Hyper Parameters
Final Report

Developed By

Fin Schuiling, SCHFIN001

Bradley Culligan, CLLBRA005

Patrick Collins, CLLPAT012

Supervised by

Johan Bontes

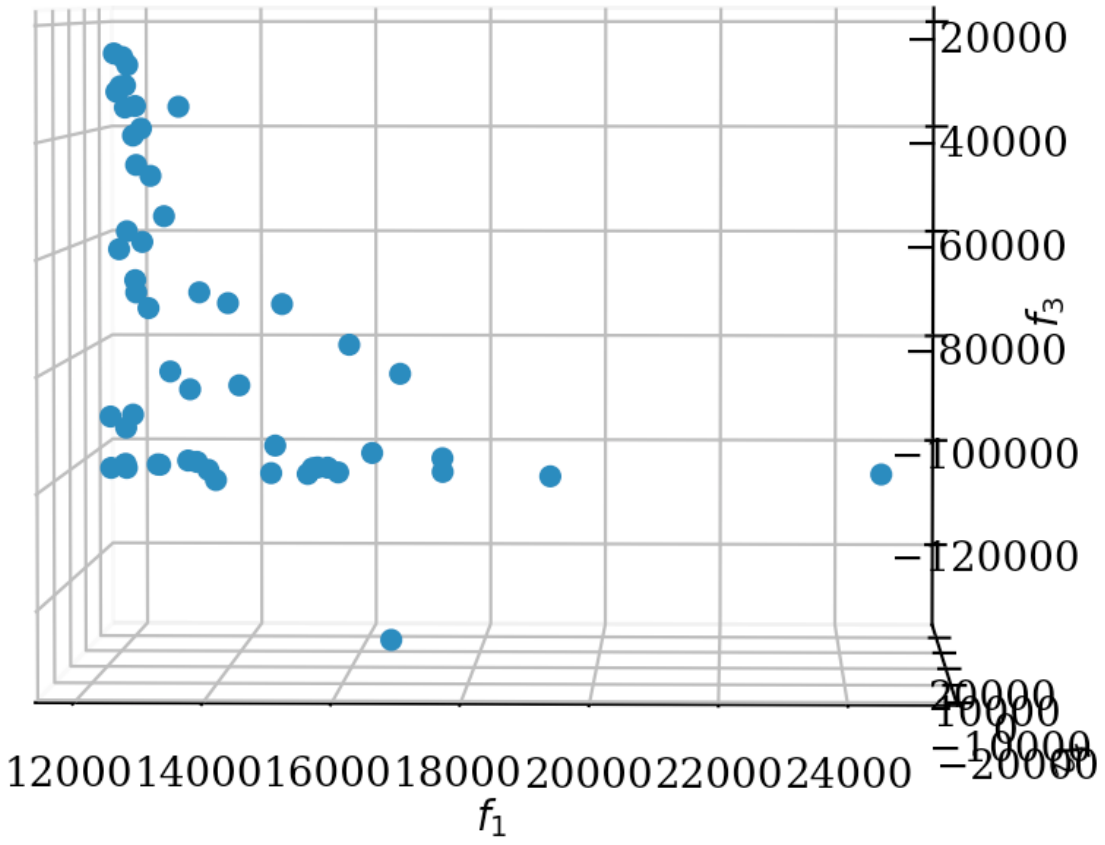
09/10/2021

Index

Abstract

The goal of this system is to make use of the carlSAT satisfiability solver by means of the pymoo genetic algorithm NSGA2. The latest prototype in the evolution of our project is that of the incremental solver. The general procedure of this algorithm is to run the carlSAT solver with a set of initial, random values. In subsequent generations of NSGA2, the state file in closest proximity to current parameters will be attached to the given member such that processing time will ideally be decreased. This enables shorter run times due to fewer instances of reprocessing the same inputs, allowing us to obtain more optimal results in the same time frame.

From this method, we achieved the following results. As can be seen in the plot below, our output data strongly follows an exponential distribution. This is exactly as can be expected from this solver as the objectives are diminishing over time. The initial few generations are composed of essentially uniformly distributed members and subsequently the members clearly improve towards the optimal solution. This is as to be expected with the genetic algorithm, where the best performers of each generation are fed into the next and so on.



Introduction

Many industrial problems can be reduced to computer science concepts. These may be a combination of but are not limited to, the travelling salesman problem, graph colouring and the bin packing problem. These are often used to optimise certain industrial procedures and thus reduce the cost or time in some manner for the industrial processes.

The problems are computationally hard for which no known method exists that can generate optimal solutions in polynomial time and as the number of items to optimise increases, the complexity of the problem increases exponentially. However, there is a large subset of these NP-Hard problems which are much easier to solve. Industrial problems often fall into this subset because they have a lot of internal structure.

We now present an optimizer program targeted towards this cohort of problems. Partial MaxSAT is a variant of a modern (2020) optimization solver, MaxSAT, which aims to optimise a set of soft requirements which we are trying to optimise whilst also ensuring that any hard requirements are satisfied (typically since violating them would break some law of nature). However, this solver will slow to some sub-optimal solution. and thus in this project, we shall be using a novel MaxSAT solver called CarlSAT. CarlSAT outperforms the current best MaxSAT solvers in the outlined problems by a factor of 100x to 10 000x because CarlSAT ‘understands’ the internal structure of the problem. Therefore, CarlSAT is a specialised tool and thus may not work or perform poorly on differently structured problems.

This solver has many tuning parameters which behave in non-intuitive ways and thus for two similar problems, a completely different set of tuning parameters may make the program perform optimally. Thus there are a large number of tuning parameters with very little cohesion in their specific choices - this makes it very difficult for someone to choose the optimal set of tuning parameters for the CarlSAT solver to function optimally. It might be useful to interpret this as a piece of technology that will function well when set up correctly, but whatever determines the ‘correct’ setup changes with every new problem given to the solver.

Thus, in this project, we aim to develop a system, called HyperOpt, which will tune the parameters of the CarlSAT program. It will achieve this by performing a series of short runs wherein with each run, the system will try to improve the current best solutions found. The program will learn how parameters from previous runs are performed and use this information to tune the parameters for future runs. The final program will then be able to learn about the optimal set of parameters as it uses CarlSAT to perform the solving and thereby optimise the performance of the solver overall. Whilst ensuring that HyperOpt obtains the optimal set of tuning parameters for CarlSAT is the main objective, the program should also use various techniques to ensure that it is functioning at its optimal level. For example, close to 100% of the processing power of the machine running HyperOpt should be functioning nearly all of the time. This is required to increase the throughput of parameter searching and thus decrease the amount of time required of the application to find the optimal set of tuning parameters. Various other techniques will be explored here to ensure an optimally performing piece of software is developed. Finally, HyperOpt’s performance will be compared to IBM’s commercial CPlex solver on several problems as a set of benchmark tests to ensure that the system developed is capable of market competition.

Upon the completion of the system, CarlSAT (in conjunction with HyperOpt) will be competitive with the current best satisfiability solvers. The practical purpose of such a system could save companies money if they have a problem that fits the solver.

Requirements Captured

This problem has been divided into two evolutionary prototypes which will follow sequentially from each other by adding and modifying methods and goals as the project progresses.

By the conclusion of this project, the required prototypes and their respective goals are defined as follows:

- HyperOpt shall be able to solve the 30 ‘simple’ problems optimally within a few milliseconds when compared with IBM’s commercial CPlex solver.
- Achieve optimisation within 0.5% of optimal, within 10 seconds for at least 2 of the 500 ‘hard’ problems when compared with IBM’s commercial CPlex solver.
- Deliver 3 working prototypes of solutions to the problem (as specified in the project specification sheet) by each prototype’s appropriate due date with the associated deliverables for each prototype
    - **Prototype 1 (20 August 2021):**
        - A Gitlab repo containing:
            - A ReadMe file listing the exact steps needed to get the project running on the client’s (Johan Bontes) machine
            - A Dockerfile that builds the project
            - The CarlSAT executable (as provided)
            - Python code containing prototype I

    - **Prototype 2 (15 October 2021):**
        - A new Gitlab repo containing:
            - A ReadMe file listing the exact steps needed to get the project running on the client’s (Johan Bontes) machine
            - A Dockerfile that builds the project
            - The CarlSAT executable (as provided)
            - Python code containing prototype II
            - Some code or other structure that contains or creates the database
            - Explanation of the text file format holding the parameter lineage data

- A SQL database, according to correct database design methods, for storing parameters and ancestry tracking for use.
- All CPU cores should be 100% occupied for most of the time during execution for all prototypes and thus working by 20 August

Design Overview

The HyperOpt class is the focal point of our project. This could be seen as the central class which either directly or indirectly depends on every other class. Here we take care of the Presentation layer of the layered architecture we have implemented. In terms of User Interface in the presentation layer, we have implemented a command-line interactive system where all relevant outputs are printed to the terminal and the user could enter the parameters for the system when calling the executor.

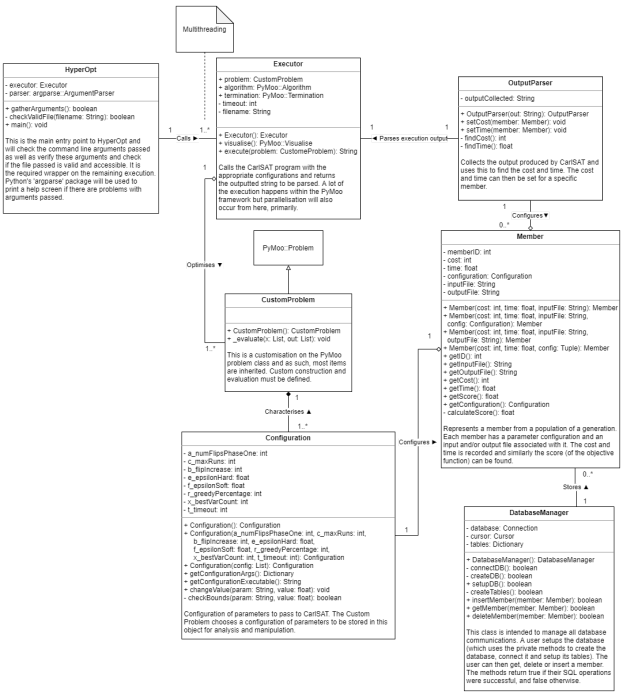
From here follows the executor class. This is the class whose sole responsibility is to initiate the thread pool which will be utilized to process all of the data. Once thread pool is initialized, each thread subsequently makes use of the execute function where the genetic algorithm, NGSA2, is stipulated as well as other parameters including the custom problem that carlSAT is solving.

To use this execute method, the custom problem class is required. This class is tasked with transforming the input data files into a form where the PyMoo algorithm can read them. To follow the idea of the Single Responsibility Principle, we extracted all this processing into their classes and used an Executor class that would handle calling the minimization. This would assist in further scaling at a later point if necessary and ensure that our code has high cohesion and low coupling.

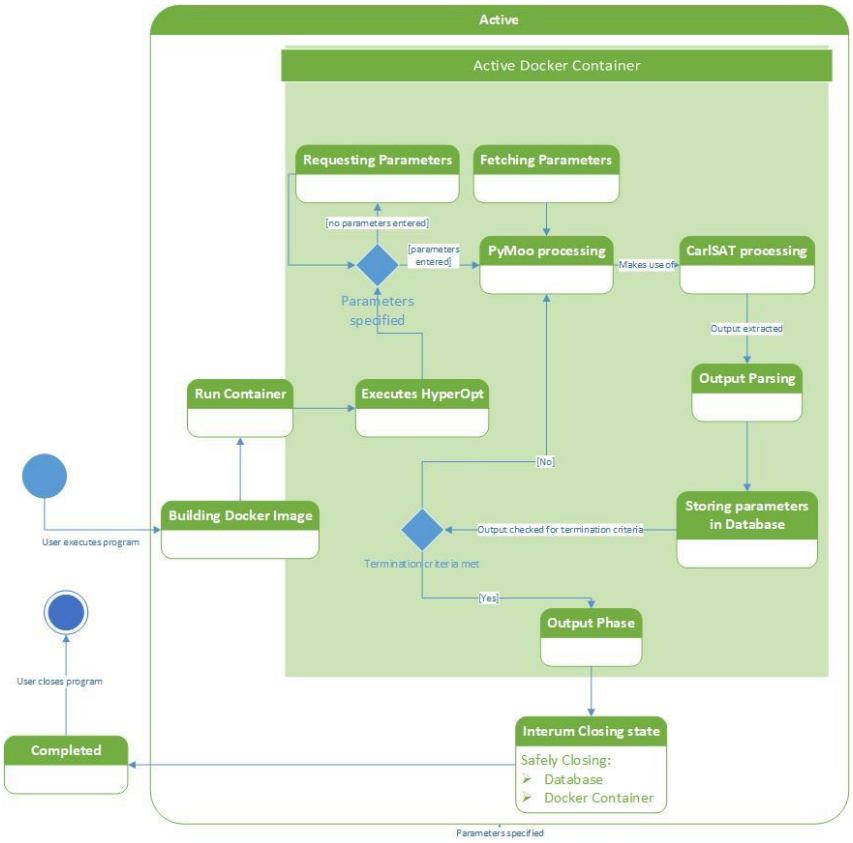
For all of these calculations to be stored, the DBWrapper class rises to the responsibility of storing these outputs in the database associated with our system. This database stores all previous runs of carlSAT. This includes the parameters used as well as the respective outputs which were achieved from these outputs. The DBWrapper writes these outputs into the database and reads out relevant records when they are required for calculation.

On each run, a member is associated with the most similar past state file. This is done to save processing time. To quantify this similarity, the OrdinaryLeastSquares class performs calculations on the A, E and F values to be able to choose a state file which it then outputs such that the member In the current generation can use this state file as a parameter in its execution.

UML Class Diagram



State flow



Database Design

InputFile used as table name

Variable Name / Description	Type	Range
ID (PK)	int	0..maxint
Member Number	int	1..maxint
Generation	int	1..maxint
Number of flips in phase one (a)	int	1..1000
Max count of runs before parameter ‘a’ is increased (c)	int	10..1000000
Increase in flips after ‘c’ runs (b)	int	1..1000
Epsilon for hard clauses (e)	float	0.1..1000
Epsilon for soft clauses (f)	float	0.1..1000
Percentage (between 0 (greedy) and 100 (random)) for selecting the best or random variable to flip (r)	int	1..100
A count ≥ 1 that limits the variable picker to the best ‘x’ variables (x)	int	1..10000
m	float	1..maxint
infilexx	int	1..maxint
Cost (output)	int	0..maxint
Time (output)	float	0..maxfloat
ABig	int	1..maxint
BBig	int	1..maxint
CBig	int	1..maxint
DBig	int	1..maxint

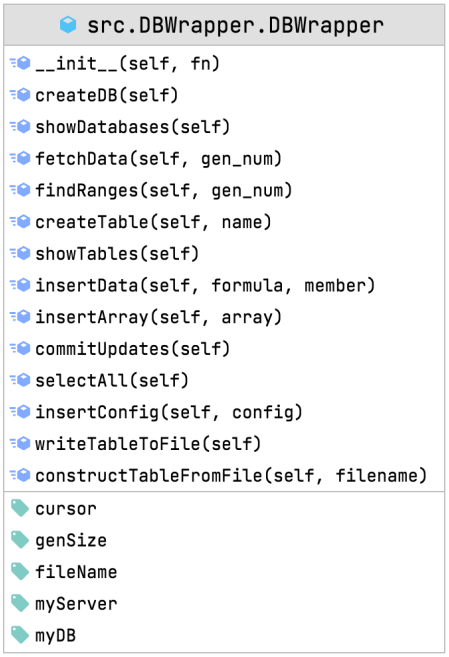
Implementation

Database

The DBWrapper class is used to manage a connection to a MySQL database. Using a database allows us to mitigate certain concurrency issues associated with multithreading, which is required to run hyperopt iterations in parallel.

The DBWrapper class handles everything from connecting to a server, creating a database, creating tables and posting and fetching entries to said table.

The DBWrapper class is used to generate an instance of the MySQL server connection. This instance is then used to handle communication with the database.



Connection to the database is managed through a module named pymysql. The connect method is used to create a connection object to the database by inputting the host, port, database, user and password. We then extracted a cursor object which allows for communication with the database by executing SQL commands.

For insertion commands, we used a “formula” that allows us to substitute a tuple, which holds the configuration objects variables, into the placeholders.

DockerFile

To allow for containerised execution of our program we decided to use docker. We needed to run a MySQL database and decided to implement it inside our container instead of in a separate container to minimise any networking delays. On a larger scale it would be recommended to use a separate container.

Once the docker image is built and run, we run a bash script to launch a mysql server inside the container.

We used pymysql inside our python code to communicate with the server.

PyMoo

PyMoo is the Python multi-objective optimization framework we used to implement our genetic algorithm. Our objective was to minimize an objective function in a multi-dimensional space that requires various classes of PyMoo to function correctly. These include the Problem, Algorithm and Termination objects. We further made use of a Callback object which allowed for simplification of processing various items at the end of a generation since the object is set up to be notified at the end of a generation.

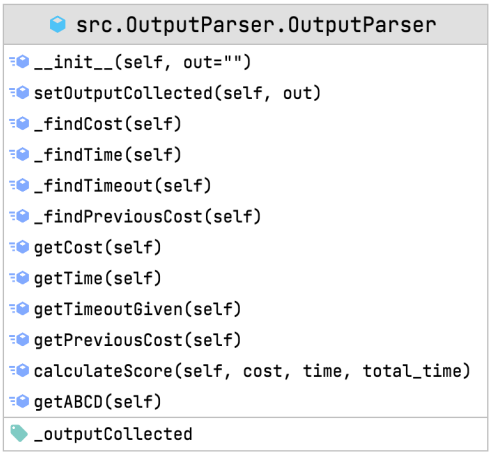
The Termination and Algorithm objects were set up by setting various supplied parameters of the PyMoo classes, however, the Problem class required further work to sufficiently solve our problem since ‘problems’ are traditionally described as mathematical equations. For this reason, the Problem class was inherited by the Custom Problem class which adjusted the features of the Problem class to solve the problem at hand.

Custom Problem was defined to optimise the objective function on the multi-dimensional surface of P1, P2 and P3, where P1 is the end score to be minimized, P2 is the score improvement that should be maximised and P3 is the stuck time which should be minimized. Not that PyMoo only works in terms of minimising and as such, instead of maximizing P2, we seek to minimize -P2.

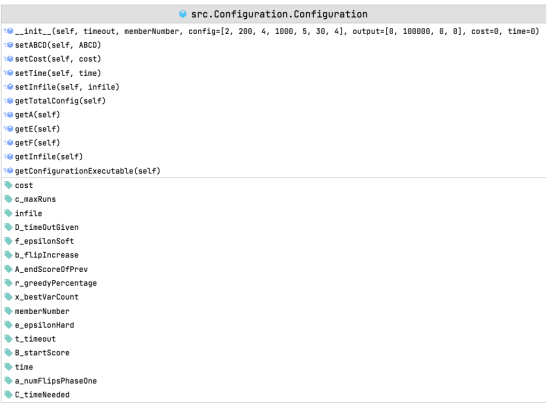
Further, the objective space defined above is not directly supplied. Instead, another class called the Output Parser was required to parse the outputs produced by CarlSAT runs and retrieve the various values to be minimized or stored otherwise.

Output Parser

This class is used to parse the output of CarlSAT to obtain various values such as the cost and time of a particular run, as well as other intermediary values such as the timeout given to a certain run. Some of these values were stored for further analysis, and others were used in the objective space. Note that the objective, score, is a function of the cost and time.

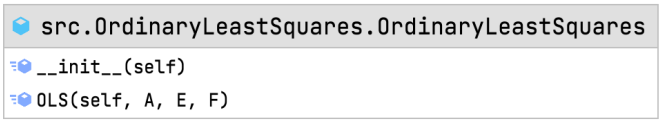


Configuration



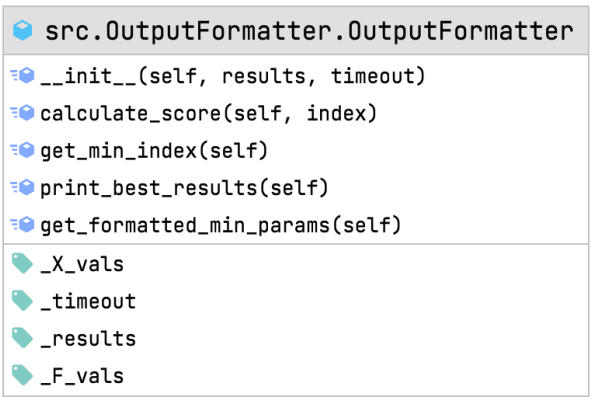
Due to the large number of different parameters to track, a Configuration class was made to handle the storing and retrieval of the various parameter configurations in their appropriate forms. So, for example, this class would store the sampled CarlSAT parameters to be used and then be able to retrieve them in a manner that PyMoo uses in the inherited _evaluate method of the Custom Problem to run CarlSAT with said parameters. After a run of CarlSAT, the further output would be given and the Output Parser is used to extract the values of interest such as the end score of the previous run, starting score, time needed to achieve the starting score, timeout given. Functions of these values are used to obtain the objective space and are stored in the database for further analysis.

Ordinary Least Squares



For the incremental solver of CarlSAT to work, it requires an ‘infile’ - a state file stored which tracks the history of a previous CarlSAT run. CarlSAT uses this to begin processing from where it finished according to the state file. Every run will therefore create a state file and when performing an incremental step, the infile to be used is needed to be known. For this reason, we created a class called Ordinary Least Squares which performs stochastic acceptance in choosing a state file that most closely matches a given set of values related to the objective space. This is used in conjunction with the _evaluate method of the Custom Problem class for an incremental run of CarlSAT to occur with the correct infile choice.

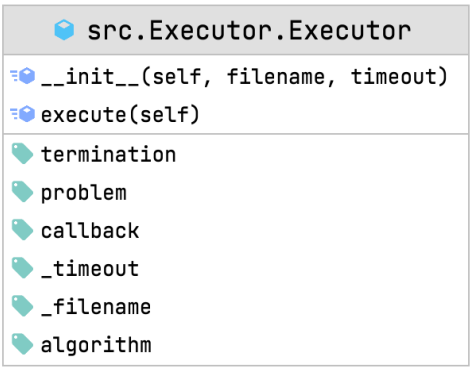
Output Formatter



An Output Formatter class is used to retrieve a set of solutions and choose the set of parameters that should be used in a future run to obtain such a score. The appropriate output values are then formatted into a readable form to standard out. This can of course be redirected to a text file through Unix piping.

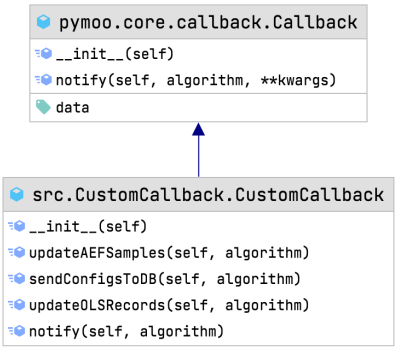
Executor

To follow the idea of the Single Responsibility Principle, we extracted all this processing into their classes and used an Executor class that would handle calling the minimization. This would assist in further scaling at a later point if necessary and ensure that our code has high cohesion and low coupling.



Custom Callback

This is used in the incremental solver to be able to input updated boundaries for parameters as well as associated state files.



Custom Problem

A problem that can be input into the pymoo genetic algorithm with the correctly implemented parameters.

NGSA-II: Non-dominated Sorting Genetic Algorithm

NGSA-II is the genetic algorithm that we chose from the pymoo library to utilise in our multi-objective optimization problem. This is a fundamentally evolutionary algorithm.
A multi-objective optimization problem has the added complexity of finding the optimal combination of objective variables. This could mean increasing x_1 while simultaneously decreasing x_2.

An underlying reason for using a genetic algorithm like NGSA-II as opposed to other gradient descent based algorithms is since these alternatives are inherently sensitive to input values. To elaborate, gradient-based techniques are sensitive to initial values as they are prone to falling into local minima on the objective surface as opposed to finding the truly optimal values.

As has been discussed before, the CarlSAT program may output entirely unique results from extremely similar inputs. This demonstrates a counter-intuitive nature of CarlSAT that requires an algorithm with as little sensitivity to initial values as possible. We want to be able to seek out the truly optimal values in the system.

NGSA-II has the following three main features:

- The best performing members in a generation are carried forward into the next generation.
- It uses a crowding distance.
    - This is a diversity preserving mechanism to ensure that the algorithm retains exploratory habits and doesn't converge before an optimal solution.
    - It requires that solutions are sufficiently 'distanced' from each other.
- It emphasises the non-dominated solutions.

But what are 'non-dominated solutions'

First, we must explore the concept of domination in state space.
A solution x_1 is dominant on x_2 if:

- It is no worse for all objectives
- It is strictly better in at least one objective

Thus, non-dominated solutions are solutions that cannot be outperformed within the generation.
As can be seen, this multi-objective optimisation algorithm is perfectly suited for our requirements in this project. Finally, the algorithm iterates as follows:

1. Sort the list of members in a generation by the level of non-domination.
2. Populate new generation including carrying forward best performers
3. Use crowding distance to ensure diversity in the solutions
4. Creates offspring generation through crowded tournament selection

Presentation Layer

Hyperopt

This is our master class which communicates with the user through the command line. The UI is a basic terminal UI that prints outputs in a formatted matter and requests command line arguments. This class depends on all classes from the Logic Layer to perform the calculations.

CustomProblem

This is used to define the problem that will be fed into carlSAT. This relies on the Configuration object.

Executor

This class is in charge of creating and executing multiple threads of the solver with the relevant algorithm and problem set supplied into each thread. The essential method in this class is the execute method which each thread performs.

OrdinaryLeastSquares

This class is used by the CustomProblem class to decide which past state file to associate with the member of the current generation.

ClassTests

This class has the responsibility of unit testing all the functions in our associated code. This is a vital stage in development where we ensure that all functions perform how they are expected to.

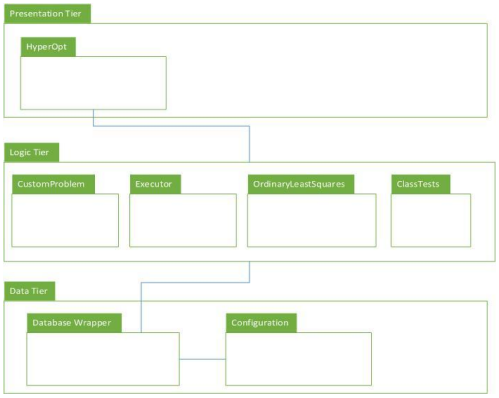
Data Layer

Database Wrapper

This class is used to communicate directly with the database. It has the functionality to read from and write to the database. This is a Data Layer as its main function is to bridge the gap between the database and the logic layer.

Configuration

This class holds the configuration of records that will be entered into the database. All members passing through the system will be ensured to be in this correct format. Thus this is an essential aspect of the data layer.



Program Validation and verification

Quality Management Tests

Our class ClassTests.py is a class whose fundamental function is to unit test all functions coded into our system. These unit tests compare the output of each function with the expected output. If the class has no return type then equivalent checks are performed to ensure the correct behaviour was experienced.

On top of unit tests, we ran integration tests. This is the practice of testing past functions when adding a new function to ensure that the addition of the new method did not negatively affect the performance of previously developed functions.

We performed regression tests on our methods. This consists of ensuring that, not only do our functions work in a vacuum, they also run with correct behaviour when run together.

Unit Testing

We have developed the unit testing class ClassTests.py whose job is to unit test all functions for appropriate behaviour. Methods that should return values are checked against dummy input which should be accepted and return appropriate values, as well as input which should raise an error. The error should be appropriately managed. Note that not all methods need invalid data checking since a large portion of the methods are used internally with their internal processing. However, those methods which had external input were tested with invalid dummy input data.

Integration Testing as we code

Perform unit testing through every addition of class functionality to check if the methods can effectively communicate with each other.

Throwaway branch

All group members performed exploratory research regarding the multiple concepts we had no experience in dealing with (i.e. PyMoo, Docker, SQL with Python, genetic algorithms). This was a useful way for us to realise how the various frameworks and tools we were to use can be implemented and used efficiently.

Version control

We were incredibly careful to branch every new feature to keep control of the working prototype in the master repository. This enabled us to develop features evolutionarily under different branches of the repository and merge them once testing was completed. We used Git to manage our version control and stored the files on UCT's GitLab server.

System testing

Once we were happy with the functionality of our code and believed that our methods were achieving their desired results, we could launch HyperOpt inside a Docker container to test the whole system. This is a valid system testing technique since even though HyperOpt will be running on a server of some sort, the server will only run the containerised Docker application. Therefore, we run the Docker container as a user on our machines and test the program's performance using various test problem .wcards. Security is not of concern for this project as it is not designed to be exposed to such threats, however, performance is key to the functionality of our system. To validate this, we simply ran the Unix command to monitor process calling and could see that indeed our system was maxing out the resources on the tested system. It is likely to perform similarly when launched elsewhere.

Verification

We compared our results to those obtained in the problems GitHub repo and found that we performed similarly well. We are satisfied with our results but further improvements (such as those mentioned in the next prototype release) could further improve the performance of our system. Also, Python has a lot of overhead which could be optimised out with further development.

Table 1: *Summary Testing Plan*

Process	Technique
1. Class Testing: test methods and state behaviour of classes	Random, Partition and White-Box Tests - Most processing occurs internally and as such, we can test with dummy inputs that are expected to be outputted via the internal processing and use these to validate that the units work correctly.
2. Integration Testing: test the interaction of sets of classes	Random and Behavioural Testing - Test that data is correctly passed to the other objects and the objects are used to achieve certain outputs. In general, the processed outputs can be used to validate that the interaction occurred successfully.
3. Validation Testing: test whether customer requirements are satisfied	Use-case based black box and Acceptance tests - The system is a scientific experiment and as such, the client was sent copies of the system to validate that it was performing the logical requirements. A to-do list was also made of the various tasks the system should perform and this list was updated as features were implemented.
4. System Testing: test the behaviour of the system as part of a larger environment	Recovery, security, stress and performance tests - As mentioned above, the application was containerised and therefore any testing of its performance in a larger system could be tested locally. Using the Unix tools to monitor system performance, we could validate that the resources were being used maximally, as a large amount of processing is required to optimise the use of time.

All test output has been appended. Please see the output of this in Appendix 1. Below is a summary of the data sets and reason for their choice used in testing as well as a summary of their performance. Note that due to the architecture of the system, little human input is allowed and thus invalid data tests are often unnecessary since all items to be processed are created internally and will always have a similar form. Any data that does not get processed internally and can affect the internal processing is tested at their input levels and rejected if invalid.

Table 2: *Summary of tests carried out*

Data Set and reason for its choice	**Test Cases**		
	Normal Functioning	*Extreme boundary cases*	*Invalid Data (the program should not crash)*
Launching HyperOpt with various flags passed to test program launch	Appropriate filename, timeout and help flags respond correctly	A single parameter or no parameters passed notifies the user how to use the application	Incorrect parameters passed or invalid parameter choice - the system tells the user what went wrong and quits
Run CarlSAT with the supplied list of parameters to see how CarlSAT performs and its output given	Correct parameters passed, CarlSAT runs and gives its output	Mixing parameters such as the -i with -z chooses the appropriate parameter and processes as required	CarlSAT returns its error or performs with default parameters supplied. Not an issue since CarlSAT is run internally at which point any values passed will be valid
Use a sampled set of values from PyMoo and store them in the Configuration object	Store and retrieval happens correctly	N/A	N/A since data supplied will always be internally sourced
CarlSAT output supplied to Output Parser object and verify that correct outputs are being parsed	Passes	Both outputs (with and without an incremental flag) should successfully be parsed and indeed this is the case	N/A since CarlSAT output is internal
Supply configuration values to DBWrapper to validate that database connection, as well as storage and retrieval of data happens correctly	Passes	Instead of full configs, some further methods are used to get random records which also pass	N/A since internal processing
Supply Ordinary Least Squares (OLS) with records and min and max AEF values to check if appropriate state file to be read in can be found	Passses	N/A due to internal processing	N/A due to internal processing

Table 3: Further, below please see a short list of the type of tests used in our unit testing, as well as their input, behaviour and expected output.

Test	**Input**	**Behaviour**	**Expected Output**
Creating a new database on the MySQL server	Database name (String)	A new database with the provided name should be created in the MySQL server	A database with the provided name should be created and therefore the function should return true. The database will be included retrieving the databases that have been created
Inserting a new table into the database	Table name (String)	A new table should be added to the database with the correct table name	The method should return true that a table has been added to the database and a table with the specified input name should be shown when retrieving the tables in the database
Inserting member values into the database table	Member attributes in tuple (Tuple)	Tuple should be parsed for values and these inserted into the table correctly with no data inconsistencies via the appropriate SQL call	The method should return true and calling to retrieve the member values should obtain the correct values
Correctly parse time and cost from the output	CarlSAT output (String)	Time and cost should be correctly extracted and stored in individuals variables	Time and cost should be the same as the output from CarlSAT
Execute CarlSAT with supplied command-line arguments	Cutoff time, wcard file and configuration string generated by the configuration class combined into a single string for command line arguments (String)	CarlSAT should run without errors and produce output for the given input values	CarlSAT output printed to standard out (terminal)
Read a problem card (wcard)	Problem card in correct format (.wcard or .wcnf)	Check correct file format and attempt to open the file	Problem file successfully opens
Get the parameters from parameter file for specified generation	Parameter file (.txt)	Finds the generation values and extracts the parameter sets for each member of the population	A list of parameter sets for each member of the generation selected
Correctly calculate score from cost and time	Output from OutputParser object	Calculates the score based on our 2D function of cost and time	Correct score values are calculated and assigned to a member

Conclusion

From this project, we have found that the satisfiability solvers consistently outperform their opponents in the world of solving hyper-parameter problems. Moreover, the CarlSAT solver has been shown to outperform similar solvers by as much as 5%. Unfortunately, in our research, we were not able to achieve this 5% increase in performance due to suboptimal algorithms in the fight for thread safety and lack of time in order to achieve prototype 3 which aimed at tackling some of these issues for finding our best performance levels.

Nevertheless, our HyperOpt program which makes use of CarlSAT through the way of PyMoo genetic algorithms has shown exceedingly positive results and is an extremely promising avenue to explore these hyperparameter problems which may have completely independent solutions.

This form of the problem is amazingly suited to genetic computational methods. It may only be solvable through such methods and not by human calculation due to the unintuitive and non-polynomial nature of the relationship between the predictor variables and the response variables, namely, cost and time, as well as the function of these variables.

HyperOpt User Manual

CSC3003S Capstone project for 2021.

HyperOpt is a hyperparameter tuning program using the PyMoo multi-objective optimization framework to aid in calculating appropriate hyperparameter values. This program is intended to work with the CarlSAT satisfiability solver to find the optimal set of hyperparameters for CarlSAT's use.

Authored by Bradley Culligan (CLLBRA005), Patrick Collins (CLLPAT012), Finley Schuiling (SCHFIN001).

How to start running the program

Launch the Dockerfile

Run the following commands in your terminal:

docker build -t hyperopt .

docker run -it hyperopt

Note that in order to obtain the outputted textfile containing the database contents on your host machine, Docker volume mounting should be used. The volume to be mounted should be the /HyperOpt - i.e. the base directory on the Docker container.

Launching HyperOpt

Please perform the following set of commands after running docker run -it hyperopt and you are in the Docker container.

Initiate the database:

chmod +x src/run.sh

./src/run.sh

Run python3 src/HyperOpt.py [inputProblem] [timeout], where inputProblem is a path to a valid wcard problem to be run, and timeout is the timeout (in seconds) per CarlSAT run. Recommended times are 1 second for small problems, 2 seconds for medium problems, and 10 seconds for the large problems. Problems can be found in the problems directory.

Alternatively, run python3 src/HyperOpt.py -h to get the help screen.

An example of launching HyperOpt in the Docker container:

Initiate the database:

chmod +x src/run.sh

./src/run.sh

Then run HyperOPT:

python3 src/HyperOpt.py problems/test-30-26011.wcard 1

What to do After Running HyperOpt

If you're still in the Docker container, run exit to exit from the Docker container and return to your own device. If you want to run another problem card, then simply follow the steps under the HyperOpt launch header.

References

Bontes, J., 2021. *GitHub - JBontes/CarlSAT_2021: CarlSAT, a local search solver for MaxSAT with cardinality support*. [online] GitHub. Available at: <https://github.com/JBontes/CarlSAT_2021> [Accessed 15 August 2021].

Appendix

PyDocs are attached in the doc folder in the project repository that support a navigable html format.