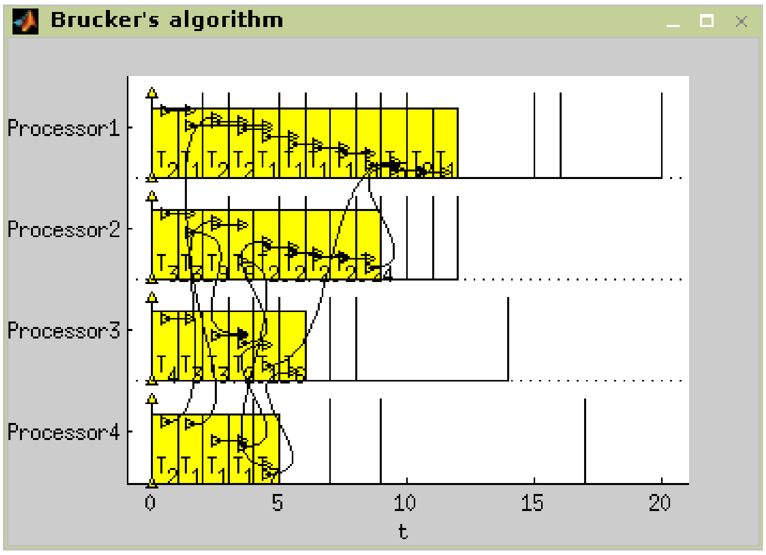
```
>> load brucker76_demo
>> T=taskset(g,'n2t',@node2task,'DueDate')
Set of 32 tasks
 There are precedence constraints
>> prob = problem('P|in-tree,pj=1|Lmax');
>> TS = brucker76(T,prob,4);
>> plot(TS);
```

7.11 Scheduling with Positive and Negative Time-Lags

Traditional scheduling algorithms (e.g., [Błażewicz01]) typically assume that deadlines are absolute. However in many real applications release date and deadline of tasks are related to the start time of another tasks [Brucker99][Hanzalek04]. This problem is in literature called scheduling with positive and negative time-lags.

The scheduling problem is given by a task-on-node graph `G`. Each task t_i is represented by node t_i in graph G and has a positive processing time p_i. Timing constraints between two nodes are represented

Figure 7.27 Brucker's algorithm - problem 1|`in-tree`,p_j=1|Lmax



by a set of directed edges. Each edge $\mathtt{e}_{\mathrm{ij}}$ from the node $\mathtt{t}_{\mathrm{i}}$ to the node $\mathtt{t}_{\mathrm{j}}$ is labeled with an integer time lag $\mathtt{w}_{\mathrm{ij}}$. There are two kinds of edges: the *forward edges* with positive time lags and the *backward edges* with negative time lags. The forward edge from the node t_i to the node $\mathtt{t}_{\mathrm{j}}$ with the positive time lag $\mathtt{w}_{\mathrm{ij}}$ indicates that $\mathtt{s}_{\mathrm{j}}$, the start time of $\mathtt{t}_{\mathrm{j}}$, must be at least $\mathtt{w}_{\mathrm{ij}}$ time units after $\mathtt{s}_{\mathrm{i}}$, the start time of $\mathtt{t}_{\mathrm{i}}$. The backward edge from node $\mathtt{t}_{\mathrm{j}}$ to node t_i with the negative time lag $\mathtt{w}_{\mathrm{ji}}$ indicates that $\mathtt{s}_{\mathrm{j}}$ must be no more than $\mathtt{w}_{\mathrm{ji}}$ time units after $\mathtt{s}_{\mathrm{i}}$. The objective is to find a schedule with minimal $\mathtt{C}_{\mathrm{max}}$.

Since the scheduling problem is NP-hard [Brucker99], algorithm implemented in the toolbox is based on *branch and bound* algorithm. Alternative implemented solution uses *Integer Linear Programming* (ILP). The algorithm call has the following syntax:

```
TS = spntl(T,problem,schoptions)
```

problem

an object of type problem describing the classification of deterministic scheduling problems (see Section Chapter 5, “Classification in Scheduling”). In this case the problem with positive and negative time lags is identified by ‘SPNTL’.

schoptions

optimization options (see Section [Scheduling Toolbox Options])

The algorithm can be chosen by the value of parameter `schoptions` - structure `schoptions` (see [Scheduling Toolbox Options]). For more details on algorithms please see [Hanzalek04].

Example 7.11.1 Example of Scheduling Problem with Positive and Negative Time-Lags.

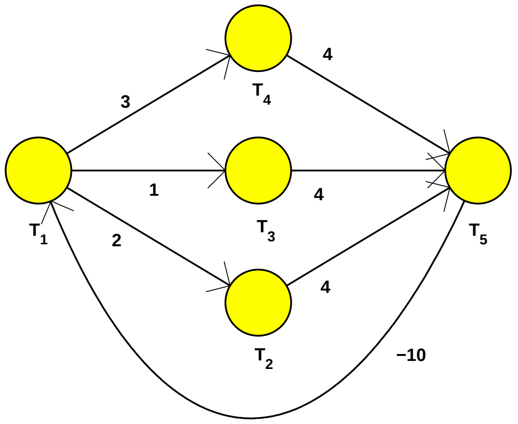
An example of the scheduling problem containing five tasks is shown in Figure 7.28 by graph `G`. Execution times are `p=(1,3,2,4,5)` and delay between start times of tasks t_1 and t_5 have to be less then or equal to 10 ($\mathtt{w}_{5,1}$`=-10`). The objective is to find a schedule with minimal $\mathtt{C}_{\mathrm{max}}$.

Solution of this scheduling problem using `spntl` function is shown below. Graph of the example can be found in Scheduling Toolbox directory `<Matlab root>\toolbox\scheduling\stdemos\benchmarks\spntl\spntl_graph.mat`. The graph `G` corresponding to the example shown in Figure 7.28 can be opened and edited in Graphedit tool (`graphedit(g)`).

Resulting graph `G` is shown in Figure 7.31. Finaly, the graph `G` is used to generate an object taskset describing the scheduling problem. Parameters conversion must be specified as parameters of function **taskset**. For example in our case, the function is called with following parameters:

```
T = taskset(LHgraph,'n2t',@node2task,'ProcTime','Processor', ...
       'e2p',@edges2param)
```

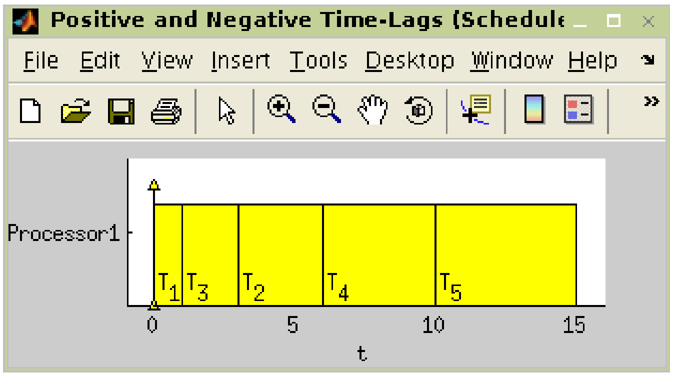
Figure 7.28 Graph `G` representing tasks constrained by positive and negative time-lags.



For more details see Section 6.5. The optimal solution in Figure 7.29 was obtained in the toolbox as is depicted below.

```
>> load <Matlab root>\toolbox\scheduling\stdemos\benchmarks\spntl_graph
>> graphedit(g)
>> T = taskset(LHgraph,'n2t',@node2task,'ProcTime','Processor', ...
      'e2p',@edges2param)
Set of 5 tasks
 There are precedence constraints
>> prob=problem('SPNTL')
SPNTL
>> schoptions=schoptionsset('spntlMethod','BaB');
>> T = spntl(T, prob, schoptions)
Set of 5 tasks
 There are precedence constraints
 There is schedule: SPNTL - BaB algorithm
>> plot(t)
```

Figure 7.29 Resulting schedule of instance in Figure 7.28.



7.12 Cyclic Scheduling

Many activities e.g. in automated manufacturing or parallel computing are cyclic operations. It means that tasks are cyclically repeated on machines. One repetition is usually called an *iteration* and common objective is to find a schedule that maximises throughput. Many scheduling techniques leads to *overlapped schedule*, where operations belonging to different iterations can execute simultaneously.

Cyclic scheduling deals with a set of operations (generic tasks t_i) that have to be performed infinitely often [Hanen95]. Data dependencies of this problem can be modeled by a directed graph `G`. Each task $\mathtt{t}_i$ is represented by the node $\mathtt{t}_i$ in the graph `G` and has a positive processing time $\mathtt{p}_i$. Edge $\mathtt{e}_{ij}$ from the node t_i to t_j is labeled by a couple of integer constants $\mathtt{l}_{ij}$ and $\mathtt{h}_{ij}$. Length $\mathtt{l}_{ij}$ represents the minimal distance in clock cycles from the start time of the task $\mathtt{t}_i$ to the start time of $\mathtt{t}_j$ and it is always greater than zero. On the other hand, the height $\mathtt{h}_{ij}$ specifies the shift of the iteration index (dependence distance) from task $\mathtt{t}_i$ to task t_j.

Assuming *periodic schedule* with *period* `w`, i.e. the constant repetition time of each task, the aim of the cyclic scheduling problem [Hanen95] is to find a periodic schedule with minimal period `w`. In modulo scheduling terminology, period `w` is called Initiation Interval (II).

The algorithm available in this version of the toolbox is based on work presented in [Hanzalek07] and [Sucha07]. Function `cycsch` solves cyclic scheduling of tasks with precedence delays on dedicated sets of parallel identical processors. The algorithm uses Integer Linear Programming

```
TS = cycsch(T,problem,m,schoptions)
```

problem
: object of type problem describing the classification of deterministic scheduling problems (see Section Chapter 5, “Classification in Scheduling”). In this case the problem is identified by ‘CSCH’.

m
: vector with number of processors in corresponding groups of processors

schoptions
: optimization options (see Section [Scheduling Toolbox Options])

In addition, the algorithm minimizes the iteration overlap [Sucha04]. This secondary objective of optimization can be disabled in parameter `schoptions`, i.e. parameter `secondaryObjective` of `schoptions` structure (see [Scheduling Toolbox Options]). The optimization option also allows to choose a method for Cyclic Scheduling algorithm, specify another ILP solver, enable/disable elimination of redundant binary decision variables and specify another ILP solver for elimination of redundant binary decision variables.

For more details on the algorithm please see [Sucha04].

Example 7.12.1 Cyclic Scheduling - Wave Digital Filter.

An example of an iterative algorithm used in Digital Signal Processing as a benchmark is Wave Digital Filter (WDF) [Fettweis86].

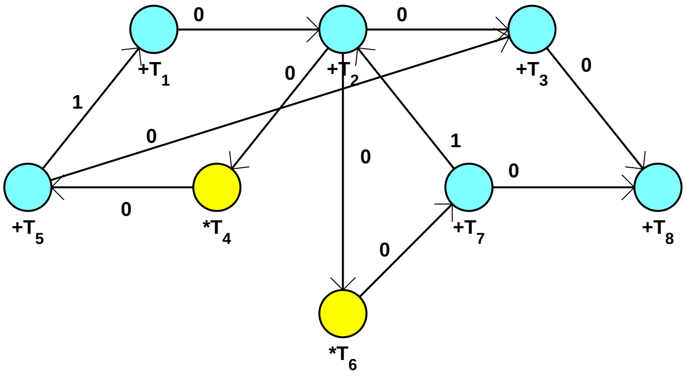
```
for k=1 to N do
   a(k) =X(k) + e(k-1)  %T1
   b(k) = a(k) - g(k-1) %T2
   c(k) = b(k) + e(k)   %T3
   d(k) = gamma1 * b(k) %T4
   e(k) = d(k) + e(k-1) %T5
   f(k) = gamma2 * b(k) %T6
   g(k) = f(k) + g(k-1) %T7
   Y(k) = c(k) - g(k)   %T8
end
```

The corresponding Cyclic Data Flow Graph is shown in Figure 7.30. Constant on nodes indicates the number of dedicated group of processors. The objective is to find a cyclic schedule with minimal period `w` on one add and one mul unit. Input-output latency of add (mul) unit is 1 (3) clock cycle(s).

To transform Cyclic Data Flow Graph (CDFG) to graph `G` weighted by $\mathtt{l}_{ij}$ and $\mathtt{h}_{ij}$ function LHgraph can be used:

```
LHgraph = cdfg2LHgraph(dfg,UnitProcTime,UnitLattency)
```

Figure 7.30 Cyclic Data Flow Graph of WDF.



LHgraph
: graph `G` weighted by $\mathtt{l}_{\mathrm{ij}}$ and $\mathtt{h}_{\mathrm{ij}}$

dfg
: Data Flow Graph where user parameter (UserParam) on nodes represents dedicated processor and user parameter (UserParam) on edges correspond to dependence distance - height of the edge.

UnitProcTime
: vector of processing time of tasks on dedicated processors

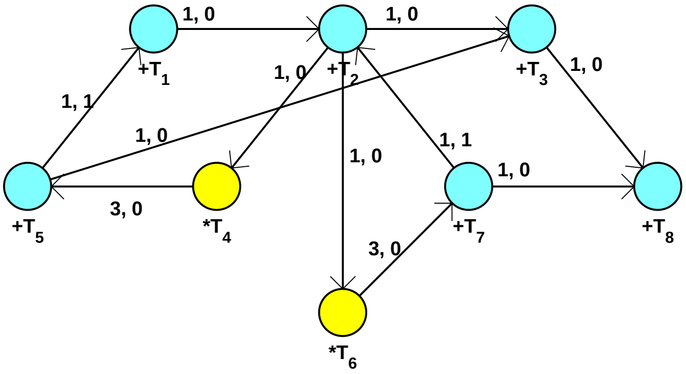
UnitLattency
: vector of input-output latency of dedicated processors

Resulting graph `G` is shown in Figure 7.31. Finaly, the graph `G` is used to generate an object taskset describing the scheduling problem. Parameters conversion must be specified as parameters of function **taskset**. For example in our case, the function is called with following parameters:

```
T = taskset(LHgraph,'n2t',@node2task,'ProcTime','Processor', ...
       'e2p',@edges2param)
```

For more details see Section 6.5.

Figure 7.31 Graph `G` weighted by $\mathtt{l}_{\mathrm{ij}}$ and $\mathtt{h}_{\mathrm{ij}}$ of WDF.

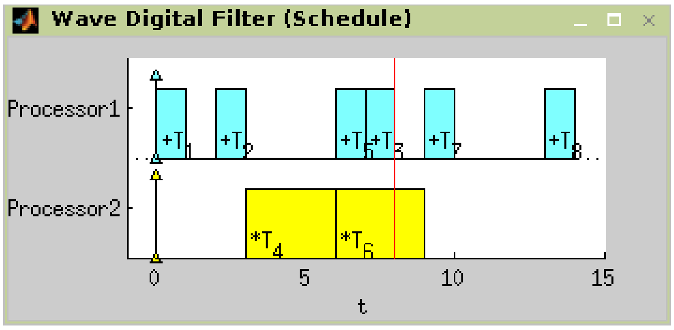


The scheduling procedure (shown below) found schedule depicted in Figure 7.32.

```
>> load <Matlab root>\toolbox\scheduling\stdemos\benchmarks\dsp\wdf
>> graphedit(wdf)
>> UnitProcTime = [1 3];
>> UnitLattency = [1 3];
>> m = [1 1];
>> LHgraph = cdfg2LHgraph(wdf,UnitProcTime,UnitLattency)

 adjacency matrix:
     0     1     0     0     0     0     0     0
     0     0     1     0     0     0     1     1
     0     0     0     1     0     0     0     0
     0     0     0     0     0     0     0     0
     0     1     0     1     0     0     0     0
     1     0     1     0     0     0     0     0
     0     0     0     0     0     1     0     0
     0     0     0     0     1     0     0     0
>>
>> graphedit(LHgraph)
>> T = taskset(LHgraph,’n2t’,@node2task,’ProcTime’,’Processor’, ...
       ’e2p’,@edges2param)
Set of 8 tasks
 There are precedence constraints
>> prob = problem(’CSCH’);
>> schoptions = schoptionsset(’ilpSolver’,’glpk’, ...
’cycSchMethod’,’integer’,’varElim’,1);
>> TS = cycsch(T, prob, m, schoptions)
Set of 8 tasks
 There are precedence constraints
 There is schedule: General cyclic scheduling algorithm (method:integer)
   Tasks period: 8
   Solving time: 1.113s
   Number of iterations: 4
>> plot(TS,’prec’,0)
```

Graph of WDF benchmark [Fettweis86] can be found in Scheduling Toolbox directory `<Matlab root>\toolbox\scheduling\stdemos\benchmarks\dsp\wdf.mat`. Another available benchmarks are DCT [CDFG05], DIFFEQ [Paulin86], IRR [Rabaey91], ELLIPTIC, JAUMANN [Heemstra92], vanDongen [Dongen92] and RLS [Sucha04][Pohl05].

Figure 7.32 Resulting schedule with optimal period w=8.



7.13 SAT Scheduling

This section presents the SAT based approach to the scheduling problems. The main idea is to formulate a given scheduling problem in the form of CNF (conjunctive normal form) clauses. TORSCHE includes the SAT based algorithm for P|prec|Cmax problem.

7.13.1 Instalation

Before use you have to instal SAT solver.

1. Download the zChaff SAT solver (version: 2004.11.15) from the zChaff web site. <http://www.princeton.edu/~chaff/zchaff.html>
2. Place the dowloaded file *zchaff.2004.11.15.zip* to the `<TORSCHE>\contrib` folder.
3. Be sure that you have C++ compiler set to the mex files compiling. To set C++ compiler call:

   ```
   >> mex -setup
   ```

   For Windows we tested Microsoft Visual C++ compiler, version 7 and 8. (Version 6 isn't supported.)

   For Linux use gcc compiler.
4. From Matlab workspace call m-file `make.m` in `<TORSCHE>\sat` folder.

7.13.2 Clause preparing theory

In the case of P|prec|Cmax problem, each CNF clause is a function of Boolean variables in the form x_{ijk}. If task $\mathtt{t_i}$ is started at time unit `j` on the processor `k` then $x_{ijk} = true$, otherwise $x_{ijk} = false$. For each task $\mathtt{t_i}$, where $i = 1 \dots n$, there are $S \times R$ Boolean variables, where `S` denotes the maximum number of time units and `R` denotes the total number of processors.

The Boolean variables are constrained by the three following rules (modest adaptation of [Memik02]):

1. For each task, exactly one of the $S \times R$ variables has to be equal to 1. Therefore two clauses are generated for each task $\mathtt{t_i}$. The first guarantees having at most one variable equal to 1 (true): $(\bar{x}_{i11} \vee \bar{x}_{i21}) \wedge \dots \wedge (\bar{x}_{i11} \vee \bar{x}_{iSR}) \wedge \dots \wedge (\bar{x}_{i(S-1)R} \vee \bar{x}_{iSR})$ The second guarantees having at least one variable equal to 1: $(\bar{x}_{i11} \vee \bar{x}_{i21} \vee \dots \vee \bar{x}_{i(S-1)R} \vee \bar{x}_{iSR})$
2. If there is a precedence constrains such that $\mathtt{t_u}$ is the predecessor of $\mathtt{t_v}$, then $\mathtt{t_v}$ cannot start before the execution of $\mathtt{t_u}$ is finished. Therefore, $x_{ujk} \rightarrow ((\bar{x}_{v1l} \wedge \dots \wedge \bar{x}_{vjl} \wedge \bar{x}_{v(j+1)l} \wedge \dots \wedge \bar{x}_{v(j+p_u-1)l})$ for all possible combinations of processors `k` and `l`, where $\mathtt{p_u}$ denotes the processing time of task $\mathtt{t_u}$.
3. At any time unit, there is at most one task executed on a given processor. For the couple of tasks with a precedence constrain this rule is ensured already by the clauses in the rule number 2. Otherwise the set of clauses is generated for each processor `k` and each time unit `j` for all couples $\mathtt{t_u}$, $\mathtt{t_v}$ without precedence constrains in the following form: $(x_{ujk} \rightarrow \bar{x}_{vjk}) \wedge (x_{ujk} \rightarrow \bar{x}_{v(j+1)k}) \wedge \dots \wedge (x_{ujk} \rightarrow \bar{x}_{v(j+p_u-1)k})$

In the toolbox we use a *zChaff* solver to decide whether the set of clauses is satisfiable. If it is, the schedule within `S` time units is feasible. An optimal schedule is found in iterative manner. First, the List Scheduling algorithm is used to find initial value of `S`. Then we iteratively decrement value of `S` by one and test feasibility of the solution. The iterative algorithm finishes when the solution is not feasible.

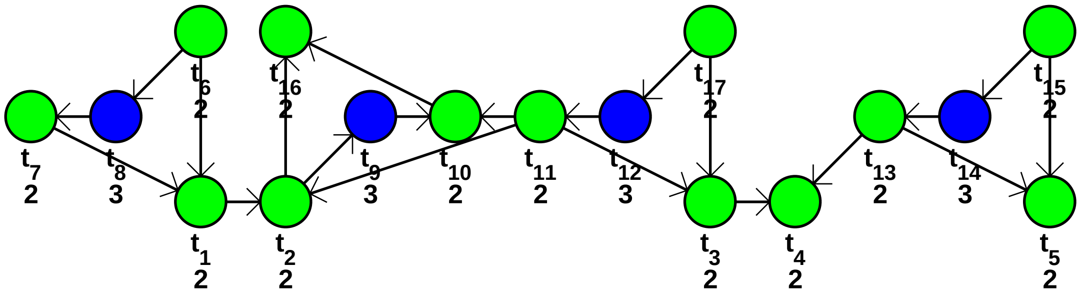
7.13.3 Example - Jaumann wave digital filter

As an example we show a computation loop of a Jaumann wave digital filter. Our goal is to minimize computation time of the filter loop, shown as directed acyclic graph in Figure 7.33. Nodes in the graph represent the tasks and the edges represent precedence constraints. Green nodes represent addition operations and blue nodes represent multiplication operations. Nodes are labeled by the processing time $\mathtt{p_i}$. We look for an optimal schedule on two parallel identical processors.

Folowing code shows consecutive steps performed within the toolbox. First, we define the set of task with precedence constrains and then we run the scheduling algorithm `satsch`. Finally we plot the Gantt chart.

```
>> procTime = [2,2,2,2,2,2,2,3,3,2,2,3,2,3,2,2,2];
>> prec = sparse(...
[6,7,1,11,11,17,3,13,13,15,8,6,2,9 ,11,12,17,14,15,2 ,10],...
[1,1,2,2 ,3 ,3 ,4,4 ,5 ,5 ,7,8,9,10,10,11,12,13,14,16,16],...
```

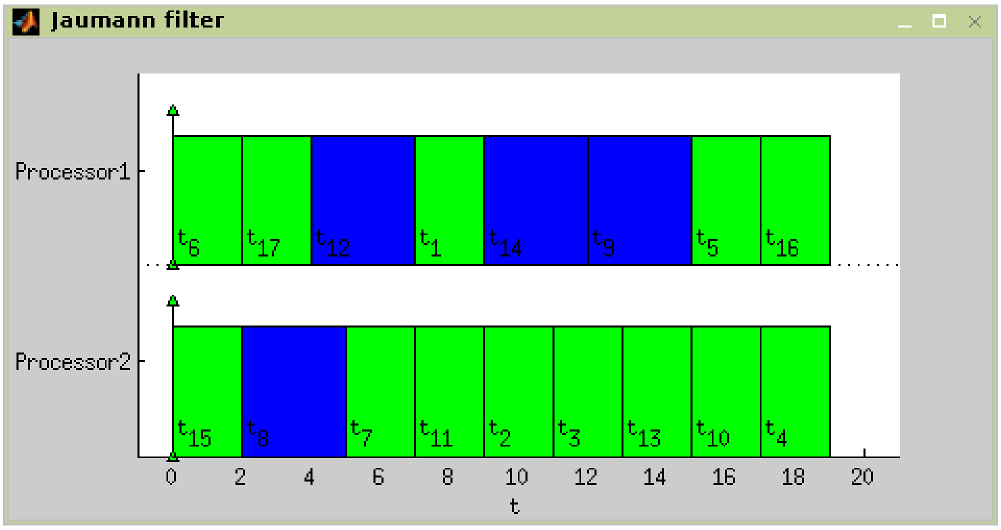
Figure 7.33 Jaumann wave digital filter



```
[1,1,1,1 ,1 ,1 ,1,1 ,1 ,1 ,1,1,1,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1],...
17,17);
>> jaumann = taskset(procTime,prec);
>> jaumannSchedule = satsch(jaumann,problem('P|prec|Cmax'),2)
Set of 17 tasks
 There are precedence constraints
 There is schedule: SAT solver
   SUM solving time: 0.06s
   MAX solving time: 0.04s
   Number of iterations: 2
>> plot(jaumannSchedule)
```

The `satsch` algorithm performed two iterations. In the first iteration 3633 clauses with 180 variables were solved as satisfiable for `S`=19 time units. In the second iteration 2610 clauses with 146 variables were solved with unsatisfiable result for `S`=18 time units. The optimal schedule is depicted in Figure 7.34.

Figure 7.34 The optimal schedule of Jaumann filter



7.14 Hu's Algorithm

Hu's algorithm is intend to schedule unit length tasks with in-tree precedence constraints. Problem notatin is P|in-tree,p_j=1|Cmax. The algorithm is based on notation of in-tree levels, where in-tree level is number of tasks on path to the root of in-tree graph. The time complexity is O(n).

```
TS = hu(T,problem,processors[,verbose])
```

or

```
TS = hu(T,problem,processors[,schoptions])
```

verbose
level of verbosity

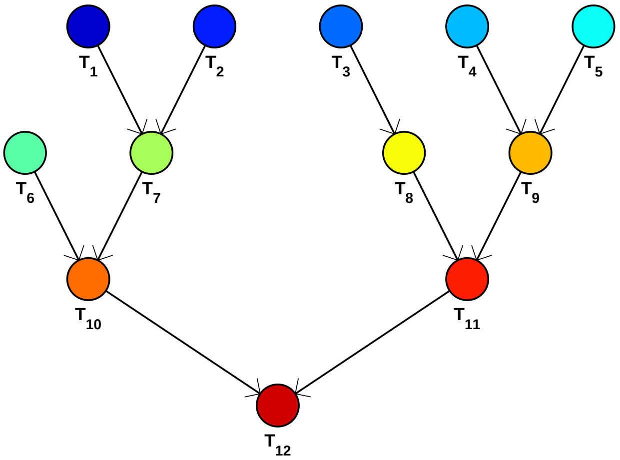
schoptions
optimization options

For more details about Hu's algorithm see [Błażewicz01].

Example 7.14.1 Hu's algorithm

There are 12 unit length tasks with precedence constraints defined as in Figure 7.35.

Figure 7.35 An example of in-tree precedence constraints



7.15 Coffman's and Graham's Algorithm

This algorithm generate optimal solution for P2|prec,p_j=1|Cmax problem. Unit length tasks are scheduled nonpreemptively on two processors with time complexity O(n^2). Each task is assigned by label, which take into account the levels and the numbers of its imediate successors. Algorithm operates in two steps:

1. Assign labels to tasks.
2. Schedule by Hu's algorithm, use labels instead of levels.

```
TS = coffmangraham(T,problem[,verbose])
```

or

```
TS = coffmangraham(T,problem[,schoptions])
```

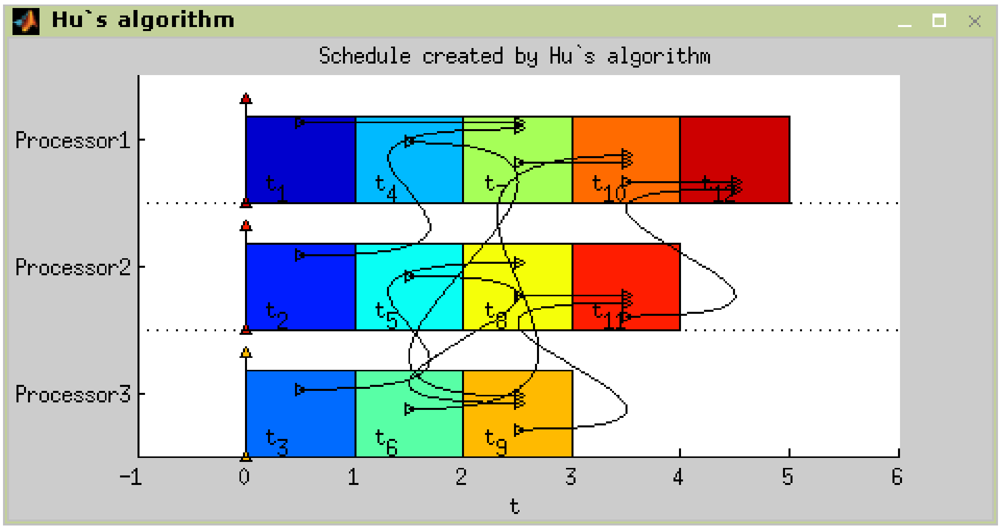
schoptions
optimization options

More about Coffman and Graham algorithm in [Błażewicz01].

Figure 7.36 Scheduling problem P|in-tree,p_j=1|Cmax using `hu` command

```
>> t1=task(’t1’,1);
>> t2=task(’t2’,1);
>> t3=task(’t3’,1);
>> t4=task(’t4’,1);
>> t5=task(’t5’,1);
>> t6=task(’t6’,1);
>> t7=task(’t7’,1);
>> t8=task(’t8’,1);
>> t9=task(’t9’,1);
>> t10=task(’t10’,1);
>> t11=task(’t11’,1);
>> t12=task(’t12’,1);

>> p = problem(’P|in-tree,pj=1|Cmax’);
>> prec = [
    0 0 0 0 0 0 1 0 0 0 0 0
    0 0 0 0 0 0 1 0 0 0 0 0
    0 0 0 0 0 0 0 1 0 0 0 0
    0 0 0 0 0 0 0 0 1 0 0 0
    0 0 0 0 0 0 0 0 1 0 0 0
    0 0 0 0 0 0 0 0 0 1 0 0
    0 0 0 0 0 0 0 0 0 1 0 0
    0 0 0 0 0 0 0 0 0 0 1 0
    0 0 0 0 0 0 0 0 0 0 1 0
    0 0 0 0 0 0 0 0 0 0 0 1
    0 0 0 0 0 0 0 0 0 0 0 1
    0 0 0 0 0 0 0 0 0 0 0 0
    ];
>> T = taskset([t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12],prec);

>> TS= hu(T,p,3);
>> plot(TS);
```

Figure 7.37 Hu’s algorithm example solution

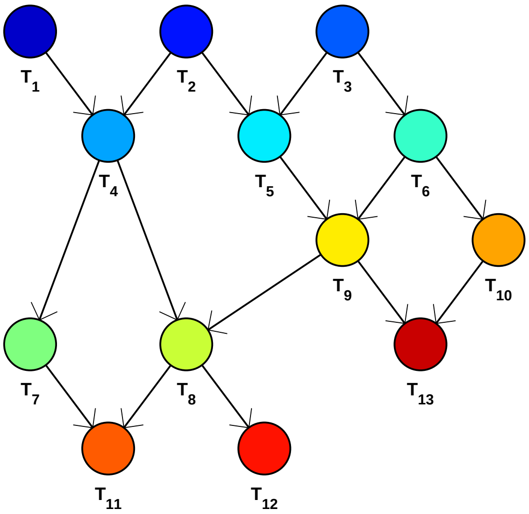


```
>> t1 = task(’t1’,1);
>> t2 = task(’t2’,1);
>> t3 = task(’t3’,1);
```

Example 7.15.1 Coffman and Graham algorithm

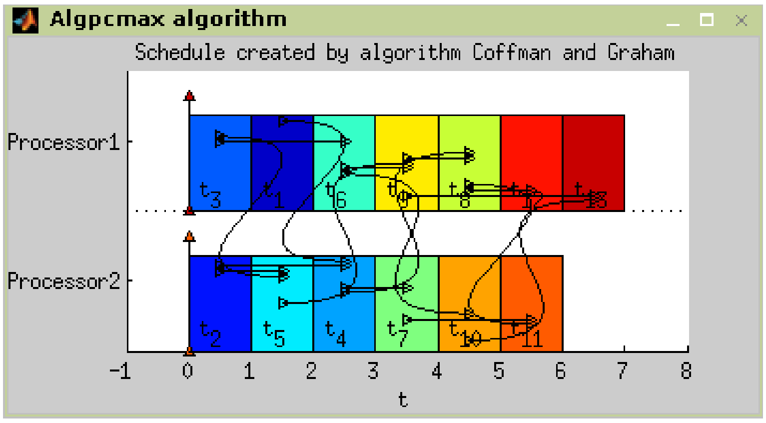
The set of tasks contains 13 tasks constrained by precedence constraints as shown in Figure 7.38.

Figure 7.38 Coffman and Graham example setting



```
>> t4 = task('t4',1);
>> t5 = task('t5',1);
>> t6 = task('t6',1);
>> t7 = task('t7',1);
>> t8 = task('t8',1);
>> t9 = task('t9',1);
>> t10 = task('t10',1);
>> t11 = task('t11',1);
>> t12 = task('t12',1);
>> t13 = task('t13',1);
>> t14 = task('t14',1);

>> p = problem('P2|prec,pj=1|Cmax');
>> prec = [
        0 0 0 1 0 0 0 0 0 0 0 0 0
        0 0 0 1 1 0 0 0 0 0 0 0 0
        0 0 0 0 1 1 0 0 0 0 0 0 0
        0 0 0 0 0 0 1 1 0 0 0 0 0
        0 0 0 0 0 0 0 0 1 0 0 0 0
        0 0 0 0 0 0 0 0 1 1 0 0 0
        0 0 0 0 0 0 0 0 0 0 1 0 0
        0 0 0 0 0 0 0 0 0 0 1 1 0
        0 0 0 0 0 0 0 1 0 0 0 0 1
        0 0 0 0 0 0 0 0 0 0 0 0 1
        0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0
    ];
>> T = taskset([t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13 t14],prec);

>> TS= coffmangraham(T,p);
>> plot(TS);
```

Figure 7.39 Coffman and Graham algorithm example solution



Chapter 8

Real-Time Scheduling

This section desribes how to use TORSCHE for analysis of real-time systems. The area of real-time scheduling is quite broad and currently only the basics are supported. We are working on addition of more advanced methods to the toolbox.

The real-time system we consider a set of periodic tasks (see Section 3.5). The sections bellow describe various algorithms that work on sets of real-time tasks.

8.1 Fixed-Priority Scheduling

Algorithms in this section assume that the tasks have assigned fixed priority (property `Weight`). The higher number, the higher priority.

8.1.1 Response-Time Analysis

The **resptime** function implements an algorithm that calculates response times for periodic tasks in a set. It is assumed, that these tasks are scheduled by a preemtive, fixed priority scheduler on one processor. Currently, this algorithm doesn't support any kind of synchronization between tasks. The syntax of the command is:

```
[resp, schedulable] = resptime(taskset)
```

where `resp` is array of response-times. There is one element for each task in the taskset. The parameter `schedulable` is non-zero if the system is schedulable, assuming the deadlines are equal to periods.

Figure 8.1 Calculating the response time using **resptime**

```
>> t1=ptask('t1',3,7);
>> t2=ptask('t2',3,12);
>> t3=ptask('t3',5,20);
>> ts=[t1 t2 t3];
>> setprio(ts, 'rm');
>> [r,s]=resptime(ts)
r =
     3    14    78
s =
     1
```

8.1.2 Fixed-Priority Scheduler

Fixed Priority Scheduling (**fps**) is an algorithm that schedules periodic tasks in taskset according to their fixed priorities (property `Weight` of task).

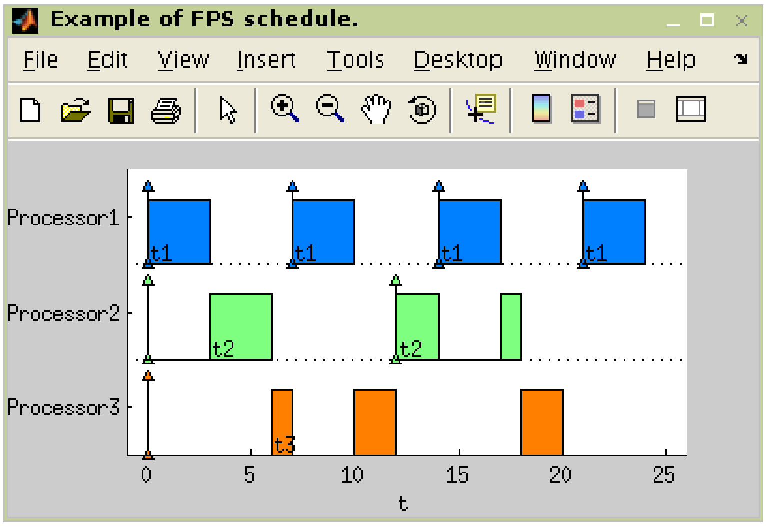
```
r = fps(TS)
```

FPS algorithm is demonstrated on the example shown in next example code. The resulting schedule is shown in the next figure.

Figure 8.2 PT_FPS example code

```
>> t1=ptask('t1',3,7);   t1.Weight = 3;
>> t2=ptask('t2',3,12);  t2.Weight = 2;
>> t3=ptask('t3',5,20);  t3.Weight = 1;
>> ts=taskset([t1 t2 t3]);
>> s=fps(ts);
>> plot(s, 'Proc', 1);
```

Figure 8.3 Result of FPS algorithm



Chapter 9

Graph Algorithms

Scheduling algorithms have very close relation to graph algorithms. Scheduling toolbox offer an object graph (see section Chapter 6, “Graphs”) with several graph algorithms.

9.1 List of Algorithms

List of algorithms related to operations with object graph are summarized in Table 9.1.

Table 9.1 List of algorithms

algorithm	command	note
Minimum spanning tree	`spanningtree`	Polynomial
Dijkstra’s algorithm	`dijkstra`	Polynomial
Floyd	`floyd`	Polynomial
Minimum Cost Flow	`mincostflow`	Using LP
Critical Circuit Ratio	`criticalcircuitratio`	Using LP
Hamilton circuit	`hamiltoncircuit`	NP-hard
Quadratic Assignment Problem	`qap`	NP-hard

9.2 Minimum Spanning Tree

Spanning tree of the graph is a subgraph which is a tree and connects all the vertices together. A minimum spanning tree is then a spanning tree with minimal sum of the edges cost. A greedy algorithm with polynomial complexity is used to solve this problem (for more details see [Demel02]). The toolbox function has following syntax:

```
gmin = spanningtree(g)
```

gmin
minimum spanning tree represented by graph

g
input graph

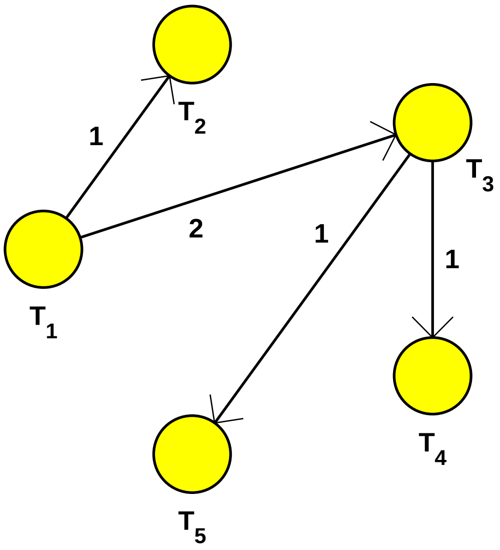
9.3 Dijkstra’s Algorithm

Dijkstra’s algorithm is an algorithm that solves the single-source cost of shortest path for a directed graph with nonnegative edge weights. Inptut of this algorithm is a directed graph with costs of invidual edges and reference node r from which we want to find shortest path to other nodes. Output is an array with distances to other nodes.

Figure 9.1 Spanning tree example

```
>> A = [inf   1  2  inf  7;...
        inf inf  3   4  inf;...
        inf  9  inf  1   1;...
         8   5  inf inf inf;...
         7  inf  4   5  inf];
>> g = graph(A);
>> gmin = spanningtree(g);
>> graphedit(gmin);
```

Figure 9.2 Example of minimum spanning tree



```
distances = dijsktra(g,r)
```

g
graph object

r
reference node

Figure 9.3 Dijkstra's algorithm example

```
>> A = [inf   1  2  inf  7;...
        inf inf  3   4  inf;...
        inf  9  inf  1   1;...
         8   5  inf inf inf;...
         7  inf  4   5  inf];
>> g = graph(A);
>> distances = dijkstra(g,2)

distances =

    11     0     3     4     4
```

9.4 Floyd's Algorithm

Floyd is a well known algorithm from the graph theory [Diestel00]. This algorithm finds a matrix of shortest paths for a given graph. Input to the algorithm is an object graph, where the weights of edges are set in `UserParam` variables of edges. Output is a matrix of shortest paths (**U**) and optionally matrix of the vertex predecessors (**P**) in the shortest path and adjacency matrix of lengths (**M**). Algorithm can be run as follows:

```
[U,P,M] = floyd(g)
```

The variable `g` is an instance of graph object.

9.5 Strongly Connected Components

The Strongly Connected Components (SCC) of a directed graph are maximal subgraphs for which hold every couple of nodes `u` and `v` there is a path from `u` to `v` and a path from `v` to `u`. For SCC searching Tarjan algorithm is usullay used.

The algorithm is based on depth-first search where the nodes are placed on a stack in the order in which they are visited. When the search returns from a subtree, it is determined whether each node is the root of a SCC. If a node is the root of a SCC, then it and all of the nodes taken off before it form that SCC. The detailed describtion of the algorithm is in [DSVF06]. SCC in a graph G can be found as follows:

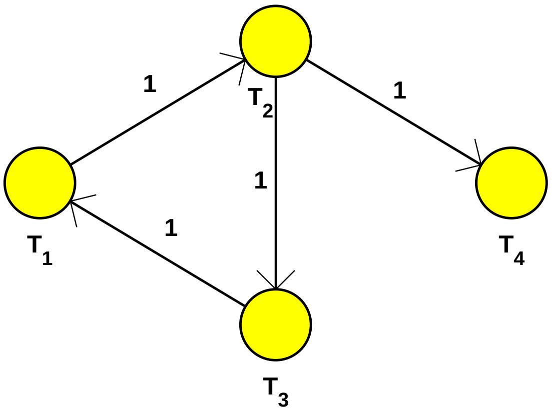
```
scc = tarjan(g)
```

where `scc` is a vector where the element `scc(i)` is number of component where the node `i` belongs to. For graph in Figure 9.5 the algorithm returns flowing results.

Figure 9.4 Strongly Connected Components example.

```
>> tarjan(g)
ans =
     2     2     2     1
```

Figure 9.5 A simple network with optimal flow in the fourth user parameter on edges



9.6 Minimum Cost Flows

The minimum cost flow model is the most fundamental of all network flow problems. In this problem we wish to determine a least cost shipment of a commodity through a network in order to satisfy demands at certain nodes from available supplies at other nodes [Ahuja93]. Let `G=(N,A)` be a directed network defined by set `N` of n nodes and a set `A` of m directed edges. Each edge `(i,j)`∈ `A` has an associated cost $\mathtt{c}_{\mathrm{ij}}$ that denotes the cost per unit flow on that arc. We also associate with each edge a *capacity* $\mathtt{u}_{\mathrm{ij}}$ that denotes the maximum amount that can flow on the arc and a *lower bound* $\mathtt{l}_{\mathrm{ij}}$ that denotes the minimum amount that must flow on the arc. We associate with each node `i`∈ `N` an integer number `b(i)` representing its suply/demand. If `b(i)`>`0`, node `i` is a *supply node*; If `b(i)`<`0`, node `i` is a *demand* of `-b(i)`; and if `b(i)=0`, node `i` is a *transshipment node*. The problem can be solved using function `mincostflow`:

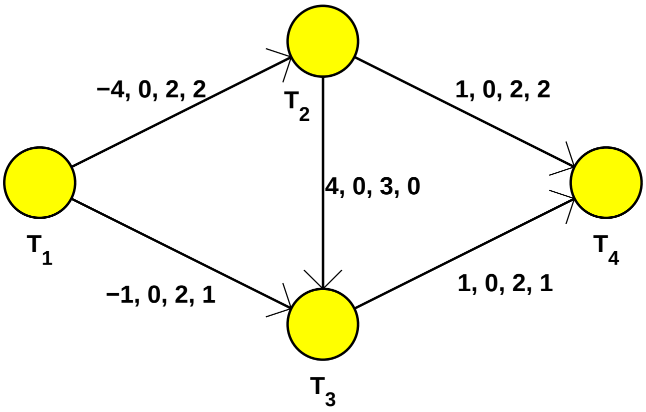
```
gminf=mincostflow(g)
```

where `g` is a graph, where suply/demand `b(i)` is stored in the first user parameter (UserParam) of nodes. Parameters ($\mathtt{c}_{\mathrm{ij}}$, $\mathtt{l}_{\mathrm{ij}}$, $\mathtt{u}_{\mathrm{ij}}$) are given in the first, second and third user parameter (UserParam) of corresponding edge $\mathtt{e}_{\mathrm{ij}}$. The function returns graph `G_minf` where the optimal flow $\mathtt{f}_{\mathrm{ij}}$ is stored in the fourth user parameter (UserParam) on edge $\mathtt{e}_{\mathrm{ij}}$. A simple example is shown in Figure 9.6 and Figure 9.7.

Figure 9.6 Mincostflow example.

```
>> gminf=mincostflow(g);
>> graphedit(gminf);
```

Figure 9.7 A simple network with optimal flow in the fourth user parameter on edges



NOTE

The algorithm use function 'ilinprog' from the TORSCHE. Solver GLPK must be installed. Installation of TORSCHE

9.7 The Critical Circuit Ratio

This problem is also called minimum cost-to-time ratio cycle problem [Ahuja93]. The algorithm assumes graph `G` where edges are weighted by a couple of constants length `l` and height `h`. The objective is to find the critical circuit ratio defined as

$$\rho = \min_{c \in C(G)} \frac{\sum\limits_{e_{ij} \in c} l_{ij}}{\sum\limits_{e_{ij} \in c} h_{ij}},$$

where `C` is a cycle of graph `G`. The circuit `C` with maximal circuit ratio is called critical circuit. Function

```
rho=criticalcircuitratio(G)
```

finds minimal circuit ratio in a graph `G`, where length `l` and height `h` are specified in the first and the second user parameters on edges (UserParam). Graph weighted by a couple `l, h` can be created from matrices `L` and `H` as shown in Example [Critical circuit ratio] where element `L(i,j), H(i,j)` contains length, height of edge `e(i,j)` respectively.

Figure 9.8 Critical circuit ratio.

```
>> L=[inf 2 inf;2 inf 1; 1 inf inf]
L =
   Inf     2   Inf
     2   Inf     1
     1   Inf   Inf
>> H=[inf 0 inf;1 inf 0;2 inf inf]
H =
   Inf     0   Inf
     1   Inf     0
     2   Inf   Inf
>> G=graph((L~=inf)*1)

 adjacency matrix:
     0     1     0
     1     0     1
     1     0     0
>> G=matrixparam2edges(G,L,1);
>> G=matrixparam2edges(G,H,2);
>> rho=criticalcircuitratio(G)
rho =
    4.0000
```

9.8 Hamilton Circuits

A Hamilton circuit in a graph `G`, is a graph cycle through `G` that visits each node exactly once. The general problem of finding a Hamilton circuit is NP-complete [Diestel00]. The solution in the toolbox is based on Integer Linear Programming.

```
gham=hamiltoncircuit(g,edgesdirection)
```

gham
hamilton circuit represented by a graph

g
input graph `G`

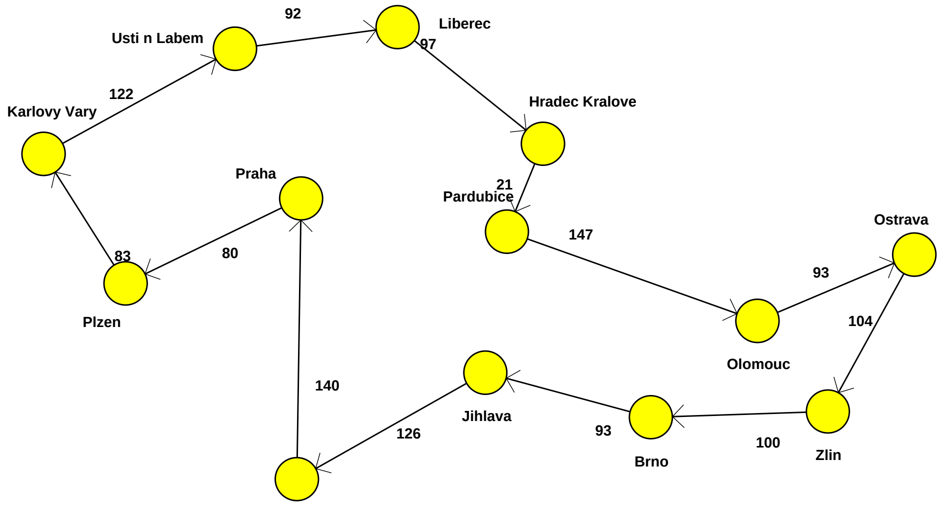
edgesdirection
specifies whether `g` is undirected ('u') or directed ('d') (directed graphs are default)

A simple example representing a traffic network in Czech Republic is shown in Example [Hamilton Circuit Identification] and Figure 9.10.

Figure 9.9 Hamilton circuit identification example.

```
>> load <Matlab root>\toolbox\scheduling\stdemos\...
   benchmarks\tsp\czech_rep
>> gham=hamiltoncircuit(g,'u');
>> graphedit(gham);
```

Figure 9.10 An example of Hamilton circuit.



NOTE

The algorithm use function 'ilinprog' from the Scheduling toolbox. Solver GLPK must be installed.Installation of TORSCHE.

9.9 Graph coloring

Graph coloring is assignment of values representing colors to nodes in a graph. Any two nodes, which are connected by an edge, cannot be assigned (colored) the same value. Graphcoloring algorithm is intended to colour graph by minimal number of colors. The least number of colors needed for coloring is called chromatic number of the graph χ. This algorithm, based on backtracking, was taken over from [Demel02]. Assigned values are of integer type saved as user parameter of each node and RGB color for nodes graphical representation.

```
G2 = graphcoloring(G1, userparamposition)
```

G1
input graph

G2
colored graph

userparamposition
specifies position (index) in userparam of node to save "color". This parameter is optional. Default index is 1.

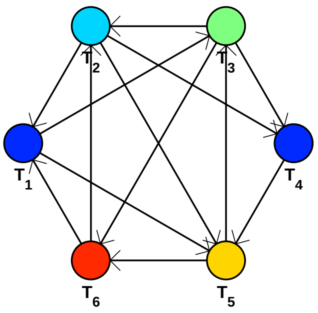
9.10 The Quadratic Assignment Problem

This algorithm solves the Quadratic Assignment Problem (QAP) [Stützle99]. The problem can be stated as follows. Consider a set of `n` activities that have to be assigned to `n` locations (or vice versa). A matrix $\mathbf{D}= [\mathtt{d}_{\mathtt{ih}}]_{\mathtt{n,n}}$ gives distances between locations, where $\mathtt{d}_{\mathtt{ih}}$ is distance between location `i` and location `h`, and a matrix $\mathbf{F} = [\mathtt{f}_{\mathtt{jk}}]_{\mathtt{n,n}}$ characterizes flows among activities (transfer of data, material, etc.), where $\mathtt{f}_{\mathtt{jk}}$ is the flow between activity `j` and activity `k`. An assignment is a permutation π of $\{1,\ldots,\mathtt{n}\}$, where

Example 9.9.1 Graph Coloring example

```
>> A = [0 0 1 0 1 0;
        1 0 0 1 1 0;
        0 1 0 1 0 1;
        0 0 0 0 1 0;
        0 0 1 0 0 1;
        1 1 0 0 0 0];
>> g1 = graph('adj',A);
>> g2 = graphcoloring(g1);
>> graphedit(g2);
```

Figure 9.11 An example of Graph coloring



π(i) is the activity that is assigned to location `i`. The problem is to find a permutation π_{m} such that the product of the flows among activities is minimized by the distances between their locations. Formally, the QAP can be formulated as the problem of finding the permutation π which minimizes the following objective function:

$C(\pi) = \sum_{i=1}^{n} \sum_{h=1}^{n} d_{ih} f_{\pi(i)\pi(h)}$

The optimal permutation π_{opt} is defined by $\pi_{opt} = \arg\min_{\pi \in \prod(n)} C(\pi)$, where $\Pi(\mathtt{n})$ is the set of all permutations of {1,...,n}.

The problem can be reformulated to show the quadratic nature of the objective function: solving the problem means identifying a permutation matrix **X** of dimension n × n (whose elements $\mathtt{x}_{\mathrm{ij}}$ are 1 if the activity `j` is assigned to location `i` and 0 in the other cases) such that:

Equation 9.10.1

$$C(\pi) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{h=1}^{n} \sum_{k=1}^{n} d_{ih} f_{jk} x_{ij} x_{hk},$$

subject to the constraints $\sum_{i=1}^{n} x_{ij} = 1, \sum_{j=1}^{n} x_{ij} = 1$and $x \in \{0,1\}$. In the toolbox the problem can be solved using Mixed Integer Quadratic Programming (MIQP).

```
[xmin,fmin,status,extra]=qap(distancesgraph,flowsgraph)
```

The function returns a nonempty output if a solution is found. Matrix `xmin` is optimal value of decision variables, `fmin` is equal to 0.5 times optimal value of the objective function, `status` is a status of the optimization (1-solution is optimal) and `extra` is a data structure containing field `time` - time (in seconds) used for solving. Parameters `distancesgraph` and `flowsgraph` are graphs, where distances and flows are specified in first user parameter on edges (`UserParam`). Graphs can be created form matrices **D** and **F** as shown in Quadratic Assignment Problem example in [Quadratic Assignment Problem]. Some benchmark instances [QAPLIB06] are located in `\scheduling\stdemos\benchmarks\qap\` directory.

Figure 9.12 Quadratic Assignment Problem.

```
>> D = [0 1 1 2 3;1 0 2 1 2;1 2 0 1 2;2 1 1 0 1;3 2 2 1 0]
D =
     0     1     1     2     3
     1     0     2     1     2
     1     2     0     1     2
     2     1     1     0     1
     3     2     2     1     0

>> F = [0 5 2 4 1;5 0 3 0 2;2 3 0 0 0;4 0 0 0 5;1 2 0 5 0]
F =
     0     5     2     4     1
     5     0     3     0     2
     2     3     0     0     0
     4     0     0     0     5
     1     2     0     5     0

%Create graph of distances
>> distancesgraph=graph(1*(D˜=0));

%Insert distances into the graph
>> distancesgraph=matrixparam2edges(distancesgraph,D,1,0);

%Create graph of flow
>> flowsgraph=graph(1*(F˜=0));

%Insert flows into the graph
>> flowsgraph=matrixparam2edges(flowsgraph,F,1,0);

>> [xmin,fmin,status,extra]=qap(distancesgraph,flowsgraph)
xmin =
     0     1     0     0     0
     0     0     0     1     0
     0     0     0     0     1
     1     0     0     0     0
     0     0     1     0     0
fmin =
    25
status =
     1
extra =
    time: 1.2660
```

Note

The algorithm use function `iquadprog` from the toolbox.

NOTE

Smaller benchmark instance (not presented in [QAPLIB06]) can be found e.g. on <http://ina2.eivd.ch/collaborateurs/etd/problemes.dir/qap.dir/qap.html>.

Chapter 10

Other Algorithms

TORSCHE is extended with several algorithms and some interfaces to external tools to facilitate develop of scheduling algorithms.

10.1 List of Algorithms

List of the supplementary algorithms is summarized in the following table.

Table 10.1 List of algorithms

algorithm	command
Scheduling Toolbox solvers settings	schoptionsset
Random Data Flow Graph (DFG) generator	randdfg
Universal interface for ILP	ilinprog
Universal interface for MIQP	iquadprog

10.2 Scheduling Toolbox Options

Lot of scheduling algorithms require extra parameters, e.g. parameters of external solvers. To create Scheduling Toolbox structure containing option parameters use

```
schoptions=schoptionsset('keyword1',value1,'keyword2',value2,...)
```

The function specifies values (V1, V2, ...) of the specific parameters (C1, C2, ...). To change particular parameters use

```
schoptions=schoptionsset(schoptions,'keyword1',value1,...)
```

Parameters of Scheduling Toolbox options are summarized in Table 10.2. Defauld values of single parameters are typed in italics.

10.3 Random Data Flow Graph (DFG) generation

This supplementary function allows to generate random Data Flow Graph (DFG). It is appointed for benchmarking of scheduling algorithms.

```
g=randdfg(n,m,degmax,ne)
```

The function generates Data Flow Graph `g`, where relation of node (task) to a dedicated processor is stored in `g.N(i).UserParam`. The first parameter `n` is the number of nodes in the DFG, `m` is the number of dedicated processors. Parameter `degmax` restricts upper bound of outdegree of vertices. Parameter `ne` is the number of edges.

```
g=randdfg(n,m,degmax,ne,neh,hmax)
```

The function with this parameters generates cyclic DFG (CDFG), where `neh` is number of edges with user parameter $0 <$ `g.E(i).UserParam` $<=$ `hmax`. Other edges has user parameter `g.E(i).UserParam=0`.

Table 10.2 List of the toolbox options parameters

parameter	meaning	value
General		
`maxIter`	Maximum number of iterations allowed.	positive integer
`verbose`	Verbosity level.	*0* = be silent, 1 = display only critical messages, 2 = display everything
`strategy`	Strategy of scheduling algorithm.	This parameter is specific for each scheduling algorithm. (e.g. `listsch` algorithm distinguishes 'EST', 'ECT', 'LPT', 'SPT' or a handler of user defined function)
`logfile`	Enables logfile creation.	*0* = disable, 1 = enable
`logfileName`	Specifies logfile name.	character array
Integer Linear Programming (ILP)		
`ilpSolver`	Specifies internal ILP solver (GLPK [Makhorin04], LP_SOLVE [Berkelaar05], CPLEX [CPLEX04], external).	'*glpk*', 'lp_solve', 'cplex', 'external'
`extIlinprog`	Specifies external ILP solver interface. Specified function must have the same parameters as function `linprog`.	function handle
`miqpSolver`	Specifies internal MIQP solver (MIQP [Bemporad04], CPLEX [CPLEX04], external).	'*miqp*', 'cplex', 'external'
`extIquadprog`	Specifies external MIQP solver interface. Specified function must have the same parameters as function `linprog`.	function handle
`solverVerbosity`	Verbosity level of ILP solver.	*0* = be silent, 1 = display only critical messages, 2 = display everything
`solverTiLim`	Sets the maximum time, in seconds, for a call to an optimizer. When solverTiLim<=0, the time limit is ignored. Default value is 0.	double
Cyclic Scheduling		
`cycSchMethod`	Specifies method for Cyclic Scheduling algorithm.	'*integer*', 'binary'
`varElim`	Enables elimination of redundant binary decision variables in ILP model.	0 = disable, *1* = enable
`varElimILPSolver`	Specifies another ILP solver for elimination of redundant binary decision variables.	'*glpk*', 'lp_solve', 'cplex', 'external'
`secondaryObjective`	Enables minimization of iteration overlap as secondary objective.	0 = disable, *1* = enable
Scheduling with Positive and Negative Time-lags		
`spntlMethod`	Specifies an method for `spntl` algorithm.	'BaB' - Branch and Bound, '*ILP*' - Integer Linear Programming

10.4 Universal interface for ILP

Universal interface for Integer Linear Programming (ILP) allows to call different ILP solvers from Matlab.

```
[xmin,fmin,status,extra] = ...
   ilinprog(schoptions,sense,c,A,b,ctype,lb,ub,vartype)
```

Function `ilinprog` has the following parameters. Parameter `schoptions` is Scheduling Toolbox Options structure (see [Scheduling Toolbox Options]). The next parameter `sense` indicates whether the problem is a minimization=1 or a maximization=-1. ILP model is specified with column vector `c` containing the objective function coefficients, matrix `A` representing linear constraints and column vector `b` of right sides for the inequality constraints. Column vector `ctype` determines the sense of the inequalities as is shown in Table 10.3.

Table 10.3 Type of constraints - ctype.

ctype(i)	constraint
'L'	'<='
'E'	'='
'G'	'>='

Further, column vector `lb` (`ub`) contains lower (upper) bounds of variables in specified ILP model. The last parameter is column vector `vartype` containing the types of the variables (`vartype(i)` = 'C' indicates continuous variable and `vartype(i)` = 'I' indicates integer variable).

A nonempty output is returned if a solution is found. Afterwards `xmin` contains optimal values of variables. Scalar `fmin` is optimal value of the objective function. Value `status` indicates the status of the optimization (1-solution is optimal) and structure `extra` consisting of fields `time` and `lambda`. Field `time` contains time (in seconds) used for solving and field `lambda` contains solution of the dual problem.

10.5 Universal interface for MIQP

Universal interface for Mixed Integer Quadratic Programming (MIQP) allows to call different MIQP solvers from Matlab. This function is very similar to function 'ilinprog', described in section Section 10.4.

```
[xmin,fmin,status,extra] = ...
   iquadprog(schoptions,sense,H,c,A,b,ctype,lb,ub,vartype)
```

Function `iquadprog` has the following parameters. Parameter `schoptions` is Scheduling Toolbox Options structure (see [Scheduling Toolbox Options]). The next parameter `sense` indicates whether the problem is a minimization=1 or maximization=-1. ILP model is specified with column vector `c` and square matrix `H` containing the objective function coefficients, matrix `A` representing linear constraints and column vector `b` of right sides for the inequality constraints. Column vector `ctype` determines the sense of the inequalities as is shown in Table 10.4.

Table 10.4 Type of constraints - ctype.

ctype(i)	constraint
'L'	'<='
'E'	'='
'G'	'>='

Further, column vector `lb` (`ub`) contains lower (upper) bounds of variables in specified MIQP model. The last parameter is column vector `vartype` containing the types of the variables (`vartype(i)` = 'C' indicates continuous variable and `vartype(i)` = 'I' indicates integer variable).

A nonempty output is returned if a solution is found. Afterwards `xmin` contains optimal values of variables. Scalar `fmin` is optimal value of the objective function. Value `status` indicates the status of the optimization (1-solution is optimal) and structure `extra` consisting of fields `time` and `lambda` contains time (in seconds) used for solving and optimal values of dual variables respectively.

WARNING

In some versions of Matlab, solver `miqp` [Bemporad04] (see Section [Scheduling Toolbox Options]) should not find optimal solution in spite of it exists or can find a solution with worst value of objective function. When you have a problem with the solver, please read the `miqp` solver documentation.

NOTE

The algorithm requires Optimization Toolbox for Matlab (<`http://www.mathworks.com/`>).

10.6 Cyclic Scheduling Simulator

CSSIM (Cyclic Scheduling Simulator) is a tool allowing to simulate iterative loops in Matlab Simulink using TrueTime tool [Cervin06]. The loop described in a language compatible with Matlab can be transformed into the toolbox structures. In the toolbox the input iterative loop is scheduled using cyclic scheduling (see [Cyclic scheduling (General)]) and optimized iterative algorithm is transformed into TrueTime code for real-time simulation. The `CSSIM` operates in three steps: input file parsing (function `cssimin`), cyclic scheduling (see [Cyclic scheduling (General)]) and True-Time code generation (function `cssimout`).

```
[T,m]=cssimin('dsvf.m');                    %input file parsing

TS=cycsch(T, problem('CSCH'), m, schoptions); %cyclic scheduling

cssimout(TS,'simple_init.m','code.m');       %TrueTime code generation
```

Input parametr of function `cssimin` is a string specifying input file to be parsed. The function returns a taskset representing the input algorithm. Extra information about algorithm structure are available in `CodeGenerationData` structure contained in `TSUserParam` of the taskset `T`. Further, the function returns the number of processors, contained in vector `m`. Function `cssimout` generates two files, which are used for simulation in TrueTime. The first input parameter specifies a taskset generated using function `cssimin`, extended with cyclic schedule. The second parameter specifies name of output TrueTime initialization file. The third one specifies name of output TrueTime code of simulated loop.

The language structure of input files is described in the following subsection.

10.6.1 CSSIM Input File

The input file for CSSIM is compatible with Matlab language but it has a simpler and fixed structure. The file is divided into four parts with fixed order: *Processors Declaration*, *Variables Initialization*, *Iterative Algorithm* and *Subfunctions*.

- *Processors Declaration*: Processors are declared as a structure with fields describing processor parameters. First one is *`operator`* assigning an operator in the iterative loop to specific processor. Second one is *`number`* representing number of processors. Following two fields (*`proctime`* and *`latency`*) represent timing parameters of the unit (see section [Cyclic scheduling (General)]).

  Example:

  ```
  struct('operator','+','number',1,'proctime',2,'latency',2);
  struct('operator','ifmin','number',1,'proctime',1,'latency',1);
  ```

  In addition, the simulation frequency for TrueTime can be defined as a structure:

  Example:

```
struct('frequency',10000);
```

- *Variables Initialization*: The aim of this part is to initialize variables and define input and output variable.

  Example:

  ```
  K = 10;
  s2{1} = zeros;
  y = num2cell([1 1 1 1 1 1 1 1 1 1]);

  struct('output',{'u'},'input','e');
  ```

- *Iterative Algorithm*: The input iterative algorithm is described as a loop (for ... end) containing elementary operations (tasks). Each operation is constituted by one line of the loop and can contain only one operation (addition, multiplication, ...).

  Example:

  ```
  for k=2:K-1
      ke(k) = e(k) * Kp;
      s2(k) = c1 + ke(k);
      s3(k) = ifmax(s2(k),umax);
      u(k) = ifmin(s3(k),umin);
  end
  ```

- *Subfunctions*: The last part contains subfunction called from the iterative algorithm. It usually corresponds to units declared in *Arithmetic Units Declaration* part. In current version of CSSIM it is used only for simulation of elementary operations. In further version the subfunctions will be also object of the optimization.

  Example:

  ```
  function y=ifmin(a1,a2)
      if(a1<a2)
          y=a2;
      else
          y=a1;
      end
  return
  ```

NOTE

Work on this function is still in progress. The authors will be appreciative of any comment and bug report.

10.6.2 TrueTime

For time-exact simulation of schedules of iterative loops TrueTime is used. The scheduled algorithm is simulated using TrueTime Kernels. The CSSIM generates two M-files. First one is an initialization code (simple_init.m) which creates a task for the simulation and initialize variables used in the scheduled algorithm. The second one is a code function (code.m) where each code segment corresponds to a task from the schedule. Lets consider an example of digital state variable filter [DSVF06], shown as CSSIM code below

```
function L=dsvf(I)

%Arithmetic Units Declaration
struct('operator','+','number',1,'proctime',1,'latency',1);
struct('operator','*','number',1,'proctime',3,'latency',3);

struct('frequency',220000);

%Variables Declaration
f = 50;
fs = 40000;
Q = 2;
K = 1000;

F1 = 0.0079;   %2*pi*f/fs  ||| ??? 2*sin(pi*f/fs)
Q1 = 0.5;      %1/Q;


%I = num2cell(simout);
I = ones(1,K);

L = zeros(1,K);
B = zeros(1,K);
H = zeros(1,K);
N = zeros(1,K);
FB = zeros(1,K);
QB = zeros(1,K);
IL = zeros(1,K);
FH = zeros(1,K);


%Iterative Algorithm
for k=2:K
    FB(k) = F1 * B(k-1);
    L(k)  = L(k-1) + FB(k);     %L = L + F1 * B
    QB(k) = Q1 * B(k-1);
    IL(k) = I(k) - L(k);
    H(k)  = IL(k) - QB(k);      %H = I - L -Q1*B
    FH(k) = F1 * H(k);
    B(k)  = FH(k) + B(k-1);     %B = F1 * H +B
    N(k)  = H(k) + L(k);        %N = H + L
end


L
```

After processing in CSSIM, as shown above, generated files (simple_init.m and code.m shown bellow) are used in simulation scheme shown in Figure 10.1. Parameter `Name of init function` is set to *simple_init* (initialization code).

```
function simple_init
%
% This file was automatically generated by CSSIM
%

ttInitKernel(1,1,'prioFP');% nbrOfInputs, nbrOfOutputs, fixed priority

data.frequency=220000;    %simulation frequency
data.reg1=0;    % initialization of variable B
```

```
data.reg2=0;    % initialization of variable H
data.reg3=0;    % initialization of variable N
data.reg4=0;    % initialization of variable FB
data.reg5=0;    % initialization of variable QB
data.reg6=0;    % initialization of variable IL
data.reg7=0;    % initialization of variable FH
data.reg8=0;    % initialization of variable L
data.const1=50;    % initialization of constant f
data.const2=40000;    % initialization of constant fs
data.const3=2;    % initialization of constant Q
data.const4=1000;    % initialization of constant K
data.const5=0.007900000000000001;    % initialization of constant F1
data.const6=0.5;    % initialization of constant Q1
w=11;
period = w/data.frequency;
deadline = period;
offset = 0;
prio = 1;
ttCreatePeriodictask('task1', offset, period, prio, 'code', data);
```

```
function [exectime,data] = code(seg,data)
%
% This file was automatically generated by CSSIM
%

 i=floor(ttCurrentTime/ttGetPeriod);

switch(seg)
case 1
data.reg4 = data.const5*data.reg1;     % T1
exectime = 3/data.frequency;
case 2
data.reg8 = data.reg8+data.reg4;
ttAnalogOut(1,data.reg8);              % T2
exectime = 0/data.frequency;
case 3
data.reg5 = data.const6*data.reg1;     % T3
exectime = 1/data.frequency;
case 4
data.reg6 = ttAnalogIn(1)-data.reg8;   % T4
exectime = 2/data.frequency;
case 5
data.reg2 = data.reg6-data.reg5;       % T5
exectime = 1/data.frequency;
case 6
data.reg7 = data.const5*data.reg2;     % T6
exectime = 0/data.frequency;
case 7
data.reg3 = data.reg2+data.reg8;       % T8
exectime = 3/data.frequency;
case 8
data.reg1 = data.reg7+data.reg1;       % T7
exectime = 1/data.frequency;
case 9
exectime = -1;
end
```

Result of the simulation is shown in Figure 10.2.

For more details see CSSIM TrueTime demo in the toolbox. For the simulation the TrueTime tool must be installed.

Figure 10.1 Simulation scheme with TrueTime Kernel block

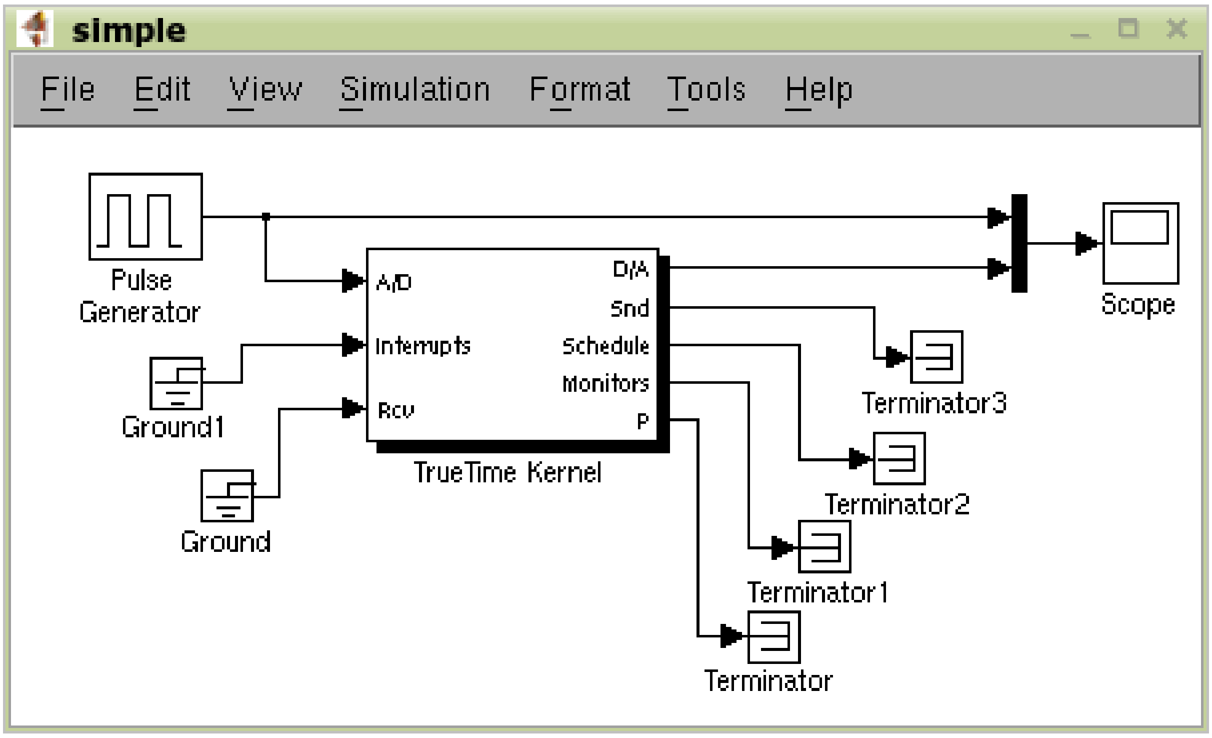
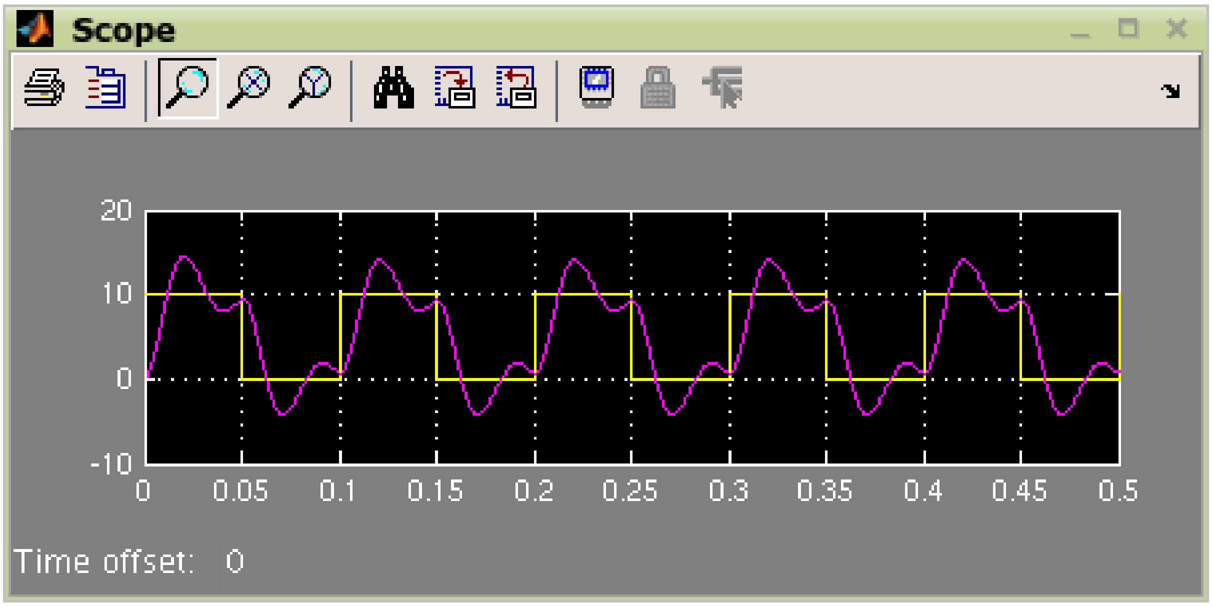


Figure 10.2 Result of simulation



10.7 Export to XML

All main objects in TORSCHE can be exported to the XML file format. XML format includes all important information about one or more objects.

XMLSAVE command is used for export to XML for more details see to xmlsave.m.

Chapter 11

Case Studies

This chapter presents some case studies fully solvable in the toolbox. Case studies describes the develop process step by step.

11.1 Theoretical Case Studies

This section shows mostly theoretical examples and their solution in the toolbox.

11.1.1 Watchmaker's

Imagine that you are a man, who repairs watches. Your boss gave you a list of repairs, which have to be done tomorrow. Our goal is to decide, how to organize the work to meet all desired finish times if possible.

List of repairs:

- Watch number 1 needs a battery replacement, which has to be finisched at 14:00.
- Watch number 2 is missing hand and has to be ready at 12:00.
- Watch number 3 has broken clockwork and has to be fixed till 16:00.
- The seal ring has to be replaced on watch number 4. It is necessary to reseal it before 15:00.
- Watch number 5 has bad battery and broken light. It has to be returned to the customer before 13:00.

Batteries will be delivered to you at 9:00, seal ring at 11:00 and the hand for watch number 2 at 10:00.

Time of repairs:

- Battery replacement: 1 hour
- Replace missing hand: 2 hours
- Fix the clockwork: 2 hours
- Seal the case: 1 hour
- Repair of light: 1 hour

Let's look at the list more closely and try to extract and store the information we need to organize the work (see Table 11.1). We consider working day starts at 8:00.

Table 11.1 Information we need to organize the work

	t1	t2	t3	t4	t5
p_j	1	2	2	1	2
r_j	9	10	8	11	9
d_j	14	12	16	15	13

Solution of the case study is shown in five steps:

1. With formalized information about the work, we can define the tasks.

```
>> t1=task('Watch1',1,9,inf,14);
>> t2=task('Watch2',2,10,inf,12);
>> t3=task('Watch3',2,8,inf,16);
>> t4=task('Watch4',1,11,inf,15);
>> t5=task('Watch5',2,9,inf,13);
```

2. To handle our tasks together, we put them into one object, taskset.

```
>> T=taskset([t1 t2 t3 t4 t5]);
```

3. Our goal is to assign the tasks on one processor, in order, which meets the required duedates of all tasks if possible. The work on each task can be paused at any time, and we can finish it later. In other words, preemption of tasks is allowed. In Graham-Blaziewicz notation the problem can be described like this.

```
>> prob=problem('1|pmtn,rj|Lmax');
```

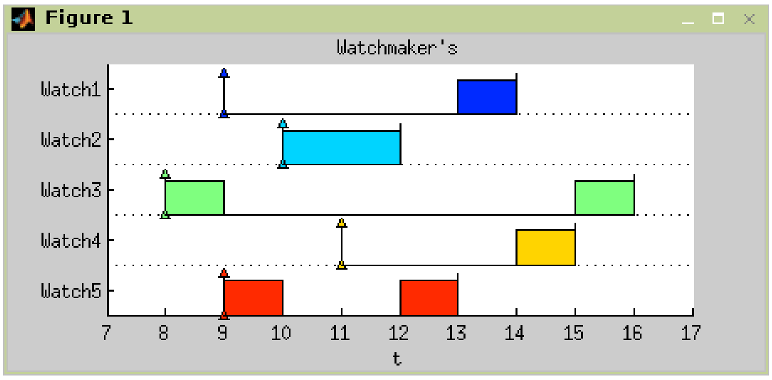
4. The algorithm is used as a function with 2 parameters. The first one is the taskset we have defined above, the second one is the type of problem written in Graham-Blaziewitz notation.

```
>> TS=horn(T,prob)
Set of 5 tasks
 There is schedule: Horn's algorithm
   Solving time: 0.046875s
```

5. Visualize the final schedule by standard `plot` function, see Figure 11.1.

```
>> plot(TS,'proc',0);
```

Figure 11.1 Result of case study as Gantt chart



11.1.2 Conveyor Belts

Transportation of goods by two conveyor belts is a simple example of using List Scheduling in practice. Construction material must be carried out from place to place with minimal time effort. Transported articles represent five kinds of construction material and two conveyor belts as processors are available. Table 11.2 shows the assignment of this problem.

Solution of the case study is shown in five steps:

1. Create a `taskset` directly through the vector of processing time.

```
>> T = taskset([40 50 30 50 20]);
```

2. Since the `taskset` has been created, it is possible to change parameters of all tasks in it.

```
>> T.Name = {'sand','grit','wood','bricks','cement'};
```

Table 11.2 Material transport processing time.

name	processing time
sand	40
grit	50
wood	30
brickc	50
cement	20

3. Define the problem, which will be solved.

```
>> p = problem('P|prec|Cmax');
```

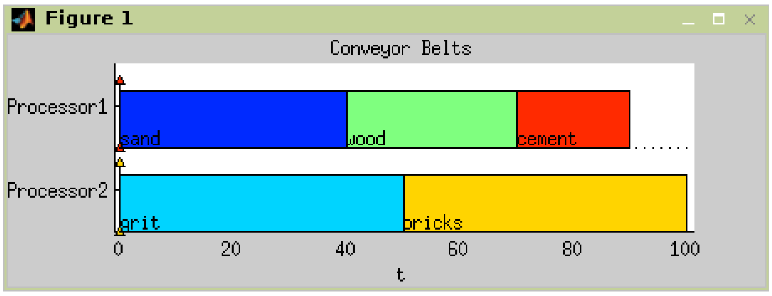
4. Call List Scheduling algorithm with `taskset` and the `problem` created recently and define the number of processors (conveyor belts).

```
>> TS = listsch(T,p,2)
Set of 5 tasks
 There is schedule: List Scheduling
```

5. Visualize the final schedule by standard `plot` function, see Figure 11.2.

```
>> plot(TS)
```

Figure 11.2 Result of case study as Gantt chart



11.1.3 Chair manufacturing

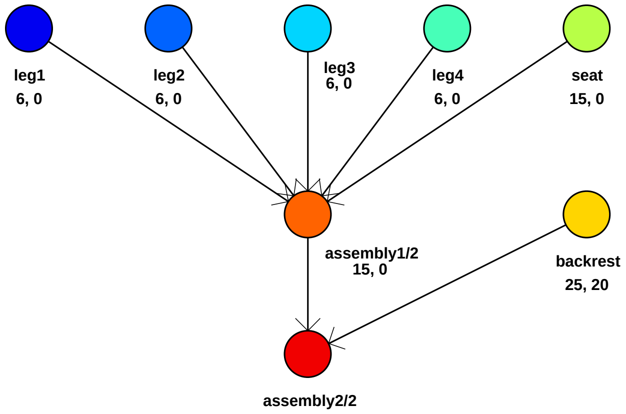
This example is slightly more difficult and demonstrates some of advanced possibilities of the toolbox. It solves a problem of chair manufacturing by two workers (cabinetmakers). Their goal is to make four legs, seat and backrest of the chair and assembly all of these parts with minimal time effort. Material, which is needed to create backrest, will be available after 20 time units of start and assemblage is divided out into two stages. Figure 11.3 shows the mentioned problem by graph representation.

Solution of the case study is shown in six steps:

1. Create desired tasks.

```
>> t1 = task('leg1',6)
 Task "leg1"
  Processing time: 6
  Release time: 0

>> t2 = task('leg2',6);
>> t3 = task('leg3',6);
>> t4 = task('leg4',6);
>> t5 = task('seat',6);
>> t6 = task('backrest',25,20);
>> t7 = task('assembly1/2',15);
>> t8 = task('assembly2/2',15);
```

Figure 11.3 Graph representation of Chair manufacturing



2. Define precedence constraints by precedence matrix `prec`. Matrix has size `n` x `n` where `n` is a number of tasks.

```
>> prec = [0 0 0 0 0 0 1 0;...
           0 0 0 0 0 0 1 0;...
           0 0 0 0 0 0 1 0;...
           0 0 0 0 0 0 1 0;...
           0 0 0 0 0 0 1 0;...
           0 0 0 0 0 0 0 1;...
           0 0 0 0 0 0 0 1;...
           0 0 0 0 0 0 0 0];
```

3. Create an object of taskset from recently defined objects.

```
>> T = taskset([t1 t2 t3 t4 t5 t6 t7 t8],prec)
 Set of 8 tasks
  There are precedence constraints
```

4. Define solved problem.

```
>> p = problem('P|prec|Cmax');
```

5. Call List Scheduling algorithm with taskset and problem created recently and define number of processors and desired heuristic.

```
>> S = listsch(T,p,2,'SPT')
 Set of 8 tasks
  There are precedence constraints
  There is schedule: List Scheduling
    Solving time: 1.1316s
```

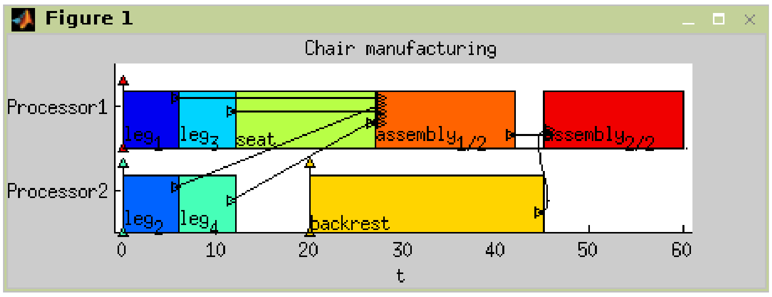
6. Visualize the final schedule by standard `plot` function, see Figure 11.4.

```
>> plot(S)
```

11.2 Real Word Case Studies

An real word example demonstrating applicability of the toolbox is shown in the following section.

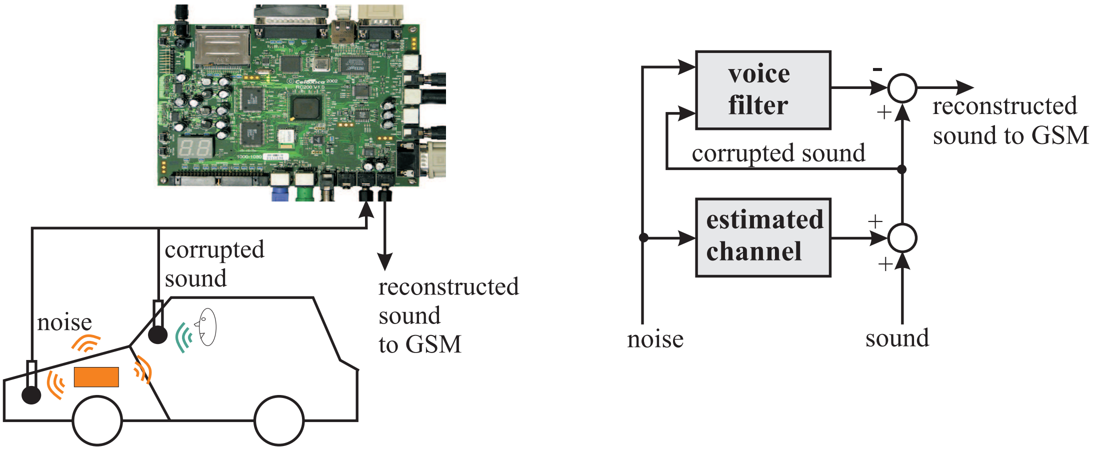
Figure 11.4 Result of case study as Gantt chart



11.2.1 Scheduling of RLS Algorithm for HW architectures with Pipelined Arithmetic Units

As an illustration, an example application of RLS (Recursive Least Squares) filter for active noise cancellation is shown in Figure 11.5 [RLS03]. The filter uses HSLA [HSLA02], a library of logarithmic arithmetic floating point modules. The logarithmic arithmetic is an alternative approach to floating-point arithmetic. A real number is represented as the fixed point value of logarithm to base 2 of its absolute value. An additional bit indicates the sign. Multiplication, division and square root are implemented as fixed-point addition, subtraction and right shift. Therefore, they are executed very fast on a few gates. On the contrary addition and subtraction require more complicated evaluation using look-up table with second order interpolation. Addition and subtraction require more hardware elements on the target chip, hence only one pipelined addition/subtraction unit is usually available for a given application. On the other hand the number of multiplication, division and square roots units can be nearly by arbitrary.

Figure 11.5 An application of Recursive Least Squares filter for active noise cancellation.



RLS filter algorithm is a set of equations (see the inner loop in Figure Figure 11.6) solved in an inner and an outer loop. The outer loop is repeated for each input data sample each 1/44100 seconds. The inner loop iteratively processes the sample up to the `N`-th iteration (`N` is the filter order). The quality of filtering increases with increasing number of filter iterations. `N` iterations of the inner loop need to be finished before the end of the sampling period when output data sample is generated and new input data sample starts to be processed.

The time optimal synthesis of RLS filter design on a HW architecture with HSLA can be formulated as cyclic scheduling on one dedicated processor (add unit) [Sucha04]. The tasks are constrained by precedence relations corresponding to the algorithm data dependencies. The optimization criterion is related to the minimization of the cyclic scheduling period `w` (like in an RLS filter application the execution of the maximum number of the inner loop periods `w` within a given sampling period increases the filter quality).

Figure 11.6 shows the inner loop of RLS algorithm. Data dependencies of this problem can be modeled by graph `rls_hsla` in Figure 11.7, where nodes represent particular operations of the RLS filter algorithm on add unit, i.e. a task on the dedicated processor. First user parameter on node represents processing time of task (time to 'feed' the add unit). The second one is a number of dedicated processor (unit).

Figure 11.6 The RLS filter algorithm.

```
for (k=1;k<HL;k=k+1)
   E(k)  = E(k-1) - Gfold(k) * Pold(k-1)
   f(k)  = Gold(k-1) * E(k-1)
   P(k)  = Pold(k-1) - Gbold(k) * E(k-1)
   b(k)  = G(k-1) * P(k-1)
   A(k)  = A(k-1) - Kold(k) * P(k-1)
   Gf(k) = Gfold(k) + bnold(k) * E(k)
   F(k)  = L * Fold(k) + f(k) * E(k-1) + T
   B(k)  = L * Bold(k) + b(k) * P(k-1) + T
   fn(k) = f(k) / F(k)
   bn(k) = b(k) / B(k)
   Gb(k) = Gbold(k) + fn(k) * P(k)
   K(k)  = Kold(k) + bn(k) * A(k)
   G(k)  = G(k-1) - bn(k) * b(k)
end
```

Table 11.3 Parameters of HSLA library.

unit	processing time	input-output latency
add	1	9
mul	1	2
div	1	2

User parameters on edges are lengths and heights, explained in Section [Cyclic scheduling (General)].

NOTE

In this case we consider two stages of the filter, i.e. one half of iterations is processed in one stage and second one on the second stage. After processing of the first half of iterations, partial results are passed to the second stage. When the second stage starts to process the input partial results, the first stage starts to process a new sample. Both stages have to share one dedicated processor (add unit), therefore we consider each operation on add unit to be represented by a task with processing time equal to 2 clock cycles. In the first clock cycle an operation of the first stage uses the add unit and in the second clock cycle the add unit is used by the second stage. For more detail see [Sucha04][RLS03] .

Solution of this scheduling problem is shown in following steps:

1. Load graph of the RLS filter into the workspace (graph `rls_hsla`).

```
>> load scheduling\stdemos\benchmarks\dsp\rls_hsla
```

2. Transform graph of the RLS filter to graph g weighted by lengths and heights.

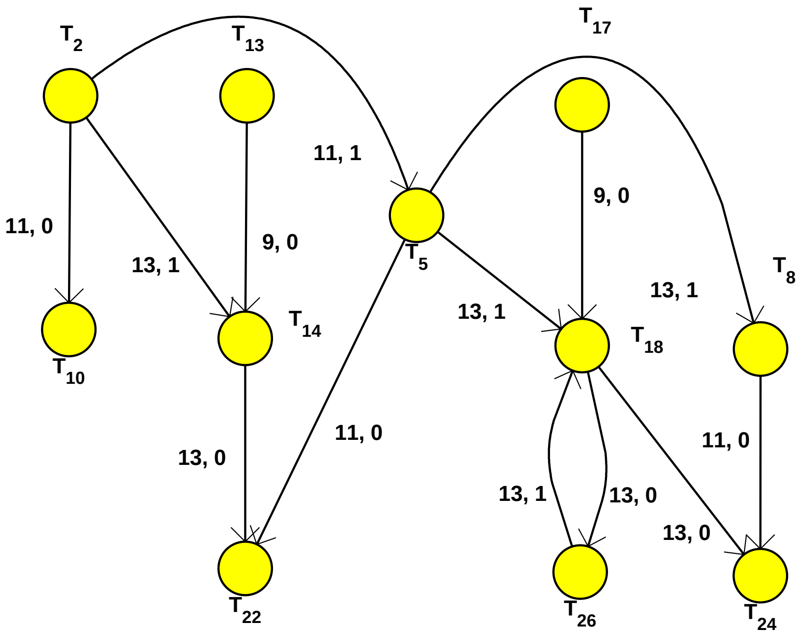
```
>> T = taskset(rls_hsla,'n2t',@node2task,'ProcTime', ...
              'Processor','e2p',@edges2param)
Set of 11 tasks
 There are precedence constraints
```

3. Define the problem, which will be solved.

```
>> prob=problem('CSCH')
 CSCH
```

4. Define optimization parameters.

Figure 11.7 Graph G modeling the scheduling problem on one add unit of HSLA.



```
>> schoptions=schoptionsset('verbose',0,'ilpSolver','glpk');
```

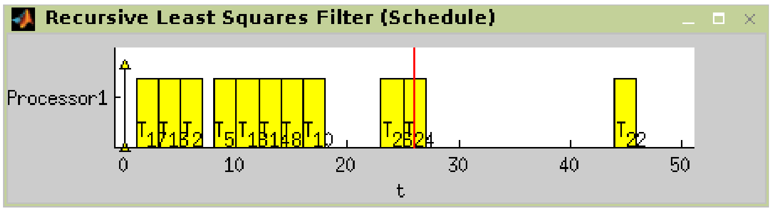
5. Call List Scheduling algorithm with taskset and problem created recently and define number of processors (conveyor belts).

```
>> TS = cycsch(T, prob, [1], schoptions)
 There are precedence constraints
 There is schedule: General cyclic scheduling algorithm
   Tasks period: 26
   Solving time: 1.297s
   Number of iterations: 1
```

6. Visualize the final schedule by standard `plot` function, see Figure 11.8.

```
>> plot(TS,'prec',0);
```

Figure 11.8 Resulting schedule of RLS filter.



Chapter 12

Reference guide

@graph/criticalcircuitratio.m

Name

criticalcircuitratio — finds the minimal circuit ratio of the input graph.

Synopsis

```
[w]=CRITICALCIRCUITRATIO(G)
[w]=CRITICALCIRCUITRATIO(L,H)
```

Description

Minimal circuit ratio of the graph is defined as w=min(L(C)/H(C)), where C is a circuit of graph G. L(C) is sum of lengths L of the circuit C and H(C) is sum of heights H of the circuit C.

[w]=CRITICALCIRCUITRATIO(G) finds minimal cycle ratio in graph G. where length and height are specified in first and second user parameter on edges (UserParam).

[w]=CRITICALCIRCUITRATIO(L,H) finds minimal circuit ratio in graph where length and height of edges is specified in matrices L and H.

See also

GRAPH/GRAPH, GRAPH/FLOYD, GRAPH/DIJKSTRA

@graph/dijkstra.m

Name

dijkstra — finds the shortest path between reference node and other nodes in graph.

Synopsis

```
DISTANCE = DIJKSTRA(GRAPH,STARTNODE,USERPARAMPOSITION)
```

Description

Parameters:

GRAPH
graph with cost betweens nodes

type inf when edge between two edges does not exist

STARTNODE
reference node

USERPARAMPOSITION
position in UserParam of Nodes where number representative color is saved. This parameter is optional. Default is 1.

DISTANCE
list of distances between reference node and other nodes

See also

GRAPH/GRAPH, GRAPH/FLOYD, GRAPH/CRITICALCIRCUITRATIO

@graph/edge2param.m

Name

edge2param — returns user parameters of edges in graph

Synopsis

```
USERPARAM = EDGE2PARAM(G)
USERPARAM = EDGE2PARAM(G,I)
USERPARAM = EDGE2PARAM(G,I,NOTEDGEPARAM)
```

Description

USERPARAM = EDGE2PARAM(G) returns cell with all UserParams. If there is not an edge between two nodes, the user parameter is empty array [].

USERPARAM = EDGE2PARAM(G,I) returns matrix of I-th UserParam of edges in graph G. The function returns cell similar to matrix if I is array.

USERPARAM = EDGE2PARAM(G,I,NOTEDGEPARAM) defines value of user parameter for missing edges (default is INF). Parameter NOTEDGEPARAM is disabled for graph with parallel edges.

See also

GRAPH/GRAPH, GRAPH/PARAM2EDGE, GRAPH/NODE2PARAM, GRAPH/PARAM2NODE

@graph/floyd.m

Name

floyd — finds a matrix of shortest paths for given digraph

Synopsis

```
[U[,P[,M]]]=FLOYD(G)
```

Description

The lengths of edges are set as UserParam in object edge included in G. If UserParam is empty, length is Inf.

Parameters:

G
object graph

U
matrix of shortest paths; if U(i,i)<0 then the digraph contains a cycle of negative length!

P
matrix of the vertex predecessors in the shortest path

M
Adjacency Matrix of lengths

Note: All matrices have the size nxn, where n is a number of vertices.

See also

GRAPH/GRAPH, GRAPH/DIJKSTRA, GRAPH/CRITICALCIRCUITRATIO

@graph/graph.m

Name

graph — creates the graph object.

Synopsis

```
G = GRAPH(Aw[[,noEdge],'Property name',value,...])
G = GRAPH('adj',A[,'Property name',value,...])
G = GRAPH('inc',I[,'Property name',value,...])
G = GRAPH('edl',edgeList[,'edgeDatatype',dataTypes][,'Property name',value,...])
G = GRAPH('ndl',nodeList[,'nodeDatatype',dataTypes][,'Property name',value,...])
G = GRAPH('ndl',nodeList,'edl',edgeList[,'nodeDatatype',dataTypes]
          [,'edgeDatatype',dataTypes][,'Property name',value,...])
G = GRAPH(TASKSET[,KW,TransformFunction[,Parameters]])
G = GRAPH(GRAPH[,'edl',edgeList][,'ndl',nodeList])
```

Description

G = GRAPH(...) creates the graph from ordered data structures.

Parameters:

Aw
Matrix of edges weigths (just for simple graph)

noEdge
Value of weigth in place without edge. Default is inf.

A
Adjacency matrix

I
Incidency matrix

edgeList
List of edges (cell): initial node, terminal node, user parameters

nodeList
List of nodes (cell): number of node, user parameters

dataTypes
Cell of data types

Name
Name of the graph - class char UserParam:

User-specified data

Color
Background color of graph in graphical projection

GridFreq
Sets the grid of graph in graphical projection - [x y]

G = GRAPH(TASKSET[,KW,TransformFunction[,Parameters]]) creates a graph from precedence constrains matrix of set of tasks:

TASKSET
Set of tasks

KW
Key word - define type of TransformFunction: 't2n' - task to node transfer function; 'p2e' - taskset's TSuserparams to edge's userparam

TransformFunction
Handler to a transform function, which transform tasks to nodes (resp. TSuserparam to userparam). If the variable is empty, standart function 'task/task2node' and 'graph/param2edge' are used.

Parameters
Parameters for transform function, frequently used for users selecting and sorting tasks parameters for setting userparameters of nodes. Parameters are colected to one parameter as cell before calling the transform function.

G = GRAPH(GRAPH[,'edl',edgeList][,'ndl',nodeList]) adds edges or/and nodes to existing graph:

GRAPH
Existing graph object

edgeList
List of edges: initial node, terminal node, user parameters

nodeList
List of nodes: number of node, user parameters

Example

```
>> Aw = [4 3 0; 0 0 5; 1 2 3]
>> g = graph(Aw,0,'Name','g1')
>> dataTypes = {'double','double','char'}
>> edgeList = {1,2, 35,[5 8],'edge1'; 2,3, 68,[2 7],'edge2'}
>> g = graph('edl',edgeList,'edgeDatatype',dataTypes)
>>
>> g = graph(T,'t2n',@task2node,'proctime','name','p2e',@param2edges)
```

See also

TASKSET/TASKSET, TASK/TASK2NODE, TASK/TASK2USERPARAM, GRAPH/PARAM2EDGE

@graph/graphcoloring.m

Name

graphcoloring — algorithm for coloring graph by minimal number of colors.

Synopsis

```
G2 = GRAPHCOLORING(G1,USERPARAMPOSITION)
```

Description

The function returns coloured graph. Algortihm sets color (RGB) of every node for graphic view and save it to UserParam of nodes as appropriate value representing the color. Input parameters are:

G1

input graph

USERPARAMPOSITION

position in UserParam of Nodes where number representative color will be saved. This parameter is optional. Default is 1.

See also

GRAPH/GRAPH, GRAPHEDIT

@graph/hamiltoncircuit.m

Name

hamiltoncircuit — finds Hamilton circuit in graph

Synopsis

```
G_HAM=HAMILTONCIRCUIT(G)
G_HAM=HAMILTONCIRCUIT(G,EDGESDIRECTION)
```

Description

G_HAM=HAMILTONCIRCUIT(G) solves the problem for directed graph G. Both G and G_HAM are Graph objects. Route cost is stored in Graph_out.UserParam.RouteCost

G_HAM=HAMILTONCIRCUIT(G,EDGESDIRECTION) defines direction of edges, if parameter EDGESDIRECTION is 'u' then the input graph is considered as undirected graph. When the parametr is 'd' the input graph is considered as directed graph (default).

See also

EDGES2PARAM, PARAM2EDGES, GRAPH/GRAPH, EDGES2MATRIXPARAM

@graph/mincostflow.m

Name

mincostflow — finds the least cost flow in graph G.

Synopsis

```
[G_FLOW, FMIN] = MINCOSTFLOW(G)
[G_FLOW, FMIN] = MINCOSTFLOW(U,C,D,N)
```

Description

[G_FLOW, FMIN] = MINCOSTFLOW(G) finds the cheapest flow in graph G. Prices in graph G, lower and upper bounds of flows are specified in first, second and third user parameter on edges (UserParam). The function returns graph G_FLOW, i.e. graph G enlarged with fourth user parameter which contains amount of flow in every edge. FMIN contains total cost.

[G_FLOW, FMIN] = MINCOSTFLOW(U,C,D,N) finds the same, but everything without using graph, only matrixes. U is matrix of prices, C means lower bounds of flows, D upper bounds. The function returns G_FLOW, matrix of minimal flows.

See also

GRAPH/GRAPH, ILINPROG, EDGES2MATRIXPARAM, MATRIXPARAM2EDGES

@graph/node2param.m

Name

node2param — returns user parameters of nodes in graph

Synopsis

```
USERPARAM = NODE2PARAM(G)
USERPARAM = NODE2PARAM(G,i)
```

Description

USERPARAM = NODE2PARAM(G) returns array or cell of all UserParams of nodes in graph G.

USERPARAM = NODE2PARAM(G,i) returns array or cell of i-th UserParam of nodes in graph G. If i is array the function returns cell similar to array.

See also

GRAPH/GRAPH, GRAPH/PARAM2NODE, GRAPH/EDGE2PARAM, GRAPH/PARAM2EDGE

@graph/param2edge.m

Name

param2edge — add to graph's user parameters datas from cell or matrix.

Synopsis

```
graph = PARAM2EDGE(graph,userparam)
graph = PARAM2EDGE(graph,userparam,i)
graph = PARAM2EDGE(graph,userparam,i,notedgeparam)
```

Description

graph = PARAM2EDGE(graph,userparam) graph - object graph userparam - matrix (simple graph and just 1 parameter in matrix) or cell (parallel edges or several parameters) with user params for edges.

graph = PARAM2EDGE(graph,userparam,i) graph - object graph userparam - matrix or cell with user params for edges i - i-th position of 1st value cell of new params (new UserParams replace original UserParams).

graph = PARAM2EDGE(graph,userparam,i,notedgeparam) graph - object graph userparam - matrix or cell with user params for edges i - i-th position of 1st value cell of new params (new UserParams replace original UserParams). notedgeparam - defines value of user parameter for missing edges. This value is used for checking consistence between graph and matrix userparam (default is INF).

See also

GRAPH/EDGE2PARAM, GRAPH/GRAPH

@graph/param2node.m

Name

param2node — add to graph's user parameters datas from cell or matrix.

Synopsis

```
graph = PARAM2NODE(graph,param)
graph = PARAM2EDGE(graph,param,N)
```

Description

graph = PARAM2NODE(graph,param) graph - object graph userparam - array (1 parameter in matrix) or cell (several parameters) with user params for nodes.

graph = PARAM2EDGE(graph,param,N) graph - object graph userparam - array or cell with user params for nodes N - N-th position in UserParam (new UserParams replace original UserParams).

See also

GRAPH/NODE2PARAM, GRAPH/GRAPH, GRAPH/EDGE2PARAM, GRAPH/PARAM2EDGE

@graph/qap.m

Name

qap — solves the Quadratic Assignment Problem

Synopsis

```
[MAP,FMIN,STATUS,EXTRA] = QAP(DISTANCESGRAPH,FLOWSGRAPH)
```

Description

The problem is defined using two graphs: graph of distances DISTANCESGRAPH and graph of flows FLOWSGRAPH.

A nonempty output is returned if a solution is found. The first return parameter MAP is the optimal mapping of nodes to locations. FMIN is optimal value of the objective function. Status of the optimization is returned in the third parameter STATUS (1-solution is optimal). The last parameter EXTRA is a data structure containing the field TIME - time (in seconds) used for solving.

Example

```
>> D = [0 1 1 2 3; ... % distances matrix
        1 0 2 1 2; ...
        1 2 0 1 2; ...
        2 1 1 0 1; ...
        3 2 2 1 0];
>> F = [0 5 2 4 1; ... % flows matrix
        5 0 3 0 2; ...
        2 3 0 0 0; ...
        4 0 0 0 5; ...
        1 2 0 5 0];
>> distancesg=graph(1*(D~=0));                       %Create graph of distances
>> distancesg=matrixparam2edges(distancesg,D,1,0);  %Insert distances into the graph
>> flowsg=graph(1*(F~=0));                           %Create graph of flow
>> flowsg=matrixparam2edges(flowsg,F,1,0);          %Insert flows into the graph
>> qap(distancesg,flowsg);
```

See also

GRAPH/GRAPH, IQUADPROG

@graph/spanningtree.m

Name

spanningtree — finds spanning tree of the graph

Synopsis

```
ST = SPANNINGTREE(GRAPH,USERPARAMPOSITION)
```

Description

GRAPH - graph with costs between nodes - type inf when edge between two edges does not exist USERPARAMPOSITION - position in UserParam of Nodes where number representative color is saved. This parameter is optional. Default is 1. ST - matrix which represetnts minimals body of the graph

See also

GRAPH/GRAPH, GRAPH/DIJKSTRA

@graph/tarjan.m

Name

tarjan — finds Strongly Connected Component

Synopsis

```
[COMPONENTS] = TARJAN(G)
```

Description

COMPONENTS = TARJAN(G) searches for strongly connected components using Tarjan's algorithm (it's actually depth first search). G is an input directed graph. The function returns a vector COMPONENTS. The value COMPONENTS(X) is number of component where the node X belongs to.

See also

GRAPH/GRAPH, GRAPH/SPANNINGTREE

@problem/problem.m

Name

problem — creation of object problem.

Synopsis

```
PROB = PROBLEM(NOTATION)
PROB = PROBLEM(SPECIALPROBLEM)
```

Description

The function creates object (PROB) describing a scheduling problem. The input parameter - NOTATION is composed of three fields alpha|betha|gamma.

alpha - describes the processor enviroment, alpha = alpha1 and alpha2

alpha1 characterizes the type of processor used:

nothing
single procesor

P
identical procesors

Q
uniform procesors

R
unrelated procesors

O
dedicated procesors: open shop system

F
dedicated procesors: flow shop system

J
dedicated procesors: job shop system

alpha2 denotes the number of processors in the problem

betha - describes task and resource characteristic:

pmtn
preemptions are allowed

prec
precedence constrains

rj
ready times differ per task

~dj
deadlines

in-tree
In-tree precedence constrains

pj=x
processing time equal x (x must be non-negative number)

gamma - denotes optimality criterion Cmax, sumCj, sumwCj, Lmax, sumDj, sumUj

Special scheduling problems (not covered by the notation) can be described by a string SPECIALPROBLEM. Permitted strings are: 'SPNTL' and 'CSCH';

Example

```
>> prob=PROBLEM('P3|pmtn,rj|Cmax')
>> prob=PROBLEM('SPNTL')
```

See also

TASKSET/TASKSET

@schedobj/get.m

Name

get — access/query SCHEDOBJ property values.

Synopsis

```
            GET(SCHEDOBJ)
            GET(SCHEDOBJ,'PropertyName')
    VALUE = GET(...)
```

Description

GET(SCHEDOBJ,'PropertyName') returns the value of the specified property of the SCHEDOBJ.

GET(SCHEDOBJ) displays all properties of SCHEDOBJ and their values.

See also

SCHEDOBJ/SET

@schedobj/get_graphic_param.m

Name

get_graphic_param — gets graphics params for object drawing

Synopsis

```
GET_GRAPHIC_PARAM(OBJ,C):
```

Description

GET_GRAPHIC_PARAM(OBJ,Parameter) return graphic parameters where:

OBJ
object

Parameter
name of parameter from the following set

Available parameters:

color
color

x
X coordinate

y
Y coordinate

See also

SCHEDOBJ/SET_GRAPHIC_PARAM

@schedobj/set.m

Name

set — sets properties to set of objects.

Synopsis

```
SET(OBJECT)
SET(OBJECT,'Property')
SET(OBJECT,'PropertyName',VALUE)
SET(OBJECT,'Property1',Value1,'Property2',Value2,...)
```

Description

SET(OBJECT) displays all properties of OBJECT and their admissible values.

SET(OBJECT,'Property') displays legitimate values for the specified property of OBJECT.

SET(OBJECT,'PropertyName',PropertyValue) sets the property 'PropertyName' of the OBJECT to the value PropertyValue.

SET(OBJECT,'Property1',PropertyValue1,'Property2',PropertyValue2,...) sets multiple OBJECT property values with a single statement.

One string have special meaning for PropertyValues: 'default' - use default value

See also

SCHEDOBJ/GET

@schedobj/set_graphic_param.m

Name

set_graphic_param — set graphics params for drawing

Synopsis

```
SET_GRAPHIC_PARAM(object[,keyword1,value1[,keyword2,value2[...]]])
```

Description

Set graphics params for drawing where:

object
object

keyword
name of parameter

value
value

Available keywords:

color
color

x
X coordinate

y
Y coordinate

See also

SCHEDOBJ/GET_GRAPHIC_PARAM

@task/add_scht.m

Name

add_scht — adds schedule (starts time and lenght of time) into a task

Synopsis

```
Tout = ADD_SCHT(Tin, start, lenght[, processor])
```

Description

Properties:

Tout
new task with schedule

Tin
task without schedule

start
array of start time

lenght
array of length of time

processor
array of number of processor

See also

TASK/GET_SCHT

@task/get_scht.m

Name

get_scht — gets schedule (starts time and length of time) from a task

Synopsis

```
[start, length, processor, period] = GET_SCHT(T)
```

Description

Properties:

T
task

start
array of start times

length
array of lengths of time

processor
array of numbers of processor

period
task period

See also

TASK/ADD_SCHT

@task/plot.m

Name

plot — graphic display of task

Synopsis

```
PLOT(T[,keyword1,value1[,keyword2,value2[...]]])
PLOT(T[,CELL])
```

Description

Properties:

T
task

keyword
configuration parameters for plot style

value
configuration value

CELL
cell array of configuration parameters and values

Available keywords:

color
color of task

movtop
vertical position of task (array if task is preempted)

texton
show text description above task (defaut value is true)

textin
show name of task inside the task (defaut value is false)

textins
structure with textin param detail. (see a. taskset/plot)

asap
show ASAP and ALAP borders (defaut value is false)

period
draw period mark

timeOfs
time offset. Used by periodic tasks.

See also

TASK/GET_SCHT

@task/task.m

Name

task — creates object task.

Synopsis

```
task = TASK([Name,]ProcTime[,ReleaseTime[,Deadline[,DueDate[,Weight[,Processor]]]]])
```

Description

Creates a task with parameters:

Name
name of the task (must by char!)

ProcTime
processing time (execution time)

ReleaseTime
release date (arrival time)

Deadline
deadline

DueDate
due date

Weight
weight (priotiry)

Processor
dedicated processor

The output task is a TASK object.

See also

TASKSET/TASKSET

@taskset/add_schedule.m

Name

add_schedule — adds schedule (starts time and lenght of time) for set of tasks

Synopsis

```
ADD_SCHEDULE(T, description[, start, length[, processor]])
ADD_SCHEDULE(T, keyword1, param1, ..., keywordn, paramn)
```

Description

Properties:

T
taskset; schedule will be save into this taskset.

description
description for schedule. It must be diferent than a key words below!

start
set of start time

lenght
set of lenght of time

processor
set of number of processor

keyword
key word (char)

param
parameter

Available key words are:

description
schedule description (it is same as above)

time
calculation time for search schedule

iteration
number of interations for search schedule

memory
memory allocation during schedule search

period
taskset period - scalar or vector for diferent period of each task

See also

TASKSET/GET_SCHEDULE

@taskset/alap.m

Name

alap — compute ALAP(As Late As Posible) for taskset

Synopsis

```
        Tout = ALAP(T, UB, [m])
 alap_vector = ALAP(T, 'alap')
```

Description

Tout=ALAP(T, UB, [m]) computes ALAP for all tasks in taskset T. Properties:

T
set of tasks

UB
upper bound

m
number of processors

Tout
set of tasks with alap

alap_vector = ALAP(T, 'alap') returns alap vector from taskset. Properties:

T
set of tasks

alap_vector
alap vector

ALAP for each task is stored into set of task, the biggest ALAP is returned.

See also

TASKSET/ASAP

@taskset/asap.m

Name

asap — computes ASAP(As Soon As Posible) for taskset

Synopsis

```
        Tout = ASAP(T [,m])
 asap_vector = ASAP(T, 'asap')
```

Description

Tout = ASAP(T [,m]) computes ASAP for all tasks in taskset T. Properties:

T
set of tasks

m
number of processors

Tout
set of tasks with asap

asap_vector = ASAP(T, 'asap') returns asap vector from taskset. Properties:

T
set of tasks

asap_vector
asap vector

See also

TASKSET/ALAP

@taskset/colour.m

Name

colour — returns taskset where tasks have set the color property

Synopsis

```
T = COLOUR(T[,colors])
```

Description

Properties:

T
taskset

colors
colors specification

Colors specification:

- RGB color matrix with 3 columns
- char with color name
- cell with combination RGB and names
- keyword 'gray' to use gray palete for coloring
- keyword 'colorcube' to use colorcube for coloring
- nothing - color palete use for coloring

For more information about colors in Matlab, see the documentation:

```
>>doc ColorSpec
```

See also

ISCOLOR, SCHEDOBJ/SET_GRAPHIC_PARAM, SCHEDOBJ/GET_GRAPHIC_PARAM, COLORCUBE

@taskset/count.m

Name

count — returns number of tasks in the Set of Tasks

Synopsis

```
count = COUNT(T)
```

Description

Properties:

T
set of tasks

count
number of tasks

See also

TASKSET/SIZE

@taskset/get_schedule.m

Name

get_schedule — gets schedule (starts time, lenght of time and processor) from a taskset

Synopsis

```
[start, lenght, processor, is_schedule] = GET_SCHEDULE(T)
```

Description

Properties:

T
taskset

start
cell/array of start times

lenght
cell/array of lengths of time

processor
cell/array of numbers of processor

is_schedule
1 - schedule is inside taskset

0 - taskset without schedule

See also

TASKSET/ADD_SCHEDULE

@taskset/plot.m

Name

plot — graphic display of set of tasks

Synopsis

```
PLOT(T)
PLOT(T[,C1,V1,C2,V2...])
```

Description

Parameters:

T
set of tasks

Cx
configuration parameters for plot style

Vx
configuration value

Properties:

MaxTime
... default: LCM (least common multiple) of task periods

Proc
0 - draw each task to one line

1 - draw each task to his processor

Color
0 - Black & White

1 - Generate colors only for tasks without color

2 - Generate colors for all tasks

default value is 1)

ASAP
0 - normal draw (default)

1 - draw tasks to their ASAP

Axis
[tmin tmax] set time interval for plot. Use NaN for automatic setting values. (NaN is default value)

Prec
0 - draw without precedens constrains

1 - draw with precedens constrains (default)

Period
0 - period mark is ignored

1 - draw one period with period mark(s) (default)

n - draw n periods witn n marks

Weight
0 - draw tasks in current order

1 - draw tasks in order by weights

Reverse

0 - draw tasks in order (top)1,2,3 .. n(bottom) (default)

1 - draw tasks in order (top)n,n-1,n-2,n-3 .. 1(bottom)

Axname

Cell with Y-axis name

Textins

Text-in setup, structure with 'fontsize' and 'textmovetop' fields

See also

TASKSET/TASKSET

@taskset/schparam.m

Name

schparam — returns parameters about schedule inside the set of tasks

Synopsis

```
param = schparam(T[, keyword])
```

Description

Properties:

T
set of tasks

keyword
schedule properties

param
output value

Keywords:

Cmax
Makespan

sumCj
Sum of completion times

sumwCj
Weighted sum of completion times

lmax
maximum lateness

period
Period

time
Solving time

memory
Memory allocation

iterations
Number of iterations

If keyword isn't defined, then struct with all properties is returned.

See also

TASKSET/ADD_SCHEDULE

@taskset/setprio.m

Name

setprio — sets priority (weight) of tasks acording to some rules.

Synopsis

```
  SETPRIO(T, RULE)
```

Description

Properties:

T
set of tasks

RULE
'rm' rate monotonic

See also

PTASK/PTASK

@taskset/size.m

Name

size — returns number of tasks in the Set of Tasks

Synopsis

```
size = SIZE(T)
```

Description

Properties:

T
set of tasks

size
number of tasks

Warning: This functions is deprecated. Please use function COUNT instead.

See also

TASKSET/COUNT

@taskset/sort.m

Name

sort — return sorted set of tasks over selected parameter.

Synopsis

```
TS = SORT(TS,parameter[,tendency])
[TS,order] = SORT(TS,parameter[,tendency])
```

Description

The function sorts tasks inside taskset. Input parameters are:

TS
Set of tasks

parameter
the propety for sorting ('ProcTime','ReleaseTime', 'Deadline','DueDate','Weight','Processor' or any vector with the same length as taskset)

tendency
'inc' as increasing (default), 'dec' as decreasing

order
list with re-arranged order

note: 'inc' tendenci is exactly nondecreasing, and 'dec' is exactly calcuated as nonincreasing

See also

TASKSET/TASKSET

@taskset/taskset.m

Name

taskset — creates a set of TASKs

Synopsis

```
setoftasks = TASKSET(T[,prec])
setoftasks = TASKSET(ProcTimeMatrix[,prec])
setoftasks = TASKSET(Graph[,Keyword,TransformFunction[,Parameters]...])
```

Description

creates a set of tasks with parameters:

T
an array or cell array of tasks ([T1 T2 ...] or {T1 T2 ...})

prec
precedence constraints

ProcTimeMatrix
an array of Processing times, for tasks which will be created inside the taskset.

Graph
Graph object

Keyword
Key word - define type of TransformFunction; 'n2t' - node to task transfer function, 'e2p' - edges' userparams to taskset userparam

TransformFunction
Handler to a transform function, which transform node to task or edges' userparams to taskset userparam. If the variable is empty, standart functions 'node/node2task' and 'graph/edges2param' is used.

Parameters
Parameters passed to transform functions specified by TransformFunction. It defines assignment of userparameters in the input graph to task properties. The transfer function will be called with one input parameter of cell, containing all the input parameters. Default value is: 'ProcTime','ReleaseTime','Deadline','DueDate', 'Weight','Processor','UserParam'

The output 'setoftasks' is a TASKSET object.

Example

```
>> T=taskset(Gr,'n2t',@node2task,'proctime','name','e2p',@edges2param)
```

See also

TASK/TASK, GRAPH/GRAPH, NODE/NODE2TASK, GRAPH/EDGE2PARAM

alg1rjcmax.m

Name

alg1rjcmax — computes schedule with Earliest Release Date First algorithm

Synopsis

```
TS = alg1rjcmax(T, problem)
```

Description

TS = alg1rjcmax(T, problem) finds schedule of the scheduling problem 1|rj|Cmax. Parameters:

T
input set of tasks

TS
set of tasks with a schedule

PROBLEM
description of scheduling problem (object PROBLEM) - '1|rj|Cmax'

See also

PROBLEM/PROBLEM, TASKSET/TASKSET, ALG1SUMUJ, BRATLEY, CYCSCH

alg1sumuj.m

Name

alg1sumuj — computes schedule with Hodgson's algorithm

Synopsis

```
TS = alg1sumuj(T, problem)
```

Description

TS = alg1sumuj(T, problem) finds schedule of the scheduling problem '1||sumUj'. Parameters:

T
input set of tasks

TS
set of tasks with a schedule

PROBLEM
description of scheduling problem (object PROBLEM) - '1||sumUj'

See also

PROBLEM/PROBLEM, TASKSET/TASKSET, ALG1RJCMAX, HORN

algpcmax.m

Name

algpcmax — computes schedule for 'P||Cmax'problem

Synopsis

```
TS = algpcmax(T, problem, No_Proc)
```

Description

TS = algpcmax(T, problem, No_Proc) finds schedule of scheduling problem 'P||Cmax'. Parameters:

T
input set of tasks

TS
set of tasks with a schedule

PROBLEM
description of scheduling problem (object PROBLEM) - 'P||Cmax',

No_Proc
number of processors for scheduling

See also

ALGPRJDEADLINEPRECCMAX, MCNAUGHTONRULE, HU, LISTSCH

algprjdeadlinepreccmax.m

Name

algprjdeadlinepreccmax — computes schedule for P|rj,prec,~dj|Cmax problem

Synopsis

```
TS = algprjdeadlinepreccmax(T, problem, No_proc)
```

Description

TS = algprjdeadlinepreccmax(T, problem, No_proc) finds schedule to the scheduling problem 'P|rj,prec,~dj|Cmax'. Parameters:

T
input set of tasks

TS
set of tasks with a schedule, PROBLEM:

description of scheduling problem (object PROBLEM) - 'P|rj,prec,~dj|Cmax'

No_proc
number of processors for scheduling

See also

PROBLEM/PROBLEM, TASKSET/TASKSET, ALGPCMAX, CYCSCH

bratley.m

Name

bratley — computes schedule by algorithm described by Bratley

Synopsis

```
TS = BRATLEY(T, problem)
```

Description

TS = BRATLEY(T, problem) finds schedule of the scheduling problem '1|rj,˜dj|Cmax'. Parameters:

T
input set of tasks

TS
set of tasks with a schedule

PROBLEM
description of scheduling problem (object PROBLEM)'1|rj,˜dj|Cmax'

See also

PROBLEM/PROBLEM, TASKSET/TASKSET, ALG1RJCMAX, SPNTL

brucker76.m

Name

brucker76 — Brucker's scheduling algorithm

Synopsis

```
TS = Brucker76(T, PROB, M)
```

Description

TS = Brucker76(T, PROB, M) returns optimal schedule of problem P|in-tree,pj=1|Lmax defined in object PROB. Parameters:

T
input taskset

PROB
problem

M
number of processors

See also

PROBLEM/PROBLEM, TASKSET/TASKSET, LISTSCH, HU

coffmangraham.m

Name

coffmangraham — is scheduling algorithm (Coffman and Graham) for P2|prec,pj=1|Cmax problem

Synopsis

```
TS = COFFMANGRAHAM(T, prob)
TS = COFFMANGRAHAM(T, prob, verbose)
TS = COFFMANGRAHAM(T, prob, options)
```

Description

The function finds schedule of the scheduling problem 'P2|prec,pj=1|Cmax'. Meaning of the input and output parameters is summarized below:

T
set of tasks, taskset object with precedence constrains

prob
problem P2|prec,pj=1|Cmax

verbose
level of verbosity 0 - no information (default); 1 - display scheduling information

options
global scheduling toolbox variables (SCHOPTIONSSET)

See also

HU, SCHOPTIONSSET, ALGPCMAX, BRUCKER76

cssimin.m

Name

cssimin — Cyclic Scheduling Simulator - input parser.

Synopsis

```
T=cssimin(filename)
T=cssimin(filename,schoptions)
```

Description

The function creates taskset T from input file describing cyclic scheduling problem. Input parameters are:

filename
specification file

schoptions
optimization options (See SCHOPTIONSSET)

Example

For more delails see User's Guide Section 10.6.

See also

CYCSCH, SCHOPTIONSSET

cssimout.m

Name

cssimout — Cyclic Scheduling Simulator - True-Time interface.

Synopsis

```
cssimout(T,ttinifile,ttcodefile)
```

Description

The function generates m-files for True-Time simulator. Input parameters are:

T
taskset with a cyclic schedule and parsed code in 'TSUserParam'

ttinifile
filename of True-Time initialization file

ttcodefile
filename of True-Time code file

See also

CSSIMIN

cycsch.m

Name

cycsch — solves general cyclic scheduling problem.

Synopsis

```
TASKSET = CYCSCH(TASKSET,PROB,M,SCHOPTIONS)
```

Description

Function returns optimal schedule for cyclic scheduling problem defined by parameters:

TASKSET
set of tasks (see CDFG2LHGRAPH)

PROB
description of scheduling problem (object PROBLEM)

M
vector with number of processors

SCHOPTIONS
optimization options (see SCHOPTIONSSET)

See also

GRAPH/CRITICALCIRCUITRATIO, TASKSET/TASKSET, ALGPRJDEADLINEPRECCMAX

graphedit.m

Name

graphedit — launch user-friendly editor of graphs able to export and import graphs between GUI and Matlab workspace.

Synopsis

```
        GRAPHEDIT(GRAPH)
        GRAPHEDIT(GRAPH1,GRAPH2,...,GRAPHN)
        GRAPHEDIT(sKeyWord)
        GRAPHEDIT(KeyWord,value,...)
    h = GRAPHEDIT(...)
```

Description

Parameters:

GRAPH
object graph

sKeyWord
single Key word 'fit' - Fits graph to canvas; 'center' - Centres drown graph

KeyWord
Keyword

h
handle to the figure object (main Graphedit window)

Available keywords:

zoom
Sets zoom to ordered value (1 == 100%)

viewedgesnames
Views/hides edges names (value: 'on','off')

viewnodesnames
Views/hides nodes names (value: 'on','off')

viewedgesuserparams
Views/hides edges user parameters (value: 'on','off')

viewnodesuserparams
Views/hides nodes user parameters (value: 'on','off')

viewparts
Views parts of graphedit (value: 'toolbar1','toolbar2','tabs', 'sliders','mainmenu','all')

hideparts
Hides parts of graphedit (value: 'toolbar1','toolbar2','tabs', 'sliders','mainmenu','all')

position
Sets position and size of graphedit window (value: [x, y, width, height])

lockup
Forgids user any interactions (value: 'on','off')

viewtab
Views graph with ordered tab (value: tab's ordinal number)

closetab
Closes canvas with ordered tab (value: tabs ordinal numbers)

createtab
Creates new canvas (value: graph object)

drawintab
Draws ordered graph in actual viewed tab (value: graph object)

importbackground
Imports picture and put it in canvas (value: picture name, cData)

fitbackground
Fits background image to height or width (value: 'height','widht')

removebackground
Removes last background image

propertyeditor
Views/hides property editor (value: 'on','off')

librarybrowser
Views/hides library browser (value: 'on','off')

nodedesigner
Views/hides node designer (value: 'on','off')

fontsizenames
Sets font size of texts Name (value: numeric value)

fontsizeuserparams
Sets font size of texts UsaerParam (value: numeric value)

arrowsvisibility
Views/hides arrows (value: 'on','off')

saveconfiguration
Saves actual graphedit configuration (value: '', 'filename')

movenode
Moves ordered nodes to required position (value: list of nodes and positions (cell))

Example

```
>>  graphedit(graph([4 3 inf; inf inf 5; 1 2 3],'Name','graph_1'))
>>
>>  graphedit('zoom',0.8,'viewedgesuserparams','off')
>>
>>  graphedit('movenode',{1, 100, 150; 2, 150, 150})
            % moves node 1 to position [100,150] and node 2 to [150,150]
```

See also

GRAPH/GRAPH

horn.m

Name

horn — computes schedule with Horn'74 algorithm

Synopsis

```
TS = horn(T, problem)
```

Description

TS = horn(T, problem) adds schedule to the set of tasks Parameters:

T
input set of tasks

TS
set of tasks with a schedule

problem
description of scheduling problem - '1|pmtn,rj|Lmax'

See also

PROBLEM/PROBLEM, TASKSET/TASKSET, ALG1RJCMAX, ALG1SUMUJ

hu.m

Name

hu — is scheduling algorithm for P|in-tree,pj=1|Cmax problem (can be called on labeled taskset with problem P2|prec,pj=1|Cmax)

Synopsis

```
TS = HU(T, prob, m)
TS = HU(T, prob, m, verbose)
TS = HU(T, prob, m, options)
```

Description

Properties:

T
set of tasks, taskset object with precedence constrains

prob
P|in-tree,pj=1|Cmax

P2|prec,pj=1|Cmax

m
processors

verbose
0 - default, no information

1 - display scheduling information

options
global scheduling toolbox variables (SCHOPTIONSSET)

See also

COFFMANGRAHAM, SCHOPTIONSSET, ALGPCMAX, BRUCKER76

ilinprog.m

Name

ilinprog — universal interface for integer linear programming.

Synopsis

```
[XMIN,FMIN,STATUS,EXTRA] = ILINPROG(OPTIONS,SENSE,C,A,B,CTYPE,LB,UB,VARTYPE)
```

Description

Parameters:

OPTIONS
optimization options (see SCHOPTIONSSET)

SENSE
indicates whether the problem is a minimization=1 or maximization=-1

C
column vector containing the objective function coefficients

A
matrix representing linear constraints

B
column vector of right sides for the inequality constraints CTYPE: - column vector that determines the sense of the inequalities (CTYPE(i) = 'L' less or equal; CTYPE(i) = 'E' equal; CTYPE(i) = 'G' greater or equal)

LB
column vector of lower bounds

UB
column vector of upper bounds

VARTYPE
column vector containing the types of the variables (VARTYPE(i) = 'C' continuous variable; VARTYPE(i) = 'I' integer variable)

A nonempty output is returned if a solution is found:

XMIN
optimal values of decision variables

FMIN
optimal value of the objective function

STATUS
status of the optimization (1-solution is optimal)

EXTRA
data structure containing the following fields (TIME - time (in seconds) used for solving; LAMBDA - dual variables)

See also

SCHOPTIONSSET, MAKE

iquadprog.m

Name

iquadprog — Universal interface for mixed integer quadratic programming.

Synopsis

```
[XMIN,FMIN,STATUS,EXTRA] = ILINPROG(OPTIONS,SENSE,H,C,A,B,CTYPE,LB,UB,VARTYPE)
```

Description

The function has folowing parameters:

OPTIONS
optimization options (see SCHOPTIONSSET)

SENSE
indicates whether the problem is a minimization=1 or maximization=-1

H
square matrix containing quadratic part of the objective function coefficients

C
column vector containing linear part of the objective function coefficients

A
matrix representing linear constraints

B
column vector of right sides for the inequality constraints

CTYPE
column vector that determines the sense of the inequalities (CTYPE(i) = 'L' less or equal; CTYPE(i) = 'E' equal; CTYPE(i) = 'G' greater or equal)

LB
column vector of lower bounds

UB
column vector of upper bounds

VARTYPE
column vector containing the types of the variables (VARTYPE(i) = 'C' continuous variable; VARTYPE(i) = 'I' integer variable)

A nonempty output is returned if a solution is found:

XMIN
optimal values of decision variables

FMIN
optimal value of the objective function

STATUS
status of the optimization (1-solution is optimal)

EXTRA
data structure containing the only one field TIME, i.e. time (in seconds) used for solving

See also

SCHOPTIONSSET, MAKE

listsch.m

Name

listsch — Computes schedule by algorithm described by Graham 1966

Synopsis

```
taskset = LISTSCH(taskset, problem, m [,strategy] [,verbose])
taskset = LISTSCH(taskset, problem, m [,options])
```

Description

Function is a list scheduling algorithm for parallel prllel processors. The parameters are:

taskset
set of tasks,

problem
description of scheduling problem (object PROBLEM),

m
number of processors,

strategy
'EST', 'ECT', 'LPT', 'SPT' or any handler of function,

verbose
level of verbosity 0 - default, 1 - brief info, 2- tell me anything,

options
global scheduling toolbox variables (SCHOPTIONSSET)

See also

PROBLEM/PROBLEM, TASKSET/TASKSET, SORT_ECT, SORT_EST, SCHOPTIONSSET

mcnaughtonrule.m

Name

mcnaughtonrule — computes schedule with McNaughtons's algorithm

Synopsis

```
TS = MCNAUGHTONRULE(T, prob, M)
```

Description

TS = MCNAUGHTONRULE(T, prob, No_Proc) finds schedule of scheduling problem 'P|pmtn|Cmax'. First input parameter T is set of tasks to be schedule. The second parameter is description of the scheduling problem (see PROBLEM/PROBLEM) and the last parameter M is number of processors. Resulting schedule is returned in TS.

See also

PROBLEM/PROBLEM, TASKSET/TASKSET, ALGPCMAX

private/bezier.m

Name

bezier — computes points on Bezier curve

Synopsis

```
[x,y] = BEZIER(x0,y0,x1,y1,x2,y2,x3,y3[,reduction])
```

Description

A cubic Bezier curve is defined by four points. Two are endpoints. (x0,y0) is the origin endpoint. (x3,y3) is the destination endpoint. The points (x1,y1) and (x2,y2) are control points.

Function remove points in which vectors given by adjacent points inclined angel with tangent smaller than input value 'reduction'. Default value is 0.005.

See also

TASKSET/PLOT

randdfg.m

Name

randdfg — random Data Flow Graph (DFG) generator.

Synopsis

```
DFG=RANDDFG(N,M,DEGMAX,NE)
G=RANDDFG(N,M,DEGMAX,NE,NEH,HMAX)
```

Description

DFG=RANDDFG(N,M,DEGMAX,NE) generates DFG, where N is the number of nodes in the graph DFG, M is the number of dedicated processors. Relation of node to a processor is stored in 'G.N(i).UserParam'. Parameter DEGMAX restricts upper bound of outdegree of vertices. NE is number of edges. Resultant graph is Direct Acyclic Graph (DAG).

G=RANDDFG(N,M,DEGMAX,NE,NEH,HMAX) generates cyclic DFG (CDFG), where NEH is number of edges with parameter '0 < G.E(i).UserParam <= HMAX'. Other edges has user parameter 'G.E(i).UserParam=0'.

See also

GRAPH/GRAPH, CYCSCH

randtaskset.m

Name

randtaskset — Generates set of tasks of random parameters selected from uniform distribution.

Synopsis

```
RTASKSET =
RANDTASKSET(nbTasks,[Name,]ProcTime[,ReleaseTime[,Deadline ...
            [,DueDate[,Weight[,Processor]]]]])
```

Description

Function has following parameters:

nbTasks
number of tasks in set of tasks

Name
name of the tasks (must by char!)

ProcTime
range of processing time (execution time)

ReleaseTime
range of release date (arrival time)

Deadline
range of deadline

DueDate
range of due date

Weight
range of weight (priotiry)

Processor
range of dedicated processor

The output RTASKSET is a TASKSET object.

See also

TASKSET/TASKSET

satsch.m

Name

satsch — computes schedule by algorithm described in [TORSCHE06]

Synopsis

```
taskset = SATSCH(taskset, problem, m)
```

Description

Properties:

taskset
set of tasks

problem
description of scheduling problem (object PROBLEM)

m
number of processors

See also

PROBLEM/PROBLEM, TASKSET/TASKSET

schoptionsset.m

Name

schoptionsset — Creates/alters SCHEDULING TOOLBOX OPTIONS structure.

Synopsis

```
SCHOPTIONS = SCHOPTIONSSET('PARAM1',VALUE1,'PARAM2',VALUE2,...)
SCHOPTIONS = OPTIMSET(OLDSCHOPTIONS,'PARAM1',VALUE1,...)
```

Description

SCHOPTIONS = SCHOPTIONSSET('PARAM1',VALUE1,'PARAM2',VALUE2,...) creates an optimization options structure SCHOPTIONS in which the named parameters have the specified values.

SCHOPTIONS = SCHOPTIONSSET(OLDSCHOPTIONS,'PARAM1',VALUE1,...) creates a copy of OLDSCHOPTIONS with the named parameters altered with the specified values. Supported parameters are summarized below.

GENERAL:

maxIter
Maximum number of iterations allowed. (positive integer)

verbose
Verbosity level. (0 = be silent, 1 = display only critical messages 2 = display everything)

logfile
Create a log file. (0 = disable, 1 = enable)

logfileName
Specifies logfile name. (character array)

strategy
Specifies strategy of algorithm.

ILP,MIQP:

ilpSolver
Specify ILP solver ('glpk' or 'lp_solve' or 'external')

extIlinprog
Specifies external ILP solver interface. Specified function must have the same parameters as function ILINPROG. (function handle)

miqpSolver
Specify MIQP solver ('miqp' or 'external')

extIquadprog
Specifies external MIQP solver interface. Specified function must have the same parameters as function IQUADPROG. (function handle)

solverVerbosity
Verbosity level. (0 = be silent, 1 = display only critical messages, 2 = display everything)

solverTiLim
Sets the maximum time, in seconds, for a call to an optimizer. When solverTiLim<=0, the time limit is ignored. Default value is 0. (double)

CYCLIC SCHEDULING:

cycSchMethod
Specifies an method for Cyclic Scheduling algorithm ('integer' or 'binary')

varElim

Enables elimination of redundant binary decision variables in ILP model (0 = disable, 1 = enable). varElimILPSolver

Specifies another ILP solver for elimination of redundant binary decision variables.

secondaryObjective

Enables minimization of iteration overlap as secondary objective function (0 = disable, 1 = enable).

qmax

Maximal overlap of iterations qmax>=0 (default [] - undefined).

SCHEDULING WITH POSITIVE AND NEGATIVE TIME-LAGS:

spntlMethod

Specifies an method for SPNTL algorithm ('BaB' - Branch and Bound algorithm; 'BruckerBaB' - Brucker's Branch and Bound algorithm; 'ILP' - Integer Linear Programming)

See also

ILINPROG, CYCSCH, SPNTL

spntl.m

Name

spntl — computes schedule with Positive and Negative Time-Lags

Synopsis

```
TS = SPNTL(T,PROB,SCHOPTIONS)
```

Description

TS = SPNTL(T,PROB,SCHOPTIONS) returns the optimal schedule TS of set of tasks defined in T for scheduling problem 'SPNTL' defined in PROB (object PROBLEM). Parameter SCHOPTIONS specifies an extra optimization options.

See also

ILINPROG, SCHOPTIONSSET, BRATLEY, CYCSCH

xmlsave.m

Name

xmlsave — saves variables to file in xml format.

Synopsis

```
        XMLSAVE(FILENAME,VARIABLE1,VARIABLE2,VARIABLE3,...)
        XMLSAVE('',VARIABLE1,VARIABLE2,VARIABLE3,...)
  out = XMLSAVE('',VARIABLE1,VARIABLE2,VARIABLE3,...)
```

Description

XMLSAVE saves variables into XML file named 'FILENAME'. Temporary file is created and immediately opened in editor if parameter FILENAME is empty string. Alternatively xmlsave returns conntents of xml file in the first output variable.

Example

```
>> t=task('t1',5,1,10);
>> txml=xmlsave('',t)
txml =
<?xml version="1.0" encoding="utf-8"?>
<matlabdata date="24-Sep-2007 08:45:33" proccessor="TORSCHE Scheduling Toolbox for Matlab" ver="0.2">
   <task id="t"><!--Basic Params-->
      <name>t1</name>
      <proctime>5</proctime>
      <releasetime>1</releasetime>
      <deadline>10</deadline>
      <duedate>Inf</duedate>
      <weight>1</weight><!--Graphics parameters-->
      <graphicparam>
         <position>
            <x>0</x>
            <y>0</y>
         </position>
      </graphicparam>
   </task>
</matlabdata>
```

See also

TASKSET/TASKSET, CSSIMOUT

Literature

[Ahuja93] *Network Flows*, Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin, Prentice Hall, February 18, 1993, 864, 013617549X. 9.6, 9.7

[Bemporad04] *Mixed Integer Quadratic Program (MIQP) solver for Matlab*, Alberto Bemporad and Domenico Mignone, Automatic Control Laboratory, ETH Zentrum, Zurich, Switzerland, 2004. 10.2, 10.5

[Berkelaar05] *lp_solve (Open source (Mixed-Integer) Linear Programming system) Version 5.1.0.0*, Michel Berkelaar, Kjell Eikland, and Peter Notebaert, 2005. 10.2

[Bru76] *Sequencing unit-time jobs with treelike precedence on m processors to minimize maximum lateness*, P.J. Brucker, Proc. IX International Symposium on Mathematical Programming, Budapest, 1976. 7.2, 7.10

[Brucker99] *A branch and bound algorithm for a single-machine scheduling problem with positive and negative time-lags*, P. Brucker, T. Hilbig, and J. Hurink, Discrete Applied Mathematics, 1999. 7.2, 7.11

[Butazo97] *Hard Real-Time Computing Systems*, G. C. Butazo, Kluwer Academic Publishers, 1997, 0-7923-9994-3. 3.1

[Błażewicz01] *Scheduling Computer and Manufacturing Process*, J. Błażewicz, K. H. Ecker, E. Pesch, G. Schmidt, and J. Węglarz, Springer, 2001, 3-540-41931-4. 3.1, 7.2, 7.4, 7.5, 7.7, 7.9, 7.9.1, 7.9.2, 7.10, 7.11, 7.14, 7.15

[Błażewicz83] *Scheduling subject to resource constrains: classification and complexity*, J. Błażewicz, J. K. Lenstra, and A. H. Rinnooy Kan, Ann. Discrete Math, 11-24, 1933. 1, 2.4, 5.1, 7

[CDFG05] *Control-Data Flow Graph Toolset*, Jinhwan Jeon and Yong-Jin Ahn, http://poppy.snu.ac.kr/CDFG/, 2005. 7.12

[CPLEX04] *CPLEX Version 9.1*, ILOG, Inc., Department for Applied Informatics, Moscow Aviation Institute, Moscow, 2004. 10.2

[Cervin06] *TRUETIME 1.4—Reference Manual*, M. Ohlin, D. Henriksson, and A. Cervin, Department of Automatic Control, Lund University, 2006. 10.6

[DSVF06] *Implementation of Digital Filters as Part of Custom Synthesizer with NI SPEEDY 33*, National Instruments, National Instruments, http://zone.ni.com/devzone/cda/tut/p/id/3476, 2006. 10.6.2

[DSVF06] *Depth-First Search and Linear Graph Algorithms*, Tarjan, R. E., SIAM J. Comput, 146-160, 1972. 9.5

[Demel02] *Grafy a jejich aplikace*, Demel, J., Academia, 2002. 9.2, 9.9

[Diestel00] *Graph Theory*, Reinhard Diestel, Springer, February, 2000, 313, 0-38-798976-5. 9.4, 9.8

[Dongen92] *A Polynomial Time Method for Optimal Software Pipelining*, V. H. Dongen and G. R. Gao, Lecture Notes in Computer Science, Springer-Verlag, 1992, 613-624, 3-540-55895-0. 7.12

[Fettweis86] *Wave digital filters: theory and practice*, A. Fettweis, Proceedings of the IEEE, 270-327, February, 1986. 7.12, 7.12

[Graham66] *Bounds for certain multiprocessing anomalies*, R. L. Graham, Bell System Technical Journal, 1966, 45:1563–1581. 7.2

[Graham79] *Optimization and approximation in deterministic sequencing and scheduling theory: a survey*, R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. Rinnooy Kan, Ann. Discrete Math, 287-326, 1979. 1, 2.4, 5.1, 7

[HSLA02] *Logarithmic number system and floating-point arithmetics on FPGA*, R. Matoušek, M. Tichý, Z. Pohl, J. Kadlec, and C. Softley, Field-Programmable Logic and Applications: Reconfigurable Computing is Going Mainstream, Lecture Notes in Computer Science 2438, 627-636, 2002. 11.2.1

[Hanen95] *A Study of the Cyclic Scheduling Problem on Parallel Processors*, C. Hanen and A. Munier, Discrete Applied Mathematics, 167-192, February, 1995. 7.2, 7.12

[Hanzalek04] *Scheduling with Start Time Related Deadlines*, P. Šůcha and Z. Hanzálek, IEEE Conference on Computer Aided Control Systems Design, September, 2004. 7.2, 7.11

[Hanzalek07] *Deadline constrained cyclic scheduling on pipelined dedicated processors considering multiprocessor tasks and changeover times*, P. Šůcha and Z. Hanzálek, Mathematical and Computer Modelling Journal (Article in Press), 2007. 7.12

[Heemstra92] *Range-Chart-Guided Iterative Data-Flow-Graph Scheduling*, Sonia M. Heemstra de Groot, Sabih H. Gerez, and Otto E. Herrmann, IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, May, 1992, 351-364, 013617549X. 7.12

[Horn74] *Some Simple Scheduling Algorithms*, W. A. Horn, Naval Res. Logist. Quart., 21, 1974, 177-185. 7.2

[Leung04] *Handbook of Scheduling*, Joseph Y-T. Leung, Chapman & Hall/CRC, April 15, 2004, 1120, 1-58488-397-9. 7.9

[Liu00] *Real-time systems*, J. W. Liu, Prentice-Hall, 2000, 0-13-099651-3. 3.1

[Makhorin04] *GLPK (GNU Linear Programming Kit) Version 4.6*, Andrew Makhorin, Department for Applied Informatics, Moscow Aviation Institute, Moscow, 2004. 10.2

[Memik02] *Accelerated SAT-based Scheduling of Control/Data Flow Graphs*, S. O. Memik and F. Fallah, 20th International Conference on Computer Design (ICCD), IEEE Computer Society, 2002, 395-400. 7.13.2

[Paulin86] *HAL: A multi-paradigm approach to automatic data path synthesis*, Pierre G. Paulin, John P. Knight, and Emil F. Girczyc, 23rd IEEE Design Automation Conf, July, 1986, 263-270. 7.12

[Pinedo02] *Scheduling*, Michael Pinedo, Prentice Hall, 2002, 586, 0-13-028138-7. 1

[Pohl05] *Performance Tuning of Iterative Algorithms in Signal Processing*, Z. Pohl, P. Šůcha, J. Kadlec, and Z. Hanzálek, The International Conference on Field-Programmable Logic and Applications (FPL'05), Tampere, Finland, 699-702, 2005. 7.12

[QAPLIB06] *QAPLIB - A Quadratic Assignment Problem Library*, Rainer E. Burkard, Eranda Çela, Stefan E. Karisch, and Franz Rendl, Institute of Mathematics, Graz University of Technology, 2006. 9.10, 9.10

[RLS03] *FPGA Implementation of the Adaptive Lattice Filter*, A. Heřmánek, Z. Pohl, and J. Kadlec, Field-Programmable Logic and Applications. Proceedings of the 13th International Conference, 1095-1098, 2003. 11.2.1, 11.2.1

[Rabaey91] *Fast Prototyping of Datapath-Intensive Architectures*, J. M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, IEEE Design and Test of Computers, 1991, 40-51. 7.12

[Rau81] *Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing*, B. R. Rau and C. D. Glaeser, Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture, 183-198, 1981.

[Stützle99] *New ideas in optimization ACO Algorithms for the Quadratic Assignment Problem (New ideas in optimization)*, Thomas Stützle and Marco Dorigo, McGraw-Hill Ltd., UK, 1999, 33-50, 0-07-709506-5. 9.10

[Sucha04] *Scheduling of Iterative Algorithms on FPGA with Pipelined Arithmetic Unit*, P. Šůcha, Z. Pohl, and Z. Hanzálek, 10th IEEE Real-Time and Embedded Technology and Applications Symposium, May, 2004. 7.2, 7.12, 7.12, 11.2.1, 11.2.1

[Sucha07] *Cyclic Scheduling of Tasks with Unit Processing Time on Dedicated Sets of Parallel Identical Processors*, P. Šůcha and Z. Hanzálek, Multidisciplinary International Scheduling Conference: Theory and Application (MISTA07), August, 2007. 7.12

[TORSCHE06] *TORSCHE Scheduling Toolbox for Matlab*, P. Šůcha, M. Kutil, M. Sojka, and Z. Hanzálek, IEEE International Symposium on Computer-Aided Control Systems Design (CACSD'06), Munich, Germany, 2006. 7.2