ChemPer Documentation

Release 1.0.0+3.gcc82c7d

Caitlin C. Bannan

Apr 30, 2020

CONTENTS:

CHAPTER

ONE

INTRODUCTION

`chemper` contains a variety of tools that will be useful in automating the process of chemical perception for the new SMIRKS Native Open Force Field (SMIRNOFF) format as a part of the Open Force Field Consortium *[1]*.

This idea originated from the tools SMARTY and SMIRKY which were designed to use an automated monte carlo algorithm to sample the chemical perception used in existing force fields. SMARTY samples SMARTS patterns corresponding to traditional atom types and was tested compared to the parm99/parm@frosst force field. SMIRKY is an extension of SMARTY created to sample SMIRKS patterns corresponding to SMIRNOFF parameter types (nonbonded, bond, angle, and proper and improper torsions). SMIRKY was tested against our own smirnoff99Frosst

One of the most important lessons learned while testing SMARTY and SMIRKY is that the combinatorial problem in SMIRKS space is very large. These tools currently use very naive moves in SMIRKS space chosing atom or bond decorators to add to a pattern at complete random. This wastes a signficant amount of time making chemically insensible moves. One of the take away conclusions on that project was that future chemical perception sampling tools would need to take atom and bond information from input molecules in order to be feasible *[2]*.

We developed `chemper` based on the knowledge of the SMARTY project outcomes. The goal here is to take clustered molecular subgraphs and generate SMIRKS patterns. These tools will use information stored in the atoms and bonds of a molecule to drive choices about SMIRKS decorators. Then will automatically generate reasonable SMIRKS patterns matching clustering of molecular subgraphs.

Currently, `chemper` provides modules for generating SMIRKS patterns from molecule objects and specified atoms. The final product will be capable of clustering molecular fragments based on reference data and then assigning SMIRKS patterns for each of those clusters. See *Installing chemper*.

1.1 Contributors

- Caitlin C. Bannan (UCI)
- Jessica Maat (UCI)
- David L. Mobley (UCI)

1.2 Acknowledgments

CCB is funded by a fellowship from The Molecular Sciences Software Institute under NSF grant ACI-1547580.

1.3 References

1. D. Mobley et al. bioRxiv 2018, 286542. doi.org/10.1101/286542
2. C. Zanette and C.C. Bannan et al. chemRxiv 2018, doi.org/10.26434/chemrxiv.6230627.v1

CHAPTER

TWO

INSTALLING CHEMPER

The only way to install `chemper` right now is to clone or download the GitHub repo. When we're ready to release version 1.0.0 we'll also make it conda installable.

2.1 Prerequisites

We currently test with Python 3.5, though we expect anything 3.5+ should work.

This is a python tool kit with a few dependencies. We recommend installing miniconda. Then you can create an environment with the following commands:

```
conda create -n [my env name] python=3.5 numpy networkx pytest
source activate [my env name]
```

This command will install all dependencies besides a toolkit for cheminformatics or storing of molecule information. We seek to keep this tool independent of cheminformatics toolkit, but currently only support RDKit and OpenEye Toolkits. If you wish to add support please feel free to submit a pull request. Make sure one of these tool kits is installed in your environment before installing chemper.

2.1.1 RDKit environment

Conda installation according to RDKit documentation:

```
conda install -c rdkit rdkit
```

2.1.2 OpenEye environment

Conda installation according to OpenEye documentation

```
cond install -c openeye openeye-toolkits
```

2.2 Installation

Hopefully chemper will be conda installable in the near future, but for now the best option is to download or clone from GitHub and then from inside the `chemper` directory install with the command:

```
pip install -e .
```

CHAPTER

THREE

CHEMPER API

ChemPer is open source. If you have suggestions or concerns please add them to our issue tracker. Documentation for contributing to this work will be available soon. Below is an outline of tools provided in ChemPer. See *examples* for more details on how to use these tools.

3.1 mol_toolkits

As noted in *installation*, we seek to keep chemper independent of the cheminformatics toolkit. *mol_toolkits* is created to keep all code dependent on the toolkit installed. It can create molecules from an RDK or OE molecule object or from a SMILES string. It includes a variety of functions for extracting information about atoms, bonds, and molecules. Also included here are SMIRKS pattern searches.

3.1.1 Mol Toolkits

chemper wraps around existing cheminformatics packages to extract information about atoms and bonds stored in molecules. These are accessed through *chemper.mol_toolks.mol_toolkit* which accesses the source code for only the installed cheminformatics package. Right now we support OpenEye Toolkits

OpenEye

Following the template outlined, chemper Mol, Atom, and Bond objects can be created from OpenEye tools.

cp_openeye.py

Cheminformatics tools using OpenEye Toolkits

The classes provided here follow the structure in adapters. This is a wrapper allowing our actual package to use openeye toolkits

class chemper.mol_toolkits.cp_openeye.**Atom**(*atom*)

atomic_number()

Returns **atomic_number** – atomic number for the atom

Return type int

connectivity()

Returns **connectivity** – connectivity or total number of bonds (regardless of order) around the atom

Return type int

degree()

Returns **degree** – degree or number of explicit bond orders around the atom

Return type int

formal_charge()

Returns **formal_charge** – the atom's formal charge

Return type int

get_bonds()

Returns **bonds** – Bonds connected to this atom

Return type list[ChemPer Bonds]

get_index()

Returns **index** – atom index in its molecule

Return type int

get_molecule()

Extracts the parent molecule this atom is from.

Returns **mol** – Molecule this atom is stored in

Return type ChemPer Mol

get_neighbors()

Returns **neighbors** – Atoms that are one bond away from this atom

Return type list[ChemPer Atoms]

hydrogen_count()

Returns **H_count** – total number of hydrogen atoms connected to this Atom

Return type int

is_aromatic()

Returns **is_aromatic** – True if the atom is aromatic otherwise False

Return type boolean

is_connected_to(*atom2*)

Parameters **atom2** (*ChemPer Atom*) – Atom to check if it is bonded to this atom

Returns **connected** – True if atom2 is a bonded to atom1

Return type boolean

min_ring_size()

Returns **min_ring_size** – size of the smallest ring this atom is a part of

Return type int

ring_connectivity()

Returns **ring_connectivity** – number of bonds on the atom that are a part of a ring

Return type int

valence()

Returns **valence** – the atoms valence (equivalent to degree when all bonds are explicit)

Return type int

class chemper.mol_toolkits.cp_openeye.**Bond**(*bond*)

get_atoms()

Returns **atoms** – The two atoms connected by this bond

Return type list[ChemPer Atoms]

get_index()

Returns **index** – index of this bond in its parent molecule

Return type int

get_molecule()

Extracts the parent molecule this bond is from

Returns **mol** – Molecule this bond is stored in

Return type ChemPer Mol

get_order()

Returns **order** – This is the absolute order, returns 1.5 if bond is aromatic

Return type int or float

is_aromatic()

Returns **is_aromatic** – True if it is an aromatic bond

Return type boolean

is_double()

Returns **is_double** – True if it is a double bond

Return type boolean

is_ring()

Returns **is_ring** – True if bond is a part of a ring, otherwise False

Return type boolean

is_single()

Returns **is_single** – True if it is a single bond

Return type boolean

is_triple()

Returns **is_triple** – True if it is a triple bond

Return type boolean

class chemper.mol_toolkits.cp_openeye.**Mol**(*mol*)

classmethod from_smiles(*smiles*)

Creates a ChemPer Mol form a SMILES string

Parameters **smiles** (*str*) – SMILES used to create molecule with wrapped toolkit

Returns **Mol**

Return type ChemPer Mol

get_atom_by_index(*idx*)

Parameters **idx** (*int*) – atom index

Returns **atom** – atom with index idx

Return type ChemPer Atom

get_atoms()

Returns **atom_list** – list of all atoms in the molecule

Return type list[ChemPer Atoms]

get_bond_by_atoms(*atom1*, *atom2*)

Finds a bond between two atoms

Parameters

- **atom1** (*ChemPer Atom*) –
- **atom2** (*ChemPer Atom*) –

Returns **bond** – If atoms are connected returns bond otherwise None

Return type ChemPer Bond or None

get_bond_by_index(*idx*)

Parameters **idx** (*int*) – bond index

Returns **bond** – bond with index idx

Return type ChemPer Bond

get_bonds()

Returns **bond_list** – list of all bonds in the molecule

Return type list[ChemPer Bonds]

get_smiles()

Returns **smiles** – SMILES string for the molecule

Return type str

set_aromaticity_mdl()

Sets the aromaticity flags in this molecule to use the MDL model

smirks_search(*smirks*)

Performs a substructure search on the molecule with the provided SMIRKS pattern. Note - this function expects SMIRKS patterns with indexed atoms that is with :n for at least some atoms.

Parameters **smirks** (*str*) – SMIRKS pattern with indexed atoms (:n)

Returns **matches** – match dictionaries have the form {smirks index: atom index}

Return type list[match dictionary]

chemper.mol_toolkits.cp_openeye.**mols_from_file**(*mol_file*)

Parses a standard molecule file into ChemPer molecules using OpenEye toolkits

Parameters **mol_file** (*str*) – relative or full path to molecule containing the molecule file that is accessible from the current working directory

Returns **mols** – list of molecules in the mol2 file as ChemPer Mols

Return type list[ChemPer Mol]

RDKit

Following the template outlined, `chemper` Mol, Atom, and Bond objects can be created from RDKit Chem module.

cp_rdk.py

Cheminformatics tools using RDKit

The classes provided here follow the structure in adapters. This is a wrapper allowing our actual package to use RDKit

class chemper.mol_toolkits.cp_rdk.**Atom**(*atom*)

atomic_number()

Returns **atomic_number** – atomic number for the atom

Return type int

connectivity()

Returns **connectivity** – connectivity or total number of bonds (regardless of order) around the atom

Return type int

degree()

Returns **degree** – degree or number of explicit bond orders around the atom

Return type int

formal_charge()

Returns **formal_charge** – the atom's formal charge

Return type int

get_bonds()

Returns **bonds** – Bonds connected to this atom

Return type list[ChemPer Bonds]

get_index()

Returns **index** – atom index in its molecule

Return type int

get_molecule()

Extracts the parent molecule this atom is from.

Returns **mol** – Molecule this atom is stored in

Return type ChemPer Mol

get_neighbors()

Returns **neighbors** – Atoms that are one bond away from this atom

Return type list[ChemPer Atoms]

hydrogen_count()

Returns **H_count** – total number of hydrogen atoms connected to this Atom

Return type int

is_aromatic()

Returns **is_aromatic** – True if the atom is aromatic otherwise False

Return type boolean

is_connected_to(*atom2*)

Parameters **atom2** (*ChemPer Atom*) – Atom to check if it is bonded to this atom

Returns **connected** – True if atom2 is a bonded to atom1

Return type boolean

min_ring_size()

Returns **min_ring_size** – size of the smallest ring this atom is a part of

Return type int

ring_connectivity()

Returns **ring_connectivity** – number of bonds on the atom that are a part of a ring

Return type int

valence()

Returns **valence** – the atoms valence (equivalent to degree when all bonds are explicit)

Return type int

class chemper.mol_toolkits.cp_rdk.**Bond**(*bond*)

get_atoms()

Returns **atoms** – The two atoms connected by this bond

Return type list[ChemPer Atoms]

get_index()

Returns **index** – index of this bond in its parent molecule

Return type int

get_molecule()

Extracts the parent molecule this bond is from

Returns **mol** – Molecule this bond is stored in

Return type ChemPer Mol

get_order()

Returns **order** – This is the absolute order, returns 1.5 if bond is aromatic

Return type int or float

is_aromatic()

Returns **is_aromatic** – True if it is an aromatic bond

Return type boolean

is_double()

Returns **is_double** – True if it is a double bond

Return type boolean

is_ring()

Returns **is_ring** – True if bond is a part of a ring, otherwise False

Return type boolean

is_single()

Returns **is_single** – True if it is a single bond

Return type boolean

is_triple()

Returns **is_triple** – True if it is a triple bond

Return type boolean

class chemper.mol_toolkits.cp_rdk.**Mol**(*mol*)

classmethod from_smiles(*smiles*)
Creates a ChemPer Mol form a SMILES string

Parameters **smiles** (*str*) – SMILES used to create molecule with wrapped toolkit

Returns **Mol**

Return type ChemPer Mol

get_atom_by_index(*idx*)

Parameters **idx** (*int*) – atom index

Returns **atom** – atom with index idx

Return type ChemPer Atom

get_atoms()

Returns **atom_list** – list of all atoms in the molecule

Return type list[ChemPer Atoms]

get_bond_by_atoms(*atom1*, *atom2*)
Finds a bond between two atoms

Parameters

- **atom1** (*ChemPer Atom*) –
- **atom2** (*ChemPer Atom*) –

Returns **bond** – If atoms are connected returns bond otherwise None

Return type ChemPer Bond or None

get_bond_by_index(*idx*)

Parameters **idx** (*int*) – bond index

Returns **bond** – bond with index idx

Return type ChemPer Bond

get_bonds()

Returns **bond_list** – list of all bonds in the molecule

Return type list[ChemPer Bonds]

get_smiles()

Returns **smiles** – SMILES string for the molecule

Return type str

set_aromaticity_mdl()
Sets the aromaticity flags in this molecule to use the MDL model

smirks_search(*smirks*)
Performs a substructure search on the molecule with the provided SMIRKS pattern. Note - this function expects SMIRKS patterns with indexed atoms that is with :n for at least some atoms.

Parameters **smirks** (*str*) – SMIRKS pattern with indexed atoms (:n)

Returns **matches** – match dictionaries have the form {smirks index: atom index}

Return type list[match dictionary]

chemper.mol_toolkits.cp_rdk.**mols_from_mol2**(*mol2_file*)
Parses a mol2 file into ChemPer molecules using RDKit

This is a hack for separating mol2 files taken from a Source Forge discussion here: https://www.mail-archive.com/rdkit-discuss@lists.sourceforge.net/msg01510.html It splits up a mol2 file into blocks and then uses RDKit to parse those blocks

Parameters **mol2_file** (*str*) – relative or absolute path to a mol2 file you want to parse accessible form the current directory

Returns **mols** – list of molecules in the mol2 file as ChemPer molecules

Return type list[ChemPer Mol]

These and any future packge support follow the template laid out as adapters.

adapters.py

This script contains adapters or the structure for molecules, atoms, and bonds. Our chemical perception code is designed to be independent of the users cheminformatics packages. For each cheminformatics package we support we will provide classes following the structure in these adapters.

class chemper.mol_toolkits.adapters.**AtomAdapter**
This is a ChemPer wrapper for an atom from one of the cheminformatics toolkits. ChemPer Atoms are initiated from the reference package object. Currently we support OpenEye and RDKit toolkits.

atom

Type Atom from reference toolkit

abstract atomic_number()

Returns **atomic_number** – atomic number for the atom

Return type int

abstract connectivity()

Returns **connectivity** – connectivity or total number of bonds (regardless of order) around the atom

Return type int

abstract degree()

Returns **degree** – degree or number of explicit bond orders around the atom

Return type int

abstract formal_charge()

Returns **formal_charge** – the atom's formal charge

Return type int

abstract get_bonds()

Returns **bonds** – Bonds connected to this atom

Return type list[ChemPer Bonds]

abstract get_index()

Returns **index** – atom index in its molecule

Return type int

abstract get_molecule()

Extracts the parent molecule this atom is from.

Returns **mol** – Molecule this atom is stored in

Return type ChemPer Mol

abstract get_neighbors()

Returns **neighbors** – Atoms that are one bond away from this atom

Return type list[ChemPer Atoms]

abstract hydrogen_count()

Returns **H_count** – total number of hydrogen atoms connected to this Atom

Return type int

abstract is_aromatic()

Returns **is_aromatic** – True if the atom is aromatic otherwise False

Return type boolean

abstract is_connected_to(*atom2*)

Parameters **atom2** (*ChemPer Atom*) – Atom to check if it is bonded to this atom

Returns **connected** – True if atom2 is a bonded to atom1

Return type boolean

abstract min_ring_size()

Returns **min_ring_size** – size of the smallest ring this atom is a part of

Return type int

abstract ring_connectivity()

Returns **ring_connectivity** – number of bonds on the atom that are a part of a ring

Return type int

abstract valence()

Returns **valence** – the atoms valence (equivalent to degree when all bonds are explicit)

Return type int

class chemper.mol_toolkits.adapters.**BondAdapter**

This is a ChemPer wrapper for a bond from one of the cheminformatics toolkits. ChemPer Bonds are initiated from the reference package object. Currently we support OpenEye and RDKit toolkits.

bond

Type Bond from reference class

abstract get_atoms()

Returns **atoms** – The two atoms connected by this bond

Return type list[ChemPer Atoms]

abstract get_index()

Returns **index** – index of this bond in its parent molecule

Return type int

abstract get_molecule()

Extracts the parent molecule this bond is from

Returns **mol** – Molecule this bond is stored in

Return type ChemPer Mol

abstract get_order()

Returns **order** – This is the absolute order, returns 1.5 if bond is aromatic

Return type int or float

abstract is_aromatic()

Returns **is_aromatic** – True if it is an aromatic bond

Return type boolean

abstract is_double()

Returns **is_double** – True if it is a double bond

Return type boolean

abstract is_ring()

Returns **is_ring** – True if bond is a part of a ring, otherwise False

Return type boolean

abstract is_single()

Returns **is_single** – True if it is a single bond

Return type boolean

abstract is_triple()

Returns **is_triple** – True if it is a triple bond

Return type boolean

class chemper.mol_toolkits.adapters.**MolAdapter**

This is a ChemPer wrapper for a molecule from one of the cheminformatics toolkits. ChemPer molecules are initiated from the reference package molecule object. Currently we support OpenEye and RDKit toolkits.

mol
Mol object from the reference cheminformatics toolkit

Type toolkit Mol

abstract classmethod from_smiles(*smiles*)
Creates a ChemPer Mol form a SMILES string

Parameters **smiles** (*str*) – SMILES used to create molecule with wrapped toolkit

Returns **Mol**

Return type ChemPer Mol

abstract get_atom_by_index(*idx*)

Parameters **idx** (*int*) – atom index

Returns **atom** – atom with index idx

Return type ChemPer Atom

abstract get_atoms()

Returns **atom_list** – list of all atoms in the molecule

Return type list[ChemPer Atoms]

abstract get_bond_by_atoms(*atom1*, *atom2*)
Finds a bond between two atoms

Parameters

- **atom1** (*ChemPer Atom*) –
- **atom2** (*ChemPer Atom*) –

Returns **bond** – If atoms are connected returns bond otherwise None

Return type ChemPer Bond or None

abstract get_bond_by_index(*idx*)

Parameters **idx** (*int*) – bond index

Returns **bond** – bond with index idx

Return type ChemPer Bond

abstract get_bonds()

Returns **bond_list** – list of all bonds in the molecule

Return type list[ChemPer Bonds]

abstract get_smiles()

Returns **smiles** – SMILES string for the molecule

Return type str

abstract set_aromaticity_mdl()
Sets the aromaticity flags in this molecule to use the MDL model

abstract smirks_search(*smirks*)
Performs a substructure search on the molecule with the provided SMIRKS pattern. Note - this function expects SMIRKS patterns with indexed atoms that is with :n for at least some atoms.

Parameters **smirks** (*str*) – SMIRKS pattern with indexed atoms (:n)

Returns **matches** – match dictionaries have the form {smirks index: atom index}

Return type list[match dictionary]

3.2 chemperGraph

The goal of *chemperGraph* was to create an example of how you could create a SMIRKS pattern from a molecule and set of atom indices. Creating SMIRKS for one molecule may not be useful for sampling chemical perception in the long run. However, it is a tool that did not previously exist to the best of our knowledge. For a detailed example, see *single_mol_smirks*.

Here is a brief usage example for using `SingleGraph` to create a SMIRKS pattern. In this case, we want to create a pattern for the carbon-carbon bond in ethene. The carbon atoms have the indices 0 and 1 in the molecule, specified using the tuple *(0,1)*. These atoms are assigned to SMIRKS indices *:1* and *:2* respectively. In this example, we also include atoms up to one bond away from the indexed atoms by specifying the variable `layers` be set to *1*.

```
from chemper.mol_toolkits.mol_toolkit import Mol
from chemper.graphs.single_graph import SingleGraph

# make molecule from SMILES
smiles = 'CCO'
mol = Mol.from_smiles(smiles)

tagged = (0,1)   # atom in carbon-carbon bond
# try multiple options for layers
for layers in [0, 1, 'all']:
    # make graph and extract SMIRKS
    graph = SingleGraph(mol, tagged, layers)
    print(graph.as_smirks())   # complex SMIRKS with all decorators are the default
    print(graph.as_smirks(compress=True))   # compressed SMIRKS have only atomic␣
↪numbers
```

3.2.1 SingleGraph

`SingleGraph``s convert a molecule into a SMIRKS pattern by converting information about the atoms and bonds. The ``SingleGraph` stores only one molecule. It can convert only certain atoms or the *entire* molecule into a SMIRKS pattern. This direction of molecule to SMIRKS string is new as far as the authors know with chemper.

3.3 ClusterGraph

The goal of *ClusterGraph* is to store all information about the atoms and bonds that could be in a SMIRKS pattern. These are created assuming you already have a clustered set of molecular fragments. Our primary goal is to determine chemical perception for force field parameterization. We imagine parameters for each molecule (for example bond lengths and force constants) could be clustered by fragment. Then we could generate a hierarchical list of SMIRKS patterns that maintain those clusters for typing purposes. However, you could imagine other reasons for wanting to store how you clustered groups of atoms – for example, using atom or bond types in a machine learning model.

For more detailed examples and illustration of how this works see *smirks_from_molecules* example. Below is a brief example showing the SMIRKS for the bond between two carbon atoms in propane and pentane.

```
from chemper.mol_toolkits.mol_toolkit import Mol
from chemper.graphs.cluster_graph import ClusterGraph

# make molecules from smiles
mols = [
    Mol.from_smiles('CCO'),
    Mol.from_smiles('CC=C')
]
# identify atoms for tagging # one set of atoms in second molecule
tagged = [[ (0,1) ],  # one set of atoms in first molecule
          [ (0,1) ]   # one set of atoms in second molecule
          ]
# try multiple options for layers
for layers in [0,1,'all']:
    # make graph
    graph = ClusterGraph(mols, tagged, layers)
    print(graph.as_smirks())   # complex is the default output
    print(graph.as_smirks(compress=True))   # and's common decorators to the end of␣
↪each atom
```

The idea with ClusterGraph objects is that they store all possible decorator information for each atom. In this case the SMIRKS indexed atoms for propane (mol1) are one of the terminal and the middle carbons. In pentane (mol2) however the first atom can be a terminal or middle of the chain carbon atom. This changes the number of hydrogen atoms (`Hn` decorator) on the carbon, thus there are two possible SMIRKS patterns for atom `:1` `#6AH2X4x0r0+0` or (indicated by the ",") `#6AH3X4x0r0+0`. But, atom *:2* only has one possibility *#6AH2X4x0r0+0*.

3.3.1 Cluster Graphs

`ClusterGraph` is an expansion of the initial `SingleGraph`. They can store information about multiple atoms and bonds simultaneously. This is a starting example for how to use `chemper`'s `ClusterGraph` class to create SMIRKS patterns from clusters of molecular graphs.

cluster_graph.py

ClusterGraph are a class for tracking all possible smirks decorators in a group (or cluster) of molecular fragments. Moving forward these will be used to find the minimum number of smirks decorators that are required to have a set of smirks patterns that maintain a given clustering of fragments.

class chemper.graphs.cluster_graph.**ClusterGraph**(*mols=None*, *smirks_atoms_lists=None*, *layers=0*)

ChemPerGraphs are a graph based class for storing atom and bond information. They use the chemper.mol_toolkits Atoms, Bonds, and Mols

class AtomStorage(*atoms=None*, *label=None*)

AtomStorage tracks information about an atom

add_atom(*atom*)

Expand current AtomStorage by adding information about a new ChemPer Atom

Parameters **atom** (*ChemPer Atom*) –

as_smirks(*compress=False*)

Parameters **compress** (*boolean*) – should decorators common to all sets be combined for example '#6X4,#7X3;+0!r...'

Returns **smirks** – how this atom would be represented in a SMIRKS string with the minimal combination of SMIRKS decorators

Return type str

compare_atom(*atom*)

Compares decorators in this AtomStorage with the provided ChemPer atom. The decorators are compared separately and the highest score is returned. For example, if this storage had two sets of decorators

- #7H1X3x0!r+0A
- #6H1X4x0!r+0A

and the input atom would have the decorators:

- #6H1X3x2!r+0a

The score is calculated by finding the number of decorators in common which would be

- **#7H1X3x0!r+0A and #6H1X3x2r6+0a** have 3 decorators in common (H1,X3,+0)
- **#6H1X4x0!r+0A and #6H1X3x2r6+0a** also have 3 decorators in common (#6, H1, +0)

However, we weight atoms with the same atomic number as more similar by dividing the score by 10 if the atomic numbers do not agree. Therefore the final scores will be:

- 0.3 for #7H1X3x0!r+0A
- 3 for #6H1X4x0!r+0A

The highest score for any set of decorators is returned so 3 is the returned score in this example.

Parameters **atom** (*ChemPer Atom*) –

Returns **score** – A score describing how similar the input atom is to any set of decorators currently in this storage, based on its SMIRKS decorators. This score ranges from 0 to 7. 7 comes from the number of decorators on any atom, if this atom matches perfectly with one of the current decorator sets then 7 decorators agree.However, if the atomic number doesn't agree, then that set of decorators is considered less ideal, thus if the atomic numbers don't agree, then the score is given by the number other decorators divided by 10. If the current storage is empty, then the score is given as 7 since any atom matches a wildcard atom.

Return type float

make_atom_decorators(*atom*)

extract information from a ChemPer Atom that would be useful in a smirks

Parameters **atom** (*ChemPer atom object*) –

Returns **decorators** – tuple of all possible decorators for this atom

Return type tuple of str

class BondStorage(*bonds=None*, *label=None*)

BondStorage tracks information about a bond

add_bond(*bond*)

Expand current BondStorage by adding information about a new ChemPer Bond

Parameters **bond** (*ChemPer Bond*) –

as_smirks()

Returns **smirks** – how this bond would be represented in a SMIRKS string using only the required number of

Return type str

compare_bond(*bond*)

Parameters **bond** (*ChemPer Bond*) – bond you want to compare to the current storage

Returns

score – A score describing how similar the input bond is to any set of decorators currently in this storage, based on its SMIRKS decorators.

1 for the bond order + 1 base on if this is a ring bond

Return type int (0,1,2)

add_mol(*input_mol*, *smirks_atoms_list*)

Expand the information in this graph by adding a new molecule

Parameters

- **input_mol** (*ChemPer Mol*) –
- **smirks_atoms_list** (*list of tuples*) – This is a list of tuples with atom indices [(indices), ...]

as_smirks (*compress=False*)

Parameters **compress** (*boolean*) – returns the shorter version of atom SMIRKS patterns that is atoms have decorators "anded" to the end rather than listed in each set that are OR'd together. For example "[#6AH2X3x0!r+0,#6AH1X3x0!r+0:1]-;!@[#1AH0X1x0!r+0]" compresses to: "[#6H2,#6H1;AX3x0!r+0:1]-;!@[#1AH0X1x0!r+0]"

Returns **SMIRKS** – a SMIRKS string matching the exact atom and bond information stored

Return type str

find_pairs (*atoms_and_bonds*, *storages*)

Find pairs is used to determine which current AtomStorage from storages atoms should be paired with. This function takes advantage of the maximum scoring function in networkx to find the pairing with the highest "score". Scores are determined using functions in the atom and bond storage objects that compare those storages to the new atom or bond.

If there are less atoms than storages then the atoms with the lowest pair are assigned a None pairing.

Parameters

- **atoms_and_bonds** (*list of tuples in form (ChemPer Atom, ChemPer Bond, ..)*) –
- **storages** (*list of tuples in form (AtomStorage, BondStorage, ..)*) –
- **can be of any length as long as they are the same, so for example, in** (*Tuples*) –
- **bond you might only care about the outer atoms for comparison so you would compare** (*a*) –
- **and (atom2,) with (atom_storage1,) and (atom_storage2,)** ((*atom1,*)) –
- **in a torsion, you might want the atoms and bonds for each outer bond** (*However,*) –
- **in that case you would compare** (*so*) –
- **bond1, atom2) and (atom4, bond3, atom3)** ((*atom1,*) –
- **the corresponding storage objects.** (*with*) –

Returns

pairs – pairs of atoms and storage objects that are most similar, these lists always come in the form (all atom/bonds, all storage objects) for the bond example above you might get [[atom1, storage1], [atom2, storage2]] for the torsion example you might get [[atom4, bond4, atom3, atom_storage1, bond_storage1, atom_storage2],

[atom1, bond1, atom2, atom_storage4, bond_storage3, atom_storage3]

Return type list of lists

get_symmetry_funct (*sym_label*)

Determine the symmetry function that should be used when adding atoms to this graph.

For example, imagine a user is trying to make a SMIRKS for all of the C-H bonds in methane. In most toolkits the index for the carbon is 0 and the hydrogens are 1,2,3,4. The final SMIRKS should have the form [#6AH4X4x0!r+0:1]-;!@[#1AH0X1x0!r+0] no matter what order the atoms are input into ClusterGraph. So if the user provides (0,1), (0,2), (3,0), (4,0) ClusterGraph should figure out that the carbons in (3,0) and (4,0) should be in the atom index :1 place like they were in the first set of atoms.

Bond atoms in (1,2) or (2,1) are symmetric, for angles its (1,2,3) or (3,2,1) for proper torsions (1,2,3,4) or (4,3,2,1) and for improper torsions (1,2,3,4), (3,2,1,4), (4,2,1,3). For any other fragment type the atoms will be added to the graph in the order they are provided since the symmetry function is unknown.

TODO: In theory you could generalize this for generic linear fragments # where those with an odd number of atoms behave like angles and an # even number behave like proper torsions, however I think that is # going to be outside the scope of ChemPer for the foreseeable future.

Parameters **sym_label** (*str or None*) – type of symmetry, options which will change the way symmetry is handled in the graph are "bond", "angle", "ProperTorsion", and "ImproperTorsion"

Returns **symmetry_funct** – returns the function that should be used to handle the appropriate symmetry

Return type function

3.4 SMIRKSifier

Let's assume you have a few clusters of fragments that you want assigned the same force field parameter. For example, you could have clusters of carbon-carbon bonds based on the type of bond between them (single, double, etc). In this case ChemPer would use the SMIRKSifier to generate a hierarchical list of SMIRKS patterns for those clusters. This process creates SMIRKS using ClusterGraph and then takes a stochastic approach to removing unnecessary decorators. See the *general_smirks_for_clusters* example for how this process could be applied to different bonding parameters.

```
from chemper.mol_toolkits.mol_toolkit import Mol
from chemper.smirksify import SMIRKSifier, print_smirks

# make molecules from smiles
mols = [
    Mol.from_smiles('CCO'),
    Mol.from_smiles('CC=C')
]
# make clusters for each of the 6 bond types:
# carbon-carbon single bond 1 ethanol, 1 propene
cc_single = ('cc_single', [ [(0,1)],  [ (0,1) ] ]  )
# carbon-carbon double bond 0 ethanol, 1 propene
cc_double = ('cc_double', [ [ ], [ (1,2) ] ]  )
# carbon-oxygen bond 1 ethanol, 0 propene
co = ('co', [ [ (1,2) ], [ ] ]  )
# hydrogen-tetrahedral carbon 5 ethanol, 3 propene
hc_tet = ('hc_tet', [ [ (0,3), (0,4), (0,5), (1,6), (1,7) ],  [ (0,3), (0,4), (0,5)]␣
↪]  )
# hydrogen-planar carbon bond 0 ethanol, 3 propene
hc_plan = ('hc_plan', [ [ ], [ (1,6), (2,7), (2,8)] ])
# hydrogen-oxygen bond 1 ethanol, 0 propene
ho = ('ho', [ [ (2,8) ],  [ ] ] )

# initiate SMIRKSifier with default max_layers = 5 and verbose = True
fier = SMIRKSifier(mols, [cc_single, cc_double, co, hc_tet, hc_plan, ho])
```

(continues on next page)

(continued from previous page)

```
# print initial SMIRKS
print_smirks(fier.current_smirks)
# Reduce SMIRKS with default 1000 iterations
fier.reduce()
# print final SMIRKS
print_smirks(fier.current_smirks)
```

language python

3.4.1 SMIRKSifier

`SMIRKSifier` is a first attempt at ChemPer's ultimate goal of creating hierarchical lists of SMIRKS patterns. When provided with clusters of molecular fragments, the `SMIRKSifier` generates an ordered list of SMIRKS patterns.

smirksify.py

In this script, we start with a set of clustered molecular fragments with specified indexed atoms as those you would use to build a ClusterGraph. We then build cluster Graphs to create the initial SMIRKS patterns and check that the generated SMIRKS patterns retain the typing from the input cluster. Next we run a series of iterations removing SMIRKS decorators. If this "move" doesn't change the way the molecules are typed then the change is accepted.

This class takes inspiration from the tool SMIRKY previously published by the Open Force Field Initiative: github.com/openforcefield/smarty

In theory, it is possible this process of removing decorators could be more systematic/deterministic, however this is a first approach to see if extracted SMIRKS patterns can do better than SMIRKY. Also, this approach will be more general since the input clusters do not rely on a reference force field.

exception chemper.smirksify.**ClusteringError**(*msg*)
Exception for when the SMIRKSifier is unable to create a list of SMIRKS to maintain the input clusters.

class chemper.smirksify.**Reducer**(*smirks_list*, *mols*, *verbose=False*)
Reducer starts with any list of SMIRKS and removes unnecessary decorators while maintaining typing on input molecules. This was created to be used as a part of the SMIRKSifier.reduce function. However, if you have complex SMIRKS and a list of molecules you can also reduce those patterns independently.

current_smirks
current SMIRKS patterns in the form (label, smirks)

Type list of tuples

mols
molecules being used to reduce the input SMIRKS

Type list of chemper molecules

cluster_dict
Dictionary specifying typing using current SMIRKS in the form: {mol_idx:

{ (tuple of atom indices): label } }

Type dictionary

remove_all_bases(*input_ors*)
convert all bases to [*] i.e. [(#6, [X4, +0]), (#7, [X3])] –> [(*, [X4, +0]), (*, [X3])]

Parameters **input_ors** (*list of two tuples*) – OR decorators in the form from ChemicalEnvironments that is [(base, [decorators,]), ...]

Returns **new_ors** – New OR decorators

Return type list of two tuples

remove_all_dec_type(*input_ors*)
remove all decorators of the same type, like all 'X' decorators i.e. [(#6, [X4, +0]), (#7, [X3])] –> [(#6, [+0]), (#7, [])]

Parameters **input_ors** (*list of two tuples*) – OR decorators in the form from ChemicalEnvironments that is [(base, [decorators,]), ...]

Returns **new_ors** – New OR decorators

Return type list of two tuples

remove_and(*input_all_ands*)
removes a decorator that is AND'd in the original SMIRKS

Parameters **input_all_ands** (*list*) – List of AND decorators

Returns **new_ands** – List of new AND decorators

Return type list

remove_decorator(*smirks*)
Chose an atom or bond in the input smirks pattern and then remove one decorator from it.

Parameters **smirks** (*str*) – A SMIRKS string which should be reduced

Returns

- **new_smirks** (*str*) – A new SMIRKS pattern
- **is_changed** (*bool*) – True if some of the decorators were successfully removed

remove_one_sub_dec(*input_ors*, *ref_idx*)
Remove one OR decorator from the specified index # i.e. [(#6, [X4, +0]), (#7, [X3])] –> [(#6, [+0]), (#7, [X3])]

Parameters

- **input_ors** (*list of two tuples*) – OR decorators in the form from ChemicalEnvironments that is [(base, [decorators,]), ...]
- **ref_idx** (*int*) – The index from this list to use when removing one sub-decorator

Returns **new_ors** – New OR decorators

Return type list of two tuples

remove_or(*input_all_ors*, *is_bond=False*)
Changes the OR decorators by removing some of them

Parameters

- **input_all_ors** (*list of tuples*) – these are the OR decorators for an atom or bond from a ChemicalEnvironment
- **is_bond** (*boolean*) – are these decorators from from a bond (False for atom)

Returns **new_ors** – new OR decorators

Return type list of two tuples

remove_or_atom(*input_all_ors*, *or_idx*)
makes specific types of changes based on atom OR decorators

Parameters

- **input_all_ors** (*list of OR decorators*) – [(base, [decs]), . . .]
- **or_idx** (*index that should be used to guide changes*) –

Returns **new_ors** – new or decorators

Return type list of two tuples

remove_ref(*input_ors*, *ref_idx*)
Remove the decorator at the referenced index i.e. [(#6, [X4, +0]), (#7, [X3])] –> [(#7, [X3])]

Parameters

- **input_ors** (*list of two tuples*) – OR decorators in the form from ChemicalEnvironments that is [(base, [decorators,]), . . .]
- **ref_idx** (*int*) – The OR decorators at ref_idx will be removed entirely

Returns **new_ors** – New OR decorators

Return type list of two tuples

remove_ref_sub_decs(*input_ors*, *ref_idx*)
Remove all of the ORdecorators at the specified index i.e. [(#6, [X4, +0]), (#7, [X3])] –> [(#6, []), (#7, [X3])]

Parameters

- **input_ors** (*list of two tuples*) – OR decorators in the form from ChemicalEnvironments that is [(base, [decorators,]), . . .]
- **ref_idx** (*int*) – The index from this list to use when removing one set of sub-decorators

Returns **new_ors** – New OR decorators

Return type list of two tuples

run(*max_its=1000*, *verbose=None*)
Reduce the SMIRKS decorators for a number of iterations

Parameters

- **max_its** (*int*) – The specified number of iterations
- **verbose** (*boolean*) – will set the verboseness while running (if None, the current verbose variable will be used)

Returns **final_smirks** – list of final smirks patterns after reducing in the form [(label, smirks)]

Return type list of tuples

class chemper.smirksify.**SMIRKSifier**(*molecules*, *cluster_list*, *max_layers=5*, *verbose=True*, *strict_smirks=True*)
Generates complex SMIRKS for a given cluster of substructures and then reduces the decorators in those smirks

make_smirks()
Create a list of SMIRKS patterns for the input clusters. This includes a determining how far away from the indexed atom should be included in the SMIRKS (or the number of max_layers is reached)

Returns

- **smirks_list** (*list of tuples*) – list of tuples in the form (label, smirks)
- **layers** (*int*) – number of layers actually used to specify the set clusters

reduce(*max_its=1000*, *verbose=None*)
Reduce the SMIRKS decorators for a number of iterations

Parameters

- **max_its** (*int*) – default = 1000 The specified number of iterations
- **verbose** (*boolean*) – default = None will set the verboseness while running (if None, the current verbose variable will be used)

Returns **final_smirks** – list of final smirks patterns after reducing in the form [(label, smirks)]

Return type list of tuples

types_match_reference(*current_types=None*)

Determine best match for each parameter with reference types

Parameters **current_types** (*list of tuples with form [(label, smirks),]*) –

Returns **type_matches** – pair of current and reference labels with the number of fragments that match it

Return type list of tuples (current_label, reference_label, counts)

chemper.smirksify.**print_smirks**(*smirks_list*)

Prints out the provided smirks list in a table like format with label | SMIRKS

Parameters **smirks_list** (*list of tuples*) – list in the form [(label, SMIRKS), . . .]

CHAPTER

FOUR

EXAMPLES

This page provides examples on how to use `chemper`. Each subsection is a Jupyter notebook, all are available in the examples folder on github so you can try them out yourself.

4.1 Single molecule SMIRKS

The `ChemPerGraph` objects are intended to create SMIRKS from molecules based on specified atoms. The easiest way to do this is to provide `ChemPerGraphFromMol` a molecule and dictionary of key atoms.

These objects were created largely as a precursor to `ClusterGraph`. However, it is possible some people will find them useful as a standalone object. To the best of my knowledge, there was not previously a tool to create a SMIRKS pattern for a whole molecule. `RDKit` has a way to write molecules as "SMARTS" but as far as I can tell it just writes a SMILES string with square brackets around the atoms.

Below are a variety of examples for `ChemPerGraphs` for a variety of molecules.

[ ]:
```
from chemper.mol_toolkits import mol_toolkit
from chemper.graphs.fragment_graph import ChemPerGraphFromMol
```

4.1.1 Simple SMIRKS

The most simple SMIRKS patterns have only the indexed atoms.

`ChemPerGraphFromMol` objects are initiated with a chemper molecule and a dictioanry storing atom indices by desired SMIRKS index:

[2]:
```
mol = mol_toolkit.MolFromSmiles('C1CCC1')
# store atom 1 in smirks index 1 and atom 4 in smirks index 2
smirks_atoms = (0, 4)
graph = ChemPerGraphFromMol(mol, smirks_atoms)
graph.as_smirks()
```

[2]:
```
'[#6AH2X4x2r4+0:1]-!@[#1AH0X1x0r0+0:2]'
```

Extend away from the indexed atoms

You can also extend away from the indexed atoms using the optional layers argument. If layers is greater than 0 then atoms up to that many bonds away from the indexed atoms are added to the graph.

Lets start with just 1 layer away

[3]:
```
graph = ChemPerGraphFromMol(mol, smirks_atoms, layers=1)
graph.as_smirks()
```

[3]:
```
'[#6AH2X4x2r4+0:1](-!@[#1AH0X1x0r0+0:2])(-!@[#1AH0X1x0r0+0])(-@[#6AH2X4x2r4+0])-@[
↪#6AH2X4x2r4+0]'
```

Encode the whole molecule

The other option for layers is 'all' which will continue adding atoms until there are no more atoms in the molecule. These SMIRKS become really unreadable for humans, but do encode all information about the molecule.

[4]:
```
graph = ChemPerGraphFromMol(mol, smirks_atoms, layers='all')
graph.as_smirks()
```

[4]:
```
'[#6AH2X4x2r4+0:1](-!@[#1AH0X1x0r0+0:2])(-!@[#1AH0X1x0r0+0])-@[#6AH2X4x2r4+0](-!@[
↪#1AH0X1x0r0+0])(-!@[#1AH0X1x0r0+0])-@[#6AH2X4x2r4+0](-!@[#1AH0X1x0r0+0])(-@[
↪#6AH2X4x2r4+0](-!@[#1AH0X1x0r0+0])-!@[#1AH0X1x0r0+0])-!@[#1AH0X1x0r0+0]'
```

[ ]:
```

```

4.2 Make SMIRKS from clustered fragments

This notebook will showcase how `ChemPer`'s `ClusterGraph` creates SMIRKS patterns from a group of user specified molecular fragments.
For example, imagine we wanted to create a SMIRKS pattern for an angle type that appears in many molecules. `ClusterGraph` collects the SMIRKS decorators from every molecule and stores them in a highlyspecific SMIRKS pattern.

The ultimate goal for chemper is to create a hierarchical list of SMIRKS patterns that retains fragment clustering. We could use this tool to generate SMIRKS patterns for the SMIRNOFF force field format allowing use to create data driven, direct chemical percpeption.

For example, if your initial clusters had 4 types of carbon-carbon bonds (single, aromatic, double, and triple), you would expect the final SMIRKS patterns to reflect those four categories.

The first step here is to store possible decorators for atoms and bonds in a given cluster. In this notebook we will use example SMIRKS patterns as a way of identifying groups of molecular fragments. Then we will use `ClusterGraph` to create highly specific SMIRKS for these same fragments.

[1]:
```
# import statements
from chemper.mol_toolkits import mol_toolkit
from chemper.graphs.cluster_graph import ClusterGraph
from chemper.chemper_utils import create_tuples_for_clusters
```

4.2.1 create_tuples_for_clusters

This is a utility function inside ChemPer which extracts atom indices which match a specific SMIRKS pattern.

Help on function create_tuples_for_clusters in module chemper.chemper_utils: For example, lets assume you wanted to find all of the atoms that match this SMIRKS list * "any", `'[*:1]~[*:2]'` * "single", `'[*:1]-[*:2]'`

In this case, the "any" bond would match all bonds, but then the "single" would match all single bonds. If you were typing Ethene (C=C) then you expect the double bond between carbon atoms 0 and 1 to match "any" bond and all C-H bonds to match "single".

The output in this case would be:

```
[ ('any', [[ (0, 1) ]] ),
  ('single', [[ (0, 2), (0, 3), (1,4), (1,5) ]] )
]
```

Clustering from other SMIRKS

This example attempts to show how `ClusterGraph` creates a SMIRKS for already clustered sub-graphs.

Here, we will consider two types of angles around tetrahedral carbon atoms. In this hierarchical list `c1` would match ANY angle around a tetrahedral carbon (indicated with the connectivity `X4` on atom `:2`). Then `c2` would match angles where both outer atoms are hydrogens, just H-C-H angles, meaning those angles would be assigned `c2` and NOT `c1`.

We will use the utility function `create_tuples_for_clusters` (described above) to identify atoms in each example moleucle that match each of these angle types.

[2]:
```
smirks_list = [
    ("c1", "[*:1]~[#6X4:2]-[*:3]"),
    ("c2", "[#1:1]-[#6X4:2]-[#1:3]"),
]
for label, smirks in smirks_list:
    print(label,'\t',smirks)
```

```
c1       [*:1]~[#6X4:2]-[*:3]
c2       [#1:1]-[#6X4:2]-[#1:3]
```

Start with a single molecule

For the first example, we will start with just one molecule (ethane) and extract the clusters of atoms matching each angle type.

Ethane has a total of 12 sets of angles, all of which can be categorized by the two SMIRKS patterns `c1` or `c2` * 6 with the form H-C-C - type c1 * 6 with the form H-C-H - type c2

First we need to extract the atoms for each of these categories. We use tuples of atom indices to represent these two clusters which are identified using the `create_tuple_for_cluster` utilities function.

[3]:
```
mol = mol_toolkit.MolFromSmiles('CC')
atom_index_list = create_tuples_for_clusters(smirks_list, [mol])
for label, mol_list in atom_index_list:
    print(label)
    for mol_idx, atom_list in enumerate(mol_list):
        print('\tmolecule ', mol_idx)
```

(continues on next page)

(continued from previous page)

```
        for atoms in atom_list:
            print('\t\t', atoms)
```

```
c1
        molecule  0
                  (1, 0, 3)
                  (0, 1, 7)
                  (0, 1, 6)
                  (1, 0, 4)
                  (1, 0, 2)
                  (0, 1, 5)
c2
        molecule  0
                  (5, 1, 7)
                  (5, 1, 6)
                  (6, 1, 7)
                  (3, 0, 4)
                  (2, 0, 4)
                  (2, 0, 3)
```

Next, we will look at the `ClusterGraph` for the set of atoms matching the angle type `c1` (`[*:1]~[#6X4:2]-[*:3]`). `ClusterGraph` works by only storing the unique combination of atom decorators. That means that even though we are using six sets of atoms there is only one set of decorators for each atom in the SMIRKS patterns

[6]:
```
c1_atoms = atom_index_list[0][1]
graph = ClusterGraph([mol], c1_atoms)
print(graph.as_smirks())
```

```
[#6AH3X4x0!r+0:1]-;!@[#6AH3X4x0!r+0:2]-;!@[#1AH0X1x0!r+0:3]
```

4.2.2 Adding Layers

Similar to the `ChemPerGraph`s described in the `single_mol_smirks` example. We can add atoms outside those indexed in `ClusterGraph`. This is done with the key word `layers`. The specified number of layers corresponds to the number of bonds away from an indexed atom should be included in the SMIRKS. As with `ChemPerGraph`s, you can also use the keyword `"all"` to include all atoms in a molecule in the SMIRKS pattern. For ethane, this would result in the same SMIRKS as specifying 1 layer:

[7]:
```
print("layers = 0")
graph = ClusterGraph([mol], c1_atoms, layers=1)
print(graph.as_smirks())
print('-'*80)
print("layers='all'")
graph = ClusterGraph([mol], c1_atoms, layers='all')
print(graph.as_smirks())
```

```
layers = 0
[#6AH3X4x0!r+0:1](-;!@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!@[
↪#6AH3X4x0!r+0:2](-;!@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!@[#1AH0X1x0!r+0:3]
--------------------------------------------------------------------------------
layers='all'
[#6AH3X4x0!r+0:1](-;!@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!@[
↪#6AH3X4x0!r+0:2](-;!@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!@[#1AH0X1x0!r+0:3]
```

Multiple molecules

Now that you have the general idea, lets consider a more complex case, Lets create a `ClusterGraph` for both labels in the `smirks_list` from above for the hydrocarbons listed below.

First we need to create the molecules and use `create_tuple_for_cluster` to find group the angles by category.

[8]:
```
smiles = ['CC', 'CCC', 'C1CC1', 'CCCC', 'CC(C)C', 'C1CCC1', 'CCCCC']
mols = [mol_toolkit.MolFromSmiles(s) for s in smiles]
atom_index_list = create_tuples_for_clusters(smirks_list, mols)
for label, mol_list in atom_index_list:
    print(label)
    for mol_idx, atom_list in enumerate(mol_list):
        print('\tmolecule ', mol_idx)
        for atoms in atom_list:
            print('\t\t', atoms)
```

```
c1
        molecule  0
                  (1, 0, 3)
                  (0, 1, 7)
                  (0, 1, 6)
                  (1, 0, 4)
                  (1, 0, 2)
                  (0, 1, 5)
        molecule  1
                  (1, 0, 3)
                  (1, 2, 8)
                  (1, 0, 5)
                  (1, 0, 4)
                  (2, 1, 7)
                  (0, 1, 2)
                  (1, 2, 9)
                  (0, 1, 7)
                  (1, 2, 10)
                  (0, 1, 6)
                  (2, 1, 6)
        molecule  2
                  (2, 0, 4)
                  (1, 2, 8)
                  (2, 0, 3)
                  (1, 0, 3)
                  (0, 2, 8)
                  (1, 2, 7)
                  (1, 0, 2)
                  (2, 1, 5)
                  (0, 2, 7)
                  (0, 1, 2)
                  (0, 1, 6)
                  (0, 2, 1)
                  (2, 1, 6)
                  (0, 1, 5)
                  (1, 0, 4)
        molecule  3
                  (2, 1, 7)
                  (0, 1, 8)
                  (0, 1, 7)
                  (0, 1, 2)
                  (1, 2, 9)
```

(continues on next page)

(continued from previous page)

```
                (2, 3, 12)
                (1, 2, 3)
                (1, 2, 10)
                (1, 0, 6)
                (1, 0, 4)
                (3, 2, 10)
                (1, 0, 5)
                (2, 1, 8)
                (2, 3, 11)
                (2, 3, 13)
                (3, 2, 9)
        molecule  4
                (2, 1, 7)
                (1, 2, 8)
                (0, 1, 7)
                (0, 1, 2)
                (1, 2, 9)
                (0, 1, 3)
                (1, 2, 10)
                (1, 0, 6)
                (3, 1, 7)
                (2, 1, 3)
                (1, 0, 4)
                (1, 3, 13)
                (1, 0, 5)
                (1, 3, 12)
                (1, 3, 11)
        molecule  5
                (1, 0, 3)
                (1, 2, 8)
                (0, 1, 7)
                (2, 1, 7)
                (0, 1, 2)
                (1, 2, 9)
                (3, 0, 4)
                (1, 2, 3)
                (3, 0, 5)
                (1, 0, 4)
                (2, 1, 6)
                (0, 1, 6)
                (1, 0, 5)
                (2, 3, 10)
                (2, 3, 11)
                (0, 3, 11)
                (3, 2, 8)
                (0, 3, 2)
                (0, 3, 10)
                (3, 2, 9)
        molecule  6
                (0, 1, 8)
                (0, 1, 2)
                (0, 1, 9)
                (2, 3, 12)
                (1, 2, 3)
                (1, 2, 10)
                (4, 3, 13)
                (1, 0, 6)
```

(continues on next page)

(continued from previous page)

```
                (1, 2, 11)
                (1, 0, 7)
                (3, 4, 16)
                (3, 2, 10)
                (1, 0, 5)
                (2, 1, 8)
                (3, 2, 11)
                (2, 1, 9)
                (2, 3, 13)
                (3, 4, 14)
                (2, 3, 4)
                (4, 3, 12)
                (3, 4, 15)
c2
        molecule  0
                (5, 1, 7)
                (5, 1, 6)
                (6, 1, 7)
                (3, 0, 4)
                (2, 0, 4)
                (2, 0, 3)
        molecule  1
                (8, 2, 9)
                (6, 1, 7)
                (3, 0, 4)
                (3, 0, 5)
                (9, 2, 10)
                (4, 0, 5)
                (8, 2, 10)
        molecule  2
                (5, 1, 6)
                (3, 0, 4)
                (7, 2, 8)
        molecule  3
                (11, 3, 13)
                (11, 3, 12)
                (9, 2, 10)
                (7, 1, 8)
                (5, 0, 6)
                (4, 0, 6)
                (12, 3, 13)
                (4, 0, 5)
        molecule  4
                (11, 3, 13)
                (11, 3, 12)
                (9, 2, 10)
                (12, 3, 13)
                (8, 2, 9)
                (5, 0, 6)
                (4, 0, 6)
                (4, 0, 5)
                (8, 2, 10)
        molecule  5
                (6, 1, 7)
                (8, 2, 9)
                (10, 3, 11)
                (4, 0, 5)
```

(continues on next page)

(continued from previous page)

```
        molecule  6
                  (8, 1, 9)
                  (12, 3, 13)
                  (14, 4, 15)
                  (5, 0, 6)
                  (15, 4, 16)
                  (6, 0, 7)
                  (14, 4, 16)
                  (5, 0, 7)
                  (10, 2, 11)
```

Now lets make a `ClusterGraph` object for both `c1` and `c2`. In these patterns you will see lists of decorators on each atom. In the SMIRKS lanage `','` stands for 'OR'. So in the case of `"[#6AH1X4x0!r+0,#6AH2X4x0!r+0:1]"` both decorator sets (`"#6AH1X4x0!r+0"` or `"#6AH2X4x0!r+0"`) could match up with atom `:1`

[9]:
```
c1_graph = ClusterGraph(mols, atom_index_list[0][1])
print('c1\n'+'-'*50)
print(c1_graph.as_smirks())
c2_graph = ClusterGraph(mols, atom_index_list[1][1])
print()
print('c2\n'+'-'*50)
print(c2_graph.as_smirks())
```

```
c1
--------------------------------------------------
[#6AH1X4x0!r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0:1]-[#6AH1X4x0!
↪r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0:2]-[#1AH0X1x0!r+0,
↪#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0:3]

c2
--------------------------------------------------
[#1AH0X1x0!r+0:1]-;!@[#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0:2]-;!@[
↪#1AH0X1x0!r+0:3]
```

4.2.3 Identifying common decorators

You might notice that some SMIRKS decorators in each atom list are very similar. For example, all of our atoms are neutral so they all have the decorator `"+0"` to indicate a formal charge of zero.

We can take advantage of these commonalities and group decorators together using the SMIRKS `";"` symbol for ANDing decorators. For example, in `"[#6,#7;+0:1]"` the atom is either carbon (`#6`) or (`,`) nitrogen (`#7`) and (`;`) it has a zero formal charge (`+0`).

In the `ChemPer` graph language you can group like decorators using the keyword `compress`. In that case we get these SMIRKS patterns for `c1` and `c2` instead:

[10]:
```
print('c1\n'+'-'*50)
print(c1_graph.as_smirks(compress=True))
print()
print('c2\n'+'-'*50)
print(c2_graph.as_smirks(compress=True))
```

```
c1
--------------------------------------------------
[*!rH1x0,*!rH2x0,*!rH3x0,*H2r3x2,*H2r4x2;#6;+0;A;X4:1]-[*!rH1x0,*!rH2x0,*!rH3x0,
↪*H2r3x2,*H2r4x2;#6;+0;A;X4:2]-[#1!rH0X1x0,#6!rH2X4x0,#6!rH3X4x0,#6H2X4r3x2,
↪#6H2X4r4x2;+0;A:3]
```

(continues on next page)

(continued from previous page)

```
c2
--------------------------------------------------
[#1AH0X1x0!r+0:1]-;!@[*!rH2x0,*!rH3x0,*H2r3x2,*H2r4x2;#6;+0;A;X4:2]-;!@[#1AH0X1x0!r+0:
↪3]
```

4.2.4 Adding layers

As shown above we could also add `layers` to the `ClusterGraph`s with multiple molecules.

[11]:
```
for l in [1,2,3]:
    print('layers = ', l)
    c1_graph = ClusterGraph(mols, atom_index_list[0][1], layers=l)
    print('c1\n'+'-'*50)
    print(c1_graph.as_smirks())
    c2_graph = ClusterGraph(mols, atom_index_list[1][1], layers=l)
    print()
    print('c2\n'+'-'*50)
    print(c2_graph.as_smirks())
    print('\n', '='*80, '\n')
```

```
layers =  1
c1
--------------------------------------------------
[#6AH1X4x0!r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0:1](-[#1AH0X1x0!
↪r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0])(-;!@[#1AH0X1x0!r+0,
↪#6AH3X4x0!r+0])(-;!@[#1AH0X1x0!r+0])-[#6AH1X4x0!r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,
↪#6AH2X4x2r4+0,#6AH3X4x0!r+0:2](-[#1AH0X1x0!r+0,#6AH2X4x0!r+0,#6AH2X4x2r4+0,
↪#6AH3X4x0!r+0])(-;!@[#1AH0X1x0!r+0,#6AH3X4x0!r+0])-[#1AH0X1x0!r+0,#6AH2X4x0!r+0,
↪#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0:3](-;!@[#1AH0X1x0!r+0,#6AH2X4x0!r+0,
↪#6AH3X4x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!@[#1AH0X1x0!r+0]

c2
--------------------------------------------------
[#1AH0X1x0!r+0:1]-;!@[#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0:2](-[
↪#1AH0X1x0!r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0])(-[#6AH1X4x0!
↪r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0])-;!@[#1AH0X1x0!r+0:3]

 ================================================================================

layers =  2
c1
--------------------------------------------------
[#6AH1X4x0!r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0:1](-[#1AH0X1x0!
↪r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0](-[#1AH0X1x0!r+0,
↪#6AH2X4x0!r+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!@[#1AH0X1x0!
↪r+0])(-;!@[#1AH0X1x0!r+0,#6AH3X4x0!r+0](-;!@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!
↪@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])-[#6AH1X4x0!r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,
↪#6AH2X4x2r4+0,#6AH3X4x0!r+0:2](-;!@[#1AH0X1x0!r+0,#6AH2X4x0!r+0,#6AH3X4x0!r+0](-;!@[
↪#1AH0X1x0!r+0,#6AH2X4x0!r+0,#6AH3X4x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!@[#1AH0X1x0!
↪r+0])(-;!@[#1AH0X1x0!r+0,#6AH3X4x0!r+0](-;!@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!
↪@[#1AH0X1x0!r+0])-[#1AH0X1x0!r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,
↪#6AH3X4x0!r+0:3](-;!@[#1AH0X1x0!r+0,#6AH2X4x0!r+0,#6AH3X4x0!r+0](-;!@[#1AH0X1x0!r+0,
↪#6AH3X4x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!@[
↪#1AH0X1x0!r+0]
```

(continues on next page)

(continued from previous page)

```
c2
--------------------------------------------------
[#1AH0X1x0!r+0:1]-;!@[#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0:2](-[
↪#1AH0X1x0!r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0](-[#1AH0X1x0!
↪r+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!@[#1AH0X1x0!r+0])(-[
↪#6AH1X4x0!r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0](-;!@[
↪#1AH0X1x0!r+0,#6AH2X4x0!r+0,#6AH3X4x0!r+0])(-;!@[#1AH0X1x0!r+0,#6AH3X4x0!r+0])-;!@[
↪#1AH0X1x0!r+0])-;!@[#1AH0X1x0!r+0:3]

 ================================================================================

layers =  3
c1
--------------------------------------------------
[#6AH1X4x0!r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0:1](-[#1AH0X1x0!
↪r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0](-[#1AH0X1x0!r+0,
↪#6AH2X4x0!r+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0](-;!@[#1AH0X1x0!r+0,#6AH3X4x0!r+0])(-;!@[
↪#1AH0X1x0!r+0])-;!@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!@[#1AH0X1x0!r+0])(-;!@[
↪#1AH0X1x0!r+0,#6AH3X4x0!r+0](-;!@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!@[
↪#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])-[#6AH1X4x0!r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,
↪#6AH2X4x2r4+0,#6AH3X4x0!r+0:2](-;!@[#1AH0X1x0!r+0,#6AH2X4x0!r+0,#6AH3X4x0!r+0](-;!@[
↪#1AH0X1x0!r+0,#6AH2X4x0!r+0,#6AH3X4x0!r+0](-;!@[#1AH0X1x0!r+0,#6AH3X4x0!r+0])(-;!@[
↪#1AH0X1x0!r+0])-;!@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!@[#1AH0X1x0!r+0])(-;!@[
↪#1AH0X1x0!r+0,#6AH3X4x0!r+0](-;!@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!@[
↪#1AH0X1x0!r+0])-[#1AH0X1x0!r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!
↪r+0:3](-;!@[#1AH0X1x0!r+0,#6AH2X4x0!r+0,#6AH3X4x0!r+0](-;!@[#1AH0X1x0!r+0,#6AH3X4x0!
↪r+0](-;!@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!
↪r+0])-;!@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!@[#1AH0X1x0!r+0]

c2
--------------------------------------------------
[#1AH0X1x0!r+0:1]-;!@[#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0:2](-[
↪#1AH0X1x0!r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0](-[#1AH0X1x0!
↪r+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0](-;!@[#1AH0X1x0!r+0])-;!@[#1AH0X1x0!r+0])(-;!@[
↪#1AH0X1x0!r+0])-;!@[#1AH0X1x0!r+0])(-[#6AH1X4x0!r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,
↪#6AH2X4x2r4+0,#6AH3X4x0!r+0](-;!@[#1AH0X1x0!r+0,#6AH2X4x0!r+0,#6AH3X4x0!r+0](-;!@[
↪#1AH0X1x0!r+0,#6AH2X4x0!r+0,#6AH3X4x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!@[#1AH0X1x0!
↪r+0])(-;!@[#1AH0X1x0!r+0,#6AH3X4x0!r+0](-;!@[#1AH0X1x0!r+0])(-;!@[#1AH0X1x0!r+0])-;!
↪@[#1AH0X1x0!r+0])-;!@[#1AH0X1x0!r+0])-;!@[#1AH0X1x0!r+0:3]

 ================================================================================
```

Where do you go from here

As you see above, the `ClusterGraph` SMIRKS are significantly more complicated and specific than the input SMIRKS. For example, the input SMIRKS for `c1` is `[*:1]~[#6X4:2]-[*:3]`, however `ClusterGraph` creates this monstrosity:

```
[#6AH1X4x0!r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0:1]-[#6AH1X4x0!
↪r+0,#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0:2]-[#1AH0X1x0!r+0,
↪#6AH2X4x0!r+0,#6AH2X4x2r3+0,#6AH2X4x2r4+0,#6AH3X4x0!r+0:3]
```

Although this pattern becomes a bit less complex with the compression:

```
[*!rH1x0,*!rH2x0,*!rH3x0,*H2r3x2,*H2r4x2;#6;+0;A;X4:1]-[*!rH1x0,*!rH2x0,*!rH3x0,
↪*H2r3x2,*H2r4x2;#6;+0;A;X4:2]-[#1!rH0X1x0,#6!rH2X4x0,#6!rH3X4x0,#6H2X4r3x2,
↪#6H2X4r4x2;+0;A:3]
```

Our goal is to generate a hierarchical list of SMIRKS would could recover the same chemistry in a different list of molecules. In order to do this we would want to generate the SMIRKS patterns for different clusters and then remove unnecessary decorators.

To meet this purpose we created the `SMIRKSifier`. For details on this topic see the notebook `smirksifying_clusters` in this example folder.

[ ]:

4.3 Generating reasonable SMIRKS patterns

In this notebook we will demonstrate how `chemper`'s `SMIRKSifier` works to generate SMIRKS patterns for a list of molecules and assigned clustering. In other words, the goal is to generate a list of SMIRKS patterns which will maintain the clustering of molecular fragments the user specifies. For example, imagine you have determined the force field parameters for all bonds in your molecules set. You could group the bonds based on those which should be assigned the same force constant and equilibrium bond length. The goal of `chemper`'s tools is to generate a hierarchical list of SMIRKS that will maintain your clustering.

`chemper`'s `ClusterGraph` can create a single SMIRKS pattern for a group of molecular fragments. These SMIRKS patterns are fully specified using all possible SMIRKS decorators for each atom. The `SMIRKSifier` takes advantage of the `ClusterGraph` functionality and then removes unnecessary SMIRKS decorators so the final list of patterns is as generic as possible.

In this example, we assume that we want to group bonds based on their bond order so all single bonds should be in one group, all double in another group, and so on. The steps shown below are as follows:

1. Create `chemper` molecules for a list of SMILES
2. Classify the bonds in each molecule as single, double, aromatic, triple. Then group those bonds (based on atom indices) into each of those categories.
3. Use `chemper.optimize_smirks.smirksify` to automatically create a hierarchical SMIRKS pattern list, then run it to remove unecessary decorators.

[1]:
```
from chemper.mol_toolkits import mol_toolkit
from chemper.chemper_utils import create_tuples_for_clusters
from chemper.smirksify import SMIRKSifier, print_smirks
```

4.3.1 1. Create a list of Molecules

Here we chose a list of SMILES strings and then use `chemper.mol_toolkits` to create a list of molecule objects.

[2]:
```
smiles_list = ["CCCCC", "c1ccccc1", "C1=CNC=C1", "CC=CC", "C(=O)OC", "C1C=CCCC1"]
molecules = [mol_toolkit.MolFromSmiles(s) for s in smiles_list]
```

4.3.2 2. Classify bonds

In this section we classify bonds based on the categories

- single
- aromatic
- double
- triple

This is done with the utility function `create_tuples_for_clusters` which creates a list atom indices (as tuples), for each molecule `[ ('label', [ [(tuple of atoms for each molecule), ...] ...]) ]`.

In a bond we have two indexed atoms in a SMIRKS (1 and 2) because you need two atoms in order to identify a bond. For example, in the first molecule above, there is a single bond between atoms 0 and 1. The cluster_list would specify that bond with a tuple `(0,1)`. There is a list of tuples for each molecule associated with each label.

In this example, there are six molecules. As an illustration of how the cluster_list is stuctured, consider the aromatic bonds, at `cluster_list[1]`. Only molecules 1 and 2 have aromatic bonds. The bonds in these molecules are specified by tuples showing the atom indices for each of the aromatic bonds below. The other four molecules have zero aromatic bonds so the associated lists are empty as no bonds need to be specified.

Moving forward

Obviously in the long run we don't want to start with SMIRKS pattern, however, you could imagine identifying the equilibrium bond length and force constant for a variety of bonds. You could then cluster those bonds based on which parameters they should be assigned. You could give `chemper` these clusters of bonds as well.

[3]:
```
smirks_labels = [('sing', '[*:1]-[*:2]'),
                 ('aromatic', '[*:1]:[*:2]'),
                 ('double', '[*:1]=[*:2]'),
                ('triple', '[*:1]#[*:2]'),
                ]
cluster_list = create_tuples_for_clusters(smirks_labels, molecules)
```

[4]:
```
cluster_list[1]
```

[4]:
```
('aromatic',
 [[],
  [(0, 1), (1, 2), (4, 5), (0, 5), (2, 3), (3, 4)],
  [(1, 2), (0, 1), (0, 4), (2, 3), (3, 4)],
  [],
  [],
  []])
```

4.3.3 3. Generate SMIRKS and remove unnecessary SMIRKS decorators

The goal in this step is to create a generic, hierarchical list of SMIRKS patterns which will maintain the clustering of bonds we specified above.

First we will create a `SMIRKSifier` object. This takes your molecules and the list of classified bonds and automatically creates SMIRKS patterns using ALL possible decorators. As you can see this process leads to highly specific patterns which would not be practical assuming you want your clustering to be applied to molecules outside your training set.

There area also two optional arguments for the `SMIRKSifier`:

- `max_layers`: this is the maximum number of atoms away from the indexed atoms you want included in your automatically generated SMIRKS patterns.
  - Internally, `SMIRKSifier` starts with 0 layers and only adds atoms if necessary
  - the default is `max_layers = 5`
- `verbose`: if `True` (the default) the `SMIRKSifier` prints out information while matching the automatically generated SMIRKS with the initially assigned clusters.

3a. Create the `SMIRKSifier` printing out initial SMIRKS patterns

[5]:
```
bond_smirksifier = SMIRKSifier(molecules, cluster_list)
```

```
 Label                | SMIRKS
==========================================================================
 zz_sing              | [#6!rAH1X3x0,#6!rAH2X4x0,#6!rAH3X4x0,#6AH1X3r6x2,#6AH2X4r6x2,
↪#6H1X3ar5x2,#6H1X3ar6x2,#7H1X3ar5x2,#8!rAH0X2x0;+0:1]-[#1!rH0X1x0,#6!rH1X3x0,#6!
↪rH2X4x0,#6!rH3X4x0,#6H1X3r6x2,#6H2X4r6x2,#8!rH0X2x0;+0;A:2]
--------------------------------------------------------------------------
 zz_aromatic          | [#6r5,#6r6,#7r5;+0;H1;X3;a;x2:1]:;@[#6r5,#6r6,#7r5;+0;H1;X3;a;
↪x2:2]
--------------------------------------------------------------------------
 zz_double            | [#6!rx0,#6r6x2;+0;A;H1;X3:1]=[#6!rH1X3x0,#6H1X3r6x2,#8!
↪rH0X1x0;+0;A:2]
--------------------------------------------------------------------------
```

3b. Start removing decorators

The `SMIRKSifier.reduce` function attempts to remove a single decorator from a randomly chosen SMIRKS pattern during each iteration, it has two argument:

- `max_its`(optional, default=1000): Number of iterations to remove
  - currently it does run for this many we are working on determining if there is a way to determine if we are done before the number of iterations is reached
- `verbose` (optional, default=do not change the setting): This will temporarily change the `SMIRKSifier`'s verboseness, so you could make a long run more quiet.

This run returns the current set of SMIRKS patterns at the end of the simulation. You can use the internal `SMIRKSifier.print_smirks` function to print these in a semi-nicely formatted way.

[6]:
```
smirks10 = bond_smirksifier.reduce(max_its=10)
```

```
Iteration:  0
Attempting to change SMIRKS #2
[#6!rx0,#6r6x2;+0;A;H1;X3:1]=[#6!rH1X3x0,#6H1X3r6x2,#8!rH0X1x0;+0;A:2]  -->  [#6x0,
↪#6x2r6;+0;A;H1;X3:1]=[#6,#6x2r6H1X3,#8H0x0X1;+0;A:2]
Accepted!
--------------------------------------------------------------------------------
↪----
Iteration:  1
Attempting to change SMIRKS #1
[#6r5,#6r6,#7r5;+0;H1;X3;a;x2:1]:;@[#6r5,#6r6,#7r5;+0;H1;X3;a;x2:2]  -->  [#6r5,#6r6,
↪#7r5;a;X3;+0;H1:1]:;@[#6r5,#6r6,#7r5;+0;H1;X3;a;x2:2]
Accepted!
--------------------------------------------------------------------------------
↪----
Iteration:  2
Attempting to change SMIRKS #1
[#6r5,#6r6,#7r5;a;X3;+0;H1:1]:;@[#6r5,#6r6,#7r5;+0;H1;X3;a;x2:2]  -->  [*;a;X3;+0;H1:
↪1]:;@[#6r5,#6r6,#7r5;+0;H1;X3;a;x2:2]
Accepted!
--------------------------------------------------------------------------------
↪----
Iteration:  3
Attempting to change SMIRKS #1
[*;a;X3;+0;H1:1]:;@[#6r5,#6r6,#7r5;+0;H1;X3;a;x2:2]  -->  [*;a;+0;H1:1]:;@[#6r5,#6r6,
↪#7r5;+0;H1;X3;a;x2:2]
Accepted!
--------------------------------------------------------------------------------
↪----
Iteration:  4
Attempting to change SMIRKS #0
[#6!rAH1X3x0,#6!rAH2X4x0,#6!rAH3X4x0,#6AH1X3r6x2,#6AH2X4r6x2,#6H1X3ar5x2,#6H1X3ar6x2,
↪#7H1X3ar5x2,#8!rAH0X2x0;+0:1]-[#1!rH0X1x0,#6!rH1X3x0,#6!rH2X4x0,#6!rH3X4x0,
↪#6H1X3r6x2,#6H2X4r6x2,#8!rH0X2x0;+0;A:2]  -->  [#6x0X3AH1,#6x0X4AH2,#6x0AX4H3,
↪#6x2X3AH1r6,#6x2X4AH2r6,#6aX3x2r5H1,#6aX3r6x2H1,#7aX3x2r5H1,#8H0x0AX2;+0:1]-[
↪#1H0x0X1,#6x0X3H1,#6x0X4H2,#6x0X4H3,#6x2X3r6H1,#6x2X4H2r6,#8H0x0X2;A:2]
Accepted!
--------------------------------------------------------------------------------
↪----
Iteration:  5
Attempting to change SMIRKS #1
[*;a;+0;H1:1]:;@[#6r5,#6r6,#7r5;+0;H1;X3;a;x2:2]  -->  [*;a;+0;H1:1]:;@[#6r5,#6r6,
↪#7r5:2]
Accepted!
--------------------------------------------------------------------------------
↪----
Iteration:  6
Attempting to change SMIRKS #1
[*;a;+0;H1:1]:;@[#6r5,#6r6,#7r5:2]  -->  [*;a;+0;H1:1]:;@[#6r5,#6,#7r5:2]
Accepted!
--------------------------------------------------------------------------------
↪----
Iteration:  7
Attempting to change SMIRKS #0
[#6x0X3AH1,#6x0X4AH2,#6x0AX4H3,#6x2X3AH1r6,#6x2X4AH2r6,#6aX3x2r5H1,#6aX3r6x2H1,
↪#7aX3x2r5H1,#8H0x0AX2;+0:1]-[#1H0x0X1,#6x0X3H1,#6x0X4H2,#6x0X4H3,#6x2X3r6H1,
↪#6x2X4H2r6,#8H0x0X2;A:2]  -->  [#6x0AH1X3,#6x0AH2X4,#6x0AX4H3,#6x2AH1X3r6,
↪#6x2AH2X4r6,#6aH1x2r5X3,#6aH1r6x2X3,#7aH1x2r5X3,#8H0x0AX2;+0:1]-[#1H0x0X1,#6x0X3H1,
↪#6x0X4H2,#6x0X4H3,#6,#6x2X4H2r6,#8H0x0X2;A:2]
```

(continues on next page)

(continued from previous page)

```
Accepted!
--------------------------------------------------------------------------------
↪----
Iteration:  8
Attempting to change SMIRKS #2
[#6x0,#6x2r6;+0;A;H1;X3:1]=[#6,#6x2r6H1X3,#8H0x0X1;+0;A:2]  -->  [#6x0,#6x2r6;+0;A;H1;
↪X3:1]=[#6,#6x2H1X3,#8H0x0X1;+0;A:2]
Accepted!
--------------------------------------------------------------------------------
↪----
Iteration:  9
Attempting to change SMIRKS #0
[#6x0AH1X3,#6x0AH2X4,#6x0AX4H3,#6x2AH1X3r6,#6x2AH2X4r6,#6aH1x2r5X3,#6aH1r6x2X3,
↪#7aH1x2r5X3,#8H0x0AX2;+0:1]-[#1H0x0X1,#6x0X3H1,#6x0X4H2,#6x0X4H3,#6,#6x2X4H2r6,
↪#8H0x0X2;A:2]  -->  [#6x0X3AH1,#6x0X4AH2,#6x0AX4H3,#6x2X3AH1r6,#6x2X4AH2r6,
↪#6aX3x2r5H1,#6aX3r6x2H1,#7aX3x2r5H1,#8H0x0AX2;+0:1]-[#1H0x0X1,#6x0X3H1,#6x0X4H2,
↪#6x0X4H3,#6,#8H0x0X2;A:2]
Accepted!
--------------------------------------------------------------------------------
↪----

 Label                | SMIRKS
================================================================================
 zz_sing              | [#6x0X3AH1,#6x0X4AH2,#6x0AX4H3,#6x2X3AH1r6,#6x2X4AH2r6,
↪#6aX3x2r5H1,#6aX3r6x2H1,#7aX3x2r5H1,#8H0x0AX2;+0:1]-[#1H0x0X1,#6x0X3H1,#6x0X4H2,
↪#6x0X4H3,#6,#8H0x0X2;A:2]
--------------------------------------------------------------------------------
 zz_aromatic          | [*;a;+0;H1:1]:;@[#6r5,#6,#7r5:2]
--------------------------------------------------------------------------------
 zz_double            | [#6x0,#6x2r6;+0;A;H1;X3:1]=[#6,#6x2H1X3,#8H0x0X1;+0;A:2]
--------------------------------------------------------------------------------
```

[7]:
```
print_smirks(smirks10)
```

```
 Label                | SMIRKS
================================================================================
 zz_sing              | [#6x0X3AH1,#6x0X4AH2,#6x0AX4H3,#6x2X3AH1r6,#6x2X4AH2r6,
↪#6aX3x2r5H1,#6aX3r6x2H1,#7aX3x2r5H1,#8H0x0AX2;+0:1]-[#1H0x0X1,#6x0X3H1,#6x0X4H2,
↪#6x0X4H3,#6,#8H0x0X2;A:2]
--------------------------------------------------------------------------------
 zz_aromatic          | [*;a;+0;H1:1]:;@[#6r5,#6,#7r5:2]
--------------------------------------------------------------------------------
 zz_double            | [#6x0,#6x2r6;+0;A;H1;X3:1]=[#6,#6x2H1X3,#8H0x0X1;+0;A:2]
--------------------------------------------------------------------------------
```

3c. Continue removing decorators

Now we will continue trying to reduce the SMIRKS. Note, in this case we set verbose to False and just print the final SMIRKS since 3,000 is a lot of steps.

[8]:
```
smirks3k = bond_smirksifier.reduce(max_its=3000, verbose=False)
print_smirks(smirks3k)
```

```
 Label                | SMIRKS
==========================================================================================
 zz_sing              | [*:1]~[*:2]
------------------------------------------------------------------------------------------
 zz_aromatic          | [*:1]:[*:2]
------------------------------------------------------------------------------------------
 zz_double            | [*:1]=[*:2]
------------------------------------------------------------------------------------------
```

4.3.4 4. What have we learned for the future

Is there a systematic way to remove decorators that doesn't introduce too much human wizardary?

Right now the removal of decorators is stochastic, so you don't guarentee the same SMIRKS will be created for the same clustering of atoms every time. This is because it is possible to have multiple combinations of decorators that which maintain the same clustering.

We could consider looking for differences in the ClusterGraphs and start by removing any decorators that are in common for all SMIRKS since those are clearly not distinguishing features. However, it seems unlikely that a systematic removal wouldn't be biased by the choices of the human who chose the order for checking the removal of the decorators.

[ ]:
```

```

Let us know if you have any problems or suggestions through the `chemper` issue tracker.

PYTHON MODULE INDEX

C

INDEX

G

H

S

T

V