ELIX

GCN-K Molecule Optimization with Monte Carlo Tree Search

Handbook v2.0

March 24, 2020

ELIX

Handbook v2.0

March 24, 2020

GCN-K Molecule Optimization with Monte Carlo Tree Search

We attempt to optimize the molecules generated from the GCN-K adjacency matrices using a Monte Carlo Tree Search approach. Several filtering options and reward functions are available for target optimization including rdKit and OpenBabel energy calculations.

Contents

Introduction

1.1 Project Overview

The present work is an extensions of the GCN-K library, which is a variational autoencoder based graph structure generation model. Instead of applying direct modifications, we postprocess the outcome of the original solution using Monte-Carlo Tree Search. To generate input data, please follow the instructions from the original repository or reach out to the person in charge from Kyoto University.

The input datasets/files expected by the Monte-Carlo model are in JBL format; files pickled with the joblib library. There are 2 possible formats for the dataset file; one with connection information, and one without (both are supported).

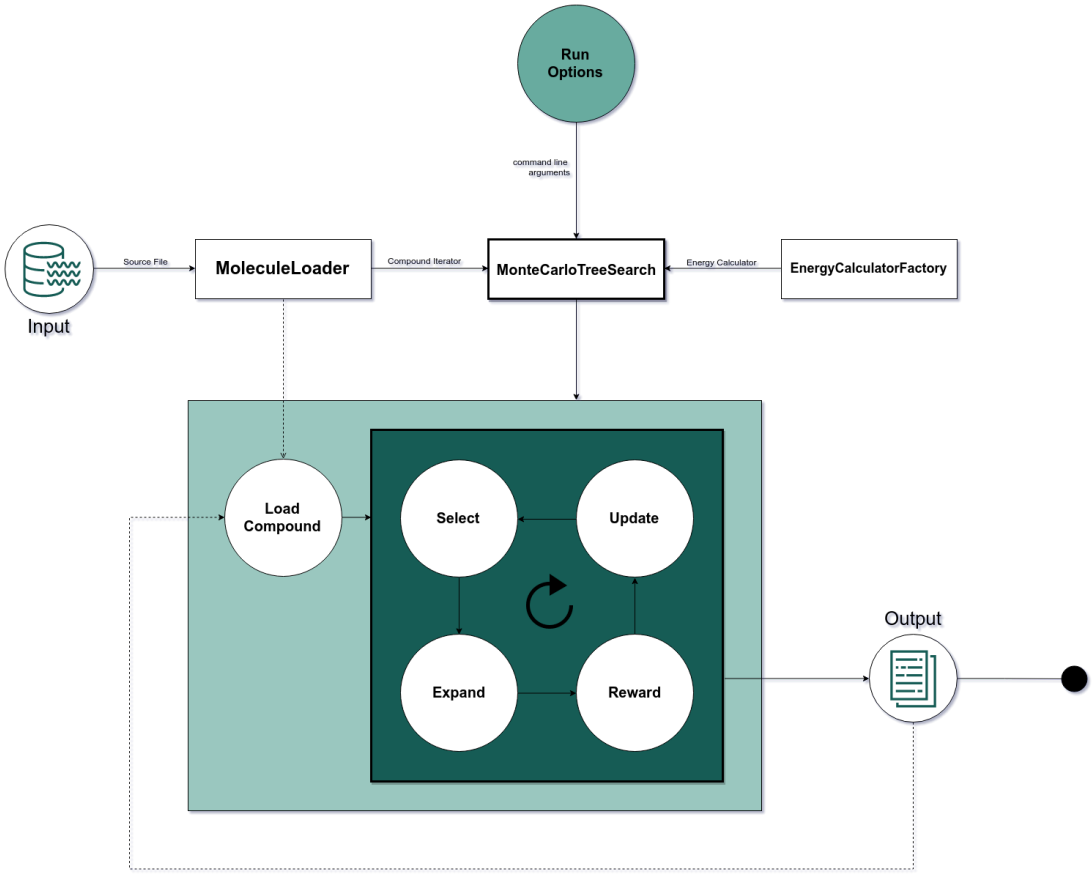


Figure 1.1: Overview of the optimization process.

On one hand, the files contain a feature probability matrix, including: symbol mapping, hybridization, formal charge, and aromaticity, among other things. On the other hand, bond information is represented in adjacency probability matrices.

From the feature probability matrix we only retrieve the list of available atoms and their most likely atomic symbol. From the adjacency probability matrices, we filter out all bonds that are below a certain threshold, which is set as a hyperparameter. Then, we use a Monte-Carlo Tree Search algorithm to optimize the energy of the chemical compound (other reward functions are available).

We initialize the tree with the list of atoms, with no bonds between them. In each iteration, a new bond is added, and the reward is calculated fot the partial compound. This process is repeated either until we run out of bond combinations, or a predefined iteration number is reached. Finally, the best performing node is selected as winner.

Technical Documentation

2.1 Installing Dependencies

Dependencies must be installed before any experiments are executed. The easiest way to install the requirements is by using the conda snapshot provided. If you don't have conda installed locally, you may follow the conda installation guide.

As a first step, create a new environment using the provided snapshot

```
conda env create -f environment.yml
```

Note, this step has to be performed only one time. Once the environment is created and the depencies are installed, you may use the new environment at any time by activating it.

```
conda activate graph_mcts
```

2.2 Folder Structure

The main folder structure is described on the next page. In addition to the main files, archived libraries might contain some optional/trivial directories.

Functionality would not be altered if these are deleted or renamed.

- **bugs**: This folder may includes tests used to reproduce known bugs;
- **data**: This folder usually contains datasets and/or experiment configuration files
- **img**: This folder contains sample generated molecules (the project documentation, README.md, makes use of some of them)

- **lib**: Main codebase;
  - **calculators.py**: Reward/Cost calculation tools
  - **config.py**: Configuration loading, and management tools;
  - **data_providers.py**: Load, prepare and serve molecules from input .jbl files;
  - **data_structures.py**: Resources for compounds, nodes, and trees
  - **filters.py**: Code for filters
  - **helpers.py**: Helper classes (ie: drawing molecules)
  - **models.py**: Monte Carlo Tree Search implementation;
- **scripts**: Additional code, not used by the main model;
- **config.json**: A sample configuration file;
- **environment.yml**: A conda environment snapshot;
- **README.md**: Project documentation;
- **run.py**: Main entry point.

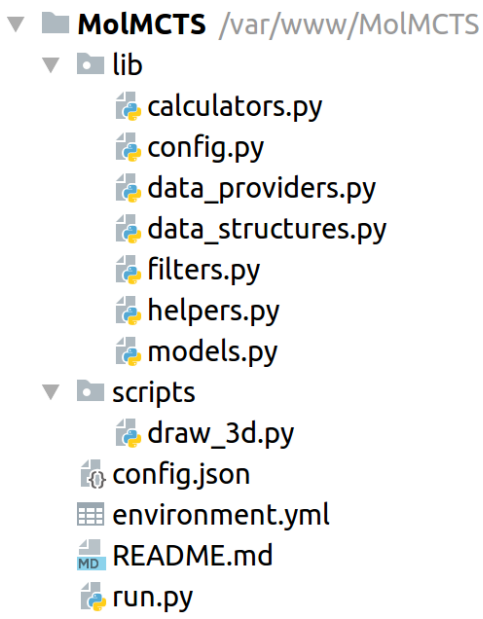


Figure 2.2: Folder structure overview.

2.3 Configuration File

A configuration file is needed to generate molecules. These files specify hyperparameters, dataset paths, and all other run options; it is in JSON format.

The list of available options are:

dataset (mandatory)

Specify the path to the dataset file. It is in JBL format and is the output of the GCN-K model.

generate (10)

Specify how many molecules to generate

minimum_output_depth (20)

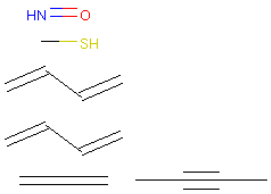
Specify the minimum depth in the tree a node must have to qualify as output. Nodes with depth smaller than this are ignored.

threshold (0.15)

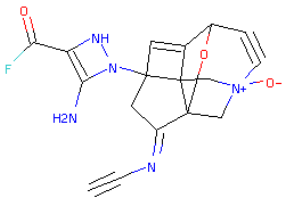
The adjacency matrix provided by GCN-K has a specific structure. For each molecule, each atom has a probability to be of a certain type (ie: atom 1 could have a 70% chance to be C, 20% to be O and 10% to be one of the other 42 types the model allows). Similarly, each atom pair has a certain probability to have a bond between them.

With this threshold, the minimum value required to consider a bond can be controlled. As it is a percentage, it accepts values between 0 and 1.

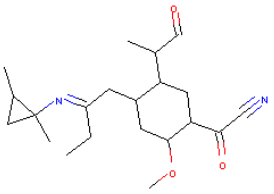
A large threshold will have a very small bond-to-atom ratio, while a very small one will have a very high one. This will result in sparse outputs, with several small molecules in the first case, or very unstable molecules in the second one. Values between 0.10 and 0.15 seem to work best.



figurethreshold=0.25
too big



figurethreshold=0.05
too small



figurethreshold=0.12
just right

monte_carlo_iterations (1000)

Monte Carlo Tree Search is an optimization algorithm that runs for an infinite number of iterations. This parameter is used to specify when to stop.

In our approach, we start from the set of atoms, with no bonds between them and add a new one in each iteration. If the parameter is too small, it might not be enough to add enough bonds to form a large molecule or an optimal one. Generally, the deepest levels of the tree are not good enough because only a few iterations had the opportunity to expand on them.

It is also worth mentioning that this is the parameter which influences execution time the most.

output_type (fittest)

Specify how to select the winning node. 3 options are available:

- **fittest**: Will output the molecule with the smallest reward. But note, smaller molecules tend to be more stable and have smaller energy, often resulting in 2 carbon atoms connected by a single bond when energy calculators are used as reward function. This strategy should always be combined with an adequate "minimum_output_depth" parameter.
- **deepest**: This approach will output the fittest molecule, but only if it is a

node from the deepest level of the tree. The molecules from the deepest levels are usually not stable enough since the states haven't been visited many times.

- **per_level**: Several molecules will be printed, one for the best molecule from each level/depth of the tree. Since the tree can become very deep, it is recommended to use this option along with "minimum_output_depth" as well.

breath_to_depth_ratio (1)

Optimize the exploitation to exploration ratio. Molecule energy is not a good way to select the node to expand since it tends to favor smaller molecules. The best working solution we found is a two-factor pseudo-random one.

As a first step, we perform a weighted random choice for the level to perform the expansion based on the "breath_to_depth_ratio". The higher the value, the more "random" the selection will be. Lower values will result in outputs skewed towards deeper levels. To achieve an opposite effect, a negative value can be used:

As an example, assuming our tree currently has a depth/level of 5, the following "breath_to_depth_ratio" values might produce similar probabilities:

Ratio	Level 1 %	Level 2 %	Level 3 %	Level 4 %	Level 5 %
0.5	0.000047	0.000062	0.003793	0.360444	0.635651
1	0.048671	0.087203	0.160737	0.291216	0.412170
10	0.106983	0.171757	0.180209	0.217732	0.323316
100	0.188933	0.192691	0.201109	0.208459	0.208805
-0.1	0.993638	0.006341	0.000014	0.000006	0
-1	0.410542	0.334045	0.138279	0.111738	0.005394
-10	0.219435	0.219109	0.204048	0.184172	0.173234
-100000	0.200854	0.200463	0.200139	0.200045	0.198496

Step 2 Once a level is selected, we perform a weighted random choice based on the fitness of each node in the level.

reward_calculator (compound_energy_babel_mmff)

The reward calculator option specifies how the cost/reward is calculated. It is important to note that the objective of the model is to minimize this cost. Calculators can be divided into 3 categories:

Basic energy calculators use the energy of the molecule as reward for the algorithm. The smaller the energy, the better. To calculate the energy we need to create a force field, and there are 2 types of implementation for supported force fields: rdkit force fields and open babel force fields. The 7 options available are:

- **compound_energy_rdkit_uff**: The Universal Force Field is an all atom potential which considers only the element, the hybridization and the connectivity (implemented in rdKit)
- **compound_energy_rdkit_mmff**: The Merck Molecular Force Field is similar to the MM3 Force Field (implemented in rdKit)
- **compound_energy_babel_uff**: The same Universal Force Field (implemented in OpenBabel)
- **compound_energy_babel_mmff**: The same Merck Molecular Force Field (implemented in OpenBabel)
- **compound_energy_babel_mmffs**: MMFF94S is a "static" variant of the MMFF force field. (implemented in OpenBabel)
- **compound_energy_babel_gaff**: The Generalized Amber Force Field (implemented in OpenBabel)
- **compound_energy_babel_ghemical**: The Ghemical Force Field (implemented in OpenBabel)

Under *atom-wise energy calculators*, the same options are available, but the energy output is divided by the number of atoms. That is, $E_{atom} = E_{compound}/n_{atom}$:

- **atomwise_energy_rdkit_uff**
- **atomwise_energy_rdkit_mmff**
- **atomwise_energy_babel_uff**

- **atomwise_energy_babel_mmff**
- **atomwise_energy_babel_mmffs**
- **atomwise_energy_babel_gaff**
- **atomwise_energy_babel_ghemical**

All other calculators are *property calculators*.

- **log_p**: The octanol/water partition coefficient. The exact formula is $\max(LogP - 5, 0)$
- **qed**: The Quantitative Estimate of Drug-likeness The exact formula is $1 - qed$
- **sa**: The Synthetic Accessibility score. No modifications were made to the base score, as rdKit already returns a negative value
- **mw**: The Molecular Weight. No modifications were made to the base score.
- **ring_count**: The number of rings present in the molecule The exact formula is $-1 * n_{rings}$

Modifications to the formulas can be easily added in `lib/calculators.py` .

filters (["positive_reward","molecular_weight"])

More than 1 filter can be specified at the same time. If the filter conditions are not met, the compound will not be included in the final node selection. Depending on the reward calculator used, it might be helpful to specify one or more filters.

The list of currently available filters are:

- **non_zero_reward**: Filters out all null/zero rewards
- **positive_reward**: Filters out all rewards below zero
- **molecular_weight**: Filter out all molecules with a molecular weight outside the 300-500 range

Tips for the *"non_zero_reward"* filter.

- Energy calculators can benefit from this filter. Although this is the result we aim for, if the energy is 0 then the compound is likely to be very small.
- Property calculators don't benefit from this filter, except for some specific usecases (ie: we don't want compounds without any rings for example).

Tips for the *"positive_reward"* filter.

- Some energy calculators can return negative values. These are not helpful, so it can be beneficial to use this filter with energy calculators.
- Some rewards have been negated to comply with the minimization process. The "ring_count" for example will always be a negative value, so it will not work with this filter.
- Other calculators are always positive (ie: mw, qed), so the filter is redundant.

Tips for the *"molecular_weight"* filter.

- This filter can be used with any calculator, but note that many early (small depth) molecules will be filtered out. If no results are printed, adjusting some of the other settings like the "minimum_output_depth", "threshold" or the number of "monte_carlo_iterations" might help.

draw (null)

If specified, will draw the molecules to this folder (default: null)

logging (50)

How many logs should be printed. Accepts values between 10 (all messages) and 50 (only very important messages), with increments of 10. What to expect:

- **10**: Everything is logged, including energy calculations and exceptions.
- **20**: All molecules and their energy is logged
- **30+**: Only the result is logged

2.4 Generating Molecules

The following command can be used to generate molecules:

```
python run.py {config_file}
```

For each molecule, the output is printed in JSON format, containing the following fields:

- **smiles**: the SMILES string of the generated molecule
- **depth**: the depth on which the compound is found in the tree (also equal to the number of bonds)
- **score**: The reward of the generated compound

```
(graph_mcts) elix@bash:/var/www/MolMCTS$ python run.py config.json
[
    {
        "smiles": "CCC.CCC(C)C(C)C(=O)C(C)(C)C(C)(CC)CC(C)C",
        "depth": 20,
        "score": 777021354.038241
    }
]
[
    {
        "smiles": "C=O.CCC1C2(C)SC3(OO)C14C(N1C(=O)C1(C)F)C234",
        "depth": 23,
        "score": 665485783.5191505
    }
]
```

Figure 2.3: Visualization of a sample output.

2.5 3D Visualization

A helper script was included which can be used to visualize molecules in 3D. In order to use the script, a PyMol server must be running on the host machine. PyMol is installed in the conda environment, but a license may be required after 30 days.

The PyMol server can be started by running the following command from the terminal:

```
conda activate graph_mcts
pymol -R
```

From a separate terminal, the 3D drawing script can then be called:

```
conda activate graph_mcts
python -m scripts.draw_3d {input_file} {output_folder}
```

Where:

- **{input_file}**: Is a file containing SMILES strings (one per line);
- **{output_folder}**: is the folder where the generated images are saved to.