Fragmentation of PDB compounds

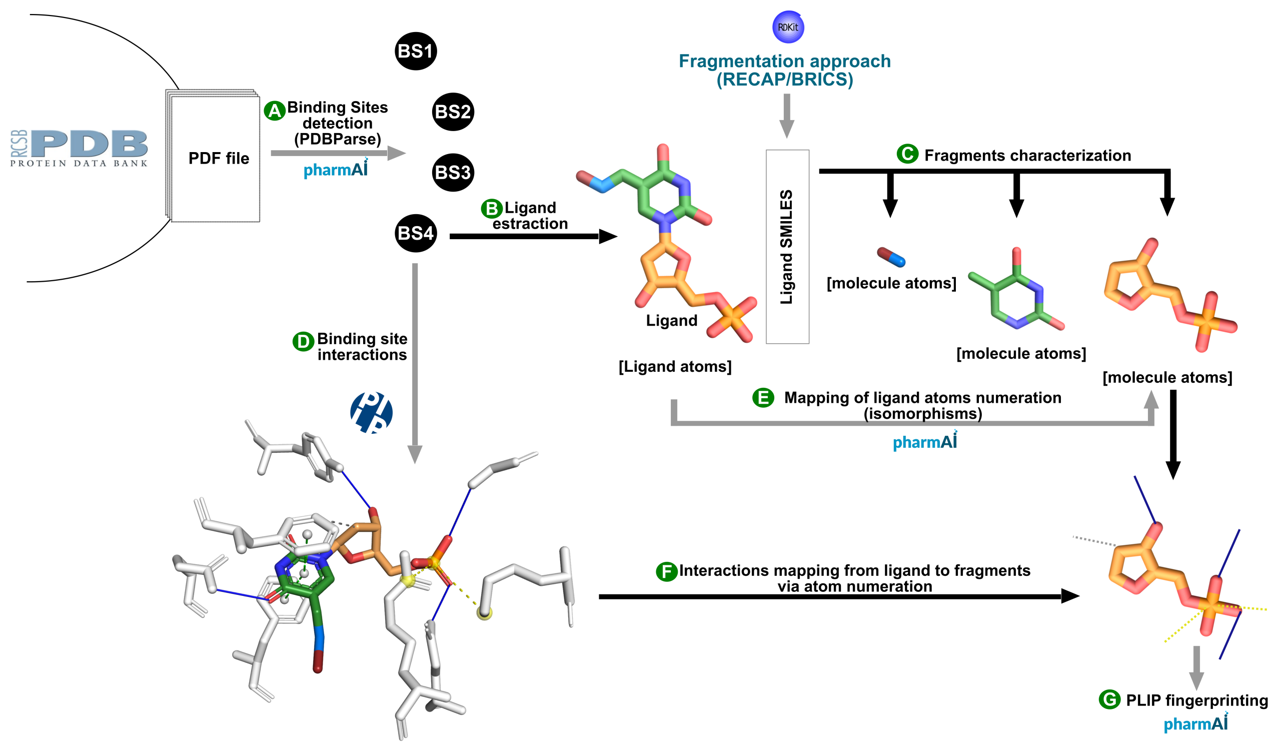
PDB
PROTEIN DATA BANK
PDF file
A Binding Sites detection (PDBParse)
pharmAI
BS1
BS2
BS3
BS4
B Ligand estraction
D Binding site interactions
Fragmentation approach (RECAP/BRICS)
Ligand
[Ligand atoms]
Ligand SMILES
C Fragments characterization
[molecule atoms]
[molecule atoms]
[molecule atoms]
E Mapping of ligand atoms numeration (isomorphisms)
pharmAI
F Interactions mapping from ligand to fragments via atom numeration
G PLIP fingerprinting
pharmAI

Figure 1: Fragmentation workflow used in the test case. The fragmentation applied to the test case, incorpores 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. For this test case it was used the PDBParse module provided by pharmAI, however this can be easily done by other cheminformatics tool such as RdKit or OpenBabel.

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 algorithms. 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.

```python
"""Defines a PDB file object and detects the binding ligands"""
class PDB(filename):
    pdbid = filename.split('.')[0] #extracts the PDB id from filename
    pdbparsed = PDBParser(filename) #instance of a class to parse the PDB file
    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.

```python
"""Defines a Ligand class to extract the molecule data"""
class Ligand(lig, plipxml, ftype):
    #Inputs are: ligand mol object parsed from PDB file, a plipxml object containing all PLIP recognized interactions,
    and the fragmentation approach.
    fragments = {} #Defines an empty dictionary that will store the fragments objects
    bsid = lig.bsid #binding site id consisting of hetid:chain:position
    lig_smiles = lig.cansmiles #Smile representation of the ligand
    inchikey = lig.inchikey #Inchikey representation of the ligan
    lig_to_pdb = lig.can_to_pdb #gets a mapping of ligand atom position to PDB atom position
    if bsid in plipxml.bsites:
        has_plip_data = True
        bsite = plipxml.bsites[bsid] #From plipxml obtains the binding site data for the given ligand
        fragments_smiles = get_fragments_smiles(lig_smiles, ftype) #calls the fragmentation approach
        for fragsmiles in fragments_smiles:
            fragment = Fragment(fragsmiles, lig_smiles, bsid, bsite, lig_to_pdb) #instance of class
            fragments[fragsmiles] = fragment #Adds the fragment object to the dictionary
    else:
        has_plip_data = False


    def get_fragment_RECAP_leaves(smile):
        """Fragmentation with RECAP algorithm to obtain the main leaves of the tree"""
        #Return a list of fragments for a given ligand smiles
        fragsmiles = None
        mol = Chem.MolFromSmiles(smile) #Get the ligand as rdkit mol
        if mol:
            #Rdkit Recap algorithm to decompose the ligand
            hierarch = Recap.RecapDecompose(mol)
            recap_fragments = hierarch.GetLeaves() #last leaves in tree
            if len(list(recap_fragments.keys()))>0: #If at least one fragment
                fragsmiles = list(recap_fragments.keys())
        return fragsmiles


    def get_fragment_RECAP_tree(smile):
        """Fragmentation with RECAP algorithm to obtain the full tree"""
        #Returns a list of fragments for a given ligand smiles
        fragsmiles = None
        mol = Chem.MolFromSmiles(smile) #Obtains the ligand molecule from its smiles
        if mol:
            #Rdkit Recap algorithm to decompose the ligand
            hierarch = Recap.RecapDecompose(mol)
            recap_fragments = hierarch.GetAllChildren() #full tree
            If at least one fragment:
                fragsmiles = list(recap_fragments.keys())
        return fragsmiles


    def get_fragment_BRICS_leaves(smile):
        """Fragmentation with BRICS algorithm to obtain the main leaves of the tree"""
        #Return a list of fragments for a given ligand smiles
        fragsmiles = None
        mol = Chem.MolFromSmiles(smile)
        if mol:
            #Rdkit BRICS algorithm to decompose the ligand
            if len(list(BRICS.BRICSDecompose(mol,keepNonLeafNodes=False)))>0:
                fragsmiles = list(BRICS.BRICSDecompose(mol, keepNonLeafNodes=False))
        return fragsmiles

    def get_fragment_BRICS_tree(smile):
        """Fragmentation with BRICS algorithm to obtain the full tree"""
        #Return a list of fragments for a given ligand smiles
        fragsmiles = None
        mol = Chem.MolFromSmiles(smile)
        if mol:
            #Rdkit BRICS algorithm to decompose the ligand
            if len(list(BRICS.BRICSDecompose(mol,keepNonLeafNodes=True)))>0:
                fragsmiles = list(BRICS.BRICSDecompose(mol, keepNonLeafNodes=True))
        return fragsmiles

    def get_fragments_smiles(smile, ftype):
        """Fragmentation selected as parameter by the user"""
        if ftype == "FRL":
            fragment_smiles = get_fragment_RECAP_leaves(smile)
        elif ftype == "FRT":
            fragment_smiles = get_fragment_RECAP_tree(smile)
        elif ftype =="FRBL":
            fragment_smiles = get_fragment_BRICS_leaves(smile)
        elif ftype =="FRBT":
            fragment_smiles = get_fragment_BRICS_tree(smile)
        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.

```python
class Fragment(fragsmile, ligsmile, pdbid, bsid, bsite_original, lig_to_pdb):
    """Defines a fragment object"""
    bsite = copy.deepcopy(bsite_original) #gets a copy of binsind site
    fragsmile = remove_dummy_atom(fragsmile) #removing [*] from fragments smiles
    fragmol = Chem.MolFromSmiles(fragsmile)
    ligmol = Chem.MolFromSmiles(ligsmile)
    lig_mapping = mapping_to_ligand(fragmol, ligmol) #Maps fragments atoms to ligand atoms
    pdb_atoms = get_pdb_atoms(lig_mapping, lig_to_pdb) #Maps from fragments atoms to PBD atoms via ligand
    mapping
    if pdb_atoms:
        reduce_bsite(bsite_original) #gets only interactions found in fragments atoms
        fingerprint = Fingerprints(bsite) #Generate a fingerprint object with the interactions of the fragment
    else:
        fingerprint = None

    def mapping_to_ligand(fragmol, ligmol):
        """Gets the mapping of fragment atoms to the ligand atoms"""
        isomorphs = isomorphism(fragmol, ligmol) #gets possible matching of fragment atoms within the ligand
    atoms.
        if isomorphs:
            lig_mapping = isomorphs.pop() #gets first match
            return lig_mapping
        else:
            return None

    def get_pdb_atoms(lig_mapping, lig_to_pdb):
        """Gets the pdb original atoms for the fragment via ligand mapping"""
        pdb_atoms = []
        if lig_mapping:
            for mapp in lig_mapping:
                can_id = mapp[1]+1 #lig_to_pdb indexes start with 1 instead of 0
                if lig_id in lig_to_pdb:
                    atom = lig_to_pdb[lig_id]
                    pdb_atoms.append(atom)
            if pdb_atoms:
                return pdb_atoms
            else:
                return None
        else:
            return None

    def check_ele_in_list(element, lis):
    """Check if an element (tuple, list, string, int, etc) is in a given list"""
        if type(element) is tuple:
            sublist = list(element)
            isin = set(sublist).issubset(lis)
        elif type(element) is list:
            isin = set(element).issubset(lis)
        else:
            isin = element in lis
        return isin

    def reduce_bsite(bsiteo):
        """Keeps only the binding site interactions done by the specific fragment by comparing fragment pdb atoms to the full ligand pdb atoms and replacing data with new updated list"""
        #Hydrophobic interactions in fragment
        bsite.hydrophobics = [hydroph for hydroph in bsiteo.hydrophobics if hydroph.ligcarbonidx in pdb_atoms]
        #Hydrogen bonds in fragment
        bsite.hbonds = [hbond for hbond in bsiteo.hbonds if hbond.donoridx in pdb_atoms or hbond.acceptoridx in pdb_atoms]
        #Water bridges interactions in fragment
        bsite.wbridges = [wbridge for wbridge in bsiteo.wbridges if wbridge.donor_idx in pdb_atoms or wbridge.acceptor_idx in pdb_atoms]
        #Halogen interactions in fragment
        bsite.halogens = [halogen for halogen in bsiteo.halogens if halogen.don_idx in pdb_atoms or halogen.acc_idx in pdb_atoms]
        #Salt bridges interactions in fragment
        bsite.sbridges = [sbridge for sbridge in bsiteo.sbridges if check_ele_in_list(sbridge.lig_idx_list, pdb_atoms)]
        #Pi-stacking interactions in fragment
        self.bsite.pi_stacks = [pi_stack for pi_stack in bsiteo.pi_stacks if check_ele_in_list(pi_stack.lig_idx_list, self.pdb_atoms)]
        #Pi-cation interactions in fragment
        self.bsite.pi_cations = [pi_cation for pi_cation in bsiteo.pi_cations if check_ele_in_list(pi_cation.lig_idx_list, self.pdb_atoms)]
        #Metal complexes in fragment
        self.bsite.metal_complexes = [metal for metal in bsiteo.metal_complexes if metal.target_idx in self.pdb_atoms and metal.location == "ligand"]
```