# DESIGN SPECIFICATION

Editor: Isak Bohman

Version 0.2

Status

| | | |
|---|---|---|
| Controlled | | |
| Accepted | | |

# Project identity

TSRT62, CDIO project 2015 (autumn), Optimore
Linköpings tekniska högskola, MAI

| 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

**Supervisor:** Torbjörn Larsson

Customer description, 581 00 LINKÖPING,
Customer phone number 013-11 00 00, fax: 013-10 19 02

**Responsible for customer (examinor)**: 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

# Table of Contents

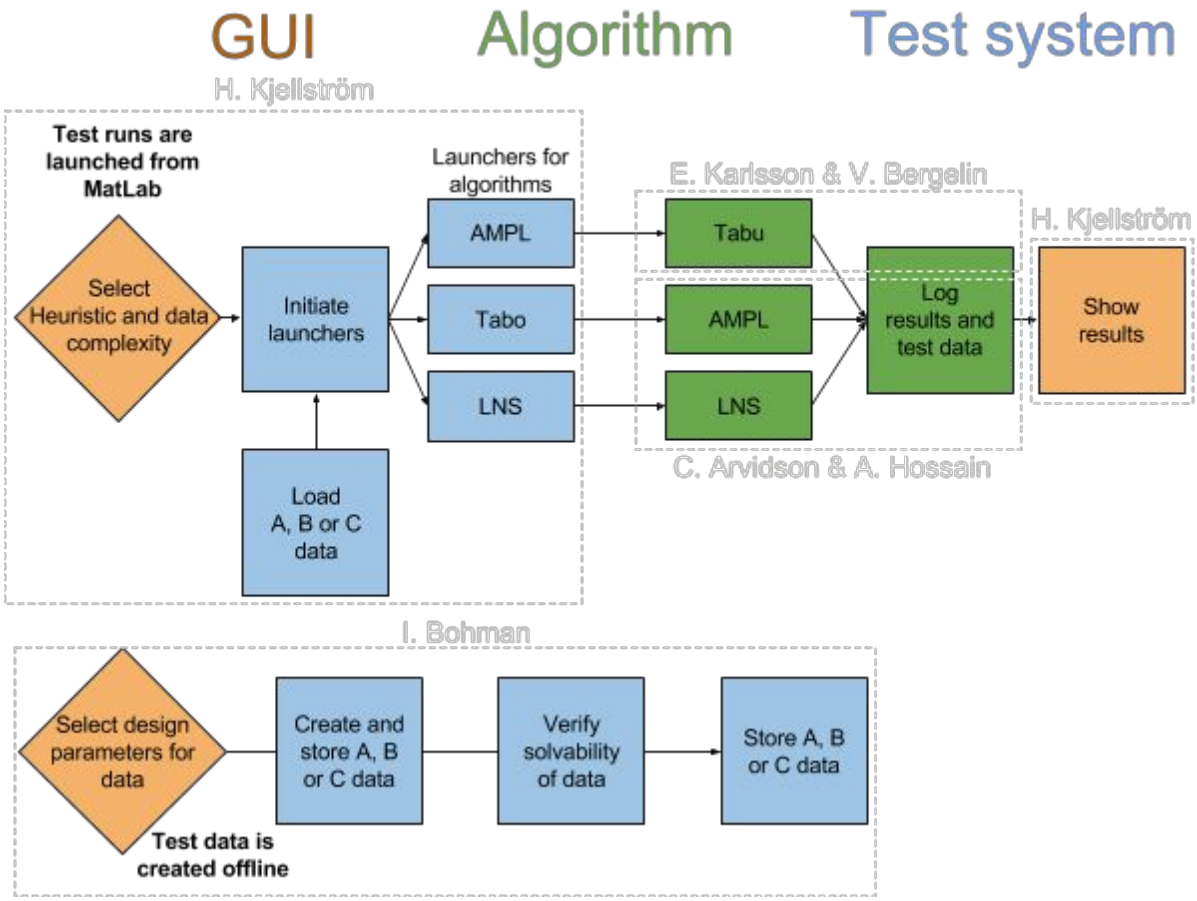
# Document history

| Version | Date | Modifications | Made by | Revised by |
|---|---|---|---|---|
| 0.1 | 2015-10-11 | First draft | all | IB |
| 0.2 | 2015-10-16 | Second draft | all | IB |
| | | | | |

# 1. Background information

Data will be created using an interactive, GUI-based application written in MATLAB. This data will then be supplied to three different solvers, based on Tabu search, large neighborhood search and a regular MIP model, respectively. Test results will be written to files and subsequently displayed in the GUI. Results from multiple test runs will be presented and construed in a final report. All data from the different processes will also be logged for reproduction of the experiments.

# 2. Overview of the syste



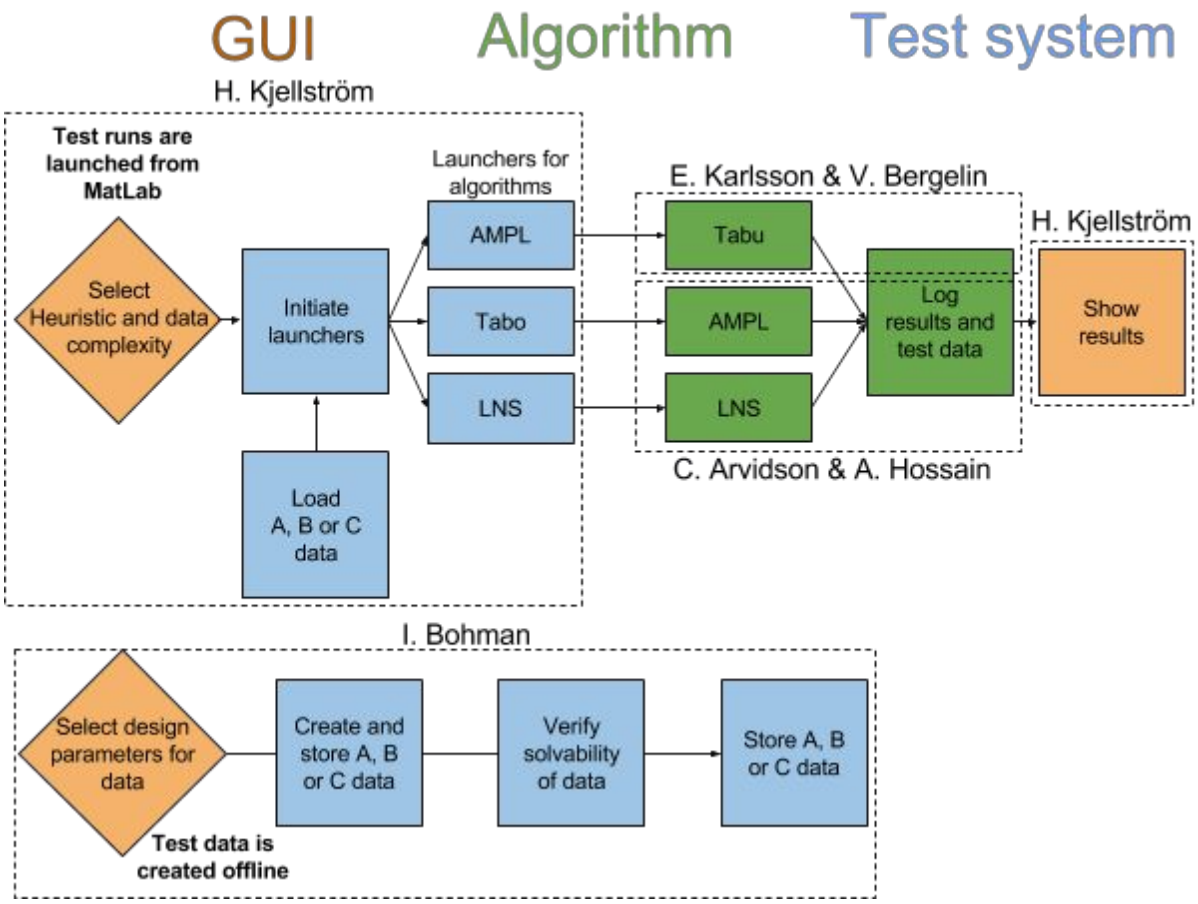



Figure: System overview

## 2.1 Short description of the program

The system is defined as an analytical platform for the evaluation Tabu search and LNS methods using various tests. Users interact with the system through a setup GUI in addition to a dashboard for interaction with the input data.

## 2.2 Implementation responsibilities

As shown in the system overview figure, implementation responsibilities are distributed mostly with with design responsibility:

Hendric Kjellström is responsible for designing and implementing the GUI. With the GUI, launchers and data loaders for the search algorithms are also included in the implementation responsibility. This

implementation also includes a scheduling service to run multiple long running tests consequently without user input. In his responsibility is also an analytical platform for visualizing and analyzing the result data and run logs.

Isak Bohman is in charge of creating the GUI for making test data, and creating the needed data for initial test according to test-design decisions from Jonathan.

The algorithm teams are responsible for documenting and implementing a command line (or similar) run interface for Hendric to access from GUI run files.

## 2.3 Coding conventions

Coding conducts point at the Google coding conventions[1] for C++ and MatLab project files. Code is always developed in branches with merge sable merges to master. The Design manager handles major merges to main repo and publishes stable- and nightly builds regularly. The readme and wiki on GitHub will be continuously developed with leanings and new conducts.[2]

## 2.4 Version managing

GIT and SVN will be used for version managing development repositories. A git repository is maintained at GitHub's public free service and used for day-to-day developments. A master/build repo will be managed by the Design manager for final testing, stable builds and system integration.

The procedure is as follows:

1. Every group develop on their own repository, push to their own branches and merge to the repository master with stable builds.
2. The design manager merges the repository master with integration tests to the main repository. While testing the system integration on separate branches, the main-master is used for stable builds.
3. Stable builds are published using SVN by Design manager.

## 2.5 Directory overview

The project uses four different repositories on GitHub.com organized under the organisation 'Optimore'. These include:

[1] Google coding conventions:
https://google-styleguide.googlecode.com/svn/trunk/cppguide.html
https://sites.google.com/site/matlabstyleguidelines/home
[2] GitHub wiki:
https://github.com/optimore/main/wiki/

1: main: the overall build and delivery repository.
2: tabu: the development platform for the tabu search algorithm
3: ampl: the development platform for the LNS and AMPL algorithm team.
4. guitest: dedicated to all gui and test data as well as test system scripts

File structure are as follows:

Read this document first:
- ./README.md

Notice for anyone trying to run or build project: dependencies etc:
- ./NOTICE.txt

Needed libraries etc:
- ./lib

Project files and scripts/builds:
- ./src/main
- ./src/main/amplbuild
- ./src/main/cppbuild
- ./src/main/gui
- ./src/main/resources
- ./src/main/resources/BashExample
- ./src/main/resources/BashExample/myBashScript.sh
- ./src/main/resources/BashExample/testBashScript.m

Files and scripts with testing with real test data:
- ./src/test
- ./src/test/README.md
- ./src/test/resources
- ./src/test/testdata

Integration test of code:
- ./src/integrationtests

Save all results and logs here:
- ./target
- ./target/logs
- ./target/results

# 3. Sub system 1: Algorithms

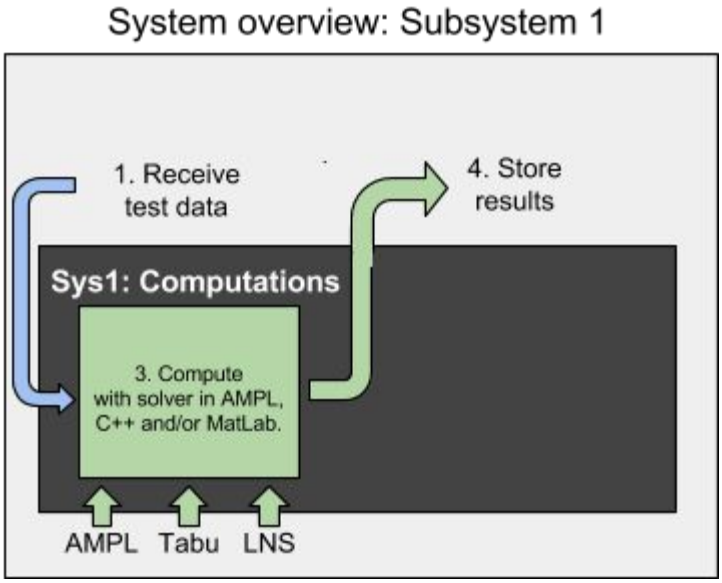



Figure:

## 3.1 AMPL and CPLEX

Any problem solved in AMPL requires three files: a model file(.mod), a data file(.dat) and a run file(.run). The mathematical model will be written in the .mod file and the data will be supplied by the test data group. The run file decides amongst other things what solver will be used -- in our case CPLEX. When solving for the mathematical model without using any heuristics, the .run file will be quite simple.

The main idea behind the mathematical model was given to the group by the customer. The group has used the main idea and developed it further, emanating in a developed model which is to be implemented by the group. The model supplied by the customer and the model created by the group are at display in attachment 1. The group has added artificial variables to the conditions (5), (7) and (8). The artificial variables will have values greater than zero and will add or subtract from the left side of each condition so that the condition is satisfied; the objective function will then minimize the sum of all the artificial variables, weighted using different constants. A high value on such a constant will make the solver try to decrease the value of the associated artificial variable. A value of zero for the objective function will then imply that all conditions are satisfied without the use of artificial variables.

The group will implement the mathematical model in two ways, both with and without artificial variables. Without any artificial variables, the objective function will be constantly zero -- whereas if

there are artificial variables, it will consist of constants and so-called artificial variables, expressing the degree to which the non-artificial variables transgress their constraints.

# 3.2 Tabu search

This section concerns the Tabu search algorithm and implementation thereof, to be performed in C++ but first tested in MatLab.

## 3.2.1 Summary

The following characterizes the method of tabu search:

- It is an intelligent search method and a flexible memory technique. The algorithm is implemented in order to avoid getting trapped at a local optimum.
- Will presumably move up- and downhill in a smarter way over the cost curve.
- Tabus prevent cyclic behaviour and getting stuck in local optima. The algorithms find move patterns, not only position in space.
- Highly dimensional, variable neighbourhood search method.
- Restrictions can be violated in the algorithm.
- Tabu search uses many types of memory, including short- and long term.
- Expect efficient discovery and a high quality of solutions.

Tabu search is a modern optimization method classified as a metaheuristic, which was first developed in the 1970s as an effective search method for solving combinatorial problems. The meta-heuristic is a local search method as it searches the neighbourhood of a solution for better solutions. What is unique about the method is that it guides the solutions out of local optima by using a tabu list. The tabu list will contain recently visited solutions, which cannot be visited again, forcing the search to explore new neighbourhoods. As opposed to other search methods, this means that the search is allowed to move in a direction that will reduce quality of the solution, in order to enable the exploration of new regions.

As is the case for most metaheuristics, tabu search does not guarantee that an optimal solution is found; however, the search is usually terminated once a certain citeria is met. This can be a specified number of search iterations or a threshold value for the solution. The method is especially suitable for scheduling problems as these are problems of a combinatorial nature.

## 3.2.2 Implementation

### Problem formulation

The tabu search method will be explored in order to schedule tasks in avionics, since it is a method highly efficient for large scale problems. It is possible to implement the method in many different

ways and we intend to try a few basic methods of disparate nature. Within each method, it will be possible to adjust parameters. All methods will, however, have a very similar structure and objective function. The objective function will penalize transgression of three constraints:

1. Tasks are overlapping
2. Task is not placed in the correct interval
3. Dependent tasks are not placed with correct spacing between them

The objective function to be minimized will then be identical to the one in the AMPL module, with artificial variables penalising violation of the constraints. A feasible solution to the scheduling problem will thus be found if the objective function is zero, indicating that all tasks have been scheduled. In the implementation, neighbourhoods for searching are defined as either swaps between starting times of two tasks, or as one or several time steps of tasks in a chosen direction. A basic implementation of this will be presented below, followed by a discussion concerning how parameters in the method may be varied.

### Basic implementation

Firstly, a starting solution has to be generated. How this is done further discussed in the section below. The solution method will subsequently be divided into two search phases. The first one will carry out a rough search, where possible swaps between tasks will be considered. The swap minimizing the objective function will be chosen. The swap will then be performed, and based on this new state, tasks will be moved a fixed number of smaller steps. This is the beginning of phase 2. All such possible steps will be considered and the best one is chosen. In this phase, the order of tasks is fixed and only their starting time is varied. The whole process will then be repeated from phase 1 until the objective function is zero or aborted if it takes too long time. After each iteration, the solution will be saved in a data file for future study of convergence.

In order for the process to not get stuck in a local optimum, tabu lists will be used in both phases. The swap tabu list for phase one will make sure that recently visited orders of tasks will not be revisited and for phase two, the time tabu list will ensure that recently visited solutions in the local neighbourhood are not revisited. The process is displayed in the figure bellow.

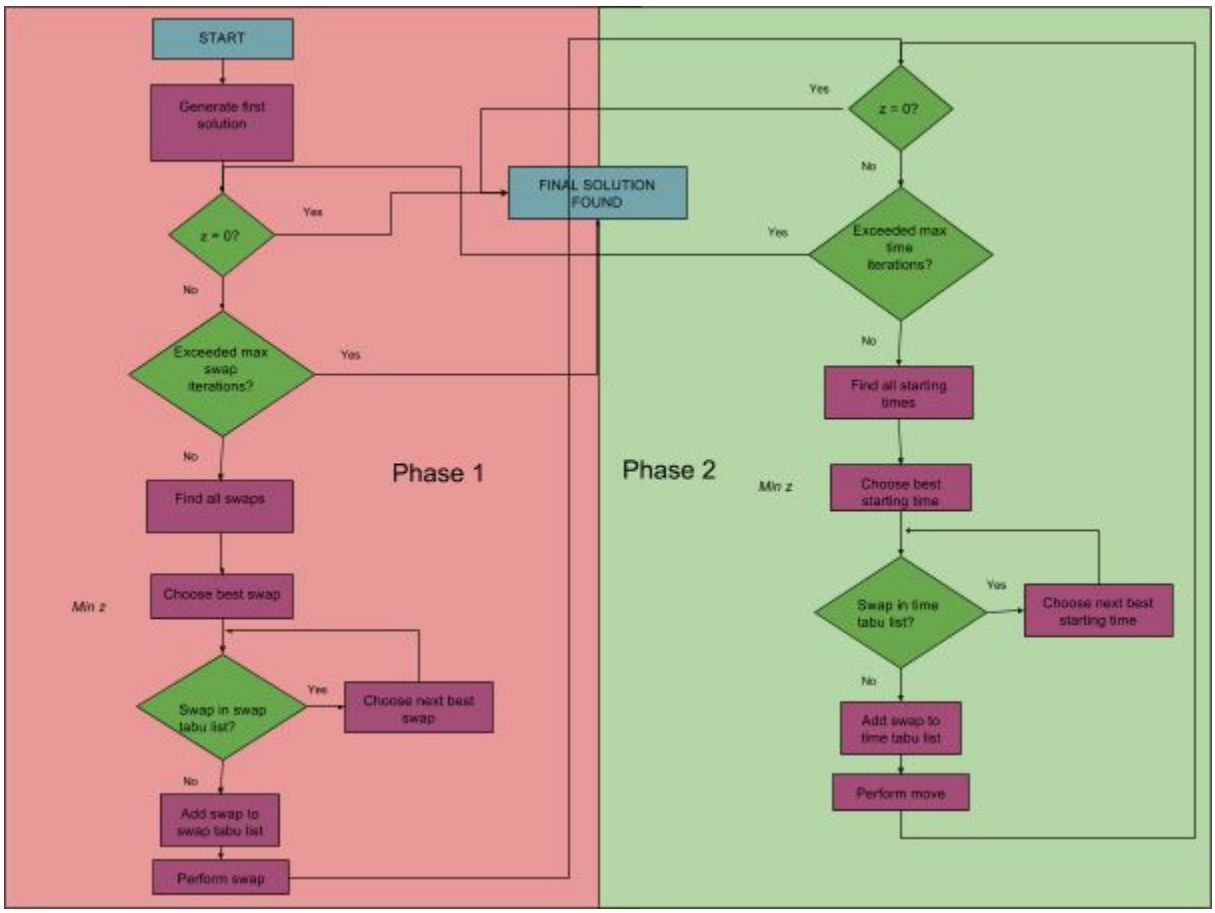



Figure: Flow chart of the rough search and the fine search phases.

## Variations of basic implementation

There exist many parameters in the implementation described above. For instance, may different focus be put on the phase one search and the phase two search. It is possible to do several swaps before experimenting with starting times in order to get a promising starting solution. It is, however, not certain that a good starting solution will yield a better final solution than a bad one although it might converge to a good solution faster.

Another variation would be to vary the weights in the objective function for different constraints. If, for example, constraint 1 is given more weight, the search will be guided towards all solutions without overlapping tasks before dealing with other constraints. This might prove to be an effective way of avoiding searches iterating between different bad solutions only with different constraints being violated.

Furthermore, the tabu lists generated may be varied in length as well as in the level of penalty they provide. It is, for example, possible to save many solutions in the tabu list, but where older solutions give a punishment in the objective function and newer ones are forbidden solutions.

### Generating a starting solution

Starting solutions can be generated in many ways. One way of doing this is to place the tasks at random, irrespective of overlapping and other constraint violations. Another way is to place them within their respective time window, thus making sure that constraint 2 is not violated.

### Additional memory structures

The implementation described above can be further developed using more sophisticated tabu search methods. The concepts of diversification and intensification are used to describe a more intelligent way of directing of the search. Diversification will make the search jump away from neighbourhoods that only generate bad results to other areas while intensification searches good areas with more scrutiny. In our implementation, the two phases can be be identified as a diversification and intensification process. In addition to this, it is also possible to use statistical information about the search in order to restart the search from a promising area (a type of diversification).

## 3.2.2 Process

1. List candidates for neighbourhood search
2. Choose from candidates which are not in the tabu list
3. Movements:
    3.1. Intelligent move, uphill or downhill in cost function. Save move in tabu list.
    3.2. When a tabu move would result in a solution better than any visited so far, its tabu classification will be updated.
    3.3. Diversification should be restricted to operate only on particular occasions. If no admissible move improving the solution exists, a diversifying move could be chosen.

### 3.2.3. Algorithm

1. Initialize

    1.1 Select a starting solution x_now, which is added to the set X.

    1.2 x_best = x_now, best_cost = cost(x_best).

    1.3 Set the history record set H to be equal to the empty set.

2. Choice and termination

2.1 Determine Candidate_N(x_now) as a subset of N(H, x_now).

2.2 Select x_next from Candidate_N(x_now) in order to minimize c(H,x)

2.3 Terminate by a chosen iteration cut-off rule.

3. Update

3.1 Re-set x_now = x_next

3.2 If c(x_now) < best_cost, perform step 1.b

3.3 Update the history record H

3.4 Return to step 2.1 and continue

**Algorithm challenges:**

- How to define the state x?
- How define and use the history record?
- How to determine Candidate_N(x_now)?
- How to evaluate c(H,x) efficiently?

**Memory structure:**

- Short term memory: Aggressive exploration
- Long term memory: Diversification
- Medium-term memory: Intensification

**Neighbourhood definition alternatives:**

- Change two tasks on one timeline?

- Cost function must be quickly evaluated
- Incremental changes are preferred
- Every point in time is an index
- Every start of task ts_i can be at any tl_i
- All errors are penalised in the cost function

### 3.2.2. Concepts

ts : time start
tl : time line
Diversification : Chose subset **S** of full solution sequence. Uses Penalty function to "jump out of the local optima".
Intensification : Chose subset **S** of the elite solutions (high quality, similar and near each other). Uses incentive function to encourage the incorporation of "good attributes".

### 3.2.3. Model parameters

Here the variable model parameters are presented. The user can interact with these settings before launching the script.

- Number of time lines
- Number of tasks
- Number of dependencies
- Type of data complexity (A, B or C)
- Number of tests to run

### 3.2.4 Implementation plan

In order to gain a better understanding of the challenges involved, a Matlab model will be implemented, which will be subsequently transferred to C++.

The model will be developed using design decisions presented in this document.

Continuous testing and data analysis will be presented alongside the development.

### 3.2.5 Future work

The model may need to be expanded upon in order to enable the handling of cyclic scheduling. This is a task of low priority and design decisions regarding this matter are therefor on hold for now.

# 3.3. LNS method

## 3.3.1 Summary

Large Neighborhood Search (LNS) is a metaheuristic that given an initial solution gradually improves upon it by alternately destroying and repairing it. The utility of using an LNS method is that it quickly moves through the solution space and returns a local optima of high quality, depending on the efficiency of the implemented functions.

Based on how large the neighborhood is, different filters can be used in order to improve the search algorithm by choosing different accept, destroy or repair functions.

By implementing an accept function it will be possible to accept a temporary solution as the current solution even though it offers no improvement of the objective function, resulting in a wider scan of the search space.

Based on a feasible solution, created after inserting non-negative artificial variables, the solution will be alternately destroyed and repaired with the aim of reaching an objective function value of zero. This will be attained when all of the artificial variables are set to zero, which represents that a feasible solution to the original problem has been found.

Implementation of the LNS will be done in AMPL using the CPLEX solver. The sole difference between the implementation of LNS and the method described in 3.1 will be found in the .run file. In LNS, the .run file will be modified so that the LNS heuristic is used when solving for the mathematical model.

## 3.3.2 Process

The process of finding a feasible solution to the problem will be as follows:

- Insert a feasible solution of the new problem where the artificial variables are present.
- Denote this variable by the current solution and set this as the best found solution.
- The destroy function will decide which variables that will be held fixed and which ones that will be free. How this could be implemented can be seen in section 3.3.4.
- The repair function will, in this case, be to use the CPLEX solver for reoptimizing the problem.
- Call the solution given by destroying and repairing the feasible solution a temporary solution.

- Accept this temporary solution as the current solution if it offers an improvement compared to the previous one, or if it is accepted based on a probability function.
- If this temporary solution is an improvement to the previously best solution, then set this is as the new best solution and reiterate by destroying and repairing the current solution.
- The iteration will stop when the objective function value becomes zero.

### 3.3.3 Algorithm

The pseudocode for the LNS heuristic will be the following:

1. insert a feasible solution, x
2. x_b = x;
3. **repeat**
4. x_t = r(d(x));
5. **if** accept(x_t,x) **then**
6. x = x_t;
7. **end if**
8. **if** c(x_t) < c(x_b) **then**
9. x_b = x_t;
10. **end if**
11. **until** c(x_b) = 0
12. **return** x_b

where x_b is the best found solution, x is the current solution and x_t is the temporarily created solution, d(x) and r(x) are the destroy and repair functions, respectively, applied on the argument x. c(x) checks by how much the solution x has exceeded the constraints.

### 3.3.4 Destroy function

The most crucial choice when implementing the destroy function is the degree of destruction. If the degree of destruction is low, then the heuristic will have trouble leaving local optima. On the other hand, if the degree of destruction is high, then the search will be very time consuming or yield poor results depending on the repair function. Due to this, a challenge for the group is to decide a satisfying degree of destruction.

When choosing the degree of destruction, there are two common approaches. The first one is to gradually increase the degree of destruction after each iteration, and the second one is to randomly choose the degree of destruction in each iteration from a specified interval depending on the instance size.

In this case the most clever approach could be to randomly choose the degree of destruction. The reason for this would be because CPLEX is the repair function, which will always yield equally good or better solution than the previous one. Both ways will probably be evaluated, since it currently is unclear which one is the most efficient.

## 3.4 Results

All algorithms will use the same format for reporting and saving the results from the runs. This convention is handled by the test requirements document and will be formulated as soon as a common format is formulated.

The data that is presented in the result files will be represented with understandable pictures in the form of graphs and maps. The result will represented in such a way that comparisons between different solvers can be made easily. The representation of the result shall also be made so that the group easily can understand what part of the scheduling problem that a specific solver has complications with.

**Representation of the objective function value on each iteration.**

In order to represent how the objective function has behaved during the process of a algorithm a graph will be used. In the graph curves of the objective function over number of iteration will be showed. When comparing several solvers. The objective function graph for each of the solvers will be showed in the same graph. Since the solvers can have different objective functions there is a need to scale the objective functions so they have roughly the same value and can be easily compared.

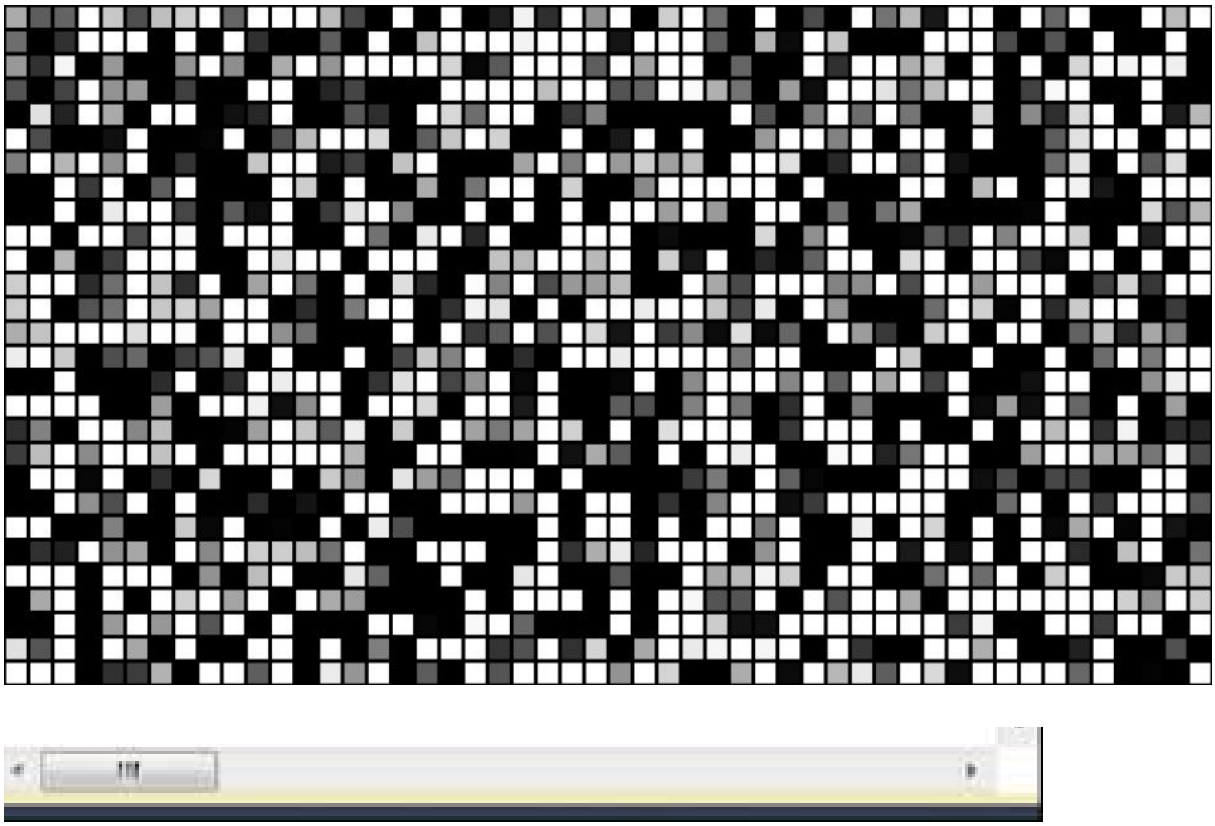
**Representation of different constraints contribution to the objective function value on each iteration**

For each solver a representation of different constraints contribution to the objective function value on each iteration is needed. This will be made in the form of several pixelated maps. For each of the test-data parts: attributes, dependency, dependencyattributes a map of the constraints that these represent will be made.

The map will be divided into several pixels with one pixel for each constraint.For each iteration the constraints contribution to the objective function will be showed. To show the contribution to the objective function of the constraints. The pixels will be filled with a certain tone off black and white or the colour green. If the pixel is green this will mean that the constraint is satisfied and contributes nothing to the objective function. We then let the constraints that contributes less to the objective function have a more whitish colour then the constraint that contribute more to the objective function.

The observer of the test shall also be able to see how the constraints contribution to the target function evolves over time. This will be solved by putting a scroller beneath the map where the user can scroll between different iteration and for each iteration a map will be showened.

**Example of pixeleted map**



Picture of the pixelated map: Each pixel represent a constraints contribution to the objective function. The whiter pixels represent constraint the doesn't contribute much to the objective function. this is because the constraint is almost satisfied and/or has less importance and therefore is weighed less in the objective function. The darker pixels represent constraint the does contribute much to the objective

function. This is because the constraint is far from being satisfied and/or has more importance and therefore is weighed more in the objective function.

The scroller beneath the map is used to look at different iterations. In the picture it is now placed furthest to the left indicating that we are lookin at the constraints before any iteration has been done.

There are no green pixels so none of the constraints are satisfied. This might indicate that we have done a bad initial guess.

## 3.5 Error handling and Run logs

All scripts and algorithms will handle unforeseen errors by implementing error handling. If the errors are of such a degree that the script crashes, error logs should still be created to capture the essential data from each test run and save log files at path: /target/logs/.

### 3.5.1 Log file format

**Name:** method_YYMMDD_HHMMSS.log

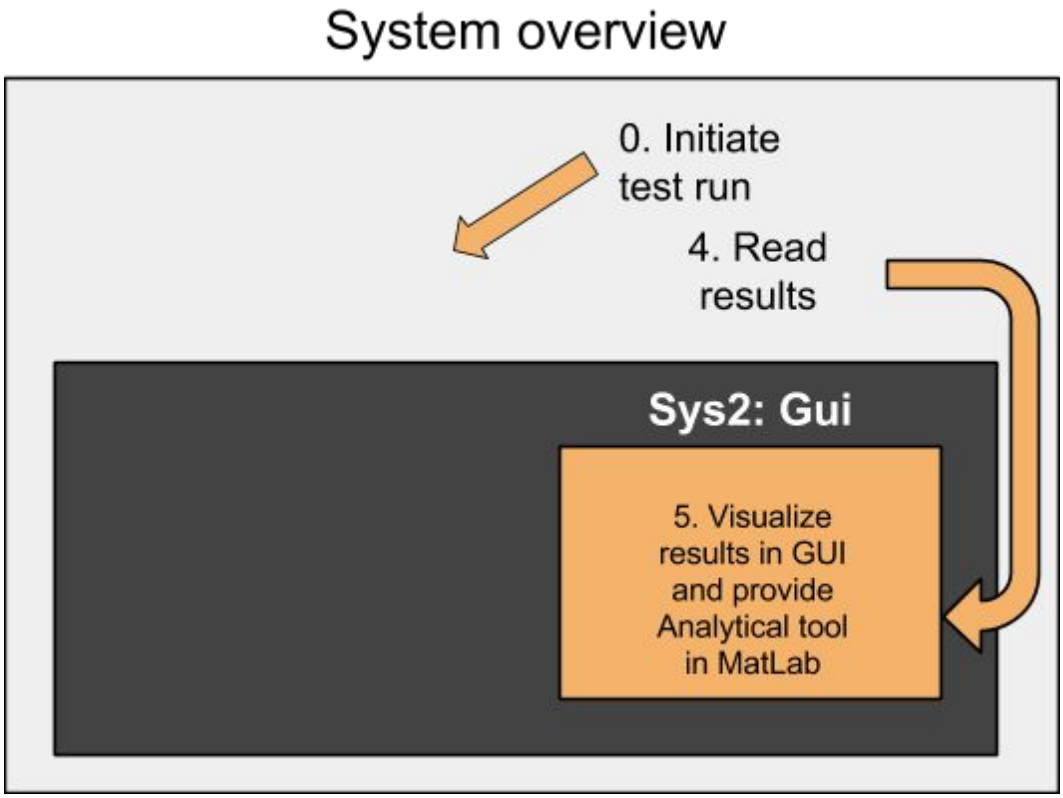
Where method is ether: gui, ampl, lns, tabu or testsys. If several logs are created on the same second, “_2”, “_3” can be added after the date.

**One line file content:** total_execution_time_for_method_in_milliseconds, status_code

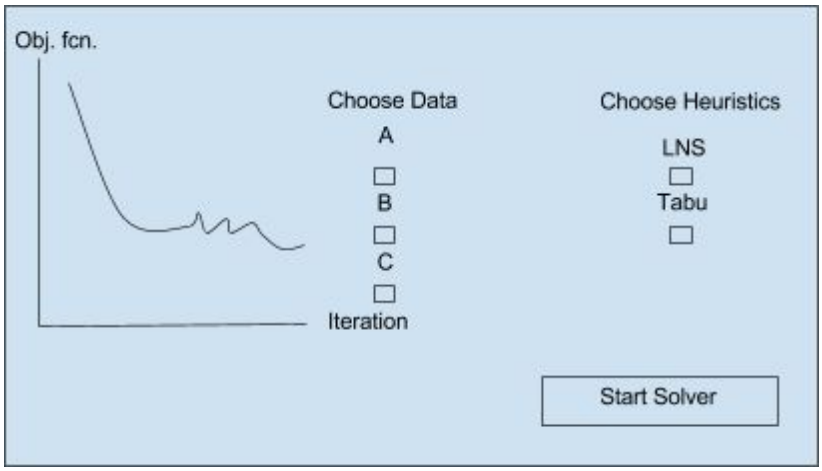
Where status_code is ether: 0: major crash, 1: OK, 2: warning or minor crash, 3-9: method specific parameters to be decided later.

# 4. Sub system 2: GUI

A user interface will be used for launching solvers, displaying test results, and for the creation, loading and saving of test data.




# 4.1 Interface




1. There will be three different levels of test data, from A to C, presented in the interface. Each specified under the requirement specification. For simplicity, only one “clickable” button

made it into the picture above. There will approximately 4 or 5 different types of data for each level of difficulty. One simply clicks on the desirable data to solve the problem live.

2. One may also choose a specific level to test the heuristics on, or one can choose different types of levels. On each level, different lengths, time lines and distributions will be available. The user can simply click on the specific type of test data which has been stored and verified.

3. When the test data has been chosen, the user must select a heuristic. A simple start-button will enable the user to start the solver of preference. Since both heuristics are written in MATLAB(R), a bash script from MATLAB(R) will communicate with AMPL to start the LNS or start Tabu search via the terminal. Since test data already have been generated by the test generator, the only part that will run live is the solver.

4. After the solver has finished, a graph on the interface will display the values of the objective function as a function of iterations. The graph will contain different colors depending on the validity of the solution. In order to compare the heuristics, both solutions will be presented in the graph.

5. Each time a solver is done, the result will be stored as a .dat file. The saved file will appear in the path that one works in.

## 4.2 Error handling and monitoring

During the computation of a data set using a heuristic, each iteration will be stored or logged in a result file. If an iterated value does not meet the conditions, a displayed char ‘Error’ will be presented in the result file. This value however, will be plotted in the final graph, but result-wise, its not valid. If a heuristic crash during a simulation, the error message from AMPL or C++ will also be presented in MATLAB(R). These logs from AMPL or C++ must be available for the MATLAB(R) interface.

# 5. Sub system 3: Test

The overall system is performing tests of different data complexity. Sub system 3 is therefor a more overview system to run and monitor the algorithms, and then visualized in the GUI. This subsystem also include the creation and specification of data.

## 5.1 Test system

The test script is focused on logging data and running multiple scripts autonomously.

### 5.1.1 System Initiation

The system runs on 64 bit UNIX with basic bash, MatLab and other standard libraries installed. The overall procedures are initiated with a MatLab Scripts.

Usage flow:

1. The user opens the GUI from matlab.
2. Test configurations and start-test-option are shown in GUI.
3. The Matlab GUI selects data and which algorithm to launch with launcher application.
4. Script selects and executes an appropriate bash script or executes other external scripts.

### 5.1.2 Algorithm launcher AMPL

The launch will be done by running the .run file in the terminal with a command line similar to the following:

- ampl >> “nameoffile”.run

### 5.1.3 Algorithm launcher Tabu

The Tabu-search C++ project is sporadically uploaded to the project repository in stable builds, and it is launched using the MatLab - C++ launcher control. The MatLab GUI launches the script by calling on an internal method.

### 5.1.4 Algorithm launcher LNS

Since the LNS solver is using AMPL the launch will look very similar to the launch used for the mathematical model, that is by running a command in the terminal which looks something like the following:

- ample >> “nameoffile”.run

### 5.1.5 Test recursion and scheduling

The test is initiated with automatic scheduling of tasks that are performed. This scheduling is performed by the heuristics given the data set. In order to keep track of the progress, a live updating plot will be presented. For each iterated value the solver computes, an updated log will also be present. One could check the graph for expected or unexpected behaviour in order to draw early conclusions.

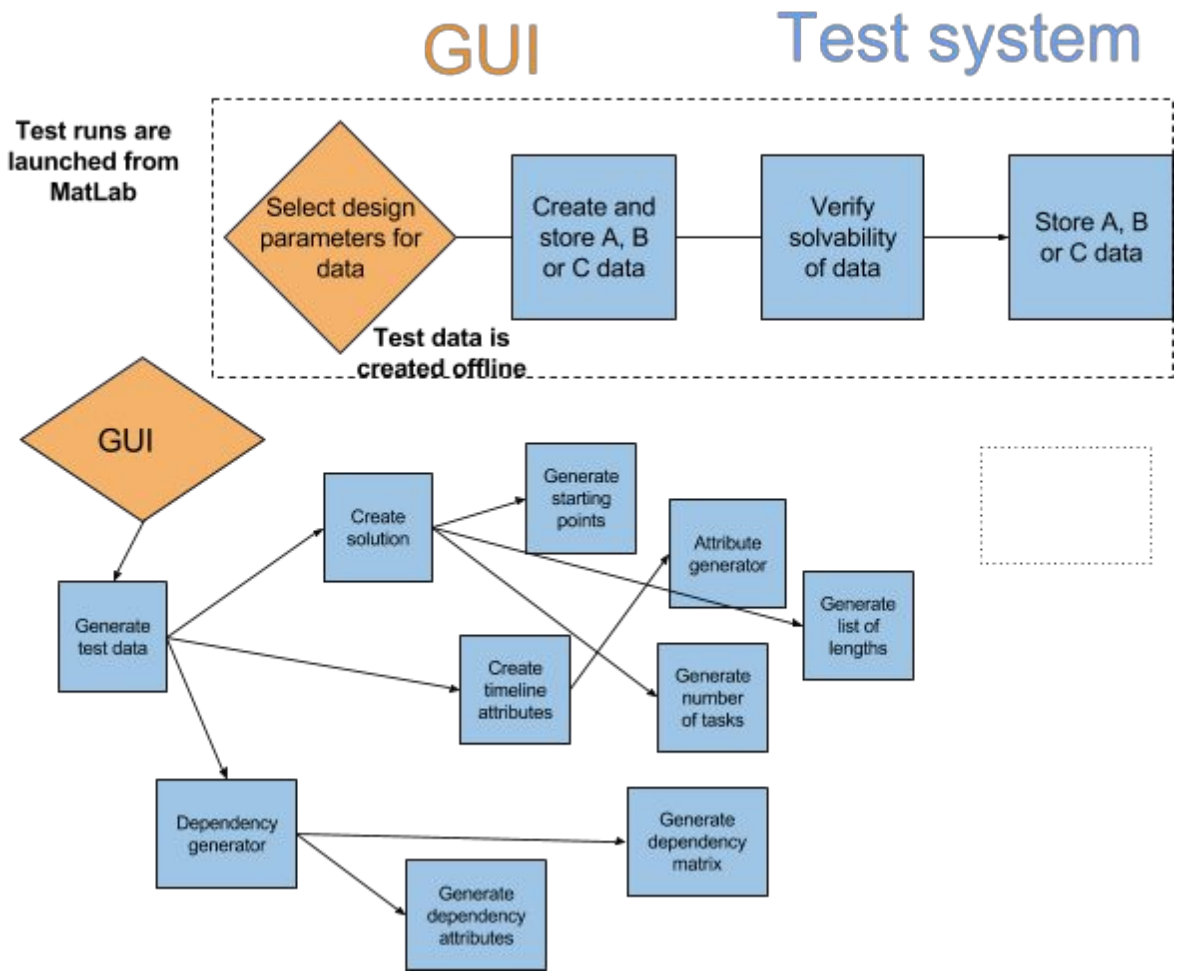
### 5.1.6 Test results

Results from the test runs are saved in a local storage unit on the server. The GUI is updated and the test finishes. Please see 3.4 Results for more details.

## 5.2 Test data

### 5.2.1 Create data

Data is created using the following principle. We only want to create data that is actually solvable, and we do this by building an admissible solution. The way in which this is done is covered here.




## Creating a solution

What distribution? With what max lengths and min lengths? The time scale in use will consist of 10^9 points in time. A valid solution to the problem is generated in the following way:

- Starting times are randomly generated based upon the order statistics of the uniform distribution over this interval. We check that all starting times generated are distinct, and separated by a small minimum distance.
- If we denote the starting time for task j by x_j, and the corresponding ending time is y_j, then we generate y_j from x_{j} and x_{j+1} by adding a randomly generated value e_j, deriving from a positive random distribution A. We check that y_j<= x_{j+1}, and if this condition is violated, we generate y_j anew.

- The first starting time is generated from the distribution A and the last ending time is generated by subtracting a number y ~ A from the last point on the timeline. Then, we check that conditions of feasibility are satisfied.
- The length of a task is computed as the difference between its ending and starting times.
- When there are multiple timelines, tasks for each timeline are generated in a similar fashion as the one considered above. Tasks are allocated based on the difference n_j-n_{j-1} where n_j is the j-th order statistic of a uniform distribution on an interval consisting of the number of tasks to be allocated.
- Dependencies, minimum starting times and deadlines are created using the methods described below. The problem that is fed into the solvers is obtained by deleting the starting times previously computed, thereby scrambling the data.

### Creating dependencies

The number of dependencies to be generated may be selected in the GUI. Dependencies are subsequently generated in the following way:

Let n be the number of dependencies to be allotted. We select the tasks to be assigned dependencies at random, in a way that will be decided later. Then, for each task to receive a dependency, we randomly select a task terminating at a time earlier and create a dependency. If a task already has dependencies, a check is made to ascertain that the new dependency does not manifold already existing ones. If no such choice is possible, we randomly select a new task to receive a dependency, and repeat the process. For each dependency, a minimum length of time from the time the first task finishes and the second task may start is generated at random, as well as a corresponding length of time for the maximum amount of time that can pass from the first task finishes to the second task starts. Dependencies may span different timelines, and will be modelled using two-dimensional random distributions.

### Create minimum starting times and deadlines

Minimum starting times for tasks are generated randomly, based on actual starting times of tasks in an admissible solution. Deadlines are created analogously.

## 5.2.3 Store data

The test data is stored as .dat files after they are generated.

We estimate that file sizes pertaining to test data files will by far be manageable.

## Level A data

Level A is the lowest difficulty of data, and no solvers should have difficulties in solving data of this type. Typically, solutions in this set will exhibit the following properties:

- Low level of occupancy for the vast majority of timelines
- Few timelines, few dependencies, few tasks

- Intervals for the spacing of tasks in dependencies will be very lenient
- Short intervals of allowable placement of tasks (which should reduce the number of iterations as there are fewer admissible solutions)

The levels of B and C will modify each of these aspects, making the problem successively more difficult.

## Level B data

Data with lots of dependencies and around 100 task and 3 timelines.

## Level C data

This is the highest level of difficulty yet conceived. This is where the heuristics will prove their mettle,

# Test data structure

### TimelineAttribute structure

A TimelineAttribute is a matrix with three columns, representing the attributes of the tasks (ignoring dependencies) in a certain timeline.

Each row corresponds to the attributes of a certain task in a timeline.

The first columns represents the earliest allowed starting time. The second column represents the last allowed starting time. The third column represents the task length.

### Dependencies

A dependency is described by a DependencyMatrix and a DependencyAttribute matrix described below

### DependencyMatrix.

A DependencyMatrix will consist of four columns and an arbitrary number of rows. Each row in a dependency matrix represents a dependency between two tasks in a sample of test data. The first column represents the id of the first task in the dependency. The second column represents the timeline that the first task belongs to. The third column represents the timeline id of the second task in the dependency. The fourth column represents the timeline that the second task belongs to.

In a dependency, the first task has to be completed before the second task begins.

### DependencyAttribute matrix

A DependencyAttribute matrix describes the attributes of a dependency involving two tasks.

The first column represents the shortest allowed time that has to pass from the time the first task finishes to the time the second task starts. The second column represents the longest allowed time that is allowed to pass from the time the first task finishes to the time the second task starts.

### Test data

Test data will be represented by a list of TimelineAttribute matrices and a dependency and also a solution. Every data set will be written in a separate file labelled with the data level and an index. For example, the first file with level A data will have the name “dataA1.dat” and the 15th file with level C data will be called “dataC15.dat”. 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.

# Creation of test data

Before a test data file is produced, a solution to the problem needs to be created, in order to ensure the solvability of the generated problem. The solution will be included in the test data.

#### TimelineSolution

A TimelineSolution is a matrix where each row represents information about the position and length of a task in a certain timeline in the solution. The order of the rows represents the order of tasks in the solution.

The first column in a TimelineSolution represents starting times of the tasks in the solution.

The second column in a TimelineSolution represents lengths of tasks in the solution.

#### Solution

A solution will be represented by TimelineSolutions, one TimelineSolution for each timeline.

In an admissible solution, tasks don’t overlap and all tasks are completely contained in a timeline.

## Solution generation

For the creation of solvable test data, a solution generator called createsolution will be implemented. This generator will have as arguments: number of tasks in the solution, timeline length, number of timelines as well as the functions described below.

**generateNumberoftasksinTimelinevector**(N=total number of tasks in a testdata,T=number of timelines in a test data)

generateNumberoftasksinTimelinevector randomly generates a vector where the first element represents the number of tasks in timeline one, the second element represents the number of tasks in timeline 2 and so on. Norm1(NumberoftasksinTimelinevector) has to be equal to N

**generatelistofstartingpoints**(L=length of a timeline, N=number of task in timeline)

A random function that generates an ordered list (Nx1 matrix) of startingpoints in a timeline that agrees with L. In other words, for all starting points it holds that 0<startingpoint<L

**generatelistoflength**(listofstartingpoints,L=length of a timeline)

A random function that given L and a listofstartingpoints generates a list (vector) with lengths where row x corresponds to task number x. Each task should agree with L and the tasks should not overlapp. In other words startingpoint+tasklength<L and startingpoint(x)+length(x)<startingpoint(x+1)

**Pseudocode**

**createsolution**(number of tasks, timeline length,numberoftimelines,generatelistoflength,generatelistofstartingpoints,generateNumberoftasksinTimelinevector)

N=number of task

L=timeline length

T=number of timelines

solution=matrix with two columns

**NumberoftasksinTimelinevector:=generateNumberoftasksinTimelinevector**(N,T)

**for n=1:T**

N=NumberoftasksinTimelinevector(n)

**listofstartingpoints=generatelistofstartingpoints**(L, N)

Listoflength:=**generatelistoflength**(listofstartingpoints,L)

**return** listofstartingpoints and Listoflength in the form of a TimelineSolution matrix

**end**

## Test data generator

**TimelineAttributegenerator**(TimelineSolutionElement, param_list)

A random function that given a TimelineSolution generates a TimelineAttribute such that the corresponding TimelineSolution element probably agree to the TimelineAttribute. Distribution parameters are contained in the vector param_list.

**createtimelineattribute(**TimelineSolution, Attributegenerator, L) creates a TimelineAttribute matrix using a TimelineSolution and an Attributegenerator

**Pseudocode**

**createtimelineattribute**(TimelineSolution,Attributegenerator,L)

repeat

TimelineAttribute:=Attributegenerator(TimelineSolution)

**if**

solution doesn't agree to TimelineAttribute

**then** send warning then **repeat**

**return** TimelineAttribute

**end:**

Given a solution and a dependency generator an Lx4, L<N*(N-1)/2 DependencyMatrix will be generated using a function called Generatedependencymatrix. Each row in the DependencyMatrix represents a dependency between two tasks. The two first columns in a DependencyMatrix represent the tasks that are dependent of each other. The task in the first column must be done before the task from the second column according to the dependency. The third column of the DependencyMatrix represents the earliest time after the first task is finished until the second task is allowed to start.

The last column of the dependency matrix represents the earliest time after the first task is finished until the second task is allowed to start.

## Dependencies

**Generatedependencymatrix**(solution)

Given a solution, Generatedependencymatrix generates a DependencyMatrix.

**Generatedependencyattributes**(solution,DependencyMatrix )

Given a solution and a DependencyMatrix, generatedependencyattributes generates a DependencyAttribute.

**Dependencygenerator(solution,Generatedependencymatrix,Generatedependencyattributes)**

Given a solution, Generatedependencymatrix and a generatedependencyattributes generatedependency generates a dependency.

**Pseodocode**

**Dependencygenerator(solution,Generatedependencymatrix,Generatedependencyattributes)**

**repeat1**

**DependencyMatrix:=Generatedependencymatrix**(solution)

**if**

something is wrong with DependencyMatrix

then send warning and go to repeat1

**repeat2**

**DependencyAttribute:=Generatedependencyattributes**(solution)

**if**

something is wrong with DependencyAttribute

then send warning and go to repeat2

**end:**

**Testdatagenerator**(number of tasks, timeline length,numberoftime lines,generatelistoflength,generatelistofstartingpoints,generateNumberoftasksinTimelinevector,Attributegenerator,) ))

The Testdatagenerator generates test-data as well as returning the solution upon which the test data is based.

**pseudocode**

**Testdatagenerator**(number of tasks=N, timeline length=L,numberoftime lines=T,generatelistoflength,generatelistofstartingpoints,generateNumberoftasksinTimelinevector,Attributegenerator,Generatedependencymatrix,Generatedependencyattributes) ))

**solution:=createsolution**(N,L,T,generatelistoflength,generatelistofstartingpoints,generateNumberoftasksinTimelinevector)

**listoftimelineattributes=()**

for n=1:T

**attributes=attributes+createtimelineattribute((**n:th TimelineSolution in solution) ,Attributegenerator) )

#add a timelineattribute to attribute

**Dependency=Dependencygenerator(**solution,Generatedependencymatrix,Generatedependencyattributes**)**

**return solution, attributes,Dependency**

**end:**

# Example of test data

timeline length=100

number of time lines= 2

number of tasks=13

**given solution:**

timeline solution 1

| id | start time | task length |
|---|---|---|

| | | |
|---|---|---|
| 1 | 0 | 10 |
| 2 | 10 | 20 |
| 3 | 35 | 2 |
| 4 | 40 | 10 |
| 5 | 55 | 30 |
| 6 | 90 | 5 |
| 7 | 97 | 1 |

timeline solution 2

| id | start time | task length |
|---|---|---|
| 1 | 0 | 10 |
| 2 | 12 | 10 |
| 3 | 25 | 5 |
| 4 | 32 | 5 |
| 5 | 40 | 20 |
| 6 | 70 | 20 |

**Attributes:**

Timeline attribute 1

| Earliest allowed start time | Latest allowed finish time | Length |
|---|---|---|
| 0 | 30 | 10 |
| 0 | 30 | 20 |

| | | |
|---|---|---|
| 30 | 40 | 2 |
| 30 | 100 | 10 |
| 50 | 100 | 30 |
| 50 | 100 | 5 |
| 60 | 100 | 1 |

Timeline attribute 2

| Earliest allowed start time | Latest allowed start time | Length |
|---|---|---|
| 0 | 25 | 10 |
| 0 | 25 | 10 |
| 20 | 40 | 5 |
| 30 | 100 | 5 |
| 50 | 100 | 20 |
| 50 | 95 | 20 |

**Dependency :**

Dependencymatrix

| Id of first task in dependency | timeline line of the first task in dependency | Id of second task in dependency | timeline line of the second task in dependency |
|---|---|---|---|
| 1 | 1 | 3 | 1 |
| 3 | 2 | 3 | 1 |
| 5 | 1 | 6 | 1 |

Dependencyattributes

| Time from first task is finished until second task is allowed to start | Time from first task is finished till second task has to start |
|---|---|
| 20 | 30 |
| 0 | 10 |
| 0 | 3 |

end of example of test data

**Create Test data GUI**

**The create test data GUI exist for creating and storing test data. When creating test data the user should be able to combine different attributegenerator, dependencygenerator and so on, that has been pre constructed.Structure of Ampl data set**

## Structure of Tabu Search data set

# References