E-state index

1 Introduction

In [2] the authors present different approaches to predict the energy gap of porphyrins. They discuss four different methods to encode the chemical structure in a mathematical form. Then, for each method, four different algorithms are used to predict the energy gap. Here we focus on the method to encode porphyrins that has the best results, namely the E-state index.

In the first chapter, we explain the theory of the E-State index. In the second chapter, we discuss how to model molecules as graphs in Python and present an algorithm that uses this representation to calculate the E-state index. Finally, we discuss an algorithm that uses the structure of porphyrins to generate the graph representation of porphyrins.

2 Electrotopological state index

In the following we introduce the Electrotopological state index (E-state index) for atoms and explain how to calculate it using the molecule ethyl acetate as an example. The example and explanation is based on the presentation of the E-state index in Chapter 11 of [1].

In a molecule, the electrotopological state index attributes a value S_i to each atom x_i except hydrogen. This value is calculated from the intrinsic state value I_i and the perturbation ΔI_i. We first explain how we mathematically model a molecule as a graph, then how we derive the two values I_i and ΔI_i, and finally the E-state index of the atom x_i is simply given by $S_i = I_i + \Delta I_i$.

We interpret a molecule as a graph, where the nodes represent the atoms and the edges the bonds between the atoms. In each node and edge we save the type of atom and bound as an attribute, respectively. Note that we suppress the hydrogen in the molecule and take them with the atoms they are bonded to.

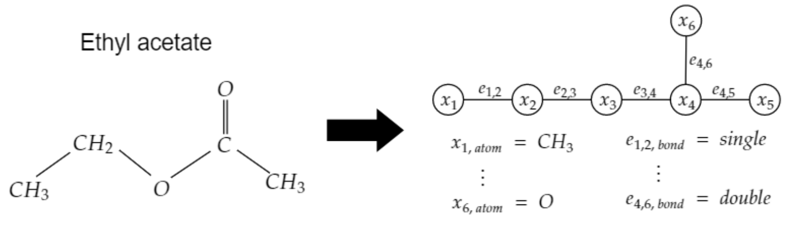


Figure 1: *Graph representation of ethyl acetate.*

To calculate the intrinsic state value, we must first determine the delta value δ, the valence delta value δ^v, and the principal quantum number N of each atom:

- The delta value δ is the count of adjacent atoms other than hydrogen.
- The valence delta value δ^v is the number of valence electrons of the atoms minus those bonding hydrogen.
- As for the principal quantum number N, I'm still not quite sure what it is (Lara didn't know much more). For the moment, we treat it as if it were the number of shells corresponding to the row in the periodic table.

To illustrate the different values we look again at ethyl acetate from Figure 3. The carbon C has 4 valence electrons. In the molecule we have 4 appearances of C, at positions x_1, x_2, x_4 and x_5. At the positions x_1 and x_5 the carbon axiom is bond to 3 hydrogen, this implies δ^v is equal to 1. Similarly δ^v for x_2 and x_4 is 2 and 4, respectively. The delta value δ can be easily determined by counting the number of adjacent nodes/atoms in the graph.

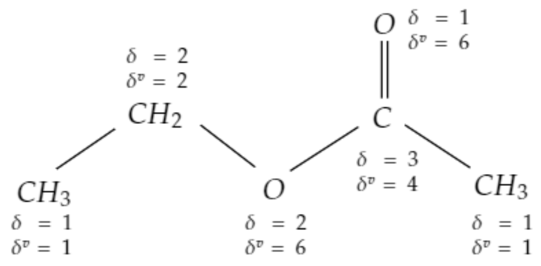


Figure 2: *The delta values of the atoms in ethyl acetate.*

The principal quantum number N does not depend on the graph. In the case of ethyl acetate, the carbon C and the oxygen O both have $N = 2$. Next, the intrinsic state value I_i of the atom at position i is given by

$$I_i = ((2/N_i)\delta_i^v + 1)/\delta_i.$$

For example, the intrinsic state value of the oxygen atom in ethyl acetate at position x_3 is $I_3 = ((2/2)6 + 1)/2 = 3.5$. Once the intrinsic state value I_i of each atom in the graph is determined, we can compute the perturbation ΔI_i. For the atom at position i the perturbation is given by

$$\Delta I_i = \sum_{j \neq i} (I_i - I_j)/r_{i,j}^2,$$

where $r_{i,j}$ is the graph distance distance $d_{i,j}$ between the atoms at position i and j plus one, that is, $r_{i,j} = d_{i,j} + 1$. For example, regarding ethyl acetate we have $r_{1,2} = 2$, $r_{1,5} = 5$, $r_{1,6} = 5$ etc. Finally. the E-state value S_i of the atom at position i is given by

$$S_i = I_i + \Delta I_i.$$

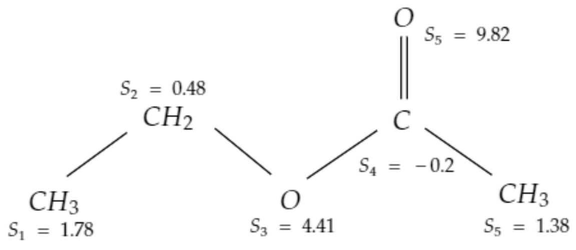


Figure 3: *The calculate E-state values for ethyl acetate.*

3 Computation

We use the *NetworkX* package to work with graphs in Python. See click me! for a tutorial. We first explain how to write a graph representing a molecule, then the function that calculates the E-state index of a graph.

3.1 Graphs

To illustrate how we represent the graph of a molecule in Python with this package, we again use ethyl acetate as an example.

First we have to initialize the graph and then we can add the nodes, six in this case. Note that the nodes do not have to be numbered, but can also be named with strings.

Listing 1: *Initialize the graph and add the nodes*

```
G = nx.Graph()  #initialize the graph

nodes_G= [1,2,3,4,5,6] #define the nodes
G.add_nodes_from(nodes_G) #add the nodes
```

In the next step we add the edges between the nodes. For two nodes x_i and x_2j, the edge between them is written as (x_i, x_j).

Listing 2: *Add the edges*

```
edges_G = [(1,2),(2,3),(2,4),(4,5),(5,6)]  #define the edges
H.add_edges_from(edges_G)  #add the edges
```

Once the structure of the graph is defined, we can add attributes to the nodes and edges. To complete the graph representation of a molecule, we need to add to each node the type of atom and the number of hydrogen atoms bonded to the atom, and for the edges we need to specify what type of bond it is (a = aromatic, s = singular, b = double and t = triangular). To access node i or edge (i, j) of a graph G, we write *G.node[i]* or *G.edges[i,j]*, respectively, then we can add an attribute with *G.nodes[i]['attribute_name'] = attribute* and *G.edges[i,j]['attribute_name'] = attribute*, respectively. We show only the attributes added to the first nodes and edges. Note that the attributes *atom* and *bond_type* are strings and the number of hydrogen bonds *nb_H* is a number.

Listing 3: *Add attributes*

```
#attributes of the nodes
G.nodes[1]['atom']="C"
G.nodes[1]['nb_H']=3

G.nodes[2]['atom']="C"
G.nodes[2]['nb_H']=2
....

#attributes of the edges
G.edges[1,2]['bond_type']="s"
....
G.edges[4,5]['bond_type']="d"
```

3.2 E-state calculator

The next goal is to define a function which for a given graph representation of a molecule computes the E-state index of the atoms in the molecule. First, note that the information needed to calculate the E-state index is not included in the graph representation of a molecule, which was explained before. The delta value δ of a node x_i can be easily determined by counting the number of neighboring nodes, this is done with *G.degree[i]*. To obtain the number of valence electrons and the principal quantum number, we define a dictionary (one of the 4 built-in data types in Python used to store collections of data) that holds this information for each atom.

Listing 4: *The periodic table as dictionary in Python*

```
# 1.entry = number of valence electrons, 2.entry = principal quantum number
periodic_table = {"H": [1, 1], "Li": [1, 2], "Be": [2, 2], "B": [3, 2],
                  "C": [4, 2], "N": [5, 2], "O": [6, 2], "F": [7, 2],
                  "Na": [1, 3], "Mg": [2, 3], "Al": [3, 3], "Si": [4, 3],
                  "P": [5, 3], "S": [6, 3], "Cl": [7, 3]}
```

Of course, this periodic table does not contain all information, but only the part that is important for our calculations. To access the information of a certain atom, for example C, we write *periodic_table["C"]=[4,2]*. Hence, if we want to access the number of valence electrons of the node i we write

periodic_table[[G.nodes]['atom']][0].

We now begin with the definition of the function. In the first step, we add to each atom the attributes necessary to compute the intrinsic state index I_i, and then directly compute the intrinsic state index.

Listing 5: *E-state calculator: intrinsic state index*

```
def E_state_calculator(G):   #start with the definition of the function,
                             #the input G is a graph

    for i in G.nodes:
        #add delta value
        G.nodes[i]['delta']= G.degree[i]

        #add delta^v
        G.nodes[i]['delta_v']=
             periodic_table[G.nodes[i]['atom']][0]-G.nodes[i]['nb_H']

        #add principal quantum number
        G.nodes[i]['N']= periodic_table[G.nodes[i]['atom']][1]

        #compute the intrinsic state index
        G.nodes[i]['I'] = ((2/G.nodes[i]['N'])**2 * G.nodes[i]['delta_v']+1)
                            /G.nodes[i]['delta']
```

In the next step, we compute the perturbation ΔI_i of each node. For each node i, we need to access every other node j and find the graph distance $d_{i,j}$ between these nodes. Then we compute

$$\Delta I_{i,j} = (I_i - I_j)/r_{i,j}^2,$$

where $r_{i,j} = d_{i,j} + 1$. Next, we calculate the perturbation

$$\Delta I_i = \sum_j \Delta I_{i,j}.$$

This is done in the following way. For each node, we use an empty list *delta_I_temp*, subsequently compute the values $\Delta I_{i,j}$ and add them to the list. Then we compute ΔI_i, which is and the E-state index S_i.

Listing 6: *E-state calculator: perturbation*

```
    for i in G.nodes:
        delta_I_temp = []   #reset the empty list

        for j in G.nodes:   #access the other nodes
            if j != i:
                #find the graph distance and add 1
                r_temp=nx.shortest_path_length(G, source=i, target=j)+1

                #compute \Delta I_{i,j} and add it to the list
                delta_I_temp.append(
                    (G.nodes[i]['I']-G.nodes[j]['I'])/(r_temp**2))

        #compute the perturbation: sum of all values in delta_I_temp
        G.nodes[i]['perturbation']=np.sum(delta_I_temp)

        #compute the E-state index: S_i = I_i + \Delta I_i
        G.nodes[i]['S']=G.nodes[i]['I']+G.nodes[i]['perturbation']
```

Before we turn to the last part of the function, we must explain how the numerical representation of the porphyrins as vectors (from now on called data) is constructed. Essentially, the data is a list of possible atoms along with the bond. For example, for carbon C there are the following possibilities: sCH3, dCH2, ssCH2, tCH, dsCH, aaCH, sssCH, ddC, tsC, dssC, aasC, aaaC, ssssC. Each string represents one way a carbon atom can be part of a molecule. The lowercase letters at the beginning indicate the bonds that do not involve hydrogen, and the number of hydrogen bonds is indicated at the end. For example, dsCH stands for a carbon atom with one double bond and one single bond in the graph and one hydrogen bond. For ethyl acetate we have $x_1 = sCH3, x_2 = ssCH2, x_3 = ssO, x_4 = dssC, x_5 = sCH3, x_6 = dO$. Then, for each entry in the data, we sum the E-state values of each node corresponding to that entry.

To generate the data with the function we first define a dictionary containing all the entries from the data. At the beginning the value is set to zero.

Listing 7: *E-state calculator: generate data*

```
    descriptor={
            "sLi":[0],
            "ssBe":[0],
            "ssssBe":[0],
            ...
            "ssssPb":[0],
            }
```

Next, for each node we need to find the corresponding entry. To do so we define a new attribute for each node: *G.nodes[i]['descriptor']*. Note that the entries in the dictionary *descriptor* are strings, so this attribute must also be a string. First, we add the name of the atom. Then we access each adjacent node with $G.adj$ and add the letter corresponding to the bond between the atoms. Note that we need to add the bond type in the correct order $a \to t \to d \to s$, so we need to add each bond type separately. Finally, we add an H if there is an hydrogen bonded to the atom, and if there are more than one, we also add the number.

Listing 8: *E-state calculator: find the entry names of the nodes*

```
for i in G.nodes:
# add the name of the atom
    G.nodes[i]['descriptor']= G.nodes[i]['atom']

# add "s" for each singular bond
    for j in G.adj[i]:
        if G.edges[i,j]['bond_type']=="s":
            G.nodes[i]['descriptor']="s"+ G.nodes[i]['descriptor']

# add "d" for each double bond
    for j in G.adj[i]:
        if G.edges[i,j]['bond_type']=="d":
            G.nodes[i]['descriptor']="d"+ G.nodes[i]['descriptor']

# add "t" for each triangular bond
    for j in G.adj[i]:
        if G.edges[i,j]['bond_type']=="t":
            G.nodes[i]['descriptor']="t"+ G.nodes[i]['descriptor']

# add "a" for each aromatic bond
    for j in G.adj[i]:
        if G.edges[i,j]['bond_type']=="a":
            G.nodes[i]['descriptor']="a"+ G.nodes[i]['descriptor']

# add "H" if there is an hydrogen bond
    if G.nodes[i]['nb_H']>0:
        G.nodes[i]['descriptor'] = G.nodes[i]['descriptor']+ "H"

        # add the number of hydrogen bonds (if more than 1)
        if G.nodes[i]['nb_H']>1:
            G.nodes[i]['descriptor'] =
                G.nodes[i]['descriptor']+str(G.nodes[i]['nb_H'])
```

Lastly, we generate the data and finish the definition of the function. The function will output the updated graph and data.

Listing 9: *E-state calculator: output*

```
# generate data
for i in G.nodes:
    descriptor[G.nodes[i]['descriptor']][0]+=G.nodes[i]['S']

# define the output
return [G, descriptor]
```

4 Graph creator

All porphyrins have the same core structure, which we refer to here as the core system. This core system has four points, $A, R1, R2, R4$, where additional features can be attached. In addition, a metal can be attached to the center of the core system. However, the connection between the metal and the core system is complex in nature and is therefore omitted from the graph representation of the porphyrin and considered separately in the data.

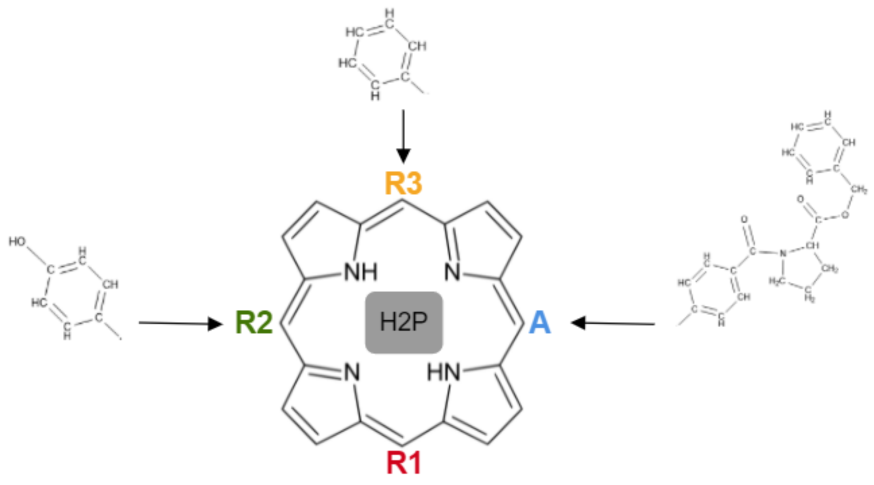


Figure 4: *The structure of porphyrins with H2P as central metal.*

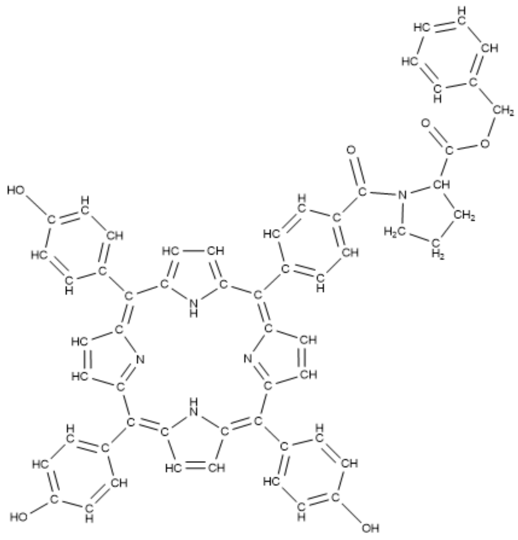
The structure of porphyrins allows to characterize each porphyrin by the attached features and the metal in the center.
The next goal is to define a function that takes the features as input and returns as output the graph representing the porphyrin with these features. For this purpose, we first define the graph of the core system and the different features separately. The nodes of the graph of the kernel system are called numerically $(1, 2, 3, \ldots)$, except for those to which we attach the features, they are called A, $R1$, $R2$, $R3$.

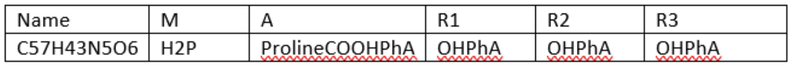
Listing 10: *The graph of the core system*

```
G = nx.Graph()

nodes=[1,2,3,4,5,"R3",7,8,9,10,11,"A",13,14,15,16,17,"R1",19,20,21,22,23,"R2"]
G.add_nodes_from(nodes)
...
```

For the graphs of the features, we append the name of the feature to the node number, and the node that is connected to the core system is called *nameconnecter*. In addition we add the name of the feature to each graph *feature.graph['name'] = "feature"* and the type of the bond the feature will have to the core system *feature.nodes["featureconnecter"]['connecter']="bond_type"*.





Name	M	A	R1	R2	R3
C57H43N5O6	H2P	ProlineCOOHPhA	OHPhA	OHPhA	OHPhA

Figure 5: *The characterization of a porphyrin.*

Listing 11: *The graph of the feature Ph*

```
Ph = nx.Graph()
Ph.graph['name'] = "Ph"

Ph_nodes = ["Phconnecter","Ph2","Ph3","Ph4","Ph5","Ph6"]
Ph.add_nodes_from(Ph_nodes, atom="C")

Ph.nodes["Phconnecter"]['connecter']="a"
....
```

Once the graphs of the core system and the possible features are defined we can start with the definition of the function. The inputs $A, R1, R2, R3$ correspond to the graphs of the features at this position. In the first step we subsequently add the graphs of the features to the graph of the core system. To do this, we use the function *union*, and each time we add a graph of a feature, we add the name of the feature's position to the names of the new nodes, leaving the old nodes the same. This is done in the second part *rename()* of the *union* function. Note that at this point the graphs are not connected.

Listing 12: *Porphyrin creator: get the graphs*

```
def porphyrin_creator(A,R1,R2,R3):
    # load the graph of the feature at position A
    Porphyrin_G = nx.union(G, A, rename=('', 'A-'))

    # load the graph of the feature at position R1
    Porphyrin_G = nx.union(Porphyrin_G, R1, rename=('', 'R1-'))

    # load the graph of the feature at position R2
    Porphyrin_G = nx.union(Porphyrin_G, R2, rename=('', 'R2-'))

    # load the graph of the feature at position R3
    Porphyrin_G = nx.union(Porphyrin_G, R3, rename=('', 'R3-'))
```

Next, we need to generate the names of the nodes of the features that we attach to the core system. These names consist of three parts: The first part is the position (this was added in the last step), the second part is the feature name (stored as *'name'* of the graph), and the third part is *'connecter'*, as explained earlier. Once we have the names of these nodes, we can create the edges that connect them, and thus the features, to the core system.

Listing 13: *Porphyrin creator: connect the features*

```
    # get names of the nodes that connect the core system
    A_connecter_name = 'A-' + A.graph['name'] + "connecter"
    R1_connecter_name = 'R1-' + R1.graph['name'] + "connecter"
    R2_connecter_name = 'R2-' + R2.graph['name'] + "connecter"
    R3_connecter_name = 'R3-' + R3.graph['name'] + "connecter"


    # make edges between core system and A, R1, R2, R3
    edges_connections=[("A",A_connecter_name),("R1",R1_connecter_name),
                       ("R2",R2_connecter_name), ("R3",R3_connecter_name)]
    Porphyrin_G.add_edges_from(edges_connections)
```

In the last step, we need to add the bond type of the edges between the core system and the features. Also, we need to set the number of hydrogen bonds of the nodes in the core system to which we attach the features from 1 to 0. This is because this bond will be replaced by the bond to the feature. Recall that the bond type is stored in *feature.nodes["featureconnecter"]['connecter']="bond_type"*.

Listing 14: *Porphyrin creator: connect the features*

```
    # specify bond type of the edges between core and A, R1, R2, R3
        Porphyrin_G.nodes["A"]['nb_H'] = 0
        Porphyrin_G.edges["A",A_connecter_name]['bond_type'] =
                Porphyrin_G.nodes[A_connecter_name]['connecter']

        Porphyrin_G.nodes["R1"]['nb_H'] = 0
        Porphyrin_G.edges["R1",R1_connecter_name]['bond_type'] =
                Porphyrin_G.nodes[R1_connecter_name]['connecter']

        Porphyrin_G.nodes["R2"]['nb_H'] = 0
        Porphyrin_G.edges["R2",R2_connecter_name]['bond_type'] =
                Porphyrin_G.nodes[R2_connecter_name]['connecter']

        Porphyrin_G.nodes["R3"]['nb_H'] = 0
        Porphyrin_G.edges["R3",R3_connecter_name]['bond_type'] =
                Porphyrin_G.nodes[R3_connecter_name]['connecter']


    return Porphyrin_G
```

References

[1] Alexandru T Balaban James Devillers. *Topological Indices and Related Descriptors in QSAR and QSPR*. CRC Press, 2000.

[2] Zheng Li et al. “Machine-learning energy gaps of porphyrins with molecular graph representations”. In: *The Journal of Physical Chemistry A* 122.18 (2018), pp. 4571–4578.