Technical Documentation: Optimore

Isak Bohman
MAI, LiU

Victor Bergelin
MAI, LiU

Akdas Hossain
MAI, LiU

Emelie Karlsson
MAI, LiU

Claes Arvidson
MAI, LiU

Jonathan Andersson
MAI, LiU

Hendric Kjellström
MAI, LiU

January 18, 2016

Status

Controlled		
Accepted		

Document History

Version	Date	Changes	Made by	Supervised
0.1	2015-12-11	First draft	All	IB

Project Identity

TATA62, CDIO project 2015, Optimore
MAI, Linköpings Tekniska Högskola

Name	Responsibility	Phone number	Mail
Akdas Hossain	2nd Document manager	073-784 12 58	akdho545@student.liu.se
Claes Arvidson	Requirement manager	070-345 73 04	claar324@student.liu.se
Emelie Karlsson	Project manager	070-519 48 20	emeka813@student.liu.se
Hendric Kjellström	GUI manager	073-671 27 84	henkj080@student.liu.se
Isak Bohman	Document manager	076-209 40 35	isabo487@student.liu.se
Jonathan Andersson	Test manager	070-932 65 33	jonan860@student.liu.se
Victor Bergelin	Design manager	070-826 34 53	vicbe348@student.liu.se

Customer: Elina Rönnberg, Linköpings Universitet 581 83 LINKÖPING
Customer phone number: 013-28 16 45
Eximiner: Danyo Danev, phone number 013-281335, e-mail address danyo.danev@liu.se
Supervisor: Torbjörn Larsson, phone number 013-282435, e-mail address torbjörn.larsson@liu.se

Contents

1 Introduction

The following document constitutes the most comprehensive description of this project to date.

The aim of this project were to find a feasible solution to a large scheduling problem. In order to do this the group tested two different approaches. The first one was to solve it by using an LNS algorithm and the other one a so called Tabu heuristic.

2 The overall file system

This section highlights the overall system and how to proceed with implementation of more algorithms and further tests.

2.1 Implementation concepts

The Optimore software foundation lies on a standard Linux distribution. This system uses built in shell commands to run code developed in AMPL (LNS) and Tabu algorithms only run in MatLab. All algorithms are launched through an interface MatLab test and analytics.

All peaces of this system was implemented with consideration to further development. The interface can handle more algorithms being tested with it, as it is a universal launcher with a standardized format. It include both logging and analyzing results for the used heuristics.

2.2 Directories

The project uses a deploy ready file structure to meet standard formats. By using this implementation method and relative paths, a parallel working style could be adopted with modules developed and integrated simultaneously. A brief explanation of the folders and their function follows in Table 1.

Table 1: Directory structure

```
target/                     <- all results and logs
        logs/               <- full logs are always saved here after a run
        resultplots/        <- a directory to automate figure prints of results
        results/            <- full data tables of results from runs
shell/                      <- scripts for automation
src/                        <- source code
        main/               <- main source directory
                ampl/       <- mathematical model directory
                guitest/    <- launcher and data gui directory
                resources/  <- external resources
                tabu/       <- tabu algoritm
        test/               <- sources related to test
                resources/  <- resources to tests
                testdata/   <- all test data files
```

Main run files are located in the project root folder and accesses other file further down into the file structure by including paths recursively. All path specifications are relative to the root folder.

2.3 Target folders

Upon every new run, the main launch script creates a new log file. This file uses the following structure: log_2015-11-30T14-47-51, using date and time as name. All captured errors are logged in this document to monitor unexpected failures more efficiently.

Every test uses a result file directory with standardized name as specified here: results_2015-11-30T10-26-01.

Each directory will then contain one file for every combination of test data and algorithm that has been tested. The file names look as follows: T_A20_3_2015-11-16T15-02-57. The first letter is T, for Tabu search. L is used for LNS algorithm and M for Mathematical model. Following this letter is the name of the test data file, including the data type, complexity, data set id and date is was created.

Every result file will then include six columns, and one row for each iteration. The columns are: iteration id, total cost, overlap cost, dependency cost, bounds cost and time passed since start of algorithm.

2.4 Adding models

To add more algorithms to this testing and analysis platform use the following short setup guide: 1. Add new files to src / main / *your new directory*. 2. Implement connections to you external scripts using the mainlauncher.m file. To graphically integrate the new algorithms, also make new buttons in the GUI interface using the matlab GUI editor on the existing file. 3. Verify that your algorithm saves result and log files using the specified standard. All paths must also be relative to the root folder.

3 LNS Algorithm

When solving this specific problem by using LNS heuristics the group had a few different but still closely related ways of doing this. They consisted of either using different destroy functions or changing the way of calculating the parameters in the objective function. In this project a repair function was not developed since CPLEX is used as the repair function. The focus was therefore on creating a good mathematical model and a good destroy function.

3.0.1 Variations of the destroy function

The destroy function destroys according to a given destroy degree chosen by the user. The destroy degree is a value chosen between 0 and 1, where the value 0 corresponds to not destroying the solution and 1 corresponds to destroying a large part of the solution. Based on the destroy degree, the function calculates how many variables it needs to unfix depending on the instance size. An integer called "destroy count" represents that amount. The group has implemented two different ways to choose which variables will be destroyed. However, both ways unfixes one variable at a time until the number of unfixed variables has reached the "destroy count".

3.0.2 Random destroy

In the first version of the destroy function it chooses which variables to unfix randomly by guessing index numbers. Each variables index is saved in a list and before a new variable is unfixed the function looks through the list to make sure it is unique. This ensures that it will always unfix the correct number of unique variables.

3.0.3 Active destroy

In the second version of the destroy function an active choice on which variables to unfix is made. The active choice is made by finding all non-zero "violation variables", and unfixing the variable corresponding to that constraint. This makes sure that the function destroys the solution where the violations occur. If the number of violation variables are fewer than the destroy count, additional variables will be unfixed by using the random destroy function.

3.1 AMPL code structure

The LNS heuristic is written using syntax according to AMPL standards. Three files are needed to be able to solve a problem using AMPL, a run file (.run), a data file (.dat) and a model file (.mod). The .dat file is generated through the data generation GUI. The model file contains a model which has been developed using a base model given by the customer. The majority of the actual programming of the heuristic is done in the .run file and this is where the majority of the future work will be implemented. If one chooses to use the same heuristic one can simply run the test launcher with the files as they are. However, if changes needs to be made, such as the destroy degree, changes in the run file needs to be made. To debug the AMPL code the group suggests to add display lines followed by an exit command.

3.1.1 Files for LNS

All relevant files for the LNS heuristic can be found in src/main/ampl directory. The clone files exists to in a simple way change out which data and model file to use as these files needs to be specified in the .run file. Depending on the choice made by the user the GUI calls the appropriate '-main.m' file, such as 'LNSmain.m'. The '-main.m' file calls on the appropriate .run file and its clone, it makes the appropriate changes in the clone and then runs it using terminal commands. Comments can be found in all .run, .mod and .m files.

Table 2: src/main/ampl

File	Description
LNSmain.m	MATLAB script for LNSModel
LNSlistmain.m	MATLAB script for LNSModelList
MathModelmain.m	MATLAB script for Mathmodel
LNSModel.run	Run file using LNS with 'random destroy'
LNSModelList.run	Run file using LNS with 'active destroy'
LNSmodel.mod	Model file for LNS, with objective function
Mathmodel.run	Run file for the mathematical model without any implemented heuristic
Mathmodel.mod	Model file for the mathematical model, no objective function

3.1.2 How to use the code

The best run file as of now is titled 'LNSModelList.run'. It is using the active destroy function and is suggested to be used as base for further development. In the .run file one can see which .mod file is used and if needed changes can be made in them. The group suggests to read up on AMPL syntax as many of the major hurdles this group faced were due to misunderstanding of how AMPL works.

4 Tabu Search algorithm

The files for the tabu search algorithm are placed in the src/main/tabu directory. How these files function together will be described in this section.

4.1 Files

The top level of the file system will look like Table 3, where the different folders also are described. In the subsequent tables, all files are described and information is given about from where they are run and what other files they run in order to give an understanding of the structure of the tabu solver. Output and input of each file can be read in the function definitions on the top of each file.

The function *tabumain.m* is the actual tabu solver, which calls the other functions in order to find a solution to a given tabu problem. The function creates an initial solution and improves this for a maximum number of iterations or until a solution for which the cost is zero is found. *tabumain.m* also writes to target folders and plots the progress of the solution.

Table 3: Top level: src/main/tabu/

Top level	Description
Display/	Functions used to visualize progress of tabu solver
CostFunction/	Functions used to calculate costs
Initialization/	Functions used to initialize models, figures and result and log files
Models/...	All tabu models developed
tabumain.m	Main function for solving the scheduling problem

Table 4: Files in: src/main/tabu/

Level 1	Run from	Runs
tabumain.m	mainlauncher.m	Display/, Initialization/ and Models/

4.1.1 Cost functions/

Table 5: Files in: src/main/tabu/CostFunctions/

Level 2: CostFunctions/	Run from	Runs
BoundsCost.m	CostFunction.m	-
CostFunction.m	Models/	BoundsCost.m, DependencyCost.m, OverlapCost.m
DependencyCost.m	CostFunction.m	-
OverlapCost.m	CostFunction.m	-

4.1.2 Display/

The display functions can be enabled or disabled in *tabumain.m*. The visualization tool uses the costs generated by the cost functions and the original task data in order to create graphs over the progress of the solver.

Table 6: Files in: src/main/tabu/Display/

Level 2: Display/	Run from	Runs
DisplayCostFunction.m	tabumain.m	-
DislayCurrentSolution.m	tabumain.m	-
DisplayIntervals.m	tabumain.m	-

4.1.3 Initialization/

In an initial phase of the tabumain-function figures and models are set up and files for logs and results are created in the target directory. Subsequently, an initial solution is created through *InitialSolutionLauncher.m* which uses given parameters to chose what initial solution will be used. At this the only initial solution is the one given by *SimpleSortandPlace.m*, which places all tasks in the middle of their allowed interval.

Table 7: Files in: src/main/tabu/Initialization/

Level 2: Initialization/	Run from	Runs
CreateFigures.m	tabumain.m	-
CreateModel.m	tabumain.m	-
CreateResult.m	tabumain.m	-
GetData.m	tabumain.m	-
GetLog.m	tabumain.m	-
InitialSolutionLauncher.m	tabumain.m	SimpleSortandPlace.m
SimpleSortandPlace.m	InitialSolutionLauncher.m	-

4.1.4 Models/

The tabu solver has 10 different model folders which contain model phases. Each model contains two or more phases, which are displayed in the level 3 column in 8. Some model folders contain more than one model as phases are paired {M1_1, M1_2}, {M1_10, M1_20, M1_30} and so on. The phases that are modeled are unique classes with local variables and functions. How a class looks like is described in the next section.

Only one model can be used at the time and the solver will switch between its phases in order to solve the scheduling problem.

Table 8: Files in: src/main/tabu/Models/

Level 2: Models/	Level 3:	Run from	Runs
BasicModel/	BASIC_1	tabumain.m	CostFunction.m
	BASIC_2	tabumain.m	CostFunction.m
M1/	M1_1	tabumain.m	CostFunction.m
	M1_2	tabumain.m	CostFunction.m
	M1_10	tabumain.m	CostFunction.m
	M1_20	tabumain.m	CostFunction.m
	M1_30	tabumain.m	CostFunction.m
M2/	M2_1	tabumain.m	CostFunction.m
	M2_2	tabumain.m	CostFunction.m
M3/	M3_1	tabumain.m	CostFunction.m
	M3_2	tabumain.m	CostFunction.m
	M3_10	tabumain.m	CostFunction.m
	M3_20	tabumain.m	CostFunction.m
M4/	M4_1	tabumain.m	CostFunction.m
	M4_2	tabumain.m	CostFunction.m
	M4_10	tabumain.m	CostFunction.m
	M4_20	tabumain.m	CostFunction.m
M5/	M5_1	tabumain.m	CostFunction.m
	M5_2	tabumain.m	CostFunction.m
	M5_10	tabumain.m	CostFunction.m
	M5_20	tabumain.m	CostFunction.m
	M5_100	tabumain.m	CostFunction.m
	M5_200	tabumain.m	CostFunction.m
ModA/	MA_1	tabumain.m	CostFunction.m
	MA_2	tabumain.m	CostFunction.m
ModB/	MB_1	tabumain.m	CostFunction.m
	MB_2	tabumain.m	CostFunction.m
ModC/	MC_1	tabumain.m	CostFunction.m
	MC_2	tabumain.m	CostFunction.m
ModD/	MD_1	tabumain.m	CostFunction.m
	MD_2	tabumain.m	CostFunction.m
	MD_3	tabumain.m	CostFunction.m

4.2 Models and phases

In order to launch a tabu solver, the model used needs to be specified in the launcher GUI. In Table 9 all valid models are listed. The phases are numbered so that they can easily be selected by the user. The numbers are used by *CreateModel.m* in the initialization phase in the tabumain-function and make up a complete model. The selected phases are entered as a vector in the launcher GUI.

Table 9: Table over the different tabu models.

Model name	src/main/tabu/Models/...	Phase files	Phase numbers in CreateModel
BASIC	BasicModel/	{BASIC_1, BASIC_2}	[1,2]
M1_1	M1/	{M1_1, M1_2}	[101,102]
M1_2	M1/	{M1_10, M1_20, M1_30}	[103,104,105]
M2_1	M2/	{M2_1, M2_2}	[201,202]
M3_1	M3/	{M3_1, M3_2}	[301,302]
M3_2	M3/	{M3_10, M3_20}	[303,304]
M4_1	M4/	{M4_1, M4_2}	[401,402]
M4_2	M4/	{M4_10, M4_20}	[403,404]
M5_1	M5/	{M5_1, M5_2}	[501,502]
M5_2	M5/	{M5_10, M5_20}	[503,504]
M5_3	M5/	{M5_100, M5_200}	[505,506]
MA	MA/	{MA_1, MA_2}	[3,4]
MB	MB/	{MB_1, MB_2}	[5,6]
MC	MC/	{MC_1, MC_2}	[7,8]
MD	MD/	{MD_1, MD_2, MD_3}	[9,10,11]

For example, if model MD is to be run, one needs to enter [9,10,11] in the launcher GUI. A phase is also a class, which means it has member functions and member variables. These are described in Tables 10 and 11. The member variables and functions are identical in all classes or phases.

Table 10: Member variables of a phase class

Member variables	Description
Name	Name of phase
TabuList	Tabu list of phase
Logfile	Logfile in target
Resultfile	Resultfile in target
NrTasks	Number of tasks to solve for
CostList	Vector of most recent costs, used to change phase
LowestCost	Overall lowest cost, shared between phases
NrOfBadIterationsBeforeExit	Constant, length of CostList
CostWeight	Adaptable weights in cost function

Table 11: Member functions of a phase class

Member functions	Description
<Name> (Constructor)	Initializes all memeber variables.
CreateTabuList	Creates an empty tabu list with solutions or with tasks
GetAndPerformAction	Makes a list of all possible moves of tasks and chooses the best which is not in the tabu list.
GetStoppingCriteria	Uses CostList in order to see if the solutions have been deteriorating in this phase. If so, the phase is changed.
SetTabulistCost	Transfers TabuList and LowestCost to the new actie phase when the phase is changed.
AreConditionsMet	Checks if a cost function value zero is reached.
SetWeights	Updates dynamic weights for the cost funciton with a certain interval
GetCost	Uses CostFunction to find the cost of a solution.

A few of the member variables are shared, such as LLowestCost, and in some implementations also TabuList. Most variables and functions are the same in all phases, but certain variations exists. This is how we have created different models.

4.3 Implementing new models

In order to implement a new model, it is suggested that the following steps are taken:

1. Create a folder in src/main/tabu/Models/ with a new model name.
2. Create phases in this folder. The phase files must contain phase classes and follow the same structure as the previous ones. Things such as the length of the tabulist or the number of bad iterations before the phase is changed can be modified.
3. Add phases to the file *CreateModel.m* and give them unique numbers in the same file.
4. Run launcher GUI with the phase numbers used in *CreateModel.m* on vector form.

4.4 Helping features

In order to debug code and to develop code the following features can be used:

- Plots of the solution progress. These are activated in the top of *tabumain.m*. The plots show the movement of tasks in the top left corner, the transformation of the cost function in the right hand plot and the allowed intervals in the bottom left corner. The scale of the cost function is logarithmic.
- text_launcher.m. This is a file in the root of the file system which is a simplified launcher. Currently, it can be used to launch a single tabu model for a certain datatype. It can, however, be further developed easily.
- Log files and result files. These are located in target/logs/ and target/results/. In the logs, all errors are written and in the results all the cost function values are recorded. The structure of this file is described below.

4.5 Target files from search

The target files are useful when debugging and checking results. Log files contain status messages such as errors and successes. Most files in the tabu directory write to the log file when an error occurs. One exception is the cost functions, and these could be further developed to also write errors in the log file. A result and log file is produced each time a data set is solved.

The result file contains rows and comma separated columns of data generated by running the tabu solver. The rows correspond to an iteration and the first column is the iteration number. The second row is the total cost of the solution at this iteration and the third column is the time taken so far. The three last columns are the dependency, bounds and overlap costs, given in that specific order. An example is shown in Table 12.

Table 12: Example of a tabu result file

Iteration Id	Total cost	Time taken	Dependecy Cost	Bounds cost	Overlap cost
1	6750978555	0.019849	5817848055	0	933130500
2	4385615380	0.036556	3452484880	0	933130500
3	2333606885	0.058517	1778981105	0	554625780
4	673797940	0.098221	673797940	0	0
5	365284557	0.10813	342048015	23236542	0
6	...	...	...	...	...

4.6 Running tabu search

The actual run is performed by following four simple steps:

1. Open the launcher GUI.
2. Click "tabu", select run and type of data.
3. Select model by inserting a vector with phases using one of the vectors defined above.
4. Run and look up results in the target result files or in the files generated by the GUI.

The inner working of this run involve several steps. First, the GUI passes the information about what algorithm is to be used to *mainlauncher.m* as modelParameters. The modelParameters and the other parameters indicated in Table 13 are given as input to *tabumain.m* which is the file which actually solves the problem. The mainlaucner makes only sure the tabumain function is run for each selected data set.

Table 13: tabumain.m

Input	Description	Output	Description
dataParameters	struct: name and path of data	status	1: successful, -1: error, 0: no files are run
tabuParameters	struct: active, initial solution, phase vector	-	-
logfileParameters	struct: path to log file	-	-
resultParameters	struct: path to result file and id	-	-

5 Creating data

A qualitative description of how to use the data creation interface is given in a separate manual.

Figure 1: The user interface where all data parameters for test data are specified, and where test data are created and saved.

6 The Inner Workings of the Test Data GUI

Here, we first present the files used in the creation of test data, and describe the contents of each file. Note that function names in Matlab are equal to the corresponding file name, without the .m file extension. We will use the words "function" and "file" interchangeably throughout this section.

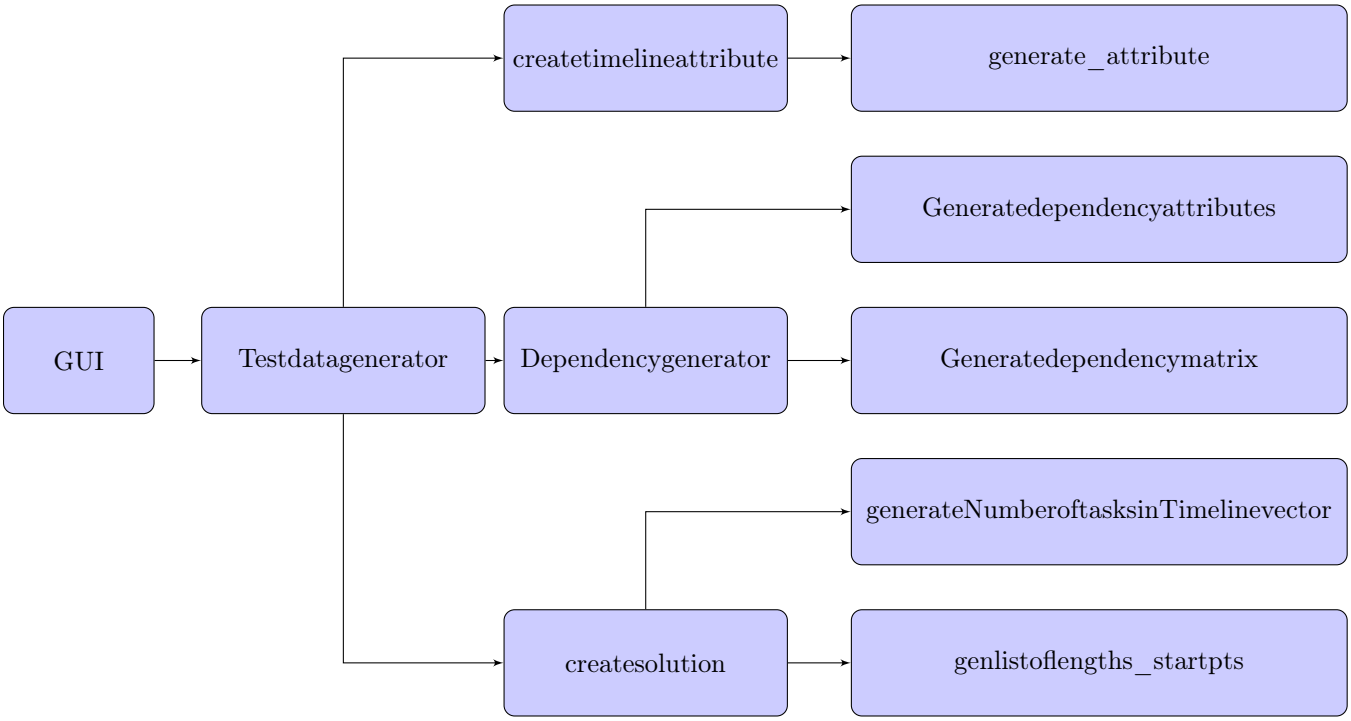
6.1 Overview of functions

File	Description
chi2_distribution.m	Function which generates chi2-distributed random variables.
createdat.m	Creates AMPL-friendly data
createsolution.m	Creates an admissible solution of the problem with specified parameters.

File	Description
createtimelineattribute.m	A function which generates a time-line attributes matrix.
Dependencygenerator.m	A function which calls the function for generating dependencies and their attributes.
generate_attribute.m	Generates a single time-line attribute, which is the allowable interval of placement of a task.
Generatedependencyattributes.m	A function which generates the attributes of dependencies, given a dependency matrix. The attributes are the admissible intervals within which two tasks in a dependency must be completed.
Generatedependencymatrix.m	A function which generates a dependency matrix.
generateNumberoftasksinTimelinevector.m	Function for generating the number of tasks on each time-line.
genlistoflengths_startpts.m	Function for generting a list of starting points and lengths of tasks.
GUI.m	The file from which the GUI is run.
ModifiedAMPLmatrix.m	Helps transform data from the original implementation into an AMPL-friendly shape.
normal_distribution.m	A function for generating a folded Gaussian distributed random variable.
test_gui.m	The rudiments of a testing GUI, which is not used.
Testdatagenerator.m	A function ordering the creation of all data.
uniform_distribution.m	Function for generating a uniformly distributed random variable.

6.2 Flow chart

Here, a simple flow chart is exhibited, which attempts to clarify the situation to the reader. The relations between the most important functions are given here. Also, it shall be noted that several of these functions use the functions uniform_distribution, chi2_distribution or normal_distribution; this is however not displayed here in order not to obfuscate the picture.



6.3 Detailed function descriptions

Here, we intricately describe the functions being used, where the function name is just the file name, excluding the .m file extension.

6.3.1 chi2_distribution.m

Input arguments	Output
mu, sigma	num

This function simply creates a number (=num) based on a chi-squared distribution (+ a fixed length) based on a prescribed standard deviation (=sigma) and expected value (=mu). The number of degrees of freedoms used is 3.

6.3.2 createdat.m

Input arguments	Output
taskfile, depfile, datafilename	none

This function transforms the created data sets into a single file which can be used in AMPL. In order to accomplish this, the proprietary function createdat is used.

6.3.3 createsolution.m

Input arguments	Output
N, L, T, generateNumberoftasksinTimelinevector, occupancy, genlistoflengths_startpts, std7, distrib7, distrib8, std8, distrib4, std4	TimelineSolution

Creates a time-line solution. This file is essentially a collection of function calls to the functions genlistoflengths_startpts and generateNumberoftasksinTimelinevector. For every time-line in the cell matrix created by generateNumberoftasksinTimelinevector, lists of lengths and starting times of tasks are created and stored.

6.3.4 createtimelineattribute.m

Input arguments	Output
TimelineSolution, L, std3, distrib3, mu11, distrib5, std5, mu3	TimelineAttribute

This function creates the attributes of tasks, i.e. the admissible intervals of placement for tasks. It does so by iteratively calling the function generate_attribute.

6.3.5 Dependencygenerator.m

Input arguments	Output
TimelineSolution, Generatedependencymatrix, Generatedependencyattributes, Ndependencies, L, N, T, rectify, distrib6, std6, mu4, std2, distrib2, mu2, constrain, chains	DependencyMatrix, DependencyAttribute

Contains function calls to the functions Generatedependencymatrix and Generatedependencyattributes and nothing more.

6.3.6 generate_attribute.m

Input arguments	Output
TimelineSolution, L, mu1, sigma1, distribution1, mu2, sigma2, distribution2	TimelineAttribute

This function generates single dependency attribute, which is an admissible interval of placement for a task on a timeline. In order to accomplish this, the function must know a number of things. First a ballpark measure of the typical lengths of intervals. Here, we have made it so that the number of intervals tasks (with lengths comparable to the length of the corresponding task) is approximately 25 %, the proportion of tasks with intervals comparable to the length of the entire timeline is also 25 %, while a medium length setting is applied to the other half of all tasks. The precise lengths are adjusted using the mu3 and mu11 parameters (which set the expected lengths of the lower and the upper limits, respectively), standard deviations are given by std3 and std5 while the distributions being sampled are dsitrib3 and distrib5. The input arguments variance and mu are not used. For each task in the given TimelineSolution, a timeline attribute is generated using the function generate_attribute; previously, a function called Attributegenerator was used, although this approach was subsequently scrapped in favor of a more general one, where the selection of distributions is made explicit.

6.3.7 Generatedependencyattributes.m

Input arguments	Output
TimelineSolution, DependencyMatrix, L, N, T, distrib6, std6, mu4, std2, distrib2, mu2	DependencyAttribute

This function generates dependency attributes (=DependencyAttribute) for every entry in the dependency matrix (=DependencyMatrix), i.e. a minimum amount of time that must pass between the end of the first task in the dependency and the onset of the second task, as well as a maximum such time. The distribution of interval lengths is determined in the same way as timeline attributes, with distrib6, std6 and mu4 pertaining to the lower limit of the interval and std2, distrib2 and mu2 determining the upper limit.

6.3.8 Generatedependencymatrix.m

Input arguments	Output
TimelineSolution, Ndependencies, rectify, constrain, L, chains	DependencyMatrix

A function which generates dependencies between tasks. This function is by far the second longest currently in use, and will require some clarification. The argument Ndependencies determines the number of dependencies that will be created. The arguments variance and mu are not in use, as of current. The argument rectify determines if dependency probabilities are determined in proportion to lengths of tasks, or if each task has an equal probability of partaking in a dependency, regardless of its length. The argument constrain determines if we are to require that all dependencies should be between tasks on the same timeline. Chains have been implemented and can be utilized by setting the argument chains to 1. The specifications of chains were provided by the project customer and are the following. The minimum length of a chain is 3 tasks, while the maximum length is 10 tasks; each chain jumps from one timeline to another exactly once. The length requirements are approximately satisfied in the current implementation, while the jump requirement is always satisfied. Before the implementation of chains, this function was fairly easy examine, which may not be the case now. If chains are not to be used, the procedure of creating chains is rather simple. In short, this approach is randomly select a candidate task from the timeline, which is to take part in a dependency. This selection process obviously depends on what have have chosen the argument rectify to be. Now, depending on what we have chosen the argument constrain to be, an 'admissible set' of tasks from which we are to choose the second task in the dependency is created. However, in the creation of this set, we don't take into account the tasks the current task may be dependent upon. Hence, we will have to create a set of tasks that the current task is dependent upon and take this into account when randomly selecting the next task in the dependency. This is essentially what is done.
If we shall utilize dependency chains, the implementation above must be modified. If the use of chains is activated, we first randomly determine if we shall begin on a chain or if a regular dependency is to be created. For the latter choice we proceed as previously described. If, however, a chain is to be created, then we first set the chain counter and the current task in the chain. Since we are at the first element of the chain, we first check if a dependency shall be generated by going backward or by going forward on the timeline. We first attempt to create one by going in the forward direction, but this may not always work out, if, for instance, the current task is the last one on a timeline. Then, we will have to check if we can create one by going in the backward direction instead. In rare, cases, even this might not work, and the chain will have to be scrapped. Subsequently, an admissible set is created, which takes into account all eventualities. This admissible set differs in its creation when compared to the admissible set when chains are not utilized. Here, we subtract the elements that the current task is dependent upon already from the beginning, so that the second task in the dependency can be chosen by simply randomizing a number from the admissible set.
In slightly more detail, the creation of the admissible set proceeds as follows. There are a few different variables that give rise to cases, and combinations of these. The various cases differ slightly, or significantly, from each other. First, there is the binary-valued direction parameter, the function of which has already been explained. Then there is the constrain parameter, which also has been explained. The parameter first_pass determines if we are to prioritize that every task receives a dependency. When it is activated, the admissible set is created so that every task involved in at least one dependency is excluded. If, however, this set turns out to be empty, we enter an amended mode where we don't demand that we only choose tasks without any dependencies. The two parameters timeline_jumps and pot_jump work in essentially the same fashion, although the functions of these have not been completely implemented (for instance, if timeline_jumps is 1, then it should be completely impossible for the chain to jump between two timelines). Experiments show that there is a high degree of variability as to the proportion of tasks involved in chains when compared to those which are not. The creation of dependencies also takes into account that it is of high priority that every task partakes in at least one dependency. This condition is essentially guarranteed to be satisfied if the number of dependencies is at least as great as the total number of tasks, which is the case for all test data sets used in the project.

6.3.9 generateNumberoftasksinTimelinevector.m

Input arguments	Output
Ntasks, Ntimelines, distrib8, std8	NumberoftasksinTimelinevector

This function call for the assignment of tasks to different timelines. The output, NumberoftasksinTimelinevector, is a vector consisting of a list of numbers which are the number of tasks that each timeline shall contain. These numbers are randomly selected from the distribution distrib8, with an expected value equal to the average number of tasks per timeline (Ntasks/Ntimelines) and the standards deviation std8.

6.3.10 genlistoflengths_startpts.m

Input arguments	Output
L, N, occupancy, distrib4, std4, std7, distrib7	start_time_list, length_list

This function is a combination of two functions which were previously used. However, a more unified approach is obtained when combing these, as done here. For a single timeline, this function generates a list of starting points and a list of lengths of tasks. This is carried out in the following way. First the level of occupancy is be used in order to determine the expected amount of space between tasks as well as the expected level of occupancy of the timelines. Subsequently, lists of task lengths and starting points are generated simultaneously in an iterative fashion, where an element is added to each list every other time in the iteration. The iterator terminates when the length of the timeline has been matched or exceeded. The lengths of tasks are determined from the distribution distrib4, with a standard deviation std4 and expected value as previously described. The spacing's between tasks are determined by the distribution distrib7, with a standard deviation of std7 and expected value as previously described.

6.3.11 GUI.m

Input arguments	Output
none	none

The main and by far the longest function. This is where the user interface is defined, using which data are created. Much of the function contains logic pertaining to rather mechanical and repetitive tasks such as handling callbacks of button presses, sliders, radio buttons and checkboxes; all of this will not be described here as a space-saving measure. The more interesting parts of the function are the following. It is possible to save data using a button in the interface, the functionality of which is detailed in the function

testdatasave_callback. This function will save a created test data set in a new folder, which in its name will contain the current date as well as the type of data created, i.e. type A, B, C, or D data, or possibly X or Z if data sets have been created in any the more customizable modes. The button "Create test data" calls upon the function testdatagen_callback and does roughly the following. First, global variables pertaining to previously saved test data sets are cleared. Then, the function Testdatagenerator is run in order to create new test data sets. However, since the user may be requesting the creation of multiple test data sets, created using the same parameters, we iterate over the number of data sets that are to be created. Subsequently, the data sets returned from the function Testdatagenerator are converted into the forms in which they are saved, when the button "save test data" is pressed. In order to illustrate the results, the function print_correct is run, whereupon three different figures are updated.

The button "Make and save multiple test data" exhibits roughly the same functionality as the button "Create test data", only that it iterates over several different modes and difficulty settings, while saving the created test data sets between iterations. The exact basis of this was decided in an ad-hoc manner due to the specifics of the project at hand.

6.3.12 ModifiedAMPLmatrix.m

Input arguments	Output
fid, pname, p, colhdrtype, varargin	count

A proprietary function used for creating data suitable for application in an AMPL setting.

6.3.13 normal_distribution.m

Input arguments	Output
mu, sigma	num

This function generates a number (=num) using a folded normal distribution. Since the numbers generated are required to be non-negative, the input arguments of standard deviation (=sigma) and expected value (=mu) are only approximately accurate.

6.3.14 Testdatagenerator.m

Input arguments	Output
N, L, T, generateNumberoftasksinTimelinevector, Generatedependencymatrix, Generatedependencyattributes, Ndependencies, occupancy, genlistoflengths_startpts, rectify, std1, std2, std3, std4, std5, std6, std7, std8, std9, mu11, mu21, mu3, mu4, distrib1, distrib2, distrib3, distrib4, distrib5, distrib6, distrib7, distrib8, distrib9, constrain, chains	TimelineSolution, TimelineAttributeList, DependencyMatrix, DependencyAttribute

This the the function which is called upon when pressing the "Create data" button in the interface. This function contains a collection of calls to other functions; among them createsolution, createtimelineattribute as well as Dependencygenerator. It shall be noted that createtimelineattribute is run iteratively, in order to obtain timeline attributes for the entire timeline solution.

6.3.15 uniform_distribution.m

Input arguments	Output
mu, sigma	num

This function takes as input a standard deviation (=sigma) and expected value (=mu), using which a number (=num) is generated from a distribution exhibiting the desired characteristics.

7 Specifications of test data sets

Test data are represented by collection of TimelineAttribute matrices, a dependency matrix and as well as an admissible solution (called TimelineSolution), which, however, is not saved. Every data set is written in a separate directory, labeled with the data type, complexity, an index as well as a time stamp. To see how data are stored, let us consider some examples. A20_2_2015-11-16T15-02-57 is a valid directory name and its meaning is the following. 'A' is the type of data which is stored inside the directory, in this case type A data. 20 is the level of complexity. 2 is an index indicating the order of the test data set in a succession of test data sets created using the same parameters, each of which data set is stored in a unique directory. The rest of the name is just a time stamp indicating the time of creation. Inside of each folder there are four files, which are described below.

The data will be arranged in columns and rows where each row corresponds to a task and every column corresponds to a data parameter according to table 2.
In addition to these, a TimelineSolution is created, which exhibits an admissible solution to the problem.

7.1 The Dependencies File (Dependencies.dat)

A dependency is described by a DependencyMatrix element and a DependencyAttribute matrix element as well as a dependency id appended as the first element, as described below. The number of rows equals the number of dependencies.

The first column is simply the dependency id, which is provided for compatibility reasons. The second column is the id of the first task in the dependency, while the third is the id of the second task; these columns correspond to a condensed form of the DependencyMatrix. The fourth and fifth columns are, in the notation established in the appendix or somewhere else, f_d^{min} and f_d^{max}, respectively, or the minimum and maximum times allowed between the end of the first task and the onset of the second task in a dependency; these columns correspond to the DependencyAttribute matrix. In a dependency, the first task has to be completed before the second task begins.

7.2 The Tasks File (Tasks.dat)

The tasks file consists of what is called a TimelineAttribute matrix, which, in turn, consists of five columns, representing the attributes of all tasks (ignoring dependencies) on all time-lines. Each row corresponds to the attributes of a certain task in a time-line. The first column corresponds only to the id's of tasks, which are not really necessary to have, yet are provided here for compatibility reasons. The second column corresponds to the minimum allowed starting times of tasks, while the third column represents the corresponding deadlines. The fourth column represents the time-lines of tasks, while the fifth represents the lengths of tasks.

7.3 The Settings File (Testinfo.dat)

This file contains all data settings that were used when creating the corresponding data set. The parameters saved in this file are, in order, the following.

Parameter	Meaning
dummy2_selector	Which type of data selected for the further simplified mode.
N	The expected total number of tasks
L	The length of any timeline
T	The number of timelines
Num_data	The number of data sets created using the same settings
Ndeps	The total number of dependencies
occupancy	The level of occupancy across all time-lines
constrain	Binary variable telling us if dependencies were constrained to stay within time-lines.

Parameter	Meaning
rectify	Binary variable telling us if dependency probabilities were rectified based upon task lengths
std1	Temporal distribution of dependencies standard deviation. Not used.
std2	Maximum time between dependent tasks standard deviation.
std3	Minimum starting time distribution standard deviation
std4	Task length standard deviation
std5	Deadline distribution standard deviation
std6	Minimum time between dependent tasks standard deviation
std7	Task spacing distribution standard deviation
std8	Task distribution across time-lines standard deviation.
std9	Spatial distribution of dependencies standard deviation. Not used.
mu1	Minimum starting time distribution expected value
mu2	Maximum time between dependent tasks expected value
mu3	Deadline distribution expected value
mu4	Minimum time between dependent tasks expected value
difficulty_number	The level of difficulty of the data created, where 0 and and 100 are the lower and upper limits, respectively.
dist_sel1	Temporal distribution of dependencies. Not used
dist_sel2	Maximum time between dependent tasks distribution
dist_sel3	Minimum starting time distribution
dist_sel4	Task length distribution
dist_sel5	Deadline distribution

Parameter	Meaning
dist_sel6	Minimum time between dependent tasks distribution
dist_sel7	Task spacing distribution
dist_sel8	Task distribution across time-lines
dist_sel9	Spatial distribution of dependencies. Not used.

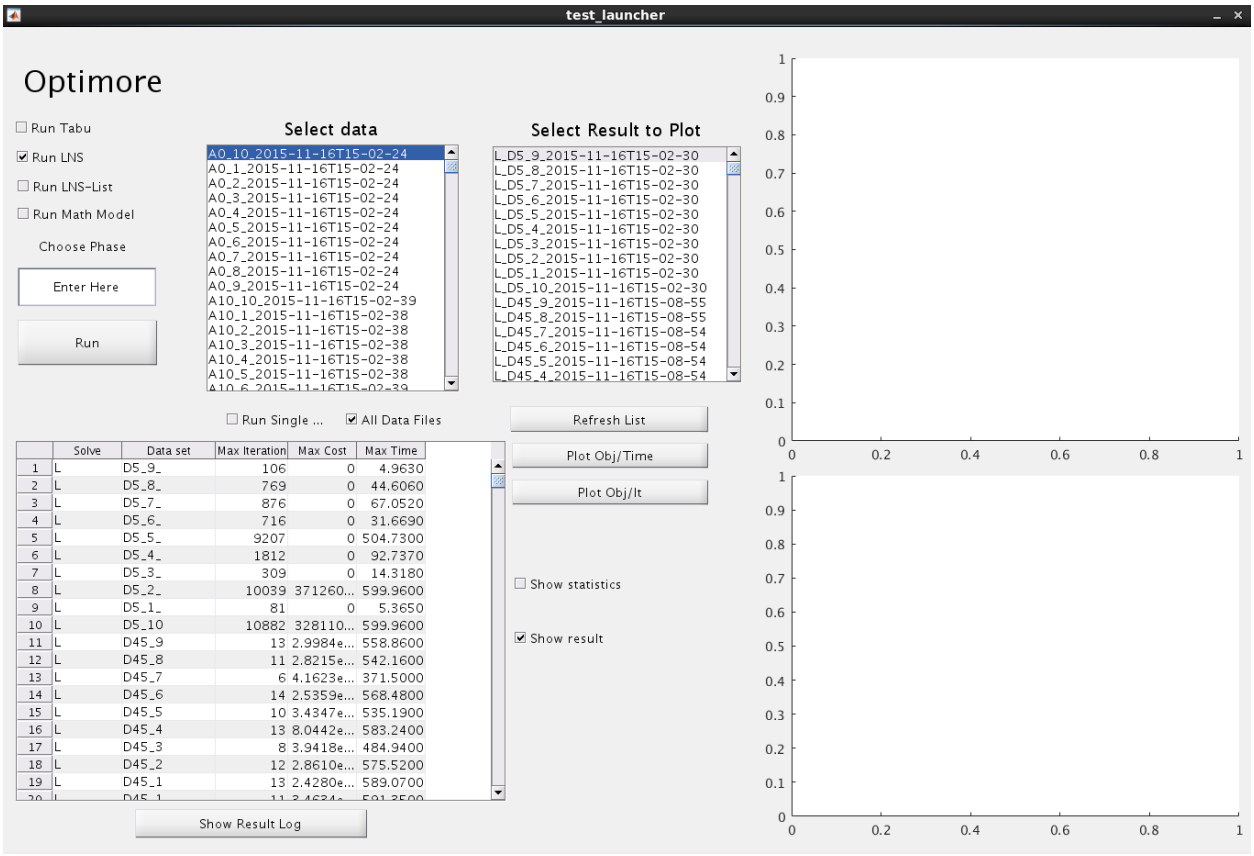
7.4 The AMPL data file (AMPL.dat)

The AMPL data file contains all of the above information, except for the information found in the settings file. The file is written in the special syntax used by AMPL. In particular, the sets being used are defined in the rather idiosyncratic way that one only encounters when attempting to get a grasp of AMPL.

8 Test Launcher GUI

The figure below shows the GUI for testing out heuristics on selected data-sets.

Figure 2: Test Launcher GUI



8.1 Files for the GUI

The following table shows the files the user needs to run the GUI.

Parameter	Meaning
test_launcher.m	The entire GUI script is within this file.
test_launcher.fig	This is the graphical content of the GUI.
t_dist_values_095.dat	A small .dat file containing statistical data for students distribution.

8.2 Parameters from test_launcher.m

Every parameter described below belongs to test_launcher.m. It is from this script we can obtain our visual results through the test_launcher.fig.

- The check-boxes furthest to the left in Figure 6 shows the four different heuristics.

Parameter	Meaning
checkbox1	Sets Tabu as active.
checkbox2	Sets LNS as active.
checkbox14	Sets LNS-list as active.
checkbox3	Sets the mathematical model as active.

- A complete list of all the available data-sets is listed in the list-box to the left in Figure 6 by looping over the entire data-set folder (main/src/test/testdata/). This fills a vector inside the listbox2_CreateFcn which is called test_data_value that displays the value in listbox2.

Parameter	Meaning
listbox2	Fills the entire listbox with data-sets.

- The "Run Single" checkbox as seen below the "Select data" list in Figure 6 enables multiple data-sets to be solved since each selected data-set is saved in a vector that pushbutton1 use.

Parameter	Meaning
checkbox10	Enables the user to pick data-sets.

- The "All Data Files" checkbox next to "Run Single" in Figure 6 makes sure to loop over every data-set available in the data-set folder (main/src/test/testdata/).

Parameter	Meaning
checkbox11	Tells the solver to solve every data-set available.

- Below the check-boxes in Figure 6 indicating the different heuristics we find a "Phase" box. This box is only available for Tabu-heuristic since it contains different approaches. In the function pushbutton1_Callback, the edit1 is saved as an input with the UI-commando "input = get(handles.edit1,'String');".

Parameter	Meaning
edit1	Waits for row vector input for phase setup.

- When the requested heuristic, data-set/-sets and phase have been chosen the "Run"-button will be waiting to be pushed. Once the user pushes the button the following sequence happen:
  - A data object is defined using struct-format, namely "dataObj.name" and "dataObj.path". The "dataObj.name" is set to either all data-sets available in the data-set folder or by selected data-sets. "dataObj.path" is set to the path of all the data-sets or selected data-sets from (main/src/test/testdata/).
  - The modelParameters for the chosen heuristic is set to active along with the input phase (only available for Tabu).
  - The dataParameters and the modelParameters works as inputs for the mainlauncher that communicates with each heuristic main-function.

Parameter	Meaning
pushbutton1	Runs the solver with selected data-sets using all above parameters.

- In the middle of Figure 6 there is a "Refresh List"-button. Before pressing the "Run"-button the list-box in the middle of the GUI does the following:
  - listbox3_CreateFcn targets the result-path (main/target/results/) where the result/results from a previous run is located. All the results are stored in a vector just like the data-sets (explained above).
  - By using the matlab function strcat the GUI (test_launcher.fig) displays each result (given multiple data-sets to evaluate) along with a tag noting which heuristics that has been used.

– listbox3_CreateFcn and pushbutton4 uses the same code but relies on the user to refresh.

Parameter	Meaning
listbox3	Is filled by default with previous run.
pushbutton4	Enables the refresh feature that fills listbox 3 with the latest results.

- Under the "Refresh List"-button in Figure 6 there are two plot-buttons, "Plot Obj/Time" and "Plot Obj/It". The script for these buttons are identical but differs in what to visualize.
  – The Tabu heuristic plots 'Total Cost','Dependency Cost','Bounds Cost' and 'Overlap Cost' versus time/iteration. The mathematical model, LNS and LNS-List can only plot 'Total Cost' versus time/iteration. This requires an if statement depending on which heuristic the user have used. Inside the if statement the load function loads the selected result file (.csv format) stored in (main/target/results/) only to be converted to log-scale and then ready to be visualized. Since the mathematical model, LNS and LNS-List uses AMPL, the output (raw result file) needs to be manipulated. This process requires the fopen function in matlab. The result file is opened using fopen, replaces every string with a white-space and saves every number. This list is now a vector with numbers and follows a simple algorithm to copy the structure from a Tabu result file.
  – The script for the plot functions can only handle one result file at a time.
  – Since the aim for the algorithm is to reach 0 as the total cost, ln(0) is set to 0.

Parameter	Meaning
pushbutton3	Creates a plot for objective function versus time.
pushbutton6	Creates a plot for objective function versus iteration.
axes3	This plot space is reserved for pushbutton3.
axes4	This plot space is reserved for pushbutton6.

- The two remaining check-boxes in Figure 6 displays "Show result" and "Show statistics". First we explain the "Show result" check-box.

Parameter	Meaning
uitable1	Waits for cell arrays to be displayed.
pushbutton5	Creates elements for uitable1 depending on checkbox13 or checkbox12.

- The result-path within (main/target/results/) is fetched and we create a matrix $(n \times 3)$ (where n is the number of data-sets) which contains the maximum iteration, final cost and final time for every evaluated data-set. We then set the uitable (specific table for Matlab GUI:s) with 'Solve','Data set','Max Iteration','Max Cost' and 'Max Time'.
- In order to retrieve the data-set name and solver name we use cellstr. The uitable handles strings very well, so we use mat2str for every number that appears. When the conversion is done we use strcat to create a string.
- It might be vital to save these results and therefore a default save was implemented. We open a new .dat file with fopen and use the option "w" which creates a file that the user can write to or read from.
- The .dat file is then being moved to the current result-folder within (main/target/results/). In every result-folder, a unique .dat file with "Show result"-table is present, given that the user pushes the "Show Result Log"-button.

Parameter	Meaning
checkbox13	Enables the "Plain Table" to be displayed.

- The following list contains information about the "Show Statistics" check-box.
  - Just like the "Show result"-check-box we start off with a $n \times 3$ matrix containing the maximum iteration, final cost and final time for every evaluated data-set. Instead of seeking information about the data-sets separately, we gather every A0- or B25-data together. This is simply done by creating a vector that tells the script how many data-sets of the same type that is present. For example, let us say that we have the following data: $A0_1$,$A0_2$,$A0_3$,$A0_7$,$A0_5$,$B10_1$, $B10_2$,$B10_3$,$B10_4$,$B15_1$,$B15_2$,$B15_2$,and $C0_4$. The vector would then look like a=[5 4 3 1]. This tells us that for a certain type and complexity of a data set there is 5, 4, 3, or 1 instances.
  - The "Show statistics"-check-box displays the following, 'Solve','Data set','Upper prediction limit','Lower prediction limit','Mean Iteration','Mean Time','Iteration standard deviation','Time standard deviation','Iteration Max','Time max','Iteration min','Time min' and 'Failurerate'.
  - Each of the above mentioned tags in the uitable follows from mathematical definitions.
  - The prediction limits were calculated by using the student-t distribution and loading t-values (included in the same directory as the Launcher) for a two sided distribution with 95% of being

inside the limit. This limit will tell the user how likely it is for the time to end up inside the interval. Since no negative time exist we set every "Lower prediction limit" to 0 if it is negative.

– This statistical table is then saved by default and appears in the same folder as "Show result".

Parameter	Meaning
checkbox12	Enables the "Statistics Table" to be displayed.