F3

Faculty of Electrical Engineering
Department of Cybernetics

Bachelor's Thesis

An Environmnet for Evaluation of Robotic Experiments

Tomáš Verner

May 2023

Supervisor: RNDr. Miroslav Kulich, Ph.D.

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Verner Tomáš** Personal ID number: **492388**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Cybernetics and Robotics**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

An Environmnet for Evaluation of Robotic Experiments

Bachelor’s thesis title in Czech:

Prostředí pro vyhodnocování robotických experiment

Guidelines:

In the Intelligent and Mobile Robotics (IMR) group, we develop a range of algorithms whose properties with different parameter settings need to be experimentally verified and compared with existing methods. The work will aim to create an application in the programming language Julia that will process experimental results and generate LaTeX tables in the user-specified format. Specifically:
1. Design and implement an application that processes a experimental results as a batch of text files and stores them in a database.
2. Design, based on provided examples, a language for specification of an output table format.
3. Design and implement a tool that produces a user-specified table from experimental results stored in a database.
4. Choose 1-2 robotic problems with existing implementation of several approaches. Run experiments to compare these approaches, and generate tables using the realized tools to demonstrate usability and flexibility of the tools.

Bibliography / sources:

[1] S. Nagar. Beginning Julia Programming for Engineers and Scientists, Apress Berkeley, CA, 2017
[2] K. Okumura, Y. Tamura and X. Défago. Iterative Refinement for Real-Time Multi-Robot Path Planning, IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Prague, Czech Republic, 2021, pp. 9690-9697
[3] J. Rosol, Fast Computation of Visibility Polygons, diploma thesis, Dept. of Cybernetics, FEE, CTU in Prague, 2023

Name and workplace of bachelor’s thesis supervisor:

RNDr. Miroslav Kulich, Ph.D. Intelligent and Mobile Robotics CIIRC

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **01.02.2023** Deadline for bachelor thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

RNDr. Miroslav Kulich, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

. | Date of assignment receipt

Student’s signature

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Verner** Jméno: **Tomáš** Osobní číslo: **492388**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra kybernetiky**

Studijní program: **Kybernetika a robotika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Prostředí pro vyhodnocování robotických experiment

Název bakalářské práce anglicky:

An Environmnet for Evaluation of Robotic Experiments

Pokyny pro vypracování:

Seznam doporučené literatury:

[1] S. Nagar. Beginning Julia Programming for Engineers and Scientists, Apress Berkeley, CA, 2017
[2] K. Okumura, Y. Tamura and X. Défago. Iterative Refinement for Real-Time Multi-Robot Path Planning, IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Prague, Czech Republic, 2021, pp. 9690-9697
[3] J. Rosol, Fast Computation of Visibility Polygons, diploma thesis, Dept. of Cybernetics, FEE, CTU in Prague, 2023

Jméno a pracoviště vedoucí(ho) bakalářské práce:

RNDr. Miroslav Kulich, Ph.D. inteligentní a mobilní robotika CIIRC

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **01.02.2023** Termín odevzdání bakalářské práce: **26.05.2023**

Platnost zadání bakalářské práce: **22.09.2024**

RNDr. Miroslav Kulich, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. Datum převzetí zadání

Podpis studenta

Acknowledgement / Declaration

I would like thank the members of the Intelligent and Mobile Romotics (IMR) Group of the Czech Institute of Informatics, Robotics and Cybernetics (CIIRC) for providing me with great feedback for my implementation. I would like to especially thank Miroslav Kulich Ph.D., my supervisor, for keeping up with my often lazy approach and maintaining me in a productive state at all times.

I hereby declare that this thesis is the result of my work and my work only. All sources have been properly cited in accordance with the methodical instructions for observing the ethical principles in the preparation of university thesis.

In Prague, on May 10th 2023

.....

Abstrakt / Abstract

Tato práce implementuje prostředí pro vyhodnocování výsledků robotických experimentů, pro které se dá takové prostředí označit za dosud neprobádané téma, jelikož momentálně žádný takový program není k mání. Aplikace se soustředí na zpracovávání sady souborů s experimentálními výsledky do databáze, ze které poté data exportuje do LaTeXové tabulky. Pro nastavení uživatelských požadavků bylo navrženo rozhraní s cílem maximalizovat komfort uživatele.

Klíčová slova: SQLite databáze, LaTeX tabulky, vyhodnocování robotických experimentů, analýza dat.

Překlad titulu: Prostředí pro vyhodnocování robotických experiment

This work implements an environment for experimental evaluation. In particular, it focuses on robotic experiments, for which it can currently be classified as an unexplored area, as no programs of similar functionality were found online. The application focuses on processing a batch of experimental result files into a more organized and usable database, after which it extracts the data into a table. The table is written in LaTeX format to suit the parameters of robotic and other technical works. As the table design and the evaluation process should be to the desires of the user, an interface for the input of table evaluation specifications was designed with the attempt to minimize time consumption and maximize user comfort. The final version of the environment implementation should satisfy all of the user's prerequisites while being as easy-to-use as possible.

Keywords: SQLite database, LaTeX tables, robotic experiment evaluation, data analysis.

/ Contents

1 Introduction **1**
1.1 Premise/Scenario 1
1.2 Design decisions 2

2 Saving results to a database **3**
2.1 Parsing files 3
2.2 Creating a database 3
2.3 Optimization 5

3 User interface **6**
3.1 Database input 6
3.2 Table input 8
3.2.1 General 8
3.2.2 Top 11
3.2.3 Ratio 13
3.2.4 Left 13
3.2.5 Order (Left) 14
3.2.6 Bottom 16
3.3 Mixing YAML with LaTeX . . 17
3.4 How To Run 17

4 Testing process **19**
4.1 General examples 20
4.2 Performance v. Comfort 22
4.3 OpTeX conversion 24

5 Libraries **26**
5.1 MAPF-IR 26
5.2 MAPF-LNS2 27

6 Conclusion **28**

A Evaluation Environment
Source Code **29**

References **30**

Tables /

1.1 The general output table 2
2.1 PRAGMA statements used for optimization 5
3.1 Database input parameter options 7
3.2 Table input sections 9
3.3 Table input: General parameter options 9
3.4 Table parts 10
3.5 Table input: Top parameter options 12
3.6 Table input: Ratio parameter options 13
3.7 Table input: Left parameter options 14
3.8 Table input: Bottom parameter options 16
4.1 Default table example 20
4.2 Selected problem and method items 21
4.3 Table with arrows 21
4.4 Transposed table 22
4.5 Table with a tail 23
4.6 Simple table example 23
4.7 Complex table example 24
4.8 LaTeX table 25
4.9 OpTeX table 25
5.1 Table example 27

Chapter 1
Introduction

After successfully conducting an experiment, the researcher's work is not done as the produced experimental results still need to be evaluated. It is no secret that this particular part of experimenting, in any field, can be a prolonged and boringly repetitive experience. All the fun from the testing phase suddenly disappears as the researcher has to dig deep into data analysis.

Specifically in robotics, the experimental results usually contain mountains of data and the researchers need to implement programs that go through them all and store them in a neat place, this is more often than not a database. Then after processing the data their work continues as the stored results need to be transferred into a table or a graph to actually display the experimental evaluation. The table is usually preferred, and this can create more time problems for the researchers as they spend great amounts of time writing the results into the tables and then editing the table to their liking.

The thesis describes the implementation for an environment that can both process results from robotic experiments into a database, and subsequently generate a table, where the table appearance is specified through a user interface. The created tables are in LaTeX format, as LaTeX is the most commonly used environment for writing technical papers.

The current state of this problem is that every person has to write their own program for the data evaluation. These programs are not universal most of the time, and it can be stated thet there are currently no known alternatives for an environment that could be consideered a solution to this assignment. The main goal of the thesis is therefore not to outperform some already existing version, but to focus purely on creating a functional application that will provide a solution to the given assignment.

The environment was written in the Julia language, which was chosen for its speed and clear syntax. The syntax takes inspiration from the Python, MATLAB, R, and Ruby languages, while the speed often nears that of statically-compiled languages like C [1]. The Julia REPL (read-eval-print loop) can also be considered one of Julias main strengths and is perfect for continuous function testing, as it provides fast and easy statement evaluation.

The thesis is written in the *CTUstyle3* template [2] often used by the students of CTU in Prague for the purpose of writing papers about technical topics. This raises a few problems, as the assignment is meant to generate tables that are in LaTeX format, while *CTUstyle3* is OpTeX-based [3]. The tables presented in this paper are all in the LaTeX format to satisfy the proposed thesis assignment and are imported as figures.

1.1 Premise/Scenario

In robotic experiments, there is usually a problem or a set of problems. These problems are solved using several methods and the goal of the experiment is to compare the quality of each method. The problems can be given as one parameter, but it is possible to have an extra specification that divides each problem into subproblems.

After an experiment is conducted, the researcher has generated a data set of experimental results. The data is stored in data files, which contain different parameters, that each has its own assigned value. The individual pairs will be referred to as the *parameter:value* pairs. The file format should look like this:

```
problem_title: problem,
divider_title: divider,
method_title: method,
parameter1: value1,
parameter2: value2,
parameter3: value3
```

Looking at the proposed data file format, it is expected that at least two of the parameters define the problem and method titles, and if a problem specifier is present, a parameter setting the divider title is also mandatory. The three parameters, or two if a divider is not included, are the basis for creating tables, as they specify what the row and column variables will be. It is for this reason why these are the *main* experimental parameters and they will be referred to as such from now on.

The scenario is that after using the environment discussed in this thesis on a batch of such data files, the output will be a table written in LaTeX format, that should look something like table 1.1.

Problem	**Divider**	**method1**		**method2**		**method3**	
		avg	**min**	**avg**	**min**	**avg**	**min**
problem1	divider1						
	divider2						
problem2	divider1						
	divider2			Data			
problem3	divider1						
	divider2						

Table 1.1. The general table format for the application output relating to one parameter.

1.2 Design decisions

After about 4 months of work, the assignment progress was presented to members of the Intelligent and Mobile Robotics (*IMR*) Group of the Czech Institute of Informatics, Robotics and Cybernetics (*CIIRC*). It was with their feedback that it became apparent how divisive the user requirements could be. Many argued that if there is too much information needed to be provided by the user, then the program fails to achieve its goal of saving the user time. The argument was why use a program where the time it takes to write the YAML files is longer than the time it would take to write the table in LaTeX? Then there were some who liked the presented variety of parameters and were looking forward to creating very specific tables without the need to edit them in LaTeX afterwards.

The end result is made with the desire to please both sides. This is done mainly via default settings for each parameter, which means the user doesn't *need* to specify any parameter, although there are some parameters that are recommended to be set explicitly. In addition, two functions that each generate an input file template for the user to fill out have been implemented. Chapter 3 discusses both the usage of default values and the template generating functions.

Chapter 2
Saving results to a database

After finishing the testing phase of an experiment, one is left to deal with a large chunk of data. The data is usually stored in files which are formatted in a way where defined parameters are assigned their respective values. These will be referred to as *data pairs* or *parameter:value pairs*. The path from raw data files to a sophisticated database where all the useful experimental results are stored has many steps.

2.1 Parsing files

For the purposes of this assignment, a YAML parser was chosen as the default file reader. It was chosen because *YAML* is a superset of *JSON*, so both *YAML* and *JSON* file formats are acceptable and ready for parsing without additional modifications.

There is no such thing as a universal parser for data files. The truth is there is no one correct format and each person is going to prefer a specific file structure for their data representation. It is not uncommon for the user to have the data files in a format that is not YAML-friendly. In that case they will unfortunately be left to implement their own file-reading function to suit their data file formatting. The variance in user data formatting is evident just from the few examples that have been used for testing the code, see chapters 4 and 5.

The end result of the parsing process for each file is a dictionary containing the parameters (keys) and their found values.

2.2 Creating a database

After the parsing process of a file is complete, we are left with a dictionary of parameters and their respective values. It is now time to store the found data into a database. This will be done with the *INSERT INTO* SQL query for every data file. The database path is set as *"./database.db"* by default.

For the database part implementation, the decision was made to use SQLite. The main reason behind this decision was the popularity of this SQL database engine. According to the official SQLite webpage [4], it is the most used database engine in the world. It is also very fast, without the need for many dependencies (stand-alone). In addition, as explained in chapter 2.3, SQLite provides a great utility for database optimization, the PRAGMA statement, which is an SQL extension specific to SQLite.

During the creation of the database, many tables with different purposes are generated. This ensures that the program doesn't use excessive data where it is not necessary. It also organizes the data in a very neat way and simplifies the usage of specific groups of values later on.

All the parameter values extracted from the data files are stored in the main one, which is titled *results* by default. Both the database path and the name of the main table in the database can be changed from their default values through a database

input file, which is the concern of chapter 3.1. The main table consists of a primary key column (identification integers starting at 1) and columns named after every different parameter found in the data files.

The problems can sometimes come with the inclusion of a numeric specification at the end of their name. For example *"berlin52"* or even something like *"berlin52-13.2"* can be classified as having numerical information at their ends.

Additional columns are added to the main table if the problem names are formatted as *problem_nameN*, where *N* is the numeric specification. This can define anything from of the amount of vertices a map has (pathfinding environments) to the version number of a problem.

The environment discussed by this thesis was tested mainly on agent path-finding search algorithms. This is the reasoning behind the design decision to have the database column names relating to the problem numerical specifications be *Nodes* and *NodeType*. For example, *"berlin52"* is a problem definition that will have a value of *52* stored in the *Nodes* column, while *"berlin52-13.2"* will also have a value of *13.2* added under the *NodeType* column. These database columns are useful for the ordering of problem columns in the final LaTeX table, more on this topic in chapter 3.2.5.

An interesting scenario is when some result files contain a parameter that others do not. In this case the value stored for such parameters for the files is either the value found in the data file, or the *NULL* value. *NULL* cells in the database do not interfere with any of the calculations later conducted by SQL queries, for example the *SELECT GROUP BY* command used for aggregate functions like the average or minimum values.

This is another set of very useful data and it is also saved into a separate table. Because this table consists of values calculated with aggregate functions, the table is titled *Aggs*. It has the problem, divider, and method columns, and columns which contain the data for each aggregate function provided by the *GROUP BY* SQL query. These columns are *count*, *avg*, *min*, and *max*. They contain the total amount of instances, the average, minimum, and maximum values respectively.

When using the *GROUP BY* option for the *SELECT* SQL query for the creation of the *Aggs* table, the grouping is done for the problem, divider, and method columns. That means that the aggregate parameters are computed for each permutation of the problem, divider, and method names.

A very important parameter which is used in many tables and is otherwise often utilized for additional parameter calculations is the Best-Known-Solution (*BKS*) parameter. It was considered as too important not to warrant its own table. The table is titled *BKS* and has the primary key, problem_name, divider_name, and best columns, with numeric specifier columns being added for ordering purposes if they are present in the problem names.

It is not expected nor required for the BKS values to be provided by the user, but if a path to the BKS files is given as one of the inputs (chapter 3.1), the BKS table will be created through the parsing of these files, the same way the main table is created from the data files.

In most cases, the Best-Known-Solution values will have to be computed separately. This is where the *Aggs* table comes in handy. The program looks through all methods to find the minimum value, which is then stored in the *best* column in th *BKS* table.

It is not always that every instance of an experiment finishes successfully and provides valid results. If it is given as one of the input parameters (chapter 3.1), the data file validity rate will be examined during the data parsing process based on parameters set

by the user. This brings us to the last table that can be generated in the database, the *Valid* table.

As opposed to the main table, where only valid instances are stored in order to preserve only the relevant data and protect the parameter calculations from being corrupted by invalid values, the *Valid* table stores data from every single file, regardless of validity. Apart from the primary key column and the file parameter columns, this table also includes the *valid* column. The column consists of ones and zeros, where ones are assigned to the valid instances and zeros to the invalid ones.

The table for the validity information creates a new parameter option for the method comparison, the *valid* parameter. This and all the other parameter options will be described in detail in chapter 3.2.

2.3 Optimization

We live in a time and age where the speed at which an application operates is at the top of the user's priority list. A program, especially one made with the intention of saving the user's time, becomes redundant if it runs for too long.

For the proposed environment, the database creation during the data file parsing process was the most time-consuming part of the solution. For example, one instance of the environment dealt with a directory containing 1440 data files, averaging approximately 12 parameters each and without optimization, the database would take anywhere from 2 to 5 minutes to store the experimental data from these files.

To minimize the runtime, SQLite provides many tools. As mentioned at the beginning of chapter 2, one of the best ways to simplify database operations is with the SQLite-specific *PRAGMA* statements. Table 2.1 showcases the statements included in the solution, which modify the database operation settings. The particular values used were recommended by *Karel Košnar Ph.D.*, a member of the *IMR Group*.

Statement	Set Value	Default Value
synchronous	OFF	FULL
page_size	4096	4096
cache_size	16384	-2000
temp_store	MEMORY	DEFAULT
journal_mode	OFF	DELETE
locking_mode	EXCLUSIVE	NORMAL

Table 2.1. PRAGMA statements and their settings used for optimizing the evaluation environment.

To fully understand the purpose of each PRAGMA statement, see the official SQLite documentation [5]. It is important to mention that the *journal_mode = OFF* setting disables the *COMMIT* and *ROLLBACK* commands for the transaction SQLite feature. Transactions are otherwise also a great optimization tool for SQL queries.

After the implementation of the utilities listed in table 2.1, the program was tested numerous times and its average runtime can be estimated at around 3 seconds for a set of approximately 17 000 *parameter:value* pairs. It goes without saying that for larger data sets, say 100 000 data pairs and above, the evaluation will of course take longer.

Chapter 3
User interface

Probably the most important and definitely the most challenging part of the environment implementation was designing the interface through which the user gives the program instructions on how to both save the experimental results into a database and how to then make a LaTeX table out of that database.

The two input formats considered for the design of the interface were *JSON* and *YAML*. Though *JSON* is more popular among programmers, this application is meant to be used by all people. *YAML* was chosen as the default format mainly for its human-friendly syntax. However, it is also great because if the user is a *JSON* conservative who refuses to use any other input format, their input files will still be parsed correctly. This is because *YAML* is a superset of *JSON*, as was already mentioned in chapter 2.1.

The files are parsed into a dictionary, therefore the order in which the parameters are listed doesn't have an effect on the functionality, unlike the data files, where the paremeter order can have great impact on how the environment runs.

Due to our solution being separated into two major sections (data files -> database, database -> LaTeX table), I have opted to split the input into two separate YAML files.

3.1 Database input

The first part concerns the experimental result files and their parsing and storing to an SQLite database. For this section the user may want to change the outlook of their database, define which parameters are relevant for the evaluation, if any parameters are missing, how the values should be saved, and set whether or not all the results are valid.

The database YAML input contains specifications providing the user with the means to handle all of these requests. The possible input specifiers with their respective default values are listed in table 3.1.

There are 11 key options, with the limitation that only one of the *Include* and *Exclude* parameters can be used at once. So each database input file can contain anywhere from 0 up to 10 different specifications.

For some of the input keys, if a value is not set for them, the first data file of the parsing process is used for their definition. The particular parameters are *Main* and *Examine*.

The path to the data files containing experimental results used for analysis, more precisely for this thesis table generation, can and needless to say in most cases should be specified through the use of the *Path* parameter. If not specified, the assumed path will be *"./results/"*.

One of the most useful parameters when it comes to comparing different solvers is the best-known solution (BKS). While it is not a necessity, the user can have BKS values already calculated and stored (in files separate from the data files). It is in this case where the user can utilize the *BKS* input to specify the path to such files. The

Name	Type	Default Value	Examples
Path	String	`./results/`	-
BKS	String	`none`	`./bks/`
Database	String	`./database.db/`	-
Title	String	`results`	-
Main	String, Vector{String}	`none`	[`Problem`, `NumRobots`, `Method`]
Include	String, Vector{String}	`all`	`WCost`, [`WCost`, `Time`]
Exclude	String, Vector{String}	[]	`Path`, [`Cost`, `Path`]
Examine	String	`none`	`WCost`
Add	String, Vector{Any}	`none`	`delay`, [`collision`, 0.9], [`delay`, [`collision`, 0.9]]
Precision	Int, String	2	`all`
Valid	String, VectorAny	`none`	[`makespan`, -1]

Table 3.1. Parameter options for the database input with their default value.

best-known solution values are otherwise computed during the parsing process from the given data files.

The *Database* input specifies the file path to the SQL database, where all important data is saved. The default file path is *"./database.db"*. If the database does not exist a new one will be created at the given file path.

Out of the 10 specifiers, *Title* is the least complicated. It defines a title for the main table in the user's SQL database. In other words it serves a purely cosmetic purpose. The default table name is set as "results".

As mentioned in chapter 1.1, in robotic experiments, there are 2 or 3 main variables. The *"problem"* parameter which for example defines the environment that is being searched, which is often then further specified by the *"divider"* parameter, e.g., the amount of agents conducting the exploration. Then there is the *"method"* parameter, specifying the method or solver used to search said environment. The method is the subject of comparison in most cases and will almost exclusively be placed at the top of a table. These variables are defined with the *Main* input. If a *"divider"* is not present *Main* should be given as a *[problem, method]* 2-item vector, where the items are the corresponding names of the previously described parameters as they are written in the data files. A 3-item vector *[problem, divider, method]* is to be given otherwise.

The default value of *Main* is *"none"* and if the user doesn't specify the names of the main variables, then their values will be defined using the first parsed file. If the third parameter is not a string, the *"problem"* and *"method"* names, in that order, will be set to the first 2 parameters found, otherwise the first 3 parameters will assume the names for the *"problem"*, *"divider"*, and *"method"* variables respectively. This is where the order of parameter listings in the data files matters.

Usually there is a lot of parameters in each data file, and it is often too many. Not all the data found in the result files has to be relevant for the evaluation. If it is not necessary for the user to store every parameter from every file in the database, it is

recommended to utilize one of the *Include* or *Exclude* inputs. Quite self-explanatory, the former specifies parameters that should be stored, discarding any that are not specified, while the latter discards the specified parameters and keeps those that are not included. For a large number of parameters, the *Include* option should be chosen in the case of the need to discard more than half of the parameters and the *Exclude* option otherwise.

When the data files are parsed into a database, a table titled "Aggs" is created. The table contains the SQL-friendly aggregate functions *count*, *avg*, *min*, and *max*, which have been calculated for the "examined parameter", see chapter 2.2. This parameter can be specified using the *Examine* key. Its default value is *"none"* and if left unspecified, the "examined parameter" is equal to one of two options. If there are specific parameters given with the *Include* input, it is equal to the first of them. It is otherwise the last numeric parameter found in the first file during the parsing process.

For some experiments there are parameters which are not included in the data files, but the user would still like to have them stored in their database for further use. This can be done via the *Add* input being given as a list of parameters that should be added to the database. A string is also acceptable if the user only wants to add 1 parameter. It needs to be said that these parameters will be too specific in most cases and a way to calculate them will have to be implemented manually.

The *Precision* parameter specifies how numeric values should be rounded, if at all, when being stored into the database. It can either be given as *"all"* in which case no rounding will take place or it can be set as a non-negative integer. After examining a few tables from theses about robotic experiments, path-finding algorithms in particular ??, the default value has been set to 2.

It is not always that every instance of a robotic experiment runs as planned and finishes successfully. If that is the case, the user can provide the program with a validity check option by setting the *Valid* key as a *[parameter, invalid_value]* 2-item vector. The environment then searches every file for a *parameter:invalid_value* data pair and if such a pair is found, deems the file as an invalid instance. The default option is *"none"* and assumes all data files as valid instances.

3.2 Table input

As for the table generation part of the solution, it can be expected that the user will want to control how the table is designed, like setting what lines should be present, where the table cells should be bold, which columns to align where, etc. They can also want only some of the database data to be displayed in the table. There can also be a desire to order the cells by certain criteria, or to expand the table with a tail. Maybe the final table is too wide and the user would like to have it be transposed.

Based on requirements like these, the table input file specifies how the final product, the LaTeX table, should look according to the user. For this file there are 5 main parameters which each has its own subkeys. Due to this it is more complex than the input file for our database and requires a deeper description.

3.2.1 General

The *General* key contains general information regarding our table. This mainly defines the overall appearance of the table (inner and outer lines, bold sections, and table transposition), but it can also be used for parameter and unit definitions, or set if BKS values are to be incorporated.

Table 3.3 gives an overview of the 8 subkey options for the *General* parameter that help deal with such specifics.

Section	Description
General	Contains general information about the table design.
Top	Information about the header.
Left	Information about the first columns.
Bottom	Information about the tail.
Ratio	Information about comparison columns at the end of the table.

Table 3.2. The available keys for the table YAML input.

Name	Type	Default Value	Examples
BKS	Bool, String	`True`	`Best`
Transpose	Bool	`False`	`-`
Bold	String	`top`	`none`, `xbottom`
Verticals	String	`none`	`left`, `Method`
Horizontals	String	`none`	`header`, `Problem`
Border	String	`none`	`all`, `ver`
Params	Vector{String}	[`Min`, [`PDB`, `\%`], `Avg`, [`PDM`, `\%`], `SD`]	`-`
Units	String, Vector{String}	[side, square]	`under`

Table 3.3. Parameter options for the General key in the table input with their default values.

Whether the table should contain a column with BKS values is specified with the *BKS* key. Apart from all the other boolean options in both YAML files, *BKS* has the default value *True*. It can also be given as a *"new_bks_title"* string which then changes the column title from *"BKS"* to the input.

The final table can sometimes be too wide to fit on a page. In this scenario the user can set the *Transpose* parameter to *True* to transpose the table. This utility is off by default.

To define which part of the table should be in bold, the *Bold* input is used. As can be seen in Figure 3.4, there are 6 major parts to a table. For this program, all parts except for Data can be in bold.

The possible values for *Bold* are *"top"*, *"left"*, and *"bottom"*. *"top"* includes the top-left and top parts, *"left"* includes the top-left, left, and bottom-left parts, and *"bottom"* includes the bottom-left and bottom parts of the table. In addition, the input can be adjusted in 2 ways. If no table part is supposed to be in bold, the parameter is set to *"none"*.

The option can begin with 'x' to reverse the effected part. For instance, the default value *"top"* bolds the top-left and top sections, whereas *"xtop"* bolds the top-left, left, bottom-left, and bottom table parts.

Next up is the *Verticals* input. It defines vertical lines inside the table and is set to *"none"* by default. Other values are as follows,

Top-left	Top
Left	Data
Bottom-left	Bottom

Table 3.4. The chosen names for each table part.

- *left* - one vertical line precedes the Data table part (see figure 3.4);
- *method_title* - vertical lines surround each method column;
- *all* or *params* - vertical lines surround every column.

Working in a very similar way is the *Horizontals* parameter, though it applies to horizontal lines instead. Its default setting is *"none"* as well and the rest of the options are

- *header* - one horizontal line follows the top part of the table;
- *tail* - one horizontal line precedes the bottom part of the table;
- *both* or *data* - a combination of the *header* and *tail* options;
- *problem_title* - horizontal lines surround each problem row;
- *all* or *divider_title* - horizontal lines surround every row.

The *Border* key deals with lines outside the table. As usual, it is set to *"none"* by default and can otherwise have these values,

- *top*, *left*, *bottom*, or *right* - one line on the top, left, bottom, or right of the table respectively;
- *hor* - horizontal lines at the top and bottom of the table;
- *ver* - vertical lines on the left and right of the table;
- *all* - the border is on all sides.

For the first 4 options, the ones with a 1-line border, there is some resemblance to the *Bolden* parameter. They can all be preceded by 'x' in order to flip their effects and generate a 3-line border opposite to the 1-line one.

To specify variables for the examined parameter (see *Examine* in chapter 3.1) and their units, if there are any, the user can list them in the *Params* input. Parameters that have been stored in the database but are not being exmined can also be included, but only their average values will be displayed. If there is only 1 parameter the user wants to use, a *"param_name"* string can be used, though it is recommended that there be more then 1 parameter. The expected input is a vector of parameters, which can each be defined as the previously mentioned *"param_name"* string, or a *[param_name, param_unit]* 2-item vector. The default list is *[Min, [PDB, %], Avg, [PDM, %], SD]*.

Acceptable parameters are:

- *count*, *avg*, *max*, *min* - Aggregate functions provided by the *GROUP* SQL query, for which the data is stored in the *Aggs* table in the database, see chapter 2.2;
- *PDB* - The percent deviation of the best solution value found by the method to BKS, calculated as `100*(best - BKS)/BKS`;

- *PDM* - The percent deviation of the mean solution value found by the method to BKS, calculated as `100*(avg - BKS)/BKS`;
- *SD*, *stdev* - Standard deviation;
- *Valid* - The percentage of valid instances relating to the total;
- *param_name* - If the name of a one of the data file parameters is given, the output will be the calculated average for that parameter;
- *custom* - Any additionally implemented parameters can also be given.

The custom parameters should be implemented in the *getAdditionalParams()* function in the *customParameters.jl* file with input specifications and the means to calculate them.

The position and appearance of the parameter units can be specified via the *Units* key. Initially set to display units inside square brackets next to their respective parameters, it can be given as a *"unit_position"* string or a *unit_position]* 1-item vector to define a new unit position. There are only two options other than *"side"*,

- *under* - units and parameters each have their own row, the units are below the parameters;
- *none* - disables the displaying of units.

It can otherwise be set as a *[unit_position, bracket_type]* 2-item list, where the bracket types are

- *square* - [unit], the default setting;
- *round* - (unit);
- *curly* - {unit};
- *angle* - ⟨*unit*⟩;
- *none* - unit is displayed without brackets.

3.2.2 Top

Information about the header of our table is stored in the *Top* input. This includes method specifications, parameter choices, column alignments, whether or not the method title should be displayed in the top-left corner, etc. As was the case for the *Table* key, there are 8 subkey options, though they have nothing in common with the 8 *Table* parameters. The 8 new options are listed in table 3.5 below.

Title provides an option to rename the method row from its default *"Method"* title. The renaming process precedes the calling of the *method_title* parameter, which defines what particular methods should be included in the header. If left unspecified (set to *"all"*), the table will contain every method that has been saved to the database during the parsing process.

The *Params* input expects a vector of the *[first_method, ..., last_method]* format, where each item defines parameters for the respective method with either a *"param_name"* string or a *[param_name1, param_name2, ...]* vector. Given parameter names can be substituted by index values corresponding to those in the *Params* list from the *Table* key. The parameter name or index must always exist in or be in range of the vector from the *Table* part. The input vector must not consist of more items than the number of defined methods. If it contains less, the final item will be used for all remaining methods. For example, the default value of *[[1, 2]]* assigns the first and second parameter from the *Table* list to each method.

When the user wants to change the appearance of specific methods, the *Rename* and *Rotate* utilities are used. The *Rename* input renames one method specified as

Name	Type	Default Value	Examples
Title	String	`method_title`	`Solver`
method_title	String, Vector{String}	`all`	[`greedy`, `kmeans`, `multi`]
Params	Vector{Any}	[[1, 2]]	[`avg`, [`min`, `PDB`], [1, 2, 3]]
Rename	Vector{Any}	`none`	[`greedy`, `Gr`], [[1, `Greedy`], [`kmeans`, `KMeans`]]
Rotate	Bool, String, Int, Vector{Any}	`False`, `none`	`greedy`, [[`greedy`, `kmeans`], 90]
Lines/Line	Bool	`False`	-
Display	Bool	`False`	-
Align	String, Vector{String}	`right`	`center`, `left`

Table 3.5. Parameter options for the Top key in the table input with their default values.

[old_name, new_name] or multiple methods *[[old_name1, new_name1], [old_name2, new_name2], ...]*. Indexes relating to the defined methods, in the order as they are in the table header, can be given in the place of old names.

Rotate can rotate method names in the top part of our table. The possible input options are

- *False* or *"none"* - nothing in the header is rotated, the default setting;
- *True* or *"all"* - rotates all method names by 90 degrees;
- *"method"* - the method name is rotated by 90 degrees;
- *degrees* - if the input type is an integer then all method names are rotated by that many degrees;
- *[method1, method2, ...]* - the method names listed get rotated by 90 degrees;
- *[method_name, degrees]* - the given method name is rotated by the given amount of degrees;
- *[[method1, method2, ...], degrees]* - the method names listed are rotated by a set amount of degrees;
- *[[method1, degrees1], method2, ...]* - a list of methods where each is either a string and is rotated by 90 degrees (method2) or a 2-item vector where method1 is rotated by degrees1.

The *Line(s)* key is set to *False* by default. When set to *True* there are horizontal lines inside the header. *Display* is also set to *False* by default and when turned on (*True*) the top-left part of the table displays the *"method_title"* next to the method row.

Finally the *Align* parameter sets how each column pertaining to a method name is aligned. It can be given as *"left"*, *"middle"*, or *"center"*, but the default value is *"right"*, as this alignment effects the data part of our table, where it is best the numbers be aligned to the right.

3.2.3 Ratio

The *Ratio* input describes whether columns with method comparisons should be included after the method columns. It can be set without any subkeys as

- *False* - no ratio columns will be included, the default setting;
- *True* - there will be *method_number - 1* ratio columns containing the ratios *method*:*reference*, where the first method is the reference and each column is titled R_{method}, where *"method"* is every method except the first one;
- *"ratio_title"* - creates a ratio column comparing the second method to the first;
- *[ratio1_title, ratio2_title, ...]* - n-item vector, where the ratio columns are comparisons of the n-th method to the first.

However, there are multiple subkey options as well. These are listed in table 3.6.

Name	Type	Default Value	Examples
Name	String, Vector{String}	`none`	`$R_{greedy}$`, [R_1, R_2]
Ref	String	`first_method`	`kmeans`
Set	String, Vector{String}	`all`	`greedy`, [`greedy`, `multi`]

Table 3.6. Parameter options for the Ratio key in the table input with their default values.

Name works as the general settings listed above, where *False* is not an option and *"none"* is expected in place of *True*, but the rest is the same. *Ref* sets the reference method, the default being the first one and *Set* specifies which methods should be compared to the reference method.

Set is *"all"* by default and works as the *Ratio*: *True* setting. It can also be given as *"none"*, which is equivalent to the *Ratio*: *False* setting or it can list the methods that should be compared to the reference. A string is acceptable if there is only 1 expected ratio column.

3.2.4 Left

To describe how the first columns are supposed to look, the instructions are given with the *Left* key. The columns included are the ones before the data section (see table 3.4). The 10 subkey options and their respective default settings are shown in table 3.7 below.

There are a few subkeys that are shared with the previously described *Top* key. The *Rename* and *Rotate* parameters are identical in usage to the ones in the top part of the table, with the effected table cells being the problem and divider names in place of the method names in *Top*.

The *Title* input renames the problem column with a *"problem_title"* string or a *[problem_title]* 1-item vector. A *[problem_title, divider_title]* list is used if the divider column title is also desired to be changed. In addition, *"problem_name"* and *"divider_name"* replace the *"method_name"* input, they work the same for the problem and divider columns as they did for the method row in *Top*, and they keep *"all"* as their default value.

Similar to other parameters carried over from the *Top* section, the *Line(s)* key functions the same (*False* by default), but refers to vertical lines inside the left part of the table instead of horizontal lines in the top part.

Name	Type	Default Value	Examples
Title	String, Vector{String}	[problem_title, divider_title]	Map, [Instance, Agents]
problem_title	String, Vector{String}	all	[berlin52, lin318, rat575]
divider_name	Int, Float, String, Vector{Any}	all	[50, 70, 100]
Rename	Vector{Any}	none	[berlin52, berlin], [[1, Berlin52], [lin318, Lin318]]
Rotate	Bool, String, Int, Vector{Any}	False, none	berlin52, [[berlin52, lin318], 90]
n	Bool, String	False	Nodes
Lines/Line	Bool	False	-
Arrow	Bool	False	-
Align	String, Vector{String}	center, middle	right, left
Order	Bool	problem	xproblem, [[NumRobots, desc], [Map, problem]]

Table 3.7. Parameter options for the Left key in the table input with their default values.

A new input option is the *n* key, which is used to set whether or not a column containing the numeric specification at the end of problem names, as was described in detail in chapter 2.2, should be added to the table. It is not included by default (set to *False*), but can be with *True* or a *"n_title"* string, which, appart from adding it to the table as the second column (after the problem column), also renames the column (it is titled *"n"* by default).

Another *Left*-specific parameter is a cosmetic option which adds a top-to-bottom arrow icon to the left column names. It is off by default, but can be activated by setting the *Arrow* key to *True*. This utility can be used to clarify that a column title does not relate to the row to the right of it.

The *Align* parameter has a default value of *"middle"* or *"center"*, instead of *"right"* as it was in the top table part. It aligns the problem, divider, and "n" columns if they exist.

3.2.5 Order (Left)

Though it is still a part of the *Left* parameter, the *Order* input is a rather complicated and very important one, and it deserves its own chapter. It defines the ordering type for the problem and divider table columns.

In its simpler form, the input is a string. If possible, it applies to both columns and keeps the priority sequence: problem first, divider second. If it is not possible, the columns which can not be affected by the given order type keep their default value, see

equations (2) and (3) for further clarification. To set the ordering type, it can have the following values:

- *asc* - the basic alphabetical (a-z) or numerical (min-max) order for numbers;
- *desc* - the reverse alphabetical (z-a) or numerical (max-min) order for numbers;
- *problem* - is an option only for the problem column, where the problems go from least amount of vertices/nodes to most and if node numbers are not present, it behaves as the *asc* order;
- *xproblem* - which is the ordering opposite to *problem* and when node numbers are not present, it behaves as the *desc* order. It is again available only for the problem column.

To set which of the 2 columns should be sorted with priority, the string is equivalent to the name of the first-to-be-sorted column (see *NumRobots* in equation (4)). The default setting is *problem* for the problem column and *asc* for the divider column.

In the case the user wants to set different ordering rules for each column or change the priority from its default state with specific sorting definitions, the parameter can be given as a 2-item vector. If the user wants to change the default ordering for a single column and have it prioritized, the input is given as *[column name, order type]*, see *[NumRobots, desc]* in equation (5). The other way to set the *Order* parameter is with the vector *[primary column, secondary column]*, where each column is defined either as a *"column name"* string with default ordering for that column, see *[NumRobots, Map]* in equation (4), or as a *[column name, order type]* vector, see the first item in equations (1)-(5).

Here are a few examples of the *Order* parameter settings that have been described above. Each of the following equations shows different acceptable inputs which set identical sorting rules for our problem column called *"Map"* and divider column titled *"NumRobots"*,

$$[[Map, problem], [NumRobots, asc]] = [Map, problem] = problem = \texttt{no input}, \quad (1)$$

$$[[Map, desc], [NumRobots, desc]] = desc, \quad (2)$$

$$[[Map, xproblem], [NumRobots, asc]] = xproblem, \quad (3)$$

$$[[NumRobots, asc], [Map, problem]] = [NumRobots, Map] = NumRobots, \quad (4)$$

$$[[NumRobots, desc], [Map, problem]] = [NumRobots, desc]. \quad (5)$$

Equation (1) shows the default setting, where the first sorted column is the problem column titled *"Map"* and it is ordered from the least amount of nodes to the most. The divider column called *"NumRobots"* is second and is sorted alphabetically (a-z). In equation (2), the columns keep their default sorting order and are both ordered reverse alphabetically. For equation (3), *"Map"* is ordered from most amount of nodes to least and *"NumRobots"* is ordered numerically (min-max). The columns sorting priorities are switched and have the columns have their default ordering in equation (4). Equation (5) again has the flipped order of sorting by *"NumRobots"* first, then by *"Map"*, and has the "NumRobots" column in reverse numerical order.

Name	Type	Default Value	Examples
Name	String, Vector{Any}	`none`	`Title`, [`none`, 4]
Names	Vector{Any}	-	[`none`, 3, 4], [[`Title1`, `Title2`], 4]
Data	String, Vector{Any}	`none`, []	[`none`, 4], [[`A`, `B`, `C`], [2, 4, 4]]
Lines/Line	Bool	`False`	-
Align	String, Vector{String}	`center`, `middle`	`right`, `left`

Table 3.8. Parameter options for the Bottom key in the table input with their default values.

3.2.6 Bottom

On rare occasions, perhaps to group or further describe the table columns, the user may want to add a tail section to the bottom of the table.

If that is the case, the *Bottom* key is utilized. It is set to *False* by default as there is no tail expected in a basic table. If it is desired though, there are 5 subkeys that specify the tail details. The default values of these subkeys are listed in table 3.8.

There are 2 scenarios:

1) The table tail has one row of data, where the *Name* input is used to title this row and the bottom-left table corner is left blank if it is set to *"none"*. It can be given as a *"row_title"* string, which assumes the width of the title so that it fits the bottom-left section of our table. Otherwise it is a *[row_title, title_width]* type of vector.

   For the *Name* option, the bottom part of the table is specified via the *Data* key, which can be given as

   - *"none"* or *[ ]* - The default setting, which means that each cell after the tail title has a 1-column width and is assigned a capital letter, going alphabetically from left to right;
   - *[none, cell_width]* - Same as *"none"*, but the cells have a set width;
   - *[data1, data2, ...]* - The values for cells after the tail title are listed and the cells have a 1-column width;
   - *[[data1, data2, ...], cell_width]* - Same as the previous option, but the cells have a set width;
   - *[[data1, data2, ...], [width1, width2, ...]]* - Again, same as the previous option, but here each cell has its own specific width.

2) The table tail has multiple rows and the *Names* parameter is used. The input options are

   - *[none, tail_height]* - The title cells are blank and the tail consists of *tail_height* rows;
   - *[none, tail_height, title_width]* - Same as the first option, with a custom width set for the blank title cells;
   - *[title1, title2, ...]* - A list of titles is given and the amount of rows present is the length of this list. If *"none"* is given as an item the respective cell will be blank;
   - *[[title1, title2, ...], title_width]* - In addition to the previous setting, here the title cells have a set width.

The *Data* parameter is given as a vector where each item is a row definition which has the same input options as in the *Name* scenario.

Out of the two scenarios, *Name* is set as the default one, hence *Names* do not have a default value.

Then there are parameters which are not concerned with how many rows the table tail has. The *Line(s)* key is set to *True* if horizontal lines are supposed to be inside the table tail. The tail columns are aligned to the center by default and can be set to *"left"* or *"right"* with the *Align* parameter.

3.3 Mixing YAML with LaTeX

When it comes to YAML files, the user must be careful when using YAML sensitive characters in their inputs. Some of the main examples of such characters are: '[', ']', '{', '}', ':' with a space after, '?', '!', or '%'. For a few of these, like the exclamation mark and the percent character, an escape character will ensure that no problems will arise during the parsing process. For all the other ones the desired input must be put inside quotation marks.

The most frequent scenario is that the user will want to give a TEX math expression as an input. The '$' character is surprisingly not one of the YAML sensitive characters and it is therefore not always necessary to enquote the input.

Here are some YAML-friendly expressions which do not need any additional modifications to be parsed correctly,

$$\$Solver\_1: Solver\_2\$,\ xbottom+.$$

The underscore character is not YAML sensitive. Neither is the colon, because in this case, there is no space after it. The '+' character also does not cause any problems.

In contrast, the following expressions need to be adjusted in some way, before being in the correct YAML format,

$$\$Ratio\_\{greedy\}\$,\ !top,\ [PDB, \%].$$

The underscore is ok, but the curly brackets raise problems. The whole first expression has to be enquoted as a result. Quotation marks are not necessary for the remaining two, but a backslash character must be placed in front of the '!' and '%' characters. The *!top* example will be further discussed in the paragraph below.

The initial character that was used for reversing effects of inputs was '!', as it is used as the *NOT* command for boolean expressions in programming. Due to the inconvenience of it being YAML sensitive, it was eventually decided to change it to the 'x' character, so that there would be no need to add escape characters or quotation marks to the inputs. This change includes the *Border* input in the *Table* key and the *Order* parameter in the *Left* key, both of which are in the YAML input for the table.

3.4 How To Run

The environment is implemented in the *main.jl* file and the instructions for the usage of this application are as follows.

The program should be run from the command line with

$$julia\ main.jl\ [switches],$$

where the switch options are

- `-h`, `--help` – Prints a help message with switch explanations and instructions on how to run the program. It is a shorter version of this chapter;
- `-s`, `--save` – Enable the data file parsing process and create a new database before generating a table. If this parameter is not given, an already existing database is expected to be located in the workspace;
- `-d`, `--db-input [FILE_PATH]` – Set the database input file path. A *.yaml*, *.yml*, or *.json* extension is expected;
- `-t`, `--table-input [FILE_PATH]` – Set the table input file path. As for the previous switch, a *.yaml*, *.yml*, or *.json* extension is expected;
- `-o`, `--output [FILE_PATH]` – Set the table output file path. The default one is *./table1.tex*;
- `-p`, `--optex` – Convert the resulting table from LaTeX format to OpTeX format;
- `-D`, `--db-template [FILE_PATH]` – Create a template file for the database input;
- `-T`, `--table-template [FILE_PATH]` – Create a template file for the table input.

In most cases, the user will want to specify the database and table inputs, the output file, and set whether the data needs to be stored in a database or whether a database already exists. The usual form of the run command is therefore something along the lines of

julia main.jl `-d db_input.yaml -t table_input.yaml -o output.tex -s`.

The format of the help message displayed after using one of the `-h` or `--help` switches takes inspiration from the format used by Keisuke Okumura PhD. in his mapf-IR library [6], which was one of the robotic problems used to test our environment, see chapter 5.1.

Chapter 4
Testing process

The purpose of this chapter is to showcase a variety of input files and the corresponding tables, in order to give a graphic explanation of the YAML input files described in chapter 3.

During the implementation of the environment, the current state of the solution was constantly tested on a set of data files, which were provided by the assignment supervisor Miroslav Kulich PhD. The format for each of these data files is can be seen below.


```
{
    "Problem": "berlin52",
    "NumRobots": "2",
    "Method": "greedy",
    "Route_0": {
        "path": "0 21 30 ... 32 16",
        "cost": "161650"
    },
    "Route_1": {
        "path": "0 48 31 ... 6 1",
        "cost": "317831"
    },
    "Cost": "479481",
    "Total": "322",
    "WCost": "1489.071428571429",
    "Time": "0.318"
}
```


The files are the results of a multi-agent pathfinding experiment, where an environment, defined under the *Problem* parameter, is searched by a total number of 2, 4, or 8 agents (*NumRobots*). The particular problems are berlin52, bier127, gil262, lin318, pcb442, and rat575.

The experiment's goal is to compare methods for searching these environments with the solver options being greedy, grwmp, kmeans, and multi. The method is specified with the *Mathod* parameter.

The files are written in *JSON* syntax and are therefore parsable by a YAML parser. This means that no additional file-reading functions were needed for the testing of these files.

The first three parameters were set as the *main* parameters. We therefore have the following settings,

$$problem = Problem,$$
$$divider = NumRobots,$$
$$method = Method.$$

The *WCost* or weighted cost parameter was chosen to be the *examined parameter*, as it is the most relevant out of all the options. The route path and route cost parameters were discarded for the sake of keeping the database easy to navigate and to save time on the database generation process.

The following database input file is going to be the default one for all the tables in this chapter, unless a different data input file is specified,

```
Path:       ./results
BKS:        none
Database:   ./database.db
Include:    [WCost, Cost, Total, Time]
Examine:    WCost
Precision:  2
```

4.1 General examples

Problem	NumRobots	BKS	greedy		grwdmp		kmeans		multi	
			Min	PDB [%]	Min	PDB [%]	Min	PDB [%]	Min	PDB [%]
berlin52	2	1183.28	1489.07	25.84	1322.19	11.74	1289.37	8.97	1183.28	0.00
	4	661.76	775.49	17.19	711.96	7.59	748.29	13.08	661.76	0.00
	8	471.17	525.30	11.49	510.92	8.44	539.63	14.53	471.17	0.00
bier127	2	16449.88	19807.69	20.41	16922.80	2.87	17119.76	4.07	16449.88	0.00
	4	8521.63	9931.36	16.54	9015.81	5.80	9334.79	9.54	8521.63	0.00
	8	4736.58	5420.84	14.45	5297.57	11.84	5911.70	24.81	4736.58	0.00
gil262	2	542.52	628.76	15.90	555.46	2.39	555.53	2.40	542.52	0.00
	4	302.66	353.51	16.80	324.61	7.25	356.33	17.73	302.66	0.00
	8	202.59	217.63	7.42	208.03	2.69	242.54	19.72	202.59	0.00
lin318	2	9335.22	11413.25	22.26	10287.64	10.20	9790.94	4.88	9335.22	0.00
	4	5190.72	6135.12	18.19	5620.29	8.28	5732.17	10.43	5190.72	0.00
	8	3432.13	3896.34	13.53	3728.53	8.64	3933.45	14.61	3432.13	0.00
pcb442	2	11193.62	12978.45	15.95	12104.62	8.14	11750.33	4.97	11193.62	0.00
	4	5985.17	6428.72	7.41	6077.00	1.53	7001.67	16.98	5985.17	0.00
	8	3479.95	3755.38	7.91	3634.55	4.44	4461.43	28.20	3479.95	0.00
rat575	2	1565.36	1834.54	17.20	1628.98	4.06	1626.49	3.91	1565.36	0.00
	4	833.91	993.22	19.10	886.68	6.33	940.49	12.78	833.91	0.00
	8	477.37	558.01	16.89	528.37	10.68	581.84	21.88	477.37	0.00

Table 4.1. Table created with an empty table input YAML file.

```
General:
    Horizontals:    header
    Border:         hor
    Params:         [avg, min, stdev]

Top:
    Method:         [greedy, kmeans, multi]
    Params:         [[1, 2, 3]]

Left:
    Title:          [Problem, M]
    Problem:        [bier127, lin318, gil262, rat575]
    n:              N

Ratio:
    Set:            kmeans
```

Problem	N	M	BKS	greedy			kmeans			multi			R_{kmeans} [%]
				avg	min	stdev	avg	min	stdev	avg	min	stdev	
		2	16449.88	19807.69	19807.69	0.00	18865.86	17119.76	1414.27	17148.06	16449.88	335.18	95.25
bier127	127	4	8521.63	9931.36	9931.36	0.00	10658.23	9334.79	953.96	8723.96	8521.63	104.47	107.32
		8	4736.58	5420.84	5420.84	0.00	6429.14	5911.70	304.12	4818.49	4736.58	31.68	118.60
		2	542.52	628.76	628.76	0.00	577.17	555.53	16.26	553.77	542.52	7.19	91.79
gil262	262	4	302.66	353.51	353.51	0.00	357.79	356.33	0.79	311.83	302.66	4.15	101.21
		8	202.59	217.63	217.63	0.00	254.53	242.54	4.78	205.36	202.59	1.46	116.96
		2	9335.22	11413.25	11413.25	0.00	9883.74	9790.94	38.49	9560.68	9335.22	126.81	86.60
lin318	318	4	5190.72	6135.12	6135.12	0.00	5955.45	5732.17	82.21	5337.20	5190.72	55.79	97.07
		8	3432.13	3896.34	3896.34	0.00	4104.41	3933.45	161.42	3483.42	3432.13	35.43	105.34
		2	1565.36	1834.54	1834.54	0.00	1642.10	1626.49	6.43	1594.95	1565.36	13.42	89.51
rat575	575	4	833.91	993.22	993.22	0.00	974.37	940.49	25.62	845.63	833.91	6.07	98.10
		8	477.37	558.01	558.01	0.00	588.38	581.84	7.58	483.15	477.37	3.05	105.44

Table 4.2. Table with selected problem and method items.

```
General:
    BKS:            False
    Verticals:      left
    Horizontals:    Problem
    Bold:           left
    Params:         [avg, max, [Time, ms]]

Top:
    Method:         [greedy, kmeans, multi]
    Params:         [[1, 3], [1, 2, 3]]
    Display:        True

Left:
    n:              N
    Arrow:          True
    Order:          xproblem
```

Method			greedy		kmeans			multi		
Problem ↓	N ↓	NumRobots ↓	avg	Time [ms]	avg	max	Time [ms]	avg	max	Time [ms]
		2	1834.54	18.04	1642.10	1650.78	47593.50	1594.95	1616.74	35887.94
rat575	**575**	**4**	993.22	12.07	974.37	1011.63	21655.86	845.63	854.19	9007.18
		8	558.01	11.66	588.38	605.60	8614.40	483.15	487.33	9527.15
		2	12978.45	11.47	12046.45	12266.84	20864.45	11484.09	11727.02	14385.78
pcb442	**442**	**4**	6428.72	7.77	7040.14	7114.21	8855.52	6060.83	6155.10	3541.36
		8	3755.38	7.51	4551.94	4664.96	3649.27	3518.77	3540.65	2848.93
		2	11413.25	5.09	9883.74	9944.53	6918.84	9560.68	9914.32	3471.43
lin318	**318**	**4**	6135.12	3.64	5955.45	6134.60	2490.07	5337.20	5445.77	1361.71
		8	3896.34	3.83	4104.41	4510.05	1354.28	3483.42	3545.77	1517.15
		2	628.76	3.47	577.17	638.21	4407.77	553.77	566.29	2494.33
gil262	**262**	**4**	353.51	2.53	357.79	359.06	2700.82	311.83	320.65	779.66
		8	217.63	2.65	254.53	265.62	1079.10	205.36	207.62	696.21
		2	19807.69	1.00	18865.86	20887.70	623.40	17148.06	17778.53	289.10
bier127	**127**	**4**	9931.36	0.88	10658.23	12989.03	311.14	8723.96	8953.88	100.51
		8	5420.84	0.94	6429.14	6980.32	153.98	4818.49	4855.01	74.10
		2	1489.07	0.33	1381.59	1525.12	64.66	1194.99	1227.79	25.87
berlin52	**52**	**4**	775.49	0.32	785.09	825.11	40.59	666.22	671.83	9.67
		8	525.30	0.37	566.23	596.96	15.23	474.43	477.75	9.21

Table 4.3. Table with arrow signs and problem order flipped based on the problem specifier N.

```
General:
    Transpose:      True
    Verticals:      Method
```

```
    Horizontals:    all
    Params:         [avg, [PDM, \%], [Time, ms]]

Top:
    Method:         [greedy, kmeans, multi]
    Params:         [[1, 3], [1, 2, 3]]
    Display:        True

Left:
    Problem:        [berlin52, lin318, pcb442]
    Rename:         [[berlin52, Berlin], [lin318, Lin], [pcb442, PCB]]
    Rotate:         True
    n:              N
```

Method	Problem	Berlin			Lin			PCB		
	N	52			318			442		
	NumRobots	2	4	8	2	4	8	2	4	8
	BKS	1183.28	661.76	471.17	9335.22	5190.72	3432.13	11193.62	5985.17	3479.95
greedy	avg	1489.07	775.49	525.30	11413.25	6135.12	3896.34	12978.45	6428.72	3755.38
	Time [ms]	0.33	0.32	0.37	5.09	3.64	3.83	11.47	7.77	7.51
kmeans	avg	1381.59	785.09	566.23	9883.74	5955.45	4104.41	12046.45	7040.14	4551.94
	PDM [%]	16.76	18.64	20.18	5.88	14.73	19.59	7.62	17.63	30.80
	Time [ms]	64.66	40.59	15.23	6918.84	2490.07	1354.28	20864.45	8855.52	3649.27
multi	avg	1194.99	666.22	474.43	9560.68	5337.20	3483.42	11484.09	6060.83	3518.77
	PDM [%]	0.99	0.67	0.69	2.42	2.82	1.49	2.59	1.26	1.12
	Time [ms]	25.87	9.67	9.21	3471.43	1361.71	1517.15	14385.78	3541.36	2848.93

Table 4.4. Transposed table that utilizes the Rename and Rotate parameters.

```
General:
    Verticals:      Method
    Horizontals:    header
    Params:         [min, [PDB, \%], [Time, ms]]
    Units:          [under, none]

Top:
    Params:         [[1, 2]]

Left:
    Problem:        [berlin52, bier127, lin318, pcb442]

Bottom:
    Title:          [Letter, 3]
    Data:           none
```

4.2 Performance v. Comfort

Here is an example of a very simple set of YAML files. The database input

```
Include:    [WCost, Time]
```

Problem	NumRobots	BKS	greedy		grwdmp		kmeans		multi	
			min	PDB %	min	PDB %	min	PDB %	min	PDB %
berlin52	2	1183.28	1489.07	25.84	1322.19	11.74	1289.37	8.97	1183.28	0.00
	4	661.76	775.49	17.19	711.96	7.59	748.29	13.08	661.76	0.00
	8	471.17	525.30	11.49	510.92	8.44	539.63	14.53	471.17	0.00
bier127	2	16449.88	19807.69	20.41	16922.80	2.87	17119.76	4.07	16449.88	0.00
	4	8521.63	9931.36	16.54	9015.81	5.80	9334.79	9.54	8521.63	0.00
	8	4736.58	5420.84	14.45	5297.57	11.84	5911.70	24.81	4736.58	0.00
lin318	2	9335.22	11413.25	22.26	10287.64	10.20	9790.94	4.88	9335.22	0.00
	4	5190.72	6135.12	18.19	5620.29	8.28	5732.17	10.43	5190.72	0.00
	8	3432.13	3896.34	13.53	3728.53	8.64	3933.45	14.61	3432.13	0.00
pcb442	2	11193.62	12978.45	15.95	12104.62	8.14	11750.33	4.97	11193.62	0.00
	4	5985.17	6428.72	7.41	6077.00	1.53	7001.67	16.98	5985.17	0.00
	8	3479.95	3755.38	7.91	3634.55	4.44	4461.43	28.20	3479.95	0.00
Letter		A	B	C	D	E	F	G	H	I

Table 4.5. Table has non-default unit settings and has a tail.

and the table input

```
General:
    Horizontals:    header
    Border:         hor
```

The input files above generate table 4.6.

Problem	NumRobots	BKS	greedy		grwdmp		kmeans		multi	
			Min	PDB [%]	Min	PDB [%]	Min	PDB [%]	Min	PDB [%]
berlin52	2	1183.28	1489.07	25.84	1322.19	11.74	1289.37	8.97	1183.28	0.00
	4	661.76	775.49	17.19	711.96	7.59	748.29	13.08	661.76	0.00
	8	471.17	525.30	11.49	510.92	8.44	539.63	14.53	471.17	0.00
bier127	2	16449.88	19807.69	20.41	16922.80	2.87	17119.76	4.07	16449.88	0.00
	4	8521.63	9931.36	16.54	9015.81	5.80	9334.79	9.54	8521.63	0.00
	8	4736.58	5420.84	14.45	5297.57	11.84	5911.70	24.81	4736.58	0.00
gil262	2	542.52	628.76	15.90	555.46	2.39	555.53	2.40	542.52	0.00
	4	302.66	353.51	16.80	324.61	7.25	356.33	17.73	302.66	0.00
	8	202.59	217.63	7.42	208.03	2.69	242.54	19.72	202.59	0.00
lin318	2	9335.22	11413.25	22.26	10287.64	10.20	9790.94	4.88	9335.22	0.00
	4	5190.72	6135.12	18.19	5620.29	8.28	5732.17	10.43	5190.72	0.00
	8	3432.13	3896.34	13.53	3728.53	8.64	3933.45	14.61	3432.13	0.00
pcb442	2	11193.62	12978.45	15.95	12104.62	8.14	11750.33	4.97	11193.62	0.00
	4	5985.17	6428.72	7.41	6077.00	1.53	7001.67	16.98	5985.17	0.00
	8	3479.95	3755.38	7.91	3634.55	4.44	4461.43	28.20	3479.95	0.00
rat575	2	1565.36	1834.54	17.20	1628.98	4.06	1626.49	3.91	1565.36	0.00
	4	833.91	993.22	19.10	886.68	6.33	940.49	12.78	833.91	0.00
	8	477.37	558.01	16.89	528.37	10.68	581.84	21.88	477.37	0.00

Table 4.6. Table created with the above database and table inputs.

On the other hand, here is a set of more complex input files. The one for the database input

```
Path:       ./results
Main:       [Problem, Method]
Include:    [WCost, Time]
Precision:  1
```

and the one for the table input

```
General:
    Horizontals:    header
    Border:         hor
```

```
    Params:         [avg, min, stdev]

Top:
    Method:         [greedy, kmeans, multi]
    Params:         [[1, 2, 3]]

Left:
    Problem:        [bier127, lin318, rat575]
    n:              N

Ratio:              ["$R_{km:gr}$", "$R_{mu:gr}$"]
```

and together these files generate table 4.7.

Problem	N	BKS	greedy			kmeans			multi			$R_{km:gr}$ [%]	$R_{mu:gr}$ [%]
			avg	min	stdev	avg	min	stdev	avg	min	stdev		
bier127	127	4736.6	11720.0	5420.8	6058.7	11984.4	5911.7	5298.8	10230.2	4736.6	5192.2	102.3	87.3
lin318	318	3432.1	7148.2	3896.3	3177.9	6647.9	3933.5	2432.3	6127.1	3432.1	2565.9	93.0	85.7
rat575	575	477.4	1128.6	558.0	534.3	1068.3	581.8	439.2	974.6	477.4	466.9	94.7	86.4

Table 4.7. Table created with the database input from the input files listed above.

4.3 OpTeX conversion

For the following table YAML input, table 4.8 is created if the program is ran from the command line without the `-p` or the `--optex` parameter. If one of these switches is given, the table is generated in the OpTeX format, as is illustrated by table 4.9.

```
General:
    BKS:            False
    Verticals:      Method
    Horizontals:    Problem
    Params:         [avg, min]

Top:
    Method:         [greedy, kmeans]
    Params:         [[1, 2]]

Left:
    Title:          [Problem, M]
    Problem:        [berlin52, lin318, pcb442]
    n:              N
```

Problem	N	M	greedy		kmeans	
			avg	min	avg	min
berlin52	52	2	1489.07	1489.07	1381.59	1289.37
		4	775.49	775.49	785.09	748.29
		8	525.30	525.30	566.23	539.63
lin318	318	2	11413.25	11413.25	9883.74	9790.94
		4	6135.12	6135.12	5955.45	5732.17
		8	3896.34	3896.34	4104.41	3933.45
pcb442	442	2	12978.45	12978.45	12046.45	11750.33
		4	6428.72	6428.72	7040.14	7001.67
		8	3755.38	3755.38	4551.94	4461.43

Table 4.8. A classic LaTeX table.

Problem	N	M	greedy		kmeans	
			avg	min	avg	min
berlin52	52	2	1489.07	1489.07	1381.59	1289.37
		4	775.49	775.49	785.09	748.29
		8	525.30	525.30	566.23	539.63
lin318	318	2	11413.25	11413.25	9883.74	9790.94
		4	6135.12	6135.12	5955.45	5732.17
		8	3896.34	3896.34	4104.41	3933.45
pcb442	442	2	12978.45	12978.45	12046.45	11750.33
		4	6428.72	6428.72	7040.14	7001.67
		8	3755.38	3755.38	4551.94	4461.43

Table 4.9. Table 4.8 in OpTeX format.

Chapter 5
Libraries

5.1 MAPF-IR

Implemented by Keisuke Okumura Ph.D., it is a simulator and visualizer of Multi-Agent Path Finding (MAPF), which means the iterative refinement of path planning for multiple robots. [6]

Given a map and agents, where each agent is defined by its initial location and its goal coordinates, the solution is a set of paths. It is important to note that collisions are avoided and no two agents share the same coordinates at the same time.

There are many solvers implemented in this library, some are deterministic and others are random. For the testing of the evaluation environment, the deterministic algorithms were focused on. These include the priority inheritance with backtracking (PIBT, win-PIBT, PIBT_COMPLETE) algorithm, the hierarchical A star (HCA, WHCA) algorithm, and the conflict-based search (CBS, ECBS, ICBS).

The testing focuses on the comparison of the three PIBT versions. The basic PIBT solver is a decoupled method that solves MAPF iteratively with flexible priorities. [7]

winPIBT stands for windowed PIBT and it is a generalization of the PIBT solver, which adds a configurable time window to the original PIBT algorithm, and it allows the agents to look multiple steps ahead.

Table 5.1 nicely showcases how the application works for this library. It was generated from a database which used the following database input,

```
Path:       /mapfIR_path
Title:       MapfIR
Main:       [map_file, agents, solver]
Include:    [soc, makespan, comp_time]
Valid:      [makespan, -1]
```

and the table input,

```
General:
    Verticals:      Solver
    Horizontals:    all
    Border:         none
    Params:         [Avg, Min, makespan, [Valid, \%], [PDB, \%], [PDM,
\%]]

Left:
    Title:          [Map file, Agents]
    n:              False
    Arrow:          True

Top:
```

```
Title:          Solver
Solver:         [PIBT, winPIBT-5, PIBT_COMPLETE]
Params:         [[min, pdb, valid]]
Line:           True
Display:        True
```

Solver			PIBT			winPIBT-5			PIBT_COMPLETE		
Map file ↓	Agents ↓	BKS ↓	Min	PDB [%]	Valid [%]	Min	PDB [%]	Valid [%]	Min	PDB [%]	Valid [%]
arena.map	300	9504.00	12664.00	33.25	100.00	15089.00	58.76	100.00	12664.00	33.25	100.00
brc202d.map	1500	73981.00	863971.00	1067.83	100.00	-	-	0.00	863971.00	1067.83	100.00
lak307d.map	300	16455.00	19604.00	19.14	100.00	20634.00	25.40	100.00	19604.00	19.14	100.00
random-32-32-20.map	50	1096.00	1238.00	12.96	100.00	1363.00	24.36	100.00	1284.00	17.15	100.00
	70	1647.00	1848.00	12.20	100.00	2055.00	24.77	100.00	1848.00	12.20	100.00
	100	2562.00	2923.00	14.09	100.00	-	-	0.00	2923.00	14.09	100.00

Table 5.1. Table created with the database input from the input files listed above.

5.2 MAPF-LNS2

Another library used for testing is the MAPF-LNS2 library, but because it largely resembles the MAPF-IR library, with the only real difference being the data file format, table presentation was chosen to be ommited.

Chapter 6
Conclusion

The objective of this thesis was to create an environment in the Julia language, that would evaluate results of robotic experiments. The environment would first process these results as a batch of text files and subsequently store this data in a database. The program would then generate a LaTeX table from that database based on the user's specifications given with an interface designed to maximize the user's comfort.

The database creation was implemented using the SQLite database engine. It was optimized to work at high speeds and different tables were added to the database hierarchy to help with organization.

The user interface was created utilizing the YAML format for the input files. There are two of these files, one for the result file processing and database creation specifications, and one with the table parameters. With the feedback of the *IMR Group*, the design of the input files was then modified to best cover the possible demands a user could propose. There is an abundance of parameters so a table can still be described in great detail if desired. However, each parameter has its default value and the user can therefore give 2 empty files as the input and the environment will still create a database and generate a table. In addition, as was explained in chapter 3.4, a compromise between the two was implemented, where the program generates a YAML template with all the input options and the user can then edit only the parameters they need.

The environment was then tested on two libraries which implemented two different robotic problems. The program can be deemed a success, because the functionality was confirmed through this testing and all the implemented features work as intended.

The thesis was written in the OpTeX-based *CTUstyle3* template [3] often used by the students of CTU in Prague, which posed a problem with the table formatting that was in line with the LaTeX format. A special function that converts a table, in the form of a string, from the LaTeX to the OpTeX format was created to provide a quick and easy solution to this conflict of table formatting. Nevertheless, the presented tables are all in the LaTeX format to abide by the assignment specifications.

As for future improvements, a more general file processor will be implemented to limit the amount of times a user would need to reprogram the parsing process. The options for the *main parameters* will be expanded and a possibility of customized rounding for each parameter will be added.

Appendix A
Evaluation Environment Source Code

The up to date version of the application is located at `https://gitlab.ciirc.cvut.cz/verneto4/julia-latex`

References

[1] *Julia 1.9 Documentation.*
https://docs.julialang.org/en/v1/.

[2] Petr Olšák. *CTUstyle - Plain TeX Template for Theses on CTU in Prague.*
http://petr.olsak.net/ctustyle-e.html.

[3] Petr Olšák. *Macros for student's theses at CTU in Prague, using OpTeX.*
https://github.com/olsak/CTUstyle3.

[4] *SQLite.*
https://www.sqlite.org/index.html.

[5] *SQLite: PRAGMA Statements.*
https://www.sqlite.org/pragma.html.

[6] Keisuke Okumura. *Kei18/MAPF-IR: Iterative Refinement for Real-Time Multi-Robot Path Planning (IROS-21).*
https://github.com/kei18/mapf-IR.

[7] Xavier Defago Keisuke Okumura, Yasumasa Tamura. *Iterative Refinement for Real-Time Multi-Robot Path Planning.*
https://arxiv.org/abs/2102.12331.