

Fragmentation of PDB compounds

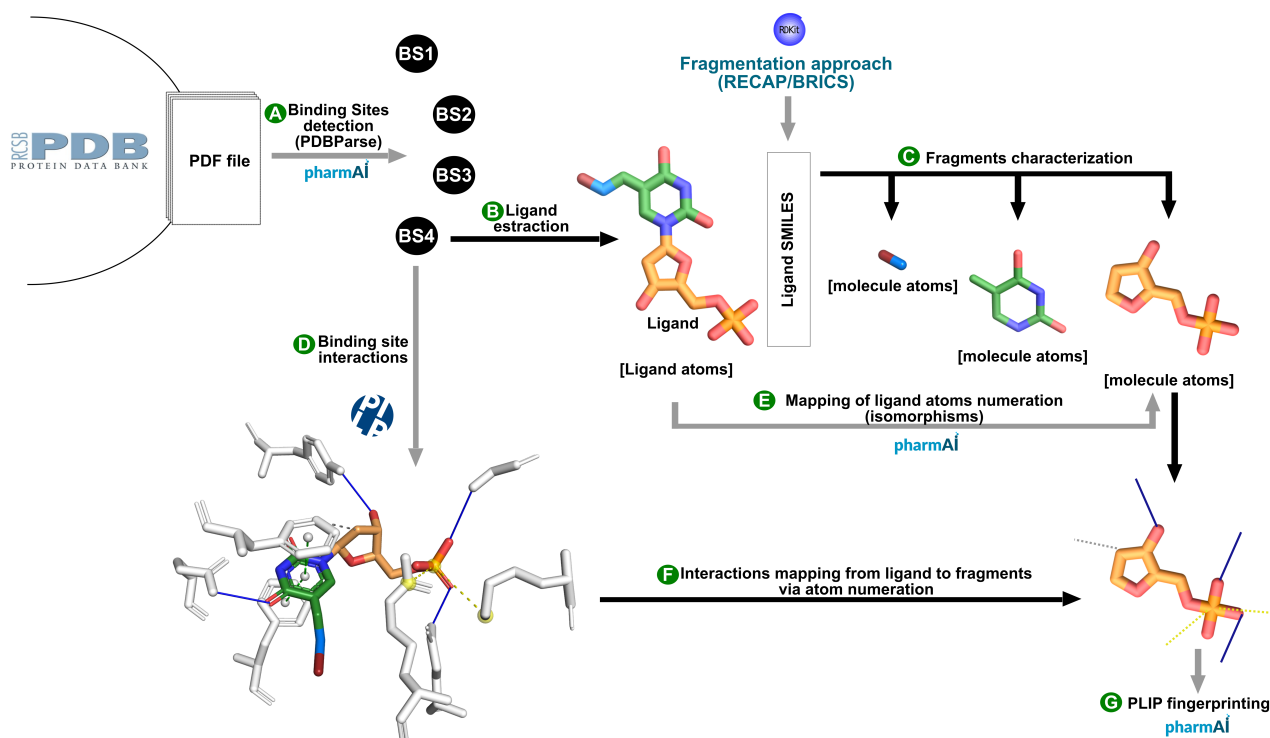


Figure 1: Fragmentation workflow used in the test case. The fragmentation applied to the test case, incorporates multiple methods/services from other sources. Such steps could be easily replaced by the approaches that fit better the user's preferences

The fragmentation process follows the following steps:

- A The input of the process is a PDB file containing the 3D coordinates of protein-ligand complexes. The file must be properly parsed to recognize all different binding sites.
- B The ligand must be extracted from each binding site and is possible characterized in terms of chemical properties (this will be of great help to further analysis). This can be easily done by any cheminformatics tool such as RdKit or OpenBabel. The most important feature to obtain is the compound SMILES representation, as this will serve as input for the following fragmentation step.
- C There are several fragmentation approaches depending on the user's intentions. For this test case, it was used RdKit implementation of the RECAP and BRICS algorithm. Given the compound SMILES, such methods split the molecules according to predefined cleavage rules and obtain the corresponding fragments.
- D PLIP was used to characterize the non-covalent interactions defining the binding mode of PDB compounds. However, it is worth to mention that any other type of representation can be used depending on the user's preferences.
- E The fragments atoms (from the molecules obtained in step C) should be correctly mapped back to the ligand molecule in order to obtain the right binding site information coming from the PDB file. For this test case we have used a predefined method provided by pharmAI, however any other implemented in RdKit, OpenBabel, or any other cheminformatics tool can be used as well.
- F The interactions obtained with PLIP are then mapped from ligand to fragment guided by the atom mapping previously done in step E.

G Finally, the binding mode is encoded into a PLIP binary fingerprint. This step could also be easily replaced to use the desired fingerprinting representation.

1 Example pseudocode

The following python-based pseudocode underlines the most important steps on the approach for the fragmentation of the PDB compounds. There are three main classes defining the levels of molecular entities among the fragmentation steps: PDB, Ligand, and Fragment. The input is a PDB file that should be parsed to extract all binding ligands as mol objects. In the presented work, the PDB parsing was done using the PDBParser class provided by PharmAI GmbH as a service.

```
1 """Defines a PDB file object and detects the binding ligands"""
2 class PDB(filename):
3     pdbid = filename.split('.')[0] #extracts the PDB id from filename
4     pdbparsed = PDBParser(filename) #instance of a class to parse the PDB file
5     ligands = pdbparsed.ligands #all the PDB ligands as mol objects
```

For each ligand in the PDB, the PLIP interactions are extracted and the fragmentation process is carried out. Different fragmentation algorithms are implemented according to the type of fragmentation applied to obtain the molecular fragments. In this work, we have considered RECAP and BRICS algorithms as RdKit implementations, both offering a full tree decomposition or just the main tree leaves.

```
1 """Defines a Ligand class to extract the molecule data"""
2 class Ligand(lig, plipxml, ftype):
3     #Inputs are: ligand mol object parsed from PDB file, a plipxml object containing all PLIP recognized interactions,
4     #and the fragmentation approach.
5     fragments = {} #Defines an empty dictionary that will store the fragments objects
6     bsid = lig.bsid #binding site id consisting of hetid:chain:position
7     lig_smiles = lig.cansmiles #Smile representation of the ligand
8     inchikey = lig.inchikey #Inchikey representation of the ligand
9     lig_to_pdb = lig.can_to_pdb #gets a mapping of ligand atom position to PDB atom position
10    if bsid in plipxml.bsites:
11        has_plip_data = True
12        bsite = plipxml.bsites[bsid] #From plipxml obtains the binding site data for the given ligand
13        fragments_smiles = get_fragments_smiles(lig_smiles, ftype) #calls the fragmentation approach
14        for fragsmiles in fragments_smiles:
15            fragment = Fragment(fragsmiles, lig_smiles, bsid, bsite, lig_to_pdb) #instance of class
16            fragments[fragsmiles] = fragment #Adds the fragment object to the dictionary
17    else:
18        has_plip_data = False
19
20    def get_fragment_RECAP_leaves(smile):
21        """Fragmentation with RECAP algorithm to obtain the main leaves of the tree"""
22        #Return a list of fragments for a given ligand smiles
23        fragsmiles = None
24        mol = Chem.MolFromSmiles(smile) #Get the ligand as rdkit mol
25        if mol:
26            #Rdkit Recap algorithm to decompose the ligand
27            hierarch = Recap.RecapDecompose(mol)
28            recap_fragments = hierarch.GetLeaves() #last leaves in tree
29            if len(list(recap_fragments.keys()))>0: #If at least one fragment
30                fragsmiles = list(recap_fragments.keys())
31        return fragsmiles
32
33
34    def get_fragment_RECAP_tree(smile):
35        """Fragmentation with RECAP algorithm to obtain the full tree"""
36        #Returns a list of fragments for a given ligand smiles
37        fragsmiles = None
38        mol = Chem.MolFromSmiles(smile) #Obtains the ligand molecule from its smiles
39        if mol:
40            #Rdkit Recap algorithm to decompose the ligand
41            hierarch = Recap.RecapDecompose(mol)
42            recap_fragments = hierarch.GetAllChildren() #full tree
```

```

43         If at least one fragment:
44             fragsmiles = list(recap_fragments.keys())
45         return fragsmiles
46
47
48     def get_fragment_BRICS_leaves(smile):
49         """Fragmentation with BRICS algorithm to obtain the main leaves of the tree"""
50         #Return a list of fragments for a given ligand smiles
51         fragsmiles = None
52         mol = Chem.MolFromSmiles(smile)
53         if mol:
54             #Rdkit BRICS algorithm to decompose the ligand
55             if len(list(BRICS.BRICSDecompose(mol, keepNonLeafNodes=False)))>0:
56                 fragsmiles = list(BRICS.BRICSDecompose(mol, keepNonLeafNodes=False))
57         return fragsmiles
58
59     def get_fragment_BRICS_tree(smile):
60         """Fragmentation with BRICS algorithm to obtain the full tree"""
61         #Return a list of fragments for a given ligand smiles
62         fragsmiles = None
63         mol = Chem.MolFromSmiles(smile)
64         if mol:
65             #Rdkit BRICS algorithm to decompose the ligand
66             if len(list(BRICS.BRICSDecompose(mol, keepNonLeafNodes=True)))>0:
67                 fragsmiles = list(BRICS.BRICSDecompose(mol, keepNonLeafNodes=True))
68         return fragsmiles
69
70     def get_fragments_smiles(smile, ftype):
71         """Fragmentation selected as parameter by the user"""
72         if ftype == "FRL":
73             fragment_smiles = get_fragment_RECAP_leaves(smile)
74         elif ftype == "FRT":
75             fragment_smiles = get_fragment_RECAP_tree(smile)
76         elif ftype == "FRBL":
77             fragment_smiles = get_fragment_BRICS_leaves(smile)
78         elif ftype == "FRBT":
79             fragment_smiles = get_fragment_BRICS_tree(smile)
80         return fragment_smiles

```

Finally, a fragment object is created for each fragment extracted from the ligand. The fragments atoms are mapped to the PDB atoms via ligand mapping and the PLIP interactions extracted for specific binding site are then transferred to the fragment level by reducing the binding site atoms to the ones that are part of a given fragment. In the presented work, the isomorphism mapping was done using the isomorphism class provided by PharmAI GmbH as a service.

```

1 class Fragment(fragsmile, ligsmile, pdbid, bsid, bsite_original, lig_to_pdb):
2     """Defines a fragment object"""
3     bsite = copy.deepcopy(bsite_original) #gets a copy of binsind site
4     fragsmile = remove_dummy_atom(fragsmile) #removing [*] from fragments smiles
5     fragmol = Chem.MolFromSmiles(fragsmile)
6     ligmol = Chem.MolFromSmiles(ligsmile)
7     lig_mapping = mapping_to_ligand(fragmol, ligmol) #Maps fragments atoms to ligand atoms
8     pdb_atoms = get_pdb_atoms(lig_mapping, lig_to_pdb) #Maps from fragments atoms to PBD atoms via ligand
9     mapping
10     if pdb_atoms:
11         reduce_bsite(bsite_original) #gets only interactions found in fragments atoms
12         fingerprint = Fingerprints(bsite) #Generate a fingerprint object with the interactions of the fragment
13     else:
14         fingerprint = None
15
16     def mapping_to_ligand(fragmol, ligmol):
17         """Gets the mapping of fragment atoms to the ligand atoms"""
18         isomorphs = isomorphism(fragmol, ligmol) #gets possible matching of fragment atoms within the ligand
19         atoms.
20         if isomorphs:
21             lig_mapping = isomorphs.pop() #gets first match
22             return lig_mapping
23         else:

```

```

22         return None
23
24     def get_pdb_atoms(lig_mapping, lig_to_pdb):
25         """Gets the pdb original atoms for the fragment via ligand mapping"""
26         pdb_atoms = []
27         if lig_mapping:
28             for mapp in lig_mapping:
29                 can_id = mapp[1]+1 #lig_to_pdb indexes start with 1 instead of 0
30                 if lig_id in lig_to_pdb:
31                     atom = lig_to_pdb[lig_id]
32                     pdb_atoms.append(atom)
33             if pdb_atoms:
34                 return pdb_atoms
35             else:
36                 return None
37         else:
38             return None
39
40     def check_ele_in_list(element, lis):
41         """Check if an element (tuple, list, string, int, etc) is in a given list"""
42         if type(element) is tuple:
43             sublist = list(element)
44             isin = set(sublist).issubset(lis)
45         elif type(element) is list:
46             isin = set(element).issubset(lis)
47         else:
48             isin = element in lis
49         return isin
50
51     def reduce_bsite(bsiteo):
52         """Keeps only the binding site interactions done by the specific fragment by comparing fragment pdb atoms to
53         the full ligand pdb atoms and replacing data with new updated list"""
54         #Hydrophobic interactions in fragment
55         bsite.hydrophobics = [hydroph for hydroph in bsiteo.hydrophobics if hydroph.ligcarbonidx in
56         pdb_atoms]
57         #Hydrogen bonds in fragment
58         bsite.hbonds = [hbond for hbond in bsiteo.hbonds if hbond.donoridx in pdb_atoms or hbond.
59         acceptoridx in pdb_atoms]
60         #Water bridges interactions in fragment
61         bsite.wbridges = [wbridge for wbridge in bsiteo.wbridges if wbridge.donor_idx in pdb_atoms
62         or wbridge.acceptor_idx in pdb_atoms]
63         #Halogen interactions in fragment
64         bsite.halogens = [halogen for halogen in bsiteo.halogens if halogen.don_idx in pdb_atoms or
65         halogen.acc_idx in pdb_atoms]
66         #Salt bridges interactions in fragment
67         bsite.sbridges = [sbridge for sbridge in bsiteo.sbridges if check_ele_in_list(sbridge.
68         lig_idx_list, pdb_atoms)]
69         #Pi-stacking interactions in fragment
70         self.bsite.pi_stacks = [pi_stack for pi_stack in bsiteo.pi_stacks if check_ele_in_list(
71         pi_stack.lig_idx_list, self.pdb_atoms)]
72         #Pi-cation interactions in fragment
73         self.bsite.pi_cations = [pi_cation for pi_cation in bsiteo.pi_cations if check_ele_in_list(
74         pi_cation.lig_idx_list, self.pdb_atoms)]
75         #Metal complexes in fragment
76         self.bsite.metal_complexes = [metal for metal in bsiteo.metal_complexes if metal.target_idx
77         in self.pdb_atoms and metal.location == "ligand"]

```