Inverse Drug Design Training Environment

Luke Miloszewski
Department of Computer Science
University of Cape Town

Antony Fleischer
Department of Computer Science
University of Cape Town

Jesse Smart
Department of Computer Science
University of Cape Town

***Abstract*—The goal of this software development project is to develop an OpenAI Gym training environment to be used for inverse drug design. OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. The Training Environment for Drug Design, known as TEDD, provides a Machine Learning solution to the construction and development of molecules used in drug research. The existing TEDD framework has been extended to allow a user to observe the actions of a learning agent. Furthermore, the changes in the environment state are accessible and easily comprehensible to a first time user or an advanced research team. Fundamentally, given an initial molecule and an optimisation goal, the training environment passes information to a learning agent - which modifies a molecule in pursuit of its optimisation target. Additionally, the environment has the functionality to accommodate a more advanced Reinforcement Learning policy. The agent's behaviour and interaction with the environment clearly demonstrates the fundamental workings of the OpenAI Gym framework, as well as the basic principles of Reinforcement Learning. Furthermore, the environment framework uses pragmatic chemistry-based algorithms to ensure that only valid chemical modifications are allowed within the environment. This was done to ensure that the TEDD framework has both research and educational value in the future.**

I. Introduction

The goal of this project was to extend an existing software development project - with the aim of providing assistance to other inverse drug design research programmes. Given its intention, the project development team was asked to modify an existing project - a reinforcement learning training environment - so that it contained the required functionality as requested by the client - Mr Geoff Nitschke.

Although the scope of the specifications for TEDD was extensive, additional functionalities were added to the environment to ensure that the environment runs efficiently and informatively. Essentially, the shell of a training environment was provided, and the project development team was required to implement various classes, packages and intra-class functionalities. More specifically, methods were added to allow the environment to: set the current environment state to either a starting molecule containing a single carbon atom; at each step, given the agent's chosen action, set it's current state to the molecule resulting from the agent's chosen action, and consequently, determine the list of chemically valid modifications that can be made to the current state; and to render the current state of the environment graphically.

The TEDD program provides the classes and methods that allow an agent - a learning piece of software that uses reinforcement learning policies to make decisions - to modify a molecule and receive rewards. The repetition of the modification and rewards process provides the agent with the notion of good actions and 'bad' actions. A 'good' action is one that modifies the molecule to be closer to the optimisation goal, whilst a 'bad' action creates an invalid or less similar molecule. By studying how the agent's actions change the state of the environment (the molecule), a research team would be provided with great insight into the step-by-step process of a molecule's creation. Additionally, the software development team implemented two primary out-of-spec functionalities: firstly, graphical molecule structures that provide the user with insight into the state of the environment, and secondly, advanced chemical manipulation methods that provide the environment and user with more information.

Lastly, credit must be given to the creator of the original TEDD framework - Robert Maccallum (2020). His framework and advice allowed this software development project to be extremely accessible and educational. Although this research report provides valuable insight into the principles of OpenAI Gym and inverse drug design, the TEDD framework still has so much potential in the future - both educationally, and in the drug design industry.

II. Requirements Analysis

1) Gathering user requirements and specifications: It is important to understand the users when building a system that seeks to fulfil these users' requirements. The primary users we have identified are: machine learning engineers, biomedical engineers and pharmaceutical engineers. Using this information, we identified our supervisor as a suitable user to speak with. His experience in biologically inspired machine learning as well as agent-based systems made him a suitable user to survey for program specifications. This process allowed us to determine specific feature expectations, resolve feature ambiguities and avoid adding unnecessary features (gold-plating). This ensured that the final system conforms to the client needs rather than moulding client expectations to fit the requirements.

2) Regular meetings with our supervisor: The project was initiated with three weekly meetings with our supervisor. These were extensive discussions that allowed us to define the scope and goals of the project and develop a working time frame in which to realise these goals. Each team member familiarised themselves with this information before development began. This ensured that there was a clear understanding of what was required and that there was

an open channel of communication in order to clarify any further queries. Throughout the development of the project, we held regular meetings with our supervisor either virtually or through email conversations which ensured that the system being built aligned with the specified user requirements. This ensured that any changes to the system were reviewed and that any challenges to development were reviewed and resolved timeously.

3) Collaboration amongst our team: The project was worked on collectively by all three team members. We made use of daily check-ins (over voice calls or video calls) during each feature development to ensure that we reviewed what was done the day before and what was planned for the day ahead. We made use of weekly check-ins (over voice call or video calls) to ensure the features being developed aligned with the requirements at hand.

4) Prototyping: We developed a prototype within the first 4 weeks and demoed this to the client. This was to demonstrate that we had fulfilled the correct user requirements and that we could extend on this basic functionality. We chose an evolutionary style of prototyping that aligned with our incremental development and evolving understanding of specifications. This served as the core of the eventual system. Furthermore, we chose a combination of vertical and horizontal prototyping methods. Firstly, the vertical method ensured that we provided a detailed implementation of a few important methods. Secondly, the horizontal method ensured that we could work iteratively and cover a wide range of functions that would be present in the final program, without completing all the implementation details or adding unnecessary complexity.

5) Final user requirements: Upon completing the user requirements survey, we broke them down into functional requirements and non-functional requirements. Functional requirements refer to the expected behaviour of the system and determine what the system must or must not do. In contrast, non-functional requirements refer to how a system should achieve its functionality and serve as constraints on the system design.

The functional requirements that we gathered for the training environment include: allow the user to choose a starting molecule (either a carbon molecule or a random molecule), allow the user to choose a target molecule (either a specific molecule or a random molecule), allow the user to choose an optimisation goal (a percentage describing the similarity between the starting molecule and the target molecule), allow the user to initialize the environment, allow the agent to choose an action to perform (based on its decision-making policy and the actions available from the action space), allow the environment to facilitate the execution of an action and, allow the environment to render the graphical representation of the molecule.

The non-functional requirements that we gathered for the training environment cover aspects such as performance, scalability, capacity, recoverability, maintainability, data integrity and interoperability. They include: the training environment will run for 10 episodes with 20 steps per episode (unless a terminal state has been reached), the dataset contains the SMILES representation of a collection of chemically-valid molecules, the modification log will be saved at program termination, modification to a molecule must be chemically-valid, the code will be hosted under a private repo on Gitlab, the agent class allows for a custom reinforcement learning algorithm to be implemented and, the dependencies necessary to run this system are included in the requirements.txt file and the setup.py file.

6) Usability: The training environment provides the user with a terminal-based input and output interface whilst providing the user with graphical outputs. In terms of usability, the user will be able to: interact with and choose options based on a given set of actions, input responses based on entering a numerical value for a given response and receive updates regarding the progress of learning, including: the current state of the environment (similarity, number of atoms, number of bonds), the most recent action, the reward value associated with that action and any information necessary from the environment.

7) Use Cases: The use cases that have been identified include: User chooses starting molecule, User chooses target molecule, User chooses optimization goal (similarity %), User initialises environment object, User initialises agent object, User updates environment state, Agent chooses an action to perform, Environment executes action chosen by agent, Environment determines chemical validity of molecule, Environment calculates numerical reward and updated state, Environment renders current state of environment, Agent updates decision-making policy and User terminates environment. To view the detailed use case descriptions, please refer to the Use Case Descriptions table (Table 2) in the appendix.

III. Design and Architecture

A. Overview

This inverse drug design training environment builds a chemically valid molecule by making small incremental changes to a molecule object. The environment is designed to be paired with a learning algorithm implemented within the MoleculeAgent class which will contain information about the possible manipulations that can be performed on the molecule in the environment. These manipulations include the addition or removal of atoms to the front or back of a molecule with a specified type of bond. With a learning algorithm implemented, the environment will be directed towards the desired state by optimizing the reward the agent receives from the environment after each iteration.

B. Molecules, RDKit and Chemistry

The molecule is represented as an RDkit SMILES object, which is a Simplified Molecular Input Line Entry System. This allows manipulations to be performed to the string representation of the SMILES object and tested for chemical validity by temporarily converting the string into an RDkit Mol object. This was chosen since string manipulation is simple and computationally inexpensive. The chemistry of our

environment is handled by an object that stores the initial molecule and the optimal molecule. This class has a one to one relationship with the environment. Any modification to the molecule being built is handled by the methods in this class which utilises the methods of the RDKit Chem package's Mol object. This package allows the environment to get any information that a periodic table would be able to provide.

C. Agent and Decision-Making Policy

The MoleculeAgent class contains a policy which acts as the probability distribution of the possible decisions the agent can take. This policy is updated after every iteration by the learning algorithm. The distribution of the policy is a simple conditional probability which fine-tunes a vector of parameters in order to take the best action given an observed state. This can be statistically described as:

$$\pi(a|s, P) = Pr\{At = a|St = s, Pt = P\}$$

Where:

- At is the action space
- St is the state space
- Pt is the policy containing the vector of parameters.

D. Project Directory Structure

The structure of our environment follows the design framework of an OpenAI Gym training environment. This provided us with an established blueprint to manage the complex communication among the system components. The simplified file hierarchy skeleton in the project directory of such an environment looks as follows:

- Gym_molecule
- __init__.py
- envs
- __init__.py
- molecule_env.py
- README.md
- setup.py

E. Environment behaviour

The functionality of these training environments is handled by methods that complement the basic iterative course of a reinforcement learning environment. These methods are contained in the molecule_env.py as seen below in the analysis class, and they interact with a learning agent and user:

- __init__
- Step
- Reset
- Render (outputs to the user)
- Close

F. System Architecture

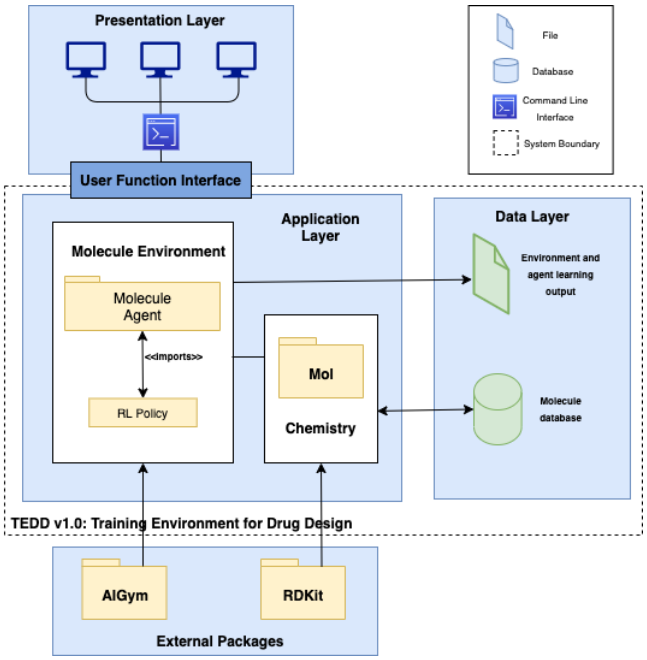


Fig. 1. Architecture diagram

1) Application layer: The application layer processes the changes to the state of the environment. The molecule Environment contains the functionality to implement an RL algorithm and allow the agent to act accordingly to its policy; returning a decision to the environment. The chemistry package contains the functionality to change the variables of the Mol object that determine the state of the environment. The main variables that determine the state of the environment are recorded in the Mol object of the Chemistry package that can be seen in the analysis class diagram included below.

2) Data Layer: The Mol class interacts with the database containing the name and SMILES structure of a collection of molecules. An output is provided to a file in the TEDD directory that contains valuable information about the learning session, the Mol manipulations, and the changes of the environment.

3) Presentation Layer: The presentation layer interacts with the user through command-line outputs. The render function within the molecule environment has the ability to display the changes of the molecule being developed as a graph structure.

4) External packages: The OpenAI Gym package is a toolkit for developing reinforcement learning algorithms and provide a solid structure to support the molecule environment. RDKit (Rational Discovery Kit; developed by Greg Landrum)

is an open-source cheminformatics package that ensures that the changes to the Mol object are valid on a chemical level.

G. System Class Diagram

This Analysis Class diagram shows the specific variables and data types of the environment members and shows the interactions and relationships between them. Further, it models the system dependencies on the actors' attributes and methods.

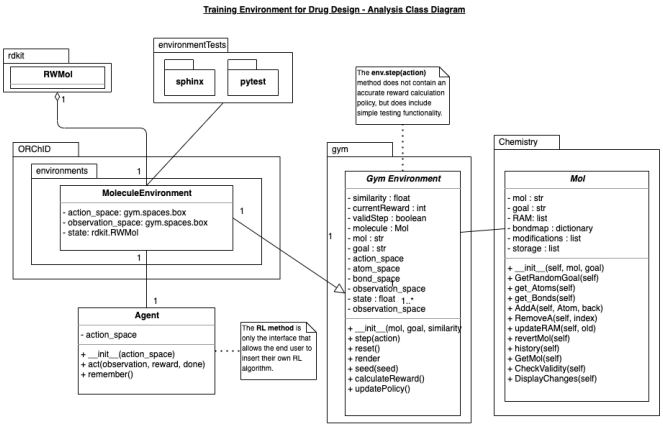


Fig. 2. Analysis Class Diagram

IV. IMPLEMENTATION

A. Overview

The system runs in an active virtual environment called molecule environment. The environment ensures that our required system dependencies are identical on all operating systems and machines. The implementation of our system within this environment ensures that the inputs received by another class are valid and no information has been lost and processed invalidly. This is achieved by encapsulating the methods that operate on our data. Moreover, the classes that process the data ensures that it is validated before being passed to another class. The manipulation of the variables that determine the state of the environment are done so via the RDKit package. The robustness of the molecule manipulation relies on this package since it has been extensively researched and tested.

B. State diagram

The state machine diagram describes the relevance of the implementation of the core functions of our molecule environment mentioned in the design overview. These functions control the communication between our objects. The diagram depicts the different states of the system and is not relevant to the modelling of the state of the environment. The state of the environment changes within the developing state of this diagram. The ready state contains the state of the environment at instantiation.

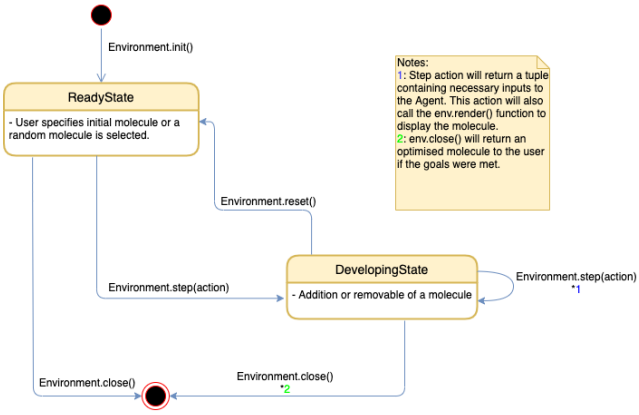


Fig. 3. State Machine Diagram displaying the states of the system

C. Change of State

- The init method is called by the main method that the user interacts with, this instantiates an instance of the Mol class and molecule agent class. Once the ReadyState has been reached, an observation of the initial state of the environment and the possible actions are passed to the molecule agent.
- The step method executes the decision process of the agent and makes an appropriate change to the environment via the accessor and mutator methods of the Mol class. The environment is now in the DevelopingState, in which the iterative development of the molecule can take place via the step method.
- The environment can return to the ReadyState if the reset method is called, this will reset the molecule's attributes to their initial state that was specified by the user.
- If the optimization goal is met or the iterations of each training episode are completed, the close method will end the iterative development.

D. Sequence diagram

The diagram below displays the flow of data between the actors in our system. Different sequential scenarios are covered, starting and ending with the user and flowing from top to bottom. Everything to the left of the blue dashed line indicates the functionality of the system at present. A reinforcement learning algorithm will receive an observation of the state and an associated reward as inputs from the environment and alter its decision making policy accordingly. The RL algorithm adds intelligence to the development process and will set the trajectory of the state towards optimisation. The case for ending the loop in this diagram would be either

when the learning iterations have ended or the molecule being built reaching the target similarity to the target molecule.

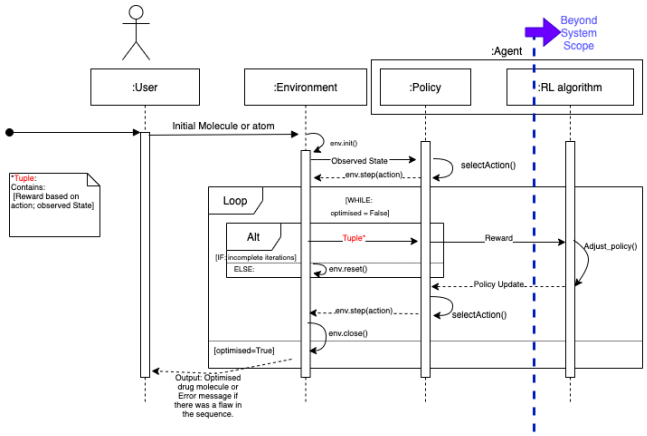


Fig. 4. Sequence Diagram

E. Optimization of State

The desired state of the environment is specified by the user as a percentage. A target molecule will act as the states trajectory. The RDKit contains functionality to return a statistic called Tanimoto Similarity. One can extrapolate that similar fingerprints of molecules have similar physical properties such as solubility. The environment can develop a molecule to have a target similarity to our goal molecule.

F. Technologies

The code is written in Python3 and Jupyter notebook files are used for presentation of outputs to the user. Jupyter creates a localhost that allows for the files within the work tree to be accessed, edited and executed. Jupyter's line by line output style was helpful for testing, experimenting with and debugging the system and packages during the development process. The management of the environment is handled by Ananconda, an open source agnostic package manager and environment management system. Python libraries and dependencies essential to the functionality are listed in the environment.yml file.

V. Validation and Verification

Testing was done in order to ensure the system was operational and achieved the functional and non-functional user requirements. Testing was performed using the pytest library and included unit testing, integration testing and validation testing. A summary of the test processes and techniques used is shown in the Testing Summary table (Table 1) below. Furthermore, please refer to the README file hosted on the code repository for testing information. Otherwise, in order to run the tests with pytest, please change directory to the main directory of the repo and run the following command:

$$python\ -m\ pytest$$

Process	Technique
Unit Testing: test methods and state behaviour of classes.	White-box testing
Integration Testing: test the interaction of sets of classes.	Top-down testing
Validation Testing: test whether customer requirements are satisfied.	Use-case based black-box testing and acceptance tests.

TABLE I: Testing Summary

A. Unit Testing

Unit testing was conducted to determine the functional correctness of stand-alone modules. This was done using a white-box approach in which the internal structure, design and coding of software was tested to verify the flow of information within the system and to improve design and usability. Unit testing was done iteratively throughout development after each new feature was added. This approach ensured that incremental updates were made to the code which allowed any difficulties to be minimised or addressed before further concerns could manifest.

Unit testing was done for each of the following files: Chemistry.py, Molecule_Agent.py and molecule_env.py.

For Chemistry.py, each method was tested to ensure it functioned correctly regarding the valid modification of the molecule. This included: returning the number of atoms in a molecule, returning the number of bonds in a molecule, adding an atom to a molecule, removing an atom from a molecule, checking the validity of a molecule and checking the similarity between two molecules. Refer to Chemical Validity Using RDKit below to determine how we ensured molecule integrity and chemical validity.

For Molecule_Agent.py, each method was tested to ensure it functioned correctly regarding the selection and storage of an action. This included: determining an appropriate action to perform based on the state of the environment and updating the memory component of the agent with each chosen action.

For molecule_env.py, each method was tested to ensure it functioned correctly regarding each of the necessary steps in the training environment process. This included: ensuring the step() method returns the state of the environment, the seed() method returns the pseudo-random number used for random number generation, the render() method returns the graphical representation of the molecule object, the reset() method resets the state of the environment and the init() method initialises the environment with the appropriate fields.

Unit testing (by use of white-box testing) was effective because it allowed us to ensure the functionality of the code

was met as the most basic level. This ensured that our code could scale successfully as more features were added and allowed us to verify the flow of information within the system and to improve design and usability

B. Chemical Validity Using RDKit

To ensure chemical validity when modifying the molecule, the system made use of RDKit. RDKit is an extensively well-researched collection of chem-informatics and machine-learning software written in C++ and Python. This toolkit is open source and has built-in methods to modify molecules and check their validity.

C. Integration Testing

Our team has built code both collectively (sharing a screen while one person codes) and individually (delegating a certain feature to build). Integration testing was done to expose defects in the interaction between these software units when they were integrated. This was done using a top-down approach in which integration testing takes place from top to bottom following the control flow of the software system. Higher level units are tested first and then lower level units. Integration testing was done after each sprint. This ensured that there were no major changes to the code at any point in time, thus minimising the risk of errors and ensuring that the code interacted well.

The main classes to integrate were: Main_Molecule.ipynb, MoleculeAgent.py, Molecule_env.py and Chemistry.py. Main_Molecule.ipynb facilitates the functioning of each of these modules and served as the backbone of our integration testing. With top-down testing, we started by simply initialising the agent and environment class from within the main class. Once this functionality passed the integration test, we expanded the coverage of methods included in the main class, iteratively including more functionality until all methods were incorporated from both the agent and environment.

Separately, we had to integrate the chemistry class into the environment class. We started by first simply initializing a chemistry (or molecule) object from within the environment. We then expanded the scope of the chemistry class to include all the necessary methods needed to maintain chemical integrity and chemical validity.

We noticed our main integration error when we included the Chemistry.py file. This file ensures that a molecule is chemically valid and controls the modifications of a molecule. The error arising was a result of how the agent would make actions randomly and not based on chemical validity. We resolved this bug by simplifying the molecule object and ensuring that the agent may only perform actions in relation to what is valid.

Integration testing (by use of top-down testing) was effective because it allowed us to ensure that all our modules within the system interacted as expected and that we could validate the control flow of the software system.

D. Validation Testing

Validation testing was conducted to ensure that all functional requirements as previously specified were fulfilled. First, acceptance testing was done during the prototype demo. This ensured that the system under development was aligned with the user needs and requirements and that future developments were on track to be accepted by the user. Second, use-case based black-box testing was done in which the internal design of the system was not considered and instead focused on the inputs of the user and the expected output of the system. For each use case, a test case was developed that evaluated the input, the behaviour being tested and expected output. Please refer to the Test Case Descriptions table (Table 3) in the appendix for a detailed outline of the testing done. Furthermore, to view the results of each test case, please refer to the Test Case Screenshots figure (Figure 9) in the appendix.

VI. Conclusion and Results

A. Discussion of Results

Given the goal of this project, to develop an AIGym training environment to be used for inverse drug design, the results are represented in the successful execution and understanding of the environment. The TEDD is intended to be used further in research and education, and so although it is a well polished, complete program, it is intended as an intermediate product in the context of actual inverse drug design. Before explaining the outputs and results of the environment, it is extremely important to understand what information the environment is providing the user. An explanation of how to read the results of each of the environment's iterations is provided in the User Manual section at the end of this report. Additionally, the TEDD framework has two spheres within which results can be analysed and discussed.

1) Environment outputs: A considerable amount of the aforementioned iteration is implicit in the TEDD program, however, the explicit, visual aspect of the program provides the necessary output to substantiate the existence of a successful interaction between the environment and a learning software agent.

Given the initialization of the environment by a user, the environment iteratively returns a graphical output. This output provides a concise summary of the environment's state, and additional information about its previous interaction with the agent.The following output demonstrates a simple episode of the environment's lifespan:

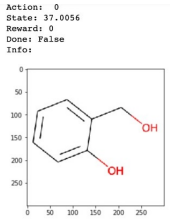


Fig. 5. Environment initialization output

After one iteration of an environment-agent interaction, the state of the environment has changed, and the environment has provided the agent with a reward based on the value of its action. This can be seen above: the state - which represents the chemical similarity between the developing molecule and the optimization goal - has increased to 37.0056%; the agent's action of 0 - adding an atom to the back of the molecule - was evaluated by the environment and given a reward of 0; and the graphical representation of the developing molecule has updated to reflect the new atom.

The environment-agent interactions will continue until the Done modifier is true - either when the environment reaches its optimisation goal or the end of its lifespan. The other two iterations of this simple example can be seen below:

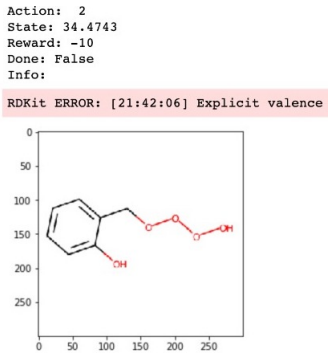


Fig. 6. Environment chemical validity output

This is an important iterative step in the context of the overall environment. The agent attempted to add an atom with a bond to the end of the molecule, however, the consequence of that action was that the developing molecule would be chemically invalid. Therefore, the environment provides an error, and penalizes the agent heavily.

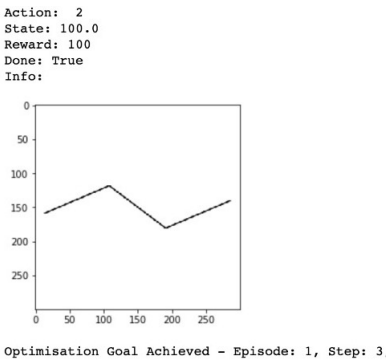


Fig. 7. Environment completion output

As is evident, once the optimisation goal is achieved, the environment stops iterating. It presents a summary of how the initial molecule reached its target molecule optimisation goal. In the example above, the environment reached its goal in 3 steps.

It is important to note that the possibilities that the TEDD environment presents are endless, and so the example above simply provides a concise summary of the results of a general episode of the environment. More importantly though, the TEDD program provides a comprehensive framework for inverse drug design, as well as setting a foundation for further analysis of Reinforcement Learning and chemical molecular modification techniques

2) Computational Complexity: It is of paramount importance to understand the computational complexity of any computer program. Considering that, a theoretical analysis of the complexity of the TEDD framework's main computational tasks is provided here.

3) Environment and Agent Interactions: The TEDD program provides a user with multiple episodes and multiple iterations of the training environment. Each episode contains, at most, 20 iterations of the environment-agent interaction, and by default, there are five episodes per execution - although this can be changed by the user.

Algorithmically, the repeated occurrence of a repeated interaction is represented by a nested for loop. Using the Big-O notation to describe the limiting behaviour of this program's performance, the computational complexity of the environment's interaction with the agent can be represented as:

$$O(f(n)) \; where \; f(n) = (5n * (20n + 1 + 1 + 1)$$

However, given the principles of the analysis of computation complexity, this can be simplified to:

$$O(n^2)$$

VII. User Manual

A. README

The README.md file within the TEDD parent directory provides an explanation on how to install and set up the environment and its necessary packages. Furthermore, it provides a step by step process of how to initiate the environment and how to start automated drug design. Therefore, please ensure you have consulted the README.md document before attempting to use the TEDD program.

B. User instructions

1) : After installation and setup, the program can be run via a locally hosted Jupyter Notebook on the user's device. Once Jupyter is open, open the Main_Molecule.ipynb file.

2) : Once open, the TEDD framework can be run by selecting the run button within Jupyter Notebook.

3) : Once running, the user will be provided with the following output: Inputting 1 into the text field will cause the environment to initiate with a carbon molecule, whereas inputting 2 will allow the environment to choose a random initial state for the environment. The random options come from a provided .csv file of molecules.

4) : Once the user has selected an initial state, they will be prompted to enter a target molecule. The first option, 1, allows the user to enter a custom goal. For example, if the user wishes to see how an agent modifies a single carbon into carbon-dioxide, they should enter CO2 as the target molecule. The second option, 2, allows the environment to select a random target molecule.

5) : Lastly, the user will be prompted to enter an optimisation goal. This the desired similarity between the developing molecule - the initial molecule - and the target molecule. For example, a value of 80 would set the optimisation goal at 80% similarity.

6) : Finally, the user will be provided with the iterative results of the interactions between the environment and the agent. This is represented in the format shown below. The explanation of the figure follows:

```
Action:  5
State: 15.7895
Reward: -10
Done: False
Info:
```

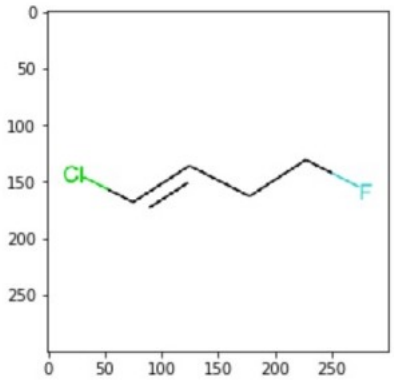


Fig. 8. iteration output

- Action: the previous action selected by the software agent. Each action represents a molecular modification. The eight available actions to the agent are:
  1) Add Atom to Back of Molecule
  2) Add Atom to Front of Molecule
  3) Add Atom with Bond to Back of Molecule
  4) Add Atom with Bond to Front of Molecule
  5) Add Bracketed Atom to Molecule
  6) Remove Atom from Molecule
  7) Remove Bond from Molecule

7) : As mentioned above, the environment will run until the Done modifier is true. At the completion of every iteration, the user will be presented with a similar output pane. Additionally, the environment may be reset or stopped at any point, by using the control functions on Jupyter Notebook.

ACKNOWLEDGMENT

The authors would like to thank...

Maccallum, Robert 2020. Training Environment for Drug Design. https://github.com/robmacc/capstone-molecule-environment

APPENDIX

```
Welcome to TEDD, the Training Environment for Drug Discovery.
In order to run the environment, you will need to input a starting molecule,
a target molecule and an optimisation goal.
```

Test Case 1

```
Step 1) Choose starting molecule:
Input '1' to choose a CARBON molecule.
Input '2' to choose a RANDOM molecule.
Input:  3
Incorrect Input.
```

Test Case 3

```
Step 1) Choose starting molecule:
Input '1' to choose a CARBON molecule.
Input '2' to choose a RANDOM molecule.
Input:  2
```

Test Case 2

```
Step 2) Choose target molecule:
Input '1' to SPECIFY a molecule.
Input '2' to choose a RANDOM molecule.
Input:  3
Incorrect Input.
```

Test Case 5

```
Step 2) Choose target molecule:
Input '1' to SPECIFY a molecule.
Input '2' to choose a RANDOM molecule.
Input:  2
```

Test Case 4

```
Step 3) Choose optimisation goal:
Input a floating-point value between 0 and 1.
Input:  0.5
```

Test Case 6

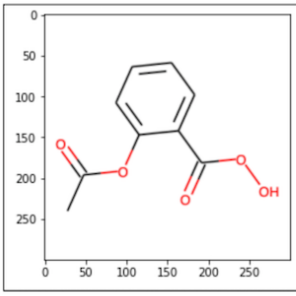
```
Environment initialiased.
Agent initialiased.
```

Test Case 7 and 8

```
Episode 1, Starting State: (similarity = 60.4%, atoms = 4, bonds = 4)

Step 1
Action: 1
Updated State: (similarity = 61%, atoms = 5, bonds = 4)
Reward: 1
Done: False
Info: {}
```

Test Case 9, 10, 11 and 12



Test Case 10

Fig. 9. Test Case Screenshots

Use Case	Agent	Description
Choose starting molecule	User	User chooses a starting molecule; either a carbon molecule or a random molecule generated from the in-built dataset of molecules. If the user inputs an invalid starting molecule, the user will be prompted to re-enter a valid starting molecule.
Choose target molecule	User	User chooses a target molecule; any molecule which is chemically valid. If the user inputs an invalid target molecule, the user will be prompted to re-enter a valid target molecule and will be shown a list of possible molecule choices as reference.
Choose optimisation goal (similarity %)	User	User chooses an optimisation goal (similarity %). The similarity % is a value between 0 and 1. If the user inputs an invalid optimisation goal, the user will be prompted to re-enter a valid optimisation goal and will be shown the range of valid options.
Initialise environment object	User	User creates an environment object which is used as an interface between the client, the agent and the environment. If the environment name does not exist, the gym package will return an error message stating this.
Initialise agent object	User	User creates an agent object which uses a decision-making policy to execute actions within the environment.
Update environment state	User	The reset() method of the environment is called in order to set the current environment state to either a starting molecule containing a single carbon atom or to a random molecule from the given dataset. If the chosen molecule does not exist, the environment will return an error message stating this.
Choose an action to perform	Agent	Agent determines an action to execute based on its decision-making policy (see Use Case 12). The action is passed as an argument to the step() method of the environment.
Execute action chosen by agent	Environment	The environment receives an action to execute from the agent. If the action is invalid, the environment will throw an error and prompt the agent to choose a different action. If the action is valid, the environment proceeds with the following: Set the current state to the molecule chosen by the agent, Determine the list of chemically valid modifications that can be made to the current state (see Use Case 9), Generate a batch of resulting molecules, each one the result of making a modification from the list of possibilities, Calculate the reward vector associated with the chosen action (see Use Case 10), Convert each molecule object into a graph object using the provided methods and Render the current state of the environment for visualisation/demonstration purposes (see Use Case 11).
Determine chemical validity of molecule	Environment	Environment will use RDKit to evaluate the characteristics of the molecule in the current environment state, ensuring it is chemically valid and prescribes to the required optimisation goal. If it is not chemically valid, the reward function used by the environment will return a smaller reward (see Use Case 10). If it does not prescribe to the optimisation goal, the reward function used by the environment will return a smaller reward (see Use Case 10).
Calculate numerical reward and updated state	Environment	Environment calculates a numerical reward and the next state given an action selected by the agent and the current state. The reward is calculated using an underlying reward function. The environment returns the updated state, the observations of possible next states and the numerical reward to the agent.
Render current state of environment	Environment	The render() method of the environment is called and returns the molecule represented by the current state of the environment for visualisation/demonstration purposes. If the molecule cannot be rendered, the string representation of that molecule will be printed to the console.
Update decision-making policy	Agent	Agent updates its decision-making policy based on an underlying reinforcement learning algorithm. The agent seeks to maximise the sum of discounted rewards received by actions that are executed in the environment over time.
Terminate environment	User	User terminates environment if optimisation goal has been reached, if a certain time period has been exceeded or if they explicitly choose to. Before termination, the user will be prompted with the final similarity % and the final modification log.

TABLE II: Use Case Descriptions

Test Use Case	Test Overview	Test Result
Test Case 1: Run system	Input: Run system. Behaviour being tested: User prompts are functional and user interaction is appropriate. Expected output: User is prompted with the inputs needed to activate the environment.	Test Passed
Test Case 2: Choose starting molecule	Input: User chooses a valid starting molecule object. Behaviour being tested: recognition of a valid starting molecule object. Expected output: the class recognizes the valid molecule and stores this information in order to be passed to the environment upon initialisation	Test Passed
Test Case 3: Choose starting molecule	Input: User inputs an invalid starting molecule object. Behaviour being tested: recognition of a invalid molecule object. Expected output: the class recognizes the invalid molecule and returns an appropriate error message to the user	Test Passed
Test Case 4: Choose target molecule	Input: User inputs a valid target molecule. Behaviour being tested: recognition of a valid target molecule object. Expected output: the class recognizes the valid molecule and stores this information in order to be passed to the environment upon initialisation	Test Passed
Test Case 5: Choose target molecule	Input: User inputs an invalid target molecule. Behaviour being tested: recognition of an invalid molecule object. Expected output: the class recognizes the invalid molecule and returns an appropriate error message to the user	Test Passed
Test Case 6: Choose optimisation goal (similarity %)	Input: User inputs a valid optimisation goal (similarity %). Behaviour being tested: recognition of a valid optimisation goal. Expected output: the class recognizes the optimisation goal and stores this information in order to be passed to the environment upon initialisation	Test Passed
Test Case 7: Initialise environment object	Input: User initialises the environment by calling gym.make(), passing in three parameters, including: starting molecule, target molecule and optimisation goal (similarity %). Behaviour being tested: Correct initialisation of environment. Expected output: Environment is initialized with the appropriate observation and action space. The current state of the environment is updated to represent the passed arguments.	Test Passed
Test Case 8: Initialise agent object	Input: User initialises the agent. Behaviour being tested: Correct initialisation of agent. Expected output: Agent is initialized with the appropriate observation and action space. The agent will determine appropriate actions to make based on its decision-making policy.	Test Passed
Test Case 9: Determine chemical validity of molecule	Input: A molecule is provided to the environment. Behaviour being tested: Environment's ability to determine a list of chemically valid modification actions that can be made to the current state. Expected output: The action_space is composed of a valid set of actions that should be provided to the agent for the next iteration.	Test Passed
Test Case 10: Execute action chosen by agent	Input: Environment receives an action chosen by the agent. Behaviour being tested: Environment's ability to successfully set its state to the molecule resulting from the agent's chosen action. Expected output: The environment successfully renders the new state after performing the given action.	Test Passed
Test Case 11: Calculate numerical reward and updated state	Input: Environment receives good action chosen by the agent. Behaviour being tested: Environment's ability to observe whether an action is favourable (the updated state is closer to the optimisation goal). Expected output: Environment returns a positive reward to the agent, which will update the decision-making policy.	Test Passed
Test Case 12: Calculate numerical reward and updated state	Input: Environment receives bad action chosen by the agent. Behaviour being tested: Environment's ability to observe whether an action is unfavourable (the updated state is closer to the optimisation goal). Expected output: Environment returns a negative reward to the agent, which will update the decision-making policy.	Test Passed

TABLE III: Test Case Descriptions