Pharmacophore Representation and Searching

In this article we describe the design and use of a set of Java classes to represent pharmacophores and use such representations in pharmacophore searching applications.

Rajarshi Guha & John Van Drie

Introduction

Pharmacophores can be defined in a number of ways [1, 2]. From a physical standpoint a pharmacophore is a collection of electronic and steric features of a molecule that allow it to exhibit activity against a given biological target. More specifically, a pharmacophore can be defined by a set of atoms and various geometric (distances, angles, torsions) relationships between them. The pharmacophore concept has been used to study a variety of biological targets such as the dopamine D2 receptor [3], HIV integrase [4] and reverse transcriptase [5] and COX-2 [6] amongst others.

The key component to these studies is the ability to generate databases of 3D conformations of candidate ligands and then search for a given pharmacophore within the database. Currently open source solutions are available for chemical databases [7] and conformer generation. However there are no open source tools for pharmacophore representation and searching. The lack of freely available solutions led to the design of the pharmacophore classes in the CDK. The rest of this article will describe the design of the pharmacophore handling classes and highlight their usage.

It should be noted that the current implementation is not a comprehensive pharmacophore modeling solution as it does not provide complex geometric constraints (torsions and excluded volumes) or any graphical or pharmacophore discovery capabilities. However the framework is sufficiently general for these features to be incorporated in the future.

Representing a Pharmacophore

Pharmacophore representation involves two components: atoms or groups of atoms that exhibit a certain property (such as aromaticity, hydrophobicity and so on) and geometrical constraints between these groups. Though geometrical constraints can include distances, angles, torsions and so on, we restrict ourselves to simple distance constraints. The formalism that has been employed in the design of the CDK pharmacophore classes is to view a pharmacophore as a graph.

Thus a pharmacophore group is termed a *pharmacophore node* and is constructed from one or more atoms in the original molecule. A distance constraint is termed a *pharmacophore edge* and is constructed from two pharmacophore nodes and the distance between them. The idea can be extended to angle constraints, in which case the "edge" would involve three pharmacophore nodes and the angle between them. This aspect is described in more detail later on. Thus one can consider a pharmacophore in terms of a *pharmacophore graph* constructed from a set of pharmacophore nodes and pharmacophore bonds. This terminology allows us to describe pharmacophore matching in an intuitive manner. Thus in Fig. 1, we start of with the original molecule and then identify H-bond acceptor, H-bond donor and aromatic groups (red, green and purple circles respectively) which are termed the pharmacophore nodes. Next we identify the distances between some of these groups. These distance can be converted to ranges for the purposes of querying, resulting in a query pharmacophore groups constructed from the pharmacophore groups and distance ranges between them.

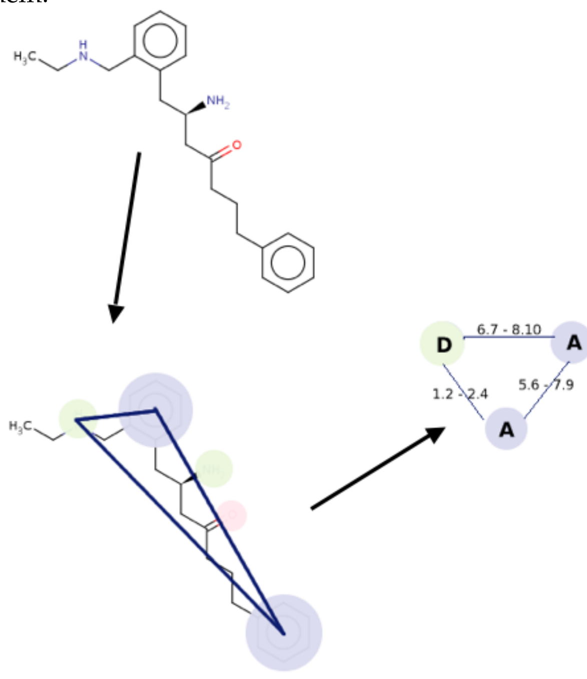

Figure 1: Conversion of a traditional molecule to a pharmacophore graph.

Given the above description one must note that there can be two forms of pharmacophore representations. The main difference between the two forms is the nature of the constraints and the mapping to a real 3D molecular structure. Thus given a molecule one can apply the pharmacophore definition and identify pharmacophore nodes which in turn can be mapped to sets of atoms in the real molecule, that match the SMARTS pattern of the given pharmacophore node. However the pharmacophore edges (i.e., distance ranges) are not really ranges since in a given conformation the coordinates are fixed. On the other hand a *pharmacophore query* will generally involve distance ranges but will not identify specific atoms.

The usage of such representations also differs. From the users point of view, the input to a pharmacophore matching algorithm is the pharmacophore query. On the other hand, once a query has been matched, they will get back the first type of the pharmacophore, where the distance is an exact, observed distance and the pharmacophore nodes can be mapped to real atoms in the input structure.

The two situations are handled by the classes **PharmacophoreAtom**, **PharmacophoreBond**, **PharmacophoreQueryAtom** and **PharmacophoreQueryBond** from the package `org.openscience.cdk.pharmacophore`. The first two classes are used to identify specific instances of a pharmacophore definition and the last two classes are used to represent a general pharmacophore query.

Constructing a Pharmacophore Query

As noted above, a pharmacophore query requires us to identify pharmacophore query nodes and specify distance constraints. Pharmacophore query nodes are specified using SMARTS patterns. In general such nodes are of a few types such as aromatic centers, H-bond donor groups and so on. In addition, to specifying the SMARTS pattern and distance ranges the CDK implementation requires one to also provide a string label for the node. Minimal checking is performed on the label, so if two pharmacophore query nodes are constructed using the same underlying SMARTS they are expected to have the same label. A code snippet to construct a set of pharmacophore query nodes is:

```
PharmacophoreQueryAtom o =
   new PharmacophoreQueryAtom("D", "[NX2]");
PharmacophoreQueryAtom n1 =
   new PharmacophoreQueryAtom("A", "c1ccccc1");
PharmacophoreQueryAtom n2 =
   new PharmacophoreQueryAtom("A", "c1ccccc1");
```

Here we define three query pharmacophore nodes, viz., a single H-bond donor group and two aromatic centers.

Given the pharmacophore nodes, the next step is to specify distance constraints. This is done by constructing pharmacophore query edges:

```
PharmacophoreQueryBond b1 =
  new PharmacophoreQueryBond(o, n1, 4.0, 4.5);
PharmacophoreQueryBond b2 =
   new PharmacophoreQueryBond(o, n2, 4.0, 5.0);
PharmacophoreQueryBond b3 =
   new PharmacophoreQueryBond(n1, n2, 5.4, 5.8);
```

It is also possible to specify an exact distance constraint rather than a distance range.

At this point we have a set of query pharmacophore nodes and edges. Though the pharmacophore query nodes and edges are conceptual constructs, they have been implemented as subclasses of **IQueryAtom** and **IQueryBond** respectively, which are located in the `org.openscience.cdk.isomorphism.matchers` package. As a result we can place them all in an **IQueryAtomContainer** resulting in *pharmacophore query graph*. Using the objects we created above we can do:

```
QueryAtomContainer query =
    new QueryAtomContainer();

query.addAtom(o);
query.addAtom(n1);
query.addAtom(n2);

query.addBond(b1);
query.addBond(b2);
query.addBond(b3);
```

The query graph can now be matched against a pharmacophore representation of the target molecule. More specifically, the **IQueryAtom**'s and **IQueryBond**'s of the query graph are matched against a *p*harmacophore target graph derived from an input 3D molecular structure.

It should be noted that though we have considered a triad in this example, one is not restricted to such a query. One can include as many pharmacophore groups as desired with as many constraints as desired between the pharmacophore nodes. In a very complicated query, some distance constraints might be redundant; the CDK classes do not attempt to analyze the set of constraints to obtain a smaller set of unique constraints.

Constructing a Pharmacophore Target

Given a pharmacophore query, we must have a target to match against. This target is a 3D molecular structure. However rather than directly matching a pharmacophore query against the input structure, we process the input structure to generate a pharmacophore target graph.

The first step in generating the pharmacophore target is to make a list of the unique SMARTS patterns for the pharmacophore groups specified in the query. These SMARTS patterns are used to identify the corresponding atoms in the input structure. The left hand side of Fig. 2 highlights this step. For each SMARTS match in the input structure we can instantiate a **PharmacophoreAtom** object containing the group label and the matching atoms from the input structure. Once all the pharmacophore nodes have been constructed we generate a pharmacophore graph by connecting all the identified groups by pharmacophore edges. Each such edge stores the distance between the two participating groups.

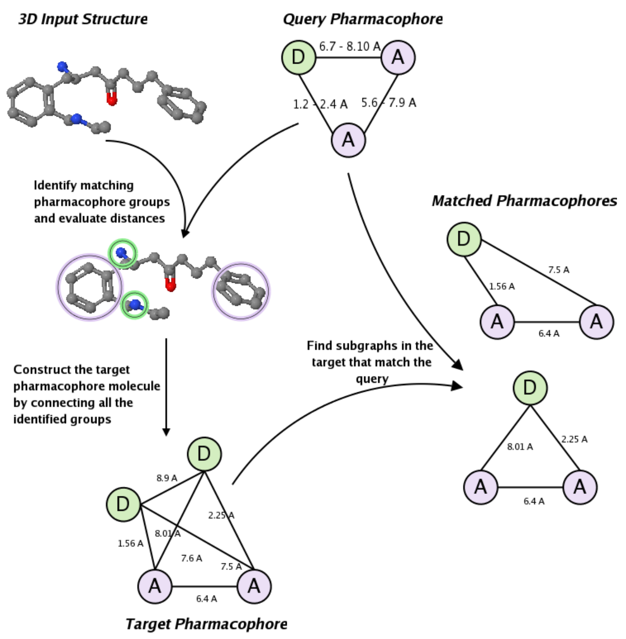
3D Input Structure
Query Pharmacophore
6.7 - 8.10 A
1.2 - 2.4 A
5.6 - 7.9 A
Identify matching pharmacophore groups and evaluate distances
Matched Pharmacophores
7.5 A
1.56 A
6.4 A
Find subgraphs in the target that match the query
Construct the target pharmacophore molecule by connecting all the identified groups
8.01 A
2.25 A
6.4 A
8.9 A
2.25 A
8.01 A
1.56 A
7.6 A
7.5 A
6.4 A
Target Pharmacophore

Figure 2: A summary of the steps involved in performing a pharmacophore query against a 3D input structure. Here D and A represent H-bond donor and aromatic pharmacophore groups respectively.

Note that distances are calculated between the *effective* coordinates of pharmacophore nodes. Thus for a multi-atom pharmacophore group (say an aromatic center such as a phenyl ring) the coordinates of the resultant pharmacophore node are set to the coordinates of the geometric center of the atoms in the original molecule (in this case, the 6 carbon atoms of the phenyl ring). If the SMARTS match results in a single atom from the original molecule, then the coordinates of that atom are used as the coordinates of the pharmacophore node.

The resultant target pharmacophore graph is shown in the lower left of Fig. 2. This construct thus represents all the possible pharmacophores (given a set of query groups) in the input structure. It should be noted that this stage is not visible to the user and is performed internally before pharmacophore matching.

Pharmacophore Matching

At this point we have a user supplied pharmacophore query graph and a pharmacophore target graph generated from a 3D input structure. The next step is to identify pharmacophores in the target that match the query. It is clear from Fig. 2 that this is essentially a subgraph isomorphism problem. Since the pharmacophore node and edge classes are subclassed from **IAtom** and **IBond** respectively, this means that we can reuse the **UniversalIsomorphismTester** class (from the `org.openscience.cdk.isomorphism` package) that is traditionally used for substructure searching in ordinary molecules.

As with traditional substructure searching, the pharmacophore query nodes and edges implement a `matches()` function which return `true` if the query node (or edge) matches an node (or edge) in the target graph and `false` otherwise. More specifically, a query pharmacophore node matches a target pharmacophore node if the labels of the two nodes are the same and a query pharmacophore edge matches a target pharmacophore edge if the distance represented by latter is contained within the range specified for the former. Note that it is possible to specify an exact distance for a query edge, in which case the target bond distance must match exactly. When distance constraints are matched the distances involved are evaluated to two decimal places. The isomorphism algorithm employed by the CDK is based on the method described by Tonnelier et al. [8].

It is clear that the procedure for pharmacophore matching is analogous to substructure searching, the difference lying in what properties are being compared (e.g., bond types versus distance ranges). Usage of the pharmacophore matching class is quite simple. Using the pharmacophore query that was constructed above we can do

```
IAtomContainer aMolecule;

// load in the molecule

PharmacophoreMatcher matcher =
    new PharmacophoreMatcher(query);
boolean status = matcher.matches(aMolecule);
if (status) {
  // get the matching pharmacophore groups
  List<List<PharmacophoreAtom>> pmatches =
    matcher.getUniqueMatchingPharmacophoreAtoms();
}
```

The right hand side of Fig. 2 displays the results of matching a pharmacophore query against a pharmacophore target graph derived from an 3D input structure.

After having performed the match one can get the list of matching pharmacophores in the input structure as shown above. The return value is a List of a List of **PharmacophoreAtom**. That is, each match will be a list of pharmacophore nodes (corresponding to the types in the query) and multiple matches may be present. For a given match, one can then loop over the individual pharmacophore nodes in the match:

```
List<PharmacophoreAtom> patoms = pmatches.get(0);
for (PharmacophoreAtom patom : patoms) {
  String label = patom.getSymbol();
  int[] indices = patom.getMatchingAtoms();
}
```

Note the last line of the loop. Since a given pharmacophore node represents the results of a SMARTS match within the original input molecule, we can access those matching atoms by their indices. As a result of this, one must ensure that the atom ordering in the 3D input structure does not change while pharmacophore matching is being performed.

Thus pharmacophore matching is programmatically very similar to SMARTS matching and uses much of the same semantics. The only difference is an extra layer of abstraction. The result of a pharmacophore match does not provide direct access to a set of matching **IAtom** objects. Rather we get a set of **PharmacophoreAtom** objects, from which we can access the information about the actual **IAtom** objects corresponding to that pharmacophore group.

Handling Multiple Conformations

Though the pharmacophore classes are more or less self-contained, a general solution to pharmacophore matching requires one to handle multiple conformers of a given molecule. Recently two classes were added to the CDK, viz., **IteratingMDLConformerReader** and **ConformerContainer**. The first class allows one to iterate over an SD file containing multiple conformers for multiple molecules such that at each iteration one receives all the conformers for the current molecule. These conformers are provided to the user as an object of type **ConformerContainer**. This class represents a memory-efficient approach to the problem of handling multiple conformers. Rather than storing multiple conformers as individual objects of type **IAtomContainer**, it stores a single **IAtomContainer** object and a list of **Point3d[]**. Each element of the list represents the coordinates for a given conformer. Since the **ConformerContainer** class extends the **List** interface, one can access arbitrary conformers or iterate over all the conformers that are available. In either case the class will internally populate the **IAtomContainer** object with the appropriate coordinates and return it to the user.

Since the conformers of a given molecule will always lead to the same pharmacophore groups being identified, pharmacophore matching a set of conformers can be made much more efficient by performing the initial pharmacophore group perception just once. Given the group assignments one can then rapidly evaluate the distances between groups and construct a target pharmacophore graph for every conformer. Such an approach leads to a significant reduction in processing time. For example, if one performs a pharmacophore query against 100 conformers of (2R)-2-amino-1-[2-(ethylaminomethyl)phenyl]-7-phenyl-heptan-4-one (Fig. 3) [9] by loading them

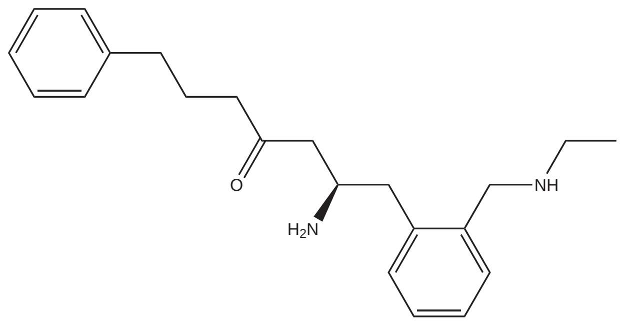

Figure 3: The structure of (2R)-2-amino-1-[2-(ethylaminomethyl)phenyl]-7-phenyl-heptan-4-one.

as individual **IAtomContainer** objects, execution time is approximately 32s versus less than 1s if they are processed as conformers via a **ConformerContainer** object (all timings on a 2.16 GHz Macbook Pro with 1GB RAM).

In terms of usage, the interface to pharmacophore matching with a set of conformers is slightly different compared to the case where one matches against a single molecule. In the conformer scenario one performs a match as before

```
ConformerContainer confs;

// load in the molecule

PharmacophoreMatcher matcher =
    new PharmacophoreMatcher(query);
boolean[] status = matcher.matches(confs);
for (boolean aStatus : status) {
  if (aStatus) {
    // this conformer matched
}
```

the difference being that rather than a single `true/false` value, one receives an array of values. The *i*'th element is `true` if the *i*'th conformer matches the query pharmacophore.

Serializing Pharmacophore Queries

Though it is relatively simple to construct a pharmacophore query programmatically, it is useful to be able to serialize and deserialize pharmacophore queries, allowing users and programs to exchanges pharmacophore models. Currently, there is no standardized format to store such queries. We have designed an XML format to represent pharmacophore queries. The format is described by a RelaxNG schema, which is available from Ref. [10]. The format allows one to define multiple pharmacophore groups using `group` elements. Currently the format supports distance and angle constraints (though the latter is ignored by the CDK matching code) using the `distanceConstraint` and `angleConstraint` elements. An example of a set of queries written in this format is available from Ref. [11]. The CDK provides the **PharmacophoreUtils** class, which contains methods to read queries stored in this format. Corresponding functionality to write out queries is in development.

Future Work

The current implementation allows one to define relatively simple pharmacophore queries involving distance constraints. However, a number of aspects must be considered to achieve a comprehensive pharmacophore matching system.

As noted above, the current pharmacophore implementation only handles distance constraints. It is relatively easy to include angle constraints (involving three pharmacophore groups). This would require two extra classes. The first class would be conceptually similar to **PharmacophoreBond** but would contain the three pharmacophore groups defining an angle. This class would be used to represent all the angles between pharmacophore groups in a target molecule. The second class would be analogous to **PharmacophoreQueryBond** and would contain the three **PharmacophoreQueryAtom**'s specified in a query along with an angle or angle range. Since these two classes would implement the **IBond** and **IQueyBond** interfaces respectively, matching angle constraints would be considered in exactly the same way as distance constraints are currently handled[1]. The downside to including more constraint types is that the target pharmacophore graph will have to include all possible constraints between the relevant groups. Thus a target pharmacophore (Fig. 2) would include such a constraint between all combinations of three pharmacophore groups, resulting in an extra $\binom{n}{3}$ "edges", where n is the number of identified pharmacophore groups. Since pharmacophore queries are usually triads or quads (implying one to four possible angle constraints in the query graph), this should not lead to a significant performance decrease.

Currently, there are no predefined pharmacophore groups and SMARTS patterns must be supplied at construction time. As progress is made on the SMARTS parser we will include a set of standard pharmacophore groups.

An important component of a pharmacophore framework is the ability to visualize a pharmacophore match in a 3D structure. As this is not directly related to pharmacophore representation and searching it will not be addressed by the current classes. Work is in progress to provide such a facility using Jmol [12] and it is expected that it will utilise the pharmacophore classes discussed here.

Finally a longer term goal is to implement pharmacophore discovery algorithms that would take a set of structures and associated activity data and attempt to identify a set of pharmacophores that would explain the structure-activity relationships.

Rajarshi Guha
Indiana University
`rguha@indiana.edu`

John Van Drie
John H Van Drie Research LLC
`johnvandrie@mindspring.com`

Bibliography

[1] J.H. van Drie. Pharmacophore discovery - Lessons learned. *Curr. Pharm. Des.*, 9(20):1649–1664, 2003.

[2] C. G. Wermuth, C .R. Gannelin, P. Lindberg, and L. A. Mitscher. Glossary of terms used in medicinal chemistry. *Pure Appl. Chem.*, 70:1129–1143, 1998.

[3] Y. C. Martin. 3D database searching in drug design. *J. Med. Chem.*, 35(12):2145–2154, 1992.

[4] J. Deng, R. Dayam, L. Q. Al-Mawsawi, and N. Neamati. Design of second generation HIV-1 integrase inhibitors. *Curr. Pharm. Des.*, 13:129–141, 2007.

[5] M. L. Barreca, L. De Luca, N. Iraci, A. Rao, S. Ferro, G. Maga, and A. Chimirri. Structure-based pharmacophore identification of new chemical scaffolds as non-nucleoside reverse transcriptase inhibitors. *J. Chem. Inf. Model.*, 47:557–562, 2007.

[6] L. Franke, E. Byvatov, O. Werz, D. Steinhilber, P. Schneider, and G. Schneider. Extraction and visualization of potential pharmacophore points using support vector machines: Application to ligand-based virtual screening for COX-2 inhibitors. *J. Med. Chem.*, 48:6997–7004, 2005.

[7] E.G. Schmid. pgchem::tigress. `http://pgchem.projects.postgresql.org/`, accessed Aug. 2007.

[8] C. Tonnelier, Ph. Jauffret, Th. Hanser, Ph. Jauffret, and G. Kaufmann. Machine learning of generic reactions: 3. An efficient algorithm for maximal common substructure determination. *Tetrahedron Comput. Method.*, 3:351–358, 1990.

[9] InChI=1/C22H30N2O/c1-2-24-17-20-13-7-6-12-19(20)15-21(23)16-22(25)14-8-11-18-9-4-3-5-10-18/h3-7,9-10,12-13,21,24H,2,8,11,14-17,23H2,1H3.

[1] Though this description is analogous to that for distance constraints, it can be difficult to visualize angle constraints as "edges" in a pharmacophore graph

[10] `http://cheminfo.informatics.indiana.edu/~rguha/code/java/pcore/pharmacophore.rng`, accessed Jan. 2008.

[11] `http://cheminfo.informatics.indiana.edu/~rguha/code/java/pcore/pcore.xml`, accessed Jan. 2008.

[12] The Jmol 3D Molecular Visualization Software. `http://www.jmol.org/`, accessed Jan. 2008.