Chemical Graph Project: Inferring Cyclic Chemical Graphs

`mol-infer/Cyclic`

March 26, 2021

Contents

Chapter 1

Module 1: Calculating a Feature Vector from an SDF File

1.1 Introduction

This note serves as a manual and explains the procedures to run Module 1 of the `mol-infer/Cyclic`project. The input and output of Module 1 are as follows.

Input: A set $D = \{G_1, G_2, \ldots, G_p\}$ of cyclic chemical graphs.

Output: A set $\mathcal{F}(D) \triangleq \{f(G_1), f(G_2), \ldots, f(G_p)\}$ of feature vectors, such that $f(\cdot)$ is a feature vector of chemical graphs as details in the accompanying article [1].

The output is written to a csv (comma-separated value) file. This csv file is used in Module 2 of the project.

The remainder of this note is organized as follows

- Section 2.2: Summary of essential terminology, as well, as the file organization in this package.
- Section 2.3: A short computational example.
- Section 2.4: Detailed explanations of the program's input and output.

1.2 Preliminaries

1.2.1 Terminology

Chemical Graph. A **graph** is an abstract combinatorial construction comprising a set of **nodes** and a set of **edges**, where an edge is an unordered pair of nodes. A **cycle** in a graph is a sequence of nodes such that except for the first and the last node, each node is unique, and there is an edge for each pair of consecutive nodes in the sequence.

A graph where each node is assigned a chemical element (such as carbon, nitrogen, oxygen, etc.) and each edge is assigned a multiplicity between 1 and 4, is called a **chemical graph**.

Descriptor. A **descriptor** is a numerical value that indicates a certain characteristic of a chemical graph. In this project, among others, descriptors include the number of non-hydrogen atoms, the number of atoms in the core of the chemical graph, the core height, etc. For a complete list of descriptors, please refer to the accompanying article [1].

Feature vector. A vector that comprises the numerical values for the descriptors of a chemical graph.

1.2.2 File Structure

The following set of files accompany this note.

- `Makefile`: A makefile for compiling the programs.
- `cycle_checker.cpp`: Source code written in C++ that for a given chemical compound checks if the chemical graph contains a cycle or not
- `eliminate.py`: A Python script that screens chemical compounds that are not considered under this project, such us inorganic compounds with less than four carbon atoms, that contain charged atoms, etc.
- `fv_ec.cpp`: Source code written in C++ of the mail program of Module 1, calculating a feature vector.
- `fv_proj.cpp`: Source code (C++) of a program that given a feature vector function f calculated over a set D of chemical graphs, and a set D' of chemical graphs that does not necessarily have the same descriptors as D does, calculates the set $\mathcal{F}(D')$ of feature vectors projected onto the domain of f. This is an auxiliary program, and usually not essential to the flow of the entire project.
- Folder `data` Contains sample input and output files used to test and demonstrate the execution of the programs in Module 1. The files in this folder as as follows.
  - `sample1.sdf`: An SDF file that contains a single chemical compound. (Please check Section 2.4 for more details on SDF files.)

- `sample1_eli.sdf`: An SDF file obtained as the output of the Python script `eliminate.py` when invoked on the file `sample1.sdf`. The contents of the files `sample1.sdf` and `sample1_eli.sdf` should be identical.
- `sample1.csv`: Contains a single feature vector constructed from the file `sample1_eli.sdf`.
- `sample2.sdf`: An SDF file that contains data on 175 chemical graphs.
- `sample2_eli.sdf`: An SDF file obtained as the output of the Python script `eliminate.py` when invoked on the file `sample2.sdf`. The contents of the files `sample2.sdf` and `sample2_eli.sdf` should be identical.
- `sample2.csv`: Contains the set of feature vectors constructed from the file `sample2_eli.sdf`.
- `sample1_on_2.csv`: Contains a single feature vector whose values are calculated from the file `sample1_eli.sdf`, however, the dimensions of the vector are projected on the domain of the feature vector obtained from the file `sample2_eli.sdf`.

1.3 Execution Example

1.3.1 Validation of the Data

Data on chemical compounds (equivalently, chemical graphs) is stored in a standard SDF file (more information on the structure of SDF files is given in Section 2.4). Each chemical graph G must satisfy the following conditions

(i) G must contain a cycle;

(ii) G must contain at least four carbon atoms, none of the atoms is allowed to be charged, and each atom must have atomic mass as listed in Section 1.4.3; and

(iii) Must not include an aromatic edge.

The Python script `eliminate.py` included in Module 1 can be used to remove the graphs that do not satisfy condition (ii).For condition (iii), the user must confirm whether it is satisfied or not on his/her own.

Confirming that a chemical graph contains a cycle. Please use the program compiled from the source file `cycle_checker.cpp` included in Module 1 to confirm whether each chemical graph in a given SDF file contains a cycle.

To compile the program, the included `Makefile` can be used by issuing the following command in the command prompt.

```
$ make CHECKER
```

In case the `make` command is not available on the system, then the program can be compiled in the following way.

```
$ g++ -std=c++11 -Wall -O3 -o CHECKER cycle_checker.cpp
```

In order to check if a given SDF file `input.sdf` contains a chemical graph that does not include a cycle by issuing the following command on the terminal.

```
$ ./CHECKER input.sdf
```

- If all chemical graphs have cycles (i.e., all satisfy (i)), then the program CHECKER does not output any message. In this case, one can go to the next step.
- Otherwise, (i.e., there is a chemical graph that does not satisfy (i)), CID of such a chemical graph is output. Before going to the next step, such a graph must be removed from the SDF file manually.

Elimination of chemical graphs that are out-of-scope. To check whether each chemical graph in a given SDF file satisfies condition (ii) or not, please use the Python script named `eliminate.py`. The script generates a new SDF file that consists of all chemical graphs in the input SDF file that satisfy (ii).

To use `eliminate.py`, execute the following command.

```
$ python eliminate.py input.sdf
```

If the `input.sdf` constains a chemical graph that does not satisfy (ii), the CID is output.

After the execution of `eliminate.py`, a new SDF file `input_eli.sdf` is output. The file consists of all chemical graphs in `input.sdf` that satisfy condition (ii). This means that, if all chemical graphs in `input.sdf` satisfy (ii), `input.sdf` and `input_eli.sdf` are equivalent.

1.3.2 Calculating a Feature Vector

Please use the program compiled from the source file `fv_ec.cpp` included in Module 1 to calculate feature vectors for an SDF file such that every chemical satisfies conditions (i), (ii) and (iii).

To compile the program, the included `Makefile` can be used by issuing the following command in the command prompt.

```
$ make FV_ec
```

In case the `make` command is not available on the system, then the program can be compiled in the following way.

```
$ g++ -std=c++11 -Wall -O3 -o FV_ec fv_ec.cpp
```

In order to calculate feature vectors from `input_eli.sdf` and to output the result in `output.csv`, issue the following command on the terminal.

```
$ ./FV_ec input_eli.sdf output.csv
```

The program `FV_ec` prints on the terminal instructions on how to provide the arguments and halts if the arguments are not provided appropriately.

1.3.3 Calculating a Feature Vector from Other SDF (Not Mandatory)

The mapping f that transforms a chemical graph into a feature vector is constructed from a given set D of chemical graphs. To calculate a feature vector of a chemical graph in another set $D' \neq D$ using f, please use the program compiled from `fv_proj.cpp`.

To compile the program, the included `Makefile` can be used by issuing the following command in the command prompt.

```
$ make FV_proj
```

In case the `make` command is not available on the system, then the program can be compiled in the following way.

```
$ g++ -std=c++11 -Wall -O3 -o FV_proj fv_proj.cpp
```

Let `descriptor.csv` be the name of the csv file that is obtained by executing `FV_ec` on the original SDF containing D. That is, the mapping f constructed from D. Let `input.sdf` be the SDF file that contains the data on $D' \neq D$. To calculate $\mathcal{F}(D')$ and obtain the result in `output.csv`, issue the following command on the terminal.

```
$ ./FV_proj descriptor.csv input.sdf output.csv
```

For example, one can run the program in the following way, using the sample files in Module 1.

```
$ ./FV_proj data/sample2.csv data/sample1.sdf data/sample1_on_2.csv
```

It is not mandatory to execute `FV_proj` to proceed to Module 2 and afterwards.

Let us describe an example of when to use `FV_proj`. Suppose that a neural network has been constructed from `descriptor.csv` in Module 2, and the neural network can be used to predict the value of a certain chemical property, say π. When one uses the neural network to predict the value of π for a chemical graph in `input.sdf`, the chemical graph must be converted into a feature vector by the mapping f. The program `FV_proj` can be used for this.

1.4 Details in the Input and Output of the Program

1.4.1 Input

The programs in Module 1 use SDF (Structure Data File), a standard format, for input. For the detail of SDF, splease check the following reference:

- `http://help.accelrysonline.com/ulm/onelab/1.0/content/ulm_pdfs/direct/reference/ctfileformats2016.pdf` (accessible on Feb 1, 2021)

1.4.2 Output

The output is in an original FV (Feature Vector) format, which is just a CSV file so that can be opened by many spreadsheet softwares. The first line shows the components of FV and the following lines show the values for those components of FV. For example, let us have a look at the FV file `sample1.csv` that is obtained by running `FV_ec` for `sample1.sdf`.

```
CID,n,cs,ch,bl_2,ms,dg_co_1,dg_co_2,dg_co_3,dg_co_4,dg_nc_1,\
dg_nc_2,dg_nc_3,dg_nc_4,bd_co_2,bd_co_3,bd_in_2,bd_in_3,\
bd_ex_2,bd_ex_3,ns_co_C3,ns_co_C2,ns_nc_O1,ns_nc_N1,ns_nc_C2,ns_nc_C3,\
ec_co_C2_C3_2,ec_co_C2_C2_1,ec_co_C2_C3_1,ec_co_C2_C2_2,\
ec_in_C2_C3_1,ec_in_C3_C2_1,\
ec_ex_C3_N1_1,ec_ex_C3_C3_1,ec_ex_C3_O1_1,ec_ex_C3_O1_2,nsH
6140,12,6,4,1,128.333,0,5,1,0,3,1,2,0,3,0,0,0,1,0,1,5,2,\
1,1,2,1,2,1,2,1,1,1,1,1,1,11
```

The symbol \ at the end of a line indicates that there is no line break between the two lines. Here is the overview of descriptors. See [1] for details.

- **CID:** Compound ID. In this example (`sample1.sdf`), it is 6140. The molecule is Phenylalanine, which is taken from `https://pubchem.ncbi.nlm.nih.gov/compound/6140`.
- **n:** Number of atoms except for the hydrogen.
- **cs:** Number of atoms in the core.
- **ch:** Core height.
- **bl:** Number of 2-leaves.
- **ms:** Average molecular mass defined by $\mathrm{ms} \triangleq \frac{1}{n}\sum_a \lfloor 10 \cdot \mathrm{mass}(a)$, where $\mathrm{mass}(a)$ represents the mass of an atom a.
- **dg_co_1, . . . , dg_co_4:** Number of atoms in the core such that the degree is 1, 2, 3 and 4, resp.
- **dg_nc_1, . . . , dg_nc_4:** Number of atoms not in the core such that the degree is 1, 2, 3 and 4, resp.

- **bd_co_2, bd_co_3:** Number of double and triple bonds in the core paths, resp.

- **bd_in_2, bd_in_3:** Number of double and triple bonds in the internal paths, resp.

- **bd_ex_2, bd_ex_3:** Number of double and triple bonds in the external paths, resp.

- **ns_co_Xd:** Number of atoms in the core such that the element symbol is X and the degree is d. For example, `ns_co_C3` represents the number of carbon atoms in the core such that the degree is 3.

- **ns_nc_Xd:** Number of atoms not in the core such that the element symbol is X and the degree is d.

- **ec_co_Xx_Yy_2, ec_co_Xx_Yy_3:** Number of double and triple bonds in the core paths such that the end nodes have X and Y as element symbols and the degrees x and y, resp. For example, `ec_co_C2_C3_2` represents the number of double bonds in the core paths such that both end nodes are carbon atoms and have the degrees 2 and 3, resp.

- **ec_in_Xx_Yy_2, ec_in_Xx_Yy_3:** Number of double and triple bonds in the internal paths such that the end nodes have X and Y as element symbols and the degrees x and y, resp.

- **ec_ex_Xx_Yy_2, ec_ex_Xx_Yy_3:** Number of double and triple bonds in the external paths such that the end nodes have X and Y as element symbols and the degrees x and y, resp.

- **nsH:** Number of the hydrogen atoms.

For the descriptors whose names begin with `ns_` and `ec_`, only those appearing the input SDF are written in the output CSV file.

1.4.3 Attention

The mass of each atom is hard-coded in the program. They are written in the function `init_MassMap()` in `fv_ec.cpp` as follows. If one needs to change values or to add another atoms, edit the source code directly and compile again.

```
M["B"]  = 108;
M["C"]  = 120;
M["O"]  = 160;
M["N"]  = 140;
M["F"]  = 190;
M["Si"] = 280;
M["P"]  = 310;
M["S"]  = 320;
M["Cl"] = 355;
M["V"]  = 510;
M["Br"] = 800;
M["Cd"] = 1124;
M["I"]  = 1270;
M["Hg"] = 2006;
M["Pb"] = 2072;
M["Al"] = 269;
```

Chapter 2

Module 2: Training an Artificial Neural Network

2.1 Introduction

In this article, we explain how to proceed to Module 2, i.e., to training an artificial neural network (ANN) in our project (`mol-infer/Cyclic`).

We denote by $D_\pi = \{G_1, G_2, \ldots, G_p\}$ a given set of chemical graphs, and by f a function that maps a chemical graph to a feature vector. We define $\mathcal{F}(D_\pi) \triangleq \{f(G_1), f(G_2), \ldots, f(G_p)\}$. Let us denote by π a chemical property that is considered. For example, π can be boiling point, heat of combustion, Kow, and so on. The input and output of the module are summarized as follows.

Input: A set $\mathcal{F}(D_\pi) = \{x_1, x_2, \ldots, x_p\}$ of molecule feature vectors, a set $\{a(x_1), a(x_2), \ldots, a(x_p)\}$ of observed values $a(x_i)$ for each $G_i \in D_\pi$ (and $x_i = f(G_i)$) for the property π, and values specifying the the architecture of a desired ANN, that is, the number of hidden layers, and the number of nodes in each hidden layer.

Output: An ANN that has architecture as specified in the input, and that estimates $a(x)$ "well" for many feature vectors $x \in D_\pi$.

Concretely, the output consists of weights of the arcs and biases of the nodes in the constructed neural network.

The article is organized as follows.

- Section 2.2: Some basic terminology and the roles of the files in the package are explained.
- Section 2.3: A brief example of executing the programs.
- Section 2.4: Details about the input and the output of the programs.

2.2 Preparation

2.2.1 Basic Terminology

Feature vector. A numerical vector that represents features of a molecule. Each dimension corresponds to a *descriptor* that describes such features as the number of atoms of different chemical elements (e.g., carbon, nitrogen, oxygen, etc), the molecular mass averaged over the number of atoms, etc.

Artificial neural network (ANN). A well-known model in machine learning. In this project, we use ANNs to solve the *regression* problem. In this problem, given a data set that consists of numerical vectors and the "correct" values assigned to each of the vectors, we are asked to predict values for vectors not in the data set as accurately as possible. In our project, the numerical vectors are feature vectors that are extracted from molecules and the values assigned to the vectors are property values of a prescribed chemical property. The number of dimensions is assumed to be equal over all the feature vectors.

The architecture of ANNs that we use in the project is restricted to feed-forward networks, which can be represented by a directed acyclic graph. Figure 3.1 shows an example.

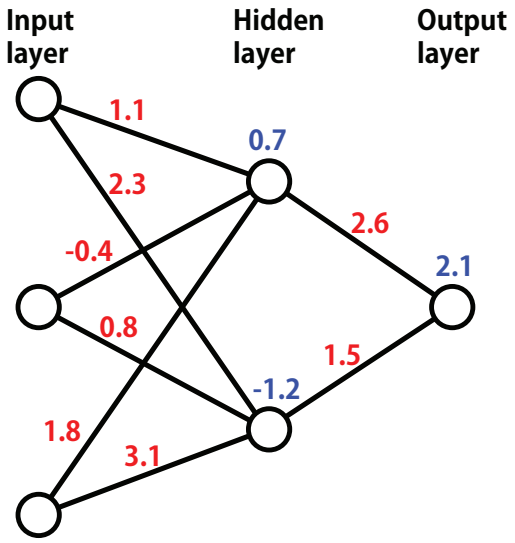


Figure 2.1: An example of an ANN. A red value indicates a weight, whereas a blue value indicates a bias. The direction of each arc is from left to right.

Layers of an ANN. In the multi-layered perceptron model of ANNs, an ANN consists of several *layers*.

- The first layer is the *input layer*. Each node in the input layer is associated with a dimension of a feature vector (i.e., a descriptor). Then the number of nodes in the input layer is equal to the number of dimensions of a feature vector.
- The subsequent layers (except the last one) are called *hidden layers*. A node in a hidden layer outputs a value based on the input values, where the input values are the outputs of all nodes

in the previous layer.

- The last layer is the *output layer* that has only one node. The output of this node is regarded as the output of the ANN.

In the ANN of Figure 3.1, the input layer has three nodes. This means that a feature vector should be have three dimensions as well. The ANN has one hidden layer that has two nodes. The output layer has exactly one node.

Weights. In ANNs, each edge is assigned a numerical value, called a *weight*. The weights are determined by *training* the ANN, based on the given data set.

In Figure 3.1, the weights of edges are indicated in red.

Biases. In ANNs, each node in the hidden layers and the output layer is assigned a numerical value, which is called a *bias*. The biases are determined together with the weight by training the ANN based on the given data set.

In Figure 3.1, the biases of nodes are indicated in blue.

Activation function. In ANNs, each node except the output one is assigned an activation function. This function computes the output of the node according to the weighted sum of the input values minus the bias of the node.

In this project, we use the Rectified Linear-Unit (ReLU) function for the activation function of each node in the hidden layers. This choice is due to Module 3, where it is assumed that the ReLU function is used as an activation function for ANNs in Module 2.

The process of deciding (appropriate) weights and biases from a given data is called a *learning*.

2.2.2 Files in the package

Table 2.1 shows the list of files in the package of Module 2.

Table 2.1: List of files in the package of Module 2

File	Description
`mol-infer_ANN.py`	A Python script to train an ANN **(To proceed to Module 3,** **it is necessary to run this script)** • Used non-standard libraries: `numpy, pandas, scikit-learn`
`predict_values.py`	A supplementary Python script that predicts the value of a chemical property for molecules in a given data set (It is not necessary to run this script) • Used non-standard libraries: `numpy, pandas`

`Manual_Module_2_Cyclic_jp.pdf`
`Manual_Module_2_Cyclic_jp.tex`
`fig/ANN_sample_jp.eps`
	PDF, LaTeX source of the manual
	and the image file (Japanese version)

`Manual_Module_2_Cyclic_en.pdf`
`Manual_Module_2_Cyclic_en.tex`
`fig/ANN_sample_en.eps`
	PDF, LaTeX source of the manual
	and the image file (English version)

Data files for BP (boiling point) [3]	
`data/BP.sdf`	SDF file containing molecule data. This file is not explicitly treated in Module 2.
`data/BP_fv.csv`	File containing feature vectors that are obtained by applying Module 1 to `BP.sdf`
`data/BP_value.csv`	File containing BP values of molecules
`data/BP_ANN.LOG`	Log of executing `mol-infer_ANN.py` for the data set that consists of `BP_fv.csv, BP_value.csv`
`data/BP_ANN_biases.txt` `data/BP_ANN_weights.txt`	The biases and weights of the constructed NN

(Cont. of Table 2.1)

File	Description
Data files for HC (heat of combustion) [3]	
data/HC.sdf	(All are analogous to BP)
data/HC_fv.csv	
data/HC_value.csv	
data/HC_ANN.LOG	
data/HC_ANN_biases.txt	
data/HC_ANN_weights.txt	
Data files for KOW (log Kow) [3]	
data/KOW.sdf	(All are analogous to BP)
data/KOW_fv.csv	
data/KOW_value.csv	
data/KOW_ANN.LOG	
data/KOW_ANN_biases.txt	
data/KOW_ANN_weights.txt	
Data files for MP (melting point) [3]	
data/MP.sdf	(All are analogous to BP)
data/MP_fv.csv	
data/MP_value.csv	
data/MP_ANN.LOG	
data/MP_ANN_biases.txt	
data/MP_ANN_weights.txt	

2.3 A Quick Start

Training an ANN. The following command constructs an ANN from the dataset such that

- the set of feature vectors is written in `data/BP_fv.csv`; and
- the set of property values (BP in this case) is written in `data/BP_value.csv`

```
$ python mol-infer_ANN.py data/BP_fv.csv data/BP_value.csv output 20 10
```

The constructed NN has two hidden layers, where the numbers of nodes in these layers are 20 and 10, respectively. The weights of arcs are output to the file `output_weights.txt` and the biases of nodes are output to the file `output_biases.txt`. **The files containing weights and biases are used in Module 3**.

Predicting the property value of a given molecule. The property value of a molecule can be predicted by using a trained ANN.

The following command invokes a Python script that reads the information (edge weights and node biases) of a trained ANN, and then uses each of the feature vectors given in the file `data/BP_fv.csv` as input to the ANN's prediction function. The edge weights of the trained ANN are read from the file `output_weights.txt`, and node biases are read from the file `output_biases.txt`. The output results are written to the file `predicted.txt`.

The symbol \ at the end of the first line indicates that there is no line break between the two lines, that is, they should be input without pressing the "enter" key.

```
$ python predict_values.py output_weights.txt output_biases.txt \
    data/BP_fv.csv predicted.txt
```

- Given an SDF file, one can generate the feature vectors of the molecules in the SDF by using the program in Module 1.
- This script is supplementary, and it is not directly used in Modules 3 and 4.

2.4 Details in the Input and Output of the Program

2.4.1 Input

Feature vectors

The feature vectors should be written in a so-called **FV format** to a csv (comma-separated value) file. The feature vector generator in Module 1 generates a csv file in the FV format from a given SDF. As long as one uses the csv file generated in Module 1, no problem should occur in Module 2.

We explain the structure of the FV format using a simple example.

```
CID,n,cs,ch,nsH
244,8,6,2,8
307,10,6,4,8
657014,11,7,1,18
16704,9,9,0,10
```

- In the first line, the descriptor names should be written in comma-separated style.
- The first descriptor **must be** CID (Compound ID). The CID values are not used to train an ANN, just for identification.
- In the above example, four descriptors are introduced: `n` (the number of atoms except hydrogens), `cs` (core-size), `ch` (core-height), and `nsH` (the number of hydrogens).
- In each of the subsequent lines, the CID and the feature vector of one molecule should be specified in comma-separated style.
- The molecules **need not be** ordered according to CID.

Property values

The property values should be described in a csv file as follows.

```
CID,a
307,11.2
244,-0.5
657014,98.124
16704,-12.8
117,5.3
```

- **The first line should be `CID,a`.**
- In each of the subsequent lines, CID and the property value of one molecule should be written in comma-separated style.
- The molecules need not be ordered with respect to CID.

2.4.2 Execution

We use the Python script `mol-infer_ANN.py` to train an ANN, as outlined in Section 2.3.

Arguments

We here revisit the commands explained in Section 2.3.

```
$ python mol-infer_ANN.py data/BP_fv.csv data/BP_value.csv output 20 10
```

The meaning of each of the command-line parameters is as follows:

- Parameter 1: A csv file the stores the feature vectors of chemical compounds,
- Parameter 2: A csv file that stores observed values of a chemical property/activity of chemical compounds,
- Parameter 3: Filename where the weights and biases of a trained ANN will be stored,
- Parameter 4: The number of nodes of hidden layers in the ANN, each layer separated by a space.

If the Python script is invoked without supplying command-line arguments, or in the case that there is a significant error in the arguments, then a brief description of the command-line parameters will be printed.

A verbatim output of the output of the program when invoked without any command-line parameters. The backslash symbol (\) indicates that there is no actual line break.

```
$ python mol-infer_ANN.py

usage: mol-infer_ANN.py (TrSet_FeatureVector.csv)(TrSet_TargetValue.csv) \
    (Output_Name)(NN_Architecture)

  - TrSet_FeatureVector.csv ... \
      The CSV file generated by our feature vector generator in Module 1.
  - TrSet_TargetValue.csv ... \
      The CSV file that contains target values.
    = The first column must contain CIDs.
    = The second column must contain target values and be named 'a'.
    = All CIDs appearing in TrSet_FeatureVector.csv must be contained in \
      TrSet_TargetValue.csv.
  - OutputName_{biases,weights}.txt will be output. \
      They contain data on the constructed ANN.
  - NN_Architecture ... Number of nodes in hidden layer(s).
```

If the program is invoked with a proper set of command-line parameters, then the training process of an ANN will begin.

To evaluate the performance of the trained ANN, we use 5-fold cross-validation. The ANN that achieved the highest coefficient of determination (R^2 score) for the test set over the five trials is selected, and its weights and biases are stored in a file. Using the above example of parameter

values, where the value of parameter 3 is set to be `output`, the following two files will be written to disk:

- `output_weights.txt` (stores the weights)
- `output_biases.txt` (stores the biases)

These two files are necessary in Module 3.

Caution on the data set The training set for training the ANN is given in the following two files

- a csv file with feature vectors of chemical compounds, and
- a csv file with observed values for a chemical property/activity.

Both files include the CID number of the corresponding chemical compound. One important point to note is that

each of the CIDs in the former file must appear in the latter

however, the opposite is not enforced. In other words, there might be a chemical compound for which an observed value of some property/activity is listed in the latter file, but whose feature vector is not included in the former. Such values will be silently ignored.

Hyper-parameters

We use the Python library `scikit-learn`[1] and its `MLPRegressor` tool to do the training of an ANN as a multi-layered perceptron. After line 133 in the Python script `mol-infer_ANN.py`, an instance of an `MLPRegressor` is initialized, and its hyperparameters can be changed at this point. Some of the various parameters are set as follows:

- `activation`: `'relu'`
  Attention: If the trained neural network is to be used in Module 3 and afterward, the only possible choice here is `'relu'` (Rectified Linear Unit Function - ReLU).
- `alpha`: 10^{-5}
- `early_stopping`: `False`
- `hidden_layer_sizes`: Passed as a command-line parameter at invocation
- `max_iter`: 10^{10}
- `random_state`: 1
- `solver`: `'adam'`

[1] `https://scikit-learn.org/stable/`

2.4.3 Output

We revisit the command issued as below, and explain the resulting output.

```
$ python mol-infer_ANN.py data/BP_fv.csv data/BP_value.csv output 20 10
```

Standard output

When the above command is issued on the command prompt, the results of the computation process will be displayed on the standard output. An example of this output is stored in the file `data/BP_ANN.LOG` which is included together with the program package and this note.

```
src/preparation/BP_fv.csv contains 181 vectors for 107 (=CID+106) features.
src/preparation/BP_value.csv contains 230 target values.
n range = [5,30]
a range = [31.5,470.0]
#instances = 181
#features = 106

D1: train 144, test 37
training time: 3.187196731567383
R2 score train = 0.9935319077425955
R2 score test = 0.7855694851759929
R2 score all = 0.9599908495442584
MAE score train = 3.8570127089010384
MAE score test = 19.756408746147212
MAE score all = 7.10716548999556

D2: train 145, test 36
training time: 5.0139079093933105
R2 score train = 0.9930416804444452
R2 score test = 0.6390572625074291
R2 score all = 0.9339056805642507
MAE score train = 3.7281172621746888
MAE score test = 27.996544836634186
MAE score all = 8.554986834995363

D3: train 145, test 36
training time: 4.264358758926392
R2 score train = 0.9961346879653636
R2 score test = 0.8566979846124056
R2 score all = 0.9637772344809027
MAE score train = 2.870368503215682
MAE score test = 22.925217956024927
MAE score all = 6.8591783391335435
```

```
D4: train 145, test 36
training time: 2.9935202598571777
R2 score train = 0.9909067339023991
R2 score test = 0.8390994594153819
R2 score all = 0.9669036994338265
MAE score train = 4.470398694611737
MAE score test = 18.691072072382227
MAE score all = 7.298819918919681

D5: train 145, test 36
training time: 4.9648661613464355
R2 score train = 0.9946933391220482
R2 score test = 0.8479544758088942
R2 score all = 0.9574271159388575
MAE score train = 3.352287445970431
MAE score test = 21.80973582058148
MAE score all = 7.023382150312958
0.9935319077425955 0.7855694851759929 0.9599908495442584 3.187196731567383
0.9930416804444452 0.6390572625074291 0.9339056805642507 5.0139079093933105
0.9961346879653636 0.8566979846124056 0.9637772344809027 4.264358758926392
0.9909067339023991 0.8390994594153819 0.9669036994338265 2.9935202598571777
0.9946933391220482 0.8479544758088942 0.9574271159388575 4.9648661613464355
Average time = 4.08476996421814
Average R2 test score = 0.7936757335040208
Average MAE test score = 22.235795886354005
```

- First, the number of feature vectors, as well as the number of descriptors are output. Next comes the range (`n range`, in the above example 5 – 30) of the number of non-hydrogen atoms (descriptor `n`) in the chemical compounds in the training set, followed by the range (`a range`, 31.5 – 470.0 above) of the observed chemical property/activity in the training data (in this case, boiling point).

- Following, the summary of each of the five runs of the 5-fold cross-validation is printed.

- Finally, the average of the calculation time, R^2 score of the test set and MAE score of the test set over the five folds are printed.

- In the above example, the ANN constructed in the third of the five folds had the highest R^2 score of 0.856697..., and therefore the values of its weights and biases are written in the files `output_weights.txt` and `output_biases.txt`, respectively.

- Note that a copy of each of the above files, `output_weights.txt` and `output_biases.txt`, is included in the set of files together with this note.

Edge weights

Following we explain the contents of the file `mol-infer_ANN.py` that contains the values of the weights of the ANN.

For simplicity, let us assume that the ANN depicted in Figure 3.1 has been obtained after training. The contents of the file with the weights of this ANN would be as follows.

```
3 2 1
1.1 2.3
-0.4 0.8
1.8 3.1
2.6
1.5
```

- The first row gives the architecture of the ANN, that is the number of nodes in each of its layers. starting with the input layer, then the hidden layers, and finally the output layer.
- Starting from the second line and onward, follow the weights of the ANN. Each row gives the weights of the edges going out of a single node.

Biases of nodes

In a similar manner, we give an explanation of the file that contains the biases of the ANN obtained by executing the Python script `mol-infer_ANN.py`.

For sake of simplicity, again consider the ANN depicted in Figure 3.1. The values for the node biases of this ANN would be given as follows.

```
0.7
-1.2
2.1
```

Each row contains the bias value of a single node. Note that the nodes of the output layer do not have biases.

Chapter 3

Module 3: Inferring a 2-Lean Cyclic Chemical Graph with Bounded Branch-Height from a Trained ANN Using MILP

3.1 Outline

This note explains how to use an implementation of a mixed-integer linear programming (MILP) formulation that can infer a vector of graph descriptors given a target value and the weights and bias values of a trained artificial neural network (ANN).

The MILP is implemented in Python, using the PuLP modeling module of the COIN-OR package [6, 7, 8, 9].

To begin with, we give a list of the files that accompany this note.

- Folder `source_code`
  A folder containing four Python scripts that implement an MILP formulation for inferring feature vectors of cyclic chemical graphs from a trained ANN, and files containing minimum and maximum values of each descriptor in the MILP formulation.

  - `ann_inverter.py`
    An implementation of an MILP formulation for the Inverse problem on ANNs [2].
  - `cyclic_graphs_MILP_ec_id.py`
    A Python script that contains functions to initialize the variables and prepare the constraints for an MILP formulation for inferring cyclic chemical graphs with a prescribed topological structure [1].
  - `infer_cyclic_graphs_ec_id.py`
    A Python script that prepares the data and executes the MILP formulation for given input data. Further details on the use of this script are given in Section 3.4.

- `read_instance_BH_cyclic.py`
  A Python script that contains necessary functions to read the topological specification from a given textual file.
- Folder `topological_description`
  A folder containing six textual files each giving a chemical specification as detailed in [1].
  * `instance_a.txt`
  * `instance_b1.txt`
  * `instance_b2.txt`
  * `instance_b3.txt`
  * `instance_b4.txt`
  * `instance_c.txt`
  * `instance_d.txt`
- Folder `ANN`
  A folder containing information on trained artificial neural networks (ANNs) for three target properties: Boiling point (Bp), Melting point (Mp), and Octanol/Water Partition Coefficient (Kow). For each of the above three properties, `property` $\in \{$`BP`, `MP`, `KOW`$\}$ three files are provided:
  * `property_desc.csv`
    A comma-separated value file containing descriptors used in the training of the ANN.
  * `property_biases.txt`
    A file containing the values of the biases of a trained ANN.
  * `property_weights.txt`
    A file containing the values of the weights of a trained ANN.

  For each of the files, the data format is explained in Section 3.3, and an actual example is given in Section 3.4.

The remaining of this note is organized as follows. Section 3.2 gives an explanation of the used terms and notation. Section 3.3 explains the input and output data of the program, and Section 3.4 gives a concrete example of input data and the results form the computation.

3.2 Terms and Notation

This section explains the terms and notation used in this note.

- **Feature vector**
  A *feature vector* stores numerical values of certain parameters, called *descriptors*. In this work, we choose graph-theoretical descriptors, such as number of non-hydrogen atoms, number of vertices of certain degree, etc.
- **Artificial neural network - ANN**
  Artificial neural networks are one of the methods in machine learning. They provide a means to construct a correlation function between pairs of feature vectors as input and target data as output.

- **Input, hidden, and output layer**
  We deal with the multilayer perceptron model of feed-forward neural networks. These neural networks are constructed of several *layers*. First comes the *input layer*, where each neuron takes as input one value of the feature vector. Next come the *hidden layers*, where the values from the input layer are propagated in a feed-forward manner, such that each node in one layer is connected to all the nodes of the next layer. Finally, the output is delivered at the *output layer*. We deal with predicting the value of a single target, and hence we assume that the output layer comprises a single node.

- **Weights**
  Each edge connecting two nodes in an ANN is assigned a real value, called a *weight*. Part of the *learning* process of ANNs is to determine values for each of the weights based on known pairs of feature vectors and target values.

- **Biases**
  Each node of an ANN except for the nodes in the input layer is assigned a real value, called a *bias*, which, just like the edge weights, is determined through the learning process.

- **Activation function**
  In an ANN, each node produces an output as a function, called the *activation function*, of its input. We assume that each node has the Rectified Linear-Unit (ReLU) function as its activation function, which can be expressed exactly in the MILP formulation for the inverse problem on ANNs [2].

- **Mixed-Integer Linear Programming (MILP)**
  A type of a mathematical programming problem where all the constraints are given as linear expressions, and some of the decision variables are required to take only integer values. For more details, see any standard reference, e. g. [4].

- **Graph**
  An abstract combinatorial construction comprising a finite set of *vertices*, and a finite set of *edges*, where each edge is a pair of vertices. We treat *undirected* graphs, i. e., graphs where edges are unordered pairs of vertices. For more information, see e. g. [5].

3.3 The Program's Input and Output

This section explains the format of the input and the output of the program. Section 3.3.1 illustrates an example of the program's input format, and Section 3.3.2 gives a concrete computational example. Following, Section 3.3.3 illustrates an example of the program's output format, and Section 3.3.4 gives a concrete computational example.

3.3.1 Program Input

This section gives an explanation of the input to the program.

First the input requires three textual files containing

- the descriptor names, in csv format

- the weights and biases of a trained ANN in textual format.

For a common prefix `TT` which the program accepts as a command-line parameter, these files must be saved with file names `TT_desc.csv`, `TT_weights.txt` and `TT_biases.txt` for the files containing the descriptor names, the weights, and the biases of a trained ANN, respectively. Next, comes the target value for which we wish to infer a chemical graph based on the trained ANN given above. Following is a chemical specification given in a textual file, as described in [1], as well as a filename prefix for the output files, which are described in Section 3.3.4

Finally, comes a choice of MILP solver program to be used. We can choose

- 1: CPLEX, a commercial MILP solver [10], free for academic use.

(Note, in this case the parameter `CPLEX_PATH` in the file `infer_acyclic_graphs.py` must be set to the correct path of the CPLEX program executable file.)

- 2: CBC, a free and open-source MILP solver. It comes together with the PuLP package for Python [6].

3.3.2 Input Data Format

This section presents an actual example of an input instance of the program. In particular, we give a concrete example of the three input files mentioned in Section 3.3.1.

The purpose of this program is to calculate a feature vector that will produce a desired output from a given trained ANN. Figure 3.1 gives an example of a trained ANN.

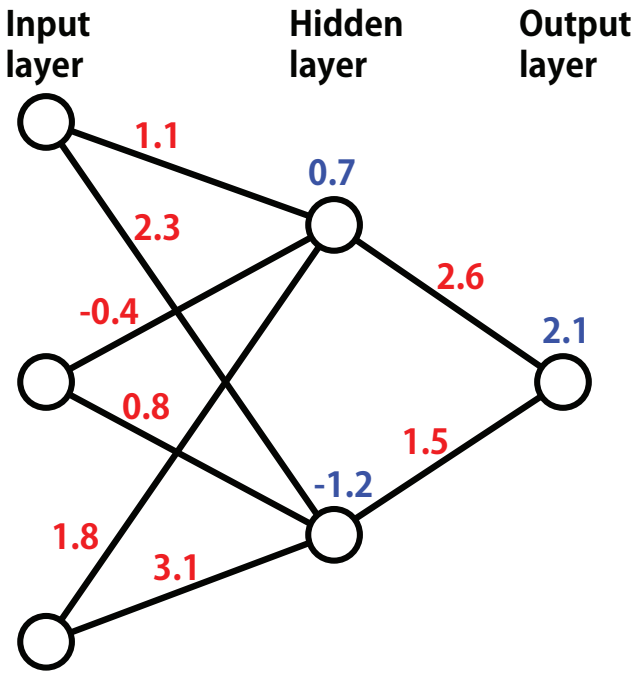


Figure 3.1: An example of a trained ANN. The ANN's weights are given in red numbers, and its biases in blue.

The information on the trained ANN is written in two text files, containing the information on the ANN's weights and biases, respectively. First, we give the structure of the file that contains the information on the ANN's weights. The first line of this text file contains the information on

the ANN's architecture, i. e., the number of nodes in each of its layers. From the second row and onward follows the information on the weights in the ANN. Each row contains the weights of the edges that are incident to one node of the ANN, first of the nodes in the input layer, and then for the nodes of the hidden layers. Following is a textual example for the ANN given in Fig. 3.1.

Organization of the text file containing weight data

```
3 2 1
1.1 2.3
-0.4 0.8
1.8 3.1
2.6
1.5
```

Next, comes the text file with the information on the ANN's biases. The bias values from Fig. 3.1 are given below.

Bias values

```
0.7
-1.2
2.1
```

Last comes a text file containing data on the feature vector. The first line of this file contains the names of the descriptors used in the feature vectors. Following form the second row onward, are the numerical values of the descriptors for each chemical graph in the training dataset, one row per chemical graph. For an example, please check one of the files `TT_desc.csv`, in the folder `ANN`, where $\texttt{TT} \in \{\texttt{BP}, \texttt{MP}, \texttt{KOW}\}$.

3.3.3 Program Output

This section gives an explanation of the output of the program. If there exists an acyclic chemical graph with a feature vector that would result with the given target value as a prediction of the given trained ANN, the program will output the feature vector. In case such a chemical graph does not exist, the program will report this. The next section gives an explanation of the output of the program.

3.3.4 Output Data Format

This section describes the output data of the program as obtained on a personal computer.

Once invoked, the program will print some messages to the standard error stream, which appear on the terminal. Once the MILP solver completes the computation, the status of the computation is printed on the terminal.

Text output on the terminal

```
Initializing Time: 0.529          # Written to stderr
Start Solving Using CPLEX...      # Written to stderr
Status: Feasible                  # Solution status
MILP y*: 223.856                  # Calculated target value in the MILP
ANN propagated y*: 223.856        # Target value calculated by the trained ANN
Solving Time: 52.534              # Time taken by the MILP solver
```

Finally, if there exists a feasible solution the program writes to disk two text files. Recall that as a part of the input the program requires a filename used for the output files. Assume that the supplied parameter is `filename`. Then, the resulting two text files are named

- `filename.sdf`
- `filename_partition.sdf`.

The file `filename.sdf` contains information on the inferred chemical graph in the SDF (Structure Data File) format. For more information see the official documentation (in English)

`http://help.accelrysonline.com/ulm/onelab/1.0/content/ulm_pdfs/direct/reference/ctfileformats2016.pdf`

for detail.

The file `filename_partition.sdf` contains a decomposition of the cyclic graph from the file `filename.sdf` into acyclic subgraphs, as described in [1].

3.4 Invoking the Program and a Computational Example

This section explains how to invoke the program and explains a concrete computational example. Following is an example of invoking the program `infer_cyclic_graphs_ec_id.py`.

3.4.1 Executing the Program

First make sure that the terminal is correctly directed to the location of the `source_code` folder, as described in Section 3.1.

```
python infer_cyclic_graphs_ec_id.py trained_ann_filename_prefix target_value
        chemical_specification output_file_name solver_type
```

As an example we use the target property Melting Point, that is, the files from the trained ANN with filename prefix `MP`, target value 220, the file `instance_1.txt` for a chemical specification, and CPLEX [10] as an MILP solver (parameter value 1).

```
python infer_cyclic_graphs_ec_id.py ANN/MP 220
     chemical_specification/instance_a.txt result 1
```

By executing the above command, the following text should appear on the terminal prompt.

Text output on the terminal

```
Initializing Time: 0.529
Start Solving Using CPLEX...
Status: Feasible
MILP y*: 223.856
ANN propagated y*: 223.856
Solving Time: 52.534
```

The contents of the output files `result.sdf` and `result_partition.txt` are as follows.

File `result.sdf`

```
 1
MILP_cyclic

 48 51  0  0  0  0  0  0  0  0999 V2000
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 N   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
```

```
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 N   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 N   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 N   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0  0  0  0
  1 28  1  0  0  0  0
  1 29  1  0  0  0  0
  2  3  1  0  0  0  0
  2  4  2  0  0  0  0
  5  6  1  0  0  0  0
  5 28  1  0  0  0  0
  5 43  1  0  0  0  0
```

```
 6 29  2  0  0  0  0
 7  8  1  0  0  0  0
 7 27  1  0  0  0  0
 7 38  1  0  0  0  0
 8  9  1  0  0  0  0
 9 10  1  0  0  0  0
 9 31  1  0  0  0  0
10 28  2  0  0  0  0
11 21  1  0  0  0  0
11 24  1  0  0  0  0
12 13  1  0  0  0  0
12 14  2  0  0  0  0
12 18  1  0  0  0  0
14 15  1  0  0  0  0
15 16  2  0  0  0  0
15 20  1  0  0  0  0
17 18  1  0  0  0  0
17 19  1  0  0  0  0
19 20  1  0  0  0  0
20 21  1  0  0  0  0
21 23  1  0  0  0  0
22 23  1  0  0  0  0
22 25  1  0  0  0  0
22 46  1  0  0  0  0
23 24  2  0  0  0  0
24 26  1  0  0  0  0
25 27  1  0  0  0  0
26 27  1  0  0  0  0
29 30  1  0  0  0  0
31 32  2  0  0  0  0
31 33  1  0  0  0  0
33 34  1  0  0  0  0
33 35  1  0  0  0  0
35 36  1  0  0  0  0
36 37  3  0  0  0  0
38 39  1  0  0  0  0
39 40  1  0  0  0  0
40 41  1  0  0  0  0
40 42  1  0  0  0  0
43 44  1  0  0  0  0
44 45  3  0  0  0  0
46 47  1  0  0  0  0
```

```
 47 48  3  0  0  0  0
M  END
$$$$
```

File `result_partition.sdf`

```
12
18
0 1
19
0 0
20
0 0
21
0 0
22
1 3
23
0 0
24
0 1
25
0 1
26
0 0
27
0 1
28
0 2
29
0 4
15
18 12 14 15 20
1 3
18 17 19
0 3
19 20
0 0
```

```
20 21
0 0
21 11 24
0 1
21 23
0 0
22 23
0 0
22 25
0 0
23 24
0 0
24 26
0 0
25 27
0 0
26 27
0 0
27 7 8 9 10 28
4 6
28 1 29
0 2
28 5 6 29
3 5
```

Chapter 4

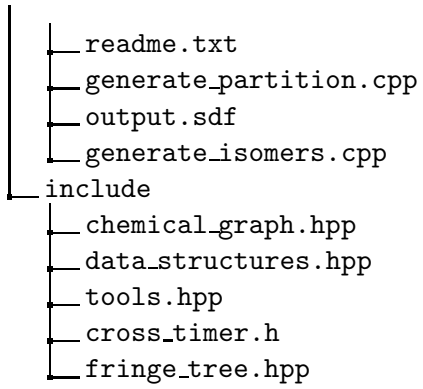
Module 4: Listing Chemical Isomers of a Given 2-Lean Cyclic Chemical Graph

4.1 Introduction

This text explains how to use the program for listing chemical isomers of a given 2-lean cyclic chemical graph [1].

The structure of files and folders of this module are as follows

```
./Module_4
├── Manual_Module_4_Cyclic_en.pdf
├── Manual_Module_4_Cyclic_en.tex
├── Manual_Module_4_Cyclic_jp.pdf
├── Manual_Module_4_Cyclic_jp.tex
├── Pseudocode_Graph_Generation.pdf
└── files
    ├── Makefile
    ├── executables
    │   ├── linux
    │   │   ├── generate_isomers
    │   │   └── generate_partition
    │   ├── osx
    │   │   ├── generate_isomers
    │   │   └── generate_partition
    │   └── windows
    │       ├── generate_partition.exe
    │       └── generate_isomers.exe
    ├── instances
    │   ├── sample_1_partition.txt
    │   ├── sample_1.png
    │   └── sample_1.sdf
    └── main
```



The contents of each of the files is as follows.

- `Manual_Module_4_Cyclic_en.pdf`
  This file.
- `Manual_Module_4_Cyclic_en.tex`
  A LaTeX source file of this manual.
- `Manual_Module_4_Cyclic_jp.pdf`
  The Japanese version of this manual.
- `Manual_Module_4_Cyclic_jp.tex`
  A LaTeX source file of the Japanese version of the manual.
- `Pseudocode_Graph_Generation.pdf`
  A pdf file with Pseudo-codes for the graph search algorithm.
- Folder `files`
  - `Makefile`
    Compilation directives for the C++ programs of this module.
  - Folder `executables`
    A folder that contains compiled executable files for each of the architectures: linux, osx, and windows.
  - Folder `instances`
    A folder containing sample input instances.
    * `sample_1.sdf`
      A cyclic chemical graph with 20 vertices (non-Hydrogen atoms), core size 18 and core height 1.
    * `sample_1_partition.txt`
      A file containing partition information into acyclic subgraphs of the cyclic chemical graph given in `sample_1.sdf`.
    * `sample_2.sdf`
      A cyclic chemical graph with 50 vertices, core size 24 and core height 6.
    * `sample_2_partition.txt`
      A file containing partition information into acyclic subgraphs of the cyclic chemical graph given in `sample_2.sdf`.

    * `sample_3.sdf`
      A cyclic chemical graph with 60 vertices, core size 31 and core height 4.
    * `sample_3_partition.txt`
      A file containing partition information into acyclic subgraphs of the cyclic chemical graph given in `sample_3.sdf`.
    * `sample_4.sdf`
      A cyclic chemical graph with 120 vertices, core size 60 and core height 4.
    * `sample_4_partition.txt`
      A file containing partition information into acyclic subgraphs of the cyclic chemical graph given in `sample_4.sdf`.
- Folder `main`
  Folder containing source files written in the C++ programming language.
    * `generate_partition.cpp`
      Functions for calculating a partition of a cyclic chemical graph into acyclic subgraphs.
    * `generate_isomers.cpp`
      Implements an algorithm for listing chemical isomers of a 2-lean cyclic chemical graph.
    * Folder `include`
      A folder that contains related header files, written in C++.
        · chemical_graph.hpp
          A header file that contains functions for manipulating chemical graphs.
        · cross_timer.h
          A header file that contains functions for measuring execution time.
        · data_structures.hpp
          Data structures implemented for storing chemical graphs.
        · debug.h
          Used for debugging purposes.
        · fringe_tree.hpp
          Header file with functions for enumerating 2-fringe trees [1].
        · tools.hpp
          Various functions used in the implementation.

The remainder of this text is organized as follows. Section 4.2 explains some of the terminology used throughout the text. Section 4.3 gives an explanation of the input and the output of the program for calculating a partition of a cyclic chemical graph into acyclic subgraphs, and provides a computational example. Section 4.4 gives an explanation of the program for generating chemical isomers of a given 2-lean cyclic chemical graph, explaining the input, output, and presents a computational example.

4.2 Terminology

This section gives an overview of the terminology used in the text.

- Chemical Graph
  A graph-theoretical description of a chemical compound, where the graph's vertices correspond to atoms, and its (multi) edges to chemical bonds. Each vertex is colored with the chemical element of the atom it corresponds to, and edges have multiplicity according to the corresponding bond order. We deal with "hydrogen-suppressed" graphs, where none of the graph's vertices is colored as hydrogen. This can be done without loss of generality, since there is a unique way to saturate a hydrogen-suppressed chemical graph with hydrogen atoms subject to a fixed valence of each chemical element.

- Feature vector
  A numerical vector giving information such as the count of each chemical element in a chemical graph. For a complete information on the descriptors used in feature vectors for this project, please see [1].

- Partition Information
  Information necessary to specify the base vertices and edges, as well as the vertex and edge components of a chemical graph. For more details, please check [1].

4.3 Program for Calculating a Partition Into Acyclic Subgraphs

4.3.1 Input and Output

This section explains the input and output information of the program that is used to calculate a partition of a cyclic chemical graph into acyclic subgraphs. We call this program `Partition`. Following, Sec. 4.3.1 explains the input format, and Sec. 4.3.1 gives an explanation the output information of the program.

The Program's Input

The program `Partition` takes two items as its input.

First, is a cyclic chemical graph, given as a "structured data file," in SDF format. This is a standard format for representing chemical graphs. For more details, please check the documentation at

`http://help.accelrysonline.com/ulm/onelab/1.0/content/ulm_pdfs/direct/reference/`
`/ctfileformats2016.pdf`

Next, is a filename of a text file where the information on a partition of the graph given in the SDF into acyclic subgraphs calculated by the program will be saved.

The Program's Output

The output of this program is a partition into acyclic subgraphs of the cyclic chemical graph given in the input. The partition information is stored as a text file with filename as provided in the input parameters.

Output Format Example

```
4
7 # C
0 0 0
15 # C
0 0 0
10 # C
0 0 0
16 # C
0 0 0
5
7 4 3 5 6 9 2 15 # C1C1N1C1C1C1O1C
0 1 0
7 10 # C2C
0 0 1
10 12 14 13 11 7 # C1C2C1C2C1C
0 0 1
15 16 # C2C
0 0 1
16 18 20 19 17 15 # C1C2C1C2C1C
0 0 1
```

Following, Table 4.1 gives a row-by-row explanation of the numerical information in the above output file.

Table 4.1: Structure of an Output File with Partition Information

Value in the file	Explanation
`4`	Number of base vertices

`7 # C`
`0 0 0`
`15 # C`
`0 0 0`
`10 # C`
`0 0 0`
`16 # C`
`0 0 0`
	The index of a base vertex in the input SDF and its element
	Core height lower and upper bound;
	the vertex component can be created or not (0/1)

`5`	Number of base edges

`7 4 3 5 6 9 2 15 # C1C1N1C1C1C1O1C`
`0 1 0`
`7 10 # C2C`
`0 0 1`
`10 12 14 13 11 7 # C1C2C1C2C1C`
`0 0 1`
`15 16 # C2C`
`0 0 1`
`16 18 20 19 17 15 # C1C2C1C2C1C`
`0 0 1`
	Indices of vertices in the base edge from the input SDF,
	their elements and bond multiplicities
	
	Core height lower and upper bound,
	the edge component can be created or not (0/1)

4.3.2 Program Execution and Computation Example

This section gives an explanation on how to compile and run the program, as well as a concrete computational example of the program's execution.

Compiling and Executing the Program for Generating a Decomposition of a Cyclic Graph into Acyclic Graphs

A compiled executable that has been tested on
- linux
- osx
- windows (cygwin)

is included in the set of files. In addition, we describe how the accompanying source files can be compiled.

- *Computation environment*
  There should not be any problems when using an ISO C++11 compatible compiler.

- *Compiling the program*
  In the terminal, navigate to the `files` subfolder. Then, if the `make` command is available on

the system, the program can be simply compiled by typing
`$ make generate_partition`
In case the `make` command is not available, then the program can be compiled as
`$ g++ -o generate_partition ./main/generate_partition.cpp -O3 -std=c++11`

- *Executing the program*
  `$ ./generate_partition instance.sdf instance_partition.txt`
  `instance.sdf` is an input SDF, and a partition information as the output of the program is stored in the file `instance_partition.txt` .

Computational Example

This section illustrates a concrete computational example of running the `Partition` program. We assume the following:

- Input file: `sample_1.sdf` from the folder `instances` (see Sec. 4.1)
- Output file to store the partition information: `partition.txt`

Run the following command in the terminal to execute the program.

```
./generate_partition ./instances/sample_1.sdf partition.txt
```

After successfully executing the program, the contents of the file `partition.txt` should be as follows.

Contents of the file `partition.txt`

```
4
7 # C
0 0 0
15 # C
0 0 0
10 # C
0 0 0
16 # C
0 0 0
5
7 4 3 5 6 9 2 15 # C1C1N1C1C1C1O1C
0 1 0
7 10 # C2C
0 0 0
10 12 14 13 11 7 # C1C2C1C2C1C
0 0 0
```

```
15 16 # C2C
0 0 0
16 18 20 19 17 15 # C1C2C1C2C1C
0 0 0
```

4.4 Program for Generating Chemical Isomers

4.4.1 Input and Output of the Program

This section gives an explanation of the input and output of the program that generates chemical isomers of a given 2-lean chemical graph. We call the program `generate_isomers`. Section 4.4.1 gives an explanation of the program's input, and Sec. 4.4.1 of the program's output.

The Program's Input

The input to the `generate_isomers` program consists of six necessary and one optional item.
First, comes information of a 2-lean chemical graph (in SDF format).
Second, comes a time limit in seconds on each of the stages of the program (for details, check the accompanying file with pseudo-codes of the algorithm).
Third is an upper bound on the number of *partial* feature vectors that the program stores during its computation.
Fourth is the number of "sample graphs" that are stored per one feature vector.
Fifth is an upper limit on the number of generated output chemical graphs.
Sixth is a filename (SDF) where the output graphs will be written.
An optional parameter is a filename with a partition information of the chemical graph given as the first parameter, and if given, it must be last in the list of parameters.

The Program's Output

After executing the `generate_isomers` program, the chemical isomers of the input graph will be written in the specified SDF, and some information on the execution will be output on the terminal. The information printed on the terminal includes:
- a lower bound on the number of chemical isomers of the given input chemical graph,
- the number of graphs that the program generated under the given parameters, and
- the program's execution time.

4.4.2 Executing the Program and a Computational Example

This section gives a concrete computational example of the `generate_isomers` program.

Compiling and Executing the `generate_isomers` Program

A compiled executable that has been tested on
- linux

- osx
- windows (cygwin)
is included in the set of files. In addition, we describe how the accompanying source files can be compiled.

- *Computation environment*
  There should not be any problems when using a ISO C++ compatible compiler.

- *Compiling the program*
  If the `make` command is available on the system, then the program can be compiled in the `files` folder by typing
  `$ make generate_isomers`
  in the terminal. If the `make` command is not available, then please run the following command in the terminal.
  `$ g++ -o generate_isomers ./main/generate_isomers.cpp -O3 -std=c++11`

- *Executing the program*
  The program can be executed by running the following command in the terminal.
  `$ ./generate_isomers instance.txt a b c d output.sdf instance_partition.txt`
  Above, `generate_isomers` is the name of the program's executable file, and the remaining command-line parameters are as follows:
  `instance.txt` a text file containing a chemical specification
  `a` upper bound (in seconds) on the computation time,
  `b` upper bound on the number of stored partial feature vectors,
  `c` upper bound on the number of sample graphs stored per feature vector,
  `d` upper bound on the number of output graphs,
  `output.sdf` filename to store the output chemical graphs (SDF format),
  `instance_partition.txt` partition information of the input chemical graph.

Computational Example

We execute the `generate_isomers` program with the following parameters.

- Input graph: File `sample_1.sdf` from the folder `instances` (see Sec. 4.1)
- Time limit: 10 seconds
- Upper limit on the number of partial feature vectors: 10000000
- Number of sample graphs per feature vector: 5
- Upper limit on the number of output graphs: 2
- Filename to store the output graphs: `output.sdf`
- Partition information of the input graph: File `sample_1_partition.txt` from the folder `instances`.

Execute the program by typing the following command into the terminal (without a line break).

```
./generate_isomers ./instances/sample_1.sdf 10 10000000 5 2
                   output.sdf ./instances/sample_1_partition.txt
```

Upon successful execution of the program, the following text should appear on the terminal.

Output Written on the Terminal

```
A lower bound on the number of graphs = 72
Number of generated graphs = 2
Total time :  0.00649s.
```

Contents of the file output.sdf

```
1
BH-cyclic
BH-cyclic
20 21 0 0 0 0 0 0 0 0999 V2000
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 N 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 O 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 O 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
1 3 2 0 0 0 0
1 5 1 0 0 0 0
1 16 1 0 0 0 0
```

2 4 2 0 0 0 0
2 11 1 0 0 0 0
2 20 1 0 0 0 0
3 13 1 0 0 0 0
4 17 1 0 0 0 0
5 6 1 0 0 0 0
6 7 1 0 0 0 0
7 8 1 0 0 0 0
8 9 1 0 0 0 0
8 10 1 0 0 0 0
10 11 1 0 0 0 0
11 12 1 0 0 0 0
13 14 2 0 0 0 0
14 15 1 0 0 0 0
15 16 2 0 0 0 0
17 18 2 0 0 0 0
18 19 1 0 0 0 0
19 20 2 0 0 0 0
M END
$$$$
2
BH-cyclic
BH-cyclic
20 21 0 0 0 0 0 0 0 0999 V2000
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 N 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 O 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 O 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0

```
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
0.0000 0.0000 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0
1 3 2 0 0 0 0
1 5 1 0 0 0 0
1 16 1 0 0 0 0
2 4 2 0 0 0 0
2 11 1 0 0 0 0
2 20 1 0 0 0 0
3 13 1 0 0 0 0
4 17 1 0 0 0 0
5 6 1 0 0 0 0
6 7 1 0 0 0 0
7 8 1 0 0 0 0
8 9 1 0 0 0 0
8 10 1 0 0 0 0
10 11 1 0 0 0 0
11 12 1 0 0 0 0
13 14 2 0 0 0 0
14 15 1 0 0 0 0
15 16 2 0 0 0 0
17 18 2 0 0 0 0
18 19 1 0 0 0 0
19 20 2 0 0 0 0
M END
$$$$
```

Bibliography

[1] T. Akutsu and H. Nagamochi. A Novel Method for Inference of Chemical Compounds with Prescribed Topological Substructures Based on Integer Programming. Arxiv preprint, arXiv:2010.09203

[2] T. Akutsu and H. Nagamochi. A Mixed Integer Linear Programming Formulation to Artificial Neural Networks, in Proceedings of the 2019 2nd International Conference on Information Science and Systems, pp. 215–220, https://doi.org/10.1145/3322645.3322683.

[3] HSDB in PubChem `https://pubchem.ncbi.nlm.nih.gov` (accessed on February 1st, 2021)

[4] J. Matousek and B. Gärtner. Understanding and Using Linear Programming. Springer, 2007.

[5] M. S. Rahman. Basic Graph Theory. Springer, 2017.

[6] A Python Linear Programming API, `https://github.com/coin-or/pulp`.

[7] Optimization with PuLP, `http://coin-or.github.io/pulp/`.

[8] The Python Papers Monograph, `https://ojs.pythonpapers.org/index.php/tppm/article/view/111`.

[9] Optimization with PuLP, `https://pythonhosted.org/PuLP/`.

[10] IBM Cplex Optimizer `https://www.ibm.com/analytics/cplex-optimizer`.