TORSCHE Scheduling Toolbox for Matlab

User's Guide

(Release 0.5.0$_{\beta 3}$)

TORSCHE Scheduling Toolbox for Matlab

User's Guide

(Release 0.5.0β3; Rev. 2967)

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

Prague, February 15, 2010

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 $\backslash scheduling\backslash contrib\backslash unzip\backslash LICENSE$).
- 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

10 Graph Algorithms

11 Optimization Algorithms

12 Other Algorithms

13 Case Studies

List of Tables

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, an 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, "Shop", Chapter 6, "Classification in Scheduling" and Chapter 7, "Graphs" presents the tool architecture and basic notation. The most interesting part is Chapter 8, "Scheduling Algorithms" describing implemented off-line scheduling algorithms demonstrated on various examples. Section Chapter 9, "Real-Time Scheduling" is dedicated to on-line scheduling and on-line scheduling algorithms. Graph algorithms are discussed in Chapter 10, "Graph Algorithms". Supplementary algorithms are described in Chapter 12, "Other Algorithms". The text is supplemented with case studies, presented in Chapter 13, "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 the Matlab environment). Before that, please specify a compiler using command `mex -setup` (also in the Matlab environment). It is recommended 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 of the help text.

2.4 How to Solve Your Scheduling Problems

Solving procedure of your scheduling problem can be divided into three 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. Deadline is not defined (the fourth parameter is equal to inf). 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
```

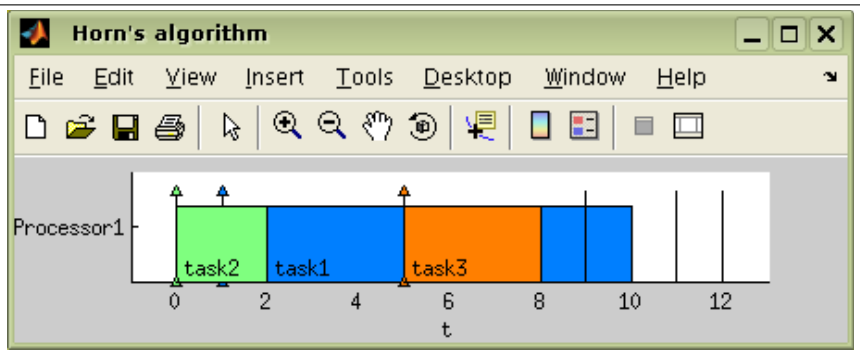
Now we can execute the scheduling algorithm, for example Horn's algorithm:

```
>> 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 Modifications of Objects

Values of object properties can be obtained or modified by several ways. The first way is the same as in Matlab user's guide, while the second is based on the dot notation. Both of them can be used for all TORSCHE objects. Dot notation allows rapid code development together with using of TAB completion feature (Matlab displays all object properties and methods that can be accessed after pressing TAB key while object and dot is written).

Command set can be used to obtain all properties together with their description.

```
>> set(t1)
              Name: name of the TASK
          ProcTime: processing time
       ReleaseTime: release time (arrival time)
          Deadline: deadline
           DueDate: duedate
            Weight: weight (priority)
         Processor: dedicated processor
         UserParam: user parameters
             Notes: An arbitrary string
```

Values of all properties can be obtained using command get.

```
>> get(t1)
       Name: 'task1'
   ProcTime: 5
ReleaseTime: 1
   Deadline: 15
    DueDate: 10
     Weight: 1
  Processor:
  UserParam:
      Notes: ''
```

Two equivalent modalities can be used to set specified property.

```
>> set(t1,'Deadline',15);
>> t1.DueDate = 10;
```

Retrieving of specified properties is very similar.

```
>> get(t1,'Deadline')
ans =
  15
>> t1.DueDate
ans =
  10
```

Object method calling can be performed by dot notation too.

```
>> start = TS.get_schedule();
```

2.6 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])

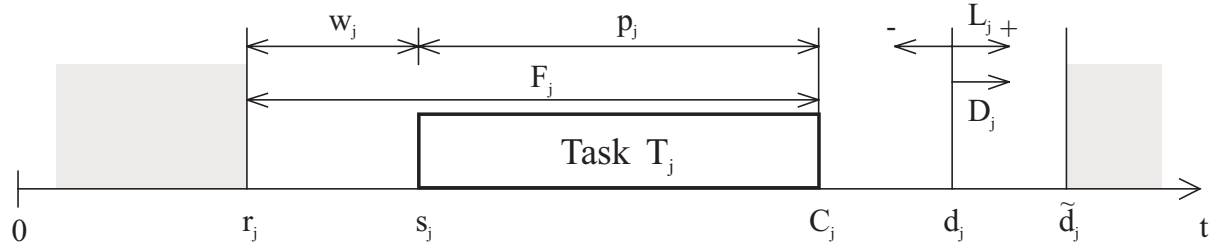
Deadline[I] d_j (`Deadline`)
specifies a time limit by which the task has to be completed, otherwise the scheduling is assumed to fail

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 which is stored into a variable, e.g. `t1`. Examples of creating task objects 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.

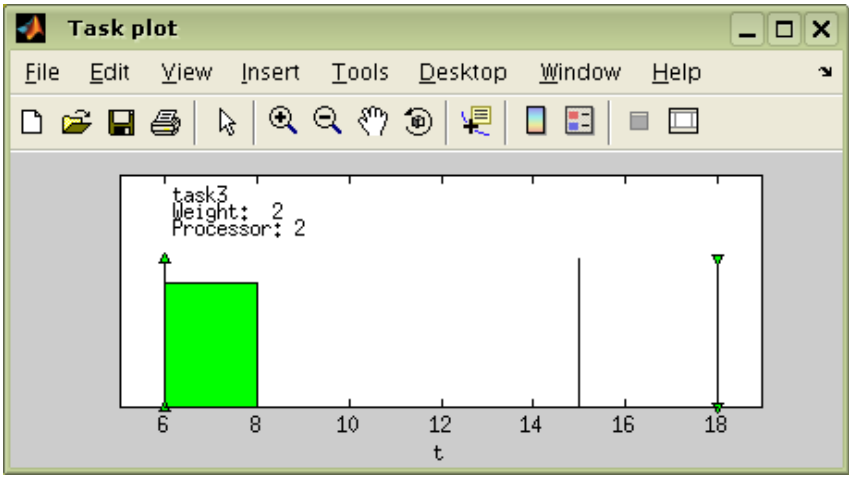
3.4 Object task Modifications

Task modification is done just as the other objects. Basic operations are described in Section 2.5.

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 each 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.

Figure 3.3 Plot example



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 shown 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 9, "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 7, "Graphs") defining a graph where nodes correspond to tasks and edges are precedence constraints between these tasks. If there is an edge from T_i to T_j in the graph, it means that T_i must be completed before T_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

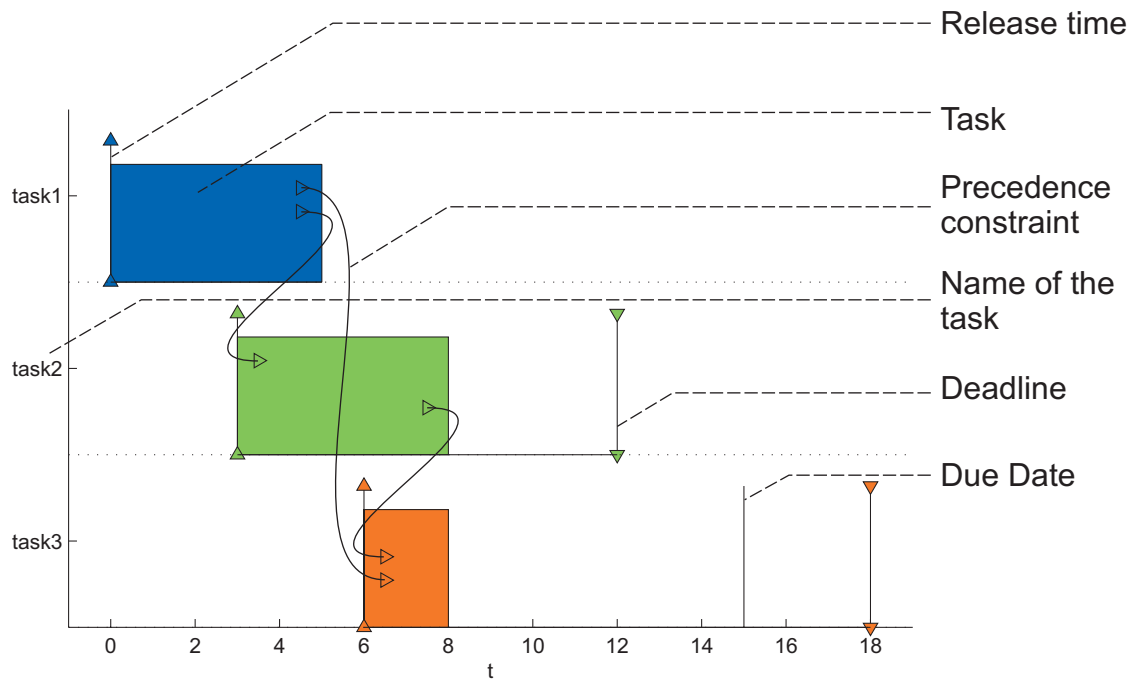
As for single tasks, command `plot` can be used to draw parameters of set of tasks graphically. An example of plot output with explanation of used marks is shown in Figure 4.2. For more details see Reference Guide @taskset/plot.m.

4.3 Set of Tasks Modification

Taskset modification is done just as the other objects. Basic operations are described in Section 2.5.

Figure 4.2 Gantt chart for a set of scheduled tasks

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



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

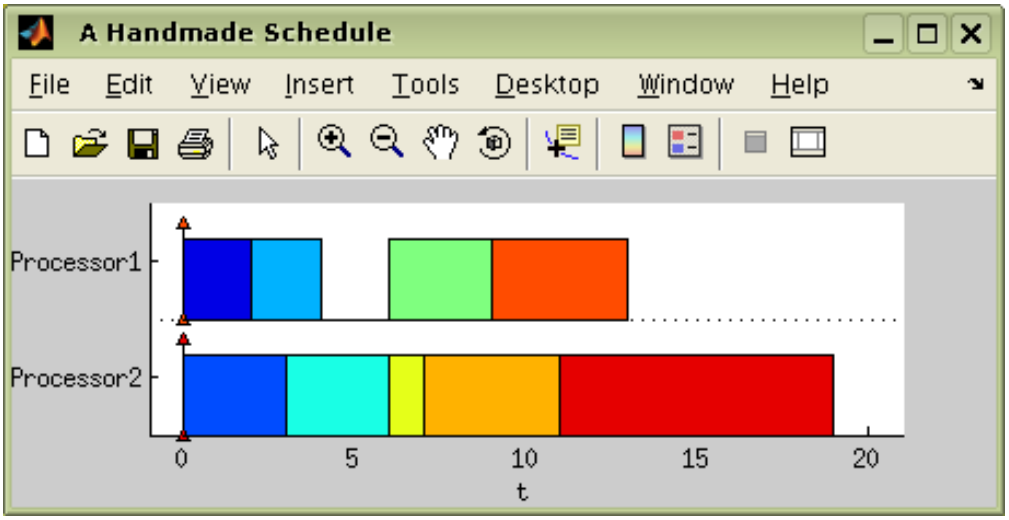
The only way how to operate with schedule of tasks is through commands `add_schedule` and `get_schedule`. Command `add_schedule` inserts a schedule (i.e. start time s_j, number of assigned processor, ...) into taskset object. Its syntax is described in Reference Guide @taskset/add_schedule.m. An example 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 first processor) and string description is a brief note on used scheduling algorithm.

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

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);
```



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 XrefId[??]. 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

Shop

5.1 Introduction

Shop is a special case of scheduling problems. Shop is formed by several jobs (number of jobs is denoted `N`) and each job consists of several tasks. Moreover, each task of the job is to be processed on different processor. In general, jobs could have different number of tasks. We consider 3 basic kinds of shops in literature [Pinedo02] and [Błażewicz01].

- Job-shop - precedence constraints exist in the model - tasks in job form a chain and must be processed consecutive. Moreover tasks in a job can have same dedicated processor.
- Flow-shop - contains same precedence constraints as in previous case, but each job has to be processed on same sequence of processors - all jobs have the same number of tasks.
- Open-shop - formulation is the same as for the Flow-shop except that the order of processing tasks of a job is arbitrary.

This chapter presents the job object in detail first of all. Then shop object is described. Two supporting objects are presented at the end of this chapter. They are used in practical situations, which cannot be modeled by basic shop types. The first supporting object is called `limitedbuffers` and is used in models where the product of some processor is hold off to some buffer for the processor to be able to process other tasks. The second object is named `transportrobots` and is used in models where non-zero time for transfer of product between processors occur.

5.2 `Job` Object

Job object is the basic element of a shop object and consists of several tasks. It can be created using command `job` from either object of type taskset or vectors of processing time and dedicated processor numbers. Variables `j1` and `j2` created in Figure 5.1 are equivalent.

Figure 5.1 Creating of the job object

```
>> procTime = [4 2 3];
>> processor = [1 3 2];
>> t = taskset(procTime);
>> t.Processor = processor;
>> j1 = job(t)
Job with 3 tasks
>> j2 = job(procTime, processor)
Job with 3 tasks
```

Parameters of the job can be retrieved using commands `set` and `get`. Results from these calls are very conformable to results from calling the same method of an object taskset - these two objects are very similar. They differ only in several parameters, methods of these objects are the same and can be used in the same way as is described in Chapter 4, "Sets of Tasks". Different parameters are `Weight`,

ReleaseTime, DueDate and DeadLine, which are scalars representing value for whole job (not vector like in taskset object).

5.3 Creating the shop Object

General shop (object that cover all mentioned kinds of shops) is implemented in the toolbox - distinction between Job-shop, Flow-shop and Open-shop is made by Problem object (described in the Chapter 6, "Classification in Scheduling") not on the level of the object shop. Basic shop can be created in two ways.

- From cell of jobs, syntax for this modality is:

  ```
  S = shop(jobs)
  ```

- Most shop scheduling algorithms work with jobs with the same number of tasks - shop can be also created by calling the following command:

  ```
  S = shop(procTime, processor)
  ```

  Both parameters are a matrix of the same size. Matrix procTime contains processing times of tasks, processors is matrix with numbers of dedicated processors for tasks. Rows of both matrices correspond to jobs.

Both modalities are illustrated in Figure 5.2.

Figure 5.2 Creating of two same shop objects by different way

```
>> procTime1 = [1 2]; procTime2 = [3 1];
>> processor1 = [1 2]; processor2 = [2 1];
>> procTime = [procTime1; procTime2];
>> processor = [processor1; processor2];
>> j1 = job(procTime1, processor1); j2 = job(procTime2, processor2);
>> s1 = shop({j1, j2});
>> s2 = shop(procTime, processor)
Shop with 2 jobs
 Shop type: none
```

5.4 Shop Modification

Basic modification operations are based on the toolbox standard described in Section 2.5. This object also supports modifications of jobs and tasks inside of the shop object.

Figure 5.3 Advanced shop modifications

```
>> s1.Processor(1,:) = 2; %the first job is to be processed on processor 2
>> s2.ProcTime(1,2) = 5;  %the second task of the first job has processing time 5
>> s2.Weight = [2 1];     %modification of job parameters
```

Parameter Type is an enumeration type and represent the character of the shop after computing a schedule. Its value is of type char and is defined from the problem object. The author of an algorithm is responsible for setting of this parameter. It can take the values

- none - default value after creation of the shop object.
- J - Job-shop
- F - Flow-shop
- O - Open-shop

This parameter is used for correct displaying of precedence constraints during plotting the resulting schedule.

Figure 5.4 Definition of shop type

```
>> s1.Type = 'J';
Shop with 2 jobs
 Shop type: J
```

5.5 Other Shop Methods

Schedule of the shop can be displayed by calling method `plot`. Syntax of `plot` is the same as for `taskset` object. The only difference is in one configuration parameter for plot style – parameter `Proc`. In this case value 1 means show all tasks to their dedicated processors; 0 show jobs in same line.

Shop is in general set of many tasks. Very useful method is also `shop2taskset` that convert a shop to a taskset and keeps schedule of shop. This method is irreversible – you cannot convert resulting set of tasks to shop via any inversion method. Syntax is clean from the following example:

Figure 5.5 Conversion to taskset

```
>> t1 = shop2taskset(s1);
Set of 4 tasks
 There are precedence constraints
```

5.6 Limited Buffers

Practical scheduling problems meet with situations when the output product from some processor must be hold off into some buffer because technology process is blocked by any other technology process that uses necessary processor. This situation is in scheduling described by limited buffers [Pinedo02] and [Brucker06].

Creation of this object has the following syntax:

```
LB = limitedbuffers(model, capacity)
```

There are 5 basic models of limited buffers used in the toolbox. Following items describes them. Each model has specified size of input `capacity` matrix. Size of the matrix is described at the end of each item (name of each model is used as a input variable `model`):

- `General` - `Q` central buffers are used by all tasks, each of them has its own capacity. Size: `1` × `Q`
- `Job-dependent` - each job has its own buffer for holding off a intermediate product. Size: `1` × `N`
- `Pair-wise` - there are buffers between all processors. Each of them has its own capacity. Size: `M` × `M`
- `Input` - in the entry of each processor is buffer with specified capacity. Size: `1` × `M`
- `Output` - in the output of each processor is buffer with specified capacity. Size: `1` × `M`

Where `Q` means number of buffers, `N` number of jobs and `M` number of processors.

Following Figure 5.6 presents creation of an output limited buffer and its connection to a shop problem.

5.6.1 Buffers Utilization Chart

The `limitedbuffer` object contains parameter `utilization`. Its default value is an empty matrix. This parameter should be set after computing of a schedule of a system with limited buffers. Setting of `utilization` depends on the buffer type.

- For `General`, `Job-dependent`, `Output` and `Input` models `utilization` should be `a` × `b` matrix where `a` is time spent for schedule and `b` is number of buffers. Element in the matrix on position (a_i , b_j) represents utilization in time `i` for the buffer `j`. Figure 5.7 and Figure 5.8 show using of the set method in this case.

Figure 5.6 Basic operations with limited buffers

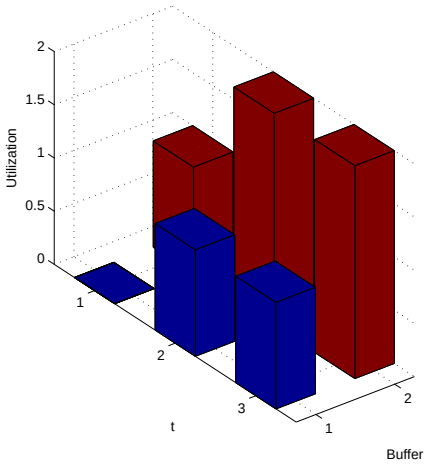
```
>> lb = limitedbuffers('output', [1, 2])
Limited buffers
 Model: output
 Capacity:
     1     2

>> s2.LimitedBuffers = lb
Shop with 2 jobs
 Limited buffers: output
 Shop type: none
```

Figure 5.7 Setting utilization of buffers (output buffer model)

```
>> lb.Utilization = [0 1; 1 2; 1 2];
>> plot(lb)
```

Figure 5.8 An example of buffers utilization chart (output buffer model)



- For `pair-wise` model `utilization` should be `a` × `b` × `c` matrix where `a` is the first processor `b` is the second processor and `c` is a time instant. Element in the matrix on position (a_i , b_j , c_k) represents utilization of the buffer between processors `i` and `j` in time instant `k`. Figure 5.9 and Figure 5.10 show using of the set method in this case.

Figure 5.9 Setting utilization of buffers (pair-wise buffer model)

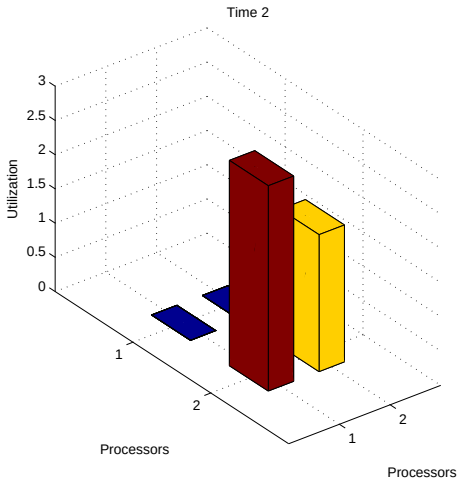
```
>> lb = limitedbuffers('pair-wise',[ 0 3; 3 0]);
>> U(:,:,1)=[1 0; 0 0];
>> U(:,:,2)=[0 0; 3 2];
>> lb.Utilization = U;
>> plot(lb);
```

5.7 Transport Robots

Transport robots are also more practical part of scheduling theory [Pinedo02] and [Brucker06]. We meet this phenomenon in very automated manufactory, for example, in a car factory where robots transports intermediate product between specialized processors.

The object representing this problem can be created:

Figure 5.10 An example of buffers utilization chart (pair-wise buffer model)



`TR = transportrobots(transportationTimes[, emptyMovingTimes])`

where `transportationTimes` is a cell of size `1` × `n` (`n` is number of robots). The cell consists of matrices of size `M` × `M` containing transportation time between each couple of processors. Each matrix is assigned to one robot (`n`-th matrix of transportationTimes cell is assigned to `n`-th robot); `emptyMovingTimes` is also a cell with the same size as `transportationTimes`. It defines time for getting back without load. Using of object transportrobots is shown in the next example.

Figure 5.11 Creating a transport robot

```
>> tr = transportrobots( {[inf 1; 2 inf],[0 inf; inf 0]}, {[inf 1; 1 inf],[0 inf; inf 0]})
Transport robots
 Number : 2
 Transportation times:

TransportationTimes{1} =

   Inf     1
     2   Inf

TransportationTimes{2} =

     0   Inf
   Inf     0

 Empty moving times:

EmptyMovingTimes{1} =

   Inf     1
     1   Inf

EmptyMovingTimes{2} =

     0   Inf
   Inf     0
```

Value `inf` in previous example means that transport robot does not connect specified two processors. This object also provides method `plot` for displaying Gantt charts if schedule is computed. Syntax of `plot` is the same as plot in Chapter 4, "Sets of Tasks".

The transport robot object has a parameter schedule with structure: schedule is a matrix `s` × `5` where `s` means number of actions all robots made. Each row of the matrix contains 5 elements (cols) that describe one event any robot made.

1. From processor
2. To processor
3. Start time of the transportation
4. ID of the transport robot
5. Direction of motion (empty or loaded - binary value)

Figure 5.12 presents creation of schedule for transport robots.

Figure 5.12 Setting a schedule of a transport robot

```
>> scht = [1 2 1 1 1; 2 1 4 1 0; 1 2 0 2 0];
>> tr.Schedule = scht;
>> tr.plot
```

Chapter 6

Classification in Scheduling

6.1 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 7

Graphs

7.1 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 `n` nodes `V`=$\{$`v`$_1$,... , `v`$_n\}$, a set of edges `E`=$\{$`e`$_1$,... , `e`$_m\}$ and relations ε : E$\rightarrow$ V$\times$V. As it is known from definition of graph `G` from graph theory: `G` = (`V`,`E`,ε). All arcs in toolbox are assumed to be directed. This means that an arc connecting node `v`$_i$ to node `v`$_j$ is not the same as an arc connecting node `v`$_j$ to node `v`$_i$. Such graph is called a *directed graph*. When there is no concern about the direction of an edge, the graph is called *undirected* [Diestel00]. Object Graph in the toolbox is described as directed graph. Undirected graph can be created by addition of another identical edge in opposite direction.

7.2 Creating Object `graph`

A graph can be created in various ways 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. Examples of graphs creation is shown in Figure 7.1.

[1] The adjacency matrix $A = (a_{ij})_{n\times n}$ of G is defined by $a_{ij} := |E_{ij}|$ where $E_{ij} = \{e_k \in E : e_k \text{ is an edge from node } v_i \text{ to node } v_j\}$.

[2] The incidency matrix $I = (i_{jk})_{n\times m}$ of G is defined by $i_{jk} := \begin{cases} 1 & \text{if initial node of edge } e_k \text{ is } v_j, \\ -1 & \text{if terminal node of edge } e_k \text{ is } v_j, \\ 0 & \text{otherwise.} \end{cases}$

[3] The matrix of weights $B = (b_{ij})_{n\times n}$ of G is defined by $b_{ij} := \begin{cases} w(e_k) & \text{if } e_k \in E \text{ is an edge from node } v_i \text{ to node } v_j, \\ 0 & \text{otherwise.} \end{cases}$

Where $w(e_k)$ is a weight of edge e_k.

Figure 7.1 Creating graph

```
>> 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],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
```

Another possibility to create the object graph is to use a tool [Graphedit], or transform an object taskset to a corresponding graph (see Section 7.6).

7.3 Object graph Modification

Common modifications are performed as described in section Section 2.5. The rest is discussed in the following section.

New edge can be added to the graph by the function `addedge`.

7.3.1 User Parameters on Edges

Many algorithms (e.g. for cyclic scheduling in Section 10.4) 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 getting/setting data from/to user parameters.

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)
```

Table 7.1 List of functions

function	description
`UserParam = edge2param(g)`	Returns user parameters of edges in graph `g` as a `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`.

and

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

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

7.4 Auxiliary Function

7.4.1 Complete

Function complete makes a complete graph. Makes edges from every node to every node. As an example it is posible to use it, when you need to solve a travelling salesman problem and graph is not complete. A simple example is shown in Figure 7.3 and Figure 7.4. A call example is shown inFigure 7.2.

```
gcomplete = complete(g[,preserveedge])
```

g
input objeck of type graph.

preserveedge
input parameter is a list of edges that remain.

gcomplete
output object of type graf is a input graph with a new edges.

Figure 7.2 Complete function call example

```
>> edgeList = {1 2 1; 2 3 1; 2 4 2; 3 4 5};
>> g = graph('edl',edgeList,'edgeDatatype',{'double'});
>> gcomplete = complete(g,[1 2 3 4])
```

7.4.2 Distance

This function completes graph by the weights of edges. This weights of edges are computed from the x and y possition both of nodes, which edge connects. Is possible to use it only for graphs, which were made in graphedit. A simple example is shown in Figure 7.7 and Figure 7.8. A call example is shown inFigure 7.6.

```
g = distance(g[,userparamposition,preserveuserparam])
```

g
input objeck of type graph

Figure 7.3 An example of complete function use - original graph .

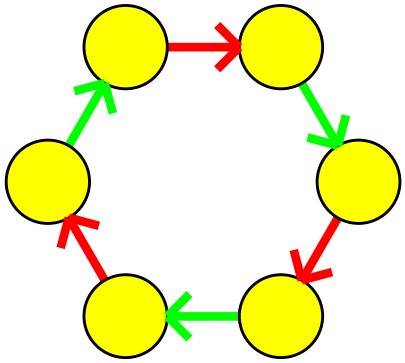
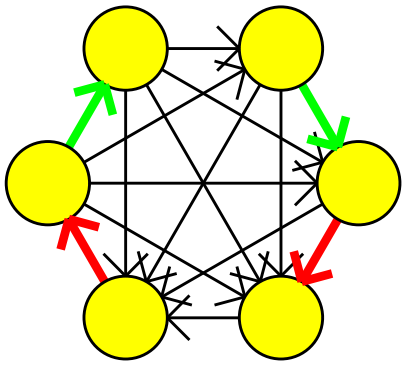


Figure 7.4 An example of complete function use - complete graph .



userparamposition
input parameter represents the position of the weight parameters in edgeList

preserveuserparam
input parametes is a list of parameters that remain.

g
output object type of graph

Figure 7.5 An example of complete function use

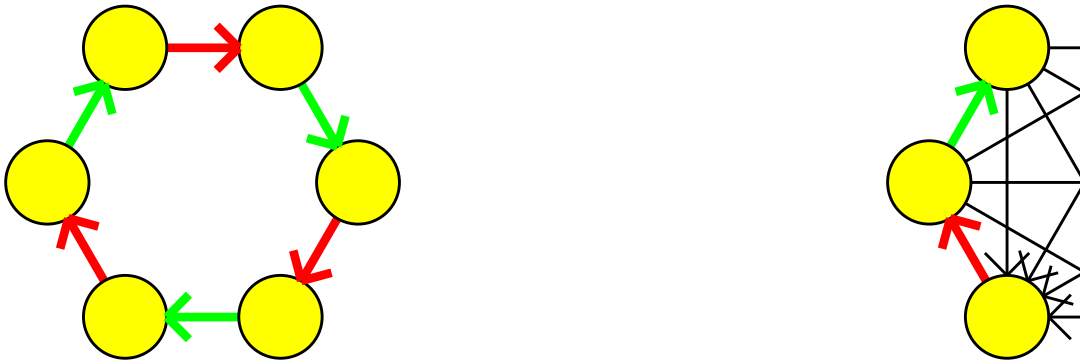


Figure 7.6 Distance function call example

```
>> g = graph('adj',[0 1 1;0 0 1;0 0 0],'name','graph');
>> position = 100+40*[0 0; 3 0; 3 4];
>> for i = 1:length(g.N)
>> g.N(i).GraphicParam(1).x = position(i,1);
>> g.N(i).GraphicParam(1).y = position(i,2);
>> end
>> g.E(3).UserParam(1)=180;
>> gd=distance(g,1,3);
>> gd.Name = 'distance gr.';
>> graphedit(gd)
```

7.5 Graphedit

The toolbox is equipped with a simple but useful editor 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 7.9 . 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 parameters, 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.

Figure 7.7 An example of distance function use - original graph .

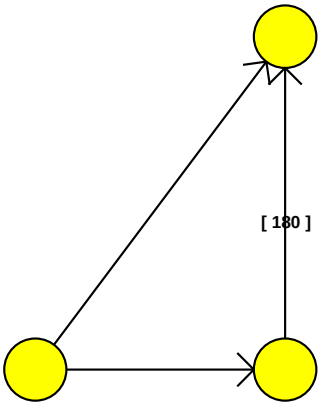
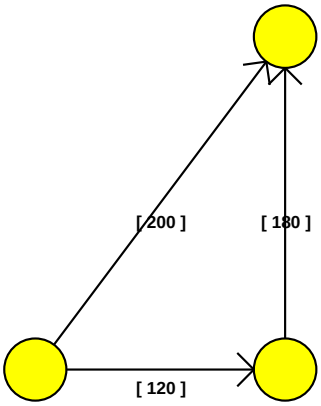


Figure 7.8 An example of distance function use - distance graph .



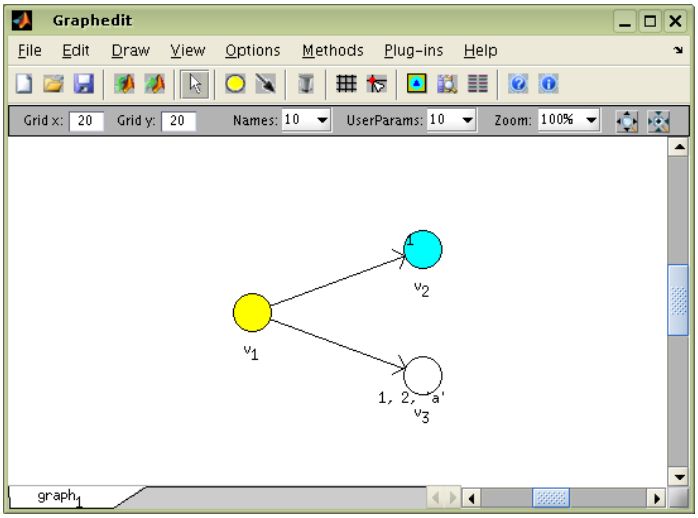
7.5.1 The Graph Construction

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 7.9) 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. Properties of nodes and edges can be edited in **Edit** mode.

Graphedit also offers a possibility to choice appearance of a node. It also contains a tool for design of

your own node by a picture which can be a bitmapped image or a geometric pattern or their arbitrary combination. This function has nothing to do with graph theory; however it is useful for presentation purposes (see below).

Figure 7.9 Graphedit



7.5.1.1 Placing of Nodes and Edges

To place a node or an edg just select an appropriate drawing mode and add it using your 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.

7.5.2 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 drawn, 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'.

7.5.3 Property editor

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

7.5.4 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 enter a name of variable to which will be current graph saved. The same pays for importing a graph object from the workspace.

7.5.5 Saving/Loading to/from Binary File

Saving and loading of the graph is performed in the same way as exporting and importing. In addition, a filename must be chosen.

7.5.6 Change of Appearance of Nodes

Graphedit also offers a possibility to choice appearance of a node. It contains a tool for design of your own node by picture. The picture can be a bitmapped image or a 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.

7.6 Transformations Between Objects `taskset` and `graph`

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

7.6.1 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.

7.6.2 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 8

Scheduling Algorithms

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

8.1 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 8.1. First of all, the algorithm must check whether the input scheduling problem can be solved by itself. In this case the function `is` is used as 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 assignement (`processor`) - if required. Finally, the resulting schedule is derived from the original taskset using function `add_schedule`.

8.2 Scheduling on One Processor

8.2.1 List of Algorithms

Table 8.1 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 8.1 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 8.1 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]
[Horn's Algorithm] (see Chapter Quick Start)	horn	1|pmtn,rj|Lmax	[Horn74], [Błażewicz01]
[Hodgson's Algorithm]	alg1sumuj	1||$\sum$ Uj	[Błażewicz01]
[Algorithm for 1||$\sum$ (wj.Dj)]	alg1sumwjdj	1||$\sum$ wj.Dj	[Błażewicz01]

8.2.2 Algorithm for Problem 1|rj|Cmax

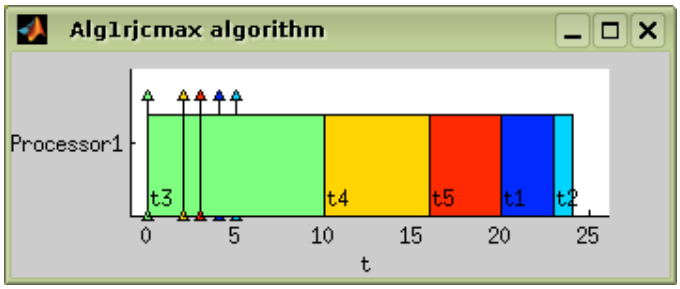
This algorithm solves 1|r_j|Cmax scheduling problem. Tha basic idea of the algorithm is to arrange and schedule the tasks in order of nondecreasing release time r_j. It is equivalent to the First Come First Served rule (FCFS). The algorithm usage is outlined in Figure 8.2 and the corresponding schedule is displayed in Figure 8.3 as a Gantt chart.

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

Figure 8.2 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 8.3 Alg1rjcmax algorithm - problem 1|r_j|Cmax



8.2.3 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 availability of schedule after adding each task. For more details about Bratley's algorithm see [Błażewicz01].

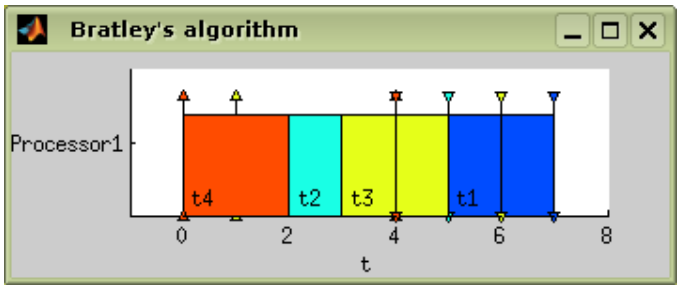
The algorithm usage is shown in Figure 8.4 and the resulting schedule is shown in Figure 8.5.

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

Figure 8.4 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 8.5 Bratley's algorithm - problem 1|r_j,~d_j|Cmax



8.2.4 Hodgson's Algorithm

Hodgson's algorithm is proposed to solve 1||$\sum$Uj problem, i.e. the algorithm minimalize number of delayed tasks on monoprocessor. 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).

The algorithm applies EDD (Earliest Due Date First) rule on taskset `T`. Tasks `ti` ∈ `T` are added successively to `Ts`. If `ti` is completed after its due date then a task in `Ts` with the longest processing time is marked to be late and removed from `Ts`. Finally, schedule is qiven by [`Ts Tn`] where `Ts` is scheduled according to EDD rule and `Tn` is scheduled in an arbitrary order. For more details about Hodgson's algorithm see [Błażewicz01].

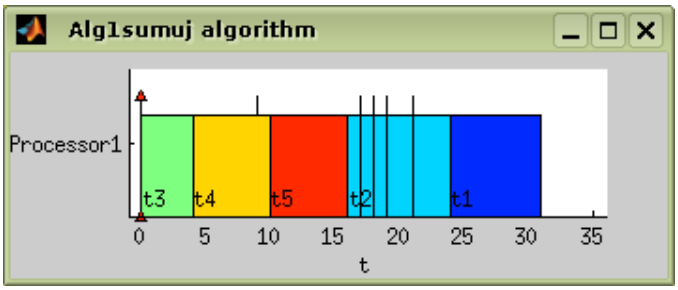
The algorithm usage is outlined in Figure 8.6 and the resulting schedule is displayed in Figure 8.7.

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

Figure 8.6 Scheduling problem 1||∑ 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 8.7 Hodgson's algorithm - problem 1||∑ Uj



8.2.5 Algorithm for Problem 1||∑ (wj.Dj)

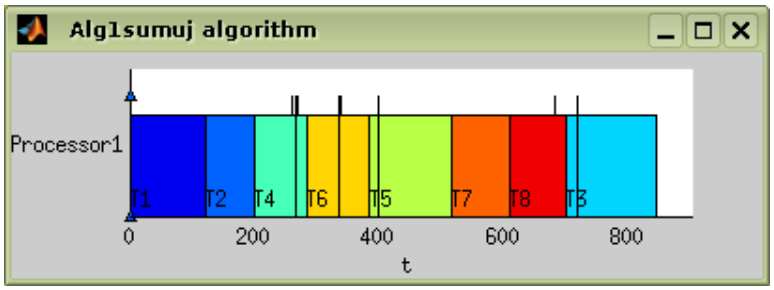
An algorithm solving mean tardiness problem on a single processor is described in [Błażewicz01]. The algorithm considers that processing times are integer and tasks are so called *agreeable*, i.e. $p_i < p_j$ implies $w_i >= w_j$ for all i, j ∈ {1, ... , n}. The algorithm is based on dynamic programming and its time complexity is $O(n^4 p)$ where $p = \sum p_i$.

In the toolbox the algorithm is implemented as `alg1sumwjdj` function. Its usage is outlined in Figure 8.8 and the resulting schedule of the example is displayed in Figure 8.9.

Figure 8.8 A solution of 1||∑ (wj.Dj) scheduling problem

```
>> T=taskset([121 79 147 83 130 102 96 88]);
>> T.DueDate = [260 266 269 336 337 400 683 719];
>> T.Weight = [3 8 1 6 3 3 5 6];
>> T.Name={'T1','T2','T3','T4','T5','T6','T7','T8'};
>> p = problem('1||sumwjDj');
>> TS = alg1sumwjdj(T,p);
>> plot(TS);
```

Figure 8.9 Algorithm alg1sumwjdj - problem 1||∑ (wj.Dj)



8.2.6 Brucker's Algorithm

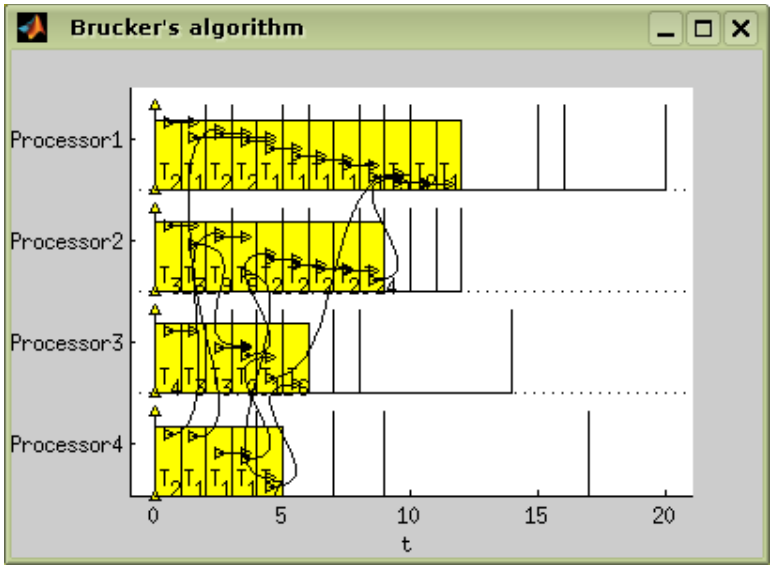
Brucker's algorithm, proposed to solve 1|in-tree,p_j=1|Lmax problem, is an algorithm which can be implemented in O(n.log n) time [Bru76][Błażewicz01]. Implementation in the toolbox use listscheduling algorithm while tasks are sorted in non-increasing order by their modified due dates subject to precedence 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 XrefId[??].

Examples in Figure 8.10 and Figure 8.11 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 8.10 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);
```

Figure 8.11 Brucker's algorithm - problem 1|in-tree,p_j=1|Lmax



8.3 Scheduling on Parallel Processors

8.3.1 List of Algorithms

Table 8.2 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.

8.3.2 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 8.12 and resulting schedule is displayed in Figure 8.13.

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

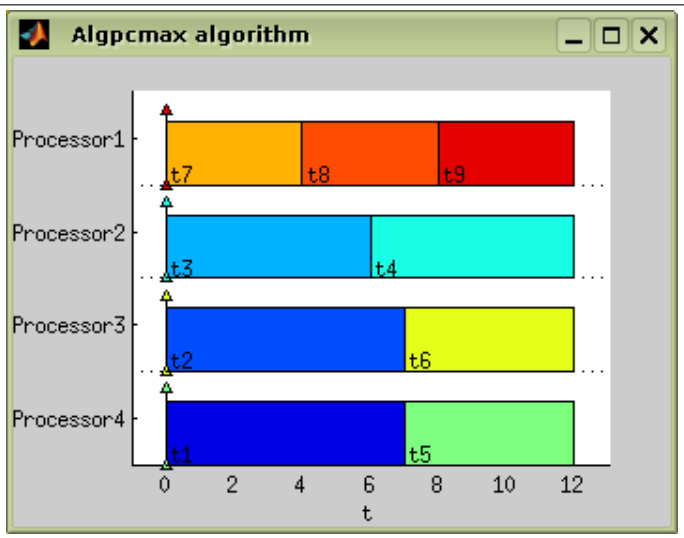
Table 8.2 List of algorithms

algorithm	command	problem	reference
[Algorithm for P||Cmax]	algpcmax	P||Cmax	[Błażewicz01]
[Dynamic Prog. and P||Cmax]	algpcmaxdp	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]
[List Scheduling]	listsch	P|prec|Cmax	[Graham66], [Błażewicz01]
[Coffman's and Graham's Algorithm]	coffmangraham	P2|prec,pj=1|Cmax	[Błażewicz01]
[SAT Scheduling]	satsch	P|prec|Cmax	[TORSCHE06]

Figure 8.12 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 8.13 Algpcmax algorithm - problem P||Cmax



8.3.3 Problem P||Cmax Solved by Dynamic Programming

This algorithm, unlike the previous one, is based on dynamic programming [Błażewicz01]. The algorithm uses full search of solutions space represented as `m`-dimesional matrix, where `m` is a number of processors. The size of each dimesion is the same and is equel to `C` - an upper bound or an estimation of C_{max}. The algorithm recursively search the solutions space while possibe solutions are marked with TRUE(one) and all unfeasible solutions are marked with FALSE(0). When the optimal C_{max} is found the backward search finds the optimal solution.

8.3.4 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

Figure 8.14 Scheduling problem P||Cmax solving by an algorithm using dynamic programming.

```
>> T=taskset([2 3 1 2 4]);
>> T.Name={'t1' 't2' 't3' 't4' 't5'};
>> p = problem('P||Cmax');
>> TS = algpcmaxdp(T,p,2);
>> plot(TS);
```

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].

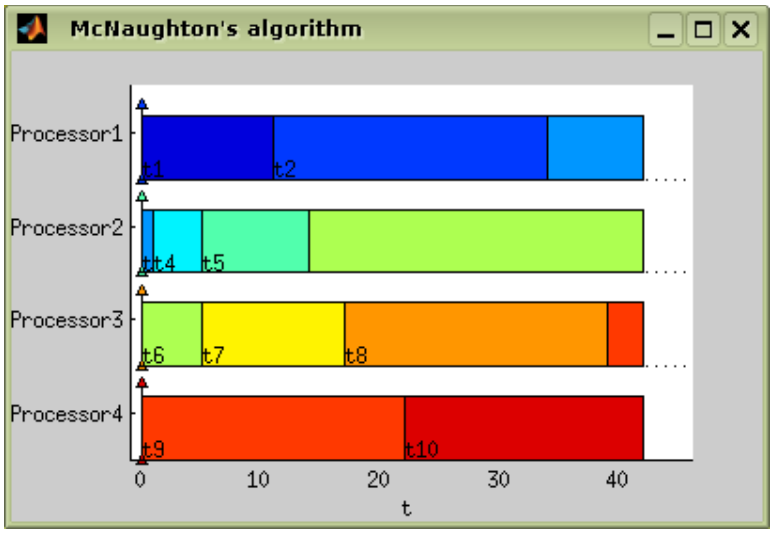
The algorithm usage is outlined in Figure 8.15. The resulting Gantt chart is shown in Figure 8.16.

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

Figure 8.15 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 8.16 McNaughton's algorithm - problem P|pmtn|Cmax



8.3.5 Algorithm for 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 8.17 the algorithm usage is shown. The resulting Gantt chart is displayed in Figure 8.18.

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

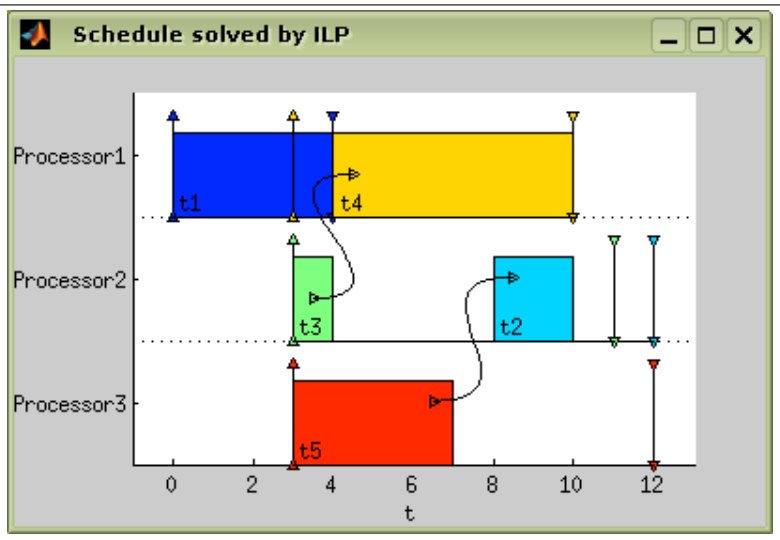
8.3.6 List Scheduling

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

Figure 8.17 Scheduling problem P|r_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,~dj|Cmax');
>> TS = algprjdeadlinepreccmax(T,prob,3);
>> plot(TS);
```

Figure 8.18 Algprjdeadlinepreccmax algorithm - problem P|r_j,prec,~d_j|Cmax



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 of the optimal solution. A subset of heuristic algorithms constitutes 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 C_{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 C_{max}(A(I)) is C_{max} obtained by approximation algorithm A, C_{max}(OPT(I)) is C_{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 R_{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 8.3.1 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 8.19.

Figure 8.19 An example of P|prec|Cmax scheduling problem.

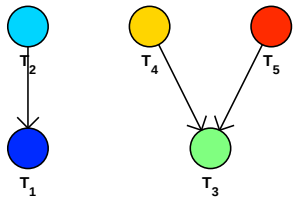


Figure 8.20 Scheduling problem P|prec|Cmax solving.

```
>> 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);
```

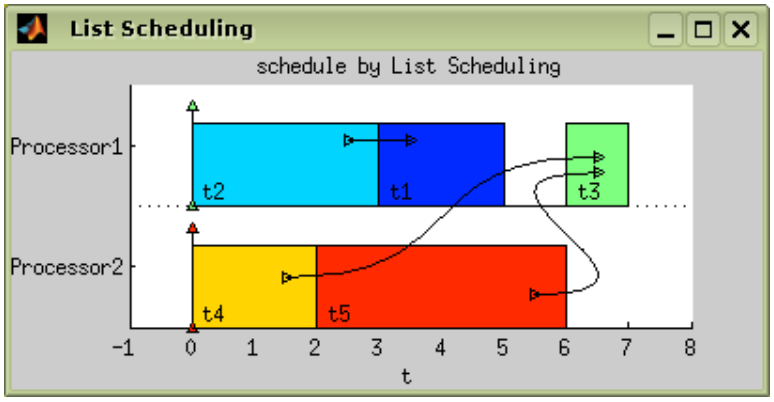
The solution of the example is shown in Figure 8.21. The LS algorithm found a schedule with C_{max}= 7.

8.3.6.1 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 non-increasing order of processing time p_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.

Figure 8.21 Result of List Scheduling.




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

LS algorithm with LPT strategy demonstrated on the example from previous paragraph is shown in Figure 8.22. The resulting schedule with C_{max}= 7 is in Figure 8.23.

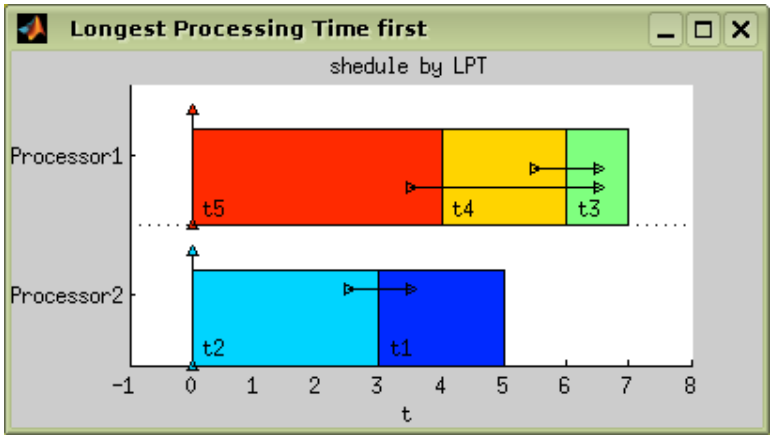
Figure 8.22 Problem P|prec|Cmax by LS algorithm with LPT strategy solving.

```
>> 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);
```

Figure 8.23 Result of LS algorithm with LPT strategy.



8.3.6.2 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 non-decreasing order of processing time p_j before the application of List Scheduling algorithm. The time complexity of SPT is also $O(n \cdot log(n))$ [Błażewicz01]

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 8.19 is shown in Figure 8.24. The resulting schedule with C_{max}= 7 is in Figure 8.25.

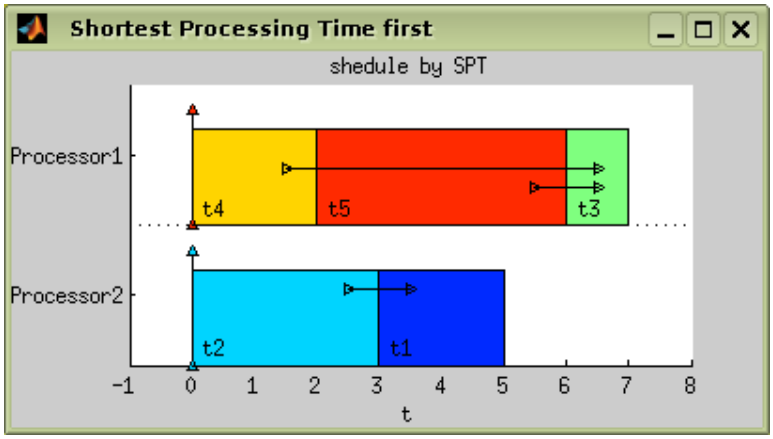
Figure 8.24 Solving P|prec|Cmax by LS algorithm with SPT strategy.

```
>> 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);
```

Figure 8.25 Result of LS algorithm with SPT strategy.



8.3.6.3 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 nondecreasing order of 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 an 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 8.3. The schedule obtained by ECT strategy with $\Sigma C_j = 58$ is shown in Figure 8.29.

Table 8.3 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

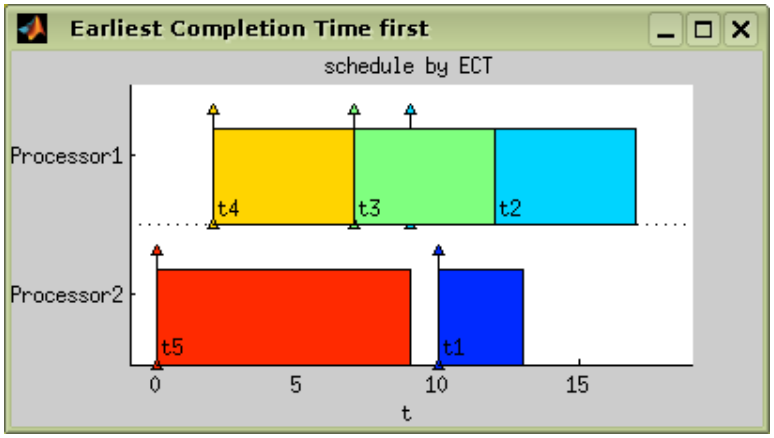
Figure 8.26 Solving P|rj|ΣCj by ECT

```
>> 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 8.27 Result of LS algorithm with ECT strategy.



8.3.6.4 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 nondecreasing order of release time 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 8.19 is shown in Figure 8.28. The resulting schedule with $\Sigma C_j = 57$ is in Figure 8.29.

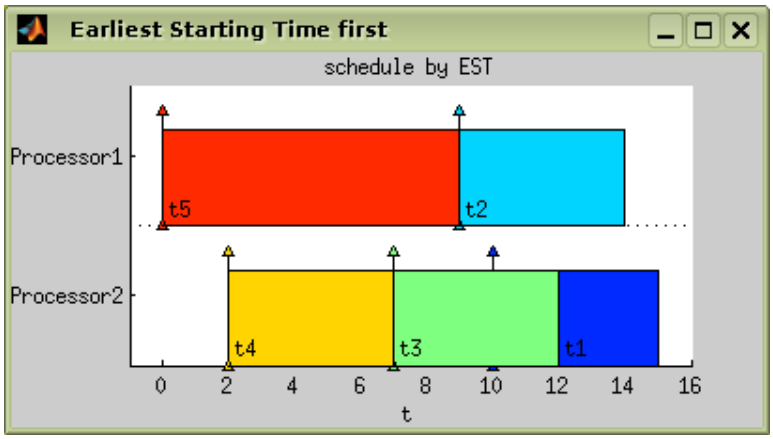
Figure 8.28 Problem P|rj|ΣCj by LS algorithm with EST strategy solving.

```
>> 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 8.29 Result of LS algorithm with EST strategy.



8.3.6.5 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.

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`.

Figure 8.30 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
```

8.3.7 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.

8.3.7.1 Instalation

First of all you have to install zChaff SAT solver as follows:

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 downloaded 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.

8.3.7.2 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 t_i is started at time unit `j` on the processor `k` then $x_{ijk} = true$, otherwise $x_{ijk} = false$. For each task 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 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 t_u is the predecessor of t_v, then t_v cannot start before the execution of 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 p_u denotes the processing time of task t_u.

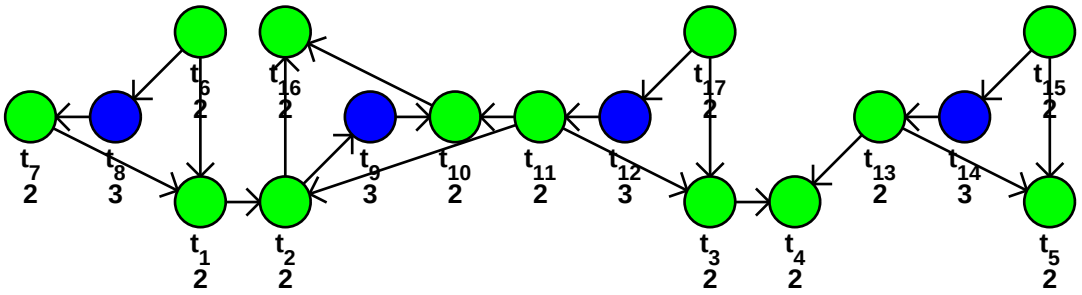
3. At any time unit, there is at most one task executed on a given processor. For the couple of tasks with a precedence constrain this rule is ensured already by the clauses in the rule number 2. Otherwise the set of clauses is generated for each processor `k` and each time unit `j` for all couples $\mathtt{t_u}$, $\mathtt{t_v}$ without precedence constrains in the following form: $(x_{ujk} \rightarrow \bar{x}_{vjk}) \wedge (x_{ujk} \rightarrow \bar{x}_{v(j+1)k}) \wedge \cdots \wedge (x_{ujk} \rightarrow \bar{x}_{v(j+p_u-1)k})$

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.

8.3.7.3 Example - Jaumann wave digital filter

As an example we show a computation loop of Jaumann wave digital filter. Our goal is to minimize computation time of the filter loop, shown as directed acyclic graph in Figure 8.31. 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 with the task name and the corresponding processing time $\mathtt{p_i}$. We look for an optimal schedule on two parallel identical processors.

Figure 8.31 Jaumann wave digital filter

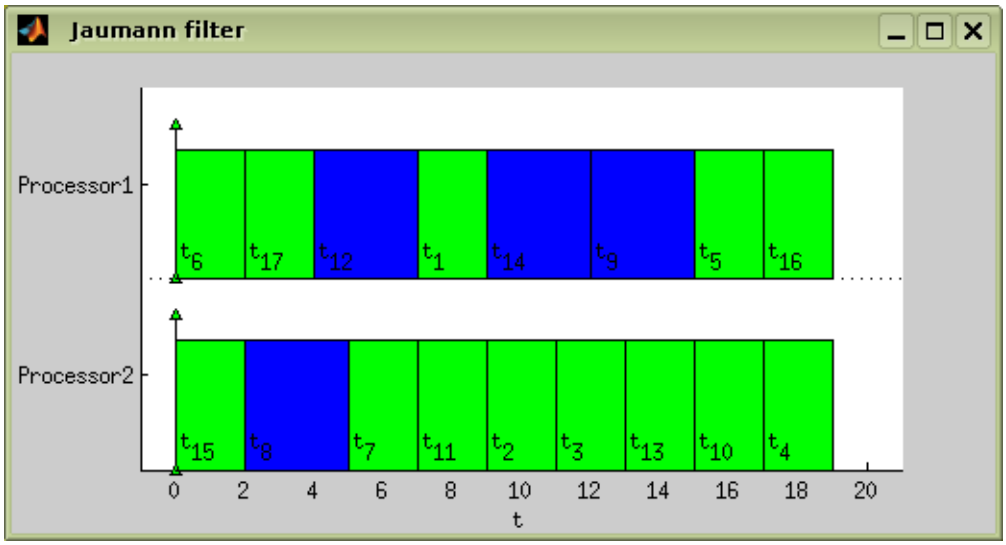


Following code shows consecutive steps performed within the toolbox. First, we define the set of task with precedence constraints and then we run the scheduling algorithm `satsch`. Finally we plot the Gantt chart.

```
>> 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],...
[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 8.32.

Figure 8.32 The optimal schedule of Jaumann filter



8.3.8 Hu's Algorithm

Hu's algorithm is intended to schedule unit length tasks with in-tree precedence constraints. The problem notation 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. Time complexity of the algorithm 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 8.3.2 Hu's algorithm

Figure 8.33 shows a problem with 12 tasks (p_j=1) with precedence constraints forming an in-tree. A problem solution using Hu's algorithm is shown in Figure 8.34.

8.3.9 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 $O(n^2)$. The algorithm is based on a label assign to each task. Labels 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])
```

Figure 8.33 An example of in-tree precedence constraints

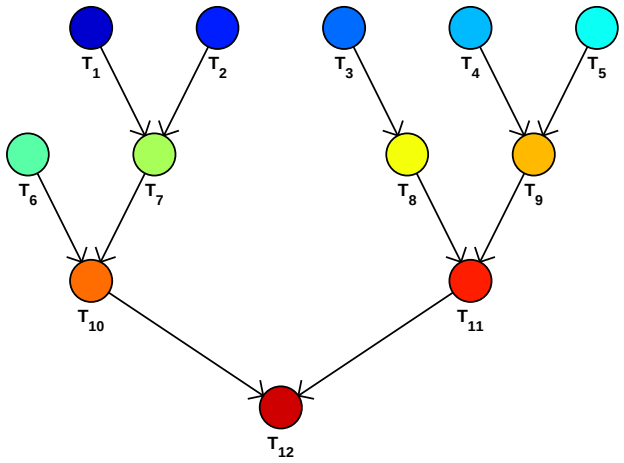


Figure 8.34 Scheduling problem P|in-tree,p_j=1|Cmax using `hu` command

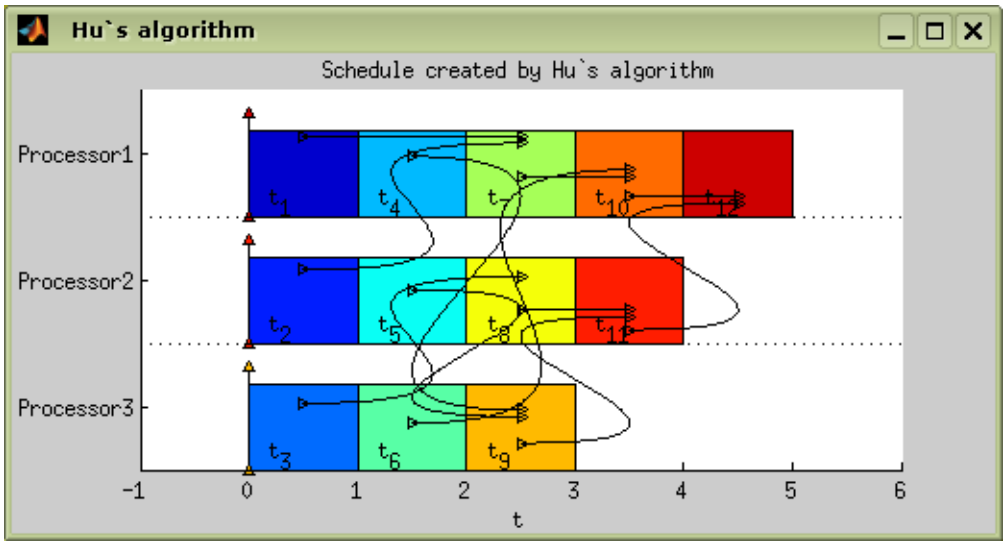
```
>> t1=task('t1',1);
>> t2=task('t2',1);
>> t3=task('t3',1);
>> t4=task('t4',1);
>> t5=task('t5',1);
>> t6=task('t6',1);
>> t7=task('t7',1);
>> t8=task('t8',1);
>> t9=task('t9',1);
>> t10=task('t10',1);
>> t11=task('t11',1);
>> t12=task('t12',1);

>> 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);
```

schoptions

Figure 8.35 Hu's algorithm example solution



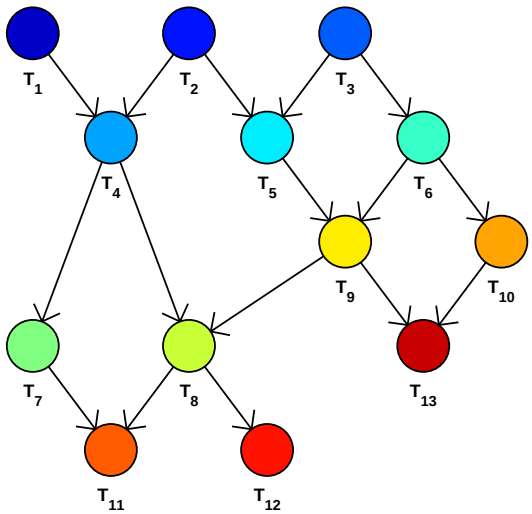
optimization options

More about Coffman and Graham algorithm in [Błażewicz01].

Example 8.3.3 Coffman and Graham algorithm

The set of tasks contains 13 tasks constrained by precedence constraints as shown in Figure 8.36.

Figure 8.36 Coffman and Graham example setting



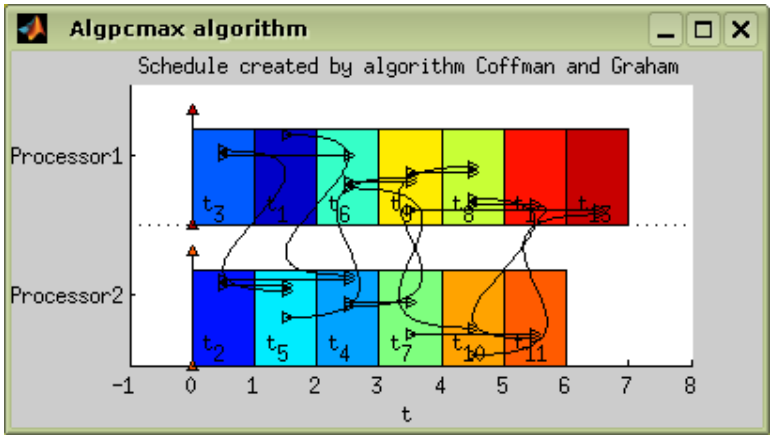

```
>> t1 = task('t1',1);
>> t2 = task('t2',1);
>> t3 = task('t3',1);
>> t4 = task('t4',1);
>> t5 = task('t5',1);
>> t6 = task('t6',1);
>> t7 = task('t7',1);
>> t8 = task('t8',1);
```

```
>> t9 = task('t9',1);
>> t10 = task('t10',1);
>> t11 = task('t11',1);
>> t12 = task('t12',1);
>> 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 8.37 Coffman and Graham algorithm example solution



8.4 Scheduling in Shops

8.4.1 List of Algorithms

Table 8.4 shows reference for all 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.

8.4.2 Johnson's Algorithm

Johnson's algorithm is one of the most classical shop scheduling algorithm. It solves problems F2||Cmax (Flow-shop with two processors, optimizing max completition time) with time complexity O(nlogn), because it is in fact a sorting procedure as shows following Figure 8.38.

Table 8.4 List of algorithms

algorithm	command	problem	reference
[Johnson's Algorithm]	johnson	F2||Cmax	[Johnson54][Błażewicz01]
[Gonzales Sahni's Algorithm]	gonzalezsahni	O2||Cmax	[GS76]
[Jackson's Algorithm]	jackson	J2|nj<=2|Cmax	[Błażewicz01]
[Algorithm cpshopscheduler]	cpshopscheduler	J||Cmax, F||Cmax, O||Cmax	[Baptiste01]
[Algorithm for Problem F2,R1—pij=1,tj—Cmax]	algf2r1pijtjcmax	F2,R1|pij=1,tj|Cmax	[Brucker04]
[Algorithm for Problem F——Cmax with Limited Buffers]	fslb	F||Cmax + limited buffers	
[Algorithm for Problem O—pij=1—∑ Ti]	algopij1sumti	O|pi=1|SumTi	[Brucker04]

Figure 8.38 Example of implementation of Johnson's algorithm

```
begin
% create subset S1 which contain jobs Ji with pi1 <= pi2 in a sequence
  of non-decreasing order of their processing times pi1
% create subset S2 which contain remaining jobs in a sequence
  of non-decreasing order of their processing times pi2
% concatenate all jobs on both machines in sequence order S1,S2
  to get the optimal schedule
end
```

```
S = johnson(S, problem)
```

Details of this algorithm describes [Johnson54].

Figure 8.39 Scheduling problem F2||Cmax solved by Johnson's algorithm

```
>> processor = [1 2; 1 2; 1 2; 1 2];
>> procTime = [7 3; 5 4; 6 6; 8 6];
>> s = shop(procTime, processor);
>> pr = problem('F2||Cmax');
>> s = johnson(s, pr);
>> plot(s);
```

8.4.3 Gonzales Sahni's Algorithm

This algorithm solves problems O2||Cmax (Open-shop, minimizing makespan) with time complexity O(n). Algorithm proceeds by dividing the jobs into two groups `A` and `B` such that `A` contains jobs J_i with $p_{i1} >= p_{i2}$ and `B` the rest of all jobs. The schedule is then built from the middle, with jobs from `A` added on at the right and those from `B` at the left. The schedule from the jobs in `A` is such that there is no idle time on processor 1 (except at the end), the part of the schedule made up with jobs in `B` is such that the only idle time on processor 2 is at the beginning.

Its details are described in [GS76].

```
S = gonzalezsahni(S, problem)
```

8.4.4 Jackson's Algorithm

Only few cases of Job-shop scheduling problems can be solved in polynomial time. Jackson's algorithm solves J2|nj<=2|Cmax problem (2 processors, at max two tasks in job, optimizing makespan). Time

Figure 8.40 Johnson's algorithm - example solution

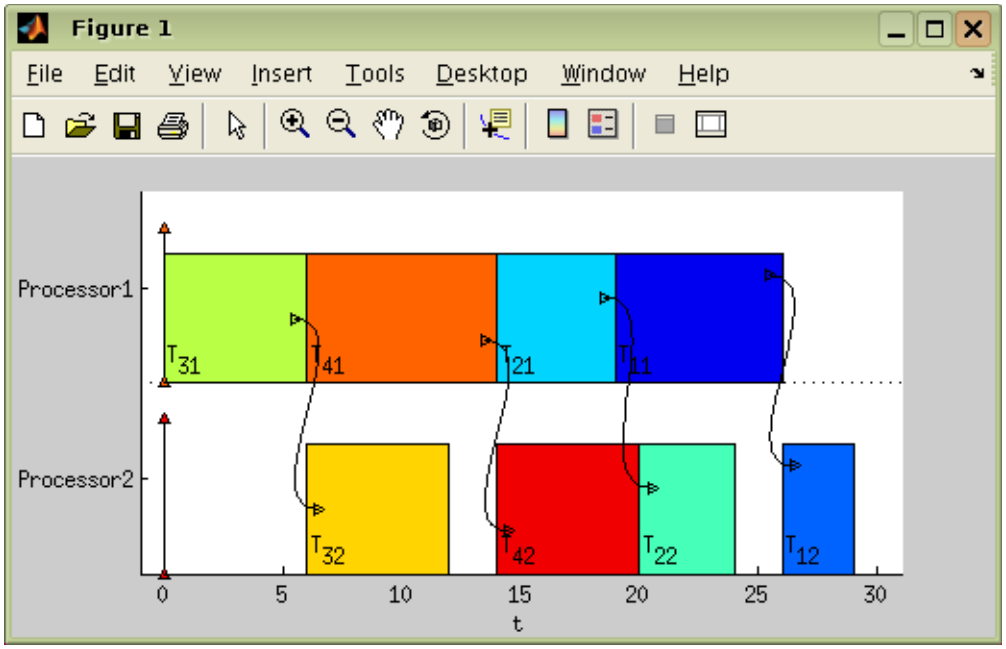
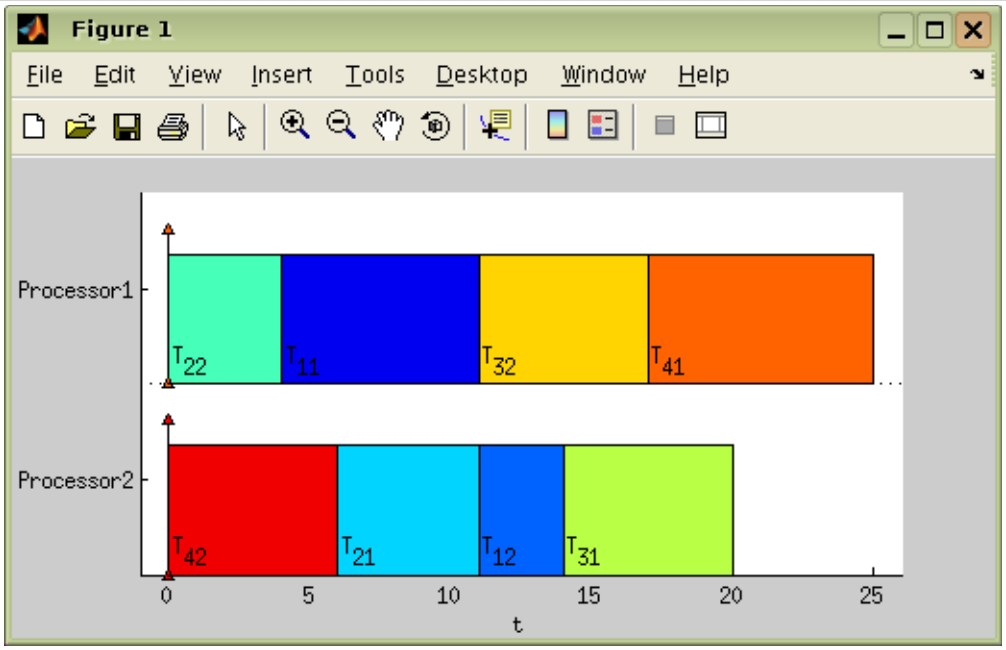


Figure 8.41 Scheduling problem O2||Cmax solved by Gonzales Sahni's algorithm

```
>> processor = [1 2; 2 1; 2 1; 1 2];
>> procTime = [7 3; 5 4; 6 6; 8 6];
>> s = shop(procTime, processor);
>> pr = problem('O2||Cmax');
>> s = gonzalezsahni(s, pr);
>> plot(s);
```

Figure 8.42 Gonzales Sahni's algorithm example solution



complexity of the algorithm is O(n log n). The algorithm divides jobs into 4 groups: J_1, J_2, J_{12}, J_{21} (indexes means dedicated processors – J_i job consists of 1 task to be performed on processor `i`, J_{ij} job has 2 tasks, the first to be processed on the processor `i`, the second on the processor `j`) [Błażewicz01].

```
S = jackson(S, problem)
```

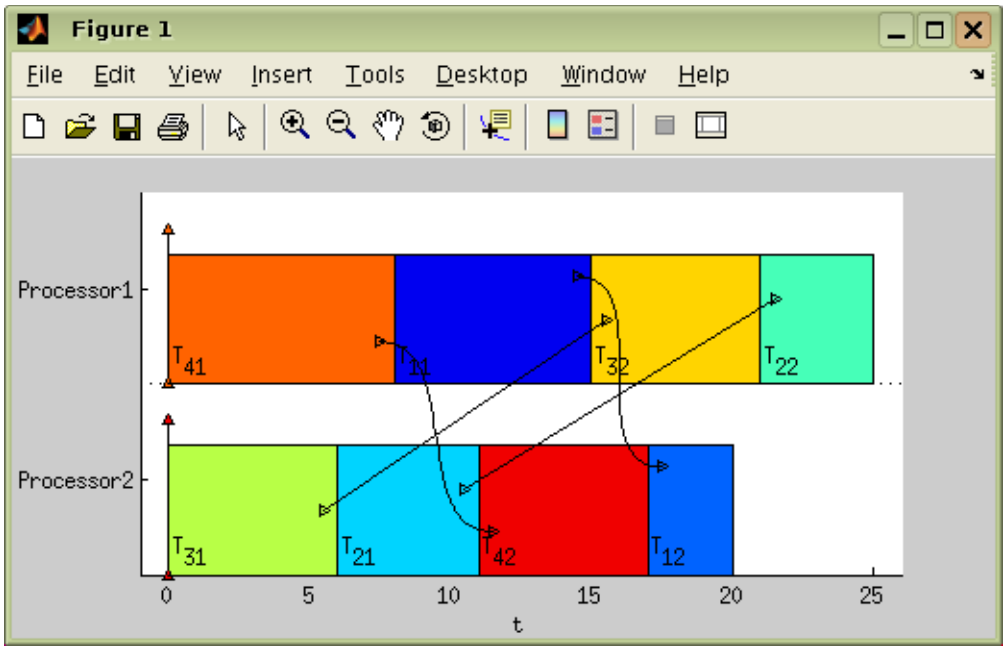
Figure 8.43 Example of implementation of Jackson's algorithm

```
begin
%create subsets J1, J2, J12, J21 (as shop)
>>J12 = johnson(J12,problem('F2||Cmax'));
>>J21 = johnson(J21,problem('F2||Cmax'));
%Assign jobs to machine 1 in order: J12, J1, J21
%Assign jobs to machine 2 in order: J21, J2, J12
end
```

Figure 8.44 Scheduling problem J2|nj<=2|Cmax using of Jackson's algorithm

```
>> processor = [1 2; 2 1; 2 1; 1 2];
>> procTime = [7 3; 5 4; 6 6; 8 6];
>> s = shop(procTime, processor);
>> pr = problem('J2|nj<=2|Cmax');
>> s = jackson(s, pr);
>> plot(s);
```

Figure 8.45 Jackson's algorithm example solution



8.4.5 Algorithm for Problem J|nj=n|Cmax

Function cpshopscheduler was developed to solve NP-hard shop problems (previous polynomial algorithms solved only small class of shop problems) using constraint programming technique. It solves all 3 basic shop problem Flow-shop, Open-shop and Job-shop with same number of tasks in all jobs. This algorithm involves several interesting features described in [Baptiste01]:

- Edge-finding algorithm finds precedence between a task and group of tasks on one processor system.
- Not-First/Not-Last algorithm does oposite work - it filters domains such that some tasks cannot be first (their release time must be higher than earliest finish time of other tasks) or last.

- Detectable precedences algorithm is also filtering algorithm with low complexity $O(n^2)$ (also in [Vilim07]).

Time complexity of the algorithm is exponential. Shop problem e.g. 7 jobs and 7 tasks in job is solved in rank of seconds. Internal algorithm uses a heuristic method to process input tasks from jobs. Tasks are sorted in non-ascending order of processing time, then constraint solver does its work.

```
S = cpshopscheduler(S, problem)
```

Constraint programming library Gecode <http://www.gecode.org> is used as a kernel for this algorithm.

NOTE

You should have installed Gecode at version 2.1.0 on you PC which is available for both Linux and MS-Windows platforms from Gecode download web page <http://www.gecode.org/download.html>.

8.4.6 Algorithm for Problem F2,R1|pij=1,tj|Cmax

The aim of this algorithm is to schedule the shop with jobs of two tasks with processing time equal to 1. Each task of each job is proceeded on different processor. Products are transported from processor 1 to processor 2 by one transport robot. The transportation time can be different for each job and has to be grater or equal to zero.

The scheduling algorithm works in three steps. First of all jobs with non-zero transportation time are scheduled so that the transport robot has no gaps in its schedule. The first task of the job is then at the first processor just before the transportation and the second task is placed at the second processor just after the transportation. Jobs with zero transportation time are scheduled in next step by placing into gaps in schedules of the first processor, the robot and the second processor. Finally tasks at processor 2 are ordered in the same way as at processor 1 according to jobs.

```
S = algf2r1pijtjcmax(S, problem)
```

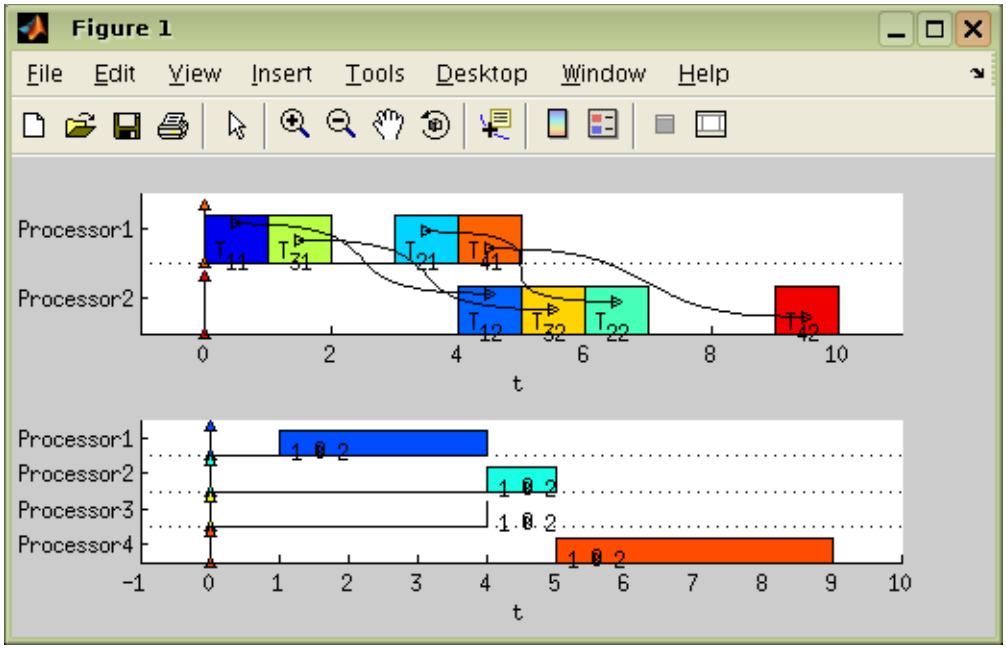
Figure 8.46 Scheduling problem F2,R1|pij=1,tj|Cmax

```
>> processor = [1 2; 1 2; 1 2; 1 2];      % processor dedication
>> procTime = [1 1; 1 1; 1 1; 1 1];       % task processing time
>> tr = transportrobots({[inf 3; inf inf], [inf 1; inf inf], [inf 0; inf inf], [inf 4; inf inf]});
                                          % transportrobots object creation
>> inputShop = shop(procTime, processor); % shop object creation
>> inputShop.Type = 'F';                  % shop type = Flow-shop
>> inputShop.TransportRobots = tr;        % adding transport robots to the shop
>> inputProblem = problem('F2|pj=1|Cmax');
>> scheduledShop = algf2r1pijtjcmax(inputShop, inputProblem);
>> subplot(2,1,1);
>> plot(scheduledShop);                   % show gantt chart of the shop and transport robots
>> subplot(2,1,2);
>> scheduledShop.TransportRobots.plot();
```

8.4.7 Algorithm for Problem F||Cmax with Limited Buffers

This algorithm is based on a branch and bound searching method with pruning. In each node (in the searching state space) average between actual utilization of all processors+sum of all unscheduled tasks is computed. If the resulting value is larger than actual Cmax, then the node is pruned. As inputs function needs a matrix describing the flow shop and limited buffers object of type input (it needs vector with maximum capacity of buffers `buff_max`). Number of buffers must be the same as number of tasks in each job.

Figure 8.47 Optimal solution for example of algf2r1pijtjcmax algorithm




```
S = fslb(S, problem)
```

Figure 8.48 Scheduling problem F||Cmax with limited buffers

```
>> procTime = [1 2 5 12; 1 2 5 12; 1 2 5 12; 1 2 5 12];
>> processor = [1 2 3 4; 1 2 3 4; 1 2 3 4; 1 2 3 4];
>> Bf = [1 1 1 1];
>> s = shop(procTime, processor);   % Create shop object
>> l = limitedbuffers('input',Bf);  % Create limitedBuffers object
>> s.LimitedBuffers = l;
>> pr = problem('F||Cmax')
>> fs = fslb(s,p);
>> plot(fs)                         % plot resulting schedule for jobs
>> fs.limitedBuffers.plot           % plot utilization of buffers
```

8.4.8 Algorithm for Problem O|pij=1|$\sum$Ti

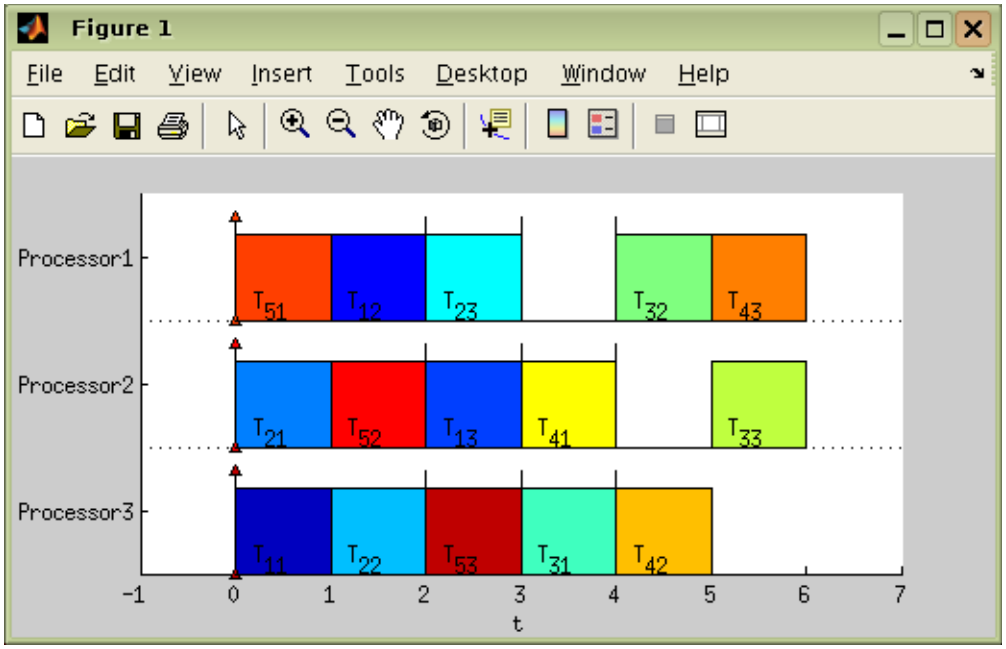
The algorithm proceeds such that jobs are scheduled in non-decreasing order of their due dates. It also operates with processing periods for each job which are computed and defines time lag for jobs. Tasks of each job are then scheduled into these time lags using the specialized procedure. Details are described in [Brucker04].

```
S = algopij1sumti(S, problem)
```

Figure 8.49 Scheduling problem O|pij=1|SumTi

```
>> DueDates = [3 2 4 3 2];          % Due dates for each job
>> S = shop(ones(5,3), ones(5,3));
>> S.DueDate = DueDates;
>> pr = problem('O|pij=1|SumTi');
>> S = algopij1sumti(S,pr);
>> plot(S);
```

Figure 8.50 Optimal solution for example of algopij1sumti algorithm



8.5 Other Scheduling Problems

8.5.1 List of Algorithms

Table 8.5 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.

Table 8.5 List of algorithms

algorithm	command	problem	reference
[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]

8.5.2 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 $\mathtt{G}$. Each task $\mathtt{t}_i$ is represented by node $\mathtt{t}_i$ in graph G and has a positive processing time $\mathtt{p}_i$. Timing constraints between two nodes are represented by a set of directed edges. Each edge $\mathtt{e}_{ij}$ from the node $\mathtt{t}_i$ to the node $\mathtt{t}_j$ is labeled with an integer time lag $\mathtt{w}_{ij}$. 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 $\mathtt{t}_i$ to the node $\mathtt{t}_j$ with the positive time lag $\mathtt{w}_{ij}$ indicates that $\mathtt{s}_j$, the start time of $\mathtt{t}_j$, must be at least $\mathtt{w}_{ij}$ time units after $\mathtt{s}_i$, the start time of $\mathtt{t}_i$. The backward edge from node $\mathtt{t}_j$ to node $\mathtt{t}_i$ with the negative time lag $\mathtt{w}_{ji}$ indicates that $\mathtt{s}_j$ must be no more than $\mathtt{w}_{ji}$ time units after $\mathtt{s}_i$. The objective is to find a schedule with minimal $\mathtt{C}_{max}$.

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 6, "Classification in Scheduling"). In this case the problem with positive and negative time lags is identified by 'SPNTL'.

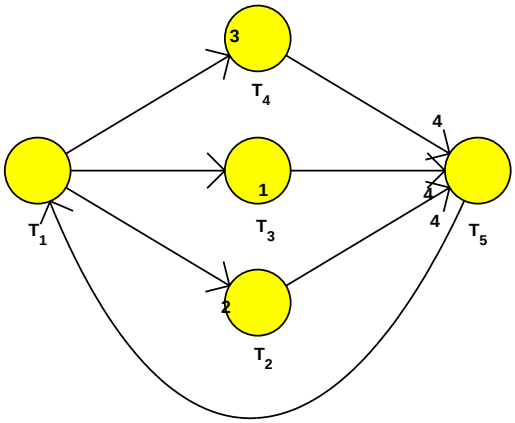
schoptions
optimization options (see Section [Scheduling Toolbox Options])

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

An example of the scheduling problem containing five tasks is shown in Figure 8.51 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 than or equal to 10 ($w_{5,1}$=-10). The objective is to find a schedule with minimal C_{max}.

Figure 8.51 Graph `G` representing tasks constrained by positive and negative time-lags.



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 8.51 can be opened and edited in Graphedit tool (`graphedit(g)`).

Resulting graph `G` is shown in Figure 8.54. Finally, 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:

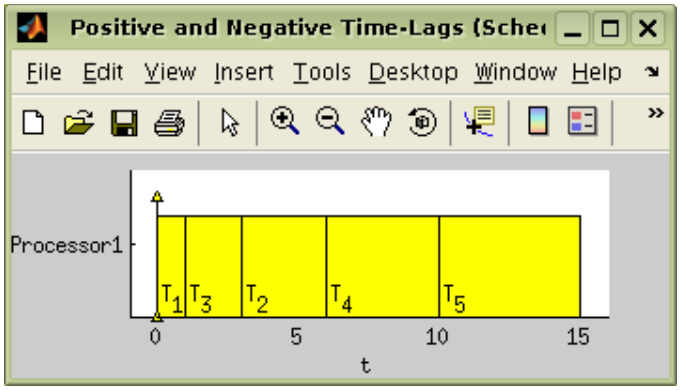
```
T = taskset(LHgraph,'n2t',@node2task,'ProcTime','Processor', ...
       'e2p',@edges2param)
```

For more details see Section 7.6. The optimal solution in Figure 8.52 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 8.52 Resulting schedule of instance in Figure 8.51.



8.5.3 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 t_i is represented by the node t_i in the graph `G` and has a positive processing time p_i. Edge e_{ij} from the node t_i to t_j is labeled by a couple of integer constants l_{ij} and h_{ij}. Length l_{ij} represents the minimal distance in clock cycles from the start time of the task t_i to the start time of t_j and it is always greater than zero. On the other hand, the height h_{ij} specifies the shift of the iteration index (dependence distance) from task 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 6, "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.

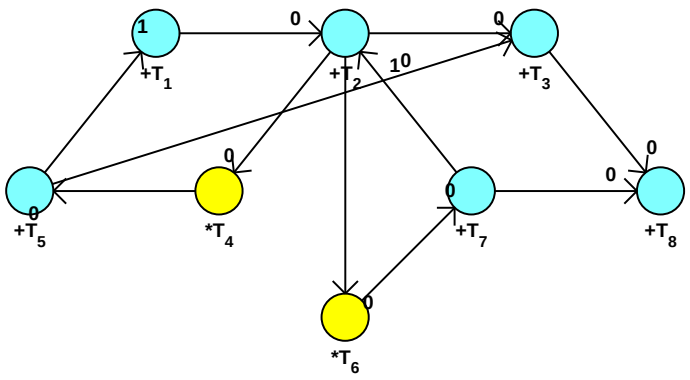
Example 8.5.2 Cyclic Scheduling - Wave Digital Filter.

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 8.53. 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).

Figure 8.53 Cyclic Data Flow Graph of WDF.



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

To transform Cyclic Data Flow Graph (CDFG) to graph `G` weighted by l_{ij} and h_{ij} function LHgraph can be used:

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

LHgraph
graph `G` weighted by l_{ij} and h_{ij}

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

UnitProcTime
vector of processing time of tasks on dedicated processors

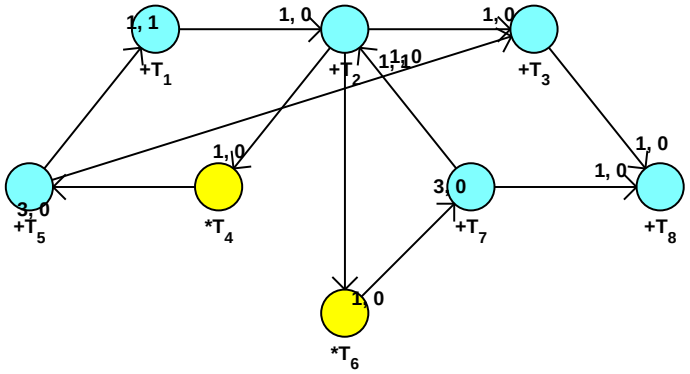
UnitLattency
vector of input-output latency of dedicated processors

Resulting graph `G` is shown in Figure 8.54. Finally, the graph `G` is used to generate an object taskset describing the scheduling problem. Conversion of the parameters 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 7.6.

Figure 8.54 Graph `G` weighted by l_{ij} and h_{ij} of WDF.



The scheduling procedure (shown below) found schedule depicted in Figure 8.55.

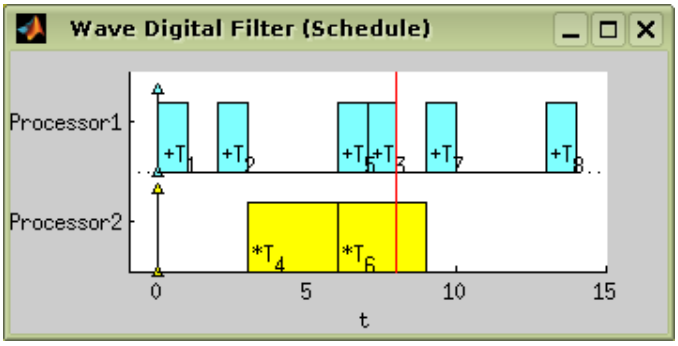
```
>> load <Matlab root>\toolbox\scheduling\stdemos\benchmarks\dsp\wdf
>> graphedit(wdf)
>> UnitProcTime = [1 3];
>> UnitLattency = [1 3];
>> m = [1 1];
>> LHgraph = cdfg2LHgraph(wdf,UnitProcTime,UnitLattency)

 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 8.55 Resulting schedule with optimal period w=8.

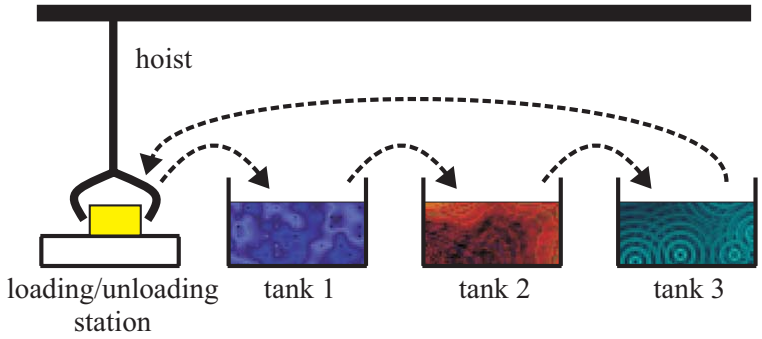


8.5.4 Cyclic Scheduling of a Single Hoist

Hoist scheduling is a typical optimization problem arising on automated electroplating lines. Such a line consists of a loading station, a series of processing tanks containing different chemical solutions, and an unloading station (see Figure 8.56). Usually, loading and unloading operation may be performed at the same station. Parts to be processed are transported between tanks in carriers by a hoist which can move only one carrier at a time. Each processed part needs to go through a number of processing stages sequentially. The processing time at each tank must be within a range defined by a lower limit and an upper limit [Liu02].

The hoist scheduling problem is to schedule the hoist moves to perform the material handling tasks in the system such that the throughput of the system is maximized or, equivalently, such that the production cycle is minimized. Cyclic hoist scheduling (CHS) is a type of hoist scheduling that deals with the scheduling that involves one hoist and produces some type of product in a multistage production line.

Figure 8.56 Graphics representation of task parameters



The hoist scheduling problem is described by following parameters:

n
number of processing stages (including the loading and unloading station),

a_i
minimum processing time in stage i ∈ <1,n> (i=1 for loading/unloading station),

b_i
maximum processing time in stage i ∈ <1,n>,

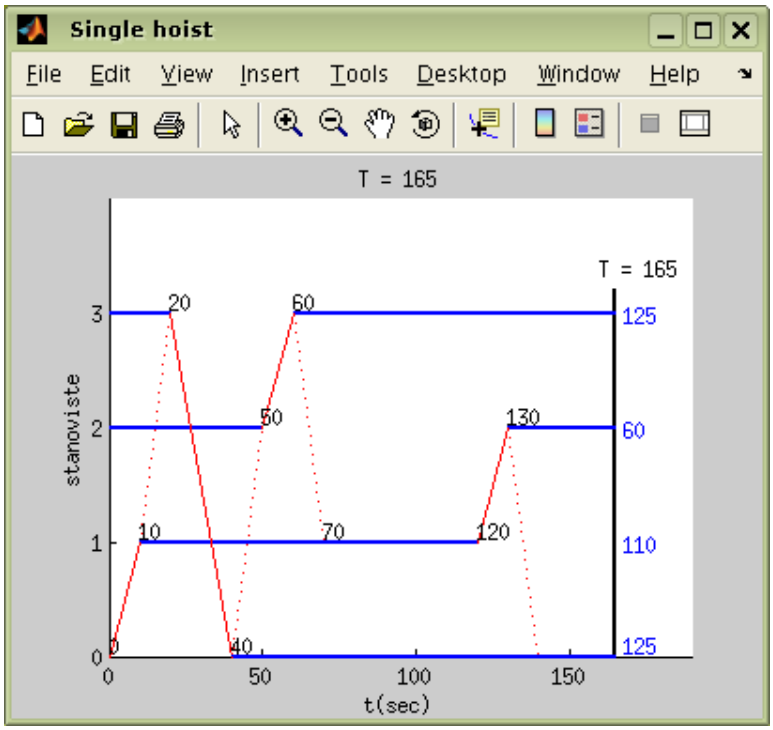
d_i
minimum time required for the hoist to move a carrier from tank i to tank mod(i+1,n) where i ∈ <1,n>,

c_{ij}
travelling time for the empty hoist from tank i to tank j where `i,j` ∈ <1,`n`>.

This scheduling problem can be solved using function `singlehoist` as is shown below. In the toolbox hoist moves are considered as tasks. Other parameters are passed to the algorithm as user defined parameters (`TSUserParam`). The algorithm is based on integer linear programming. For more details please see [Liu02]. Since results of hoist scheduling are usually presented in different form than in Gantt charts, function `singlehoist` supports generating of diagrams describing resulting schedule in a more transparent way. Resulting diagram for the example is shown in Figure 8.57.

```
>> a = [30 100 30 125];          %minimum processing time in stage
>> b = [1000 110 60 130];        %maximum processing time in stage
>> C = toeplitz([0 5 10 15]);    %traveling time of the empty hoist
>> d = [10 10 10 20];            %minimum hoist move time
>> T = taskset(d);               %Create taskset (d ~ processing time)
>> T.TSUserParam.SetupTime = C;
>> T.TSUserParam.minDistance = a;
>> T.TSUserParam.maxDistance = b;
>> TS = singlehoist(T,schoptions,1)
Set of 4 tasks
 There is schedule: single hoist scheduling algorithm
   Tasks period: 165
   Solving time: 0s
```

Figure 8.57 Cyclic scheduling of a single hoist example solution



Chapter 9

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.

For real-time systems 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.

9.1 Fixed-Priority Scheduling

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

9.1.1 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 9.1 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
```

9.1.2 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).

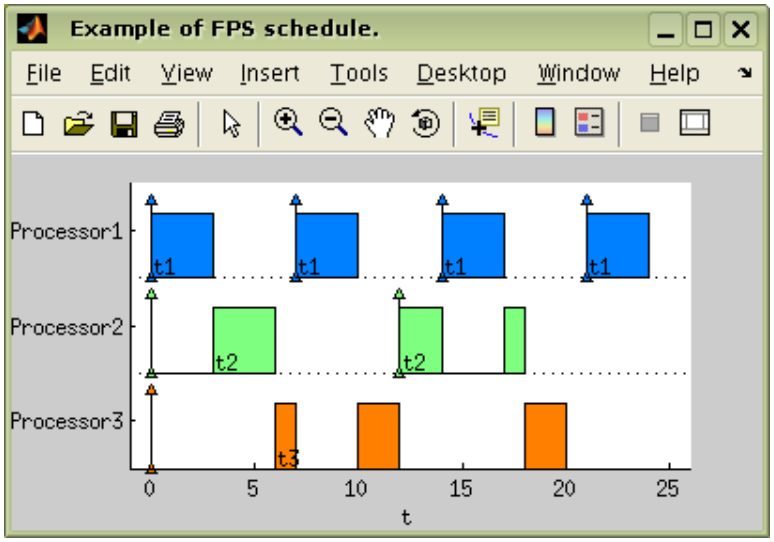
```
r = fps(TS)
```

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

Figure 9.2 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 9.3 Result of FPS algorithm



Chapter 10

Graph Algorithms

Scheduling algorithms have very close relation to graph algorithms. Scheduling toolbox offers an object graph (see section Chapter 7, "Graphs") with several graph algorithms.

10.1 List of Algorithms

List of algorithms related to operations with object graph are summarized in Table 10.1.

Table 10.1 List of algorithms

algorithm	command	note
Minimum spanning tree	`spanningtree`	Polynomial
Dijkstra's algorithm	`dijkstra`	Polynomial
Floyd's algorithm	`floyd`	Polynomial
Tarjan's algorithm	`tarjan`	Polynomial
Minimum Cost Flow	`mincostflow`	Using LP
Critical Circuit Ratio	`criticalcircuitratio`	Using LP
Hamilton circuit	`hamiltoncircuit`	NP-hard
Christofides	`christofides`	Polynomial
MWPM	`mwpm`	Using ILP
Multicommodity flow	`multicommodityflow`	Using LP
All path	`allpath`	NP complete
K shortest path	`xbestpath`	NP complete
Commodity flow	`commodityflow`	NP-complete
Graph coloring	`graphcoloring`	NP-hard
Quadratic Assignment Problem	`qap`	NP-hard

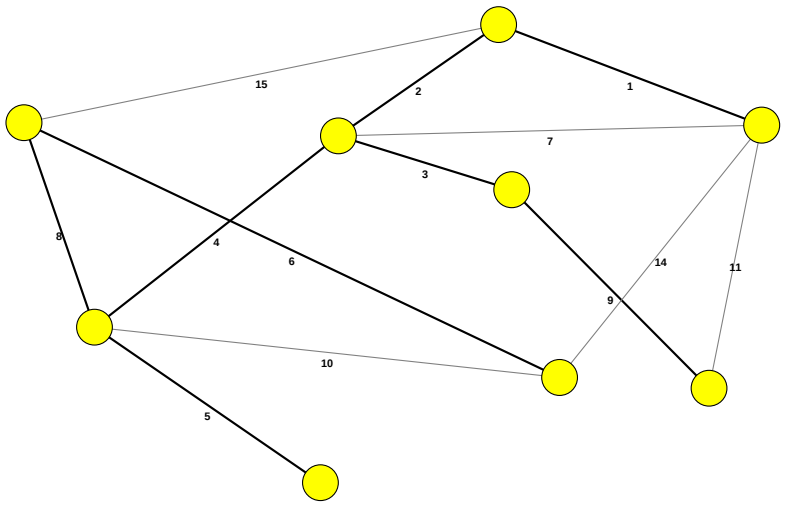
10.2 Minimum Spanning Tree

Graph's minimum spanning tree is defined by subgraf connecting all the vertices and by edges with minimum sum of the costs. Graph has to be weigthed. An example of the spanning tree is shown in figure Figure 10.1. Spanning tree can be obtained by many algorithms, that vary in the way how the edges are added to the graph. Examples can be found in sections Section 10.2.1, Section 10.2.2 and Section 10.2.3.

10.2.1 Kruskal's algorithm

This algorithm can start in any node and adds the edges according to their weigth. It means to add the cheapest edge and after every additional edge we have to check, if there is no cycle formed. In case, that the graph is not connected, algorithm finds a minimum spanning forest (Algorithm finds a spanning tree for all graphs and all components). A simple example is shown in Figure 10.2. Kruskal's algorithm time complexity is O(mn) [Korte05] page 120.

Figure 10.1 Example of minimum spanning tree




```
spanningTree = kruskal(g[,userparamposition])
[spanningTree usedEdges] = kruskal(g[,userparamposition])
```

g
input objeck of type graph which edges have to be weighted

userparamposition
input parameter represents the position of the weight parameters in edgeList

spanningTree
output object of type graf representing spanning tree of graf g

usedEdges
output matrix defines the order how the edges were used

Figure 10.2 Kruskal's algorithm call example

```
>> edgeList = {1 2 1; 2 3 1; 2 4 2; 3 4 5; 3 1 8; 5 7 2};
>> g = graph('edl',edgeList,'edgeDatatype',{'double'});
>> spanningtree = kruskal(g)

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

10.2.2 Prim's algorithm

Prim's algorithm is similar to Kruskal's algorithm. The edges are added to the only one component. Component can by described as an existing connection among the nodes. Component is magnified by adding the cheapest edge to any node of this component. As well as Kruskal, Prim finds a minimum spanning forest for the unconnected graph. A simple example is shown in Figure 10.3. Prim's algorithm time complexity is O(n^2) [Korte05] page 121.

```
spanningTree = prim(g[,userparamposition])
[spanningTree usedEdges] = prim(g[,userparamposition])
```

g
input objeck of type graph which edges have to be weighted

userparamposition
input parameter represents the position of the weight parameters in edgeList

spanningTree
output object of type graf representing spanning tree of graf g

usedEdges
output matrix defines the order how the edges were used

Figure 10.3 Prim's algorithm call example

```
>> edgeList = {1 2 1; 2 3 1; 2 4 2; 3 4 5; 3 1 8; 5 7 2};
>> g = graph('edl',edgeList,'edgeDatatype',{'double'});
>> spanningtree = prim(g);
```

10.2.3 Boruvka's algorithm

This algorithm is a little bit different from the previosly mentioned algorithms. It is capable to find a spanning tree only for a graphs, that have a different weigth of edges. So, if the weigths are different to each others, algorithm adds in every step to every node the cheapest edge. A simple example is shown in Figure 10.4. Boruvka's algorithm time complexity is O(mlogn) [Demel02][Boruvka26a][Boruvka26b].

```
spanningTree = boruvka(g[,userparamposition])
[spanningTree usedEdges] = boruvka(g[,userparamposition])
```

g
input object of type graph which edges have to be weighted

userparamposition
input parameter represents the position of the weight parameters in edgeList

spanningTree
output object of type graf representing spanning tree of graf g

usedEdges
output matrix defines the order how the edges were used

Figure 10.4 Boruvka's algorithm call example

```
>> edgeList= {1 2,1;1 3,2;2 3,4;4 3,5;5 6,6;7 8,8;7 9,7;8 9,9};
>> g = graph('edl',edgeList,'edgeDatatype',{'double'});
>> spanningtree = boruvka(g);
```

10.3 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.

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

g
graph object

r
reference node

Figure 10.5 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
```

10.4 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 node 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.

10.5 Strongly Connected Components

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

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:

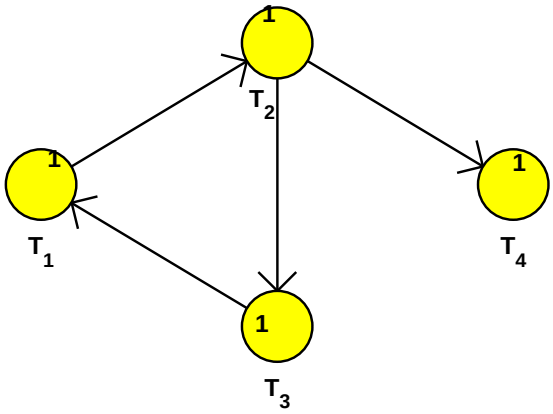
```
scc = tarjan(g)
```

where `scc` is a vector where the element `scc(i)` is number of component where the node `i` belongs to. For graph in Figure 10.7 the algorithm returns following results.

Figure 10.6 Strongly Connected Components example.

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

Figure 10.7 A simple network with optimal flow in the fourth user parameter on edges



10.6 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)`$\in$ `A` has an associated cost c_{ij} that denotes the cost per unit flow on that arc. We also associate with each edge a *capacity* u_{ij} that denotes the maximum amount that can flow on the arc and a *lower bound* l_{ij} that denotes the minimum amount that must flow on the arc. We associate with each node `i`$\in$ `N` an integer number `b(i)` representing its supply/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 supply/demand `b(i)` is stored in the first user parameter (UserParam) of nodes. Parameters (c_{ij}, l_{ij}, u_{ij}) are given in the first, second and third user parameter (UserParam) of corresponding edge e_{ij}. The function returns graph `G_minf` where the optimal flow f_{ij} is stored in the fourth user parameter (UserParam) on edge e_{ij}. A simple example is shown in Figure 10.8 and Figure 10.9.

NOTE

The algorithm uses function 'ilinprog' from TORSCHE. If you have a problem with this function please see Section Installation of TORSCHE.

10.7 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

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

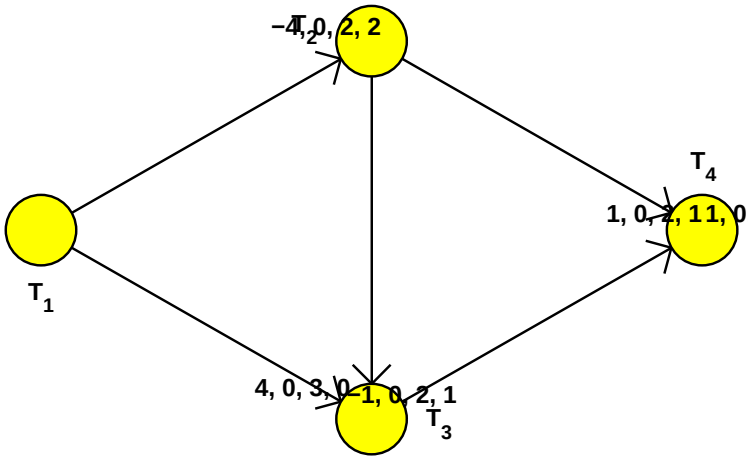
where `C` is a cycle of graph `G`. The circuit `C` with maximal circuit ratio is called critical circuit. Function

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

Figure 10.8 Mincostflow example.

```
% Definition of nodes: each row is one node. Each node is
% defined by a couple [i, b(i)].
>> nodedefinition = ...
    {1  3;...
     2  0;...
     3  0;...
     4 -3};
% Definition of edges: each row is a one edge. Each edge
% (i,j)\ensuremath{\in{}}A is defined by a vector [i, j, cij, lij, uij]
>> edgedefinition = ...
    {1 2 -4 0 2; ...
     1 3 -1 0 2; ...
     2 3  4 0 3; ...
     2 4  1 0 2; ...
     3 4  1 0 2};
% Create graph g from the above defined parameters.
>> g = graph('ndl',nodedefinition,'edl',edgedefinition);
% Compute minimum cost flows in the graph.
>> gminf = mincostflow(g);
>> graphedit(gminf);
% The last collumn of this cell matrix consif of the optimal flow.
>> flowmatrix = get(gmin,'edl')
```

Figure 10.9 A simple network with optimal flow in the fourth user parameter on edges



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.

10.8 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

Figure 10.10 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
```

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 10.12.

Figure 10.11 Hamilton circuit identification example.

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

NOTE

The algorithm uses function 'ilinprog' from TORSCHE. If you have a problem with this function please see Section Installation of TORSCHE.

10.9 Christofides

Christofides's algorithm solves a travelling salesman problem in a graph `G`. Looking for the shortes way in a graph `G`. In this graph have to exist way from every nodes to every nodes and graph has to be waigthed. For completing graph is possible to use function complete and for waigthed graph function distance - if is the graph made by graphedit. Algorithm is based on the minimum spanning tree, which is in sequence remake to TSP(travelling salesman problem). An example of the TSP in a graph is shown in

Figure 10.12 An example of Hamilton circuit.

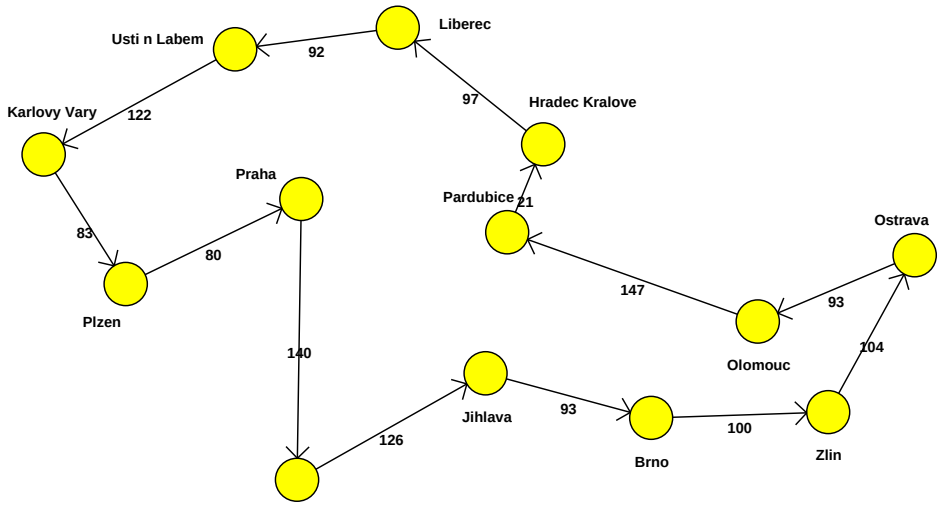


figure Figure 10.14. A simple example is shown in Figure 10.13 . Christofides's complexity is O(nˆ4logn) and approximatio is 3/2. [Korte05] page 503.

```
TSP = christofides(g)
[TSP g] = christofides(g)
```

g
input objeck of type graph which edges have to be weighted

TSP
output is list of edges, which includes order of graph's edges as were used

g
output object of type graf, the TSP solution in form of graph object.

Figure 10.13 Christofides's algorithm call example

```
>> edgeList= {1 2,1;1 3,2;1 4,2;2 3,2;2 4,2;3 4,3};
>> g = graph('edl',edgeList,'edgeDatatype',{'double'});
>> [TSP g] = christofides(g);
```

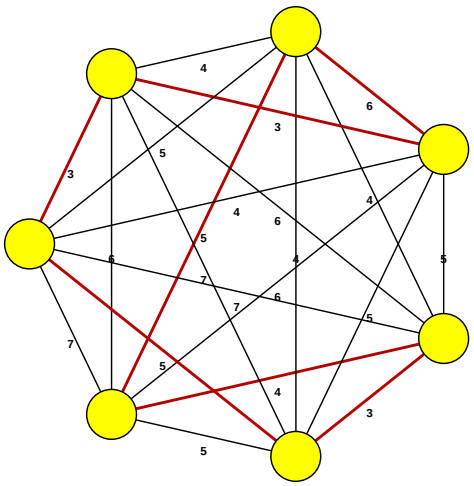
NOTE

The algorithm uses function 'ilinprog' from TORSCHE. If you have a problem with this function please see Section Installation of TORSCHE.

10.10 Minimal weight perfect matching

Mwpm algorithm means minimal weight perfect matching. This algorithm finding the minimal weight perfect matching in a graph. For the requisited have to run, that the number of the nodes has to be even. Algorithm use function 'ilinprog'. A simple example is shown in Figure 10.16 and Figure 10.14. A call example is shown in Figure 10.15

Figure 10.14 An example of Christofides's algorithm use .



```
gpair  = mwpm(g)
```

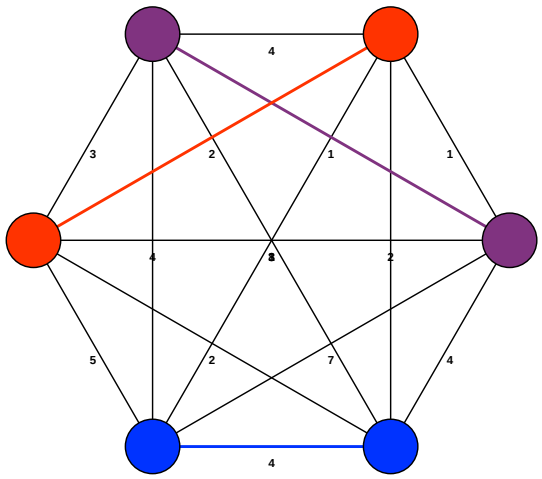
g
input objeck of type graph which edges have to be weighted

gpair
output object of type graf representing mwpm in the graf g

Figure 10.15 Mwpm algorithm call example

```
>> edgeList= {1 2,3;1 3,2;1 4,3;1 5,2;1 6 5;2 3,4;2 4,1;2 5,1;2 6,4;3 4,1;3 5,2;3 6,8;4 5,4;4 6,7;
>> g = graph('edl',edgeList,'edgeDatatype',{'double'});
>> gpair = mwpm(g);
```

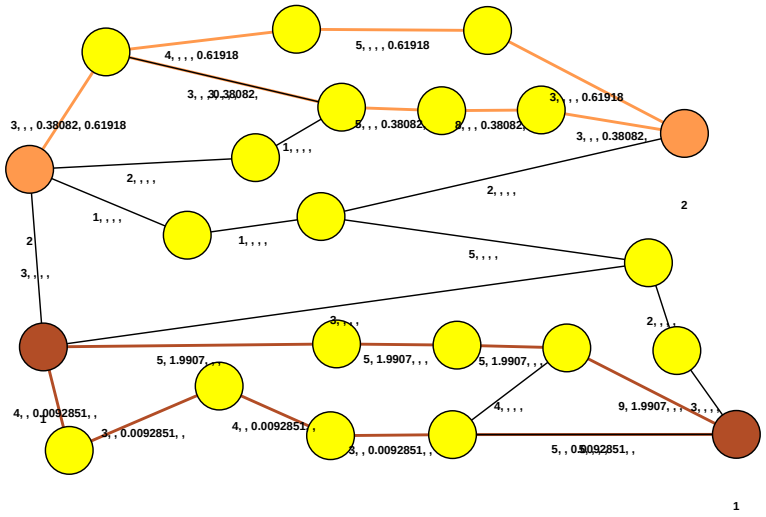
Figure 10.16 An example of MWPM function use .



10.11 Multicommodity flow

Multicommodity flow is a function, which finds flows in the graph with required quantity flows. Algorithm looks for a path in the graph from source nodes to sinks nodes. The algorithm is able to search maximum flow through the graf (maxmulticommodityflow function) or schedule some given value of the flow (multicommodityflow). A simple example is shown in Figure 10.17.

Figure 10.17 An example of multicommodityflow function use .



10.11.1 Multicommodity flow

This algorithm schedules the given flow into the set of the most suitable paths through the graph. The set of paths is prepared by an external function. It is possible to choose whether to use 'allpath' function, which finds all the path through the graph or 'kshortestpath' function, which finds just a required number of the shortest paths through the graph. Algorithm uses function 'linprog' to solve the MCF problem. A usage example is shown in Figure 10.17. A call example is shown in Figure 10.18

```
gmcf  = multicommodityflow(g,s,t,b,pathFilter)
gmcf  = multicommodityflow(g,s,t,b,pathFilter,limitation)
gmcf  = multicommodityflow(g,s,t,b,pathFilter,limitation,cap)
gmcf  = multicommodityflow(g,s,t,b,pathFilter,limitation,cap,cost)
[gmcf numberOfFlows]  = multicommodityflow(g,s,t,b,pathFilter,limitation,...)
```

g
: input object of type Graph which edges have to be weighted

s
: list of source nodes

t
: list of sinks nodes

b
: list of reqired flows

pathFilter
: type of function, which is used for finding paths,four possibilites ('all','allShortest','kShortest','kMosteCap')

limitation
: for choice 'kShortest' and 'kMosteCap' is number of finding paths

cap
: number of parameter, on which capability of edges are saved.

cost
number of parameter, on which cost of edges are saved

gmcf
output object of type graf

numberOfFlows
number of found flows

Figure 10.18 Multicommodityflow algorithm call example

```
>>edgeList= {1 2,3;1 3,4;1 7,5;1 10,3;2 13,1;2 15,3;2 18,2;3 4,3;4 5,4;
>> 5 6,3;6 9,4;6 12,5;7 8,5;8 9,5;9 12,9;10 11,2;10 14,5;11 12,3;13 14 1;
>> 14 22,2;15 16,4;16 17,5;17 22,3;18 19,1;19 15,3;19 20,5;20 21,8;21 22,3};
>> g = graph('edl',edgeList,'edgeDatatype',{'double'});
>> s = [1 2];
>> t = [12 22];
>> b = [2;1];
>> limitation = 2;
>> gmcf = multicommodityflow(g,s,t,b,'kMosteCap',limitation)
```

10.11.2 Maximum multicommodity flow

Maximum multicommodity flow is a function, which finds the maximum flow through the graph using the prepared set of paths. While having the 'pathFilter' option set into 'allpath' value, all the possible paths through the graph are used. On the other hand, setting it into 'kshortestpath' value makes the algorithm to use just the k most suitable paths. The algorithm uses 'linprog' function for solving the problem. The usage example is shown in Figure 10.17. A call example is shown in Figure 10.19

```
gmmcf  = maxmulticommodityflow(g,s,t,pathFilter)
gmmcf  = maxmulticommodityflow(g,s,t,pathFilter,limitation)
gmmcf  = maxmulticommodityflow(g,s,t,pathFilter,limitation,cap)
gmmcf  = maxmulticommodityflow(g,s,t,b,pathFilter,limitation,cap,cost)
```

g
input object of type Graph which edges have to be weighted

s
list of source nodes

t
list of sinks nodes

pathFilter
type of function, which is used for finding paths,four possibilites('all','allShortest','kShortest','kMosteCap')

limitation
for choice 'kShortest' and 'kMosteCap' is number of finding paths

cap
number of parameter, on which capability of edges are saved.

cost
number of parameter, on which weigths of edges are saved

gmmcf
output object of type graf

Figure 10.19 Multicommodityflow algorithm call example

```
>>edgeList= {1 2,3;1 3,4;1 7,5;1 10,3;2 13,1;2 15,3;2 18,2;3 4,3;4 5,4;
>> 5 6,3;6 9,4;6 12,5;7 8,5;8 9,5;9 12,9;10 11,2;10 14,5;11 12,3;13 14 1;
>> 14 22,2;15 16,4;16 17,5;17 22,3;18 19,1;19 15,3;19 20,5;20 21,8;21 22,3};
>> g = graph('edl',edgeList,'edgeDatatype',{'double'});
>> s = [1 2];
>> t = [12 22];
>> limitation = 2;
>> gmmcf = maxmulticommodityflow(g,s,t,'kMosteCap',limitation)
```

10.12 Allpath

Function allpath finds paths through the graph. If the graph contains cycle, it is necessary to set input parameter ci. A simple example is shown in Figure 10.21. A call example is shown in Figure 10.20

```
gap  = allpath(g,s,t)
gap  = allpath(g,s,t,ci)
gap  = allpath(g,s,t,ci,cap)
[gap numberOfFlows]  = allpath(g,s,t,...)
[gap numberOfFlows amplFlows]  = allpath(g,s,t,...)
```

g
: input object of type Graph which edges have to be weighted

s
: input parameter as a list of source nodes

t
: input parameter as a list of sinks nodes

ci
: cycle ignore, input parameter, which has to be set, if graph contains cycle

cap
: number of parameter, on which capacity of edges are saved

gap
: output object of type graf with paths saved as userParamPos

numberOfFlows
: output numbers of separate flows from source to sink nodes

amplFlows
: output amplitude of found flows

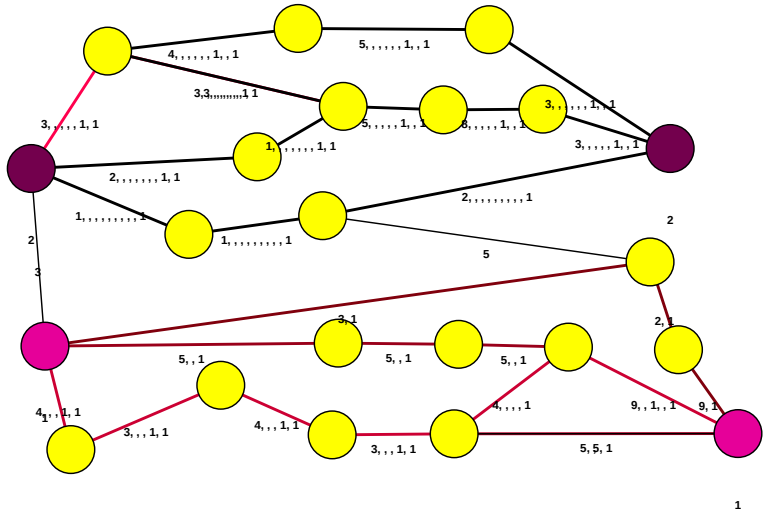
Figure 10.20 Allpath algorithm call example

```
>>edgeList= {1 3,5;3 4,4;3 4,4;4 5,5;4 6,5;2 3,5;4 3,2};
>> g = graph('edl',edgeList,'edgeDatatype',{'double'});
>> s = [1 2];
>> t = [5 6];
>> ci = 1;
>> gap = allpath(g,s,t,ci)
```

10.13 Kshortestpath

Function kshortestpath finds k most suitable paths through the graph. Number of paths to be found is set by the parameter k. It can search the most suitable paths according either the edge capacities

Figure 10.21 An example of allpath function use.



(maximize the path capacity) or edge costs (minimize the total path cost). If the graph contains cycle, it is necessary to set input parameter ci. A simple example is shown in Figure 10.23. A usage example is shown in Figure 10.23. A call example is shown in Figure 10.22

```
gksp  = kshortestpath(g,s,pathFilter)
gksp  = kshortestpath(g,s,pathFilter,k)
gksp  = kshortestpath(g,s,t,pathFilter,k,ci)
gksp  = kshortestpat(g,s,t,pathFilter,k,ci,numpath)
gksp  = kshortestpat(g,s,t,pathFilter,k,ci,numpath,cap)
gksp  = kshortestpat(g,s,t,pathFilter,k,ci,numpath,cap,cost)
[gksp numberOfFlow]  = kshortestpat(g,s,t,pathFilter,...)
```

g
: input object of type Graph which edges have to be weighted

s
: input parameter as a list of source nodes

t
: input parameter as a list of sinks nodes

pathFilter
: input parameter choice from finding the moste capabilites/the quickest path

k
: number of paths searched for

ci
: cycle ignore, input parameter, if graph contains cycle, ci has to be set

cap
: number of parameter, on which capacity of edges are saved

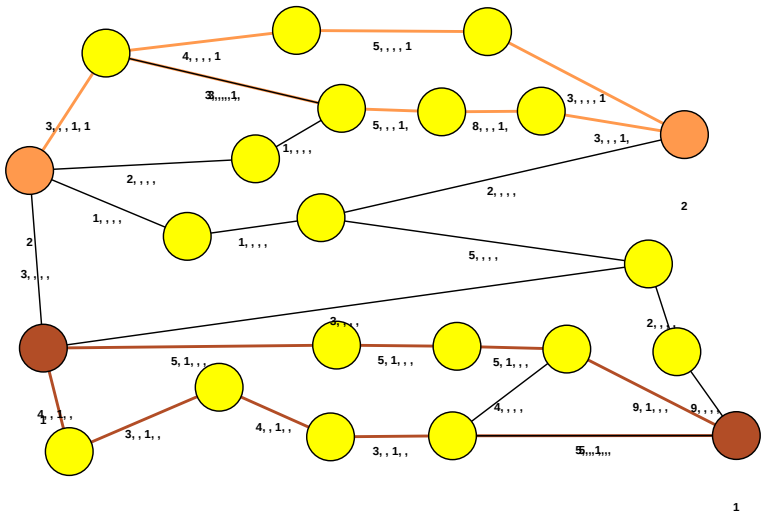
cost
: number of parameter, on which cost of edges are saved.

gksp
: output object of type graf with paths saved as userParamPos

Figure 10.22 Kshortestpath algorithm call example

```
>>edgeList= {1 3,5;3 4,4;3 4,4;4 5,5;4 6,5;2 3,5;4 3,2};
>> g = graph('edl',edgeList,'edgeDatatype',{'double'});
>> s = [1 2];
>> t = [5 6];
>> gksp = kshortestpath(g,s,t,'kShortest',1)
```

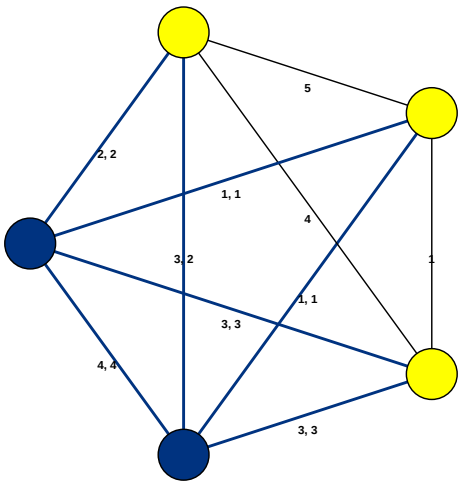
Figure 10.23 An example of kshortestpath function use.



10.14 Commodity flow

Commodity flow problem looking for the maximum flow from the one source node to one sink node. This problem can be solve by many algorithm, for example edmonds-karp algorithm or newer faster one dinic's algorithm. A simple example is shown in Figure 10.24.

Figure 10.24 An example of commodityflow function use.



10.14.1 Edmonds Karp

One of the algorithm solves the commodity flow problem. EdmondsKarp algorithm is based on breadth-first search algorithm. This algorithm lookes for path by the graph, return the maximum flow, which can stream this path and compare with the last flow. A simple example is shown in Figure 10.25. EdmondsKarp's algorithm time complexity is O(nm^2). [Korte05] page 164.

```
[f flowMatrix] = edmondsKarp(g,s,t)
```

g
input object of type graph which edges have to be weighted

s
input parameter represents source node

t
input parameter represents sink node

f
output value of maximum flow of the graph

flowMatrix
output matrix giving a legal flow with the maximum value.

Figure 10.25 EdmondsKarp's algorithm call example

```
>> edgeList= {1 2,3;1 4,3;3 1,3;2 3,4;3 4,1;3 5,2;5 2,1;4 5,2;5 6,1;4 6,6;6 7,9};
>> g = graph('edl',edgeList,'edgeDatatype',{'double'});
>> s=1;t=7;
>> [f flowMatrix] = edmondsKarp(g,s,t)
```

10.14.2 Dinic

Another algorithm solves the commodity flow problem is dinic algorithm. Algorithm is faster than edmondsKarp algorithm, because it based on finding the augmenting path in the simplifiend graph. Algorithm started with searching graph, if there exist some edges, witch have full capacity and delete it from the graph. Than is used the function level graph for other simplify. And as the last step is finded augmenting path and is increased flow. A simple example is shown in Figure 10.26. Dinic's algorithm time complexity is O(n^2m) [Korte05] page 166.

```
f = dinic(g,s,t)
[f flowMatrix] = dinic(g,s,t)
```

g
input object of type graph which edges have to be weighted

s
input parameter represents source node

t
input parameter represents sink node

f
output value of maximum flow of the graph

flowMatrix
output matrix giving a legal flow with the maximum value.

Figure 10.26 Dinic's algorithm call example

```
>> edgeList= {1 2,3;1 4,3;3 1,3;2 3,4;3 4,1;3 5,2;5 2,1;4 5,2;5 6,1;4 6,6;6 7,9};
>> g = graph('edl',edgeList,'edgeDatatype',{'double'});
>> s=1;t=7;
>> [f flowMatrix] = dinic(g,s,t)
```

10.15 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

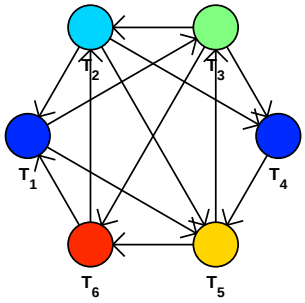
G2
colored graph

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

Example 10.15.1 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 10.27 An example of Graph coloring



10.16 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

$\mathbf{D}= [\mathtt{d}_{ih}]_{n,n}$ gives distances between locations, where $\mathtt{d}_{ih}$ is distance between location `i` and location `h`, and a matrix $\mathbf{F} = [\mathtt{f}_{jk}]_{n,n}$ characterizes flows among activities (transfer of data, material, etc.), where $\mathtt{f}_{jk}$ is the flow between activity `j` and activity `k`. An assignment is a permutation π of {1,...,n}, where 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 $\mathbf{X}$ of dimension n × n (whose elements $\mathtt{x}_{ij}$ are 1 if the activity `j` is assigned to location `i` and 0 in the other cases) such that:

Equation 10.16.1

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

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.

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>.

Figure 10.28 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
```

Chapter 11

Optimization Algorithms

.

11.1 List of Algorithms

List of algorithms related to operations.

Table 11.1 List of algorithms

algorithm	command	note
Knapsack problem	`knapsack`	NP-Hard
Knapsack problem graph	`knapsack_graph`	Polynomial

11.2 Knapsack problem

Knapsack problem is optimization problem, which is possible to solve by many algorithms. One of these algorithms is transformation to the graph problem, the other uses a dynamic programming. The second algorithm much faster, it is shown in Figure 11.1.

11.2.1 Knapsack problem

This algorithm is solved by using dynamic programming. It means, that the algorithm starts with the first subject and go through all the possible solutions. For all the solutions selects the solution, where the price level of subjects is the biggest. Than start to go back and look the right path. Example of the function result is shown in Figure 11.2. For more information read [Korte05] page 419.

```
usedEdges = knapsack(weigth,cost,maxweight)
usedEdges = knapsack(weigth,cost,maxweight)
```

weight, cost, maxWeight
: weight, cost and maxweight are inputs; the first two inputs defined subjects and maxweigth is capacity of the knapsack

usedEdges
: output is a list of used subjects

11.2.2 Knapsack problem graph

This function demonstrates the solution using transformation of the optimization problem to the graph problem. The solution is found as the longest way through the graph. Example of function call is shown in Figure 11.3, and result is shown in Figure 11.4. For more information read [Demel02].

Figure 11.1 Comparison of running times of knapsack and knapsackDP algorithms.

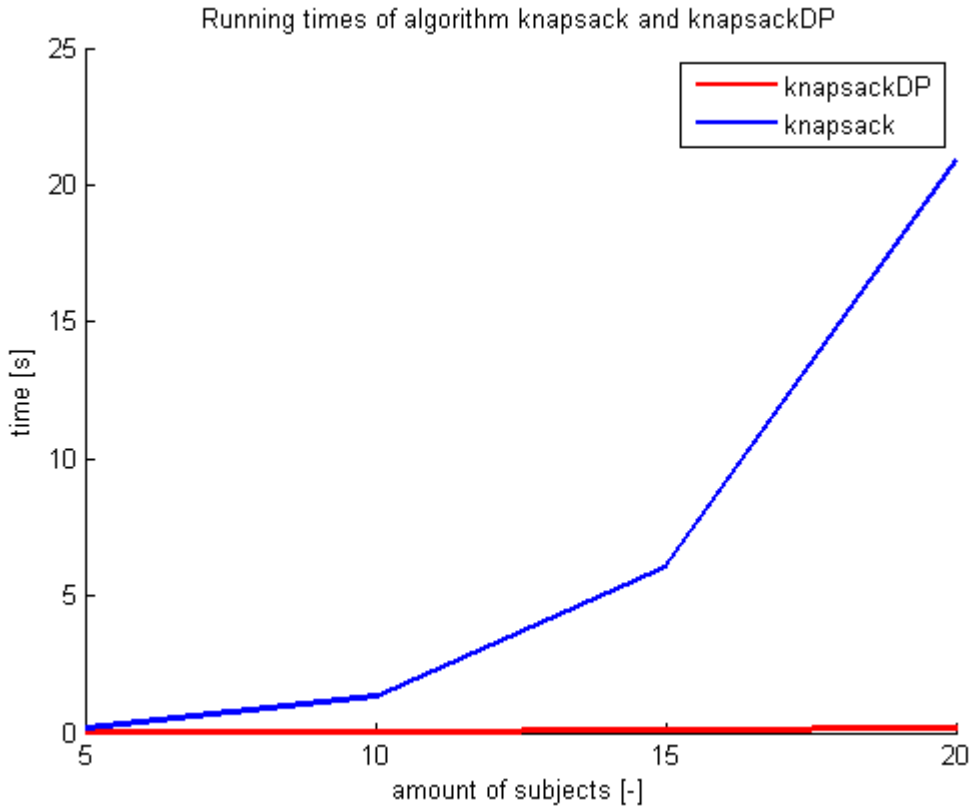


Figure 11.2 Knapsack algorithm call example

```
>> weight = [7 6 4 3];
>> cost = [2 3 4 5];
>> maxWeight = 15;
>> usedSubject = knapsack(weight,cost,maxWeight);
```

```
usedEdges = knapsack_graph(weigth,cost,maxweight)
usedEdges = knapsack_graph(weigth,cost,maxweight)
```

weight, cost, maxWeight
: inputs are weight and cost of the subjects and maxweight, capacity of the knapsack

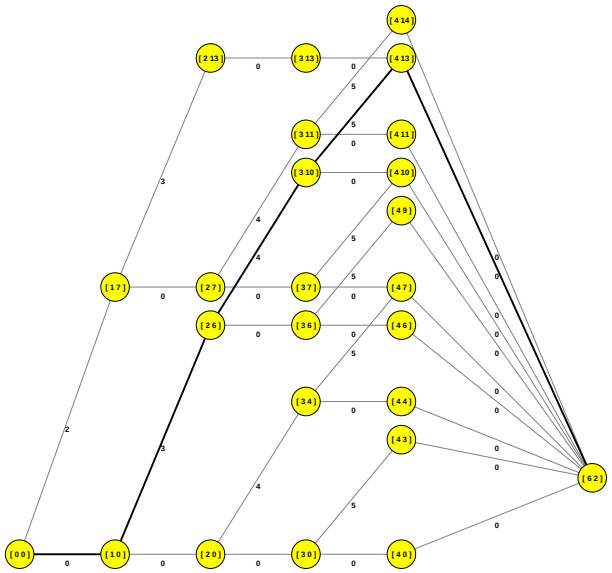
usedEdges
: output is a list of used subjects

g
: output graph

Figure 11.3 Knapsack_graph algorithm call example

```
>> weight = [7 6 4 3];
>> cost = [2 3 4 5];
>> maxWeight = 15;
>> [usedSubject g] = knapsack_graph(weight,cost,maxWeight);
>> graphedit(g)
```

Figure 11.4 An example of knapsack algorithm usege



Chapter 12

Other Algorithms

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

12.1 List of Algorithms

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

Table 12.1 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

12.2 Scheduling Toolbox Options

A 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 12.2. Default values of single parameters are typed in italics.

12.3 Random Data Flow Graph (DFG) generation

This supplementary function allows to generate random Data Flow Graph (DFG). It is intended 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 nodes. 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 have user parameter `g.E(i).UserParam=0`.

Table 12.2 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

12.4 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]). When the parameter is not given the function use the default settings (`schoptions=schoptionsset`). 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 12.3. The default value is 'L'.

Table 12.3 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 default value is 0 (`Inf`). 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). The default value is 'C'.

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.

12.5 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 12.4.

```
[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]). When the parameter is not given the function use the default settings (`schoptions=schoptionsset`). 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 12.4. The default value is 'L'.

Table 12.4 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 default value is 0 (`Inf`). 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). The default value is 'C'.

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, there is no guaranty that solver `miqp` [Bemporad04] (see Section [Scheduling Toolbox Options]) will find the optimal solution in spite of it exists. 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/>).

12.6 Cyclic Scheduling Simulator

CSSIM (Cyclic Scheduling Simulator) is a tool allowing to simulate scheduled iterative loops in Matlab Simulink using TrueTime tool [Cervin06]. The loop described in *SubLab*, 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 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` and extended with cyclic schedule obtained by e.g. using `cycsch` function. The second parameter specifies name of output TrueTime initialization file. The third one specifies name of output TrueTime code of simulated loop.

The SubLab, language used in input files, is described in the following subsection.

12.6.1 SubLab - 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 in Hz for TrueTime must 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 *Processors 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.

12.6.2 TrueTime

TrueTime is used for time-exact simulation of schedules of iterative loops. 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 it initializes 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 SubLab 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 12.1. 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 12.2.

For more details see CSSIM TrueTime demo in the toolbox (`\scheduling\stdemos\cssim_demo.m`). For the simulation the TrueTime tool must be installed.

Figure 12.1 Simulation scheme with TrueTime Kernel block

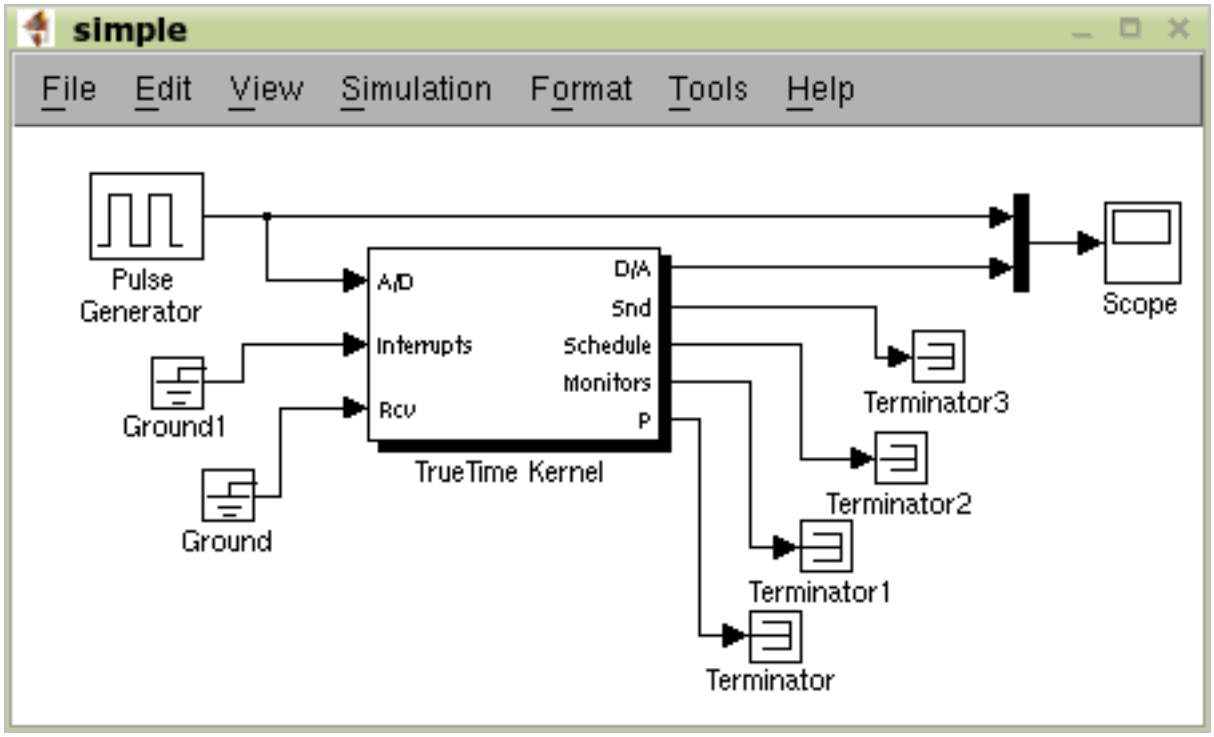
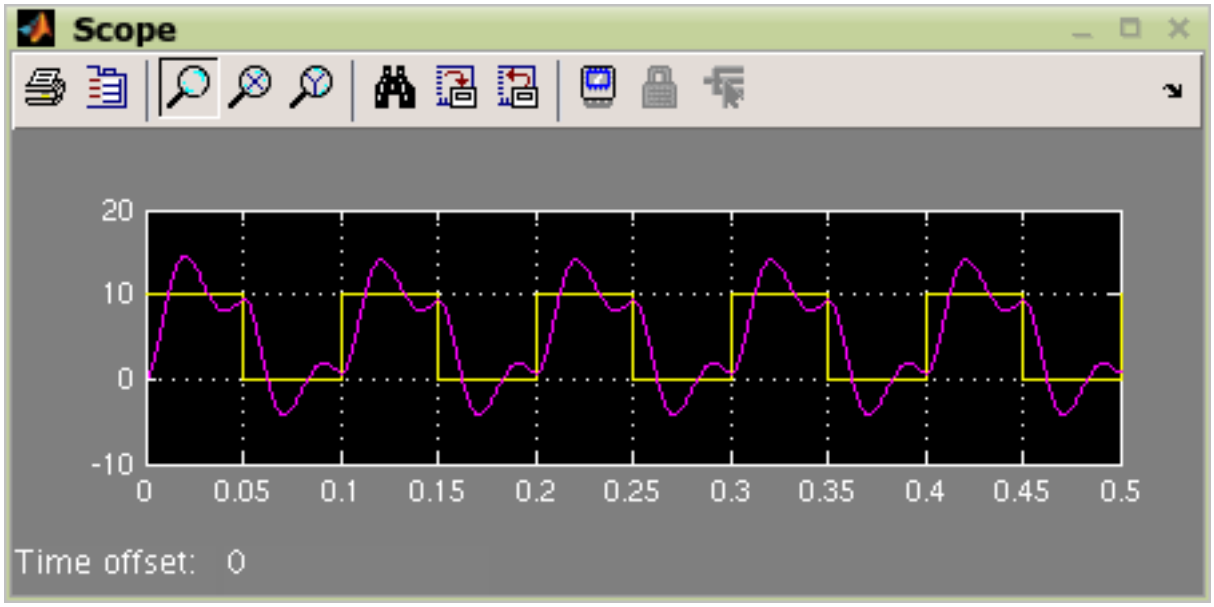


Figure 12.2 Result of simulation



12.7 VISIS

In this toolbox, there is a possibility to simulate or visualize acquired scheduling results. *VISIS* aplication (VIsualization and SImulation in Scheduling) is designed with respect to maximum user comfort and simplicity of usage, so most of the data structures and supplemental files are generated automatically. This aplication uses Simulink environment and Virtual Reality toolbox, which is standard part of Matlab. Any task can be assigned with Matlab commands and therefore any time schedule defined via data structures of TORSCHE can be simulated using these commands. Moreover, users of the toolbox can create their own project in virtual reality and then use it for the visualization of time schedule that has

to be included in taskset object. Both simulation and visualization are realized in Simulink and it is possible to store any signal for later analysis. In case of visualization, it is possible to capture any frame of final graphic representation and also stream video file of the visualization.

12.7.1 Simulation with VISIS

First necessary step after setting and scheduling the problem is to assign tasks with Matlab operations that will be executed according to given time schedule. This code has to be stored in attribute *UserParam* for each task in the taskset. Thus, a function *adduserparam* is available. This function adds code from text file to all tasks in the taskset together. First argument of this function is a taskset object and second argument is a string with name of text file. Output is a taskset with assigned code.

```
>> TS = adduserparam(TS, 'data.txt');
```

Next step is to define inputs and outputs of the main control block in the Simulink scheme that are needed for given commands. For this definition, function *visiscontrolports* is available. There are two keywords for this function: *Input* and *Output*. First of them introduces part for definition of inputs and second one for outputs. After each of them, next argument is the name of input/output and then its size (vector lenght). If only inputs (or outputs) are needed, then only one keyword can be used. Output of this function is a structure, which is one of the input arguments for the main function. Example of creating two inputs with size 1 and one output with size 2:

```
>> ports = visiscontrolports('Input','w',1,'e',1,'Output','control',2);
```

The main function of VISIS is called *taskset2simulink* and it generates Simulink scheme and control function accordingly to a given taskset and other data structures. Calling syntax:

```
taskset2simulink(ProjectName, TS, ports, VRin, stopTime [, options])
```

ProjectName
: name of virtual reality file (has to be ended by postfix *.wrl*); if virtual reality is not nedded, any name of project can be set

TS
: taskset with schedule

ports
: structure wtih information about inputs and outputs of control block

VRin
: structure wtih information about inputs of virtual reality block; empty argument if virtual reality is not used

stopTime
: stop time of simulation

options
: additional information, set in formal: '*Property_name*', *Property_value*

- Sample – sample time, default value is 1

- Period – period of the schedule, infinity as default

- Simulink – string with information about Simulink scheme generation

– 'off ' if Simulink scheme is not nedded to be created

Example of main function call for case without virtual reality, name of project is *project1*, sample time is one second, stop time is 50 seconds, period of schedule is 10 seconds and we do not need to generate Simulink scheme:

```
taskset2simulink('project1', TS, ports, [], 50, 'Period', 10, 'Simulink','off')
```

12.7.2 Visualisation with User-deifned Virtual Reality

Technique for visualisation of scheduling results is the same as for simulation with one additional step. It is necessary to define inputs for virtual reality block in Simulink. For this purpose, function *vrports* is available. Input arguments are pairs of strings, where first is the exact name of the object in virtual reality (VR) and second is property that we want to refer to. Output of this function is structure with given information. Example:

```
>> VRin = vrports('Arm1','translation','ColorMachine1','diffuseColor');
```

If there is a need to control some inputs of VR by outputs of control block, it is necessary to have theese outputs/inputs in the same order in the definition. For example, if there is a need to control the third input of VR block, appropriate output of control block has to be also on the third position. In situation when there is a need to have more outputs of control block than inputs of VR block, theese outputs have to be defined after definition of corresponding outputs/inputs. In the situation with more VR inputs than control outputs it is the same. Example how to control first input of VR block by first ouput of control block and there are needed two other outputs:

```
>> ports = visiscontrolports('Output','controlArm','out1',1,'out2',2);
>> VRin = vrports ('Arm1','translation');
```

Now it is possible to call main function of VISIS:

```
>> taskset2simulink('my_poject.wrl', TS, ports, VRin, 500);
```

12.7.3 Definition of commands for tasks

Commands assigned to tasks are executed with respect to time schedule and there are three ways how to define commands for task by relative time from its start time in schedule:

- By keyword *repeat* followed by three numbers separated by colons – start, step and stop. First number determines start time for executing block of following block of commands. This time data means how long after begin of task in schedule will theese commands start to be executed. Second number determines time space between two repetitions of this block of commands and the last number determines the end of executing commands refering to the begin of task in schedule. Time data start and stop can overreach borders of task in schedule, so start can be negative and stop can be greater than processing time of task. Therefore it's possible to define some operations before and after allocation of task in the schedule. Time data step has to be positive number.
- By keyword *divide* followed also by three numbers separated by colons – start, step and stop. Meaning of theese time data are very similar to previous subsection with only one differnce: data start and stop are not right in time units. Instead of this, they are related to processing time of task. Generally they are decimal numbers and it's also possible to execute some commands outside of task in schedule.
- By keyword *in* followed by only one time data which determines in what time related to beginning of task in schedule will be block of commands executed.

Example of code for one task:

```
task1
repeat 0:1:5
T1trans(1) = T1trans(1)+1;
in 5
ColM1 = [1 0 0];
repeat 5:1:10
T1trans(2) = T1trans(2)+1;
divide 0.5:1:1
Arm1(1) = Arm1(1)-1;
Arm2(1) = Arm2(1)+1;
endparam
```

Where *task1* is the name of task to assign following code and *endparam* is a keyword that denotes end of command block for appropriate task. For each task, block of commands for one task has to be introduced by its name or by special character # and then ordinal number in taskset. Until use of new keyword, all commands belong to the same keyword. It is possible to use all standard Matlab functions and in addition, function *vrgetpar* is available. By this function, it is possible to get numerical value of any accesible property defined in virtual reality file. First argument of this function is name of virtual reality file, second argument is string with name of object in VR and the last argument is the name of property we want to get. Output value is then numerical vector with relevant size.

```
>> value = vrgetpar('my_world.wrl','Earth','rotation');
```

To store any numerical value between two time samples, there are internal states *x(i)*. Main function will automatically define needed amount of statec acoording to the highest index used. To refer states, use format *x(3)*, it is not possible to use notation *x(1:3)*.

12.8 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 XrefId[??].

Chapter 13

Case Studies

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

13.1 Theoretical Case Studies

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

13.1.1 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 finished 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 create the schedule of the work (see Table 13.1). We consider working day starts at 8:00.

Table 13.1 The scheduling problem description

	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 to 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');
```

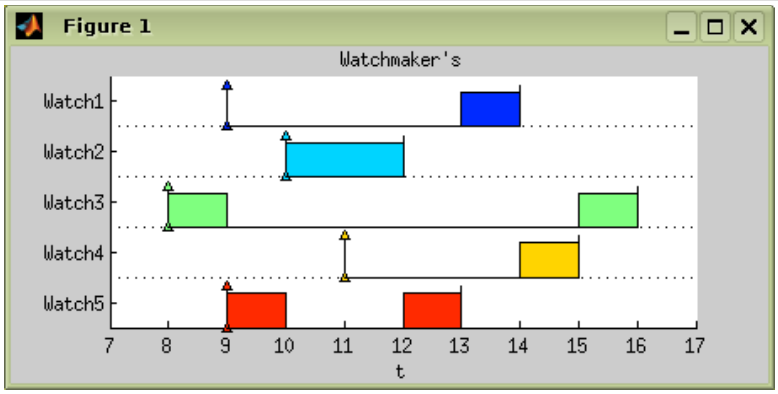
4. The algorithm is used as a function with 2 parameters. The first one is the taskset we have defined above, the second one is the type of problem written in Graham-Blaziewitz notation.

```
>> 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 13.1.

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

Figure 13.1 Result of case study as Gantt chart



13.1.2 Conveyor Belts

Transportation of goods by two conveyor belts is a simple example of using List Scheduling in practice. Construction material must be transported from place to another place with minimal time effort. Transported articles represent five kinds of construction material and two conveyor belts as processors are available. Table 13.2 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 13.2 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');
```

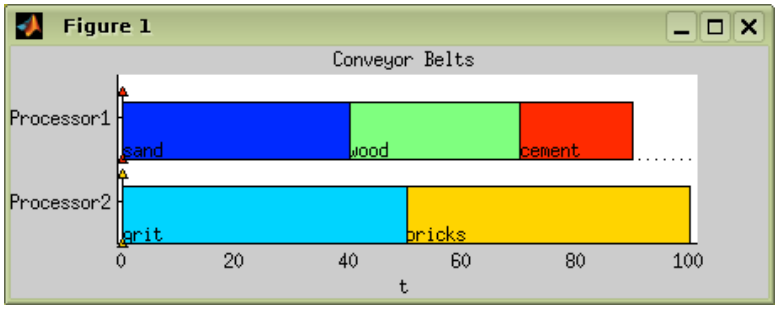
4. Call List Scheduling algorithm with `taskset` and the `problem` created recently and define the number of processors (conveyor belts).

```
>> 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 13.2.

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

Figure 13.2 Result of case study as Gantt chart



13.1.3 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 13.3 shows the mentioned problem by graph representation.

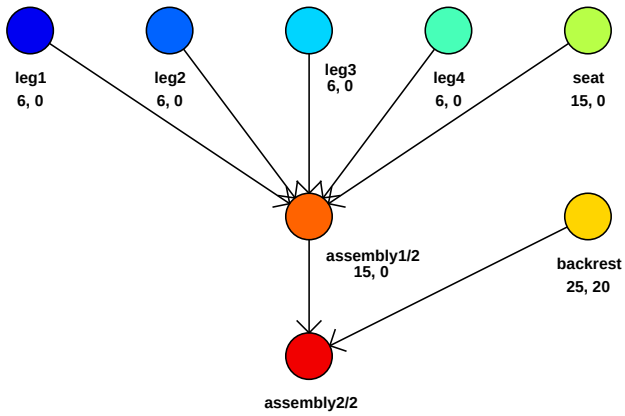
Solution of the case study is shown in six steps:

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 13.3 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 taskset object 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 as parameters 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
```

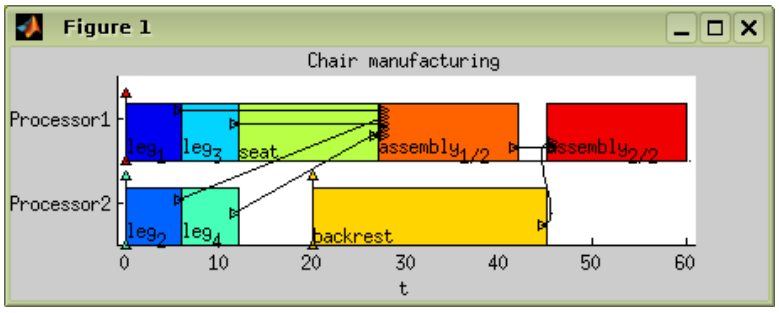
6. Visualize the final schedule by standard `plot` function, see Figure 13.4.

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

13.2 Real Word Case Studies

Two real word examples demonstrating applicability of the toolbox are shown in the following section.

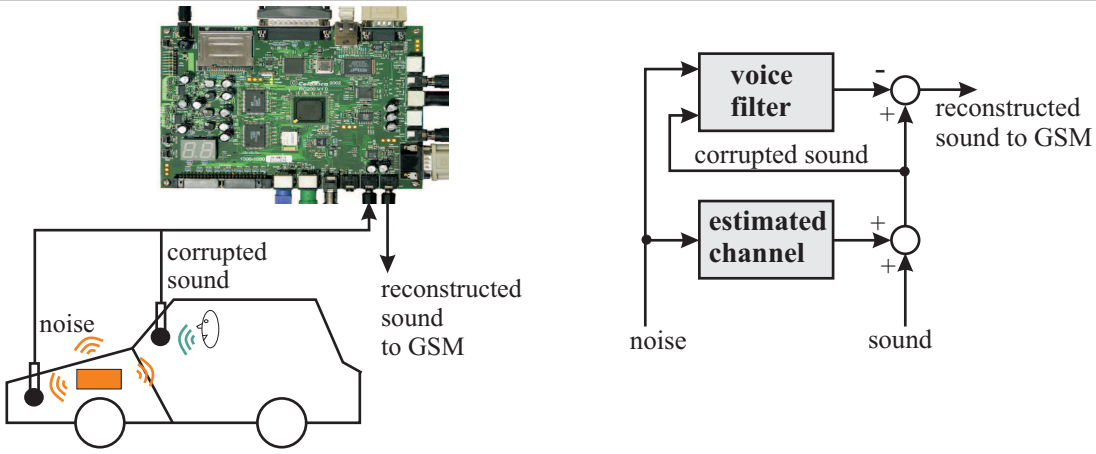
Figure 13.4 Result of case study as Gantt chart



13.2.1 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 13.5 [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 other hand, 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 unlimited.

Figure 13.5 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 13.6) 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 13.6 shows the inner loop of RLS algorithm. Data dependencies of this problem can be modeled by graph `rls_hsla` in Figure 13.7, 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 13.6 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 13.3 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.

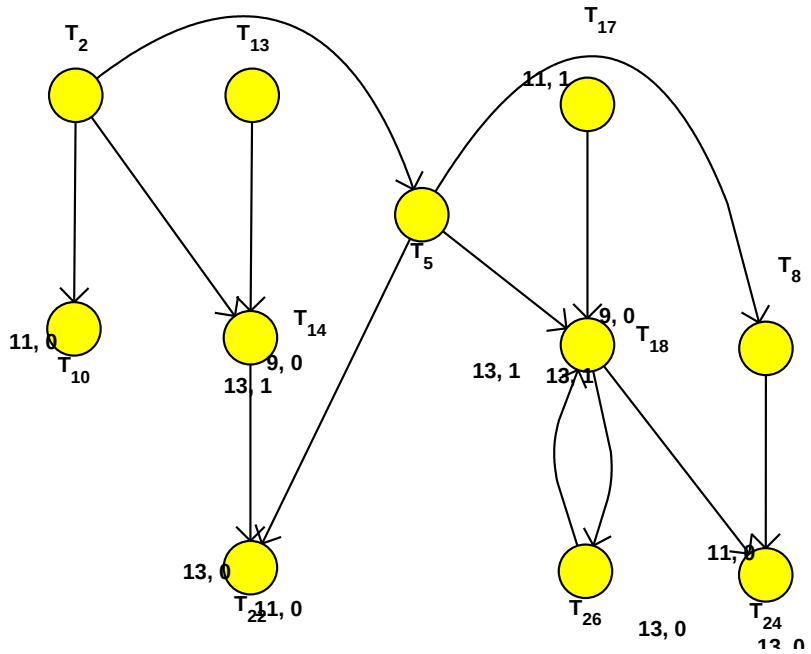
   ```
   >> T = taskset(rls_hsla,'n2t',@node2task,'ProcTime', ...
                 'Processor','e2p',@edges2param)
   Set of 11 tasks
    There are precedence constraints
   ```

3. Define the problem, which will be solved.

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

4. Define optimization parameters.

Figure 13.7 Graph G modeling the scheduling problem on one add unit of HSLA.




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

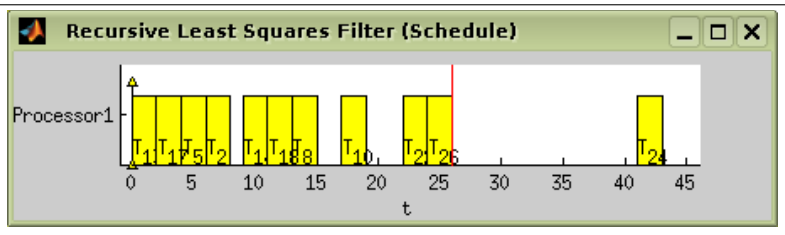
5. Call cycsch algorithm with taskset and problem created recently as parameters and define number of processors (arithmetic units).

```
>> 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. Plot the final schedule by standard `plot` function, see Figure 13.8.

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

Figure 13.8 Resulting schedule of RLS filter.



13.2.2 Visualization for the Hoist Scheduling Problem

The hoist scheduling problem, described in Section 7.18, is taken as an example for visualization with VISIS. Let us take situation with one hoist, one load/unload station and three processing stages. Minimum processing time for material in each stage is denoted by vector a and maximum processing times are denoted by vector b. Matrix C contains minimum times needed to move empty hoist between two

stations determined by row and column index. Each element of vector d denotes time needed to transport material from a station to the next one.

```
>> a = [0 70 70 30];
>> b = [0 100 200 75];
>> C = toeplitz([0 15 20 25]);
>> d = [36 36 36 51];
```

Each material transport represents one task and all previously defined parameters are stored in atribute *TSUserParam* of the taskset.

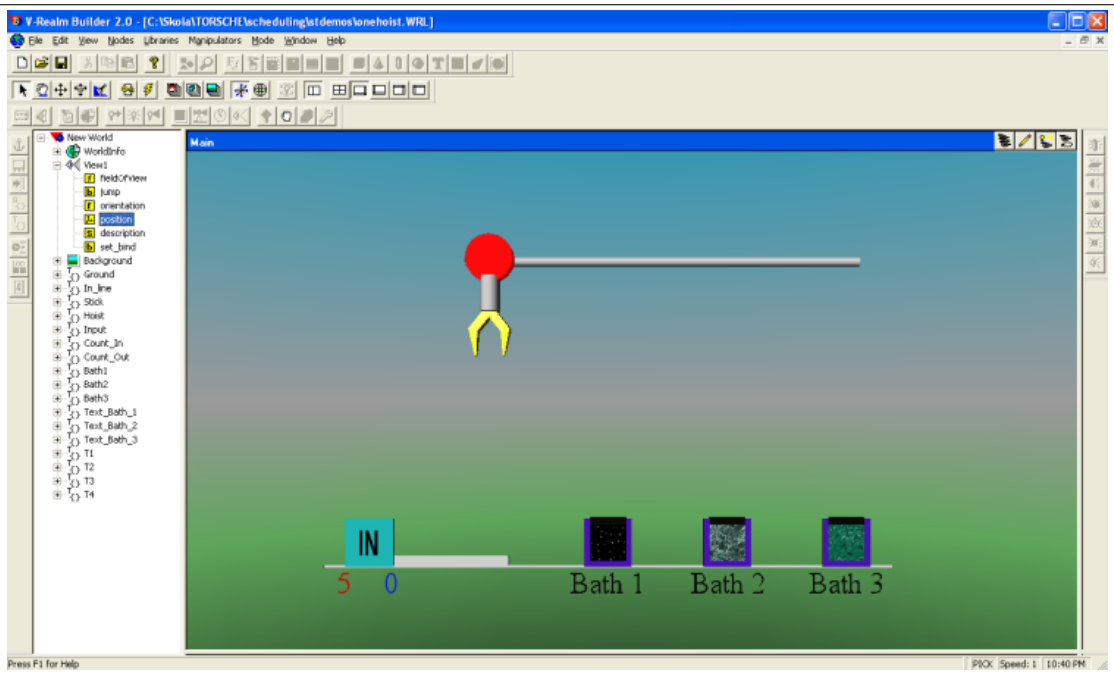
```
>> T = taskset(d);
>> T.TSUserParam.minDistance = a;
>> T.TSUserParam.maxDistance = b;
>> T.TSUserParam.SetupTime = C;
```

Now it is possible to call solving algorithm.

```
>> schoptions = schoptionsset();
>> TS = singlehoist(T, schoptions, 0);
```

The virtual reality file with graphical appearance of the problem is needed for visualization. For this purpose, VR builder is used with resulting **onehosit.wrl* file (see Figure 13.9).

Figure 13.9 Graphical definition for visualization



There are some objects that are meant to be controlled during visualization and therefore they must have unique names. Material to be handled is represented by four boxes named *T1*, *T2*, *T3* and *T4*. Hoist itself is reffered to as *Hoist*, *Arm* reffers to part of the hoist without upper red attachment, *ArmStick* reffers to middle part of the hoist and *Wrist* is name of the lower part of the hoist. These objects, their properties respectively, will be controlled during visualization and inputs of the VR block in Simulink have to be defined accordingly.

```
VRin = vrports('T1','translation','T2','translation','T3','translation' ...
               ,'T4','translation','Hoist','translation','Arm','translation' ...
               ,'ArmStick','height','Wrist','translation');
```

For each input of the VR block, there is one output of the control block to be connected with and no other control singals are needed.

```
>> ports = visiscontrolports('Output','T1trans',3,'T2trans',3,'T3trans',3,'T4trans',3 ...
                  ,'Hoist_trans',3,'Arm_trans',3,'ArmStick_height',1 ...
                  ,'Wrist_trans',3);
```

Structure of text file with Matlab commands to be executed with respect to given schedule is following:

```
#2
repeat -15:1:0
start = get_schedule(TS);
treshold = start(2)-mod(dt,period)-period*rem(start(2),period)-5;
if abs(Hoist_trans(1)-5) >= treshold
    if Hoist_trans(1) > 5
        Hoist_trans(1) = Hoist_trans(1) - 1;
    end
    if Hoist_trans(1) < 5
        Hoist_trans(1) = Hoist_trans(1) + 1;
    end
end
repeat 0:1:7
Arm_trans(2) = Arm_trans(2) - 0.5;
Wrist_trans(2) = Wrist_trans(2) - 0.5;
ArmStick_height = ArmStick_height + 1;
repeat 8:1:15
T2trans(2) = T2trans(2) + 1;
Arm_trans(2) = Arm_trans(2) + 0.5;
Wrist_trans(2) = Wrist_trans(2) + 0.5;
ArmStick_height = ArmStick_height - 1;
repeat 16:1:20
T2trans(1) = T2trans(1) + 1;
Hoist_trans(1) = Hoist_trans(1) + 1;
repeat 21:1:28
T2trans(2) = T2trans(2) - 1;
Arm_trans(2) = Arm_trans(2) - 0.5;
Wrist_trans(2) = Wrist_trans(2) - 0.5;
ArmStick_height = ArmStick_height + 1;
repeat 29:1:36
Arm_trans(2) = Arm_trans(2) + 0.5;
Wrist_trans(2) = Wrist_trans(2) + 0.5;
ArmStick_height = ArmStick_height - 1;
in 29
T2trans(1) = 5;
endparam
```

This code performs transfer of material from first to second bath with liquid and therefore it has to be assigned to second task of taskset (corresponds with key character # with task number in the beginning). Other tasks has similar coe assigned. To run the visualization, setting of the stop time and schedule period is necessary.

```
>> stopTime = 1250;
>> period = 209;
```

Now it is possible to call the main function of VISIS:

```
>> taskset2simulink(name, TS, ports, VRin, stopTime, 'Period', period);
S-Function 'S_onehoist.m' created.
Simulink file 'onehoist.mdl' created.
```

Simulink scheme is then automatically generated and open. Also the control function is automatically generated. Resulting Simulink scheme (with connected blocks) is depicted in Figure 13.10 and the virtual reality is shown in Figure 13.11.

Figure 13.10 Resulting Simulink scheme for the hoist problem visualization

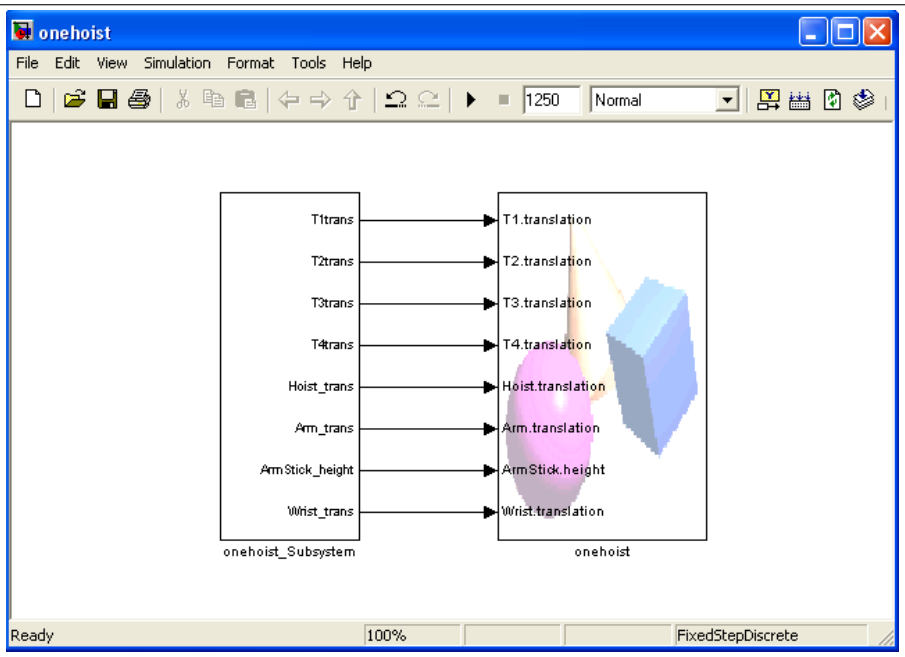
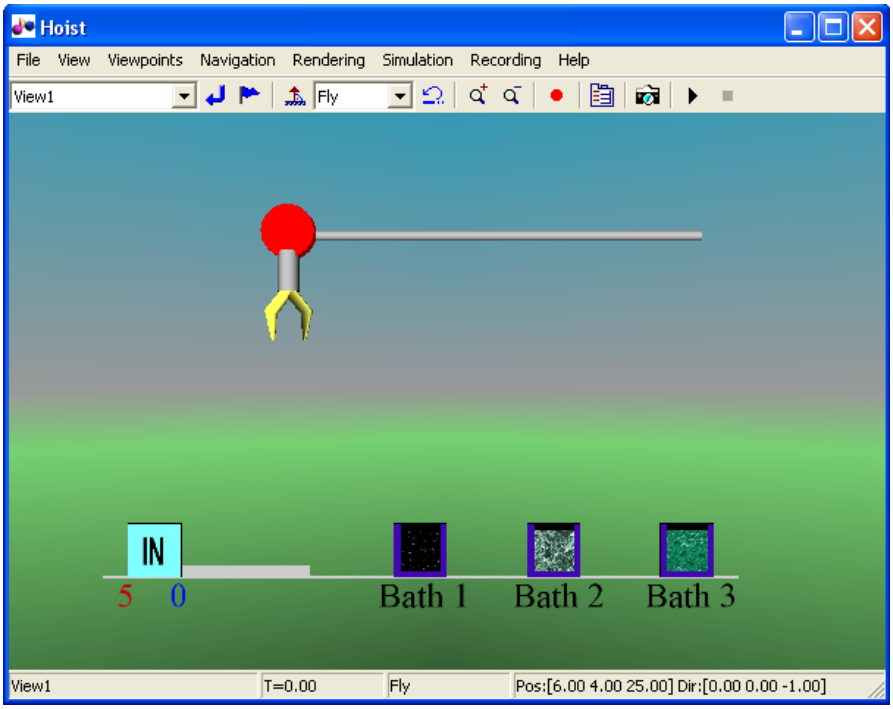


Figure 13.11 Virtual reality for the hoist problem visualization



Appendix A

Nomenclature

A.1 List of Variables

Table A.1 List of variables

Name	Description
e_i	edge of graph G
e_{ij}	edge from node v_i to v_j
h_{ij}	height (see [Cyclic scheduling (General)])
n	number of nodes and tasks
d_j	deadline of task t_i
$\tilde{d}_j$	duedate od task t_i
i, j, k	indices
l_{ij}	length (see [Cyclic scheduling (General)])
m	number of processors
p_i	processing time of task t_i
r_i	release time of task t_i
s_i	start time of task t_i
t_i	task
v_i	node of graph G
w_i	waiting time of task t_i (w_i=s_i-r_i)
C_i	completion time of task t_i
C_{max}	schedule makespan (C_{max}=max{C_i})
E	set of edges E={e_i,... , e_m}
D_i	tardiness of task t_i (D_i=max{C_i-d_i,0})
F_i	flow time of task t_i (F_i=C_i-r_i)
L_i	lateness of task L_i (L_i=C_i-d_i)
L_{max}	maximal lateness (L_{max}=max{L_i})
G	graph G = (V,E,ε)
V	set of nodes V={v_1,... , v_n}

A.2 Abbreviations

Table A.2 List of abbreviations

Name	Description
DSP	Digital Signal Processing
ILP	Integer Liner Programming
LP	Liner Programming
LPT	Longest Processing Time first
LS	List Scheduling
MIQP	Mixed Integer Quadratic Programming
TSP	Traveling Salesman Problem
VISIS	a TORSCHE tool (VIsualization and SImulation in Scheduling)

Literature

[Ahuja93] *Network Flows*, Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin, Prentice Hall, February 18, 1993, 864, 013617549X. 10.6, 10.7

[Baptiste01] *Constraint-Based Scheduling - Applying Constraint Programming to Scheduling Problems*, P. Baptiste, W. Nuijten, and C. Pape, Springer, 2001, 216. 8.4.1, 8.4.5

[Bemporad04] *Mixed Integer Quadratic Program (MIQP) solver for Matlab*, Alberto Bemporad and Domenico Mignone, Automatic Control Laboratory, ETH Zentrum, Zurich, Switzerland, 2004. 12.2, 12.5

[Berkelaar05] *lp_solve (Open source (Mixed-Integer) Linear Programming system) Version 5.1.0.0*, Michel Berkelaar, Kjell Eikland, and Peter Notebaert, 2005. 12.2

[Boruvka26a] *O jistem problemu minimalnim*, O. Boruvka, Prace Moravske Prirodovedcke Spolecnosti 3, 37-58, 1926. 10.2.3

[Boruvka26b] *Prispevek k reseni otazky ekonomicke stavby elektrovodnich siti*, O. Boruvka, Elektrotechnicky obzor 5, 153-154, 1926. 10.2.3

[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. 8.2.6, 8.3.1

[Brucker04] *Scheduling Algorithms*, P. Brucker, Springer, 2004, 367. 8.4.1, 8.4.8

[Brucker06] *Complex Scheduling*, P. Brucker and S. Knust, Springer, 2006, 284. 5.6, 5.7

[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. 8.5.1, 8.5.2

[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, 5.1, 8.2.1, 8.2.3, 8.2.4, 8.2.5, 8.2.6, 8.3.1, 8.3.3, 8.3.4, 8.3.6, 8.3.6.1, 8.3.6.2, 8.3.8, 8.3.9, 8.4.1, 8.4.4, 8.5.2

[Błażewicz83] *Scheduling subject to resource constrains: classification and complexity*, J. Błażewicz, J. K. Lenstra, and A. H. Rinnooy Kan, Ann. Discrete Math, 11-24, 1983. 1, 2.4, 6.1, 8

[CDFG05] *Control-Data Flow Graph Toolset*, Jinhwan Jeon and Yong-Jin Ahn, http://poppy.snu.ac.kr/CDFG/, 2005. 8.5.3

[CPLEX04] *CPLEX Version 9.1*, ILOG, Inc., Department for Applied Informatics, Moscow Aviation Institute, Moscow, 2004. 12.2

[Cervin06] *TRUETIME 1.4—Reference Manual*, M. Ohlin, D. Henriksson, and A. Cervin, Department of Automatic Control, Lund University, 2006. 12.6

[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. 12.6.2

[DSVF06] *Depth-First Search and Linear Graph Algorithms*, Tarjan, R. E., SIAM J. Comput, 146-160, 1972. 10.5

[Demel02] *Grafy a jejich aplikace*, Demel, J., Academia, 2002. 10.2.3, 10.15, 11.2.2

[Diestel00] *Graph Theory*, Reinhard Diestel, Springer, February, 2000, 313, 0-38-798976-5. 7.1, 10.4, 10.8

[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. 8.5.3

[Fettweis86] *Wave digital filters: theory and practice*, A. Fettweis, Proceedings of the IEEE, 270-327, February, 1986. 8.5.3, 8.5.3

[GS76] *Open Shop Scheduling to Minimize Finish Time*, Teofilo GonzalesSartaj Sahni, Journalof the Assooauonfor ComputingMachiner, October 1976. 8.4.1, 8.4.3

[Graham66] *Bounds for certain multiprocessing anomalies*, R. L. Graham, Bell System Technical Journal, 1966, 45:1563–1581. 8.3.1

[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, 6.1, 8

[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. 13.2.1

[Hanen95] *A Study of the Cyclic Scheduling Problem on Parallel Processors*, C. Hanen and A. Munier, Discrete Applied Mathematics, 167-192, February, 1995. 8.5.1, 8.5.3

[Hanzalek04] *Scheduling with Start Time Related Deadlines*, P. Šůcha and Z. Hanzálek, IEEE Conference on Computer Aided Control Systems Design, September, 2004. 8.5.1, 8.5.2

[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. 8.5.3

[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. 8.5.3

[Horn74] *Some Simple Scheduling Algorithms*, W. A. Horn, Naval Res. Logist. Quart., 21, 1974, 177-185. 8.2.1

[Johnson54] *Johnson Algorithm for Two Machines Flow Shop*, S.M. Johnson, Naval Res. Log. Quart, 1954. 8.4.1, 8.4.2

[Korte05] *Combinatorial Optimization: Theory and Algorithms*, B. Korte and J. Vygen, Springer, 2005. 10.2.1, 10.2.2, 10.9, 10.14.1, 10.14.2, 11.2.1, B, B

[Leung04] *Handbook of Scheduling*, Joseph Y-T. Leung, Chapman & Hall/CRC, April 15, 2004, 1120, 1-58488-397-9. 8.3.6

[Liu00] *Real-time systems*, J. W. Liu, Prentice-Hall, 2000, 0-13-099651-3. 3.1

[Liu02] *Cyclic scheduling of a single hoist in extended electroplating lines: a comprehensive integer programming solution*, J. Liu, Y. Jiang, and Z. Zhou, IIE Transactions, 2002, 905–914. 8.5.4, 8.5.4

[Makhorin04] *GLPK (GNU Linear Programming Kit) Version 4.6*, Andrew Makhorin, Department for Applied Informatics, Moscow Aviation Institute, Moscow, 2004. 12.2

[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. 8.3.7.2

[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. 8.5.3

[Pinedo02] *Scheduling*, Michael Pinedo, Prentice Hall, 2002, 586, 0-13-028138-7. 1, 5.1, 5.6, 5.7

[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. 8.5.3

[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. 10.16, 10.16

[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. 13.2.1, 13.2.1

[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. 8.5.3

[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. 10.16

[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. 8.5.1, 8.5.3, 8.5.3, 13.2.1, 13.2.1

[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. 8.5.3

[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. 8.3.1, 8.5.1

[Vilim07] *Global Constraints in Scheduling*, Petr Vilím, August 2007. 8.4.5

Appendix B

Reference guide

@graph/addedge.m

Name

addedge — Add edge to the graph.

Synopsis

```
graph = ADDEDGE(graph, from, to[, param])
graph = ADDEDGE(graph, from, to[, edge])
graph = ADDEDGE(graph, edgeList)
```

Description

Add edge to the graph.

Parameters:

graph
: Instance of Graph object

from
: Vector of initials nodes.

to
: Vector of conditions nodes.

param
: Cell matrix or any vector of users params. Each row includes params for one edge.

edge
: Vector of edge objects.

edgeList
: List of edges (cell): initial node, terminal node, user parameters. see also [guide:ref-GRAPH_GRAPH]

See also

GRAPH/REMOVEEDGE

@graph/adj.m

Name

adj — Return adjacency matrix of graph

Synopsis

```
matrix = ADJ(G)
```

Description

Return adjacency matrix.

Parameters:

G
: Instance of Graph object

matrix
: Adjacency matrix

See also

GRAPH/INC

@graph/between.m

Name

between — Return number of edges between initial and terminal nodes of graph or number of initial and terminal node of the edge

Synopsis

```
edge = BETWEEN(G,IN,TN)
[IN,TN] = BETWEEN(G,edge)
```

Description

Parameters:

G
object graph

IN
vector of initial node indices or char arraye of initial node name

TN
vector of terminal node indices or char arraye of terminal node name

edge
edge indices

Example

```
>> g = graph('adj',[0 0 1 0;1 0 0 1;0 1 0 0;0 0 1 0]);
>> edges = between(g,[1 4],[3])
>> [from,to] = between(g,edges)
   from =
        1
        4
   to =
        3
        3
```

See also

GRAPH/GRAPH, XrefId[?GRAPHEDIT?]

@graph/boruvka.m

Name

boruvka — is a function for searching spanning tree.

Synopsis

```
        SPANNINGTREE = BORUVKA(G,USERPARAMPOSITION)
[SPANNINGTREE USEDEDGES] = BORUVKA(G,USERPARAMPOSITION)
```

Description

Input is an object of type graph G, which is has to be weighted and every weight can be used only once. Output of the function is an graph object respresenting the spanning Tree of graph G. If a graph G was not weighted, you would have to add weightes of edges. It's necessary to enter a number of parameter like a second input parameter. In others instances will be taken a first value of edgeDatatype parameter of input graph g has to be 'double'. Variable USEDEDGES includes order of edges as was added to the spaning tree and there is order of usage in the second row.

Example

```
adding weightes of edges:
>> edgeList= {1 2,1;1 3,2;2 3,4;4 3,5;5 6,6;7 8,8;7 9,7;8 9,9};
>> g = graph('edl',edgeList,'edgeDatatype',{'double'});
>> spanningtree = boruvka(g)
```

Pseudo Code

Boruvka's algorithm

Input: A connected undirected graph G, weights $c : E(G) \to \mathbb{R}$

Output: A spanning tree T of minimum weight

Reference: <http://en.wikipedia.org/wiki/Boruvka's_algorithm>

Begin with an empty set of edges E and an empty set of edges S
1 Add the cheapest edge from the vertex in the component to another vertex in a disjoint component to S
2 Add the cheapest edge in S to E

See also

GRAPH/GRAPH, XrefId[?GRAPHEDIT?], GRAPH/PRIM, GRAPH/KRUSKAL, SPANNINGTREE_DEMO

@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/degree.m

Name

degree — return vector of graph's nodes degree.

Synopsis

```
degree    = DEGREE(graph)
[in,out] = DEGREE(graph)
```

Description

Function returns vector of graph's nodes degree.

Parameters:

graph
: Instance of Graph object

degree
: Nodes degree

in
: Number of inputs edges

out
: Number of output edges

Example

```
>> g=graph(round(0.8*rand(15)),0);
>> degree(g)
>> [in,out] = degree(g)
```

See also

GRAPH/GRAPH

@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
Keyword - 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, XrefId[?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/inc.m

Name

inc — Return incidence matrix of graph

Synopsis

```
C = inc(G)
[From,To] = inc(G)
```

Description

Return incidence matrix. Eventualy return matrixes Pre and Post. Incidence matrix C is computed as From - To.

Parameters:

G
: Instance of Graph object

C
: Incidence matrix

From
: Preincidence matrix

To
: Postincidence matrix

See also

GRAPH/ADJ

@graph/kruskal.m

Name

kruskal — is a function for searching spanning tree.

Synopsis

```
        SPANNINGTREE = KRUSKAL(G,USERPARAMPOSITION)
[SPANNINGTREE USEDEDGES] = KRUSKAL(G,USERPARAMPOSITION)
```

Description

Input is an object of type graph G, which is has to be weighted. Output of the function is an graph object respresenting the spanning Tree of graph G. If a graph G was not weighted, you would have to add weightes of edges. It's necessary to enter a number of parameter like a second input parameter. In others instances will be taken a first value of edgeDatatype parameter of input graph g has to be 'double'. Variable USEDEDGES includes order of edges as was added to the spaning tree and there is order of usage in the second row.

Example

```
adding weightes of edges:
>> edgeList = {1 2 1; 2 3 1; 2 4 2; 3 4 5; 3 1 8; 5 7 2};
>> g = graph('edl',edgeList,'edgeDatatype',{'double'});
>> spanningtree = kruskal(g)
```

Pseudo Code

Kruskal's algorithm

Input: A connected undirected graph G, weigths $c : E(G) \to \mathbb{R}$

Output: A spanning tree T of minimum weight

Reference: [Korte05]

1 Sort the edges such that $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$
2 Set $T := (V(G), 0)$.
3 For $i : 1$ to m do: if $T + e_i$ contains no circuit then set $T := T + e_i$.

See also

GRAPH/GRAPH, XrefId[?GRAPHEDIT?], GRAPH/BORUVKA, GRAPH/PRIM, SPANNINGTREE_DEMO

@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, XrefId[?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 user parameters to graph nodes from a cell or a matrix.

Synopsis

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

Description

g = PARAM2NODE(g,param) g - object graph userparam - array (a matrix or a cell) with user params for nodes. For more detail see example below.

g = PARAM2EDGE(g,param,N) g - object graph userparam - array (a matrix or a cell) with user params for nodes. For more detail see example below. N - position in UserParam of graph nodes.

Example

g = graph([Inf 1; Inf Inf]) g2 = param2node(g,{[1 2] [3 4]}) g2 = param2node(g2,[10 20],2) g2.N(1).UserParam

See also

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

@graph/prim.m

Name

prim — is a function for searching spanning tree.

Synopsis

```
              SPANNINGTREE = PRIM(G,USERPARAMPOSITION)
  [SPANNINGTREE USEDEDGES] = PRIM(G,USERPARAMPOSITION)
```

Description

Input is an object of type graph G, which is has to be weighted. Output of the function is an graph object respresenting the spanning Tree of graph G. If a graph G was not weighted, you would have to add weightes of edges. It's necessary to enter a number of parameter like a second input parameter. In others instances will be taken a first value of edgeDatatype parameter of input graph g has to be 'double'. Variable USEDEDGES includes order of edges as was added to the spaning tree and there is order of usage in the second row.

Example

```
adding weightes of edges:
>> edgeList = {1 2 1; 2 3 1; 2 4 2; 3 4 5; 3 1 8; 5 7 2};
>> g = graph('edl',edgeList,'edgeDatatype',{'double'});
>> spanningtree = prim(g)
```

Pseudo Code

Prim's algorithm

Input: A connected undirected graph G, weigths $c : E(G) \to \mathbb{R}$

Output: A spanning tree T of minimum weight

Reference: [Korte05]

1. Choose $v \in V(G)$. Set $T := (\{v\}, 0)$.
2. While $V(T) \neq V(G)$ do: ...
choose an edge $e \in \delta(V(T))$ of minimum weight. Set $T := T + e$.

See also

GRAPH/GRAPH, XrefId[?GRAPHEDIT?], GRAPH/BORUVKA, GRAPH/KRUSKAL

@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, XrefId[?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

@job/job.m

Name

job — Creation of object job

Synopsis

```
J = JOB(TS)
J = JOB(processingTime, processors)
```

Description

Constructor of job object has parameters:

- TS
Taskset with tasks that job contains

- processingTime
Vector 1xM containing processing times of tasks in job. M >=2 (Each job contains at least 2 tasks)

- processors
Vector 1xM (same size as processingTime) containing dedicated processors for each task

- J
is a JOB object

See also

TASK/TASK, TASKSET/TASKSET, SHOP/SHOP

@job/plot.m

Name

plot — graphic display of job

Synopsis

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

Description

Parameters:

J
job

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)

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

@limitedbuffers/limitedbuffers.m

Name

limitedbuffers — Creation of object limited buffers

Synopsis

```
lb = LIMITEDBUFFERS(Model, Capacity)
```

Description

To create limited buffers you have to specify Model and capacity. There is several types of model:

- general
- job-dependend
- pair-wise
- input
- output

Capacity matrix for these models are:

- general 1xQ
- job-dependend 1xN
- pair-wise MxM
- input 1xM
- output 1xM

where Q means number buffers, N number of jobs and M number of processors.

See also

SHOP/SHOP, TRANSPORTROBOTS/TRANSPORTROBOTS

@limitedbuffers/plot.m

Name

plot — Graphics display of limited buffers utilization

Synopsis

```
PLOT(LB)
PLOT(LB, C1, V1, C2, V2 ... )
```

Description

Plotting has several arguments:

LB
Limited buffers object

Cx
Configuration parameters for plot style

Vx
Configuration value

Parameters:

period
period between frames (used only in case of pair-wise model)[s]. Default 0.5s

video
Video output. Configuration value is filename

width
Width of each bar. Default 0.5

time
Detail of utilization of buffers in a given instant of time. Configuration value is a vector of several instants of time.

Example

```
>>LB = limitedbuffers('pair-wise',[0 1; 5 0]); %creating limited buffers object - model is pair-
>>%then compute schedule using appropriate algorithm
>>plot(LB) %plot utilization in time with default values
>>plot(LB,'period',0.1 ,'video','example.avi','width',1); %plot utilization in time and save fra
>>plot(LB,'time',[1 3 5]) %plot static view on utilization of buffers in times 1, 3 and 5
>>LB = limitedbuffers('general',[1 3 1]);% creating limitedbuffers buffers object - model is ger
>>%then compute schedule using appropriate algorithm
>>plot(LB, 'time',[12 1 3]) %plot utilization of buffers in times 12, 1 and 3.
```

See also

LIMITEDBUFFERS/LIMITEDBUFFERS, SHOP/SHOP

@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)

nj<=N
: Each job has maximal N tasks

gamma - denotes optimality criterion Cmax, sumCj, sumwCj, Lmax, sumDj, sumwjDj, 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/fieldnames.m

Name

fieldnames — All public properties and their assignable values and default value

See also

SCHEDOBJ/GET, SCHEDOBJ/SET

@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

@shop/add_schedule.m

Name

add_schedule — adds schedule (start times) for a shop

Synopsis

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

Description

Properties:

S
shop; schedule will be save into this shop.

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

start
matrix containing start times (size is NxM)

lenght
matrix NxM - lenght of task

processor
matrix NxM

keyword
keyword (char)

param
parameter

Available keywords 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

SHOP/SHOP

@shop/get_schedule.m

Name

get_schedule — gets schedule (starts time) from a shop

Synopsis

```
start = GET_SCHEDULE(S)
```

Description

Properties:

S
: shop

start
: matrix of start times

See also

SHOP/ADD_SCHEDULE

@shop/plot.m

Name

plot — Graphics display of shop schedule

Description

PLOT(S[,C1,V1,C2,V2...])

Parameters:

S
: shop

Cx
: configuration parameters for plot style

Vx
: configuration value

Properties:

Proc
: 0 - draw each job to one line

1 - draw each task (from all jobs) to his processor (default)

Color
: 0 - Black & White

1 - Generate colors only for tasks without color

2 - Generate colors for all tasks (default)

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)

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)

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

See also

SHOP/SHOP

@shop/shop.m

Name

shop — Creation of object shop

Synopsis

```
sh = SHOP(jobs)
sh = SHOP(jobs, robots)
sh = SHOP(jobs, buffers)
sh = SHOP(jobs, robots, buffers)
sh = SHOP(processingTime, processors)
```

Description

Constructor of shop object has parameters:

- jobs
Cell of job objects

- robots
Transport robots object

- buffers
Limited buffers object

- processingTime
Matrix NxM containing processing times of task in jobs. Each row of the matrix means one job, column means task of job. M >=2 (Each job contains at least 2 tasks)

- processors
Matrix NxM (same size as processingTime) containing dedicated processors for each task

- shop
is a SHOP object

See also

JOB/JOB, LIMITEDBUFFERS/LIMITEDBUFFERS, TRANSPORTROBOTS/TRANSPORTROBOTS

@shop/shop2taskset.m

Name

shop2taskset — Convert shop to one taskset (both precedence constraints and shedule is not changed)

Synopsis

```
ts = SHOP2TASKSET(sh)
ts = SHOP2TASKSET(sh[, separateJobs])
```

Description

Return taskset ts from shop sh. If configuration property separateJobs is set to 1 function returns taskset with tasks of jobs on same processor. If this property is not set or is 0 function returns taskset with correct dedicated processor

See also

TASKSET/TASKSET, SHOP/SHOP

@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])
handle = PLOT(...)
```

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

timeOffset
time offset. Offset of task's time.

timeMultiple
time multiple. Task's time is multiple by this value.

PLOT returns a handle to graphic object of task.

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 keywords below!

start
set of start time

lenght
set of lenght of time

processor
set of number of processor

keyword
keyword (char)

param
parameter

Available keywords 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...])
  handles = PLOT(...)
```

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

Textin
show name of task inside the task. Boolean value which is defaultly set automaticly by the type of plot.

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

TimeOffset
time offset. Offset of task's time. TimeMultiple

time multiple. Task's time is multiple by this value.

PLOT returns a column vector of handles to objects, one handle per task.

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 alocation

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
: Keyword - 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

@transportrobots/plot.m

Name

plot — Graphics display of transport robots schedule

Description

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

Parameters: T: - transport robots object Cx: - configuration parameters for plot style Vx: - configuration value

Properties:

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)

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)

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

See also

TRANSPORTROBOTS/TRANSPORTROBOTS SHOP/SHOP, LIMITEDBUFFERS/LIMITEDBUFFERS

@transportrobots/transportrobots.m

Name

transportrobots — Creation of object transport robots

Synopsis

```
tr = TRANSPORTROBOTS(TransportationTimes[, EmptyMovingTimes])
```

Description

Transportation times must be specify to create object. It is cell of transportation times matrixes. One matrix is for one processor. Values in the matrix mean transportation times between processors. Use value inf for unreachable processor. Optional parameter EmptyMovingTimes - transportation times for the way back with empty cart. Description is same as in previous case. The output tr is a TRANSPORTROBOTS object.

Example

```
>>tr = transportrobots( {[inf 1; 2 inf],[0 inf; inf 0]}, {[inf 1; 1 inf],[0 inf; inf 0]})
%Creates object TRANSPORTROBOTS with 2 robots, with defined back times
```

See also

SHOP/SHOP, LIMITEDBUFFERS/LIMITEDBUFFERS