Modifying AutoGrow4 to perform a parallel islands approach to *de novo* drug discovery

Callum-Luis Kindred
Student ID: 2012694
Programme Name: Bsc Artificial Intelligence and Computer Science with Industrial Year
Supervisor: Dr Shan He
Word Count: 5964

Abstract

The process of drug design relies heavily on the discovery of novel chemical entities with desirable biological activity. *De novo* drug design (DNDD) is a field which seeks to computationally generate such molecules. In addition to optimising the primary fitness metric, diversity of the generated molecules is also an important factor. By aiming to increase diversity, it is possible that a greater area of the chemical search space can be explored, leading to the discovery of novel molecules with favourable properties. In addition, it is thought that a more diverse set of candidate molecules will be beneficial to researchers in the Lead Optimisation stage as they will have a wider range of leads to pursue. Genetic algorithms (GAs) have been used extensively to perform DNDD. One type of genetic algorithm known as a parallel islands genetic algorithm, aims to achieve a broader search by maintaining a number of partially isolated sub-populations. This project adapts an existing DNDD GA to create a parallel islands implementation with the objective of improving the diversity of the generated compounds. Multiple runs of the new algorithm are performed and through statistical analysis it is found that the parallel islands implementation produces more diverse solutions than the original genetic algorithm. Additionally, the parallel islands algorithm is able to discover higher scoring molecules as a result of conducting a broader search.

Contents

Table of Program Listings

List of Figures

List of Algorithms

Acknowledgements

I would like to thank my supervisor, Dr Shan He, for his patience and guidance throughout this project and for his financial support in relation to running my code on AWS.

Chapter 1

Introduction

This report investigates the use of a parallel genetic algorithm implementation as a means of increasing diversity of the generated solutions, in the domain of *de novo* drug design. As part of this project, such an implementation was developed and compared against an existing *de novo* drug design genetic algorithm.

The following section aims to serve as an introduction to the topics of computational drug discovery and parallel genetic algorithms.

1.1 Computational drug discovery

Drug design is an expensive and time consuming process, with the majority of drugs failing to reach the market (Mohs and Greig 2017). In the modern era, designing a new drug is a multi-disciplinary effort and Computer Aided Drug Design (CADD) often plays a key part in reducing costs and accelerating the drug design process (*Computer Aided Drug Design (CADD): From Ligand-Based Methods to Structure-Based Approaches* 2022). Specifically, CADD is the field of study concerned with computationally generating chemical structures with desirable properties. One approach to CADD is *de novo* drug discovery (DNDD), referring to a computational method of generating novel molecules from scratch. DNDD is promising as it has been shown to be capable of generating diverse and novel structures, expanding our knowledge of the potential chemical search space which is believed to be largely unexplored (Mouchlis et al. 2021).

1.2 The role of DNDD in the drug discovery pipeline

DNDD has been extensively employed to carry out the task of Lead Discovery which is the stage in the drug discovery pipeline concerned with finding small drug-like molecules. These molecules are subsequently progressed to preclinical

testing and then, if successful, clinical trials (Hughes et al. 2011). The majority of these candidates are ultimately unsuccessful at later stages, resulting in great financial losses on behalf of pharmaceutical companies. Therefore, any incremental gains that can be made in the quality of the predicted ligands during lead discovery can be valuable if it means that they are likely to progress further in the pipeline. One such way to increase the potential value at this stage would be to produce a more structurally diverse set of solutions. It has been documented through the field of quantitative structure-activity relationships (QSAR) modelling that different molecular structures give rise to different bioactive properties. Therefore, a set of molecules with a greater range of structures and hence properties may prove to be beneficial in the preclinical stage.

1.3 Genetic algorithms

Genetic algorithms (GAs) belong to the class of algorithms that are inspired by the process of natural evolution (known as evolutionary algorithms). GAs simulate the process of evolution on a 'population' of solutions (encoded as artificial chromosomes) through a series of steps such as selection, crossover and mutation. GAs are a popular and well-established method for DNDD (Devi, Sathya, and Coumar 2015; Mouchlis et al. 2021).

1.4 Parallel computing

Parallel computing refers to the practice of breaking down a large problem into smaller tasks and processing these tasks simultaneously. Genetic algorithms can be parallelised, leading to a decrease in computation time (Cantu-Paz 1999). Such parallel genetic algorithms (PGAs) can be organised into a number of sub-categories (Harada and Alba 2020).

The Parallel Island PGA works in a similar way to the standard GA but it maintains multiple separate and partially isolated sub-populations known as 'islands'. Islands are processed in parallel and the top solutions are shared between islands through a process of 'migrations'. As populations are processed in parallel, computation time can potentially be reduced. Additionally, this approach is thought to preserve diversity due to the high degree of isolation between the multiple populations.

1.5 Key limitations of *de novo* drug design

The primary issue with DNDD is that the synthetic feasibility of the generated molecules cannot be guaranteed; the solutions generated *in silico* may not be producible *in vitro*. Another issue is that there are many variables to be optimised for when designing a new drug, such as: biological activity against a given target, toxicity, synthetic accessibility, 'drug-likeness' and cost to manu-

facture. Therefore, the researcher is faced with a challenge when deciding which properties to optimise.

1.6 Limitations of genetic algorithms

Owing to their sequential nature of performing iterative operations over many generations, genetic algorithms often suffer from long computation times. Additionally, genetic algorithms can suffer from premature convergence, meaning that optimal solutions may go undiscovered. As with all search methods that aim to minimise an objective, the search is 'narrowed' as the algorithm attempts to move toward the optimum, meaning that large portions of the search space go unexplored.

1.7 Project aims

This project aims to adapt an existing genetic algorithm to create a parallel islands implementation in an attempt to improve the diversity of the generated solutions.

Chapter 2

Literature review

This literature review provides an overview of *de novo* drug design techniques, including parallel and non-parallel genetic algorithm methods as well as other methods.

2.1 Categorisation of *de novo* drug design techniques

Mouchlis et al. (2021, p.2) define *de novo* drug design as 'a methodology that creates novel chemical entities based only on the information regarding a biological target (receptor) [structure-based design] or its known active binders [ligand-based design]'. Structure-based design is used to target a specific known receptor. The generated solutions are then evaluated using a scoring function that predicts the binding affinity between the molecule and the target receptor. It should be mentioned that, despite their popularity, there are some limitations of using docking functions to estimate binding affinity, as highlighted by Pantsar and Poso (2018). When the three-dimensional structure of the target site is unknown, ligand-based design serves as an alternative approach whereby known active binders of the target site are used to inform the search for new molecules, finding similar ligands which are likely to have similar properties.

De novo drug design techniques can be classified at three levels of granularity: atom-based, fragment-based and reaction-based (Meyers, Fabian, and Brown 2021). These correspond to the levels at which the chemicals are represented within the algorithm (i.e., at the atom level, the level of the molecular fragment or in terms of reactions between chemical entities). In general, techniques that use a higher level of granularity will produce more diverse results as they are able to search a greater area of the chemical space. Conversely, By using larger 'building blocks', low granularity approaches are able to converge quicker but as a result the solutions tend to be less diverse.

2.2 Non-genetic algorithm *de novo* drug design

Various machine learning methods have been applied to the task of *de novo* drug design, with recent breakthroughs in deep learning paving the way. Olivecrona et al. (2017) fine-tuned a recurrent neural network with reinforcement learning to guide it towards producing molecules with certain desirable properties. Li, Zhang, and Liu (2018) investigated the use of graph-based models whereby molecules are represented internally as graphs instead of the commonly used string representation. Bagal et al. (2021) applied the prominent Transfomer architecture to DNDD and found that its ability to model long-term dependencies meant that it lent itself well to this problem. In addition to this, some other non-machine learning algorithms such as particle swarm optimisation (Hartenfeller et al. 2008) have been proposed.

2.3 Categorisation of parallel genetic algorithms

Belonging to the class of soft computing techniques, evolutionary algorithms lend themselves well to multi-objective optimisation problems such as *de novo* drug design and, therefore, have been used extensively in this field (Devi, Sathya, and Coumar 2015). Genetic algorithms operate on a set or 'population' of bit vectors referred to as 'chromosomes', iteratively performing steps such as mutation and crossover and selecting the best solutions based on a 'fitness' metric.

Genetic algorithms lend themselves to a number of parallel implementations, due to their population-based approach. The simplest implementation is known as 'global' parallelisation (Harada and Alba 2020). As with the standard genetic algorithm, a single central population is maintained with the only difference being that the solutions are evaluated in parallel. This model is often implemented as a master-slave architecture. Additional operations such as mutation or any domain specific operations may also be executed by a slave node.

The 'cellular' model works by arranging the solutions in 'grids' where operations are applied to the solutions based on their neighbourhood within the grid. This model is commonly realised in a configuration whereby each grid is assigned to its own CPU, with message passing employed to operate on the neighbourhoods that span across multiple grids.

The 'island' model maintains multiple partially isolated sub-populations (or 'islands'), executing the genetic algorithm procedure on each sub-population in parallel. The islands evolve largely in isolation, with the only communication between islands occurring in the form of 'migrations' where a set number of solutions are broadcast between islands at a given interval. This is thought to conduct a better search due to the fact that the independent islands explore different regions of the search space while still sharing information about the top solutions (Whitley, Rana, and Heckendorn 1999). With this comes several parameters that must be configured such as migration frequency and migration selection/replacement strategies. Typically, each island is assigned to its own

CPU and some kind of messaging protocol is used to carry out migrations.

2.4 Genetic algorithm *de novo* drug design

The following section outlines several DNDD genetic algorithms, highlighting the key differences, in hopes of motivating the present work.

GANDI (Dey and Caflisch 2008) uses a genetic algorithm to perform fragment-based drug discovery. Solutions are evaluated based on both binding affinity and on similarity to known inhibitors, meaning that it can be considered a combination of structure-based and ligand-based approaches. ADAPT (Pegg, Haresco, and Kuntz 2001) was an early example of a structure-based method utilising the practice of reintroducing diversity into the population as a means of avoiding local minima. JANUS (Nigam, Pollice, and Aspuru-Guzik 2022) takes another approach to preserving diversity by maintaining two separate populations, one for exploitation of the search space and one for exploration. Whilst these fragment-based algorithms are capable of producing high quality solutions, they often suffer from poor synthetic feasibility (Mouchlis et al. 2021; Nigam, Pollice, and Aspuru-Guzik 2022). Reaction-based methods purport to tackle this issue by using genetic operators that correspond to sound chemical reactions.

AutoGrow4 (Spiegel and Durrant 2020) uses reaction-based mutation steps to perform structure-based DNDD. It combats the issue of reaction-based methods suffering from low diversity by progressing a number of solutions to the next generation solely by virtue of their structural novelty. Whilst this measure is effective at preserving diversity, I hypothesise that through the use of multiple populations a broader search could be achieved.

Following from this review of the literature, this project aims to implement a novel parallel genetic algorithm and answer the question: "Will a parallel islands implementation of AutoGrow4 produce more diverse solutions than the original?". Additionally, this project will investigate the effect that the parallel islands model will have on the binding affinity of the generated solutions.

Chapter 3

Overview of AutoGrow4 and RabbitMQ

3.1 AutoGrow4

This section describes the AutoGrow4 genetic algorithm developed by Spiegel and Durrant (2020), designed to perform computer aided drug discovery. It uses a structure-based design approach to iteratively evolve novel drug-like compounds by applying a set of genetic operators. It can also be used to optimise pre-existing ligands in a technique known as lead optimisation, however this technique is not the focus of the present study.

3.1.1 Population generation

At the start of execution, the AutoGrow genetic algorithm obtains the starting population from a user-specified source compound file, typically consisting of chemically diverse molecular fragments. Subsequent generations are produced by taking the previous generation and applying the elitism, mutation and crossover operators.

Seed selection

For each operator, a subset of 'seed' solutions is taken from the previous generation and the respective operation is applied. Seeds can be selected using roulette, ranking, or tournament selection, with ranking being used by Spiegel and Durrant (2020) in their experiments. Additionally, for each seed pool, some seeds are selected based on molecule fitness whilst others are selected based on molecule diversity.

Elitism

Elistim progresses a pre-determined number of the top-scoring seed molecules to the next generation with no modification

Mutation

The mutation operator performs an *in silico* reaction between the molecule to be mutated and a complimentary molecule, using the RDKit Python library. It attempts to use different combinations of reactions and reactants until a chemically sound reaction can be performed.

Crossover

The crossover operator merges two compounds from the previous (parent) generation to produce a new compound in the current (child) generation. The operator uses RDKit to find the largest common substructures between the two parent molecules and then takes this substructure and combines it with a random combination of the parent's connected substructures.

3.1.2 Molecule evaluation

Binding affinity

The primary fitness metric used is binding affinity. This is a measure of how well the generated molecule binds to a user-specified target receptor. AutoGrow can be configured to use a number of different docking programs, with QuickVina2 being used by Spiegel and Durrant (2020) in their experiments. The binding energy of the top-scoring pose is given in kcal/mol, with negative binding energy representing stronger binding affinity and therefore a better molecule.

By default, AutoGrow makes use of a kind of global parallelisation, using QuickVina2 to evaluate solutions in parallel.

Diversity

In addition to selecting molecules based on their binding affinity, molecules are also selected based on their diversity. This incentivises the generation of structurally unique molecules and aims to delay convergence of the algorithm.

In order to calculate the diversity of a molecule, AutoGrow first calculates the Morgan fingerprint (Morgan 1965) of each molecule in the population. The Dice similarity (Dice 1945; Sørensen 1948) is used to measure the structural similarity between two molecules, defined as:

$$s(F_A, F_B) = \frac{2|F_A \cap F_B|}{|F_A| + |F_B|}$$

where F_A and F_B are the fingerprints of mol_A and mol_B, respectively. The value of s ranges from 0.0 (completely different) to 1.0 (identical). The diversity

score d of a given molecule mol_M measures its uniqueness relative to the other molecules in its generation. The score is calculated by:

$$d(mol_m) = \sum_{N \neq M}^{n} s(F_A, F_B)$$

A lower diversity score therefore represents a molecule that is unique with respect to its population.

3.2 RabbitMQ

RabbitMQ is an open-source message broker that implements the Advanced Message Passing Protocol (AMPQ). The RabbitMQ server can be easily deployed via a Docker environment. The messaging architecture makes use of several components. A *producer* is simply a program that sends messages. A *queue* is a first-in-first-out data structure used to store messages. Messages are pushed to and read from queues. A *consumer* is a program that waits to receive or consume messages. A program can be both a producer and a consumer. Messages can never be sent directly from the producer to the queue. For this purpose, an *exchange* must be used. Exchanges can simply forward a message directly to a queue or it can broadcast a message to multiple queues (in the case of the *fanout* exchange).

Chapter 4

Approach

4.1 Design and implementation

This chapter describes the design and implementation of the AutoGrow4 parallel islands system.

4.1.1 Instantiation of multiple 'island' instances

At the start of development, the AutoGrow4 repository was forked. The provided Docker environment was used for development as it allows a high degree of portability accross platforms. In order to adapt the existing AutoGrow4 code to run multiple instances of the main execution loop simultaneously, AutoGrow's `RunAutogrow.py` was modified to include code that spawns n processes, each corresponding to an island executing AutoGrow's main loop: `main_execute`, where n is the number of islands specified by the user (see Listing 4.1).

To allow the user to specify the number of islands to use, a new command line parameter was defined in `RunAutogrow.py` (see Listing 4.2). When the program is first started, the value for this parameter is added to AutoGrow's `vars` dictionary and the value is then retrieved in `RunAutogrow.py` when creating the parallel instances.

```
    processes = []
    for i in range(vars["number_of_islands"]):
        process = multiprocessing.Process(target=
    AutogrowMainExecute.main_execute, args=[vars, i])
        process.start()
        processes.append(process)

    # Wait for the launches to complete.
    [process.join() for process in processes]
```

Listing 4.1: An excerpt from `RunAutogrow.py`. A number of processes are created corresponding to the value of `number_of_islands`.

```
#number of islands to be used for parallel islands mode
PARSER.add_argument(
    "--number_of_islands",
    type=int,
    default=1,
    help="Number of islands to use for parallel islands mode"
)
```

Listing 4.2: An excerpt from `RunAutogrow.py`. `PARSER.add_argument()` is used to add `number_of_islands` to the list of possible command line arguments.

4.1.2 Implementation of 'migrations'

The second key component of the parallel islands model is the 'migration' of solutions across islands. In order to facilitate these migrations, a system using RabbitMQ was devised whereby islands transmit and receive messages to and from the RabbitMQ server in a parallel and asynchronous manner. This way, the islands never communicate directly with each other, ensuring a loose coupling between processes. Additionally, this design can potentially be deployed in a number of configurations. For example: all islands and the RabbitMQ server could be hosted on a single node or each island could be hosted on its own node and the RabbitMQ server on a separate master node.

At the end of each AutoGrow generation, each island publishes its top solution to its associated fanout exchange. The fanout exchange then pushes this solution to the ingestion queues corresponding to all the other islands. After an island has published its migration, it is then ready to receive solutions from its ingestion queue. It waits until it has received the appropriate number of solutions (`number_of_islands`-1) before continuing to the next generation. See Figure 4.1 for an illustration.

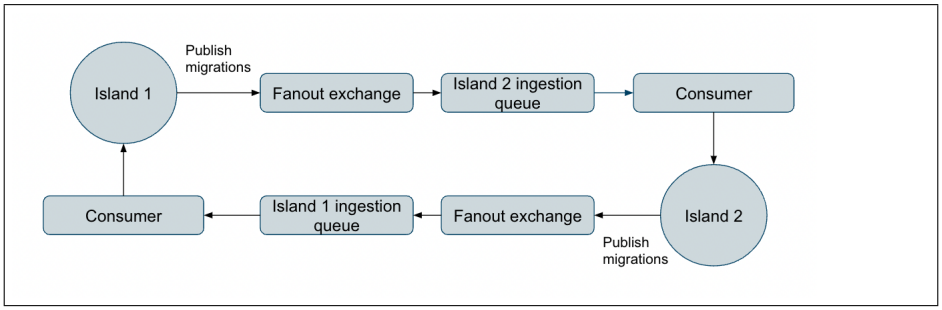

Figure 4.1: Diagram showing data flow between two islands

The Pika Python library was chosen to facilitate client-side connection to the RabbitMQ server due to its ease of use and strong documentation. The `MessagingOperations` class was created to enable the necessary message passing operations. A key method in this class is `setup_messaging()` as it constructs the communication infrastructure. At the start of execution, each island

makes a call to `setup_messaging()`, resulting in the creation of the ingestion queue for each island (see Listing 4.3).

```
        def setup_messaging(self):
        """
        Setup the queues and exchanges.

        Returns:
        :returns: BlockingChannel channel: The pika channel that
    was created.
        """
        exchange_name = "exchange_"+str(self.island_number)
        self.channel.exchange_declare(exchange=exchange_name,
        exchange_type="fanout")

        self.queue_name = "ingestion_queue_"+str(self.island_number
    )
        self.channel.queue_declare(queue=self.queue_name)

        # Bind this islands ingestion queue to all other exchanges
        for i in range(self.vars["number_of_islands"]):
            if i != self.island_number:
                external_exchange_name = "exchange_"+str(i)
                # N.B. This is an idempotent operation so we are
    declaring an new exchange only if it does not already exist
                self.channel.exchange_declare(exchange=
    external_exchange_name,
                exchange_type="fanout")

                self.channel.queue_bind(queue=self.queue_name,
    exchange=external_exchange_name)


        return self.channel
```

Listing 4.3: An excerpt from `messaging_operations.py`. This method creates an ingestion queue (for the island calling the method) configured to receive migrations from all other islands.

When transmitting and receiving Python objects to and from the RabbitMQ server, they must serialised and deserialised respectively. The built-in Pickle Python library was chosen for this task as it is specifically designed for use with Python, meaning that it supports the serialisation of all Python data types. It should be noted that whilst it is possible that Pickle may be exploited via attacks that aim to execute arbitrary code during deserialisation (*Python documentation* 2023), this was not seen as a risk for this project as all data received will be from a trusted source (the RabbitMQ server). In some circumstances, JSON provides advantages over Pickle such as human-readability and cross-language support, however neither of these properties were viewed as beneficial for the present use case.

In order to publish the migrations, code was added towards the end of the main execution loop in `autogrow_main_execute.py` (Listing 4.4). The top scoring ligand is fetched from `ranked_smiles_list`, then serialised using `pickle.dumps()` and then published to the exchange.

```
        ligand_to_migrate = ranked_smiles_list[0]
        print(str(island_number) + " Publishing ligand: " +
    ligand_to_migrate[0])
        messaging_operations_object.get_channel().basic_publish(
    exchange="exchange_"+str(island_number), routing_key='', body=
    pickle.dumps(ligand_to_migrate))
        print("Published ligand")
```

Listing 4.4: An excerpt from `autogrow_main_execute.py`. The top scoring ligand from the current generation is published to the exchange.

A consumer is the mechanism by which messages are received from queues in RabbitMQ. The method `create_consumer()` was added to assist creation of a new consumer (see Listing 4.5). This is a wrapper around the Pika `channel.basic_consume()` method, which is used to initiate the consuming of messages.

```
    def create_consumer(self):
        self.channel.basic_consume(
            self.queue_name,
            on_message_callback=lambda channel,
            method_frame, header_frame,
            body: autogrow_main_execute.on_consume_migration(
    channel, method_frame, body, self.vars["number_of_islands"],
    self.island_number))
```

Listing 4.5: An excerpt from `messaging_operations.py`

The `on_message_callback` parameter is used to specify the callback function that should be called each time a message is received. Here, the specified function is `autogrow_main_execute.consume_migration()`. This method first uses `pickle.loads()` to deserialise the received solution, then appends it to a list (`received_ligands`) and stops consuming once the expected number of solutions have been fetched (see Listing 4.6). N.B. This function is located inside `autogrow_main_execute.py` as it needs direct access to the `received_ligands` variable.

```
    def on_consume_migration(channel, method_frame, body,
    number_of_islands, island_number):
    received_ligand = pickle.loads(body)
    print(str(island_number) + " Received migration: " + str(
    received_ligand))

    received_ligands.append(received_ligand)

    # If all migrations received
    if (len(received_ligands) == number_of_islands-1):
        channel.basic_ack(delivery_tag=method_frame.delivery_tag)
        channel.stop_consuming()
    else:
        channel.basic_ack(delivery_tag=method_frame.delivery_tag)
```

Listing 4.6: An excerpt from `autogrow_main_execute.py`.

With the necessary framework now in place, code was included in `autogrow_main_execute.py`, immediately after the code for publishing migrations, (see Listing 4.7) to receive the migrations. The process is initiated via the call to

`start_consuming()`. This is a blocking call, meaning that `autogrow_main_execute.py` is halted until all migrations have been received. Each time the client receives a message, it is handled by `autogrow_main_execute.on_consume_migration()` and it is added to the `received_ligands` list. Once all solutions have been received, execution resumes and the received ligands are added to the current generation.

```
        messaging_operations_object.create_consumer()

        #Start consuming migrations
        #Blocks until all migrations have been received
        print("Start consuming migrations")

        messaging_operations_object.get_channel().start_consuming()
        print("Finished consuming migrations")
        print(str(island_number) + "received_ligands: " + str(
    received_ligands))

        # Insert migrated ligands into existing list of ligands for
     this generation
        for ligand in received_ligands:
            print(str(island_number) + "Inserting ligand: " +
    ligand[0])
            insort_right(ranked_smiles_list, ligand, key=lambda x:
    float(x[-2]))
```

Listing 4.7: An excerpt from `autogrow_main_execute.py`. N.B. A new consumer is created every generation because the only way to stop the consumer is to destroy it.

4.2 Experimental method

This section describes the experiments used to evaluate the parallel islands implementation against the vanilla AutoGrow4 implementation.

4.2.1 Performing multiple runs

Due to the stochastic nature of genetic algorithms, multiple runs are needed to provide a fair comparison between the parallel islands (PI) implementation and the AutoGrow (AG) implementation. The number of runs for each implementation was selected as 30 as this is the number of samples at which the Central Limit Theorem is typically considered to hold and it is generally adequate for performing significance tests in most cases (Canavos 1984).

4.2.2 Choice of parameters

AutoGrow has a substantial number of user-specified run parameters such as the number of mutations and crossovers to be performed per generation, the number of generations and the path for the source compound file. For the present set of experiments, the parameters used were chosen to closely match those of the

'large-scale run' described in Spiegel and Durrant (2020) (see Appendix 1 for full list of parameter values). The only changes to these parameters that were made were the number of mutations and crossovers per generation as well as the number of generations, in order to reduce computation time. The value for the number of mutations and number of crossovers was selected as 500 per generation as this was found to be the minimum value that could be used while still producing valid results. After completing a number of experimental runs with varying numbers of generations and extrapolating the time that one run took to two sets of 30 runs, it was decided that 15 generations per run would be the maximum number that would be feasible in the given time frame. In addition to the standard AutoGrow parameters, the `number_of_islands` parameter was added. For simplicity and due to time constraints, only two islands were used.

4.2.3 Performing runs on Amazon Web Services

In order to complete the two sets of 30 runs in a feasible time frame, Amazon Web Services (AWS) was used. Six c6a.2xlarge EC2 Instances were selected for the task, striking a balance between manageability, computation power and cost. The runs were distributed across the six instances, with a Bash script used on each instance to automate the execution of the runs.

Chapter 5

Results and discussion

5.1 Results

This section details the results of the AWS runs, including significance tests that compare the vanilla AutoGrow4 (AG) results with the parallel islands (PI) results. Calculations were performed using a Jupyter notebook which can be found in the `Jupyter` directory of the code submitted for this project.

5.1.1 Diversity

For AG, the diversity scores as calculated by AutoGrow4 were used. For PI, the scores were recalculated (after execution had completed) so that the diversity score for each molecule was calculated with respect to all other molecules across both islands. This was done to align with the end-user's point of view where they would effectively treat the outputs of the two separate island populations as one single output.

Upon completion of the two sets of 30 runs, the mean diversity score for the final generation of each run was calculated (see Algorithm 1, Figure 5.1).

Algorithm 1 Pseudo-code demonstrating calculation of diversity/binding means

```
diversity_means = [ ]
binding_means = [ ]
for run in runs do
  diversity_sum = 0
  binding_sum = 0
  for molecule in last_generation do
    diversity_sum += molecule[diversity]
    binding_sum += molecule[binding]
  end for
  diversity_mean_for_run = diversity_sum/len(last_generation)
  binding_mean_for_run = binding_sum/len(last_generation)
  diversity_means.append(diversity_mean_for_run)
  binding_means.append(binding_mean_for_run)
end for
```

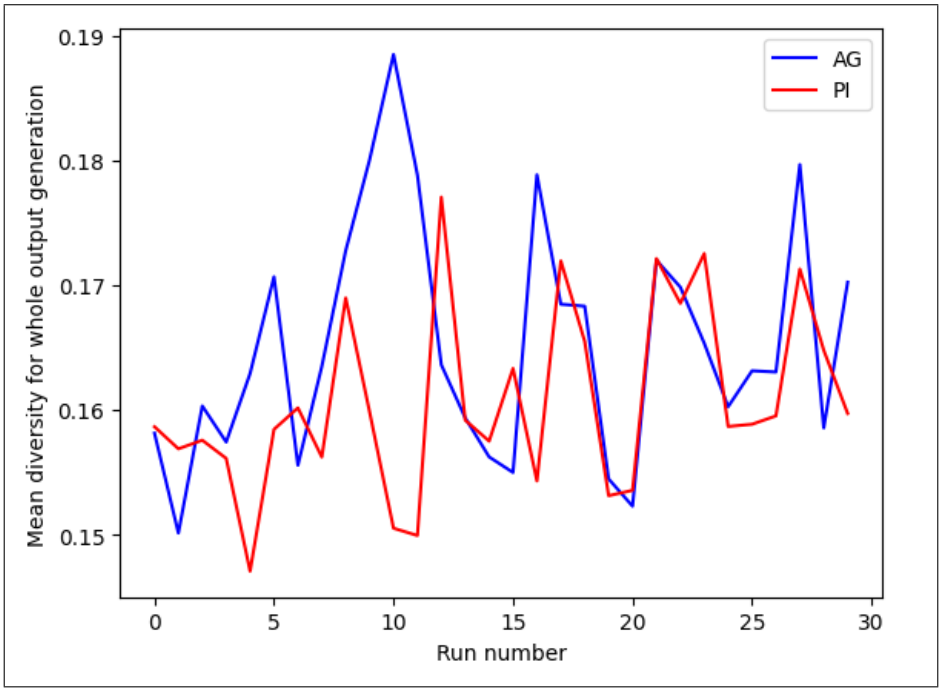

Figure 5.1: Graph showing mean diversity of final generation for each run.

In order to make sense of this data, a one-tailed Wilcoxon Rank Sums Test, making use of these means, was formulated in the following way to assess whether or not it could be said, on statistical grounds, that the diversity scores of the solutions produced by the PI implementation were significantly

lower than the scores produced by AG. It is possible that an Independent T-test could have been used for this task but the Wilcoxon Rank Sums Test was used as it makes no assumptions about the test data being normally distributed or having equal variances.

Hypotheses. H_0 : *There is no difference between the diversity of the molecules produced by the two algorithms.*

H_1 : *The molecules produced PI are more diverse than the molecules produced by AG.*

$$\textit{Significance level } \alpha = 0.05$$

This resulted in a p-value of 0.0380 (4 d.p.), meaning at the 5% significance level, there is sufficient evidence to support the claim that the molecules produced by PI are more diverse than those produced by AG.

In addition to calculating the overall means for each run, the means were also calculated over various percentages of the top solutions per run. This was done in an attempt to understand the effect of the parallel islands implementation on the diversity of a portion of the top solutions, in comparison to the diversity across the whole population. This decision was justified by the idea that it may be beneficial to consider the top solutions as it was speculated that during the Lead Optimisation stage, due to practical constraints, researchers will be interested in investigating a small subset of the highest scoring or most diverse molecules. Due to the fact that the PI population is made up of the two combined island populations, it is twice the size of the AG population. Therefore percentages were used to allow for a fair comparison.

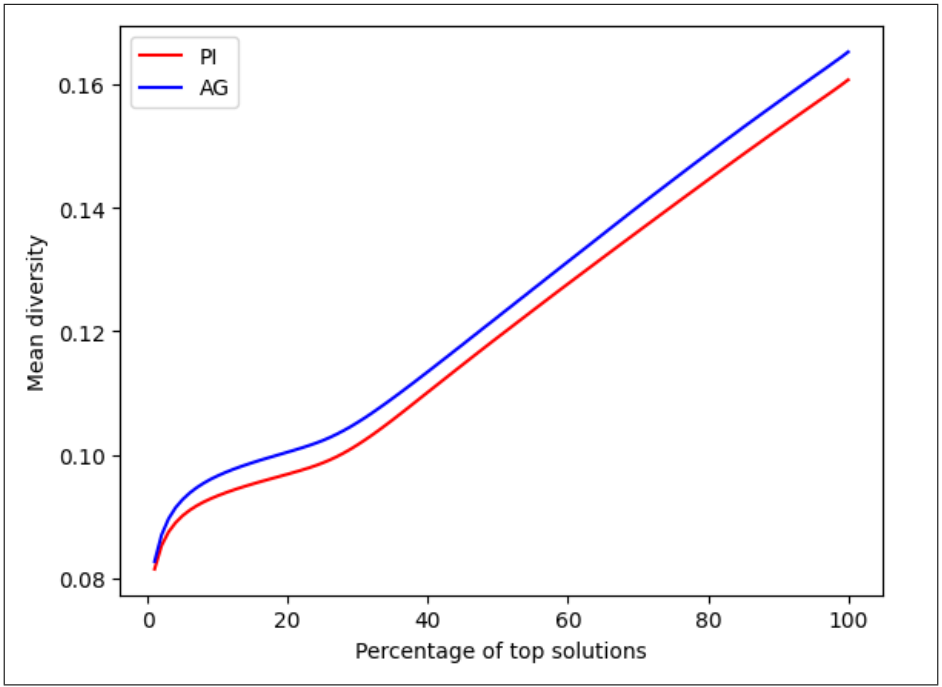

Figure 5.2: Graph showing mean diversity of the last generation across all runs, over a percentage of the most diverse solutions, ranging from 1% to 100%.

As demonstrated by Figure 5.2, the mean diversity of PI is substantially lower compared to AG (here the reader is reminded that a smaller diversity score indicates a higher degree of diversity). Only when considering a very small number of the top solutions (around the top 1%) is there less of a difference between the means, likely due to the sample size being too small to be able to capture the difference in diversity. This effect was further explored by performing the Wilcoxon Rank Sums Test again but this time, calculating the mean for each run over various percentages of the top solutions for that run (see Figure 5.3).

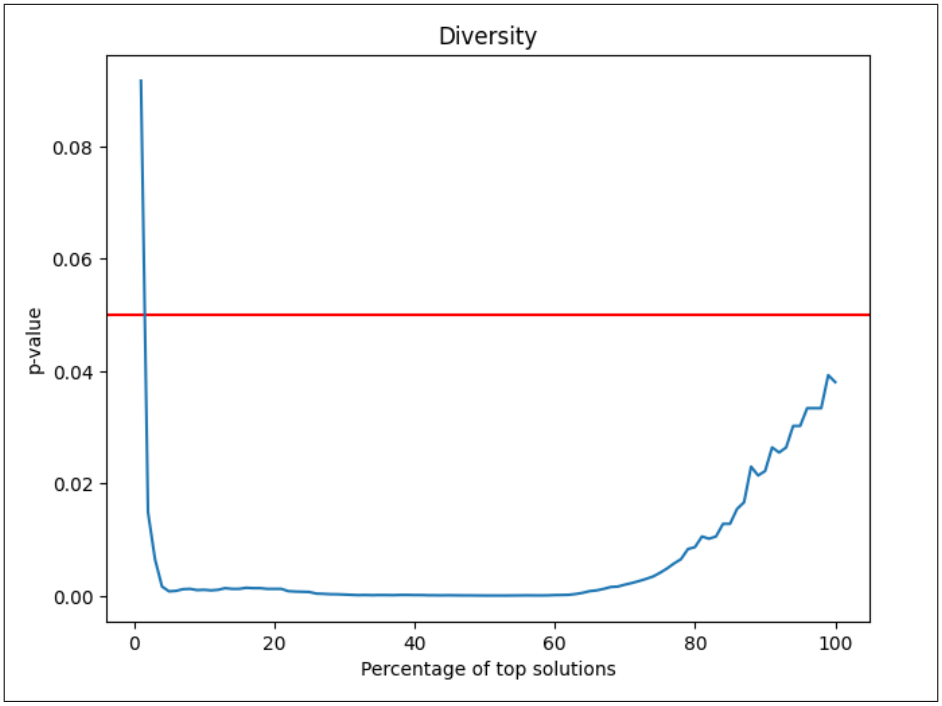

Figure 5.3: Graph showing Wilcoxon Rank Sums Test p-value vs. percentage of most diverse solutions.

Upon inspecting Figure 5.3, it becomes clear that when considering a small portion of the top solutions from each run ($\lessapprox 3\%$), the p-value is greater than the significance level, meaning that there is insufficient evidence, at the 5% significance level, to support that claim that the solutions produced by PI are more diverse than those produced by AG. When considering the top approximately 5% to the top 60%, however, the p-value is well below the significance level, meaning that, at the 5% significance level there is sufficient evidence to support the claim that the molecules produced by PI are more diverse than those produced by AG. When considering more than the top $\sim 60\%$, the p-value begins to increase, although it still remains below the significance level, meaning that there is still sufficient evidence to accept the alternative hypothesis. This increasing p-value would suggest that PI may 'boost' the diversity of the solutions by a percentage amount, meaning that there is a greater difference between the most diverse PI solutions and the most diverse AG solutions than there is between the lower diversity PI and lower diversity AG solutions. Whilst these significance tests provide some certainty that the observed difference between the two groups was not due to chance, they do not provide any information about the magnitude of differences between the diversity scores of the two sets of molecules. Therefore, Cohen's d effect size measure was used in an attempt to quantify this difference (see Figure 5.4).

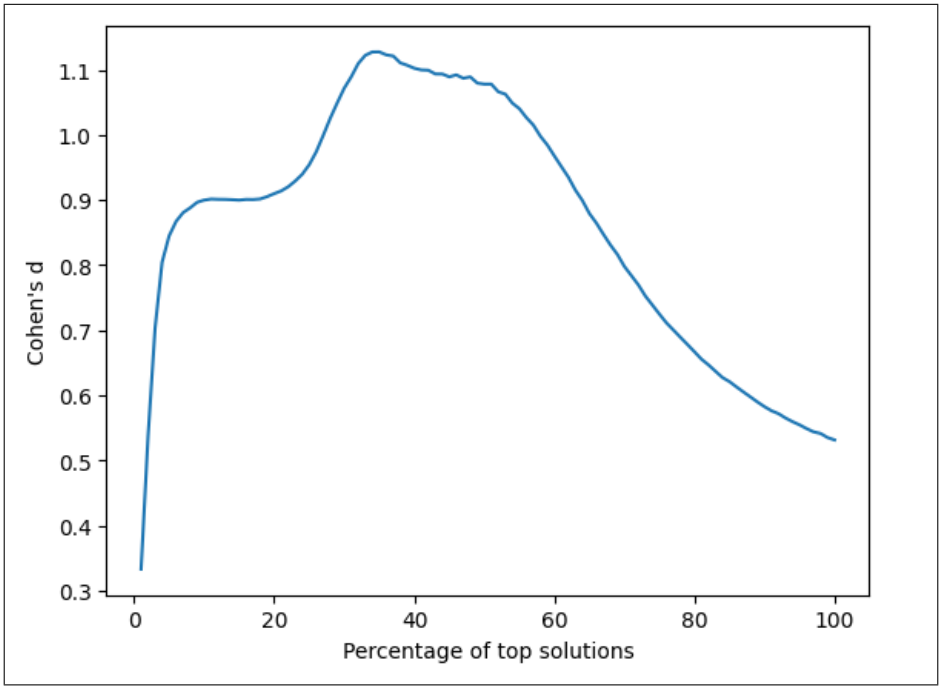

Figure 5.4: Graph showing Cohen's d vs. percentage of most diverse solutions.

Figure 5.4 provides a visualisation of the effect of taking the average over a range of percentages of the most diverse solutions on the magnitude of the difference between the two sets of solutions. This supports the notion that the greatest difference between the two sets of solutions, in terms of diversity, occurs when considering around the top 30% to the top 60% of solutions. The Cohen's d values range from 0.33 to 1.13 (2 d.p.) indicating a 'small' effect size and a 'large' effect size respectively (Cohen 1988). This result can be interpreted by calculating the probability of superiority (also known as common language effect size).

$$CL_d = \frac{1}{1 + e^{-1.7\frac{d}{\sqrt{2}}}}$$

For example, when Cohen's d is 0.33, there is a 59.2% chance that a molecule drawn at random from PI will be more diverse than a random molecule from AG. When Cohen's d is 1.13, this value is 78.8%. From these calculations, it is clear that deciding which percentage of the top solutions to use is an important consideration and the end-user may wish to experiment with different percentages of top solutions depending on the particular use case.

5.1.2 Binding affinity

In addition to diversity, the effect on binding affinity was also investigated. Whilst the aim of this project was to improve the diversity, it was also important that the binding affinity of the solutions was not negatively impacted. Two lists, each consisting of the mean binding score for the last generation of each run were calculated (see Figure 5.5, Figure 5.6) and the Wilcoxon Rank Sums Test was calculated (see Figure 5.7) in the manner described in the above section.

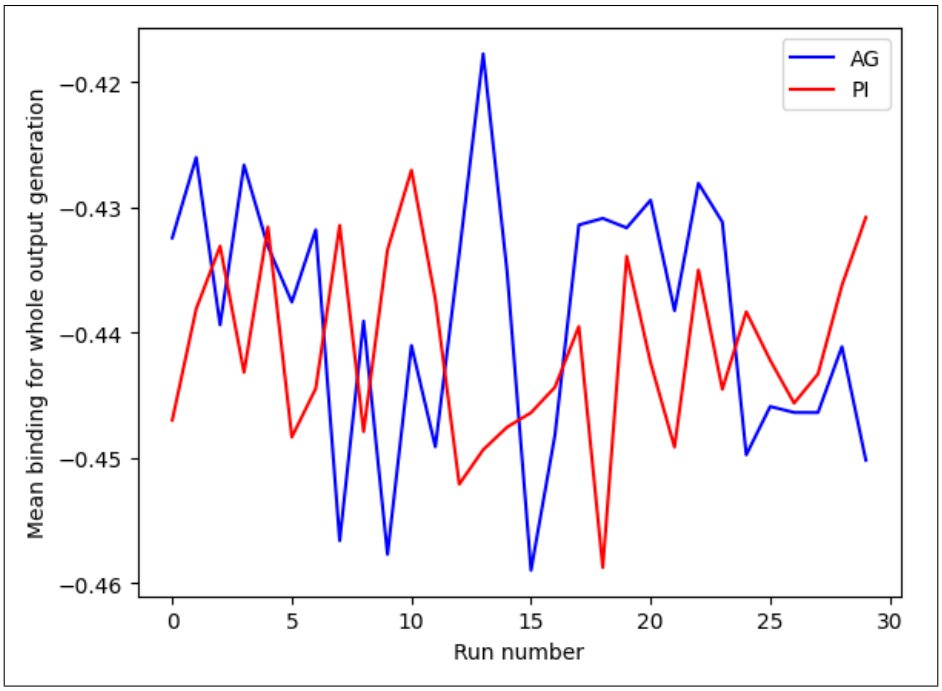

Figure 5.5: Graph showing mean binding affinity of final generation for each run.

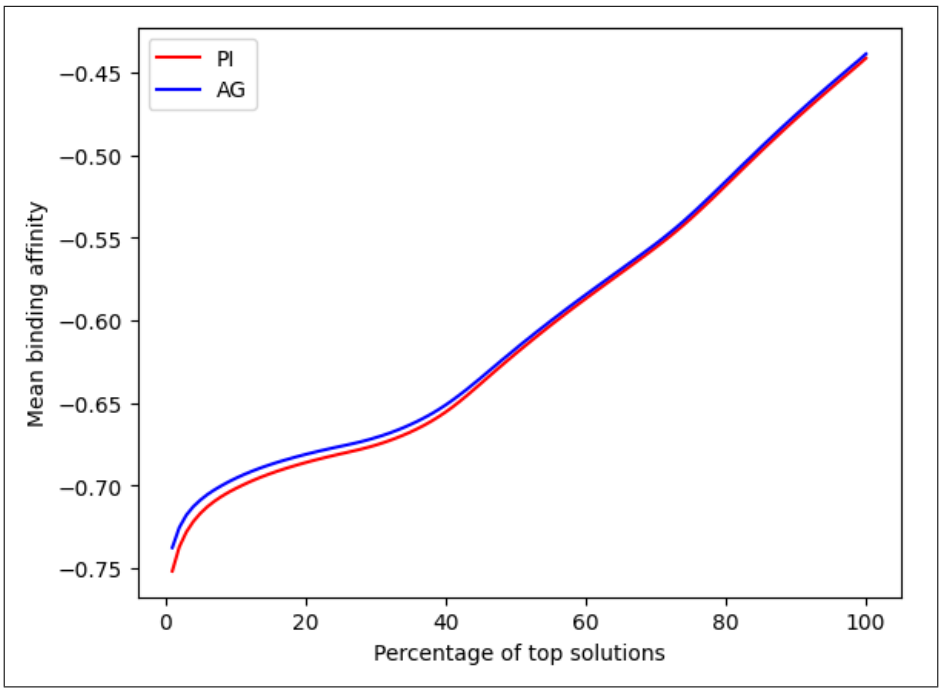

Figure 5.6: Graph showing mean binding affinity of the last generation across all runs over a percentage of the top scoring solutions (ranked by binding affinity) ranging from 1% to 100%.

Interestingly, the binding affinity score for PI appears to be slightly lower than AG (again a lower binding score indicates higher fitness). There appears to be the greatest difference when considering the top $\sim 1\%$ to the top $\sim 30\%$ of solutions. This may be a consequence of the possibility that the islands explore distinct areas of the search space, meaning PI is able to find more high scoring solutions than AG. Additionally, the sharing of these top solutions via migrations may also guide the search towards more of these high scoring solutions.

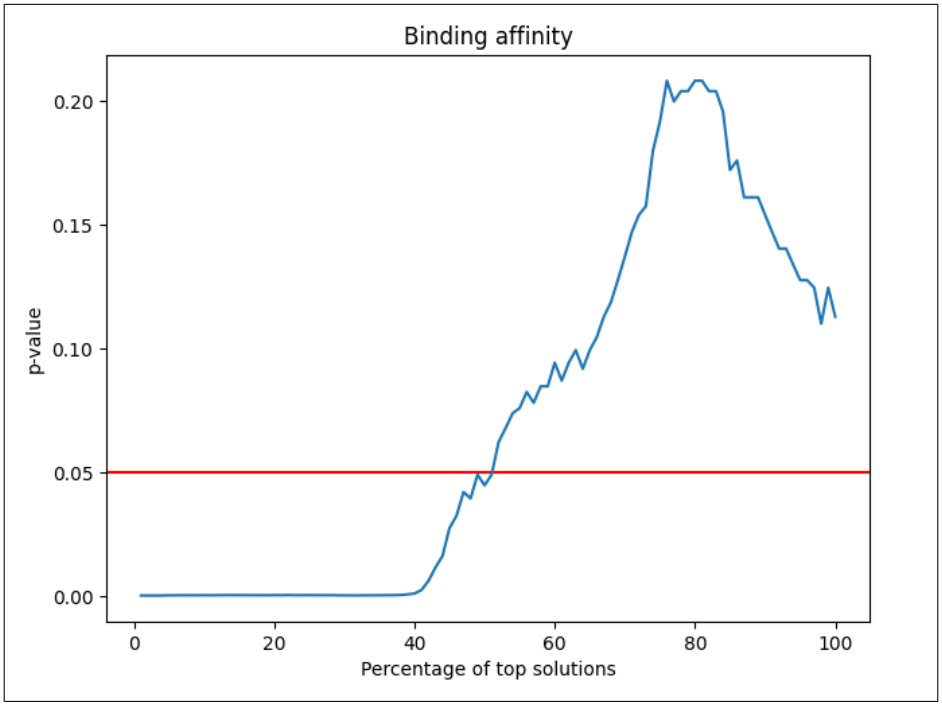

Figure 5.7: Graph showing Wilcoxon Rank Sums Test p-value vs. percentage of top scoring solutions.

Figure 5.7 exhibits a similar pattern to the diversity, where there is a significant difference when considering a portion of the top solutions, however, when the whole population is taken into account, the p-value is greater than the significance level. It is possible that as PI performs a broader search, poorer quality solutions are found as well as higher quality solutions (when compared to AG). Therefore, when averaging over the whole population, the effect that the higher quality solutions would have on increasing the mean is counteracted by the lower quality solutions so, in this case, there is no longer a significant difference between the two algorithms.

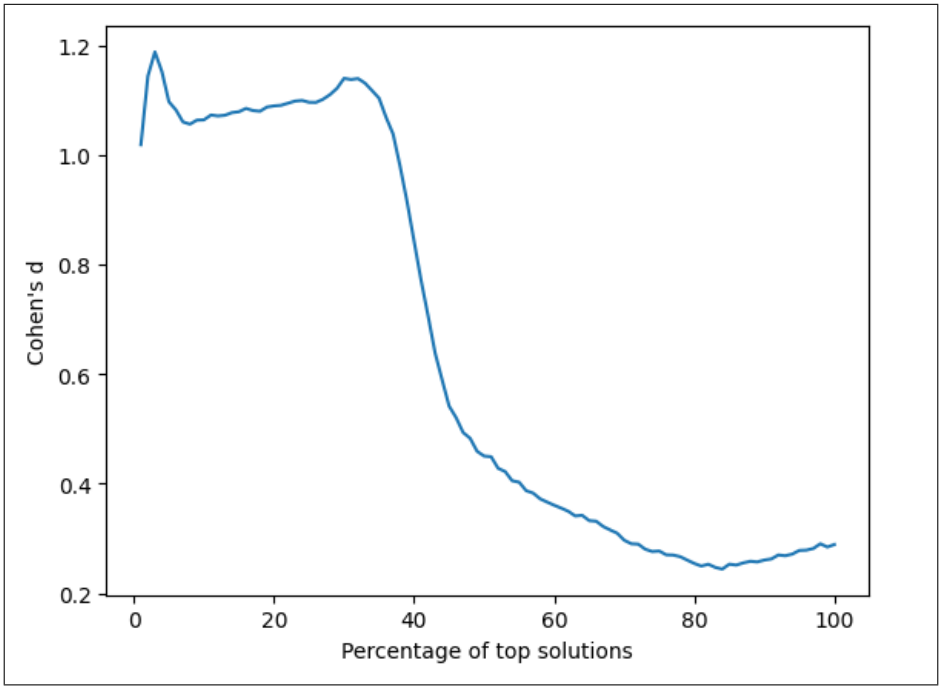

Figure 5.8: Graph showing Cohen's d vs. percentage of top binding score solutions.

When computing the Cohen's d effect size, the same pattern is observed where the top $\sim 1\%$ to $\sim 40\%$ produce the biggest increase in magnitude, with a maximum value of 1.19 (2 d.p.) indicating a 'large' effect size.

5.1.3 Computation time

On the AWS c6a.2xlarge instances, one run of AG using the parameters described in section 4.2.2 took approximately four hours to complete. One run of PI took approximately eight hours. Clearly the PI implementation takes much longer to complete than AG, however it is possible that this effect on the time required to complete a run could be greater reduced by adapting PI to be run on a high performance computing cluster (see section 5.2.2).

5.2 Discussion

These results confirm the hypothesis that the parallel islands implementation increases diversity of the generated solutions. It was also discovered that the binding affinity was improved, likely due to a broader search being conducted meaning that there is a greater chance of high quality solutions being discovered. In addition, by sharing the ligands with the highest binding affinity, the migrations likely guide the search towards high quality solutions. This is an exciting

result as it shows that binding affinity and diversity (two properties which are often thought to counteract each other) can actually both be improved upon.

5.2.1 Limitations

Unfortunately, due to the fact that AutoGrow4 already makes use of global parallelism for its computationally intensive tasks, there was little speedup to be gained from the parallel islands implementation. As PI computes two populations rather than just one as with AG, computation time is effectively doubled. This is partly due to the fact that the computers used for this project are not particularly well suited to running parallel islands genetic algorithms due to their relatively low number of cores. It would be interesting to adapt the PI implementation to be run on a high performance computing (HPC) cluster, with each island allocated its own node. In theory, this would avoid the issue of islands fighting for resources, as was thought to be the problem when executing multiple islands on a single machine.

As a result of the increased computation time, a key limitation of the findings of this project is that there was insufficient time to run the algorithms to convergence. This means that it is not possible to make any claims about PI in relation to its effect on binding and diversity scores over longer runs. We can speculate that this result will hold true for longer runs but without experimental validation of this claim we cannot be certain.

5.2.2 Future work

As mentioned above, the parallel islands implementation could be modified to execute on a HPC cluster. The existing system for migrations using RabbitMQ could easily be transferred to a scenario with multiple nodes. An orchestration node would be needed to host the RabbitMQ server and to spawn all of the islands on their respective nodes. The island nodes would communicate with each other through the RabbitMQ server in the same way as before. This would be valuable as it would mean that much larger and more meaningful runs (comparable to the large scale *de novo* runs of Spiegel and Durrant (2020)) could be carried out in a reasonable time-frame.

In particular, it would be valuable to conduct a set of runs similar to those described in section 4.2 but with each run being allowed sufficient time to converge. This would allow for a fairer comparison between the two algorithms as it would rule out the effect of any pre-convergence fluctuations in diversity and binding scores and it would be more reflective of a real-world use case.

There are a variety of ways in which the parallel islands genetic algorithm can be configured, with parameters such as: the number of islands, the number of solutions migrated per generation, selection strategy for which solutions to migrate and more (Harada and Alba 2020). This project focused on a proof of concept involving only two islands and migrating the single top solution from each island once per generation. It may be promising, in future work, to experiment with a greater number of islands in hopes of achieving an even

broader search of the space of potential candidate drugs. Different migration strategies may also provide interesting results.

5.2.3 Personal reflection

As my first piece of research focused work, this project served as a valuable learning experience. It challenged my critical thinking skills in all aspects from writing the literature review, to designing and implementing the system in an efficient way and to analysing the results. In terms of technical skills, I was able to put my Python skills into practice and I gained valuable experience using AWS which I believe will help me in the future with my career. Having little prior knowledge of genetic algorithms, I feel that I have been able to learn a lot about this interesting topic and about the different kinds of parallel genetic algorithms. I am grateful to have had the opportunity to work on such an exciting cross-disciplinary project and I have been able to expand my otherwise limited knowledge of the fields of statistics, medicine and cheminformatics.

5.3 Conclusion

This project set out with the goal of answering the question "Will a parallel islands implementation of AutoGrow4 produce more diverse solutions than the original?". By developing a proof-of-concept parallel islands implementation, conducting multiple runs and analysing the results using statistical methods, this question was answered. It was shown that not only does the parallel islands implementation improve the diversity of the solutions produced, it can also result in higher scoring molecules being found. This is a compelling result as it highlights the utility of parallel islands genetic algorithms and their applicability to *de novo* drug design, motivating future work into this area.

Appendix A

Significance runs parameter listings


```
{
"nn1_script": "/autogrow4/autogrow/docking/scoring/nn_score_exe
/nnscore1/NNScore.py",
"nn2_script": "/autogrow4/autogrow/docking/scoring/nn_score_exe
/nnscore2/NNScore2.py",
"conversion_choice": "MGLToolsConversion",
"obabel_path": "/usr/bin/obabel",
"custom_conversion_script": "",
"prepare_ligand4.py": "/mgltools_x86_64Linux2_1.5.6/
MGLToolsPckgs/AutoDockTools/Utilities24/prepare_ligand4.py",
"prepare_receptor4.py": "/mgltools_x86_64Linux2_1.5.6/
MGLToolsPckgs/AutoDockTools/Utilities24/prepare_receptor4.py",
"mgl_python": "/mgltools_x86_64Linux2_1.5.6/bin/pythonsh",
"start_a_new_run": true,
"max_time_mcs_prescreen": 1,
"max_time_mcs_thorough": 1,
"min_atom_match_mcs": 4,
"protanate_step": false,
"rxn_library": "all_rxns",
"rxn_library_file": "",
"function_group_library": "",
"complementary_mol_directory": "",
"number_of_processors": 4,
"multithread_mode": "multithreading",
"selector_choice": "Rank_Selector",
"tourn_size": 0.1,
"top_mols_to_seed_next_generation_first_generation": 200,
"top_mols_to_seed_next_generation": 200,
"diversity_mols_to_seed_first_generation": 200,
"diversity_seed_depreciation_per_gen": 2,
"filter_source_compounds": false,
"use_docked_source_compounds": false,
"num_generations": 15,
"number_of_crossovers_first_generation": 500,
"number_of_mutants_first_generation": 500,
```

```
    "number_of_crossovers": 500,
    "number_of_mutants": 500,
    "number_elitism_advance_from_previous_gen": 200,
    "number_elitism_advance_from_previous_gen_first_generation":
    200,
    "redock_elite_from_previous_gen": false,
    "LipinskiStrictFilter": false,
    "LipinskiLenientFilter": true,
    "GhoseFilter": false,
    "GhoseModifiedFilter": false,
    "MozziconacciFilter": false,
    "VandeWaterbeemdFilter": false,
    "PAINSFilter": false,
    "NIHFilter": false,
    "BRENKFilter": false,
    "No_Filters": false,
    "alternative_filter": null,
    "dock_choice": "QuickVina2Docking",
    "docking_executable": null,
    "docking_exhaustiveness": 1,
    "docking_num_modes": null,
    "docking_timeout_limit": 300,
    "custom_docking_script": "",
    "scoring_choice": "VINA",
    "rescore_lig_efficiency": true,
    "custom_scoring_script": "",
    "max_variants_per_compound": 1,
    "gypsum_thoroughness": 3,
    "min_ph": 6.4,
    "max_ph": 8.4,
    "pka_precision": 1.0,
    "gypsum_timeout_limit": 10,
    "debug_mode": false,
    "reduce_files_sizes": false,
    "generate_plot": true,
    "timeout_vs_gtimeout": "timeout",
    "filename_of_receptor": "/Outputfolder/inputs/4
    r6eA_PARP1_prepared.pdb",
    "center_x": -70.76,
    "center_y": 21.82,
    "center_z": 28.33,
    "size_x": 25.0,
    "size_y": 16.0,
    "size_z": 25.0,
    "source_compound_file": "/Outputfolder/inputs/
    source_compounds_islands_run.smi",
    "root_output_folder": "/Outputfolder/",
    "number_of_islands": 2,
    "mgltools_directory": "/mgltools_x86_64Linux2_1.5.6/",
    "chosen_ligand_filters": [
        "LipinskiLenientFilter"
    ],
    "output_directory": "/Outputfolder/Run_0/"
}
```


Listing A.1: Parameters used for parallel islands significance runs


```
    {
```

```
    "nn1_script": "/autogrow4/autogrow/docking/scoring/nn_score_exe
    /nnscore1/NNScore.py",
    "nn2_script": "/autogrow4/autogrow/docking/scoring/nn_score_exe
    /nnscore2/NNScore2.py",
    "conversion_choice": "MGLToolsConversion",
    "obabel_path": "/usr/bin/obabel",
    "custom_conversion_script": "",
    "prepare_ligand4.py": "/mgltools_x86_64Linux2_1.5.6/
    MGLToolsPckgs/AutoDockTools/Utilities24/prepare_ligand4.py",
    "prepare_receptor4.py": "/mgltools_x86_64Linux2_1.5.6/
    MGLToolsPckgs/AutoDockTools/Utilities24/prepare_receptor4.py",
    "mgl_python": "/mgltools_x86_64Linux2_1.5.6/bin/pythonsh",
    "start_a_new_run": true,
    "max_time_mcs_prescreen": 1,
    "max_time_mcs_thorough": 1,
    "min_atom_match_mcs": 4,
    "protanate_step": false,
    "rxn_library": "all_rxns",
    "rxn_library_file": "",
    "function_group_library": "",
    "complementary_mol_directory": "",
    "number_of_processors": -1,
    "multithread_mode": "multithreading",
    "selector_choice": "Rank_Selector",
    "tourn_size": 0.1,
    "top_mols_to_seed_next_generation_first_generation": 200,
    "top_mols_to_seed_next_generation": 200,
    "diversity_mols_to_seed_first_generation": 200,
    "diversity_seed_depreciation_per_gen": 2,
    "filter_source_compounds": false,
    "use_docked_source_compounds": false,
    "num_generations": 15,
    "number_of_crossovers_first_generation": 500,
    "number_of_mutants_first_generation": 500,
    "number_of_crossovers": 500,
    "number_of_mutants": 500,
    "number_elitism_advance_from_previous_gen": 200,
    "number_elitism_advance_from_previous_gen_first_generation":
    200,
    "redock_elite_from_previous_gen": false,
    "LipinskiStrictFilter": false,
    "LipinskiLenientFilter": true,
    "GhoseFilter": false,
    "GhoseModifiedFilter": false,
    "MozziconacciFilter": false,
    "VandeWaterbeemdFilter": false,
    "PAINSFilter": false,
    "NIHFilter": false,
    "BRENKFilter": false,
    "No_Filters": false,
    "alternative_filter": null,
    "dock_choice": "QuickVina2Docking",
    "docking_executable": null,
    "docking_exhaustiveness": 1,
    "docking_num_modes": null,
    "docking_timeout_limit": 120,
    "custom_docking_script": "",
```

```
    "scoring_choice": "VINA",
    "rescore_lig_efficiency": true,
    "custom_scoring_script": "",
    "max_variants_per_compound": 1,
    "gypsum_thoroughness": 3,
    "min_ph": 6.4,
    "max_ph": 8.4,
    "pka_precision": 1.0,
    "gypsum_timeout_limit": 10,
    "debug_mode": false,
    "reduce_files_sizes": false,
    "generate_plot": true,
    "timeout_vs_gtimeout": "timeout",
    "filename_of_receptor": "/Outputfolder/inputs/4
    r6eA_PARP1_prepared.pdb",
    "center_x": -70.76,
    "center_y": 21.82,
    "center_z": 28.33,
    "size_x": 25.0,
    "size_y": 16.0,
    "size_z": 25.0,
    "source_compound_file": "/Outputfolder/inputs/
    source_compounds_islands_run.smi",
    "root_output_folder": "/Outputfolder/",
    "mgltools_directory": "/mgltools_x86_64Linux2_1.5.6/",
    "chosen_ligand_filters": [
        "LipinskiLenientFilter"
    ],
    "output_directory": "/Outputfolder/Run_0/"
}
```

Listing A.2: Parameters used for AutoGrow significance runs

Bibliography

Bagal, Viraj et al. (Oct. 2021). "MolGPT: Molecular Generation Using a Transformer-Decoder Model". In: *Journal of Chemical Information and Modeling* 62.9, pp. 2064–2076. DOI: 10.1021/acs.jcim.1c00600. URL: https://doi.org/10.1021%2Facs.jcim.1c00600.

Canavos, George C (1984). *Applied probability and statistical methods*. Little, Brown.

Cantu-Paz, Erick (1999). "Designing Efficient and Accurate Parallel Genetic Algorithms (Parallel Algorithms)". AAI9952979. PhD thesis. USA. ISBN: 0599561432.

Cohen, Jacob (1988). *Statistical Power Analysis for the Behavioral Sciences*. en. Taylor Francis Group. ISBN: 978-1-134-74270-7.

Computer Aided Drug Design (CADD): From Ligand-Based Methods to Structure-Based Approaches (2022). Elsevier. DOI: 10.1016/c2020-0-04039-9. URL: https://doi.org/10.1016%2Fc2020-0-04039-9.

Devi, R. Vasundhara, S. Siva Sathya, and Mohane Selvaraj Coumar (Feb. 2015). "Evolutionary algorithms for de novo drug design – A survey". In: *Applied Soft Computing* 27, pp. 543–552. DOI: 10.1016/j.asoc.2014.09.042. URL: https://doi.org/10.1016%2Fj.asoc.2014.09.042.

Dey, Fabian and Amedeo Caflisch (Feb. 2008). "Fragment-Based de Novo Ligand Design by Multiobjective Evolutionary Optimization". In: *Journal of Chemical Information and Modeling* 48.3, pp. 679–690. DOI: 10.1021/ci700424b. URL: https://doi.org/10.1021%2Fci700424b.

Dice, Lee R. (July 1945). "Measures of the Amount of Ecologic Association Between Species". In: *Ecology* 26.3, pp. 297–302. DOI: 10.2307/1932409. URL: https://doi.org/10.2307%2F1932409.

Harada, Tomohiro and Enrique Alba (Aug. 2020). "Parallel Genetic Algorithms". In: *ACM Computing Surveys* 53.4, pp. 1–39. DOI: 10.1145/3400031. URL: https://doi.org/10.1145%2F3400031.

Hartenfeller, Markus et al. (July 2008). "Concept of CombinatorialiDe Novo/iDesign of Drug-like Molecules by Particle Swarm Optimization". In: *Chemical Biology &amp Drug Design* 72.1, pp. 16–26. DOI: 10.1111/j.1747-0285.2008.00672.x. URL: https://doi.org/10.1111%2Fj.1747-0285.2008.00672.x.

Hughes, JP et al. (Feb. 2011). "Principles of early drug discovery". In: *British Journal of Pharmacology* 162.6, pp. 1239–1249. DOI: 10.1111/j.1476-

5381.2010.01127.x. URL: https://doi.org/10.1111%2Fj.1476-5381.2010.01127.x.

Li, Yibo, Liangren Zhang, and Zhenming Liu (July 2018). "Multi-objective de novo drug design with conditional graph generative model". In: *Journal of Cheminformatics* 10.1. DOI: 10.1186/s13321-018-0287-6. URL: https://doi.org/10.1186%2Fs13321-018-0287-6.

Meyers, Joshua, Benedek Fabian, and Nathan Brown (Nov. 2021). "De novo molecular design and generative models". In: *Drug Discovery Today* 26.11, pp. 2707–2715. DOI: 10.1016/j.drudis.2021.05.019. URL: https://doi.org/10.1016%2Fj.drudis.2021.05.019.

Mohs, Richard C. and Nigel H. Greig (Nov. 2017). "Drug discovery and development: Role of basic biological research". In: *Alzheimer's &amp Dementia: Translational Research &amp Clinical Interventions* 3.4, pp. 651–657. DOI: 10.1016/j.trci.2017.10.005. URL: https://doi.org/10.1016%2Fj.trci.2017.10.005.

Morgan, H. L. (May 1965). "The Generation of a Unique Machine Description for Chemical Structures-A Technique Developed at Chemical Abstracts Service." In: *Journal of Chemical Documentation* 5.2, pp. 107–113. DOI: 10.1021/c160017a018. URL: https://doi.org/10.1021%2Fc160017a018.

Mouchlis, Varnavas D. et al. (Feb. 2021). "Advances in De Novo Drug Design: From Conventional to Machine Learning Methods". In: *International Journal of Molecular Sciences* 22.4, p. 1676. DOI: 10.3390/ijms22041676. URL: https://doi.org/10.3390%2Fijms22041676.

Nigam, AkshatKumar, Robert Pollice, and Alán Aspuru-Guzik (2022). "Parallel tempered genetic algorithm guided by deep neural networks for inverse molecular design". In: *Digital Discovery* 1.4, pp. 390–404. DOI: 10.1039/d2dd00003b. URL: https://doi.org/10.1039%2Fd2dd00003b.

Olivecrona, Marcus et al. (Sept. 2017). "Molecular de-novo design through deep reinforcement learning". In: *Journal of Cheminformatics* 9.1. DOI: 10.1186/s13321-017-0235-x. URL: https://doi.org/10.1186%2Fs13321-017-0235-x.

Pantsar, Tatu and Antti Poso (July 2018). "Binding Affinity via Docking: Fact and Fiction". In: *Molecules* 23.8, p. 1899. DOI: 10.3390/molecules23081899. URL: https://doi.org/10.3390%2Fmolecules23081899.

Pegg, Scott C.-H., Jose J. Haresco, and Irwin D. Kuntz (2001). In: *Journal of Computer-Aided Molecular Design* 15.10, pp. 911–933. DOI: 10.1023/a:1014389729000. URL: https://doi.org/10.1023%2Fa%3A1014389729000.

Python documentation (2023). URL: https://docs.python.org/3/library/pickle.html.

Sørensen, Thorvald Julius (1948). *A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on Danish commons*. I kommission hos E. Munksgaard.

Spiegel, Jacob O. and Jacob D. Durrant (Apr. 2020). "AutoGrow4: an open-source genetic algorithm for de novo drug design and lead optimization".

In: *Journal of Cheminformatics* 12.1. DOI: 10.1186/s13321-020-00429-4. URL: https://doi.org/10.1186%2Fs13321-020-00429-4.

Whitley, Darrell, Soraya Rana, and Robert B. Heckendorn (1999). "The island model genetic algorithm: On separability, population size and convergence". In: *Journal of Computing and Information Technology* 7.1. Cited by: 310, pp. 33–47. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85006678577&partnerID=40&md5=2695614c164be945b2230da4f8bea201.