PyeMap Documentation

pyemap

Aug 11, 2022

CONTENTS:

PyeMap is a python package aimed at automatic identification of electron and hole transfer pathways in proteins. The analysis is based on a coarse-grained version of Beratan and Onuchic's Pathway model, and only accounts for through-space hopping between aromatic residues side chains [Beratan1992]. Side chains of aromatic residues and non-protein electron transfer active moieties are modeled as vertices in a weighted graph, where the edge weights are modified distance dependent penalty functions.

For single proteins, PyeMap identifies the shortest pathways between a specified donor to the surface, or to a specified acceptor. For groups of proteins, PyeMap identifies shared pathways/motifs using graph mining techniques.

PyeMap serves as the backend for the web application eMap, and can also be used as a fully functional Python package.

CHAPTER ONE

CURRENT FEATURES

Single protein

- Identification of most probable electron/hole transfer pathways from a specified donor to the protein surface or a specified electron/hole acceptor
- Accepts valid .pdb or .cif structures provided by the user or fetched from RCSB database
- Automatic detection of non-protein aromatic moieties such as porphyrins, nucleobases, and other aromatic cofactors
- Automatic detection of 60+ inorganic clusters such as iron-sulfur clusters and others
- Automatic detection of redox-active metal ions
- User specified custom fragments
- Visualization of chemical structures and graphs
- Automatic identification of surface exposed residues using residue depth or solvent accessibility criteria
- Control over various parameters which determine connectivity of graph theory model
- Tested on structures as large as 5350 residues (51599 atoms)

Graph Mining

- Mining families of protein graphs for all patterns up to a given support threshold
- Mining families of protein graphs for specific patterns
- Classification of protein subgraphs based on similarity

CHAPTER TWO

IN DEVELOPMENT

- Improving the physical model of electron transfer by incorporating information on geometry-dependent electronic couplings and site sensitive energetics
- Generalization to DNA, protein-DNA complexes etc.

2.1 Installation

PyeMap officially supports Python versions 3.7 and later, and has been tested for Linux and OSX platforms.

Pip

Pip installation will only install python dependencies. This is sufficient to run PyeMap analysis, but some features will be missing:

```
$ pip install pyemap
```

For full functionality, install the following packages:

- RDKit: visualization of chemical stuctures
- MSMS: residue depth criterion for surface exposed residues (not available on MacOS Catalina)
- DSSP: solvent accessibility criterion for surface exposed residues
- MUSCLE: Multiple sequence alignment
- Graphviz: visualization of graphs
- PyGraphviz : visualization of graphs

All of these packages can be downloaded free of charge from their respective owners, and build recipes are available on the Anaconda cloud for some platforms.

2.2 Tutorial: Single Protein

This tutorial can help you start working with PyeMap.

2.2.1 Basic Usage

Parsing

The first step of PyeMap analysis is parsing of a .cif or pdb file.

You can either provide PyeMap with the path to the file:

```
>>> import pyemap
>>> my_emap = pyemap.parse("path/to/protein.pdb")
```

Or fetch from the RCSB database by 4 character PDB ID:

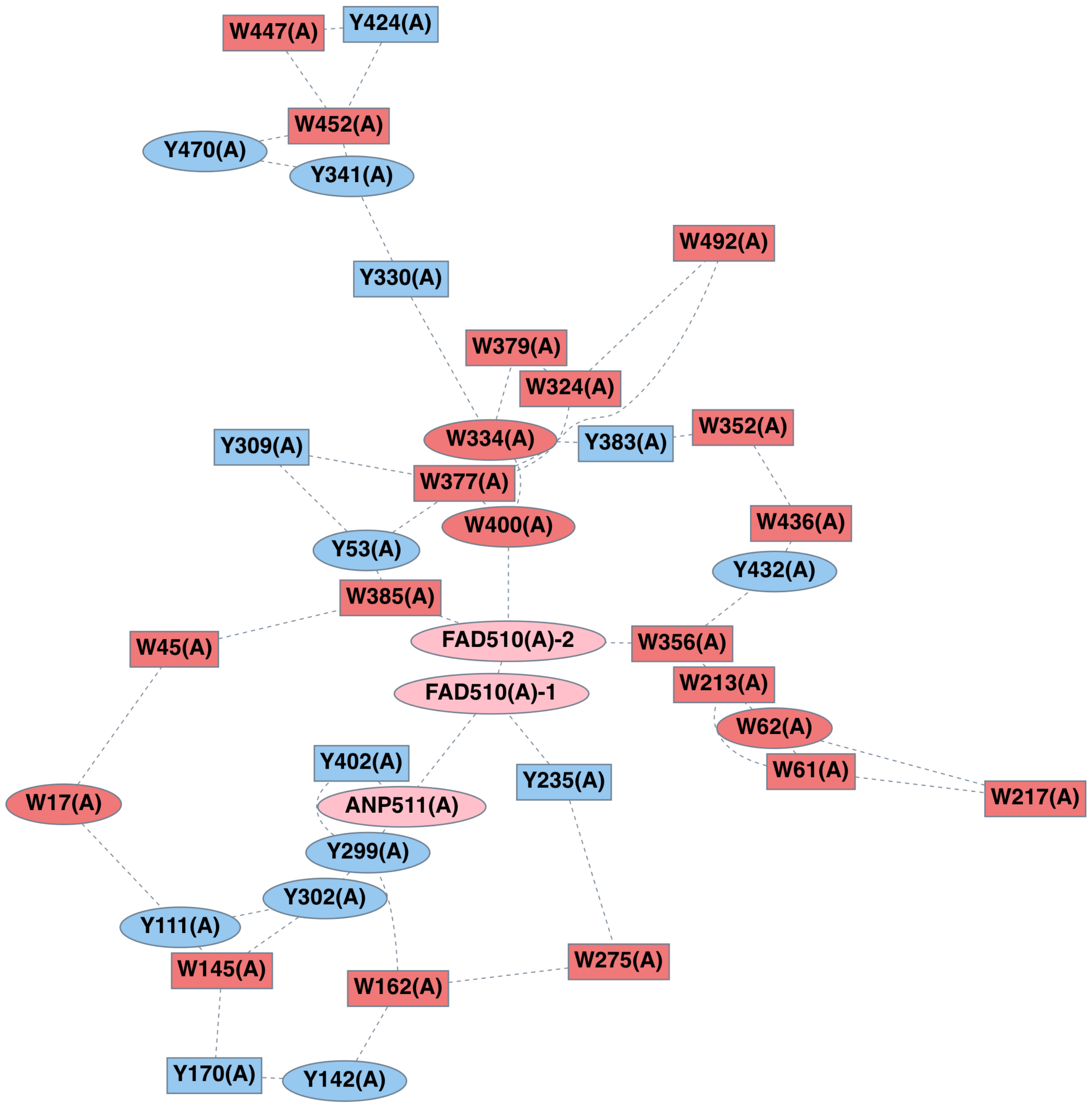
```
>>> my_emap = pyemap.fetch_and_parse("1u3d")
Fetching file 1u3d from RSCB Database...
Success!
Identified 3 non-protein ET active moieties.
```

This generates an *emap* object; a dynamic data structure which manages the data at all phases of pyemap analysis. At this stage, the *emap* object contains the parsed protein structure, and a list of automatically identified non-protein electron transfer active moieties.

Process

Next, construct the graph model of the protein crystal structure using `process()`

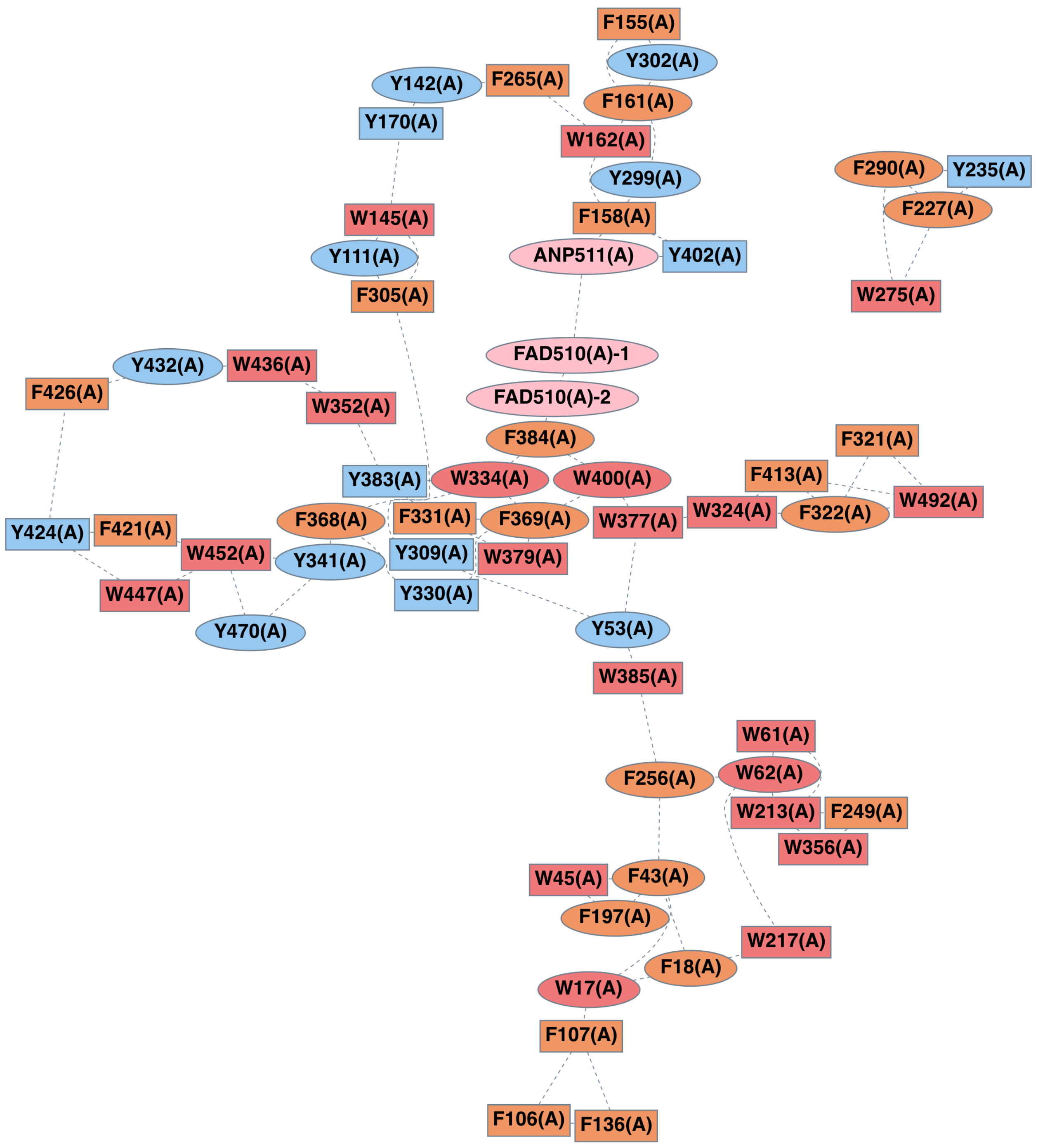
```
>>> pyemap.process(my_emap)
>>> my_emap.init_graph_to_Image().show()
```



Surface exposed residues appear as rectangular nodes, while buried residues appear as oval nodes. Edge weights are proportional to distances between residues. By default, all Trp, Tyr, and automatically detected non-protein electron transfer active moieties are included.

There are various parameters one can specify to control which residues are included in the graph, and the overall connectivity of the graph. For example, to include phenylalanine residues, and to modify the pure distance filter for edges, do:

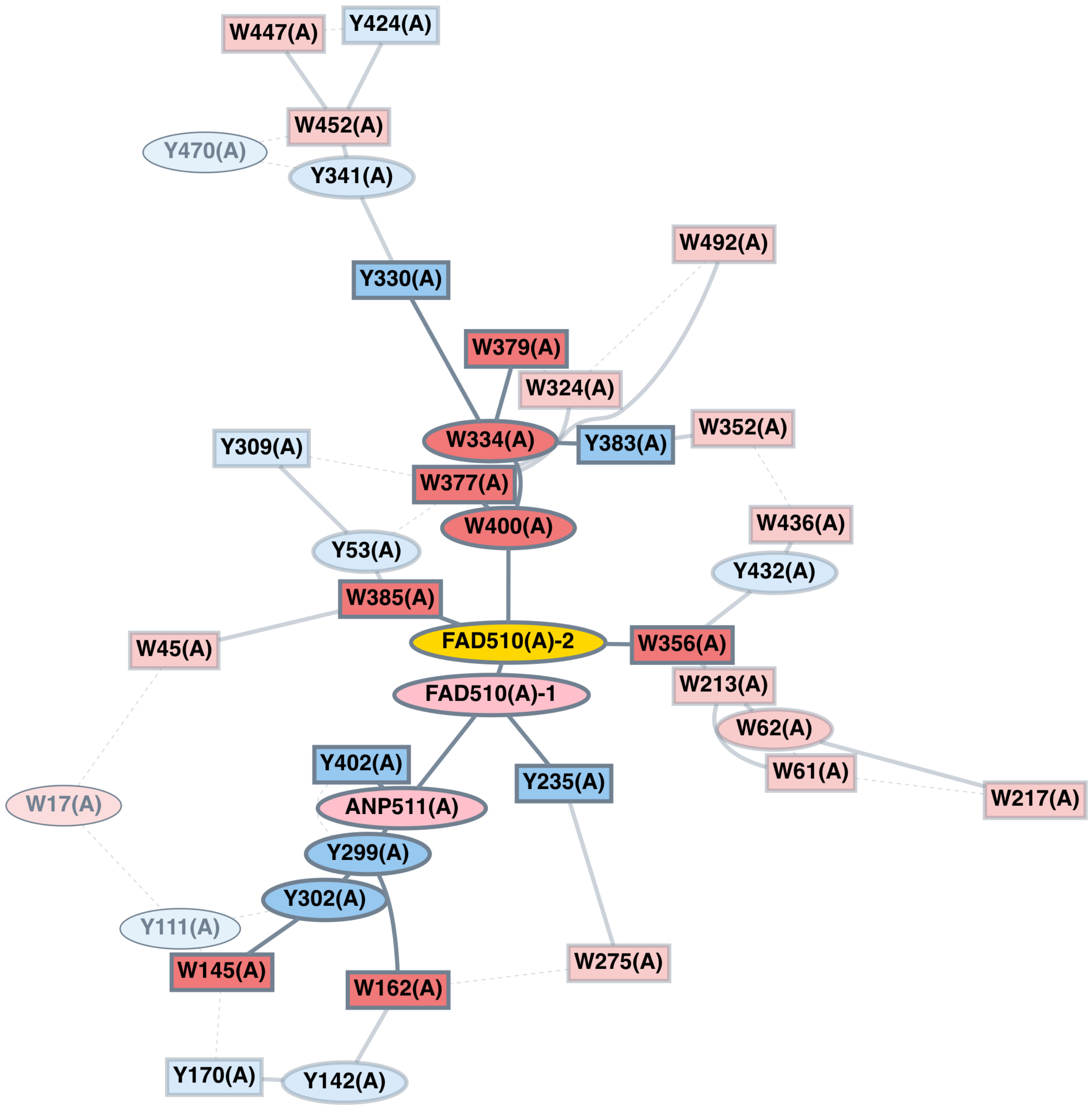
```
>>> pyemap.process(my_emap,include_residues=['Y','W','F'],distance_cutoff=15)
>>> my_emap.init_graph_to_Image().show()
```


F155(A)
Y302(A)
Y142(A)
F265(A)
F161(A)
Y170(A)
W162(A)
F290(A)
Y235(A)
Y299(A)
F227(A)
W145(A)
F158(A)
Y111(A)
ANP511(A)
Y402(A)
W275(A)
F305(A)
FAD510(A)-1
Y432(A)
W436(A)
F426(A)
FAD510(A)-2
W352(A)
F384(A)
F321(A)
Y383(A)
W334(A)
W400(A)
F413(A)
W492(A)
W324(A)
F322(A)
F368(A)
F331(A)
F369(A)
W377(A)
Y424(A)
F421(A)
W452(A)
Y341(A)
Y309(A)
W379(A)
W447(A)
Y330(A)
Y470(A)
Y53(A)
W385(A)
W61(A)
W62(A)
F256(A)
W213(A)
F249(A)
W356(A)
W45(A)
F43(A)
F197(A)
W217(A)
F18(A)
W17(A)
F107(A)
F106(A)
F136(A)

Find Pathways

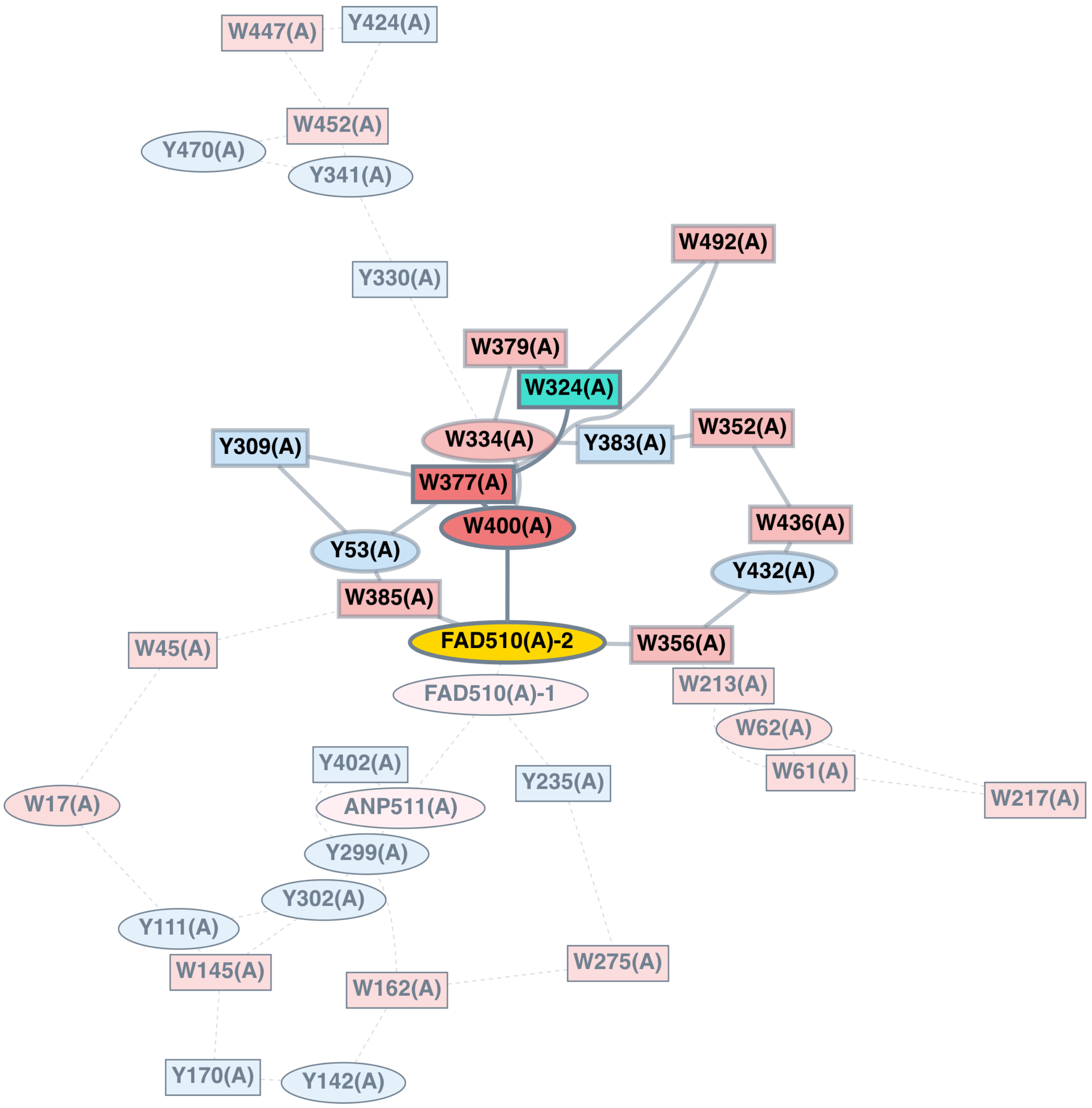
Finally, search for possible pathways from a specified electron/hole donor to the surface using *find_paths()*:

```
>>> pyemap.find_paths(my_emap,"FAD510(A)-2")
>>> my_emap.paths_graph_to_Image().show()
```



Alternatively, you can search for pathways from a specified donor to a particular acceptor:

```
>>> pyemap.find_paths(my_emap,"FAD510(A)-2", target = "W324(A)", max_paths=10)
>>> my_emap.paths_graph_to_Image().show()
```



To get a report of the pathways found by pyemap, use *report()*

```
>>> print(my_emap.report())
Branch: W324(A)
1a: ['FAD510(A)-2', 'W400(A)', 'W377(A)', 'W324(A)'] 24.15
1b: ['FAD510(A)-2', 'W385(A)', 'Y53(A)', 'W377(A)', 'W324(A)'] 35.25
1c: ['FAD510(A)-2', 'W400(A)', 'W334(A)', 'W379(A)', 'W324(A)'] 36.37
1d: ['FAD510(A)-2', 'W400(A)', 'W377(A)', 'W492(A)', 'W324(A)'] 49.20
1e: ['FAD510(A)-2', 'W385(A)', 'Y53(A)', 'Y309(A)', 'W377(A)', 'W324(A)'] 50.67
1f: ['FAD510(A)-2', 'W385(A)', 'Y53(A)', 'W377(A)', 'W492(A)', 'W324(A)'] 60.30
1g: ['FAD510(A)-2', 'W385(A)', 'Y53(A)', 'W377(A)', 'W400(A)', 'W334(A)', 'W379(A)',
↪'W324(A)'] 61.93
1h: ['FAD510(A)-2', 'W356(A)', 'Y432(A)', 'W436(A)', 'W352(A)', 'Y383(A)', 'W334(A)',
```

(continues on next page)

(continued from previous page)

```
↪'W379(A)', 'W324(A)'] 72.20
1i: ['FAD510(A)-2', 'W385(A)', 'Y53(A)', 'Y309(A)', 'W377(A)', 'W492(A)', 'W324(A)'] 75.
↪72
1j: ['FAD510(A)-2', 'W385(A)', 'Y53(A)', 'Y309(A)', 'W377(A)', 'W400(A)', 'W334(A)',
↪'W379(A)', 'W324(A)'] 77.35
```

2.2.2 Interacting with the emap object

The *emap* object manages all of the data at every stage of the analysis, and this data is accessible through its attributes and functions.

Dictionaries

Much of the data on residues is stored in various dictionaries, where the key is residue name as it appears in the graph image. For example, to directly access the Biopython Residue object corresponding to the node W324(A) do:

```
>>> residue_obj = my_emap.residues["W324(A)"]
>>> print(type(residue_obj))
<class 'Bio.PDB.Residue.Residue'>
```

The same is true of pathways, which are stored as *ShortestPath* objects. Any pathway(and by extension its attributes) can be accessed by its pathway ID. For example, if you want the selection string for visualization of pathway 1a in the NGL viewer, do:

```
>>> my_path = my_emap.paths["1a"]
>>> print(my_path.selection_strs)
'(510 and :A and .N1) or (510 and :A and .C2) or (510 and :A and .O2) or (510 and :A and␣
↪.N3) or
(510 and :A and .C4) or (510 and :A and .O4) or (510 and :A and .C4X) or (510 and :A and␣
↪.N5) or
(510 and :A and .C5X) or (510 and :A and .C6) or (510 and :A and .C7) or (510 and :A and␣
↪.C8) or
(510 and :A and .C9) or (510 and :A and .C9A) or (510 and :A and .N10) or (510 and :A␣
↪and .C10)',
'(400 and :A and .CG) or (400 and :A and .CD1) or (400 and :A and .CD2) or (400 and :A␣
↪and .NE1) or
(400 and :A and .CE2) or (400 and :A and .CE3) or (400 and :A and .CZ2) or (400 and :A␣
↪and .CZ3) or
(400 and :A and .CH2)', '(377 and :A and .CG) or (377 and :A and .CD1) or (377 and :A␣
↪and .CD2) or
(377 and :A and .NE1) or (377 and :A and .CE2) or (377 and :A and .CE3) or (377 and :A␣
↪and .CZ2) or
(377 and :A and .CZ3) or (377 and :A and .CH2)', '(324 and :A and .CG) or (324 and :A␣
↪and .CD1) or
(324 and :A and .CD2) or (324 and :A and .NE1) or (324 and :A and .CE2) or (324 and :A␣
↪and .CE3) or
(324 and :A and .CZ2) or (324 and :A and .CZ3) or (324 and :A and .CH2)'
```

Graphs

The graphs are stored in the *emap* object as NetworkX Graph objects. The attributes of edges and vertices can be accessed from these graphs in usual NetworkX fashion (see their documentation for more information). For example, to access the weight of the edge connecting vertices FAD510(A)-2 and W400(A), do:

```
>>> weight = my_emap.init_graph["FAD510(A)-2"]["W400(A)"]['weight']
>>> print(weight)
8.793106029091886
```

If what you need instead is the actual distance, this information is also kept:

```
>>> dist = my_emap.init_graph["FAD510(A)-2"]["W400(A)"]['distance']
>>> print(dist)
8.802989071175238
```

2.2.3 Visualization

Residues

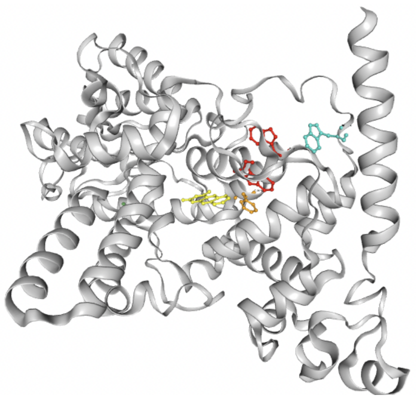
Graph images and chemical structures of non-protein electron transfer active moieties can be exported to PIL with the *residue_to_Image()*, *init_graph_to_Image()*, *paths_graph_to_Image()* functions. To save to file, use *paths_graph_to_file()*, *init_graph_to_file()*, and *residue_to_file()*.

```
>>> my_emap.residue_to_Image("FAD510(A)-2").show()
```

NGLView

Pathways can be visualized in the crystal structure using the NGLView Jupyter Widget. Pass the pathway ID of interest along with a nglview.widget.NGLWidget object to the *visualize_pathway_in_nglview()* function.

```
>>> import nglview as nv
>>> view = nv.show_file(my_emap.file_path)
>>> view.clear_representations()
>>> view.add_cartoon(color="lightgray")
>>> my_emap.visualize_pathway_in_nglview("1a",view)
>>> view
```



2.3 Tutorial: Protein Graph Mining

This tutorial can help you start working with the `pyemap.graph_mining` module of PyeMap.

2.3.1 Parsing

The first step of graph mining with PyeMap is to create a *PDBGroup* object, and populate it with *emap* objects for the PDBs of interest. Here, we'll fetch and parse a small set of flavoprotein PDBs.

```
import pyemap
from pyemap.graph_mining import PDBGroup
pdb_ids = ['1X0P', '1DNP', '1EFP', '1G28', '1IQR', '1IQU',
'1NP7', '1O96', '1O97', '1QNF', '1U3C', '1U3D', '2IYG', '2J4D',
'2WB2', '2Z6C', '3FY4', '3ZXS', '4EER', '4GU5', '4I6G', '4U63',
'6FN2', '6KII', '6LZ3', '6PU0', '6RKF']
pg = PDBGroup('My Group')
for pdb in pdb_ids:
   pg.add_emap(pyemap.fetch_and_parse(pdb))
```

2.3.2 Generating Protein Graphs

The next step is to generate the graphs for each PDB. One can specify the chains, ET active moieties, residues, and any other kwargs from *process()*. If no arguments are specified, the first chain from each PDB will be chosen, and all ET active moieties on those chains will be included. See *process_emaps()* for more details.

```
pg.process_emaps()
```

2.3.3 Generate graph database

The next step is to classify the nodes and edges. One can define "substitutions" for nodes, and thresholds for edges. See *generate_graph_database()* for more details.

```
# W and Y would be interchangeable and given the label 'X'
substitutions = ['W','Y']
# edges with weights > 12 are distinguished from those with weights < 12
edge_thresholds = [12]
pg.generate_graph_database(sub=[],edge_thresh=edge_thresholds)
```

2.3.4 Mine for subgraphs

There are two types of searches available in PyeMap.

1. mine for all possible subgraphs:

```
pg.run_gspan(19)
```

2. search for a specific pattern:

```
pg.find_subgraph('WWW#')
```

2.3.5 Analysis: Subgraph patterns

The identified subgraphs are stored in the *subgraph_patterns* dictionary.

```
>>> pg.subgraph_patterns
{'1_WWW#_18': <pyemap.graph_mining.frequent_subgraph.SubgraphPattern at 0x10d652430>,
'2_WWW#_14': <pyemap.graph_mining.frequent_subgraph.SubgraphPattern at 0x183c3a1f0>,
'3_WWW#_4': <pyemap.graph_mining.frequent_subgraph.SubgraphPattern at 0x183c3aca0>,
'4_WWW#_2': <pyemap.graph_mining.frequent_subgraph.SubgraphPattern at 0x183c2de50>,
'5_WWW#_2': <pyemap.graph_mining.frequent_subgraph.SubgraphPattern at 0x183c24a30>,
'6_WWW#_2': <pyemap.graph_mining.frequent_subgraph.SubgraphPattern at 0x183c24730>,
'7_WWW#_1': <pyemap.graph_mining.frequent_subgraph.SubgraphPattern at 0x12c680f10>}
```

The subgraph pattern can be visualized using *pyemap.graph_mining.SubgraphPattern.subgraph_to_Image()* or *pyemap.graph_mining.SubgraphPattern.subgraph_to_file()*.

```
sg = pg.subgraph_patterns['1_WWW#_18']
sg.subgraph_to_Image()
```

2.3.6 Analysis: Protein subgraphs

To identify the specific residues in each PDB involved in the identified patterns, one should first call *pyemap.graph_mining.SubgraphPattern.find_protein_subgraphs()*. The identified **protein subgraphs** are stored in the *protein_subgraphs* dictionary. Each protein subgraph has a unique ID.

```
sg.find_protein_subgraphs()
sg.protein_subgraphs
{'4U63_1': <networkx.classes.graph.Graph at 0x183d58fa0>,
'4U63_2': <networkx.classes.graph.Graph at 0x183b66be0>,
'4U63_3': <networkx.classes.graph.Graph at 0x183cf7040>,
'4U63_4': <networkx.classes.graph.Graph at 0x183d53eb0>,
'4U63_5': <networkx.classes.graph.Graph at 0x183d51220> ...
```

To visualize a protein subgraph, use *pyemap.graph_mining.SubgraphPattern.subgraph_to_Image()* or *pyemap.graph_mining.SubgraphPattern.subgraph_to_file()* and pass the ID as a keyword argument.

```
sg.subgraph_to_Image('1U3D_51')
```

2.3.7 Clustering

Protein subgraphs are clustered into groups based on sequence or structural similarity. By default, structural clustering is used. To switch to sequence clustering, call *pyemap.graph_mining.SubgraphPattern.set_clustering()*.

```
print(sg.groups)
{1: ['4U63_33', '1DNP_34', '2J4D_40', '1IQR_43', '1IQU_47'], ...
44: ['3ZXS_60'] ... }
```

2.3.8 Visualize in NGLView

Protein subgraphs can be visualized in the crystal structure using the NGLView Jupyter Widget. Pass the pathway ID of interest along with a `nglview.widget.NGLWidget` object to the *`visualize_subgraph_in_nglview()`* function.

```
import nglview as nv
view = nv.show_file(sg.support['1U3D'].file_path)
view.clear_representations()
view.add_cartoon(color="lightgray")
sg.visualize_subgraph_in_nglview('1U3D_50',view)
view
```

2.4 Developer Reference

2.4.1 Single Protein

Usage

Parse File

Introduction

PDB files can be either uploaded, or fetched from the RCSB database. At this stage, non-protein electron transfer active moieties are automatically identified by pyemap, and saved to an *emap* object which is returned to the user.

Parser

`pyemap.parse(filename, quiet=True)`

Parses pdb file and returns emap object.

Parameters

- **filename** (*str*) – Full path to file which needs to be parsed
- **quiet** (*bool, optional*) – Supresses output when set to true

Returns
my_emap – emap object reading for parsing

Return type
emap

`pyemap.fetch_and_parse(pdb_id, dest='', quiet=False)`

Fetches pdb from database and parses the file.

Parameters

- **pdb_id** (*str*) – RCSB PDB ID
- **dest** (*str, optional*) – Full path to where file should be saved
- **quiet** (*bool, optional*) – Supresses output when set to true

Returns
emap – emap object ready for processing.

Return type
emap

Process File

Once you have an *emap* object generated by a successful parse, you can construct the graph model of your protein crystal structure. The way this graph is constructed is extremely flexible and customizable. Each edge connecting two vertices is assigned a weight:

$$P' = -log_{10}(\epsilon)$$

where ϵ is a distance-dependent hopping penalty function:

$$\epsilon = \alpha \exp(-\beta(R - R_{offset}))$$

You can specify custom fragments atom by atom using the "custom" optional argument, which takes a string formatted in pyemap's custom fragment selection syntax, which is based on PDB atom serial number. The syntax is summarized below:

Syntax	
","	defines discrete range of atoms
"-"	defines continuous range of atoms
"()"	encloses atom range
"(…),(…)"	defines multiple custom fragments

Process

`pyemap.process`(*emap*, *chains=None*, *eta_moieties=None*, *dist_def='COM'*, *sdef='RSA'*, *edge_prune='PERCENT'*, *include_residues=['Y', 'W']*, *custom=''*, *distance_cutoff=20*, *max_degree=4*, *percent_edges=1.0*, *num_st_dev_edges=1.0*, *rd_thresh=3.03*, *rsa_thresh=0.2*, *coef_alpha=1.0*, *exp_beta=2.3*, *r_offset=0.0*)

Constructs emap graph theory model based on user specs, and saves it to the emap object.

Parameters

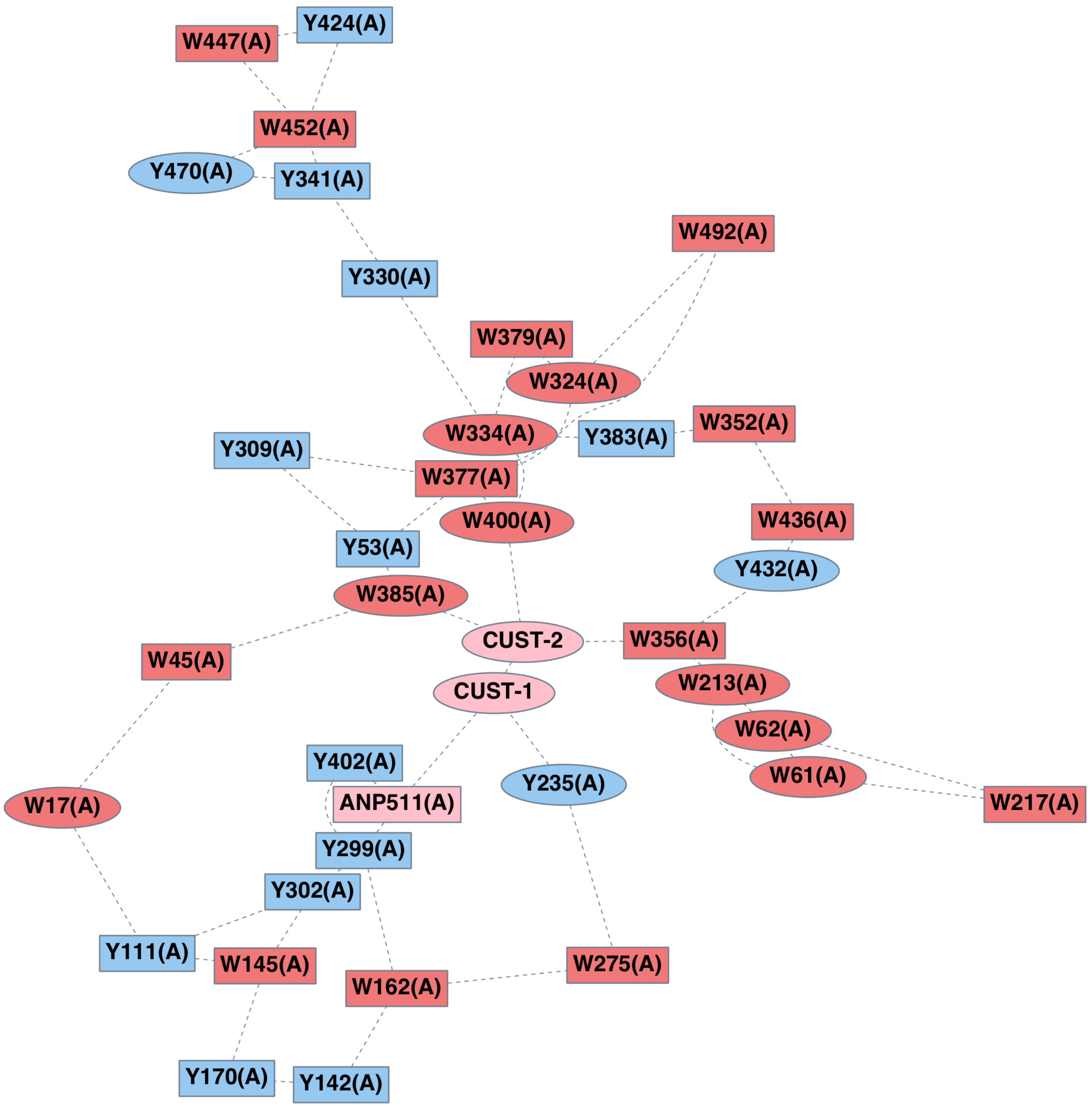
- **emap** (*emap*) – Object for storing state of emap analysis.
- **chains** (*list of str*) – List of strings corresponding to chains included in analysis
- **eta_moieties** (*list of str*) – List of strings corresponding to residue names of eta moieties
- **dist_def** (*str, optional*) – Definition of distance matrix. 'COM' for center of mass, 'CATM' for closest atom
- **sdef** (*str, optional*) – Algorithm to use for surface exposure. 'RD' for residue depth, 'RSA' for relative solvent accessibility
- **edge_prune** (*str, optional*) – Algorithm for pruning edges. 'DEGREE' for degree, 'PERCENT' for percent
- **include_residues** (*list of str*) – Included amino acids specified by 1 letter code
- **custom** (*str, optional*) – Custom atom string specified by user
- **distance_cutoff** (*float*) – Defines a pure distance threshold. PyeMap will only keep edges with distances less than or equal distance_cutoff.
- **max_degree** (*int, optional*) – Maximum degree of any vertex. Only used when edge_prune is set to 'DEGREE'.
- **percent_edges** (*float, optional*) – Percent of edges to keep for each node. Only used when edge_prune is set to 'PERCENT'.
- **num_st_dev_edges** (*float, optional*) – Number of standard deviations of edges to keep. Only used when edge_prune is set to 'PERCENT'.
- **rd_thresh** (*float, optional*) – Threshold for buried/surface exposed for residue depth
- **rsa_thresh** (*float, optional*) – Threshold for buried/surface exposed for relative solvent accessbility
- **coef_alpha** (*float, optional*) – Penalty function parameters.
- **exp_beta** (*float, optional*) – Penalty function parameters.
- **r_offset** (*float, optional*) – Penalty function parameters.

Raises

RuntimeError: – Not enough residues to construct a graph

Example

```
>>> import pyemap
>>> my_emap = pyemap.fetch_and_parse("1u3d")
>>> pyemap.process(my_emap,eta_moieties=["ANP511(A)"],custom="(3960-3969),(3970-3980,
↪3982,3984-3987)")
>>> my_emap.init_graph_to_Image().show()
```



Find Shortest Pathways

Introduction

Once you have a processed emap object which contains the graph theory model of the protein structure, you can search for pathways. There are two modes of search: source only and specified target. Pathways are stored as `ShortestPath` objects, which are organized into `Branch` objects. The data is stored in the `emap` object which was passed in. For a report of pathways found by pyemap, use `report()`.

Source only

When only a source node is selected, a NetworkX implementation of Dijkstra's algorithm is used to calculate the shortest path from the source to each surface-exposed residue. In the output, the pathways are organized into "branches" based on the first surface-exposed residue reached during the course of the pathway. The source node will be colored yellow in the graph visualization.

Specified target

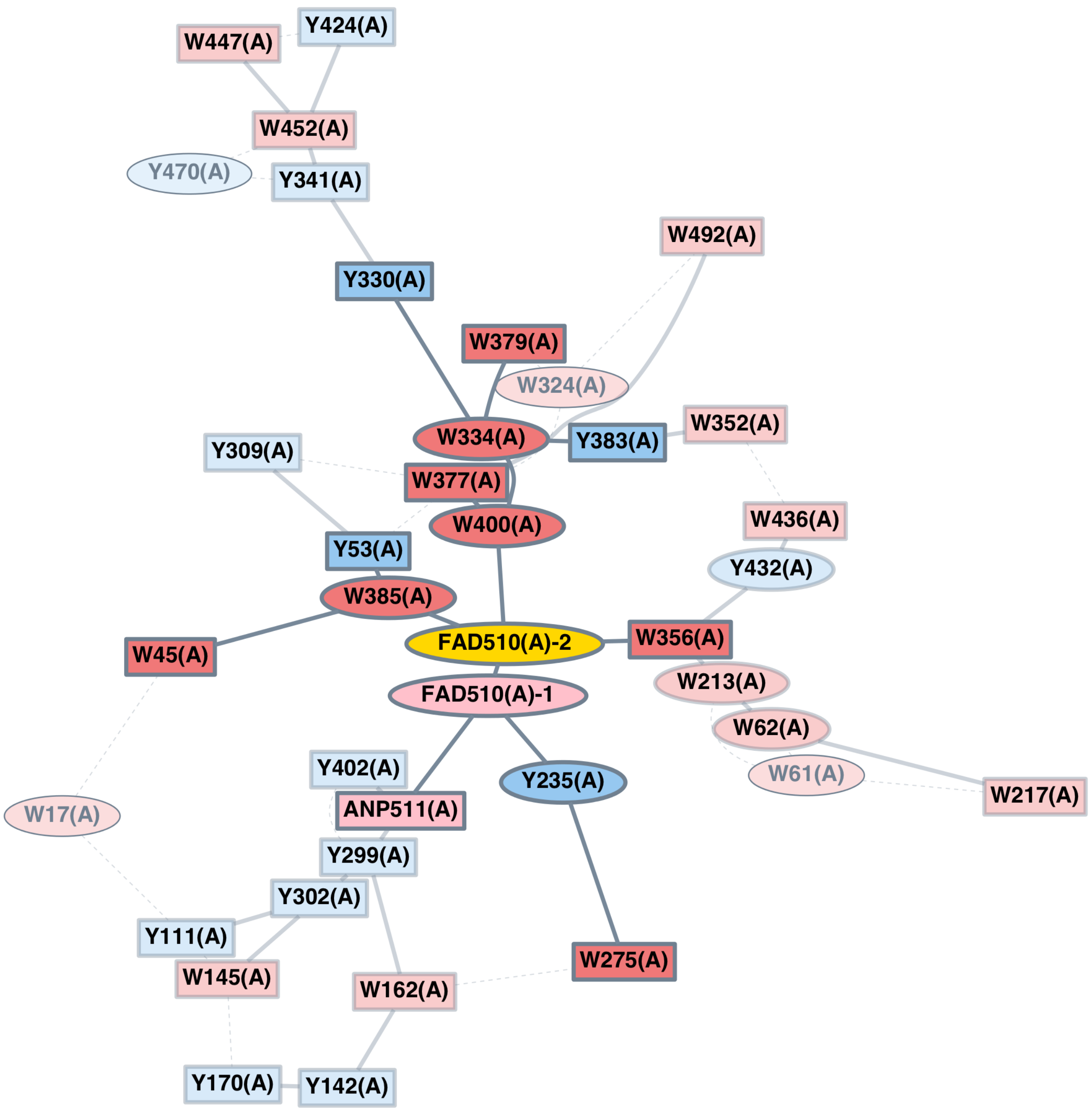
If a target is specified, a NetworkX procedure based on Yen's algorithm is used to calculate the shortest paths from source to target. The target does not need to be a surface-exposed residue. The target node will be colored blue in the graph visualization.

Examples

Source only:

```
>>> import pyemap
>>> my_emap = pyemap.fetch_and_parse("1u3d")
>>> pyemap.process(my_emap)
>>> pyemap.find_paths(my_emap,"FAD510(A)-2")
>>> print(my_emap.report())
Branch: W356(A)
1a: ['FAD510(A)-2', 'W356(A)'] 9.48
1b: ['FAD510(A)-2', 'W356(A)', 'Y432(A)', 'W436(A)'] 26.44
1c: ['FAD510(A)-2', 'W356(A)', 'W213(A)', 'W62(A)', 'W217(A)'] 34.72
Branch: ANP511(A)
2a: ['FAD510(A)-2', 'FAD510(A)-1', 'ANP511(A)'] 14.15
...
```

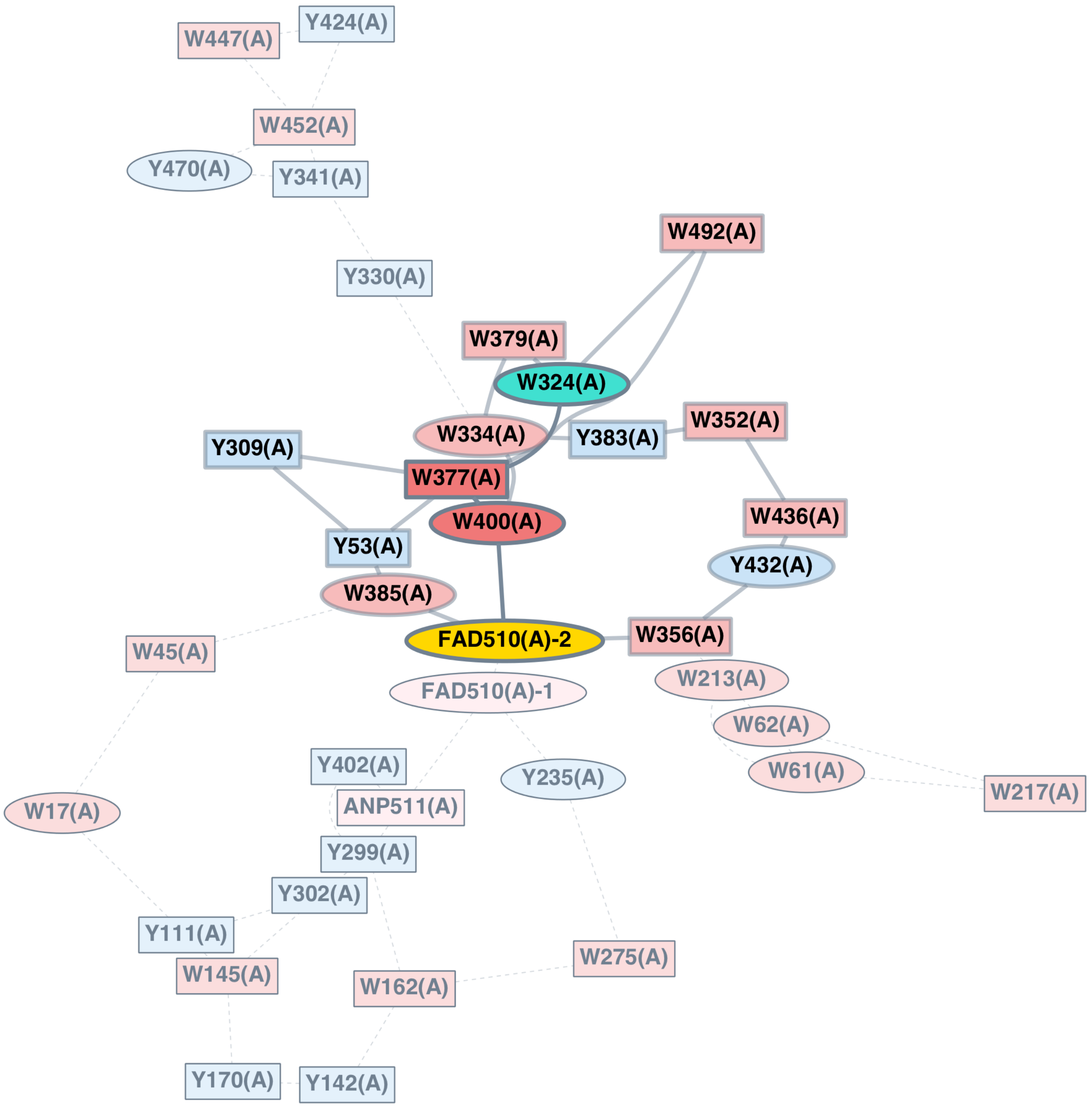
```
>>> my_emap.paths_graph_to_Image().show()
```



Specified Target:

```
>>> pyemap.find_paths(my_emap,"FAD510(A)-2", target = "W324(A)", max_paths=10)
>>> my_emap.report()
Branch: W324(A)
1a: ['FAD510(A)-2', 'W400(A)', 'W377(A)', 'W324(A)'] 24.15
1b: ['FAD510(A)-2', 'W385(A)', 'Y53(A)', 'W377(A)', 'W324(A)'] 35.25
1c: ['FAD510(A)-2', 'W400(A)', 'W334(A)', 'W379(A)', 'W324(A)'] 36.37
1d: ['FAD510(A)-2', 'W400(A)', 'W377(A)', 'W492(A)', 'W324(A)'] 49.20
1e: ['FAD510(A)-2', 'W385(A)', 'Y53(A)', 'Y309(A)', 'W377(A)', 'W324(A)'] 50.67
...
```

```
>>> my_emap.paths_graph_to_Image().show()
```



Find Paths

pyemap.pathway_analysis.**find_paths**(*emap*, *source*, *target=None*, *max_paths=10*)

Function which calculates pathways from source to target or surface exposed residues.

Performs shortest path analysis on source and (optionally) target residues.

Parameters

- **emap** (*emap*) – Object for storing state of emap analysis.

- **source** (*str*) – source node for analysis
- **target** (*str, optional*) – target node for analysis
- **max_paths** (*int, optional*) – maximum number of paths to search for in yen's algorithm

Data Structures

emap

class pyemap.**emap**(*file_path*, *pdb_id*, *eta_moieties*, *chain_list*, *sequences*)

Manages the data generated at all stages of PyeMap analysis.

file_path

Crystal structure file being analyzed by PyeMap.

Type
str

eta_moieties

Non-protein eta moieties automatically identified at the parsing step.

Type
dict of str: `Bio.PDB.Residue.Residue`

chain_list

List of chains identified at the parsing step.

Type
list of str

sequences

Amino acid sequence for each chain in FASTA format

Type
dict of str, str

residues

Residues included in the graph after the process step.

Type
dict of str: `Bio.PDB.Residue.Residue`

user_residues

Custom residues specified by the user.

Type
dict of str: `Bio.PDB.Residue.Residue`

init_graph

Graph generated after the process step.

Type
`networkx.Graph`

branches

Branches found by PyeMap analysis

Type
dict of int: *Branch*

paths

Paths found by PyeMap sorted by lowest to highest score.

Type
dict of str: *ShortestPath*

paths_graph

Graph generated after the shortest paths step.

Type
`networkx.Graph`

__init__(*file_path*, *pdb_id*, *eta_moieties*, *chain_list*, *sequences*)

Initializes emap object.

Parameters

- **file_path** (*str*) – Name of file
- **eta_moieties** (list of `Bio.PDB.Residue.Residue`) – Customized residue objects generated for automatically detected eta moieties
- **chain_list** (*list of str*) – Chains identified by the parser
- **sequences** (*dict of str:str*) – Key is chain id, value is sequence in fasta format

get_surface_exposed_residues()

Returns list of surface exposed residues.

Returns
surface_exposed – List of surface exposed residues identified by pyemap

Return type
list of str

init_graph_to_Image()

Returns PIL image of initial graph

Returns
img

Return type
`PIL.Image.Image`

init_graph_to_file(*dest=''*)

Saves image of graph generated by process step to file.

Parameters
dest (*str*) – Destination for writing to file.

paths_graph_to_Image()

Returns PIL image of pathways graph

Returns
img

Return type
`PIL.Image.Image`

paths_graph_to_file(*dest=''*)

Saves image of graph generated by pathways step to file.

Parameters
dest (*str*) – Destination for writing to file.

report(*dest=''*)

Returns report of most probable pathways. Writes to file if destination is specified.

Parameters
dest (*str, optional*) – Destination for writing to file

Returns
output – Formatted report of pathways found

Return type
str

Raises
RuntimeError – Nothing to report

residue_to_Image(*resname*, *scale=1.0*)

Returns PIL image of chemical structure. :param resname: Name of residue :type resname: str :param scale: Output scaling factor, default dimensions are (100,100) :type scale: float, optional

Returns
img

Return type
PIL.Image.Image

residue_to_file(*resname*, *dest=''*, *size=(100, 100)*)

Saves image of residue to file in .svg format.

Parameters

- **resname** (*str*) – Name of residue (node label) to be saved to file.
- **dest** (*str, optional*) – destination to save the image
- **size** (*(float,float), optional*) – dimensions of image saved to file

visualize_pathway_in_nglview(*ptid*, *view*)

Visualize pathway in nglview widget

Parameters

- **ptid** (*str*) – Pathway ID to be visualized
- **view** (nglview.widget.NGLWidget) – NGL Viewer widget

Branch

Branches are stored in the *branches* dictionary of the *emap* object. The keys are the branch numbers.

```
>>> print(my_emap.branches[1])
>>> Branch: W356(A)
>>> 1a: ['FAD510(A)-2', 'W356(A)'] 9.48
    >>> 1b: ['FAD510(A)-2', 'W356(A)', 'W213(A)'] 15.31
    >>> 1c: ['FAD510(A)-2', 'W356(A)', 'W213(A)', 'W61(A)'] 23.95
    >>> 1d: ['FAD510(A)-2', 'W356(A)', 'Y432(A)', 'W436(A)'] 26.44
    >>> 1e: ['FAD510(A)-2', 'W356(A)', 'W213(A)', 'W62(A)', 'W217(A)'] 34.72
```

class pyemap.**Branch**(*branch_id*, *target*)

Data structure used to group shortest paths with a common first surface exposed residue.

Shortest paths are classified into branches based on the first surface exposed residue reached during the course of the pathway. For example, given a source node A, and surface exposed nodes B and C, the paths [A,F,B] and [A,F,B,C] will both be part of the "B" branch. The path [A,E,C] would be part of its own 'C" branch.

branch_id

Unique identifier for a branch

Type
str

target

Target node which a branch corresponds to

Type
str

paths

List of ShortestPath objects that make up a branch

Type
list of *ShortestPath*

__str__()

String representation of Branch. First line is: "Branch: *branch_id*" and subsequent lines are the string representations of each ShortestPath object comprising the Branch.

add_path(*path*)

Adds a path to the branch and sets the path_id.

Each time a path is added, the paths in the branch are sorted. After sorting, each path is assigned a path id composed of the branch id and its location in the paths list. For example, the shortest path in branch 12 would be assigned the id '12a', the second shortest '12b' and so on.

Parameters
path (*ShortestPath*) – A ShortestPath from source to a surface exposed residue

ShortestPath

ShortestPaths are stored in the *paths* dictionary of the `emap` object. The keys are the path IDs.

```
>>> print(my_emap.paths['1a'])
>>> 1a: ['FAD510(A)-2', 'W356(A)'] 9.48
```

class pyemap.**ShortestPath**(*path*, *edges*, *length*)

Data structure used to store shortest paths.

Contains functions for comparison and string representation, both for the user output and in NGL viewer selection language for the purpose of visualization. Sorting for ShortestPath objects is done based on the length attribute.

path

List of residue names that make up the shortest path

Type
list of str

`edges`

List of edge weights that make up the shortest path

Type
list of float

`path_id`

List of residues that make up the shortest path

Type
list of str

`length`

Total distance from source to target

Type
float

`selection_strs`

NGL selection strings for visualization

Type
list of str

`color_list`

Colors of residues in visualization

Type
list of str

`labeled_atoms`

Atom names which are labeled in NGL visualization

Type
list of str

`label_texts`

Labels of residues in NGL visualization

Type
list of str

`set_id`(*path_id*)

Setter for path_id

`set_visualization`(*selection_strs*, *color_list*, *labeled_atoms*, *label_texts*)

Saves information needed for NGL visualization.

Algorithms

Non-protein electron transfer active moieties

Introduction

Frequently, protein crystal structures contain residues which are not amino acids, and do not belong to the polypeptide chain(s). Many of these residues can play significant roles in electron/hole transfer. PyeMap automatically identifies non-protein electron/hole transfer (ET) active moieties, and gives users the option to include them in the analysis. In the current implementation, non-protein ET active moieties identified by PyeMap are non-amino acid aromatic sites, extended conjugated systems, and a pre-defined list of metal clusters and redox active metal ions. For a given non-standard co-factor (e.g., flavin adenine dinucleotide), there can be multiple non-protein ET active moieties identified by PyeMap, and they will appear as separate nodes on the graph if selected for analysis.

Identification

Aromatic moieties and extended conjugated chains

After initial parsing, non-protein residues are analyzed for detection of ET active moieties. For each non-standard residue, a chemical graph is constructed using the NetworkX library, consisting of the O, C, N, P and S atoms in the residue. To isolate the conjugated systems, an edge is only drawn between two atoms j and k if:

$$r_{\mathrm{jk}} \leq \ \overline{x} - 2\sigma_{\overline{x}}$$

where x is the mean single-bond distance between those two elements, and $\sigma_{\overline{x}}$ is the standard deviation. If there are any conjugated systems, the resulting chemical graph will be a forest of connected component subgraphs. Each subgraph that contains a cycle, or consists of 10 or more atoms will be considered a non-protein ET active moiety, and can be selected for the analysis.

As one would expect, the structures generated by this procedure are not always correct, often due to poor resolution in PDBs. We have two functions which try to correct the chemical graph in order to generate proper structures. The first is `cleanup_bonding()`, which connects atoms that are within experimental single bond lengths and are only connected to 1 or 2 neighbors. This function often fixes broken aromaticity. The second is `remove_side_chains()`, which recursively removes non-aromatic side chains from aromatic moieties.

The final chemical graph is used to construct a SMILES string using the pysmiles package, which can be visualized using standard cheminformatics tools such as RDKIT.

Clusters

The PyeMap repository contains a list of 66 inorganic clusters which are automatically identified by their 3 character residue names. All atoms in the residue are collected as part of the customized residue object, and a pre-rendered image is used for visualization of chemical structure. Otherwise, they can be used and interacted with just like any other residue. The list of clusters and pre-rendered images were obtained from the Protein Data Bank in Europe (PDBe).

Redox-active Metal ions

PyeMap automatically identifies a set of redox-active metal ions to include as residues in the graph.

Table 2.1: Available metal ions

Element	Charges
Cu	+1, +2, +3
Fe	+2, +3
Mn	+2, +3
Co	+2, +3
Mo	0, +4, +6
Ni	+2
Cr	+3

Visualization

Chemical structures of residues(not including user-specified residues) can be visualized using the *residue_to_Image()*, *init_graph_to_Image()* functions. SMILES strings and NGL Viewer selection strings are also accessible through the *emap* object. Note that SMILES strings are not available for clusters and user-specified residues.

Source

pyemap.custom_residues.find_conjugated_systems(...)	Finds conjugated systems within a BioPython residue object, and returns them as individual customized BioPython Residue objects.
pyemap.custom_residues.process_custom_residues(...)	Identifies and returns customized Bio.PDB.Residue objects corresponding to electron transfer active moieties.
pyemap.structures.cleanup_bonding(res_graph)	Connects nodes that should be connected to fix broken aromaticity.
pyemap.structures.remove_side_chains(res_graph)	Removes non-aromatic sides chains on aromatic eta moieties.

pyemap.custom_residues.find_conjugated_systems

`pyemap.custom_residues.`**`find_conjugated_systems`**(*atoms*, *res_names*)

Finds conjugated systems within a BioPython residue object, and returns them as individual customized BioPython Residue objects.

Parameters

- **`atoms`** (*`array-like`*) – List of atoms in the residue
- **`res_names`** (*`arary-like`*) – List of already used names for custom residues

Returns

custom_res_list – List of customized `Bio.PDB.Residue.Residue`

Return type

array-like

pyemap.custom_residues.process_custom_residues

`pyemap.custom_residues.`**`process_custom_residues`**(*non_standard_residue_list*)

Identifies and returns customized Bio.PDB.Residue objects corresponding to electron transfer active moieties.

Parameters

`non_standard_residue_list` (list of `Bio.PDB.Residue.Residue`) – List of non-protein residues in the structure

Returns

custom_res – List of customized BioPython residue objects corresponding to electron transfer active moieties that are not part of standard protein residues

Return type

list of `Bio.PDB.Residue.Residue`

pyemap.structures.cleanup_bonding

`pyemap.structures.`**`cleanup_bonding`**`(`*res_graph*`)`

Connects nodes that should be connected to fix broken aromaticity.

Parameters
`res_graph` (`networkx.Graph`) – residue graph

pyemap.structures.remove_side_chains

`pyemap.structures.`**`remove_side_chains`**`(`*res_graph*`)`

Removes non-aromatic sides chains on aromatic eta moieties.

Parameters
`res_graph` (`networkx.Graph`) – residue graph

Graph construction

Introduction

The first step to constructing the graph theory model is constructing a pairwise distance matrix for the selected amino acid residues. The distance is calculated either between centers of mass of the side chains, or between their closest atoms. For standard protein residues, only side chain atoms are considered in the calculation. All atoms of automatically identified non-protein ET active moieties and user-specified custom fragments are considered in the distance calculations. From the distance matrix, an undirected weighted graph is constructed using NetworkX, initially with the calculated distances as weights.

Penalty Functions

The next step is to recast the weights as modified distance dependent penalty functions:

$$P' = -log_{10}(\epsilon)$$

where:

$$\epsilon = \alpha \exp(-\beta(R - R_{offset}))$$

, , and R_{offset} are hopping parameters, similar to the through-space tunneling penalty function in the Pathways model [Beratan1992]. All subsequent calculations are performed using the modified penalty functions as edge weights. When using default hopping parameters (= 1.0, = 2.3, Roffset = 0.0), the edge weights will be equal to the distances (multiplied by a prefactor of $2.3 * log_{10}(e)$ 1).

Edge Pruning

One of two algorithms is used to prune the edges of the graph, which is specified by the `edge_prune` keyword argument to `process()`.

Percent-based algorithm (default)

This algorithm considers only the smallest `percent_edges` % of edges by weight per node, and then prunes based on the mean and standard deviation of the weights of the remaining edges.

Algorithm 1 Prune by Percent

```
procedure PRUNE(G(V,E), percent_edges, num_st_dev_edges, distance_cutoff)
    for v in V do
        for e in v.edges do
            if e.distance > distance_cutoff or e.weight > percentileweight(percent_edges) then
                G.remove(e)
            end if
        end for
        $\bar{l}$ = mean_weight(v.edges)
        $\sigma$ = st_dev_weight(v.edges)
        for e in v.edges do
            if e.weight > $\bar{l}$ + num_st_dev_edges $\cdot$ $\sigma$ then
                G.remove(e)
            end if
        end for
    end for
end procedure
```

`percent_edges`, `num_st_dev_edges`, and `distance_cutoff` are specified as keyword arguments to `process()`.

Specify `edge_prune='PERCENT'` to use this algorithm.

Degree-based algorithm

This algorithm greedily prunes the largest edges by weight of the graph until each node has at most `max_degree` neighbors.

Algorithm 2 Prune by Degree

```
procedure PRUNE(G(V,E), max_degree, distance_cutoff)
    removal_candidates = []
    for e in E do
        if e['distance'] > distance_cutoff then
            G.remove(e)
        end if
    end for
    for v in V do
        if degree(v) > D then
            removal_candidates.append(v.edges)
        end if
    end for
    sort_by_weight_descending(removal_candidates)
    for e(u,v) in removal_candidates do
        if degree(u) > max_degree or degree(v) > max_degree then
            G.remove(e)
        end if
    end for
end procedure
```

`max_degree` and attr:*distance_cutoff*: are specified as keywords arguments to `process()`.

Specify `edge_prune='DEGREE'` to use this algorithm. This algorithm is recommended when doing *Protein Graph Mining*.

Visualization and further analysis

The graph can be interacted with and written to file using the *emap* object. The graph is visualized using PyGraphviz and Graphviz. The graph is stored as a `networkx.Graph` object in the `init_graph` and `paths_graph` attributes of the *emap* object.

```
>>> G = my_emap.init_graph
>>> print(G.edges[('W17(A)', 'W45(A)')]['distance'])
>>> 12.783579099370808
```

Source

pyemap.process_data.filter_by_degree(G, ...)	
pyemap.process_data.filter_by_percent(G, ...)	
pyemap.process_data.create_graph(dmatrix, ...)	Constructs the graph from the distance matrix and node labels.
pyemap.process_data.pathways_model(dist, ...)	Applies penalty function parameters and returns score.

pyemap.process_data.filter_by_degree

pyemap.process_data.**filter_by_degree**(*G*, *max_degree*, *distance_cutoff*, *coef_alpha*, *exp_beta*, *r_offset*)

pyemap.process_data.filter_by_percent

pyemap.process_data.**filter_by_percent**(*G*, *percent_edges*, *num_st_dev_edges*, *distance_cutoff*, *coef_alpha*, *exp_beta*, *r_offset*)

pyemap.process_data.create_graph

pyemap.process_data.**create_graph**(*dmatrix*, *node_labels*, *edge_prune*, *coef_alpha*, *exp_beta*, *r_offset*, *distance_cutoff*, *percent_edges*, *num_st_dev_edges*, *max_degree*, *eta_moieties*)

Constructs the graph from the distance matrix and node labels.

Parameters

- **dmatrix** (*numpy.array of float*) – Distance matrix of aromatic residues.
- **node_label** (*list of str*) – Labels for residues in the graph.
- **edge_prune** (*int*) – 0 for degree, 1 for percent
- **residue_numbers** – res numbers
- **distance_cutoff** (*float*) – Parameters that determine which edges are kept.
- **max_degree** (*float*) – Parameters that determine which edges are kept.
- **eta_moieties** (*list of str*) – Non standard residues that were automatically identified

Returns

G – Graph of aromatic residues in protein

Return type

networkx.Graph

References

Gray, H. B.; Winkler, J. R. Long-Range Electron Transfer. Proc. Natl. Acad. Sci. U. S. A. 2005, 102 (10), 3534 LP-3539. Reference for 20A filter on edges

pyemap.process_data.pathways_model

`pyemap.process_data.`**`pathways_model`**`(`*dist*, *coef_alpha*, *exp_beta*, *r_offset*`)`

Applies penalty function parameters and returns score.

$\epsilon = \alpha \exp(-\beta(R - R_{offset}))$

Parameters

- **`dist`** (*`float`*) – Actual distance in angstroms
- **`coef_alpha`** (*`float`*) – Penalty function parameters
- **`exp_beta`** (*`float`*) – Penalty function parameters
- **`r_offset`** (*`float`*) – Penalty function parameters

Returns
mod_penalty

Return type
float

Surface exposed residues

Introduction

Electron (or hole) transfer often proceeds from/to surface residues to/from an acceptor/donor inside the protein. Therefore, identification of surface-exposed residues is a key step for prediction of relevant electron/hole transfer pathways. Users can select one of two parameters to classify residues as buried/exposed: residue depth and relative solvent accessibility. In the graph images, buried residues will appear as ovals, while exposed residues will appear as rectangles.

Residue depth

Residue depth is a measure of solvent exposure that describes the extent to which a residue is buried within the protein structure. The parameter was first introduced by Chakravarty [Chakravarty1999] and coworkers, and is computed in PyeMap using the freely available program MSMS. MSMS computes a solvent-excluded surface by rolling a probe sphere along the surface of the protein, which is represented as atomic spheres. The boundary of the volume reachable by the probe is taken to be the solvent-excluded surface. The residue depth for each residue is calculated as the average distance of its respective atoms from the solvent-excluded surface [Sanner1996]. In PyeMap, the threshold for classifying residues as buried/exposed is:

$$\mathbf{RD} \leq \mathbf{3.03}$$

which is the threshold proposed by Tan and coworkers [Tan2009]. Residues 3.03 Å and shallower will be classified as exposed in the final graph; those deeper will be classified as buried. This threshold can be customized by passing the the `rd_thresh` keyword argument to `process()`.

Please note that MSMS is not available on Mac OS > 10.15, as newer Mac OS do not support 32-bit applications.

Relative Solvent Accessibility

Accessible surface area is a measure of solvent exposure, first introduced by Lee and Richards, which describes the surface area of a biomolecule that is accessible to solvent molecules [Lee1971]. To calculate the accessible surface of each atom, a water sphere is rolled along the surface of the protein, making the maximum permitted van der Waals contacts without penetrating neighboring atoms [Shrake1973]. The total accessible surface area for a residue is the sum of the solvent accessible surface areas of its respective atoms. In order to develop a threshold to classify residues as buried or exposed, calculated ASA values need to be normalized based on corresponding reference values for a given residue. This requires precomputed or predefined maximal accessible surface area (MaxASA) for all residues. MaxASA is the maximal possible solvent accessible surface area for a given residue. MaxASA values are obtained from theoretical calculations of Gly-X-Gly tripeptides in water, where X is the residue of interest. From ASA and MaxASA, the relative solvent accessibility (RSA) can be calculated by the formula:

$$RSA = \frac{ASA}{MaxASA}$$

Several scales for MaxASA have been published. PyeMap uses the most recent theoretical scale from Tien and coworkers [Tien2013]. Relative solvent accessibility is calculated using the DSSP program developed by Kabsch and Sander [Sander1983]. In PyeMap, the RSA threshold chosen for exposed residues is:

$$\mathbf{RSA} \geq \mathbf{0.05}$$

as recommended by Tien and coworkers. Residues with RSA greater than equal to 0.05 will be classified as exposed, those with lower RSA values will be classified as buried. This threshold can be customized by passing the the `rsa_thresh` keyword argument to `process()`.

Source

pyemap.process_data.calculate_residue_depth(...)	Returns a list of surface exposed residues as determined by residue depth.
pyemap.process_data.calculate_rsa(filename, ...)	Returns a list of surface exposed residues as determined by relative solvent accessibility.

pyemap.process_data.calculate_residue_depth

pyemap.process_data.**calculate_residue_depth**(*model*, *aromatic_residues*, *rd_cutoff*)

Returns a list of surface exposed residues as determined by residue depth.

Parameters

- **filename** (*str*) – Name of pdb file to be analyzed
- **aromatic_residues** (list of `Bio.PDB.Residue.Residue`) – residues included in the analysis
- **rd_cutoff** (*float*) – Cutoff for buried/surface exposed

Returns

surface_exposed_res – List of residue names corresponding to the surface exposed residues

Return type

list of str

pyemap.process_data.calculate_rsa

pyemap.process_data.**calculate_rsa**(*filename*, *model*, *node_list*, *rsa_cutoff*)

Returns a list of surface exposed residues as determined by relative solvent accessibility.

Only standard protein residues are currently supported. Non-protein and user specified custom residues cannot be classified as surface exposed using this criteria.

Parameters

- **filename** (*str*) – Name of pdb file to be analyzed
- **model** (Bio.PDB.Model.Model) – Model under analysis
- **node_list** (*list of str*) – List containing which standard residues are included in analysis
- **rsa_cutoff** (*float*) – Cutoff for buried/surface exposed

References

Tien, M. Z.; Meyer, A. G.; Sydykova, D. K.; Spielman, S. J.; Wilke, C. O. PLoS ONE 2013, 8 (11).
Reference for relative solvent accessibility cutoff of 0.05, and for MaxASA values

Shortest paths

Introduction

Once you have a processed *emap* object, you can search for electron/hole transfer pathways. There are two modes: *source only*, or *specified target*.

Source only

When only a source is specified, Dijkstra's algorithm is used to find the shortest path between the specified source and every surface-exposed residue in the graph. The pathways are classified into branches based on the first surface-exposed residue reached along the pathway. Within a branch, the pathways are ranked according to their score.

For an example of how the branches are structured, refer to the example below. In this example, the flavin group on FAD (FAD510(A)-2) is specified as the source, and W356(A), W436(A), and ANP511(A) have been identified as surface-exposed residues. For W436(A), the shortest path involves first going through W356(A), which itself is a surface-exposed residue. Thus this path is assigned to branch W356(A), and given the ID 1b, as it is the second shortest path in this branch. The shortest path to ANP511(A) is given the ID 2a, and belongs to a different branch.

Example 1: Source only

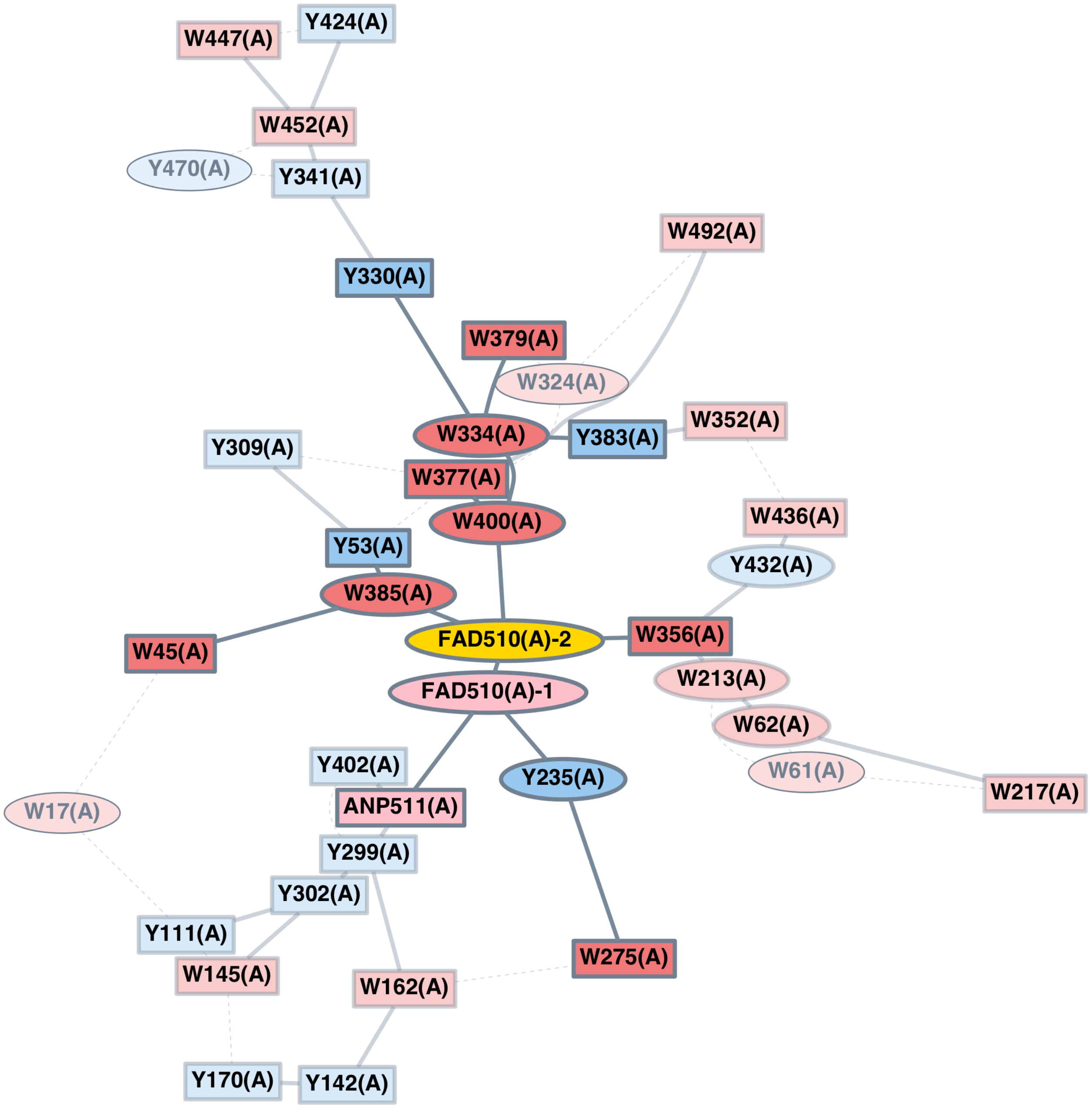
```
>>> import pyemap
>>> my_emap = pyemap.fetch_and_parse("1u3d")
>>> pyemap.process(my_emap)
>>> pyemap.find_paths(my_emap,"FAD510(A)-2")
>>> print(my_emap.report())
Branch: W356(A)
1a: ['FAD510(A)-2', 'W356(A)'] 9.48
1b: ['FAD510(A)-2', 'W356(A)', 'Y432(A)', 'W436(A)'] 26.44
1c: ['FAD510(A)-2', 'W356(A)', 'W213(A)', 'W62(A)', 'W217(A)'] 34.72
```

(continues on next page)

(continued from previous page)

```
Branch: ANP511(A)
2a: ['FAD510(A)-2', 'FAD510(A)-1', 'ANP511(A)'] 14.15
...
```

```
>>> my_emap.paths_graph_to_Image().show()
```



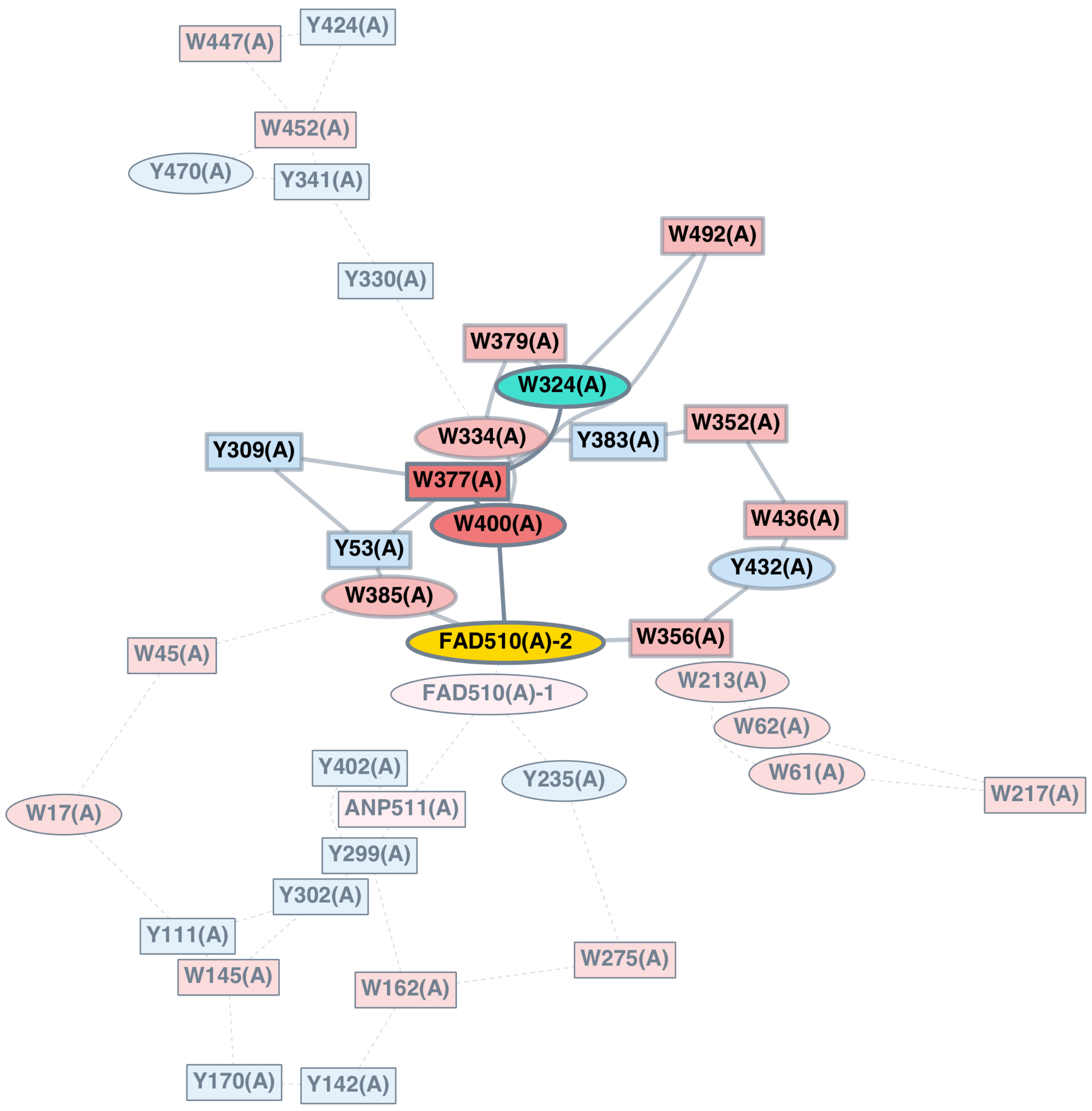
Specified target

When a source and a target are specified, a NetworkX procedure based on Yen's algorithm is used to find the 5 shortest paths from source to target. Yen's algorithm exploits the idea that shortest paths are likely to share common steps, and is able to compute the k shortest paths between nodes in a graph with non-negative weights. The procedure first finds the shortest path, then finds the next 4 shortest deviations. The pathways are ranked according to their score. See the example below.

Example 2: Specified Target:

```
>>> pyemap.find_paths(my_emap,"FAD510(A)-2", target = "W324(A)", max_paths=10)
>>> my_emap.report()
Branch: W324(A)
1a: ['FAD510(A)-2', 'W400(A)', 'W377(A)', 'W324(A)'] 24.15
1b: ['FAD510(A)-2', 'W385(A)', 'Y53(A)', 'W377(A)', 'W324(A)'] 35.25
1c: ['FAD510(A)-2', 'W400(A)', 'W334(A)', 'W379(A)', 'W324(A)'] 36.37
1d: ['FAD510(A)-2', 'W400(A)', 'W377(A)', 'W492(A)', 'W324(A)'] 49.20
1e: ['FAD510(A)-2', 'W385(A)', 'Y53(A)', 'Y309(A)', 'W377(A)', 'W324(A)'] 50.67
...
```

```
>>> my_emap.paths_graph_to_Image().show()
```



Source

pyemap.shortest_paths. dijkstras_shortest_paths(G, ...)	Returns shortest path from source to each surface exposed residue.
pyemap.shortest_paths. yens_shortest_paths(G, ...)	Returns shortest paths from source to target.

pyemap.shortest_paths.dijkstras_shortest_paths

`pyemap.shortest_paths.`**`dijkstras_shortest_paths`**`(`*G*, *start*, *targets*`)`

Returns shortest path from source to each surface exposed residue.

Performs Dijkstra's algorithm from the source to each surface exposed residue, finding the shortest path. The ShortestPath objects are organized into branches based on the first surface exposed residue reached during the course of the pathway.

Parameters

- **`G`** (`networkx.Graph`) – Undirected, weighted residue graph
- **`start`** (*`str`*) – Source node
- **`targets`** (*`list of str`*) – List of surface exposed residues

Returns

branches – A list of Branch objects representing the groups of pathways found

Return type

list of *`Branch`*

Raises

`RuntimeError:` – No shortest paths to surface found

pyemap.shortest_paths.yens_shortest_paths

`pyemap.shortest_paths.`**`yens_shortest_paths`**`(`*G*, *start*, *target*, *max_paths=10*`)`

Returns shortest paths from source to target.

Uses Yen's algorithm to calculate the shortest paths from source to target, writes out the ShortestPath objects to file, and returns the 10 pathway IDs. In the graph, nodes and edges that are part of any pathways are made opaque, and the shortest path is highlighted.

Parameters

- **`G`** (`networkx.Graph` object) – Undirected, weighted residue graph
- **`start`** (*`str`*) – Source node
- **`target`** (*`str`*) – Target node
- **`max_paths`** (*`int, optional`*) – Maximum number of paths to search for

Returns

A list of length 1 containing a single Branch object which represents the group of pathways found.

Return type

list of *`Branch`* objects

References

Jin Y. Yen, Finding the K Shortest Loopless Paths in a Network, Management Science, Vol. 17, No. 11, Theory Series (Jul., 1971), pp. 712-716.

Raises

`PyeMapShortestPathException:` – No shortest paths to target found.

2.4.2 Protein Graph Mining

Usage

Graph mining with PyeMap occurs in 4 steps:

1. Generate protein graphs
2. Classify nodes and edges
3. Find subgraph patterns
4. Find and cluster protein subgraphs

Step 1: Generate Protein Graphs

The first step is to fetch and/or parse a list of PDBs using *`fetch_and_parse()`* or *`parse()`*, and add them to the *`PDBGroup`* object. Once all of the eMap objects have been collected, the second step is to call *`process_emaps()`*, which uses the same infrastructure as *Single Protein Analysis* to generate the protein graphs.

```
pg = pyemap.graph_mining.PDBGroup("My Group")
#pdb_ids = ['1u3d','1iqr'...]
for pdb in pdb_ids:
    pg.add_emap(pyemap.fetch_and_parse(pdb))
```

Step 2: Classify nodes and edges

The next step is to classify the nodes and edges using *`generate_graph_database()`*. In many cases, this function can be called with no arguments, but in some cases it can be useful to allow for node substitutions, or to specify edge thresholds. See the *classification* section for more details.

```
pg.generate_graph_database()
```

Step 3: Find Subgraph Patterns

The next step is to find subgraph patterns which are shared among the protein graphs. One can either mine for all patterns up to a given support threshold:

```
pg.run_gspan(10)
```

Or search for a particular pattern or set of patterns:

```
pg.find_subgraph('WWW#')
```

See the *mining* section for more details.

Step 4: Find and cluster protein subgraphs

The results of the mining calculation are stored in the `subgraph_patterns` dictionary as *`SubgraphPattern`* objects. To find protein subgraphs, the *`find_protein_subgraphs()`* function must be called for the subgraph pattern of interest. The identified protein subgraphs are stored in the `protein_subgraphs` attribute of the *`SubgraphPattern`* object, and the clustering is described by the `groups` attribute. One can switch between different types of clustering using the *set_clustering()* function.

```
sg = pg.subgraph_patterns['1_WWW#_18']
sg.find_protein_subgraphs()
sg.set_clustering("sequence")
# print results, including clustering
print(sg.full_report())
```

Data Structures

PDB Group

The *PDBGroup* object is the primary data structure for graph mining analysis in PyeMap. It stores the *emap* objects which contain the graph theory models of the protein structures, the graph database defined by the classifications of nodes and edges, and the results of the mining calculation.

class `pyemap.graph_mining.`**`PDBGroup`**(*title*)

Contains all information regarding the group of proteins being analyzed, and all of the the subgraph patterns identified by the gSpan algorithm.

`title`

Title of PDB group

Type
str

`emaps`

Dict of PDBs being analyzed by PyeMap. The keys are PDB IDs, meaning that only one *emap* object per PDB ID is allowed.

Type
dict of str: *emap*

`subgraph_patterns`

Dict of subgraph patterns found by GSpan. Keys are the unique IDs of the *SubgraphPattern* objects.

Type
dict of str: *SubgraphPattern*

`fasta`

List of unaligned sequences in FASTA format

Type
str

`aligned_fasta`

List of sequences in FASTA format after multiple sequence alignment

Type
str

`__init__`(*title*)

Initializes PDBGroup object

Parameters
title (*str*) – Title of PDB group

add_emap(*emap_obj*)

Adds an *emap* object to the PDB group.

Parameters
emap_obj (*emap* object) – Parsed PDB generated by *parse()* or *fetch_and_parse()*

Examples

```
>>> my_pg.add_emap(pyemap.fetch_and_parse('1u3d'))
```

find_subgraph(*graph_specification*)

Mine for a specified subgraph pattern.

Linear chains can be specified simply by list the 1-letter amino acid codes/special characters, e.g. WWW. To specify branching, use a syntax similar to the SMILES format, where there is no specification of bonding and each character must be upper case and separated by brackets, e.g. [H]1[C][#][C]1. If edge thresholds are used, all possible combinations of edges will be searched for.

Parameters
graph_specification (*str*) – String of graph to search for

Notes

Special characters: * - wildcard character # - non-protein residue X - unknown residue

Examples

```
>>> my_pg.find_subgraph('WWW#')
```

generate_graph_database(*sub=[]*, *edge_thresh=[]*)

Generates graph database for mining.

Parameters

- **sub** (*list of str, optional*) – List of 1-character amino acid codes to be labeled as “X”. All other included standard amino acids receive their own category.
- **edge_thresholds** (*list of float, optional*) – List of edge thresholds. Edges with weight below the first value will be given the label 2, edges between the 1st and second values will be labeled as 3, and so on.

Examples

```
>>> pg.generate_graph_database(['W','Y'],[12,15])
```

mining_report(*dest=None*)

Generates general report of all subgraph patterns found in the analysis.

Parameters
dest (*str, optional*) – Destination to write report to file

Returns
report – General report of all subgraph patterns found in the analysis.

Return type
str

process_emaps(*chains={}*, *eta_moieties={}*, *include_residues=['Y', 'W']*, ***kwargs*)

Processes `emap` objects in order to generate protein graphs.

Should be executed once all of the `emap` objects have been added to the group.

Parameters

- **chains** (*dict of str: list of str, optional*) – Chains to include for each PDB. The special keyword 'All' is also accepted.
- **eta_moieties** (*dict of str: list of str, optional*) – Dict containing list of ETA moieties(specified by their residue label) to include for each PDB. By default, all on the included chains will be included.
- **include_residues** (*list of str, optional*) – List of 1-letter standard AA codes to include in the graph
- ****kwargs** – For a list of accepted kwargs, see the documentation for `process()`.

Examples

```
>>> eta_moieties = {'1u3d': ['FAD510(A)-2'], '1u3c': ['FAD510(A)-2'], '6PU0': [
↪'FAD501(A)-2'], '4I6G': ['FAD900(A)-2'], '2J4D': ['FAD1498(A)-2']}
>>> chains = {'1u3d': ['A'], '1u3c': ['A'], '6PU0': ['A'], '4I6G': ['A'], '2J4D
↪': ['A']}
>>> my_pg.process_emaps(chains=chains,eta_moieties=eta_moieties)
```

run_gspan(*min_support*, *min_num_vertices=4*, *max_num_vertices=inf*, ***kwargs*)

Mines for common subgraphs using gSpan algorithm. Results are stored as `SubgraphPattern` objects in the *subgraph_patterns* dictionary.

References

Yan, Xifeng, and Jiawei Han. "gspan: Graph-based substructure pattern mining." 2002 IEEE International Conference on Data Mining, 2002. Proceedings.. IEEE, 2002.

Parameters

- **min_support** (*int*) – Minimum support number of subgraphs in the search space
- **min_num_vertices** (*int, optional*) – Minimum number of nodes for subgraphs in the search space
- **max_num_vertices** (*int, optional*) – Maximum number of nodes for subgraphs in the search space
- ****kwargs** – See https://github.com/betterenvi/gSpan for a list of accepted kwargs.

Examples

```
>>> my_pg.run_gspan(10)
```

save_fasta(*dest=''*)

Saves fasta from multiple sequence alignment to file

Parameters
dest (*str, optional*) – Destination to write aligned fasta to file

Frequent Subgraph

The *pyemap.graph_mining.SubgraphPattern* object stores all of the data related to a subgraph pattern identified by the mining algorithm. This class is responsible for finding protein subgraphs in each PDB, and clustering them based on similarity.

class pyemap.graph_mining.**SubgraphPattern**(*G*, *graph_number*, *support*, *res_to_num_label*, *edge_thresholds*)

Stores all information regarding an identified subgraph pattern.

id

Unique identifier for subgraph pattern.

Type
str

G

Graph representation of this subgraph pattern.

Type
`networkx.Graph`

support

List of PDB IDs which contain this subgraph

Type
list of str

protein_subgraphs

Dict which contains protein subgraphs which match this pattern. Each entry has a unique identifier and a `networkx.Graph` derived from the graphs generated by the *emap* class which match the pattern of this pattern.

Type
dict of str: `networkx.Graph`

groups

Protein subgraph (IDs) clustered into groups based on similarity

Type
dict of str: list of str

support_number

Number of PDBs this subgraph pattern was identified in

Type
int

`__init__`(*G*, *graph_number*, *support*, *res_to_num_label*, *edge_thresholds*)

Initializes SubgraphPattern object.

Parameters

- **G** (`networkx.Graph`) – Graph representation of this subgraph pattern
- **graph_number** (*int*) – Unique numerical ID for this subgraph pattern
- **support** (*list of str*) – List of PDB IDs which contain this subgraph
- **res_to_num_label** (*dict of str: int*) – Mapping of residue types to numerical node labels
- **edge_thresholds** (*list of float*) – Edge thresholds which define edge labels

`find_protein_subgraphs`(*clustering_option='structural'*)

Finds protein subgraphs which match this pattern.

This function must be executed to analyze protein subgraphs.

Parameters

clustering_option (*str, optional*) – Either 'structural' or 'sequence'

Notes

Graphs are clustered by both sequence and structral similarity, and the results are stored in *self._structural_groups* and *self._sequence_groups*. The clustering_option argument used here determines which one of these groupings is used for *self.groups*. This can be changed at any time by calling `set_clustering()` and specifying the other clustering option.

`full_report`()

Returns a full report of all protein subgraphs which match this pattern.

Returns

full_str – Report of protein subgraphs which match this pattern

Return type

str

`general_report`()

Generates general report which describes this subgraph pattern.

Returns

full_str – General report which describes this subgraph pattern

Return type

str

`set_clustering`(*clustering_option*)

Sets clustering option.

Parameters

clustering_option (*str*) – Either 'structural' or 'sequence'.

Notes

Since both types of clustering are always computed by *pyemap.graph_mining.SubgraphPattern.find_protein_subgraphs()* all this function actually does is swap some private variables. The purpose of this function is to determine what kind of clustering gets shown in the report.

subgraph_to_Image(*id=None*)

Returns PIL image of subgraph pattern or protein subgraph

Parameters

id (*str, optional*) – Protein subgraph ID. If not specified, generic subgraph pattern will be drawn

Returns

img

Return type

`PIL.Image.Image`

subgraph_to_file(*id=None*, *dest=''*)

Saves image of subgraph pattern or protein subgraph to file

Parameters

- **id** (*str, optional*) – Protein subgraph ID. If not specified, generic subgraph pattern will be drawn
- **str** (*dest;*) – Destination to save the graph
- **optional** – Destination to save the graph

visualize_subgraph_in_nglview(*id*, *view*)

Visualize pathway in nglview widget

Parameters

- **id** (*str*) – Subgraph id to be visualized
- **view** (`nglview.widget.NGLWidget`) – NGL Viewer widget

Algorithms

Classification

Introduction

The efficiency and descriptive power of graph mining is enhanced when the algorithms are able to distinguish between different types of nodes and edges. Graph mining in PyeMap relies on each node and edge in the graph database being assigned a numerical label which corresponds to its category. PyeMap offers some customization of these labels in order to broaden or narrow the search space.

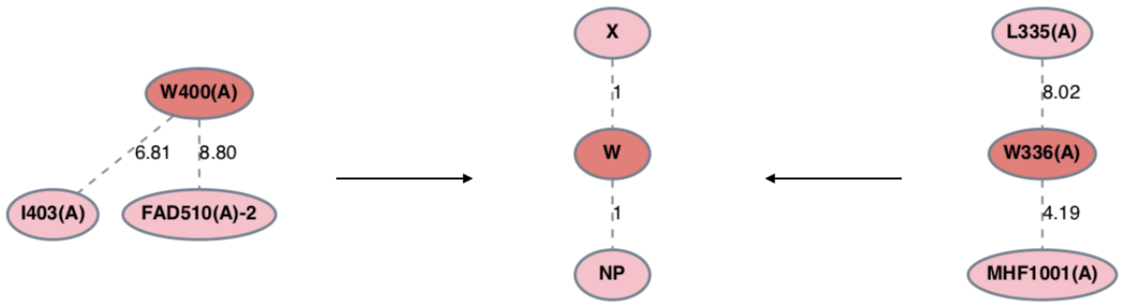
Nodes

By default, each standard amino acid residue receives its own category, and all non-standard residues included in the analysis are labeled as 'NP' for non-protein (processed internally as '#'). One can specify a group of standard amino-acid residue types to be given the label 'X' (which is standard notation for unknown residue type), which enables these residues to be substituted for another in subgraph patterns. This is done by passing a list of 1-letter amino acid characters as the `sub` keyword argument to *generate_graph_database()*.

Example

Set isoleucine and leucine to be given the label 'X' in subgraph patterns.

```
pg.generate_graph_database(sub=['I','L'])
```

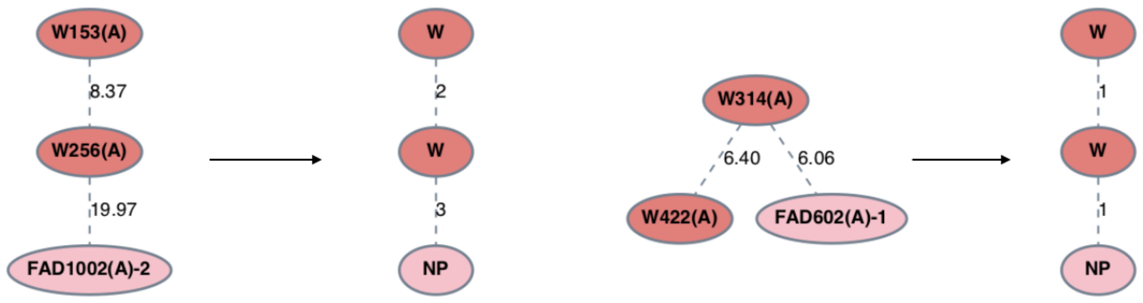


Edges

By default, all edges are assigned the same numerical label of 1. One can classify edges based on their weights by passing the `edge_thresholds` argument to *generate_graph_database()*. `edge_thresholds` should be formatted as a list of floats in ascending order, where each value indicates a cutoff threshold for an edge category.

Example

```
pg.generate_graph_database(edge_thresholds=[8,12])
```



Mining

Introduction

The goal of subgraph mining is to identify graph structures which occur a significant number of times across a set of graphs. In the context of PyeMap, this means searching for shared pathways/motifs amongst a set of protein crystal structures. In PyeMap, users can either search for all patterns which appear in a specified number of PDBs (General Pattern Mining), or search for a specified group of patterns which match a string representation of the graph of interest (Specific Pattern Mining).

General Pattern Mining

In the graph mining literature, the frequency that a pattern appears in the set of input graphs is referred to as the *support* for that pattern. In other words, if a pattern appears in 12/14 graphs, one would say it supports 12 graphs, or equivalently, has a support of 12 (regardless of whether it appears multiple times within a given graph).

In PyeMap, we use a Python implementation of the gSpan algorithm [Han2002], one of the most efficient and popular approaches for graph mining. The technical details are beyond the scope of this documentation (and can be found in the gSpan paper), but here we emphasize a few of its key features.

- gSpan is a recursive algorithm which relies on the use of minimum depth first search (DFS) codes
- gSpan is a complete algorithm, so it will find all patterns which meet a specified support threshold
- The perfomance of gSpan is greatly sped up by increasing the minimum support threshold, as this allows more aggressive pruning of candidate subgraphs

In addition to the support number, one can also specify a minimum or maximum number of vertices for the identified subgraph patterns. For a full list of accepted kwargs, please refer to the gspan_mining package's repository.

To perform general pattern mining with PyeMap, use the *run_gspan()* function.

Example

Search for all subgraph patterns comprised of 4-6 nodes with a support number of 10 or greater using PyeMap.

```
pg.run_gspan(10,min_num_vertices=4,max_num_vertices=6)
```

Specific Pattern Mining

Instead of searching for all subgraphs up to a given support threshold, one instead may be interested in finding protein subgraphs which match a previously known pattern. In this case, the problem is reduced to one of graph matching, and we simply search each PDB for a monomorphism using the NetworkX implementation of the VF2 algorithm (see the NetworkX documentation for more details).

In PyeMap, this is done for protein graphs using the *find_subgraph()* function. *find_subgraph()* accepts a string representation of a subgraph, where each character is one of the following:

- 1-character amino acid code for a standard residue
- X' for unknown amino acid types
- # for non-protein residue
- * as a wildcard character

Branching can be specified using a syntax similar to the SMILES format, where there is no specification of bonding and each amino acid or special character described above must be separated by brackets (see example below). See the `write_graph_smiles()` function, and the pysmiles repository for more details.

If edge thresholds are used (see the classification section), the search will be performed for all possible combinations of edges, and thus several subgraph patterns will be found for a set of residue types. If the * wildcard character is used, subgraph pattern(s) will be found for each combination of each residue type replacing the * placeholder character(s), including the special X and # residue types.

Examples

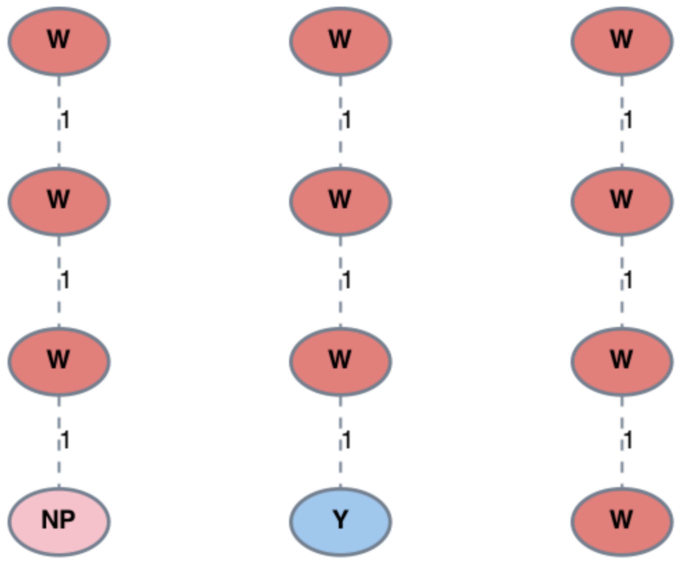


Fig. 2.1: Examples of subgraph patterns identified using string 'WWW*'.

Subgraph Patterns

The end result of either mining option is a set of *subgraph patterns*, each of which has a unique ID with the format:

{Index}_{String representation}_{support number}

e.g. 1_WWW_18.

The string representation for each pattern is a pseudo-SMILES string generated using the pysmiles package. Importantly, these strings can be used as inputs for `find_subgraph()`, as they correctly encode the structure of the graph using a syntax similar to the SMILES format.

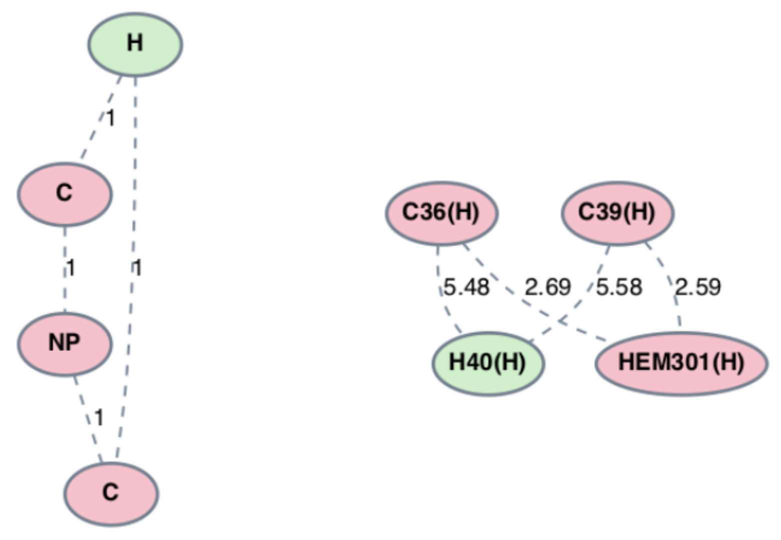


Fig. 2.2: Subgraph pattern (left) and protein subgraph (right) identified using the string '[H]1[C][#][C]1'.

Source

pyemap.graph_mining.write_graph_smiles(...)	Returns pseudo-SMILES string for supplied graph.

pyemap.graph_mining.write_graph_smiles

pyemap.graph_mining.**write_graph_smiles**(*generic_subgraph*)

Returns pseudo-SMILES string for supplied graph.

Parameters

generic_subgraph (networkx.Graph) – Graph to be transformed into string representation. The `label` attribute of each node should be set to the 1-letter amino acid code or special character.

Returns

pseudosmiles – String representation of graph of interest

Return type

str

Matching and Clustering

Introduction

The graph mining algorithms in the previous section tell us which PDBs a subgraph pattern is present in, but they do not tell us which specific residues are involved. Additionally, two subgraphs belonging to the same pattern can have different orientations or correspond to different residues in the crystal structure, and thus may not be related to one another in a meaningful way. In PyeMap, we cluster subgraphs based on their simiarlity in order to identify shared pathways/motifs.

Graph Matching

In order to identify the specific residues involved in subgraph patterns, we utitlize the NetworkX implementation of the VF2 algorithm for graph matching. We refer the reader to their documentation for more details, but for the sake of clarity, we re-iterate some key defintions.

Let G=(N,E) be a graph with a set of nodes N and set of edges E.

If G'=(N',E') is a subgraph of G, then:
N' is a subset of N,

E' is a subset of E

If G'=(N',E') is isomorphic to G, then:
there exists a one-to-one mapping between N and N',

there exists a one-to-one mapping between E and E'

If G'=(N',E') is a monomorphism of G, then:
N' is a subset of N,

E' is a subset of the set of edges in E relating nodes in N'

In PyeMap, for each PDB which supports the given subgraph pattern, we search for all subgraph monomorphisms of the protein graph which are isomorphic to the subgraph pattern. This gives us a set of *protein subgraphs*, which we can then cluster into groups based on similarity.

Each protein subgraph for a given subgraph pattern is assigned a unique ID with the format:

{PDB ID}-{Unique Index}

e.g. 1u3d-2.

Clustering

PyeMap currenly enables two types of clustering: **structural** similarity, and **sequence** simiarlity, which both use the same underlying algorithm below.

Algorithm

For a given subgraph pattern P, we have a set of protein subgraphs V which correspond to groups of specific residues in PDB structures which match pattern P. We then construct a supergraph G(V,E), where two protein subgraphs share an edge if and only if they are deemed sufficiently similar by the chosen metric. The resulting supergraph G will be composed of one or multiple connected components, and each connected component corresponds to a cluster of similar protein subgraphs.

Structural Similarity

The structural similarity between two protein subgraphs in PyeMap is computed by superimposing the two sets of atoms and computing the root mean squared distance (RMSD) using the `Bio.PDB.Superimposer` module in BioPython. However, atoms can sometimes be missing from crystal structures, and we would also like some flexibility to allow for substitutions. Starting from a one-to-one mapping between the residues, we make the following approximations to the true RMSD:

- Only the alpha carbon (CA) is considered for standard amino acid residues. If it is not present in the crystal structure, ∞ is returned.
- For non-standard amino acids, we use the first atom type both residues have in common. If no shared atom type is found, ∞ is returned.

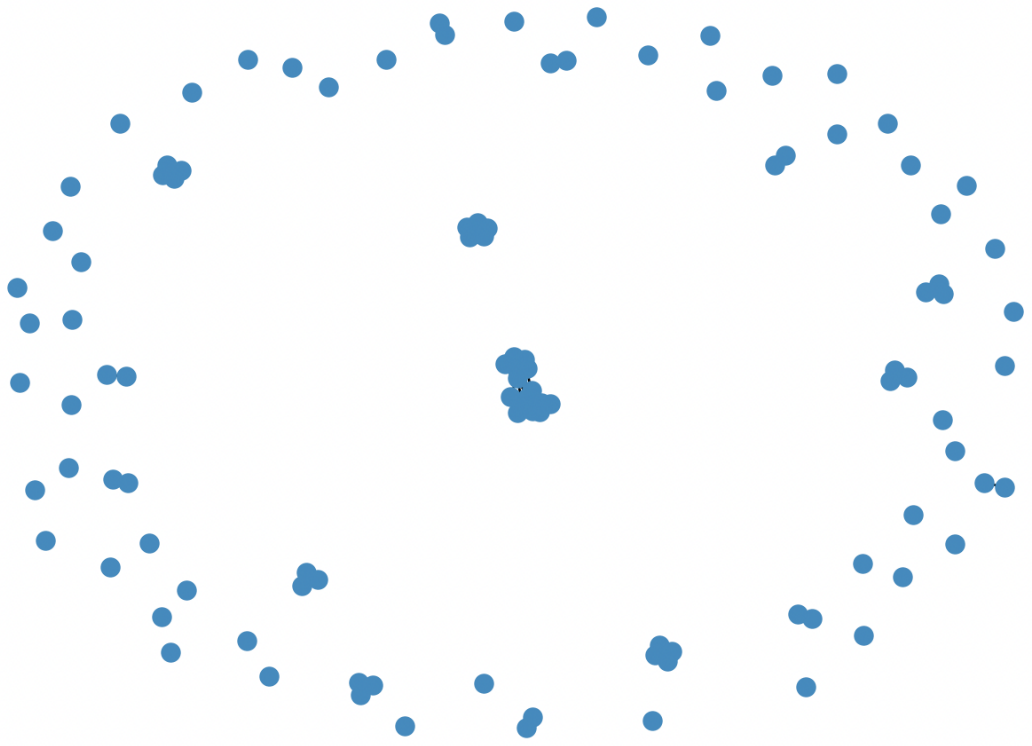


Fig. 2.3: Example supergraph G, where each node is a protein subgraph belonging to a subgraph pattern P. In this example, the supergraph G is composed of 66 connected components, and therefore, 66 clusters.

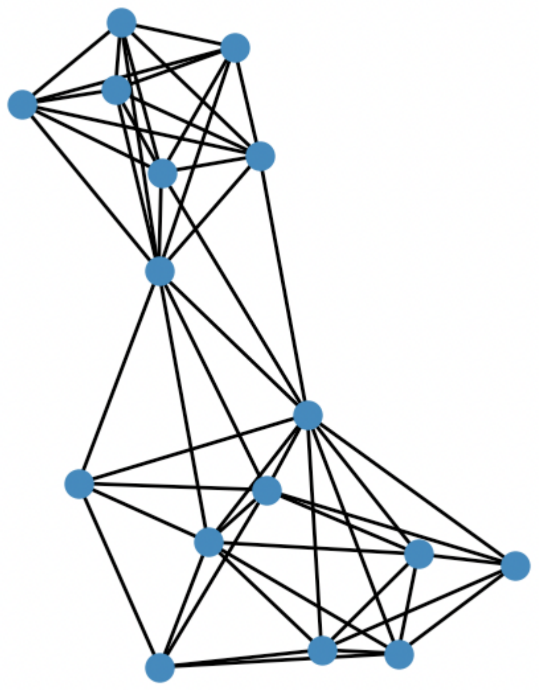


Fig. 2.4: The largest cluster in Fig. 3 zoomed in.

An edge is drawn between two protein subgraphs in the supergraph if RMSD ≤ 0.5 Å.

Sequence Similarity

Sequence simiarlity in PyeMap relies on a multiple sequence alignment, which will automatically be performed by the MUSCLE package [Edgar2004] if it is installed on your machine. Starting from a one-to-one mapping between the residues, the sequence similarity between two protein subgraphs is simply defined as the sum of the differences in the residue numbers with respect to the multiple sequence alignment. For instance, TRP50 in one PDB and TRP200 in another PDB could have a difference of 0 if they are aligned by the multiple sequence alignment. One important caveat is that **non-protein residues are not considered in sequence similarity**, only standard amino acid residues.

An edge is drawn between two protein subgraphs in the supergraph if $D \leq N$, where D is the sum of the differences in the residue numbers with respect to the multiple sequence alignment, and N is the total number of nodes comprising the subgraph pattern, which allows for slight misalignments.

Note:

If MUSCLE is not installed, the original residue numbers will be used, which is unlikely to lead to a meaningful clustering.

Source

pyemap.graph_mining.SubgraphPattern._do_clustering(...)	Compute the supergraphs and find the connected components
pyemap.graph_mining.SubgraphPattern._subgraph_rmsd(...)	Computes RMSD between two protein subgraphs
pyemap.graph_mining.SubgraphPattern._subgraph_seq_dist(...)	Computes sequence distance between two protein subgraphs

pyemap.graph_mining.SubgraphPattern._do_clustering

SubgraphPattern.**_do_clustering**(*all_graphs*)

Compute the supergraphs and find the connected components

pyemap.graph_mining.SubgraphPattern._subgraph_rmsd

SubgraphPattern.**_subgraph_rmsd**(*sg1*, *sg2*, *mapping*)

Computes RMSD between two protein subgraphs

pyemap.graph_mining.SubgraphPattern._subgraph_seq_dist

SubgraphPattern.**_subgraph_seq_dist**(*sg1*, *sg2*, *mapping*)

Computes sequence distance between two protein subgraphs

2.5 Bibliography

2.6 How to cite

2.6.1 Primary citation

R.N. Tazhigulov, J.R. Gayvert, M. Wei, and K.B. Bravaya, eMap: A Web Application for Identifying and Visualizing Electron or Hole Hopping Pathways in Proteins. *J Phys. Chem. B* **2019**, 123, 32, 6946-6951. https://doi.org/10.1021/acs.jpcb.9b04816

2.6.2 Third Party Software

In addition to the primary citation, please cite the relevant third party software we depend on:

Residue Depth: MSMS: Sanner, M. F.; Olson, A. J.; Spehner, J. C. *Biopolymers*, **1996**, 38, 305-320. RD Reference: Chakravarty, S.; Varadarajan, R. *Structure*, **1999**, 7, 723-732. Threshold: Song, J.; Tan, H.; Mahmood, K.; Law, R. H. P.; Buckle, A. M.; Webb, G. I.; Akutsu, T.; Whisstock, J. C. *PLoS ONE*, **2009**, 4, 1-14.

Solvent Accessibility: DSSP: Touw, W. G.; Baakman, C.; Black, J.; te Beek, T. A.; Krieger, E.; Joosten, R. P.; Vriend, G. *Nucleic Acids Res.*, **2015**, 43, D364-D368. Wolfgang, K.; Christian, S. *Biopolymers*, **1983**, 22, 2577-2637. Threshold: Tien, M. Z.; Meyer, A. G.; Sydykova, D. K.; Spielman, S. J.; Wilke, C. O. *PLoS ONE*, **2013**, 8, 1-8.

Multiple Sequence Alignment MUSCLE: Edgar RC. *Nucleic Acids Res.* **2004** 32(5) 1792-1797.

Graph Mining gSpan: X. Yan and J. Han. *Proc. Int'l Conf. Data Mining*, **2002**.

Biopython: Application note: Cock, P. J. A.; Antao, T.; Chang, J. T.; Chapman, B. A.; Cox, C. J.; Dalke, A.; Friedberg, I.; Hamelryck, T.; Kauff, F.; Wilczynski, B.; de Hoon, M. J. L. *Bioinformatics*, **2009**, 25, 1422-1423. PDB Parser: Hamelryck, T.; Manderick, B. *Bioinformatics*, **2003**, 19, 2308-2310.

NetworkX: Hagberg, A. A.; Schult, D. A.; Swart, P. J. *Exploring Network Structure, Dynamics, and Function using NetworkX*. Pasadena, CA USA, **2008**.

2.6.3 Other Supporting Software

RDKit , NumPy , SciPy , PyGraphviz , Graphviz, pysmiles, gspan-mining

2.7 Credits

2.7.1 Original Authors

- Ruslan Tazhigulov
- James Gayvert
- Ksenia Bravaya

2.7.2 Other Contributors

- Melissa Wei
- Alyssa Kranc

2.7.3 Funding

The web version eMap was supported by the Molecular Sciences Software Institute under National Science Foundation grant ACI-1547580, and Boston University Hariri Institute for Computational Science and Engineering (2016-10-003).

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Beratan1992] Beratan, D.; Onuchic, J.; Winkler, J.; Gray, H. *Science*, **1992**, 258, 1740–1741.

[Chakravarty1999] Chakravarty, S.; Varadarajan, R. *Structure*, **1999**, 7, 723–732.

[Sanner1996] Sanner, M. F.; Olson, A. J.; Spehner, J. C. *Biopolymers*, **1996**, 38, 305–320.

[Lee1971] Lee, B.; Richards, F. M. *J . Mol. Biol.*, **1971**, 55, 379–400.

[Shrake1973] Shrake, A.; Rupley, J. A. *J . Mol. Biol.*, **1973**, 79, 351–371.

[Tan2009] Song, J.; Tan, H.; Mahmood, K.; Law, R. H. P.; Buckle, A. M.; Webb, G. I.; Akutsu, T.; Whisstock, J. C. *PLoS ONE*, **2009**, 4, 1–14.

[Tien2013] Tien, M. Z.; Meyer, A. G.; Sydykova, D. K.; Spielman, S. J.; Wilke, C. O. *PLoS ONE*, **2013**, 8, 1–8.

[Sander1983] Kabsch, W.; Sander, C. *Biopolymers*, **1983**, 22, 2577–2637.

[Han2002] X. Yan and J. Han. *Proc. Int'l Conf. Data Mining*, **2002**.

[Edgar2004] Edgar RC. *Nucleic Acids Res.* **2004** 32(5) 1792-1797.

PYTHON MODULE INDEX

INDEX