

Global setup

In this chapter the global flowchart shown in figure 3 is first described. In the following sections the various classes, functions and decisions are described in detail. To make experiments repeatable without the need to remember all the settings, a save and open button has been added. A session consist of a text file with the main settings and one file for each game that has been created. For creating games Gamut and Gambit are used, we wrote our own function for finding the Pareto optimal cells of the payoff matrix.

Running the comparisons is the most time-consuming part and since the work machine has a quad core CPU multi-threading with a changeable amount of threads is used to lower the calculation time. The results of the comparisons are stored in an SQL database to allow lookups from both the QT program as the R code.

Some basic analysis were added to the framework because the functions were already there (NE & PO analysis). The replicator dynamics were added as well because changing the settings works easier with sliders then by changing values in a vector in R.

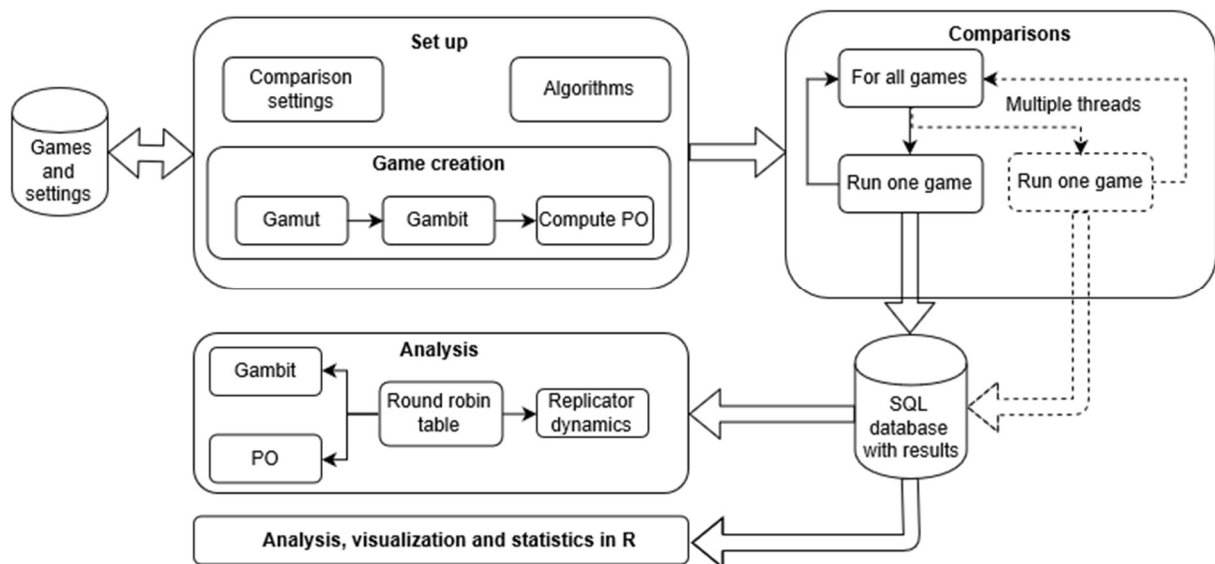


Figure 1, Global flowchart

Data container

As container we decided to use QVectors, these work similar to vectors in C++. The main reasons for this are:

- $O(1)$ lookup time of random elements, the most used functionality.
- Constructor which takes the size and initial value, for some reason this functionality is not implemented for QLists (which also has no $O(1)$ random access time)

Resizing the QVector is slower than resizing the QList, changing the code to use lists in these cases is not worth the effort as they are short vectors (size of the number of actions or players of the game.)

The only exception are the results of the comparisons, these are lists to which hundreds or thousands of items are added, for these sizes the slow copy all values to a new memory location as done with vectors becomes inefficient. Lists don't require all items to be stored sequential in memory which makes appending elements faster for big lists than vectors.

Classes

The program starts from main as shown in figure 4. Main creates the main GUI, the widget class. The GUI of the widget class contains only a QTabwidget, a window in which the other user are shown, with a tab bar to switch between them. An example is the settings window shown in figure 4. Next to housing the other GUI's the Widget class is also used to store 'global variables': it stores both the settings for the comparisons (comparison data) and the list of games. Upon creation each class gets a point to the widget class to enable them to access and edit the global variables. This is not the nicest solution but it is an easy way to synchronize the information between all classes.

Signals and slots

Next to global variable Qt allows objects to communicate with each other through signals and slots. This technique is used at various places in the framework, for example when a comparison is started to 'freeze' the GUI until the comparison is done. This is done because changes in the GUI change the values used by the comparison thread (all get pointers to the same instance).

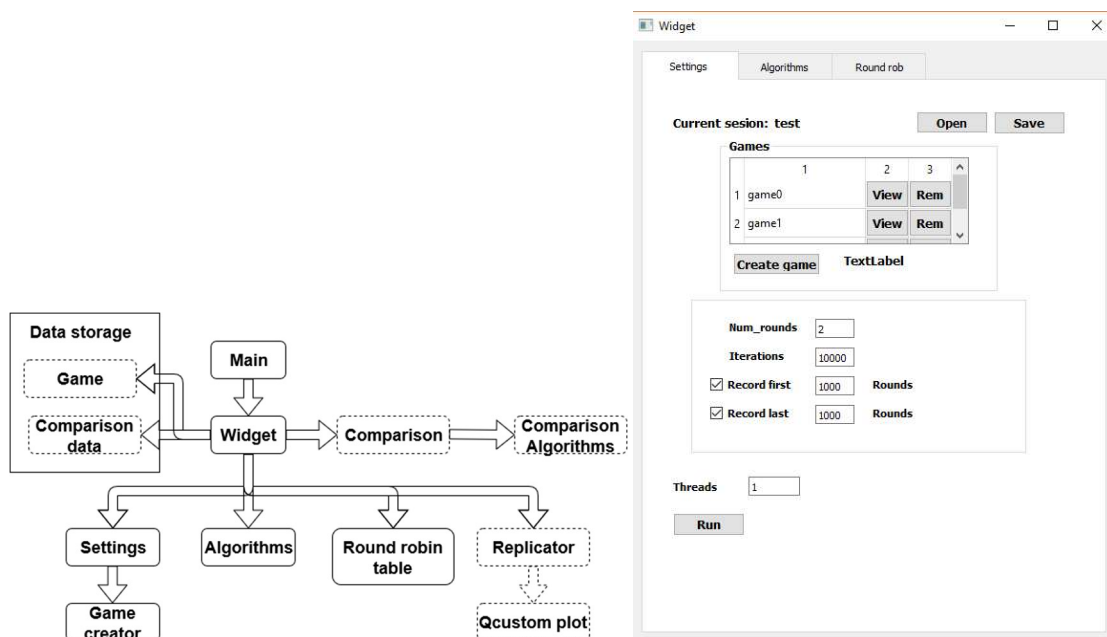


Figure 2, Left: class hierarchy, the arrows indicate the class which created them. Dotted boxes are classes that are only created when needed. Right: Settings window

When a comparison is started multiple threads can be started which each get their own instance of the algorithms class. The replicator class is created from the round robin table and gets a copy of the grandtable from the round robin class but more about this later.

File locations

In widget.h: the location of Gamut 'Gamut_loc', the folder containing the gambit tools: 'Gambit_loc' and the location of java.exe 'Java_loc'.

The folder containing the session that is opened upon starting the program is set in comp_data.h: 'Session'. The name of the database is stored in 'db_loc', set in comp_data.h.

Games

Game are shown in the settingstab, from here new games can be created, viewed or removed.

Game creation class

The game creator class handles the creation of games, the process is explained step by step.

The figure shows two side-by-side screenshots of a 'Game creation' form. Both forms have the same layout, but the data and state are different.

Left Screenshot (Game number: 3):

- Game number:** 3
- Generator:** RandomGame (dropdown)
- Players:** 2 (input field)
- Actions:** 2 (input field)
- ☒ **Normalize Payoffs**
- Min payoff:** 0 (input field)
- Max payoff:** 100 (input field)
- Nr of game:** 1 (input field)
- Payoff Matrix:** A 2x2 grid with all cells empty and containing '-,-'.
- NE:** A small orange box.
- Pareto opt: bold**
- Idle**
- Buttons:** Generate, Close

Right Screenshot (Game number: 0):

- Game number:** 0
- Generator:** RandomGame (dropdown)
- Players:** 2 (input field)
- Actions:** 2 (input field)
- ☒ **Normalize Payoffs**
- Min payoff:** 0 (input field)
- Max payoff:** 100 (input field)
- Nr of game:** 1 (input field, highlighted with a red box)
- Payoff Matrix:** A 2x2 grid with the following values:

	1	2
1	68,100	7,89
2	20,0	46,100
- NE:** A small orange box.
- Pareto opt: bold**
- Idle**
- Buttons:** Generate, Close

Figure 3, Left: creating a new game, Right viewing a game.

The game creator class gets a 'game_number' next to a pointer to the widget class. The 'game_number' is negative if a new game is being created (create game button settings tab) and opens the screen shown in figure 5(left). When a game is viewed, the 'game_number' contains the index of the game to open. This index is used to select the right game from the list of games in widget and shows it as seen in figure 5(right).

Game matrix layout

The Pareto optimal cells in the matrix are marked with a bold font. To indicate a NE the transparency (alpha level) of the background color is changed. Bright orange indicates a pure NE, a light shade of orange shows the cell is part of a mixed NE.

The layout is used to quickly get an overview of the game and also to assist in debugging the PO, NE code and the algorithms which use this information. Next to this it also shows the type of game quickly. Note: it currently only shows games with less than 4 players and less than 4 actions.

Game class

The game class is used as a custom data container with some basic functions. It has the following publicly accessible members:

- **Players:** the number of players of the game; -used by the algorithms-
- **Actions:** the number of actions of the game; -used by the algorithms-
- **Min_payoff:** the minimal payoff selected in the GUI; -only used for game creation-
- **Max_payoff:** the maximal payoff selected in the GUI; -only used for game creation-
- **Normalize:** status of the normalize checkbox; -only used for game creation-
- **Generator:** the name of the Gamut generator that was used to create the game; -used by the comparison class to check if the game is symmetric or not-
- **NE:** the list of NE's returned from gambit; -only used for game creation-
- **NE_d:** As some NE are mixed, and at least one of the gambit tools returns these as 23/69,46/69,0 instead of fractures. The output from gambit is converted to a list of doubles with a value for each action, for each NE;
- **Cel_col_alpha:** The NE are shown in orange in the reward matrix shown in figure 5, the transparency of the background color of each cell is stored in this vector;
- **Matrix:** the reward matrix as it is used by gambit -used only to show the matrix on the screen;
- **Matrix_i:** integer version of the reward matrix; -used in the calculations-
- **Pareto_opt:** vector with the rewards for all players for all pareto optimal cells. –Made because it might be use full but not used anywhere so far-
- **Pareto_opt_ind:** the indices of the pareto optimal cells in the game matrix. These are used, for example to make the text in the pareto optimal cells bold in the displayed matrix.
- **Stepsize:** a vector with the number of actions to the power of the player number. Used to convert the binplayer to the actual cell.
- **Conv_ind:** indexes for binplayer, used when converting a cell back to the actions of all players, we want player 1 first, then 0, then the rest.
- **Binplayer:** a vector with a '0' for each player.

Despite some fields being only used for game creation they are stored and saved to the hard drive anyway for debug purposes. It is also the only way to check what settings were really used to create the games.

Calculating the cell

There are various ways to store the game matrix, as a matrix, as a list, not to get started about the order. This is just a personal choice which can be changed later on but not without a lot of reprogramming. We went for an easy solution: a vector of vectors, for the game in figure 6, 27 vectors (one for each cell) each one containing a vector of 3 rewards.

Game matrix of a 3 player, 3 action game

		player 1		
		0	1	2
player 2	0	29,79,12	79,84,41	6,46,66
	1	5,71,99	12,80,59	91,14,74
	2	71,37,13	36,99,93	17,98,71
1	0	33,51,69	47,52,38	85,6,70
	1	77,27,34	14,0,51	99,66,4
	2	100,15,18	23,69,0	65,47,97
2	0	56,81,34	11,84,8	88,28,58
	1	9,17,10	99,15,2	3,75,69
	2	51,6,89	14,88,76	8,27,36

reward order in each cell is:
player 0, player 1, player 2

➔

		0	1	2
0	0	0	9	18
1	1	1	10	19
2	2	2	11	20
1	0	3	12	21
1	4	4	13	22
2	5	5	14	23
2	0	6	15	24
1	7	7	16	25
2	8	8	17	26

Figure 4, conversion from cell to index

Looping over all of player 2's actions is as simple as calculating the cell when player 2's action is '0' and then adding '9' (the step size of player 2) to the cell number each time to loop through all the possible rewards.

$$\text{stepsize for player } x = \text{number of actions}^{\text{number of players} - 1}$$

Using the step sizes of all players we can convert actions to cells and back:

	pl0	pl1	pl2	
action	0	2	1	
stepsize	3	9	1	*
	0	18	1	=19

Figure 5, example calculation of cell

There is one 'problem' in the formula: the step size for player 0 and 1 have to be switched because the column player is player 1.

From cell to players action

When calculating a player's action from a cell the problem comes back. As solution we created `Conv_ind`, a vector containing the player numbers: 1,0,2, rest of the players. As example let's reverse the example of image 7 to find the action of player 2:

The first indexed is '1', this is not the player number(2) so the modulo of the `cel/binplayer[1]` is taken, the same is done for the second index (player 0). The third value is 2, the player number. In this case the value is divided by its `binplayer` value(1) instead of taking the modulo.

- Player 1: $19\%9=1$
- Player 0: $1\%3=1$
- Player 2: $1/1=1$
- Players 2's action played action one.

This is the best method we could come up with, we challenge you to invent a better method.

Game creation process

When the create game button is pressed the 'on_Btn_create_clicked' function is called. This function does a few, **not failsafe!**, checks on the input values. Then it creates 'nr of game' instance of the selected game generator by calling Gamut with the parameters, gambit is called next. Followed by some conversion the calculation of the Pareto optimal cells. As last step the last created game is shown on the screen.

Function locations

Because the NE's and the Pareto optimal cells are calculated for both the games (game creator tab) and grand table (round robin tab) the related functions are placed in widget so both classes can access them using the pointer, our variant of global functions. Creating a base class and deriving both game_creator and grandtable from this would be nicer but it's also a less flexible solution.

Gamut

Four game generators are not implemented because these don't return a normal form game: GuessTwoThirdsAve, CongestionGame, GreedyGame, SimpleInspectionGame. The other 31 generators are implemented in the framework.

Actions

Gamut supports varying amounts of actions for players, For example: "-actions 2 4 5 7": player 1 has 2 actions, player 2: 4, player 3:5 and player 4:7. Where this functionality is interesting, it also makes processing the games more complicated. More important all games marked with 'See 3.2' in the manual are only symmetric if all players have the same number of actions. For these reasons we decided to use the same number of actions for all players.

Other settings

All games are created with integer payoffs with a 'int_mult' of '1', the games are stored in the 'GambitOutput' format. The user can change: the number of players and actions (if Gamut allows it for the selected generator, other ways the unsupported field are given a fixed value), whether the results are normalized (scaled proportionally to fall in the given range) and select the minimum and maximum rewards.

Payoffs

The payoffs are stored as integers, mainly to save memory over doubles(32 vs 64 bits. Times x1.000 recorded rounds adds up). It is worth mentioning that some algorithms use accumulated payoffs, stored in integers. The max value which fits in an integer is: $2^{32} = 4,29E^9$ divided by our selected maximal payoff of 100 means that **4,29E⁷** iterations can be recorded before an overflow occurs. This is enough for our purposes but others might have to change this to doubles.

Generator specific settings

Some generators accept additional parameters, these are set random but this can easily be changed to a similar input as used in MALT, there is also no option to create custom games. This can easily be added by replacing the output from gamut with the own reward matrix.

Implementation

Gamut is called as a process, this allows us to start it and wait for process to finish. After the process is finishes it has created a 'textfile' with the name Game 'gam number' in the session folder. This allows fast checking of the loaded games.

Gambit

For two player games the 'gambit- enummixed' tool is used, for games with more players 'gambit- enumpoly' is called. Both return all the NE of the game. Gambit reads files in a special way, it changes the player actions one by one, like a binary counter, as shown in table 1:

- the first rewards (1000000 568842 828803) goes to [0,0,0]: the first cell
- the second set of rewards (80923 64244 923769) goes to [1,0,0]: player 1 action 1, rest 0.
- Then [0,1,0], [1,1,0], [0,0,1] etc.
 - o The cell number can be calculated by multiplying the binary vector with a conversion vector: [2,4,1],[2, 4, 8, 1 for 4 players] from right to left (with player two and one switched). Note that this is a different vector then the stepsize vector!

This conversion is also made in the program (convert_gamematrix).

Gambit always moves the second players reward to the last position in the cell, this is only for displaying. Confusing enough Gambit outputs the second players (NE action profile) secondly ("1,0,1,0,1,0") as seen at the bottom of figure 8.

		1			2		
1	Player1 Payoff: 1000000.0000	1000000	828803	568842	384864	353632	50840
	Player2 Payoff: 568842.0000	185887	10000	918777	613311	770849	215550
2	Player3 Payoff: 828803.0000	80923	923769	64244	124499	841268	193089
		294776	767050	810791	144659	866299	927470
Profiles ▾ One equilibrium by simplicial subdivision in strategic game							
#	1: 1	1: 2	2: 1	2: 2	3: 1	3: 2	
1	1.0000	0.0000	1.0000	0.0000	1.0000	0.0000	

Figure 6, Gambit output for one game

Table 1, gamut output of a 3 player, 2 action game in Gambit format.

1000000	568842	828803	80923	64244	923769	384864	50840	353632	124499	193089	841268
185887	918777	10000	294776	810791	767050	613311	215550	770849	144659	927470	866299

Implementation

The command line tools are called as processes and return the output(NE) to the program. Gambit sometimes returns the NE's in separate messages and all at once on other runs. The messages are all stored in the global variable 'Gambit_NE_buff' and returned when the process finishes.

Conversion of the NE's

‘gambit-enummixed’ returns mixed NE’s fractures: 45/60, 15/60. While ‘gambit-enumpoly’ returns the NE’s as decimal values: 0.75, 0.25, the converter can handle both. The conversion function takes list of strings returned by gambit and converts them to doubles and stores the NE information of each player in a separate vector. An example is shown in table 2.

Table 2, convert NE to NE_d

NE strings for 2 players 2 action game (NE)	Converted vectors (NE_d)
0,1,1,0	{{0,1},{1,0}}
3/10,7/10,3/10,7/10	{{0.3,0.7},{0.3,0.7}}
1,0,0,1	{{1,0},{0,1}}

Finding the Pareto optimal cells

An outcome is Pareto optimal if there is no other outcome that increases at least one player’s payoff without decreasing anyone else’s. To check this the reward for each player of each cell must be compared against all other cells, well. Say we have checked cells 1-4 and are now checking cell 5, then we can ignore cell 1-4 and instead check the already marked as Pareto optimal cells. The set of Pareto optimal cells is smaller than the number of cells in the game for most games so this saves quite a few checks. It is still a time-consuming process, but a lot faster than finding all NE with Gambit. Both the rewards of the Pareto optimal cells and the indexes of the cells (position in matrix_i) are stored.

After this step the game creation is done and the game is stored in the games list in widget.

Showing the game table

After creation ‘fill_gametable’ is called to show the last created game. It only shows games with less than 4 players and actions, this is not a software restriction. Just something we made because it does not give any insight anymore. An second remark is that this function is also called when ‘create game’ is called from settings and shows a two by two game without rewards.

This is also the only place where the NE colors for the cells are calculated when they have not yet been. After this the rewards, NE color alpha levels and pareto optimal indexes(bold text) are shown on the screen.

Calculating col_cel alpha

The cell colors are determined in a tricky way: For each cell the algorithm loops over all NE for all players and using the action which is played to reach this cell. It looks up specific values in the NE_d array, say we have the values in table 2. The color of cell 0 is only influenced by the mixed NE so we can ignore the pure. To reach cell 0, player 0 and player 1 must both play ‘0’. The color value will then become: $1(\text{initial value}) * 0.3 * 0.3 = 0.09$.

We admit this does not the best but it works with multiple mixed & pure NE combined and serves its purpose of indicating what kind of game is being played.

Creating multiple games

While it is not explicitly mentioned after creating a game with the ‘Generate’ button, a new game can be created by pressing the button and another one etc. etc. When inside edit mode only the last created game will be saved (Game overwrites the previous one at each generate), in ‘normal’ mode all games are saved.

Closing the game_creator UI

Each time an instance of the game creator class is created its 'closing' signal is connected to a slot of the settings tab. The signal is sent from the close event of the UI to ensure that it is always emitted whether the UI is closed with the 'close' button or by pressing the 'X' in the top right corner.

```
connect(gamecr,SIGNAL(Game_createClosed()),this,SLOT(Game_Created()));
```

The slot in settings clears the list of games and rebuilds it using the games in the widget class. This is done to keep the list on the screen and the actual list of games in sync.

Sessions

A session consists of a text file with the comp_data, settings for the comparison and a file for each game with all the values from the game class. When the 'open session' button is pressed a directory can be selected. When the folder is selected the existing games in 'G_data' are removed and the game files in this directory are read into G_data. This is also done for the initial session when the program is started.

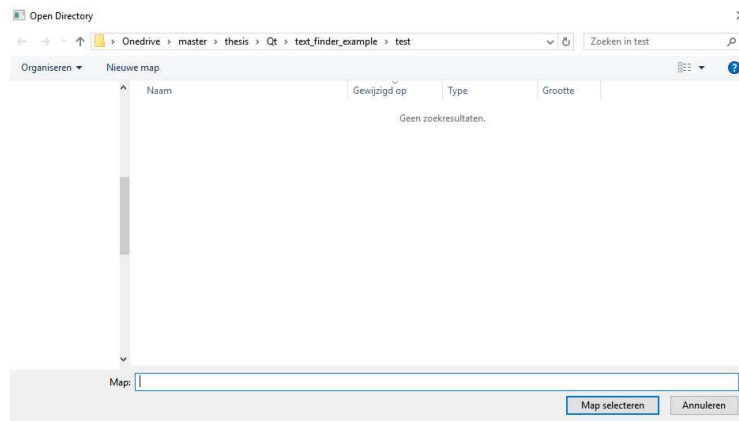


Figure 7, directory selection on a Dutch pc.

Saving a session works in a similar way, the games and compdata are written to files in the location which is currently used.

Algorithms class

The algorithms class is nothing more than a UI, shown in figure 10, that stores all settings directly in the comp_Data object in widget. There is one issue: sessions are opened from the settings tab and we would like to see these values in the UI as well.

The initial session is opened when the settings tab is created, at that moment the algorithms tab is not created yet. To alert the algorithms tab a Boolean (**signal_algorithms**) is set too true if a session was opened and the algorithms class displays the values from comp_data upon creation.

When a new session is opened after the initiation the settings tab sends a signal to the algorithms class to update the UI.

```
//In Widget
QWidget* pWid_set= ui->tabwid1->widget(0); // for the first tab:settings
QWidget* pWid_algo= ui->tabwid1->widget(1); // for the second
tab:algorithms
connect(pWid_set,SIGNAL(update_algos()),pWid_algo,SLOT(read_compmat()));
```

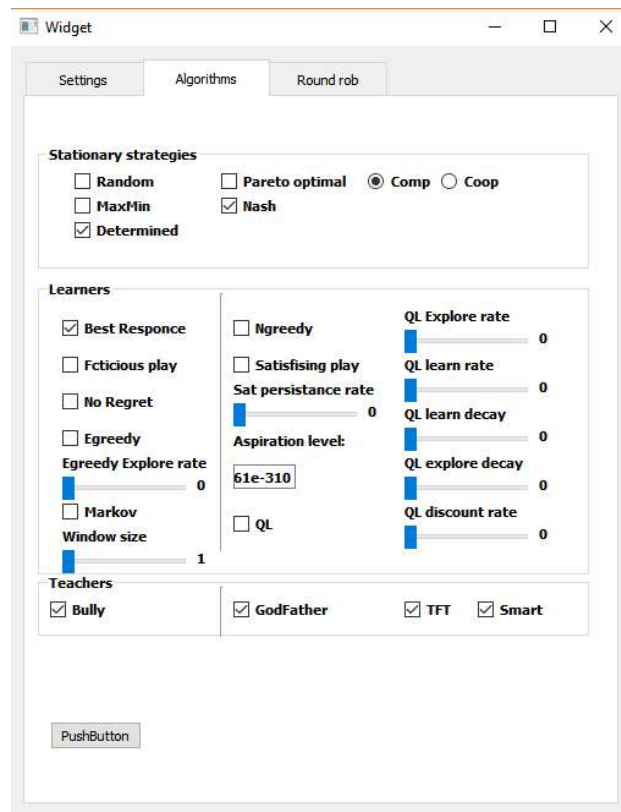


Figure 8, algorithms UI

Testing algorithms

The pushbutton at the bottom of the algorithms class is added a testing button. Behind it is the function used to test all algorithms by either taking a custom created game or one of the games from the lists and setting up a comparison with the test algorithm, faking the actions of the opponent we can check how the algorithms behaves in special situations, as example is shown below:

```
IterParam pars;
pars.game=&widget->G_data[0];
pars.plnum=0;
algos.append(new Markov(&pars,2));
pars.past_act={0,0};
qDebug()<<"init";
for(int i=0;i<20;i++)
{
    pars.iteration=i;
    qDebug()<<"-----";
    pars.plnum=0;
    pars.past_act[0]=algos[0]->action(&pars);
    int cel=pars.game->Calculate_bincel(pars.past_act,pars.Steps);
    pars.past_rews=pars.game->Matrix_i[cel];
    qDebug()<<"past_act"<<pars.past_act;
    qDebug()<<"past_rews"<<pars.past_rews;
}
```

Hyper parameter tuning

Finding the right parameters is a tricky and complicated process. We decided to use a parameter free hyper optimization: <http://blog.dlib.net/2017/12/a-global-optimization-algorithm-worth.html>

The installation & setup of the library is described below in DLIB setup along with a test program. Sadly the 'find_max_global' function cannot be used in our program as it required the function under test to be static. Luckily the library offers a solution for this: http://dlib.net/dlib/global_optimization/global_function_search_abstract.h.html

An example program:

```
#include <dlib/global_optimization.h>
dlib::function_spec spec_F({-10}, {1});
dlib::global_function_search opt(spec_F);
for (int i = 0; i < 20; ++i)
{
    dlib::function_evaluation_request next = opt.get_next_x();

    double a = next.x()(0);
    //double b = next.x()(1);
    double res=F(a);
    next.set(res); // Tell the solver what happened.

}
// Find what point gave the largest outputs:
dlib::matrix<double,0,1> x;
double y;
size_t function_idx;
opt.get_best_function_eval(x,y,function_idx);

qDebug() << "function_idx: " << function_idx;
qDebug() << "y: " << y;
qDebug() << "x0:" << x(0,0); //x0
qDebug() << "x1:" << x(0,1); //x1

where function F is:
double F(double x) {return -std::pow(x-2,2.0); }
```

This way normal functions can be used, making life a lot easier. The function is called by pressing the 'tune_params' button on the algorithms tab. As function it optimizes 'EGr_tune' which is a custom comparison ran on all games for the test_alg and a selected group of 'opponents'. The parameters for the test algorithm are generated by the parameter tuner. The other settings and games are taken from the UI.

On each call 'EGr_tune' calculated the average score of the test algorithm against the given opponents on all games in the gameslist(Widget-> G_data). For this only the comparisons with the test algorithm have to be run, only playing on both sides on asymmetric games. With our 6 opponents this results in 13 matches on asymmetric games (including selplay) and 7 matches on symmetric games.

Comp_data class

The comp_data class is just like the game class used as a custom data container. The algorithms class directly writes all values to this class and the settings class writes its values when the 'run' button on its UI is pressed. It has the following publicly accessible members:

- **db_loc:** the name of the database file in the session folder.

- **Session:** the full path of the current folder 'session' that's in use.
- **signal_algorithms :** the signal the settings class sends to algorithms when a new session is opened.
- **symmetric_games:** a list of all the symmetric game generators of Gamut.
- **Rec_first:** a Boolean flag that indicates if the first iterations are recorded or not.
- **Rec_last:** a Boolean flag that indicates if the last iterations are recorded or not.
- **Num_iteration:** the total number of iterations each round is played for.
- **Num_rounds:** number of rounds the algorithms play against each other on each game.
- **First_x_rounds:** the number of iterations that is recorded from the start of the round.
- **Last_x_rounds:** the number of iterations that is recorded counting from the end of the round.
- **Numthreads:** number of threads that is used to run the comparison.
- **Comparing_Algos:** the list of algorithms that participate in the comparison, comma separated list.
- **Par_comp:** 'true': competitive Pareto optimal is used, else cooperative.
- **Egr_expl:** exploration rate of e-greedy.
- **Sat_aspir:** Satisficing play's initial aspiration level.
- **Sat_pers_rate:** Satisficing play's persistence rate.
- **Ql_lear:** Q-Learner learning rate.
- **Ql_disc:** Q-Learner discount rate.
- **Ql_expl:** Q-Learner exploration rate.
- **Ql_dec_l:** Q-Learner decay learning rate.
- **Ql_dec_expl:** Q-Learner decay exploration rate.
- **TFT_smart:** 'True': smart Tit for Tat is used, false normal version is used.
- **Mark_win:** Markov window size.
- **Running_comp:** counter to keep track of what game is ran (too keep track since there are multiple threads fishing in the same pond).

Comparisons

A comparison is started by pressing the 'run' button on the settingstab, the function first stores all the values from the UI in the comp_data instance in widget. It also blocks all buttons on the tab and empties the list with games. After this, the comp_threadmanager in widget is called.

Starting a comparison

The 'comp_threadmanager' emits the starting comparison signal, connected to the roundrobin tab to inform it the database will be deleted. It also hides the tabbar at the top of the screen so the user cannot press buttons while the comparison runs.

```
//in roundrobin class
connect(widget, SIGNAL(starting_comparison()), this,
        SLOT(remove_files()));
connect(widget, SIGNAL(done_comparison()), this, SLOT(update_files()));
```

After this it removes all the tables from the database (setup_comp_db). In case the database file did not exist it is created automatically when the client tries to connect to it.

As last steps the manager creates an instance of the comparison class for each thread and passes a pointer to the assigned game and a pointer to the comp_Data class. For debugging purposes each thread gets a thread number and a bunch of signals is connected before the thread is released:

```
1.connect(worker, SIGNAL(error(QString)), this,
SLOT(errorString(QString)));
2.connect(thread, SIGNAL(started()), worker, SLOT(process()));
3.connect(worker, SIGNAL(start()), worker, SLOT(process())); //so thread
can restart itself.
4.connect(worker, SIGNAL(done_game(int,
comparison*)), this, SLOT(Next_game(int, comparison*)));
5.connect(worker, SIGNAL(finished()), thread, SLOT(quit()));
6.connect(worker, SIGNAL(finished()), worker, SLOT(deleteLater()));
7.connect(thread, SIGNAL(finished()), thread, SLOT(deleteLater()));
```

The connections are numbered to refer to them in the explanation:

- 1. Error catcher, in case something goes wrong it is nice to know what.
- 2. Used to start the first comparison when the thread is started.
- 3 & 4 are used to set up new games.
- 5, 6 & 7 are used to delete the class and close the thread when the last comparison is done.

Based on: <https://mayaposch.wordpress.com/2011/11/01/how-to-really-truly-use-qthreads-the-full-explanation/>

The compdata table in the database

For each game on which a comparison is run the widget class writes one line with information in the compdata table (**write_comp_line**). This information is later used in the analysis of the results. The field from left to right are:

- A comma separated list of the participating algorithms.
- The number of the game.
- Number of rounds.
- Number of iterations.
- How many first iterations were recorded.
- How many last iterations were recorded.
- The number of players in the game.
- The number of actions in the game.
- The generator with which the game was build.
- The reward matrix of the game.
- The celcol alpha values, if these were stored.
- The pareto optimal indexes of the game.

	players	gamenum	rounds	iterations	record_f	record_l	game_pl	game_a	generator	matrix	celcol	par_ind
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	best_resp,bull...	0	2	10000	1000	1000	2	2	RandomGame	68,100,20,0,7...	NULL	0,
2	best_resp,bull...	1	2	10000	1000	1000	2	2	PrisonersDile...	54,54,100,0,0...	NULL	0,1,2,
3	best_resp,bull...	2	2	10000	1000	1000	2	3	RandomGame	0,65,78,51,24...	NULL	3,6,

Figure 9, example compdata table in the sql database

Next to this the comparison class writes a table for each game, this contains one line for each recorded iteration. The 'FL' tells if the result is from a first or last recorded rounds. The rest of the fields speaks for itself (pl_ID, action and reward) for each player, **which works for x player games**.

	round	rec_FL	pl_id0	act_pl0	rew_pl0	pl_id1	act_pl1	rew_pl1
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	0	F	best_resp	1	46	best_resp	1	100
2	0	F	best_resp	1	46	best_resp	1	100
3	0	F	best_resp	1	46	best_resp	1	100
4	0	F	best_resp	1	46	best_resp	1	100

Figure 10, example compdata0 table, results of the first game, in the sql database

Managing the threads

After a thread has written its results to the database it emits signal 4 with the game number it ran and a pointer to itself. All threads do this, synchronizing them all with the 'manager', the widget class. Each thread also has mutex to prevent it from accidentally starting do changing values while the manager is changing the values.

The manager first grabs the mutex of the class that called it. Then, if there are still games left to run it gives the thread a pointer to a game that has not yet been run and releases the mutex. If there are no games left it changes the 'Running_game' variable to -1 and releases the mutex. Lastly the manager checks if there are still threads running, when he has shut the last one down the manager reactivates the UI and emit the 'done_comparison' signal. This signal is received by settings who reactivated the UI buttons and roundrobin table who shows the results of the first game.

```
//settingstab
connect(widget, SIGNAL(done_comparison()), this, SLOT(reactivate_tab()));
```

After releasing the mutex and emitting signal 4 the thread slept for a second to give the manager a chance to do its thing and then waits for the mutex. If it does not get the mutex the thread is closed after 6 seconds as safety procedure. Other ways it emits signal 3 to restart itself if 'Running_game'>0, or signal 6 if it is below zero.

Comp_algo's class

The comp_algo class houses all the algorithms in their own class, some of which have other algorithms as base class to get access to their functions. The base class has one virtual function, the action function used by all algorithms. This means they must all get the same parameters. For initialization the constructors of the derived algorithms classes are used, this way special parameters can be passed to the algorithms.

```
virtual int action(IterParam *params)=0;
```

iter_params class

The data object past to all algorithms contains the following members:

- **Iteration:** the current iteration of play.
- **Plnum:** the player number (0-x).
- ***game:** A pointer to the game object.
- **Past_act:** the past actions of all players
- **Past_rew:** the past rewards of all players

Comparison class

The comparison class itself is fairly simple, upon creation it seeds the random generator and waits for the thread to start. When the thread starts it calls the 'process' function, when a comparison is run the thread signals widget, requesting a new game.

Algorithms matcher

On each game all algorithms must play against each other, where we can make a distinction between asymmetric games on which the players have to play the game both as row and column player against each opponent and symmetric games which only have to be played once, the rewards of the other player are the same.

The 'symmetric_games' list in compdata is used to check if the game is symmetric. For asymmetric games all possible combinations with the number of players of the game are made with all algorithms. This is done by making a counter with one element for each player in the game. Let's for example say it's a 3 player game and there are 5 algorithms. This results in 3 counters from 0-4, the count of the right most counter is increased by one till the max value. When this happens the next counter is increased with one and the right counter repeats the process until all digits contain the max value, 4.

"000" > "001" etc. "004">"010" >"011"

Where each counter value represents an algorithm, zero the first and four the last.

For symmetric games the process is a little more complicated as "001", "010" and "100" are now the same game. The following solution is used: when the right most counter hits 4 it update the counter to the left of it "004">"010" but instead of going to zero it starts from '1' as well:

"000" > "001" > "002" > "003" > "004" > **"011"** > "012" > "013" > "014" > **"022"** > "023" > "024" > "033" > "034" > "044" > "111" > "112" > "113" > "114" > "122" > "123" > "124" > "133" > "134" > "144" > "222" > "223" > "224" > "233" > "234" > "244" > "333" > "334" > "344" > "444"

For each matching a comparison is ran (run_comparison), after all comparisons are run, the recorded iterations are written to the database and that's it.

Roundrobin table

This class is added to get some initial insight in the results after a comparison. We note that we don't show the results of 3 or more player games, this would require some conversions. An example is shown in figure 13, the game on top with the table below. The grand table is default shown for the average of the first & last recorded rounds, but using the dropdown menu the separate outcomes(first, last rounds) can also be selected. Both the game and the results of the comparison are queried from the database.

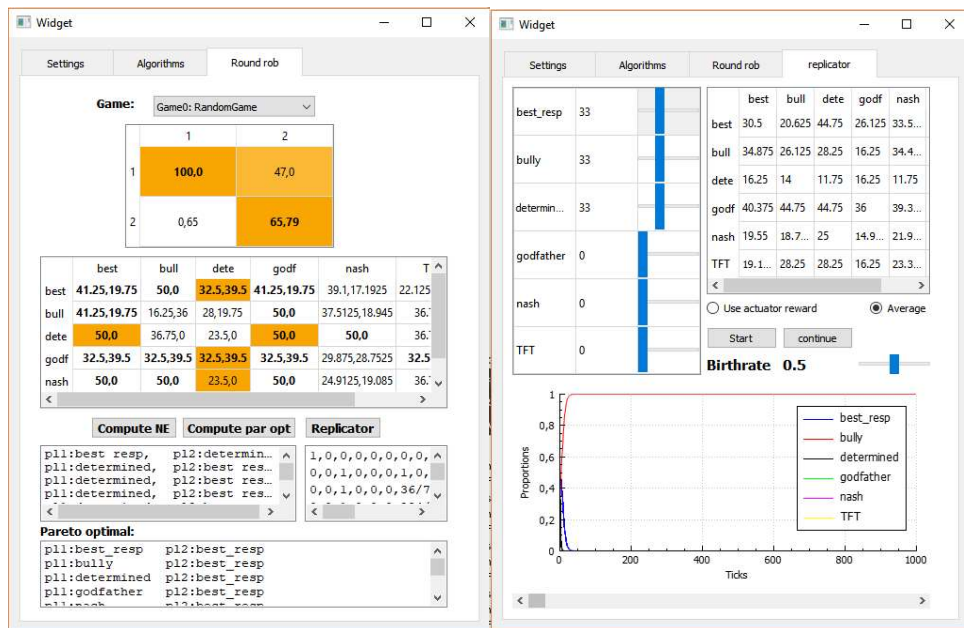


Figure 11, left: round robin table tab. Right, replicator dynamic

For an initial analysis the Pareto optimal algorithm & Gambit can be called. The results are both shown in text and through layout, similar as done with the game, in the grand table. The code is self-explanatory, only the replicator dynamic is a bit special.

Replicator dynamic

When the replicator dynamic button is pressed in the roundrobin table, the class calls the add replicator tab function in widget with the current grand table:

```
widget->add_tab_repl_dyn(conv_roundr_table_d(),Players);
```

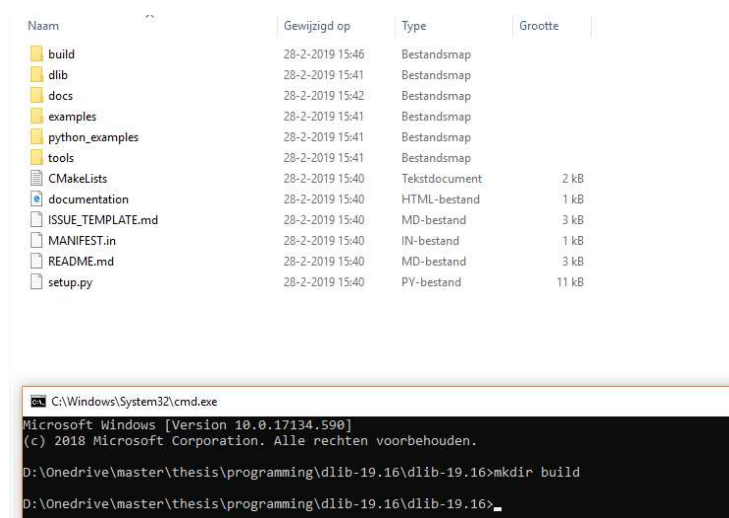
The grandtable in the replicator dynamic has only one value in each cell. The user can use the radio buttons to switch this between using the reward of the actors (column player) and averaging the rewards players earned as row & column player (showing the column players reward):

Bully earned '41,25' as row player against best response and '0' as column player, $(41,25+0)/2=20,625$ as average reward, shown in the table.

For the rest the starting proportions can be set, together with the birthrate. To show the graph 'qcustomplot' library is used: <https://www.qcustomplot.com/index.php/introduction>

Installation & setup of DLIB:

- Download from website: http://dlib.net/optimization.html#find_max_global
- Unzip.
- Inside the folder > cmd > 'mkdir build'



- Install mingw (or build using visual studio): <http://www.mingw.org/>

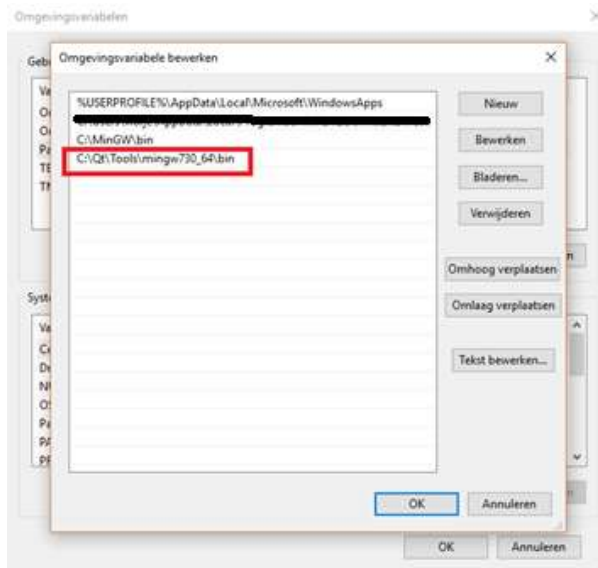
<input type="checkbox"/>	mingw-developer-toolkit-bin	2013072300	An MSYS Installation for MinGW Developers (meta)
<input checked="" type="checkbox"/>	mingw32-base-bin	2013072200	A Basic MinGW Installation
<input type="checkbox"/>	mingw32-gcc-ada-bin	8.2.0-3	The GNU Ada Compiler
<input type="checkbox"/>	mingw32-gcc-fortran-bin	8.2.0-3	The GNU FORTRAN Compiler
<input checked="" type="checkbox"/>	mingw32-gcc-g++-bin	8.2.0-3	The GNU C++ Compiler
<input type="checkbox"/>	mingw32-gcc-objc-bin	8.2.0-3	The GNU Objective-C Compiler
<input type="checkbox"/>	msys-base-bin	2013072300	A Basic MSYS Installation (meta)

- Note: when you install MinGW, the minGW/bin directory is not added to the PATH variable. So you need to add your minGW/bin directory to your path variable, so CMake can find the needed libraries.

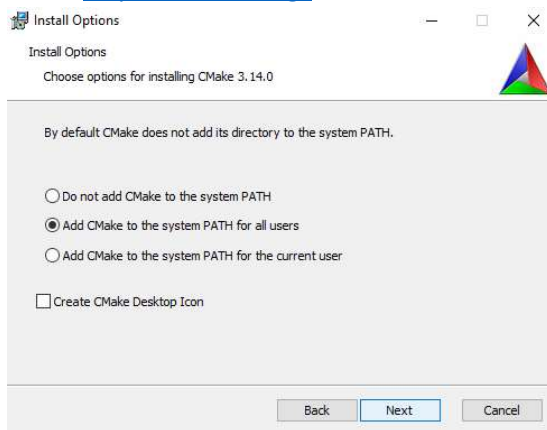
- We use QT so instead:

- 'omgevingsvariabelen' on a Dutch pc

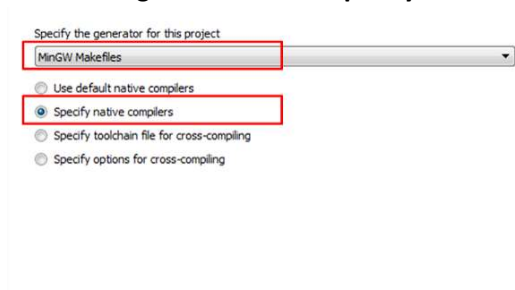
<https://www.architectryan.com/2018/03/17/add-to-the-path-on-windows-10/>



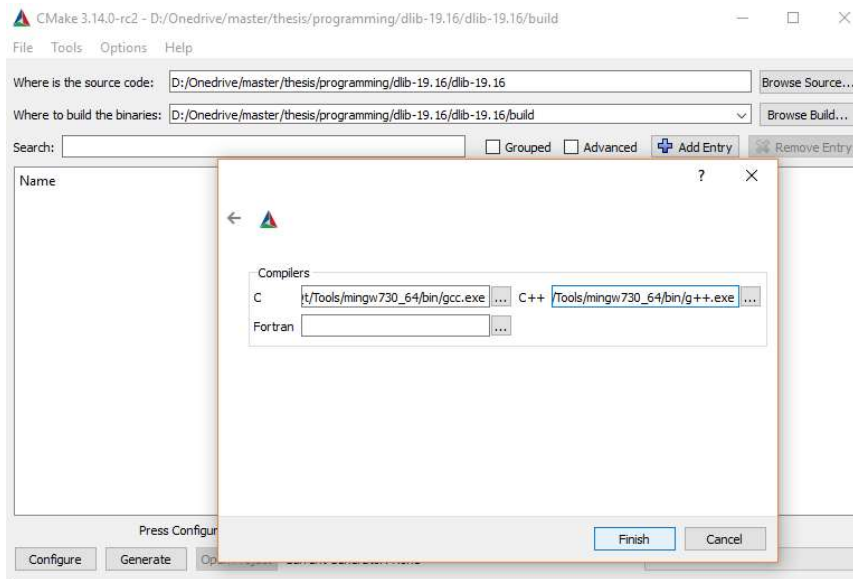
-
- Install cmake: <https://cmake.org/>



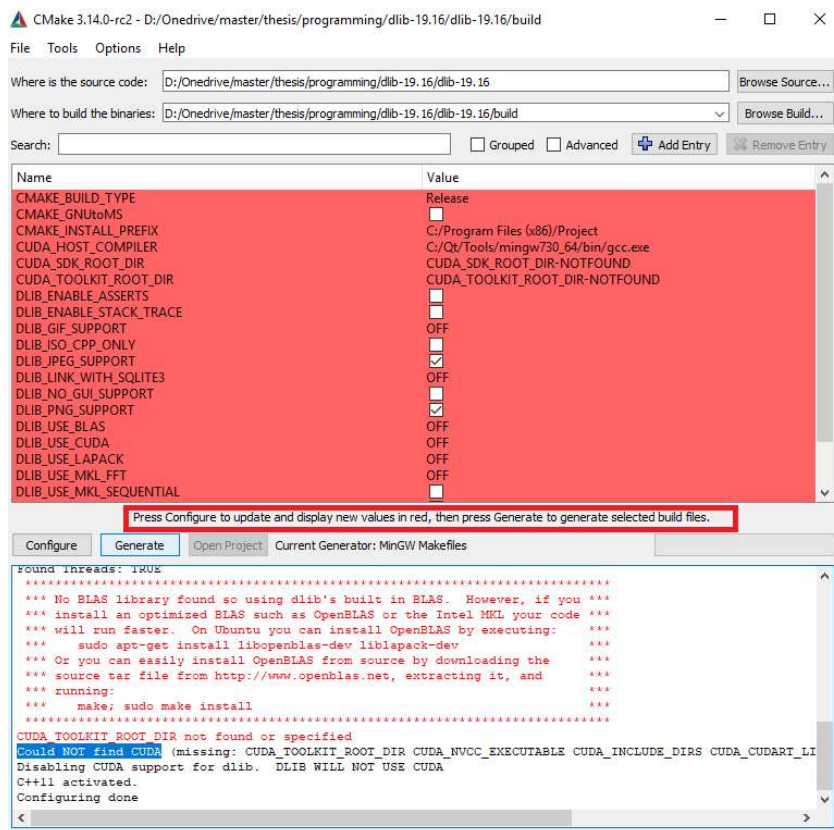
-
- Create an empty folder called "build" in the dlib folder
- Start cmake
 - Press configure and select **'specify native compilers'**



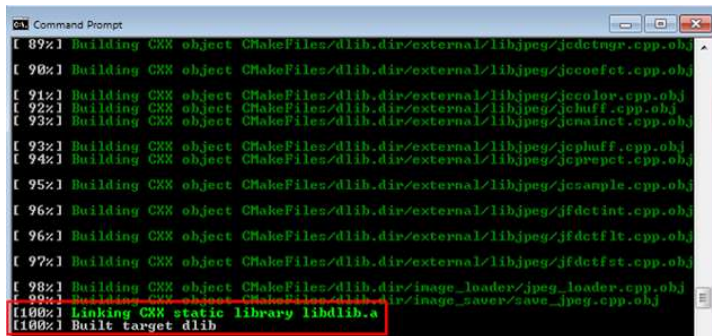
-
- Compilers,
 - C: C:\Qt\Tools\mingw730_64\bin\gcc
 - C++: C:\Qt\Tools\mingw730_64\bin\g++



- Press finish and then generate



- Open "cmd" and go to dlib-19.4/build folder and write "mingw32-make"



```

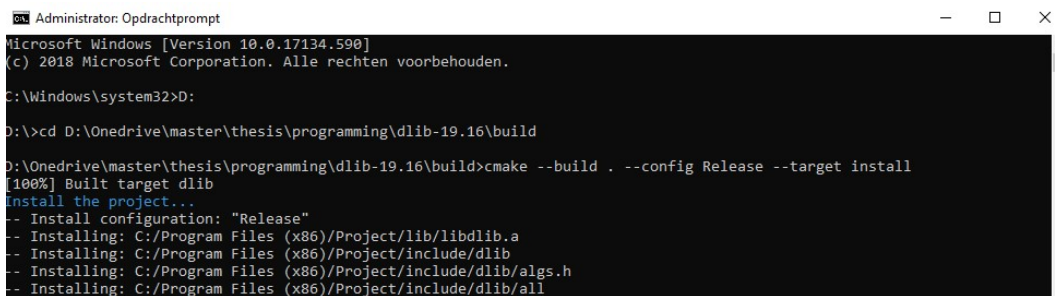
[ 89%] Building CXX object CMakeFiles/dlib.dir/external/libjpeg/jcctnrg.cpp.obj
[ 90%] Building CXX object CMakeFiles/dlib.dir/external/libjpeg/jccoeffet.cpp.obj
[ 91%] Building CXX object CMakeFiles/dlib.dir/external/libjpeg/jccolor.cpp.obj
[ 92%] Building CXX object CMakeFiles/dlib.dir/external/libjpeg/jchuff.cpp.obj
[ 93%] Building CXX object CMakeFiles/dlib.dir/external/libjpeg/jcmaint.cpp.obj
[ 94%] Building CXX object CMakeFiles/dlib.dir/external/libjpeg/jcpuff.cpp.obj
[ 95%] Building CXX object CMakeFiles/dlib.dir/external/libjpeg/jcsample.cpp.obj
[ 96%] Building CXX object CMakeFiles/dlib.dir/external/libjpeg/jfdctint.cpp.obj
[ 96%] Building CXX object CMakeFiles/dlib.dir/external/libjpeg/jfdctflt.cpp.obj
[ 97%] Building CXX object CMakeFiles/dlib.dir/external/libjpeg/jfdctfst.cpp.obj
[ 98%] Building CXX object CMakeFiles/dlib.dir/image_loader/jpeg_loader.cpp.obj
[ 99%] Building CXX object CMakeFiles/dlib.dir/image_loader/save_image.cpp.obj
[100%] Linking CXX static library libdlib.a
[100%] Built target dlib

```

<http://eyyuptemlioglu.blogspot.com/2017/04/dlib-installation-on-windows-and.html>

This did not completely work yet, adding the following helped:

- Open cmd **as admin**
 - Navigate to the build folder, in my case:
 - D (switch hdd)
 - `cd D:\>cd D:\Onedrive\master\thesis\programming\dlib-19.16\build`
 - and type: "`cmake --build . --config Release --target install`"
- The build files are stored in: C:/Program Files (x86)/Project
 - A include and a lib folder.



```

Administrator: Opdrachtprompt
Microsoft Windows [Version 10.0.17134.590]
(c) 2018 Microsoft Corporation. Alle rechten voorbehouden.

C:\Windows\system32>D:

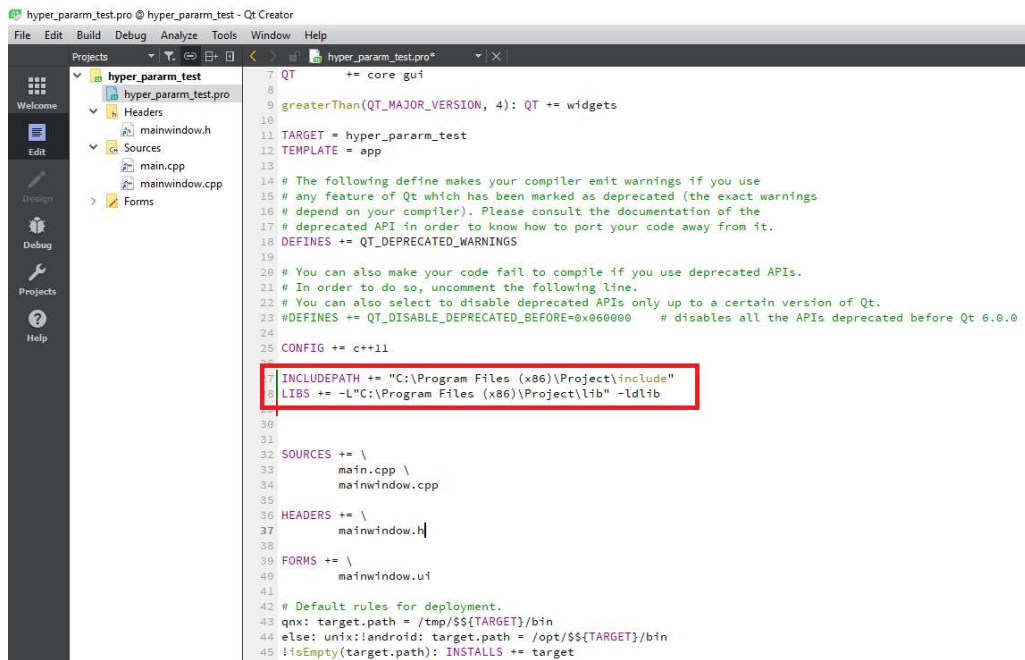
D:\>cd D:\Onedrive\master\thesis\programming\dlib-19.16\build

D:\Onedrive\master\thesis\programming\dlib-19.16\build>cmake --build . --config Release --target install
[100%] Built target dlib
Install the project...
-- Install configuration: "Release"
-- Installing: C:/Program Files (x86)/Project/lib/libdlib.a
-- Installing: C:/Program Files (x86)/Project/include/dlib
-- Installing: C:/Program Files (x86)/Project/include/dlib/algs.h
-- Installing: C:/Program Files (x86)/Project/include/dlib/all

```

Now we open QT

- Add two lines in the pro file:
 - `INCLUDEPATH += "C:\Program Files (x86)\Project\include"`
 - `LIBS += -L"C:\Program Files (x86)\Project\lib" -ldlib`



And done!

<https://stackoverflow.com/questions/38216510/dlib-on-qt-windows-the-program-has-unexpectedly-finished>

A little test code

- In the h file:
 - `#include <dlib/global_optimization.h>`
 - `static float holder_table(float x0, float x1);`
- in c file:

```
auto result = dlib::find_min_global(holder_table, {-10, -10}, {10, 10}, dlib::max_function_calls(100));
```

```
auto result = dlib::find_min_global(holder_table, {-10, -10}, {10, 10}, dlib::max_function_calls(400));
```

```
qDebug() << "size of x is:" << result.x.size();
qDebug() << "optimal value is:" << result.x(0, 0); //x0
qDebug() << "optimal value is:" << result.x(0, 1); //x1
qDebug() << "optimal value is:" << result.y; //min value of the function.
```

```
result = find_max_global([](double x){ return -std::pow(x-2, 2.0); }, -10, 1, dlib::max_function_calls(10));
qDebug() << "optimal value is:" << result.x; //optimal x value is one
qDebug() << "optimal value is:" << result.y;
```

```
float MainWindow::holder_table(float x0, float x1)
{
    float pi = 3.14159265359;
    return -abs(sin(x0) * cos(x1) * exp(abs(1 - sqrt(x0*x0 + x1*x1) / pi)));
}
```

And it works!!!

http://dlib.net/dlib/test/global_optimization.cpp.html

results also come really close to the real values:

<https://www.sfu.ca/~ssurjano/holder.html>

info about the **results matrix**:

http://dlib.net/matrix_ex.cpp.html

Bibliography

McKelvey, Richard D, Andrew M McLennan, and Theodore L Turocy. 2006. "Gambit: Software tools for game theory."

Nudelman, Eugene, Jennifer Wortman, Yoav Shoham, and Kevin Leyton-Brown. 2004. "Run the GAMUT: A comprehensive approach to evaluating gametheoretic algorithms": 880–887.

Zawadzki, Erik, Asher Lipson, and Kevin Leyton-Brown. 2014. "Empirically evaluating multiagent learning algorithms." arXiv preprint arXiv:1401.8074.

Websites

"MATLAB release history." n.d. <https://en.wikipedia.org/wiki/MATLAB>, Last accessed on 2018-12-10.