OpenGrowth v1.0.1 User Manual

Table of Contents

0. Forewords

This user manual describes the use of OpenGrowth and related programs. Be sure to read the Chapter 4 of this manual "Example of use" if you are a new user. For any questions, please send an email exclusively at opengrowth-discuss@lists.sourceforge.net. To register to the mailing list, go here: https://lists.sourceforge.net/lists/listinfo/opengrowth-discuss. To see the archives of the list, go there: https://sourceforge.net/p/opengrowth/mailman/opengrowth-discuss/.

OpenGrowth is available at the following address: http://opengrowth.sourceforge.net/. To download all the files, click on "Files" on the horizontal menu at the top (or go here https://sourceforge.net/projects/opengrowth/files/). You will need *OpenGrowth_1.0.1.zip* and *Resources_1.X.zip* to start (take the most recent version, *1.0.2* at the time of writing this manual). If OpenGrowth helped you to identify new ligands, we would appreciate that you keep us posted. The program is released under the GNU GPL license: it allows you to use the program on any projects (whether you are from industry or from academia). However, if you plan to release a program that uses OpenGrowth, it must use the GNU GPL license.

1. Presentation

OpenGrowth is a program that can be used for Computer-Aided Drug Design; it aims at constructing *de novo* ligands for proteins. It has been developed in the Shakhnovich Lab at Harvard University (Department of Chemistry and Chemical Biology) by Nicolas Chéron and Eugene Shakhnovich. OpenGrowthGUI has been developed by Naveen Jasty, Nicolas Chéron and Eugene Shakhnovich. OpenGrowth merges the approaches developed in SMoG [1,2] and FOG [3], and adds new options such as the protein flexibility or the included post-process. It has been re-written from scratch in C++ using the OpenBabel library [4]. The overall algorithm is shown in Figure 1 and is explained in Chapter 2. More details can be found in the publication "OpenGrowth: an automated and rational algorithm for finding new protein ligands" by Nicolas Chéron, Naveen Jasty and Eugene Shakhnovich [5]. The SMoG2016 scoring function is described in "A Hybrid Knowledge-Based and Empirical Scoring Function for Protein–Ligand Interaction: SMoG2016" by Théau Debroise, Eugene Shakhnovich and Nicolas Chéron [6]. Our overall goal in developing this new tool is to quickly design drug-like molecules, while predicting their affinities with good accuracy and taking into account protein flexibility. Future goals will be to allow covalent docking, to enhance binding specificity to reduce side-effect binding and toxicity and to take into account the role of water in binding (amongst other tasks). See roadmap (Chapter 15) for more details.

2. Overview

The growth process starts with the placement of an organic fragment into the active site of the target protein. In the input file (described later), the user must define the center of the active site (by means of the (x,y,z) coordinates) as well as the size of a box around it (see Figure 2-a). Once the first fragment has been selected, this fragment is put in the box at a random position and with a random orientation. We present in Figure 2-b a set of starting fragments that occupy a large part of the input box with different orientations. After the selection of the first fragment, a rotameric search is performed (see Figure 2-c) and the orientation with the lowest score is kept.

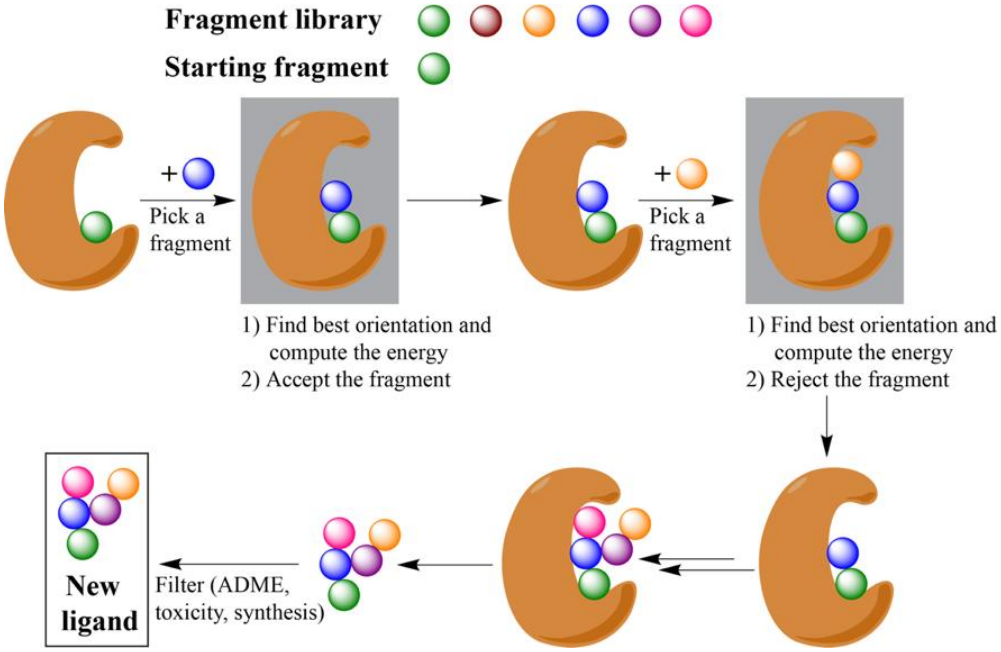
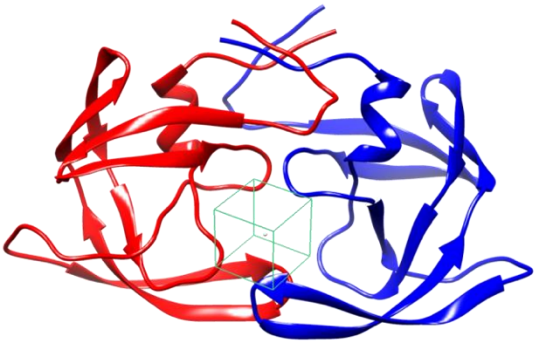
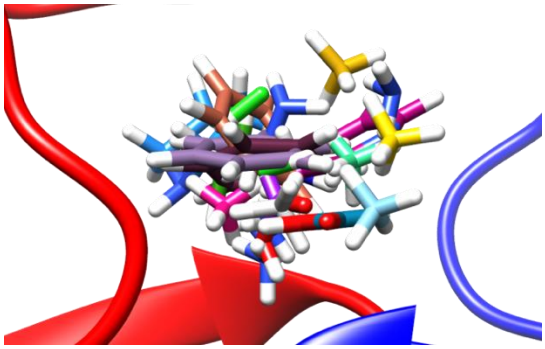


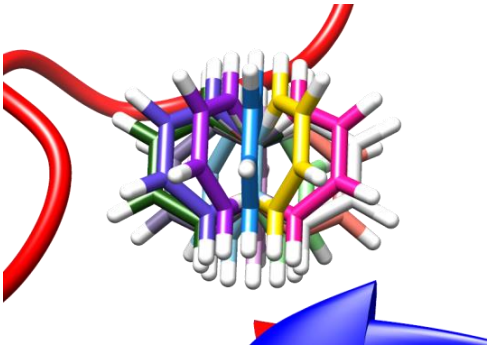
Figure 1. Overview of the OpenGrowth algorithm.



(a) Box for the starting fragment around the center of the active site.



(b) Set of starting fragments.

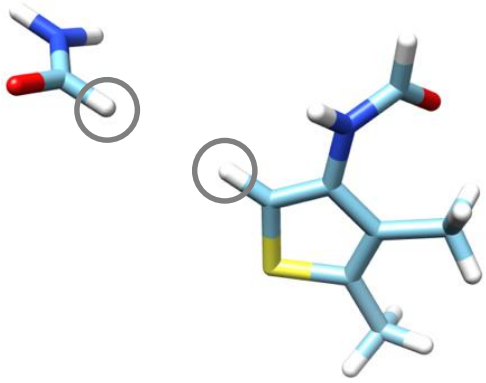


(c) Looking for the best orientation for the first rotamer.

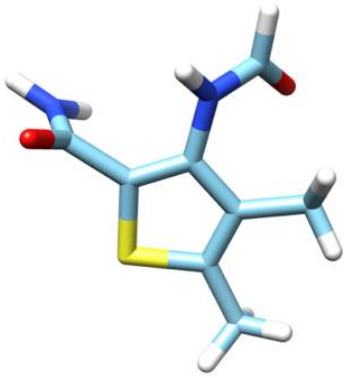
Figure 2. Starting fragment.

New chemical groups are then added to the current ligand to increase the binding affinity of the small molecule to the protein. Since random growth can lead to toxic or unstable molecules (e.g.), the growth is biased to produce more “drug-like” molecules. To achieve this, we use the approach developed in FOG [3]: the algorithm was trained on a drug database from which trends were extracted (see further). This information can then be used to grow new molecules, which will “look like” the molecules in the initial database. A Metropolis criterion (based on the protein-ligand binding free energy) is used to decide whether to accept or reject a new chemical group.

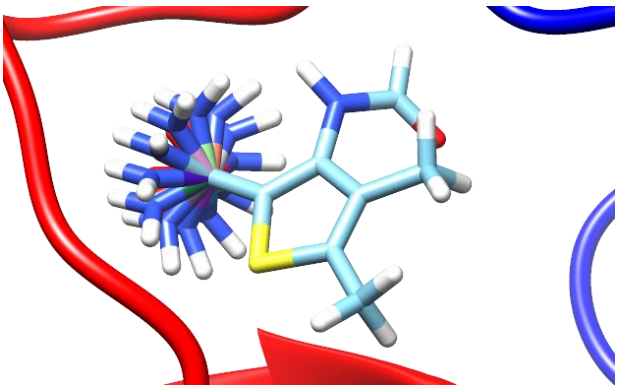
When a new fragment is added, a hydrogen atom from the current ligand is randomly selected. The fragment to which this atom belongs is known by the program, and the new fragment is chosen according to the FOG transition probabilities of the previous fragment. For example, in Figure 3-a, a hydrogen from the thiophene fragment is randomly selected, and an amide secondary fragment (from the carbonyl side) was chosen using the transition probabilities. When multiple hydrogens can potentially be chosen (for example, from the imine part of an amide), the hydrogen which will be removed is randomly selected. The two fragments are then aligned (see Figure 3-a), the two hydrogen atoms are removed, and a new single bond is created between the two fragments (see Figure 3-b). A rotameric search is then performed (see Figure 3-c) to keep the orientation of the new fragment with the lowest score.



(a) Aligning the ligand and the new fragment.



(b) Connecting the fragments.



(c) Rotameric search for the new fragment.

Figure 3. Connecting the new fragment to the current ligand.

We present below the principle of the FOG algorithm (Figure 4). If the input database is made of the five displayed molecules, we first count how many times a phenyl ring is connected to the different fragments. For example, phenyl is bonded three times to a methyl-based fragment (twice in Ibuprofen and once in Ketoprofen). This corresponds to 3/11=27% of all the connections between a phenyl and any fragment of the chosen library. Therefore, if the hydrogen of a phenyl is selected for growth initiation, a methyl will be chosen as the fragment to be added 27% of the time. Applied to drug discovery, this approach allows to discover new compounds that are more easily synthesizable and with better pharmaco-kinetic properties than with a random growth (which is also possible as an option). Probability files needed by the program are provided in the *Resources* folder (that has to be downloaded separately from the website). If one wants to change the provided set of fragments (either to remove some or to add new ones), the graphical interface OpenGrowthGUI can help to calculate the probability files in an automated way. Screening tools are also included in the program as options to further increase the similarity between the initial training database and the new molecules, for example the 3mer screen (see Chapter 6).

Two knowledge-based potentials developed to describe protein/small molecule interactions are currently available to calculate binding affinities (that are used for the Metropolis criterion during the growth). One of them is SMoG2001 [7], the other one is SMoG2016 [6]. We highly recommend the use of SMoG2016. The correlation coefficient between the SMoG2016 scores and experimental binding free energies in a set of 195 complexes (which are not present in the training set) ranges from 0.56 to 0.57 (depending on the compiler and the version of OpenBabel). To compare SMoG2016 to other popular scoring functions, this testing set is the same as the one used by Li *et al.* [8] and it appears that SMoG2016 is competitive with the most accurate scoring functions.

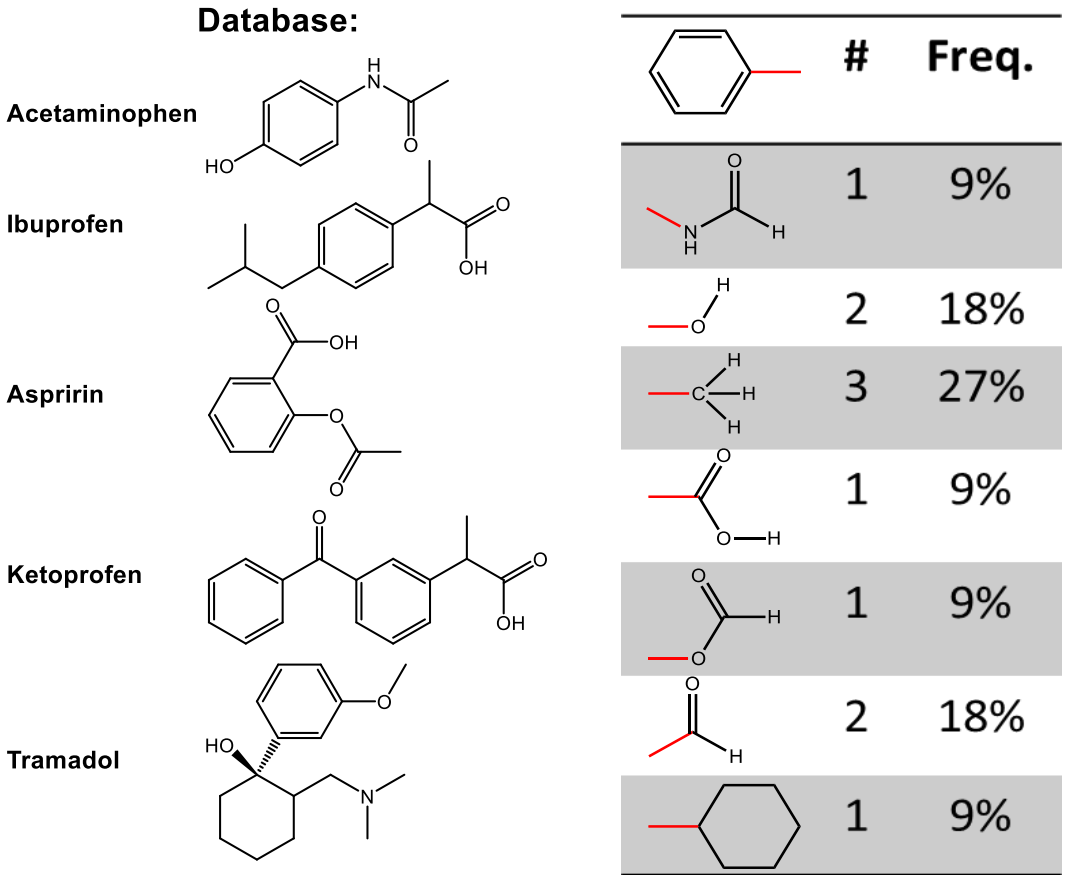


Figure 4. Principle of the FOG algorithm.

3. Installation of OpenGrowth

Pre-compiled binaries of OpenGrowth were provided for previous versions. It is no longer the case. You first need to install OpenBabel, either from the repositories or by compiling it. Note that you need to either install OpenBabel from the repositories OR compile it (you don't need to do both).

3.1. Installing OpenBabel *via* the repositories on Linux

The first solution to obtain OpenBabel is to do it *via* the repositories. You only need the OpenBabel libraries and not the program itself and you need at least the version 2.3.2 but 2.4.1 is recommended.

- In Debian-based distributions (Ubuntu or Mint, e.g.): "*sudo apt-get install libopenbabel-dev*".
- In Fedora, as a super-user: "*dnf install openbabel-devel.x86_64*"
- In OpenSuse, as a super-user: "*zypper install libopenbabel-devel*"

3.2. Compiling OpenBabel on Linux

The other solution to obtain OpenBabel is to compile it, which is recommended since the last version (2.4.1) is faster than the previous ones for the 3mer screen. You will first need some packages (accept all the additional packages that the package manager will propose):

- In Debian-based distributions (Ubuntu or Mint, e.g.): "*sudo apt-get install g++ cmake libxml2-dev zlib1g-dev libcairo2-dev libeigen3-dev*".
- In Fedora, as a super-user: "*dnf install gcc-c++ cmake libxml2-devel.x86_64 zlib-devel.i686 cairo-devel.x86_64 eigen3-devel.noarch*"
- In OpenSuse, as a super-user: "*zypper install gcc-c++ cmake libxml2-devel cairo-devel eigen3-devel*"

In older versions, you may need to change the names, for example for old Debian-based distributions, replace "*libeigen3-dev wx3.0-headers libwxbase3.0-dev"* by *"libeigen2-dev wx2.8-headers libwxbase2.8-dev"*. You can then download OpenBabel (click on "Download 2.4.1 stable release" on http://openbabel.org/wiki/Category:Installation) and follow this procedure:

- *tar zxf openbabel-2.4.1.tar.gz*
- *cd openbabel-2.4.1/*
- *mkdir build*
- *cd build*
- *cmake .. -DCMAKE_INSTALL_PREFIX=~/Programs/Openbabel_2.4.1 -DBUILD_GUI=OFF*
- *make*
- *make install*
- *make test*

We recommend to use gcc as a compiler. If you are using version 5 or above, we recommend you specify "*ulimit -s unlimited*" in the command line before. If you want to use old versions of gcc for example on Ubuntu 16.04, you can install the package "*g++-4.9*" and then use "*cmake .. -DCMAKE_INSTALL_PREFIX=~/Programs/Openbabel_2.4.1/ -DCMAKE_C_COMPILER=/usr/bin/gcc-4.9 -DCMAKE_CXX_COMPILER=/usr/bin/g++-4.9 -DBUILD_GUI=OFF*".

If you want to use the development version of OpenBabel, you will need to modify the file include/openbabel/atom.h since some functions that we need were removed. In the class OBAtom, for example at line 404, add the following lines, and then compile OpenBabel as explained:

```
//! \return Is the atom hydrogen?
bool IsHydrogen()    { return(GetAtomicNum() == 1); }
bool IsHydrogen() const { return(GetAtomicNum() == 1); }
//! \return Is the atom carbon?
bool IsCarbon()      { return(GetAtomicNum() == 6); }
//! \return Is the atom nitrogen?
bool IsNitrogen()    { return(GetAtomicNum() == 7); }
//! \return Is the atom oxygen?
bool IsOxygen()      { return(GetAtomicNum() == 8); }
//! \return Is the atom sulfur?
bool IsSulfur()      { return(GetAtomicNum() == 16);}
//! \return Is the atom phosphorus?
bool IsPhosphorus()  { return(GetAtomicNum() == 15);}
//
bool IsNotCorH()
{
  switch(GetAtomicNum())
    {
    case 1:
```

```
        case 6:
          return(false);
        }
      return(true);
    }
```

3.3. Installation of OpenGrowth on Linux

To compile the programs by yourself, go to the "*Source*" folder and edit the Makefile file according to your needs. You need to choose the INCLUDE and LDFLAGS lines that corresponds to your system (by commenting and uncommenting the appropriate lines) and specify where OpenBabel is installed. Then simply type "*make*".

Eight programs will be compiled: 3MerScreen.exe, CenterOfMolecule.exe, FOG2.exe, KBP2016-Training.exe, OpenGrowth.exe, ProcessFragments.exe, SearchGUI.exe and SMoG2016.exe. We have only tested the gcc compiler. If you are using gcc5 or above (for example in Ubuntu 16.04 or more recent), you must specify "*ulimit -s unlimited*" in the command line before using the programs. If the compiler complains about "*unrecognized command line option '-std=c++11'*", it means that it is old. Try to replace "std=c++11" by "std=c++0x" in the Makefile (this happens in Ubuntu 12.04 for example). Once the programs have been compiled, go to Chapter 4 to see an example of use.

3.4. Windows

OpenGrowth can be used on Windows. There are three options: (1) in the last versions of Windows10™ there is an included Ubuntu bash terminal (Windows Subsystem for Linux of wsl), thus you can follow the instructions provided above. Internet will be a good resource to know how to install it. (2) You can install VirtualBox (https://www.virtualbox.org/) and then install LinuxMint (e.g.) on it (choose v17.3 to avoid errors in the compilation as stated above). From there, you can install OpenBabel from the repositories and then use OpenGrowth. CPU performances are comparable between the host system and a virtual machine. (3) You can use MinGW. However, we have not tested OpenGrowth with MinGW and **we don't provide any support**. We can't guaranty that the program will behave as expected.

3.5. MacOS

OpenGrowth can be compiled on MacOS since it is UNIX-based. However, please note that we have not tested it on this platform and **we don't provide any support for OpenGrowth on MacOS**. We don't guaranty that the program will behave as expected. An alternative easy way is to use VirtualBox (see above). If you want to try to use MacOS:

- You will need the gcc compilers and cmake. You can for example use homebrew or macports (*sudo port install cmake gcc49*, for example). For cmake you can also download the .dmg (http://www.cmake.org/download) and install it.
- After that, you can download the sources of OpenBabel and compile it. See the details given for Linux, the same procedure applies. For cmake, you can use "cmake .. -DCMAKE_INSTALL_PREFIX=~/Programs/Openbabel_2.4.1 -DBUILD_GUI=OFF -DCMAKE_C_COMPILER=gcc-mp-4.9 -DCMAKE_CXX_COMPILER=g++-mp-4.9"
- Finally, OpenGrowth can be compiled. Go in the *Source* folder, and edit the Makefile file to select the options for MacOS. You may need to replace the CC line by "CC =g++-mp-4.9". Then type "*make*". If for some reasons it complains, try first to add "-rdynamic" at the beginning of the LDFLAGS line. It depends on the version of MacOS and the compilers.

4. Example of use

To use the program, you need to download the file "*Resources_1.X.zip*" and unzip it. It is

provided as a separate file since it is less likely to be changed than OpenGrowth itself. You can copy the *OpenGrowth.exe* file compiled by yourself in the *Resources* folder once it is unzipped. Then, type "*./OpenGrowth.exe Input-OpenGrowth-Complete.inp*". It should start to grow ligands by using the protein file "*Structure/2AQU-A_XRay_0.pdb*" which is a structure of the HIV-1 protease. The ligands will be written in the "*Output*" folder that should be created automatically. If this didn't work, there is a problem with the executable file. If the program crashes ("*Segmentation fault (core dumped)*") right after the message "*Prepare 3mer file*", type "*ulimit -s unlimited*" in the command line and try again.

If this first example worked, you can then modify the input file and the parameters to suit your needs (see below). If you have problem later, they will come from the input files. We provide in the *Resources* folder two sets of fragments of different sizes (Fragments-OpenGrowth-113 and Fragments-OpenGrowth-413): the smaller one (113 fragments) corresponds to the fragments presented in the original publication of OpenGrowth, the larger one (413 fragments) corresponds to the fragments shown in Chapter 11.6 below. We also provide a folder that can be useful for random growth (Fragments-Random-113). To use OpenGrowth, you need: (1) one of these folders, (2) the files AmberAtomTypes.txt and VDWParameters.txt, (3) an input protein structure file (such as *Structure/2AQU-A_XRay_0.pdb)*, (4) a file such as KBP-3.0-5.0-8.5_Openbabel2.4.1.dat if you plan to use the SMoG2016 scoring function, (5) an input file (.inp) with all the parameters.

For now, OpenGrowth is not parallelized because OpenBabel is not thread-safe and thus we can't use OpenMP. However, this is not a problem and if you have access to multiple cores, you can use the following commands (for 4 cores) which run several instances of the program at the same time:

```
#!/bin/bash
./OpenGrowth.exe Input.inp > Input1.log &
./OpenGrowth.exe Input.inp > Input2.log &
./OpenGrowth.exe Input.inp > Input3.log &
./OpenGrowth.exe Input.inp > Input4.log
```

The way it works is the following: if you asked for 1000 ligands, each instance will start at 0 and check if a ligand called Ligand_0_0.xyz has been created (assuming the OUTPUTNAME is Ligand; we will explain later the last second "_0"). If yes, it goes to 1, then 2 and so on until it finds a number that has not yet been used (let's call it i). Once it is the case, this specific instance of the program writes an empty file called Ligand_i_0.xyz. The file Ligand_i_0.xyz will be seen by the other instances and thus no other Ligand_i_0.xyz files will be created. The program then tries to grow a ligand; if the growth is successfull something will be written in Ligand_i_0.xyz. If the growth is not successfull, the file Ligand_i_0.xyz will be removed and another instance (or the same) will try later to grow something in the Ligand_i_0.xyz file. This explains why in the output summary files, you may see *Ligand_11* after *Ligand_20* for example.

5. Usage of OpenGrowth

The input file (for example Input-OpenGrowth-Complete.inp) of OpenGrowth consists of lines with "`PARAMETERNAME parametervalue`". Only one parameter is hard-coded in the file OpenGrowth.h: `MAX_FRAGMENTS` is set by default to 500. This value must be higher than the total number of fragments used (which is the size of the file `FRAGMENT_LIST`). For technical reasons it is not possible to set this parameter as an option and assigning it a too large value would result in a waste of memory. Therefore, if you need to change it you will have to recompile the program. For the resources provided, the total number of fragments is 113 and 413 so you don't need to change the `MAX_FRAGMENTS` if you are using the list of fragments we provide. Otherwise, none of the parameters are hard-coded (except those for SMoG2001 to be sure that a scoring function is always available). `PARAMETERNAME` is case sensitive and only one `parametervalue` per line is read.

The name and the value of the parameters can be separated by spaces or tabulations. All parameters are mandatory, unless a default value is given below.

5.1. Receptor

PARAMETERNAME	Parametervalue
CONFORMERS	The protein conformers filename (absolute or relative path).
CONFORMERS_NUMBER	*Default value=0.* How many rotamers of the receptor will be used.
ROTAMERS	The protein rotamers filename (absolute or relative path).
ROTAMERS_NUMBER	*Default value=0.* How many rotamers of the receptor will be used.

At least one of the parameters "CONFORMERS" or "ROTAMERS" must be provided. Similarly, at least one of the parameters "CONFORMERS_NUMBER" or "ROTAMERS_NUMBER" must be different than 0. If you are using only one receptor, its name must end with "_0.pdb". If you are using several receptor structures, their filenames must end with "_0.pdb", "_1.pdb", "_2.pdb"… and as an input you only need to provide the name of the first one (which ends with "_0.pdb"). The program knows it has to remove "_0.pdb" and substitutes it with other numbers. Note that we have used the names "*Conformers*" and "*Rotamers*", but these structures don't need to be rotamers or conformers. "*Rotamers*" describes the first set of structures used for the growth, "*Conformers*" describes the second set (if there are two sets).

When both rotamers and conformers are provided, the program first grows ligands in the rotamers set; if the growth is successful (according to parameters explained below), the ligand is regrown in the conformers. Most of the time, growing directly in conformers (with several structures if needed) should be sufficient. Another way of using this option is to grow in one structure (provided in ROTAMERS) and then in several CONFORMERS (10 for example). Rotamers and conformers must have been previously aligned, since the BINDING_SITE coordinates are given only once for all the structures. To align snapshots, you can for example open all of them with UCSF Chimera and go in Tools/Structure Comparison/MatchMaker. Then "Save PDB…".

All the protein files must be complete: there should not be any missing residues, no alternative conformations and hydrogens must be present. If this is not the case, and you don't know how to do it, you can for example use the interface of Modeller provided in UCSF Chimera (Model/Refine Loops) with default options (model « non-terminal missing structures », generate 5 models, standard « loop modeling protocol ») and keep the structure with the lowest zDOPE score. Missing atoms and ligand hydrogen atoms can be added with the Dock Prep tool (delete solvent, keep highest occupancy, incomplete side chains with Dunbrack rotamers library, add hydrogens, don't add charges).

For now, water molecules are not taken into account and it is better to remove them from the structural input files. To include them in the growth, we would need to develop a scoring function that takes into account interactions between the ligand and the water. Another possibility is to only take into account Lennard-Jones repulsion to avoid overlap. Such options are on the roadmap but not yet available. To overcome the problem, you can try to use the seeding approach.

5.2. Energy

PARAMETERNAME	Parametervalue
PROTEIN_RANGE	*Default value=40*. To save memory, we can avoid the storage of the full protein. Protein atoms which are further than this value (in Å) from the BINDING_SITE will not be stored. If the BINDINGBOX_SIZE value is 0.5, if molecules are at most 8 Å long, and if the scoring function has a cut-off of 8.5 Å, a value of at least 17 must be used. If

	more than 1 snapshot is used, a few angstroms (~3-5) should be added to take into account the flexibility. With SMoG2016, a value of 20 is usually enough. Note that with some functions (such as SMoG2001), every time a new fragment is added we check if there are steric clashes between the protein and the ligand by calculating the distances between all atoms of the protein and all atoms of the ligand. If the protein is large, it may be time-consuming if all the atoms have been stored in memory. How many atoms are read is given in the output after the parameters, in the line "Preparation of <your-structure-name> (X atoms in total, Y kept) successful" to help you monitor the influence of changing the range. If you are using the MODE ENERG with proteins that are not aligned, you may want to assign to PROTEIN_RANGE a large value (1000 for example) to be sure that all atoms are read; this allows you to avoid changing the BINDING_SITE coordinates.
SCORING_FUNCTION	*Default value=SMOG2001*. The scoring function that will be used. For now, only SMOG2001 and SMOG2016 are supported, and we highly recommend the use of SMOG2106. For SMOG2016, 3 files are needed in the same folder as *OpenGrowth.exe*: AmberAtomTypes.txt, VDWParameters.txt and the one defined in ENERGY_FILE. They are provided in the *Resources* folder.
ENERGY_FILE	This file contains the energetic parameters for the scoring function that you have chosen. It is not needed for SMOG2001 since we have hard-coded the parameters to be sure that at least one scoring function is always available. For SMOG2016, it is the knowledge-based potential. Depending on the version of OpenBabel, different atomtypes are assigned, thus slightly different potentials are obtained; we provide in the *Resources* folder two files to use according to your needs (KBP-3.0-5.0-8.5_Openbabel2.X.Y.dat). If you plan to use your own scoring function, this file can be formatted in the way you want.

5.3. Active site

PARAMETERNAME	parametervalue
BINDING_SITE_X	The x coordinate of the binding site center (in Å).
BINDING_SITE_Y	The y coordinate of the binding site center (in Å).
BINDING_SITE_Z	The z coordinate of the binding site center (in Å).
BINDINGBOX_SIZE	*Default value=1.0.* When DENOVO is selected as a MODE, a point around BINDING_SITE is randomly chosen to put the first atom of the first fragment. This parameter value is the size (in Å) of each direction around the (x,y,z) which defines the box where the point can be put (with a value of 1.0, the box is 2.0*2.0*2.0 $Å^3$).

If one of the coordinates of the binding site is exactly 0.0, the program will stop (for technical reasons: it is due to a test made to check that all the binding site coordinates are given as input parameters). Should the case arise, change the parameter to 0.01. It will not change the science and it will allow the program to work.

We provide a utility to determine the center of the active site. If you have the structure of a complex with the ligand in the active site of the protein, you can extract the ligand and save it as

an .xyz file (you can convert a .pdb file to a .xyz with OpenBabel: "*obabel File.pdb –O File.xyz*"). Then, use "*./CenterOfMolecule.exe File.xyz*" which will provide the mean cartesian values of the ligand (*CenterOfMolecule.exe* is compiled at the same time as OpenGrowth). These values can then be used in the input file.

5.4. Growth

PARAMETERNAME	parametervalue
MODE	*Default value=DENOVO*. Three options are possible: DENOVO (starts the growth from a fragment randomly chosen and creates new molecules), SEED (grows from a given ligand already in place), ENERGY (computes only the interaction energy with a ligand already in place). When SEED is selected, the initial fragment must be properly formatted, see below. With SEED or ENERGY, the LIGAND parameter must be given and as many ligands as CONFORMERS_NUMBER must be defined.
GROWTH_MODE	*Default value=RANDOM*. Defines how fragments are connected. Four options are possible: RANDOM selects randomly the new fragment, see the comments below. BIASED will use PROBA_FIRSTFRAG to select both the first and every new fragments. FOG will use PROBA_FIRSTFRAG for the first fragment and PROBA_TRANSITION for the others. REGROW allows building known ligands and you must then provide an input file (REGROW_FILE).
LIGAND	Not needed if MODE is DENOVO. This parameter is the name of the ligand file to start the growth from (when SEED is selected), or to compute the energy with (when ENERGY is selected). With SEED or ENERGY, there must be as many ligands as CONFORMERS_NUMBER and their names must end with _0.xyz, _1.xyz, _2.xyz ... Formatting requirements are explained below.
REGROW_FILE	Needed when GROWTH_MODE is REGROW. This file contains all the numbers which are usually randomly chosen. See Chapter 13 that is devoted to this option. When the REGROW mode is used, the MAX_FRAGMENT parameter (not MAX_FRAGMENTS in the source) should be equal to the number of fragment you want to grow.
BRANCHING_PROBA	*Default value=0.5.* Defines how much the new ligands are branched.
MC_TEMP	*Default value=1.3*. The Monte Carlo temperature for the Metropolis criterion during the growth. This parameter is the RT value. Increasing the value will increase the chance of acceptation of the new fragment when the score doesn't decrease upon addition.
FRAGMENT_LIST	The name of the file which contains the list of fragments that will be used during the growth. It can be an absolute or a relative path. Before using the program, check that the files listed in this file point to the good path. For example, if you are using "Fragments-OpenGrowth-113/Fragments.dat" for FRAGMENT_LIST and all the .xyz files are in the "Fragments-OpenGrowth-113" folder, all the lines in "Fragments-OpenGrowth-113/Fragments.dat" should look like "./Fragments-OpenGrowth-113/Ring0_1_1.xyz".
PROBA_FIRSTFRAG	A file which describes the probability to find each fragment. It is used to choose the first fragment when MODE=DENOVO.

PROBA_TRANSITION	A 2 dimension array with the probability to connect one fragment to the others (one line per fragment). The transition file can be tuned: if you want to forbid the transition to the 12th fragment in the list FRAGMENT_LIST, you can set all the numbers in the 12th column to 0.0. Even if the sum of the probabilities in each line will no more be 1, the program will not fail (there is a loop to take care of this) and you will keep the relative probabilities for the other fragments.

When ENERGY is used, the input ligand file can be a regular .xyz file. This option can be used to have an idea of the score for known ligands. It is possible for example to run an MD simulation of the protein with a known ligand, extract snapshots, and use the ENERGY option on the different snapshots. Then, if you want to use the protein flexibility option, you can make an arithmetic or a Boltzmann average of the scores. If you are using ENERGY with proteins that are not aligned, you may want to assign to PROTEIN_RANGE a large value (1000 for example) to avoid having to change the BINDING_SITE coordinates. When SEED is used, the input ligand file must be formatted properly. It looks like a classical .xyz file. On the first line the size of the molecule and on the second line the name of the ligand. Then on each line: the type of atom, the x/y/z coordinates, and finally a number identifying the fragment that contains the atom (according to where the fragment is in FRAGMENT_LIST). We provide below the seed that was used in the original manuscript of OpenGrowth. When a "0" is given in the last column, no growth can be made from this atom. When the number is not "0", it is used to define the type of atom. The three atoms with an "11" are from a methyl group and methyl is the 11th fragment in our list (defined in FRAGMENT_LIST). The 4th fragment in our list is the carbonyl part of an amide and is used for two atoms.

Let's say you want to grow from an imidazole : you can "grep "Imidazole" *" in the Fragments-OpenGrowth-413 folder, and you will see that this fragment corresponds to Ring1_9_?.xyz. There are four different types of hydrogen in imidazole, so there are four fragments. You need to check which fragment you want ; if you want Ring1_9_2, you will see it is at the line 45 of Fragments.dat in the Fragments-OpenGrowth-413 folder, so you need to write 45 in your input seed. To prepare the seed we have first processed the structure 2AQU with the *Protein Preparation Wizard* of Maestro. We have then extracted the ligand, manually removed the atoms we didn't want, added the missing hydrogens with UCSF Chimera, saved as an .xyz file and added manually the last column.

```
38
AZV_Seed
C          64.24700        95.98900        30.65200     0
O          63.95200        94.83000        30.37100     0
N          63.37600        97.00700        30.59600     0
C          61.94400        96.92100        30.30300     0
C          61.10700        97.09400        31.59100     0
C          61.52400        96.18200        32.72800     0
C          61.05000        94.85500        32.77200     0
C          61.43900        94.00600        33.82300     0
C          62.31800        94.46800        34.82200     0
C          62.79000        95.79400        34.78000     0
C          62.39000        96.65100        33.73900     0
C          61.53600        97.96300        29.24700     0
O          61.91300        99.23600        29.70800     0
C          62.21600        97.83600        27.88500     0
N          61.81200        96.71600        27.03400     0
N          60.43100        96.72400        26.79200     0
C          59.65200        95.65400        27.03400     0
O          60.05300        94.63500        27.59500     0
```

```
C       62.57400        96.79300        25.77500        0
H       63.71200        97.92000        30.87000        0
H       61.71700        95.93600        29.89500        0
H       61.16000        98.12300        31.94900        0
H       60.05800        96.91600        31.36200        0
H       60.38800        94.48800        32.00200        0
H       61.06700        92.99400        33.85200        0
H       62.62300        93.81000        35.62200        0
H       63.45700        96.15300        35.54900        0
H       62.75900        97.66600        33.71300        0
H       60.45300        97.95800        29.12800        0
H       62.80400        99.40900        29.43400        0
H       62.02400        98.75200        27.32300        0
H       63.29500        97.77900        28.02800        0
H       60.05700        97.52800        26.30600        0
H       62.42700        97.77300        25.31500        11
H       63.63500        96.71500        26.00600        11
H       62.26900        96.00000        25.09200        11
H       65.26100        96.19900        30.95800        4
H       58.62000        95.69100        26.71800        4
```

In case you want to perform a random growth, you must use special fragment files. Let's take the example of pyridine: during growth with the FOG mode, pyridine is described three times to take into account the three different types of hydrogens. Thus, for each pyridine fragment, the growth from some types of hydrogens is forbidden. Consequently, if you want to perform a random growth, it is important to use fragments where the growth from all hydrogens is possible. Moreover, each fragment must appear only once in the `FRAGMENT_LIST` file. We provide a set of fragments formatted for a random growth in the "*Resources*" file (coming from the set with 113 fragments).

5.5. Optimization

Once a new fragment has been added to the current ligand, its geometry and position may not be optimal even after the rotameric search. For example, steric clashes may form (either internally, or between the protein and the ligand). The user can choose to optimize the ligand geometry in the active site of the protein (all protein atoms being kept fixed, see Figure 5). This process is performed using the OpenBabel library [4] and thus uses one of the built-in force fields. The user can choose the number of steps to be performed and the cut-offs for the interactions.

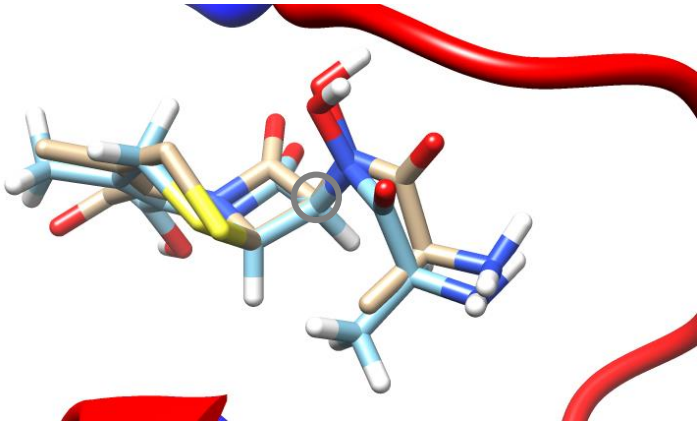
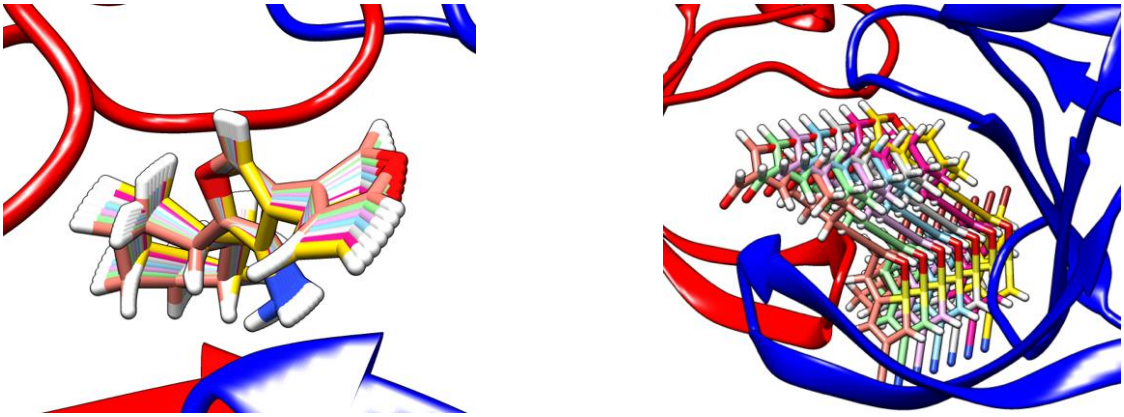


Figure 5. Geometry optimization of the ligand in the (fixed) active site of the protein. In brown the geometry before optimization, in blue after. One atom of the ligand is being kept fixed (in a gray circle): it is the atom that is the closest to the center of the active site defined by the user.

To further optimize the position of the ligand, the whole molecule can move within the active

site of the protein. When the ligand is moved, if the score in the new position is lower than the previous one (and no steric clashes are found), the new position is kept. We present in Figure 6 the superimposition of several moves for translational and rotational cases. The number of optimization steps (`OPTIMIZATION_STEEPDESC` and `OPTIMIZATION_CONJGRAD`) shouldn't be too high. We are doing these steps only to relax the ligand structure and to avoid strong clashes. If you are doing too many steps, you will end up in a minimum at the force field level whereas you want to reach a minimum at the scoring function level. Moreover, the MMFF94 force field will sometimes create strange structures, for example with bended phenyl rings. Since the goal of OpenGrowth is too generate many ligands, if you see a ligand with a strange structure, remove it.



(a) Rotation of the full ligand. (b) Translation of the full ligand.

Figure 6. Optimizing the ligand position as a whole.

PARAMETERNAME	Parametervalue
VDW_SCALE_INTER	*Default value=0.70*. The scale factor for intermolecular steric clashes. If the distance between a ligand atom and a protein atom is less than this value times the sum of their vdW radii, there is a steric clash and the ligand is discarded. It is a "hard wall potential". This value should not be too low (go below 0.6 only in extreme cases when nothing is growing otherwise).
VDW_SCALE_INTRA	*Default value=0.70*. The scale factor for intramolecular clashes.
ROTATION_PRECISION	*Default value=24*. How many rotations are made for each new fragment during the growth to find the best rotamer (see Figure 3).
OPTIMIZATION_MODE	*Default value=12.* Here two parameters are given in one: the first digit describes when the geometry optimization is done, the second digit describes when the optimization as a whole is done. Note that the geometry optimization is performed first to relax the structures. The options for the digits are: 0=no optimization is made; 1=an optimization is made once a ligand has finished growing; 2=an optimization is made after that each best rotamer has been found, 3=an optimization is made for each rotamer. For example, "12" means that the geometry of the ligand will be optimized when it has finished growing (i.e. when it has reached one of the thresholds) and the optimization as a whole will be done after each best rotamer has been found. A value of *22* is usually a good compromise.
OPTIMIZATION_NUMBER	*Default value=30*. For every round of position optimization, how many times the program will try to optimize the position of the whole ligand in the active site. Every time it will be either a

	translation or a rotation (randomly chosen) around a random axis. Note that moving the ligand is not time-consuming; if computing the score is also not time-consuming (which is the case for the SMoG family), you can increase this value without slowing the program too much.
OPTIMIZATION_ITERATIONS	*Default value=10*. The ligand will be moved this number of times by OPTIMIZATION_DISTANCE or OPTIMIZATION_ANGLE in *each* direction every time we are doing a optimization as a whole. The same comment as above concerning the speed of the program applies.
OPTIMIZATION_DISTANCE	*Default value=0.05*. The step (in Å) for the translation. This does not change the efficiency of the program, only how much you allow the molecule to move.
OPTIMIZATION_ANGLE	*Default value=1*. The angle (in degrees) for the rotation. This does not change the efficiency of the program, only how much you allow the molecule to move.

5.6. Force field for optimization

PARAMETERNAME	parametervalue
OPTIMIZATION_FORCEFIELD	*Default value=MMFF94.* Choses which force field is used to perform a geometry optimization. Options are: MMFF94, UFF, Ghemical, GAFF. Only the ligand is optimized, the receptor atoms are fixed. Note that the atom types of GAFF in OpenBabel may contain errors.
OPTIMIZATION_STEEPDESC	*Default value=100.* The number of steepest descent steps done.
OPTIMIZATION_CONJGRAD	*Default value=0.* The number of conjugate gradient steps done.
OPTIMIZATION_VDWCUTOFF	*Default value=7.0.* The cut-off for vdW interactions.
OPTIMIZATION_ELECCUTOFF	*Default value=15.0.* The cut-off for electrostatic interactions.

The parameters provided here are the ones that influence the speed of the program the most since the geometry optimization is the most time-consuming part. When using conjugate gradient, you may see errors printed to the screen (such as "`WARNING: damped steplength`"). Most of the time you can ignore these, they are due to the geometry optimization with OpenBabel.

When the first chosen fragment is hydroxyl, you may see the following: "`WARNING: damped steplength`". This can be safely ignored since it is only a warning from OpenBabel. When the first chosen fragment is a halide, you will very likely see "`ERROR: Could not setup force field`", the growth of this fragment will stop and a new growth will start. This can be ignored. To avoid it, you can edit the file *Proba_FirstFrag.dat* file and put 0.00000 at the lines defining halides. This only impacts the choice of the first fragment, not the transition probabilities so it won't change anything to the molecules properties. This disappears when halides are chosen later in the growth and not as the first fragment. To know which fragments are what, it is quite straightforward: each line in Fragments.dat (with the name of a file) correspond to a line with a probability in Proba_FirstFrag.dat. Then, on each file (Ring0_2_1.xyz for example), you will find the chemical names at the beginning (on the second line). For example: Ring0_2_1=fluorine, Ring0_7_1=chlorine, Ring0_10_1=bromine, Ring0_11_1=iodine. Here is an extract of the beginning of the *Proba_FirstFrag.dat* file with the name of the fragment files and their chemical names that were added:

```
0.11497049 Ring0_1    Hydroxyl
0.02205220 Ring0_2    Fluorine
0.00230765 Ring0_3    Cyano
```

```
0.00819452 Ring0_4    Aldehyde
0.00021193 Ring0_5    Nitroso
0.00580445 Ring0_6    Thiol
0.02081596 Ring0_7    Chlorine
0.00288456 Ring0_8    Nitro
0.00240184 Ring0_10   Bromine
0.00481546 Ring0_11   Iodine
0.51016760 Ring0_12   Methyl
0.04610594 Ring0_13   Amine
0.00128334 Ring0_15   Alkyne
```

What needs to be done here, is put 0.000 for Fluorine, Chlorine, Bromine and Iodine to avoid the problem "ERROR: Could not set up force field". To avoid " WARNING: damped steplength", put 0.000 for Hydroxyl.

5.7. Threshold

PARAMETERNAME	parametervalue
MAX_FRAGMENTS	Will stop the growth if this number of fragments is reached.
MAX_ATOMS	Will stop the growth if this number of heavy atoms is reached.
MAX_MW	Will stop the growth if this molecular weight is reached.
MAX_ITERATIONS	*Default value=20*. The max number of tries before starting again. For the first fragment, once a fragment has been chosen, the program will try to put it in the active site this number of times (every time at a random position with a random orientation). If it has failed this number of times (for example because the fragment is too large), a new fragment will be picked. For adding new fragments, the program will try to add a new fragment to the current ligand this number of times. If it fails (because of steric clashes for example), the ligand is saved and a new ligand is grown.
MIN_FRAGMENTS	*Default value=0*. If the final ligand has fewer fragments than this number, it will not be saved.
MIN_ATOMS	*Default value=5*. If the final ligand has fewer heavy atoms than this number, it will not be saved. This avoids molecules such as CH_3-I to be saved for example.
MIN_ENERGY	*Default value=0*. If the energy of the final ligand is higher than this number, it will not be saved. Note that we call it MIN_ENERGY because we are trying to decrease the energy, and we save ligand with energy **lower** than this value.

At least one of the parameters `MAX_FRAGMENT`, `MAX_ATOMS` or `MAX_MW` must be provided. When the `REGROW` mode is used, the `MAX_FRAGMENT` parameter (not `MAX_FRAGMENTS` in the source) should be equal to the number of fragments you want to grow.

5.8. Miscellaneous

PARAMETERNAME	Parametervalue
OUTPUT	*Default value="Ligand"*. The name of the output files.
SMILESONLY	*Default value=0.* If you want to only save the SMILES strings and not the 3D structures, use 1.
WRITEDESCRIPTION	*Default value=0.* If you want to regrow ligands in other target proteins later, use the value 1. It will write in the output the "description" of ligands, i.e. the sequence of numbers that were

	randomly chosen. You can use them with the REGROW option.
3MERSCREEN	If you want to perform the 3mer screen, this is the name of the file containing all the forbidden 3mers. If you are using a large set of fragments, it can be time-consuming and it is perhaps better to use this screen afterwards, see below.
AVERAGE_TYPE	*Default value=ARITHMETIC.* When several receptors are used, this defines the type of average made for the energy. It can be BOLTZMANN, ARITHMETIC or LOWESTSCORE. ARITHMETIC should be used if the structures of CONFORMERS are mutants of one protein or the same protein from different organisms. With LOWESTSCORE, no average is made: the interaction energy is the lowest score between all the structures. Note that the other ligands (i.e. the ones which don't have the lowest score) must still "fit" in the pocket: with SMoG2001, if there is a steric clash with one ligand after addition of a new fragment, the addition is always rejected. Thus, with LOWESTSCORE it is better to use SMoG2016 for example.
NUMBER_OUTPUT	*Default value=1000000.* How many output files will be created. Usually you don't need to change it: you will stop the program when you have generated enough ligands.
VERBOSE	*Default value=2.* How much information is given in standard output. Roughly: 1=info for fragments, 2=info from energy, 3=info from trials, 4=info from optimization. 5 and 6 are for debugging.

Let's say that you have asked for 1000 ligands with the name "*Ligand"* and the program is trying to grow the 11th ligand. OpenGrowth first checks if a file "*Ligand_10_0.xyz*" exists (numbering of ligands starts at 0). If it is the case, then it will increment the counter and check for "*Ligand_11_0.xyz*". If this file doesn't exist, OpenGrowth creates a file with this name and starts the growth. It the growth is successful, the molecule is written in the file. If the growth is not successful, the file is deleted. This means that during a future growth, the molecule *Ligand_11* may be created since there is no more file with this name. As a consequence, if you are running 16 instances of OpenGrowth from the same folder with the same output name (which is the easiest way to parallelize the program, see Chapter 4), in the summary files you may see *Ligand_11* after *Ligand_20* for example.

5.9. Minimum input file

We provide in the "*Resources*" file an input file with all the possible parameters. We also provide a minimal input file that is copied below:

```
CONFORMERS              Structure/2AQU-A_XRay_0.pdb
CONFORMERS_NUMBER       1
BINDING_SITE_X          61.9395
BINDING_SITE_Y          94.3025
BINDING_SITE_Z          27.8725
GROWTH_MODE             FOG
FRAGMENT_LIST           Fragments-OpenGrowth-413/Fragments.dat
PROBA_FIRSTFRAG         Fragments-OpenGrowth-413/Proba_FirstFrag.dat
PROBA_TRANSITION        Fragments-OpenGrowth-413/Proba_Transition.dat
MAX_MW                  450
```

5.10. Output files

Different output files are produced by the program. If the "`OUTPUT`" parameter is set to "*Output/Ligand*", you will find in the *Output* folder:

- A file *Ligand_Summary.smi* with: "SMILES_String Name Score" for each saved ligand.
- A file *Ligand_Summary.txt* with more information (Name, Score, Number of fragments, Number of heavy atoms, Molecular Weight, Ligand constraints, Description, SMILES string). The "Ligand constraint" is the difference of energy between the single point energy of the ligand in the active site of the protein and its optimized geometry. The calculation of the energy is made at the force field level according to the choice made in the input file (`OPTIMIZATION_FORCEFIELD`). For the optimization of geometry, 10 times more steps are performed than what is defined in the input file (if 100 steps of steepest descent and 0 steps of conjugate gradient are defined in the input file, respectively 1000 and 0 steps will be done to optimize the geometry). This value is not very accurate and is only provided for information. The higher the value, the more constrained the molecule is in the active site. What we call the "*Description*" is the sequence of numbers needed for the `REGROW_FILE`; see the paragraph devoted to the `REGROW` option below. This sequence of numbers is written only if `WRITEDESCRIPTION` is set to 1.
- If the 5th ligand that was grown satisfies the criteria to be saved (`MIN_FRAGMENTS`, `MIN_ATOMS` and `MIN_ENERGY`), a file called Ligand_4_0.xyz will be created with the 3D coordinates of the ligand in the active site of the protein (the first ligand is numbered with 0). If the growth is made in 5 conformers, the files Ligand_4_0.xyz, Ligand_4_1.xyz, Ligand_4_2.xyz, Ligand_4_3.xyz and Ligand_4_4.xyz will be created, one for each conformer.

6. 3mer screen

Growing molecules with the FOG algorithm is a two-state Markov chain that uses no information about connections to other fragments. Therefore, it is possible to connect three fragments together that are unlikely to be observed in that connected form in drugs, because of synthetic difficulties, stability, or other reasons. For example, a $-CH_2-CH_2-Br$ moiety is never found in our drug library. The 3mer-screen (implemented in OpenGrowth) consists of first creating all the possible 3mers from the fragment library; the drug library is then screened for those 3mers, and we retain in a list the ones that are never found. The user can also manually add 3mers, such as acetals or ketals, which are known to be unstable. The result is a list of "forbidden 3mers" that we call "*unfound_3mers.dat*"

The 3mer screen is thus a sorting stage that is applied to grown ligands. To use it during the growth, you need to use the keyword "`3MERSCREEN`" in the input file and provide the name of the list of the forbidden 3mers (for example *"Fragments-OpenGrowth-413/unfound_3mers.dat"*). However this can be time-consuming if you have selected a lot of fragments (the more fragments, the more forbidden fragments are possible). For example, on a standard desktop computer, it takes ~10s/ligand with 425,000 3mers to search (from *Fragments-OpenGrowth-113*) and ~1mn40s/ligand with 3,875,000 3mers (from *Fragments-OpenGrowth-413*). Thus, we provide in the "*OpenGrowth*" package a stand-alone program to do the same if you want to apply this screen only to the best output molecules. When you have compiled OpenGrowth, the program *3MerScreen.exe* has also been compiled. Move the program to the right folder and use it with "*./3MerScreen.exe File.smi unfound_3mers.dat*". It will output: "`TestMolecule FAIL`" or "`TestMolecule PASS`" depending on the case, where TestMolecule is the name of each molecule in the SMILES file. As mentioned above, if OpenGrowth has been compiled with gcc5.0 or above, you must specify "*ulimit -s unlimited*" in the command line before using it.

7. FOG2.0

It is possible to grow libraries of molecules outside the active site of a protein with FOG2.0. These libraries can then be docked with usual means. FOG2.0 performs the same task as FOG [3], however it uses the structure of OpenGrowth and input files are compatible between the two

programs. When using FOG2.0, you can choose the option "`SMILESONLY 1`" in the input file to only write the SMILES strings and not the .xyz files with 3D coordinates. When you have compiled OpenGrowth, *FOG2.exe* has also been compiled. An example of input file is provided in the "*Resources*" file with all the possible options. The use of options is the same as for OpenGrowth.

8. Knowledge-based training

You may want to prepare your own knowledge-based potential, for example to make it specific to a given family of proteins. You can use the program *KBP2016-Training.exe* that is compiled in the same time as OpenGrowth. The way it works is: "*./KBP2016-Training.exe 3.0 5.0 8.5 Training-Set.txt Library/*" if you want three shells that end at 3.0, 5.0 and 8.5Å. Training-List.txt is a text file that contains the PDB ID of your training set. We provide in the *Resources/Miscellaneous* folder the list that was used to develop SMoG2016 [6] (1038 complexes). Library is a directory where each complex has its own folder. In each folder, you need to have a file for the protein and a file for the ligand (for example, in the 10gs folder you need 10gs_protein.pdb and 10gs_ligand.sdf). We have used the PDBBind-CN database and have kept their structure. Preparing a potential takes roughly 10 minutes.

If you have compiled OpenGrowth with gcc5.0 or above, you must specify "*ulimit -s unlimited*" in the command line before. Note that depending on your compiler and your version of OpenBabel, you will find slightly different potentials (because the atom typing has evolved for example). However, when the energy is computed, these differences are minor and don't affect the correlation with experiments (R ranges from 0.56 to 0.57 on the testing set of 195 complexes developed by Li *et al.* [8]).

9. SMoG2016

If you only want to score a protein-ligand energy, you can use SMoG2016 with: "*./SMoG2016.exe Protein.pdb Protein.sdf DeltaG*" (different format can be used for the ligand, such as sdf or mol2), where DeltaG is the experimental value. The file SMoG2016.exe is compiled in the same time as OpenGrowth. The purpose of giving DeltaG in the input is to display both the score and the experimental value in the output to directly check the correlation. If you don't know DeltaG, write anything instead ("DG" for example). A script called Run_Energy.sh is provided in the *Resources/Miscellaneous* folder. It uses the file INDEX_general_PL_data.2014 coming from the PDBBinding-CN database.

Three files are needed to use this program: AmberAtomTypes.txt, VDWParameters.txt and a file for the knowledge-based potential. Its name must be "KBP-3.0-5.0-8.5.dat" since it is hard-coded. If you want to change the name, you need to edit SMoG2016.cpp and recompile it. Similarly to OpenGrowth, AmberAtomTypes.txt and VDWParameters.txt are used to assigned the Lennard-Jones parameters to calculate the repulsion part of the function. They are coming from the Amber99SB force field, and if they are not present, the program can't work. They are provided in the *Resources*.

10. OpenGrowthGUI

To prepare more easily the files needed by OpenGrowth or FOG2.0, we have prepared a Graphical User Inferface (GUI). **This interface does not grow any molecules**, it only automates the creation of files necessary to run OpenGrowth. It does not need to have OpenGrowth or FOG2.0 installed on the computer where it is run.

10.1. Installation

Linux

You can find in the folder "*OpenGrowthGUI*" pre-compiled binaries for three versions of

Ubuntu (14.04, 16.04 and 17.04). For LinuxMint 17.1 you can use the version for Ubuntu 14.04. For LinuxMint 18.1 and Fedora 25, the three versions should work. It is likely that nothing will happen by only double-clicking on the programs, you need to open them in command line ("*./OpenGrowthGUI_Ubuntu16.04.exe*" for example). If nothing happens, you will need to install Qt and compile the interface. If you don't see any molecules after clicking on "*Update*" but only numbers in the buttons, you also need to compile the GUI by yourself. See Appendix 17.1.

For the GUI to work, the program *SearchGUI.exe* must be located in the same folder as *OpenGrowthGUI.exe*. When you have compiled OpenGrowth by yourself, the program *SearchGUI.exe* has also been compiled. You must place *SearchGUI.exe* in the same folder as the GUI executable file *OpenGrowthGUI.exe*. To check if the binary files are working, the best way is to prepare a small library (10 drugs for example) and use the GUI (see "*Usage*" below). If every occurrence is 0, it didn't work. It after clicking on "*Update*" the message on the bottom left corner stays at "*Scanning fragment: 1 out of 99*" for more than 2 minutes, then it means that OpenGrowthGUI can't find the file *SearchGUI.exe* or it is not working. Should the case arise, you need to recompile *SearchGUI.exe*.

MacOS

The MacOS version of the GUI can be found in "*OpenGrowthGUI*" under the name *OpenGrowthGUI_MacOS.exe.app* (you may not see the .app extension). The same content as for Linux applies concerning *SearchGUI.exe*: you will have to compile the programs by yourself, and place *SearchGUI.exe* in the same folder as *OpenGrowthGUI*. If you double-click on the .app file it should open; if it doesn't work, you will have to compile the GUI (see Appendix 17.1).

Windows

The Windows version can be found in "*OpenGrowthGUI/Windows*". The GUI also needs OpenBabel to be installed, and the SearchGUI.exe file that is provided was compiled with OpenBabel 2.3.2. Thus, to ensure compatibility we provide the installer in the current package (*OpenBabel2.3.2a_Windows_Installer.exe*). You can then open *OpenGrowthGUI.exe* by double-clicking on the file. We have tested this on Windows XP, Seven, 8.1 and 10.

10.2. Usage

Some of the resources to use the GUI are provided in the "*Resources_1.X.zip*" file that is available as a separate file. You will have to download and unzip it. We recall that the GUI doesn't run OpenGrowth; it only prepares files needed for OpenGrowth. The interface is presented in Figure 7 and a general usage of the GUI is presented here: http://youtu.be/gaIFuXlm_rM.

When you open the GUI, start by selecting the "*Resource Directory*" folder: it is the folder that contains the file "*Fragments.dat*" and a lot of subfolders for each fragment (the one called "*Fragments-GUI-413*" in the "*Resources*" folder). Then, choose the "*Output*" folder (if no output directory is selected, the default directory will be a folder "*Output*" located where the "*Resource Directory*" is). Depending on the OS, the GUI may or may not be able to create a folder. If it fails, you will have to select an *Output* folder that already exists. It is recommended to choose an output folder that doesn't require elevated permissions. Finally, select the database file which must be a list of SMILES strings and use the .smi extension. We provide in the package (*Resources/Miscellaneous*) our drug database (8,018 molecules from ChEMBL). We recommend that you start by creating a subset of it during your first tests of the GUI ("*head Miscellaneous/ChEMBL_Subset-8018.smi > Test.smi*").

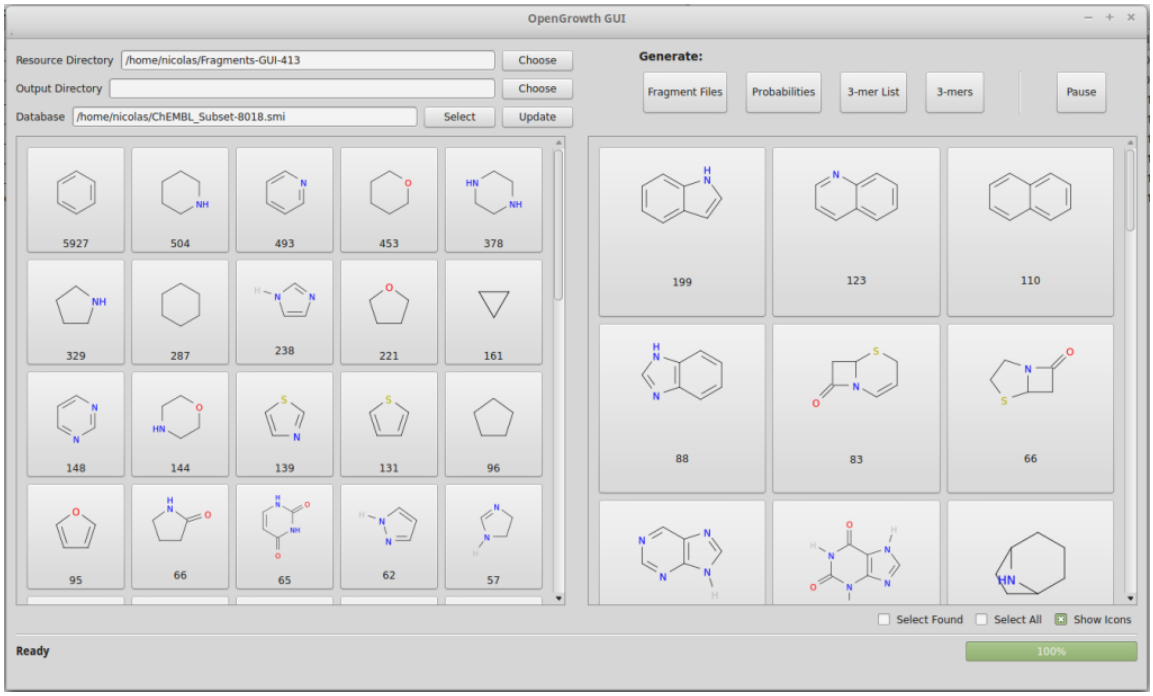


Figure 7. OpenGrowthGUI.

Reading the drug database

When you click on "*Update*", the program counts how many times each fragment listed in the file "*Fragments.dat*" (that is in the *Resource Directory*) is found. If you want to remove some fragments you can start by editing this file. Since they are identified by name, you will have to go in the subfolder to know which one to remove. The program will then display the occurrences (i.e. number of times that a particular fragment is found in the selected database) of each fragment by decreasing order: non-ring fragments are not displayed because most of the time they will all be kept, 1-ring fragments are displayed on the left and 2-ring fragments are displayed on the right (see in Figure 7). If .svg icons of the fragments can be found in the fragment folders, images will appear. Otherwise, only the name of the fragment will be displayed. If no images and no names appear, but only occurences, it means you should recompile the GUI by yourself. Each fragment is a clickable button and the user can then select which fragment to keep. The "*Select All*" box will keep all fragments and the "*Select found*" will keep fragments found at least once in the input database. Then, you must click on the "*Fragment Files*" button to output some files saying which fragments you have kept.

Choosing the fragments

The "*Fragment Files*" button will create several files in the output directory:

1. A file called "*selected_fragments.dat*" containing the names of all selected fragments.
2. A file called "*Fragments.dat*" with the relative path to the xyz file of each selected fragment.
3. One .xyz file for each fragment that was selected.

In order to proceed from this step, the user must have at least the "*selected_fragments.dat*" file in the chosen output directory (or in the default *Output* folder). This file can be manually adjusted to contain whatever fragments the user requires—one can add or delete fragments from this list. If certain non-ring fragments are not desired, the user can delete these entries to exclude them in further stages. Each line in this file must meet two criteria:

1. It must be listed as a fragment in the "*Fragments.dat*" file in the "*Fragments*" folder of the *Resource Directory*.

2. It must have a (same-named) folder in the *Resource directory*. For example, if you want to add a fragment called "*Ring2_132_2*", and the *Resource directory* is called "*Fragments*", the folder "*Fragments/Ring2_132_2*" should contain (at least) the files "*Ring2_132_2.smi*", "*Ring2_132_2.smarts*", and "*Ring2_132_2.xyz*". Contents of these files are described later in this documentation.

Calculating the probabilities

The "*Probabilities*" button reads the "*selected_fragments.dat*" file and creates the probability files (*Proba_FirstFrag.dat* and *Proba_Transition.dat*). This can be time-consuming depending on the number of fragments and the size of the database. See the "*Parallelization*" section of this documentation if you wish to run this process on external CPUs. If you wish to pause this process and continue it later, press the "*Pause*" button on the top right corner of the window. This will signal the program to pause at a convenient breaking point: the message box in the bottom left corner will display "*Paused!*" when it is safe to close the program. It may take 2 to 3mn for the program to be ready to pause depending on the size of the database and the number of fragments. When restarting, make sure that you have loaded the fragments again (by pressing "*Update"* after choosing *Resource Directory* and *Database*). There is no need to create the fragment files again, instead press "*Probabilities*" again to restart the process where it left off. At the end, one should have files "*Proba_FirstFrag.dat*" and "*Proba_Transition.dat*" in the selected output folder, with line number equal to the number of selected fragments.

Looking for forbidden 3mers

If one wishes to compute and output the list of unfound_3mers, one can press the "*3-mers*" button. Again, this will likely be a lengthy process, which can be paused if needed. When paused, a file called *3mers_left* will be created, to mark progress. This file will be deleted when the process is finished. If one wishes to do the computational work on another machine, one can press the "*3-mer List*" button to output a list of all the possible 3mers that could potentially be present in the database (no calculations are involved). This file will be called "*3mer_to_search.dat*".

Both of these processes involving 3mers require the presence of "*selected_fragments.dat*" and "*Proba_Transition.dat*". Transition probabilities are used to determine the 3mers_to_search—if a 3-mer is described as A-B-C, only 3-mers with nonzero A-B transition probability will be included in 3mers_to_search and/or unfound_3mers.

Pausing

One can use the pause button during two processes—generating probabilities and generating (unfound) 3-mers. When pausing during probability generation, the *SearchGUI.exe* program will stop execution after the completion of a line of the Proba_Transition file, which corresponds to a set of probabilities from one fragment to all the others. Thus, it may take a few minutes. Upon restarting (by pressing "*Probabilities*" again), the program will continue searching, starting from the next fragment. If the program is closed in the meantime, one will need to choose a resource directory, output directory, and database again, and press "*Update*" to load these parameters and load the fragments. Make sure that the correct output directory is chosen, or the program will not restart correctly.

When pausing during the 3mer scan, the *SearchGUI.exe* program will stop after the current 3mer being searched. Upon pausing, the 3mers_left file will contain the subset of 3mers_to_search that still needs to be searched. Upon restarting (by pressing the "*3mers*" button), the program will look for a 3mers left file, and continue from there, if found. If not found, the program will start from the beginning, so again make sure that the correct output directory is chosen.

10.3. Output

OpenGrowthGUI will create up to five .dat files in the output directory and .xyz files:

- *selected_fragments.dat*: this file contains the names of all selected fragments. It is only needed by the GUI and not by OpenGrowth.
- *Fragments.dat*: this file gives the relative path to the xyz files of the fragments that will be used by OpenGrowth or FOG2.0 (use with `FRAGMENT_LIST`). If you gather all the files in a folder "*Fragments-New*", each line must point to files in this folder. It can be a relative or absolute path, for example *"./Fragments-New/Ring0_1_1.xyz*". You will probably have to edit this file to fit your needs.
- *Proba_FirstFrag.dat*: this file describes the probability to find each fragment. It is used to choose the first fragment when *de novo* growth is used (use with `PROBA_FIRSTFRAG`).
- *Proba_Transition.dat*: this file is a 2D array describing the connection probabilities between the fragments (use with `PROBA_TRANSITION`).
- *unfound_3mers.dat*: this file is the list of the 3Mers that have not been found in the drug database (use with `3MERSCREEN`).
- All the xyz coordinates of the selected fragments will be placed in the output directory.

Examples of such files are available in the folders "*Fragments-OpenGrowth-XXX*". The "*-113*" one has been created with the 8,018 drug database and all the fragments shown in the original article. The "*-413*" one has been created with the same database and with the fragments presented below in Chapter 11.6.

Once the file "*unfound_3mers.dat*" has been created, it can be adapted by the user. Specifically, you can add 3mer that you want to forbid, such as acetal or ketals. We automatically add the following patterns to the file:

```
[O;X2;!$(OC(=O));!$(OP(=O))]-[C;X4]-[O;X2;!$(OC(=O));!$(OP(=O))]
[O;X2;!$(OC(=O));!$(OP(=O))]-[C;X4]-[N;X3;!$(NS(=O)(=O));!$(NC(=O))]
[N;X3;!$(NS(=O)(=O));!$(NC(=O))]-[C;X4]-[N;X3;!$(NS(=O)(=O));!$(NC(=O))]
```

10.4. Parallelization

SearchGUI.exe, as an external executable called by OpenGrowthGUI, can be used as a separate program. This flexibility allows a user to perform computationally heavy tasks on external CPUs. *SearchGUI.exe* has three functions:

- Searching a database for a given fragment.
- Computing and outputing *Proba_FirstFrag.dat* and *Proba_Transition.dat*.
- Computing and outputing *unfound_threemers.dat*.

To choose which function to perform, the first argument in command line must be a character corresponding to the desired function:

- For searching for a particular **f**ragment, the character is '*f*' (unlikely to be used outside GUI).
- For calculating **p**robabilities, the character is '*p*'.
- For calculating probabilities in a **s**plitted mode, the character is '*s*'.
- For searching for unfound **t**hreemers, the character is '*t*'.

Calculating probabilities

The following command will use the database "*~/ChEMBL_Subset-8018.smi*" and compute the probabilities for the fragments in the folder "*~/Fragments/*" and will write them in "*~/Output/*". In the *Fragments* folder, there must be a *Fragments.dat* file. This will create *Proba_FirstFrag.dat* and *Proba_Transition.dat*:

```
./SearchGUI.exe p ~/ChEMBL_Subset-8018.smi ~/Fragments/ ~/Output/
```

Calculating probabilities in splitted mode

The previous command will work sequentially one fragment by one fragment, which can be time consuming. You can use the '*s*' mode of *SearchGUI.exe* in the following way:

```
./SearchGUI.exe s ~/ChEMBL_Subset-8018.smi ~/Fragments/ ~/Output/ x
```

Here, the program will compute the probabilities for the fragment on the x^{th} line (starting at 0) and create a file called *Proba_Transition_x.dat*. If you have selected 413 fragments, it means that you can run 413 calculations at the same time with *x* going from 0 to 412. You must then merge all the *Proba_Transition_x.dat* files (take care to do that in the right order, for example 0/1/2/3/… and not 0/1/10/100/… as it can happen) to create the final *Proba_Transition.dat* file. Note that this command will not create *Proba_FirstFrag.dat* and you need to prepare this file with the regular mode ('*p*'). *Proba_FirstFrag.dat* is the first file that is created, so you can run *SearchGUI.exe* with the '*p*' option (or from the GUI) and as soon as you see that it has created a file called *Proba_Transition.dat* you can cancel it since it means that *Proba_FirstFrag.dat* has been created.

Searching for unfound 3-mers

If you want to parallelize the 3mer computation, the easiest way is to split the list of *3mers_to_search.dat* (created by the button "*3-Mer List*") into multiple files (as many cores as you want to run on) and place them in different folders. Then, start *SearchGUI.exe* on each core from a different folder with the command below. After, you only need to join the unfound_3mers lists (the order is not important here).

```
./SearchGUI.exe t ~/ChEMBL_Subset-8018.smi ~/Fragments/ ~/Output/
```

11. Building new fragments

11.1. Introduction

We provide in the file "*BuildingFragments_1.0.1.zip*" some resources to create your own fragments. What you want to prepare is something similar to the file "*PrepareFragments.dat*" and the folder "*XYZ*". Once you have these, you can run the script "*Run_02-PrepareFragments.sh*"; if everything was carefully prepared it will create a folder called "*Fragments-GUI*" that can be used with OpenGrowthGUI. If you run this script with the files provided in the package, the folder that will be created will be the same as the one provided in the "*Resources*" folder with the name *Fragments-GUI-413*. The "*PrepareFragments.dat*" file is a text file that lists all the fragments (pyridine (e.g.) will have to be described three times in this file because it has three types of different hydrogens). This file is made with six columns: SMILES string, name, molecular weight, SMARTS pattern with linked atom on the right, SMARTS pattern with linked atom on the left, SMARTS pattern with linked atom on the left and with marked positions for the 3Mer. We will give more details below. The XYZ folder contains optimized structures for each fragment in the .xyz format, with an additional column describing if a connection can be made to this atom.

Before going further, please note that you must have a good understanding of SMILES strings and SMARTS patterns. Several resources are available online. An introduction on SMARTS is available at the following address: http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html. We only provide here a quick explanation for an atom with the pattern [c;X3;R1;!$(c=O)]: this will be an aromatic carbon ("c"), with three neighbors ("X3"), part of only one ring ("R1") and not of the type 'c=O' ("!$(c=O)") (this is this last information which allows us to make the difference between cyclopentane and cyclopentanone, see below). The semi-colon (";") means "AND".

11.2. 3- to 11-fused rings

As explained in the original article, we have extracted all the rings from the ChEMBL drug

library by cutting all single bonds between rings and side chains. We can then assume that the rings that were found represent all the rings that can be found in drugs. We classified the rings according to the number of rings fused together per fragment:

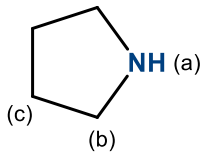
- mono-ring: 280
- 2-fused-rings: 444
- 3-fused-rings: 361
- 4-fused-rings: 171
- 5-fused-rings: 77
- 6-fused-rings: 28
- 7-fusedrings: 8
- 8-fused-rings: 6
- 9-fused-rings: 5
- 11-fused-rings: 3

In our definition, all double bonds connected to a ring are a part of the ring fragment (since we have cut single bonds to prepare the set of rings and we create single bonds during the growth). We provide in the folder "*List-Of-Rings*" all the found rings. We didn't prepare the SMARTS patterns for 3-fused to 11-fused rings. For these structures, in the corresponding files (Ring3.smi for example) you will find for each fragment: SMILES string, name, molecular weight, occurrence. Occurrence is how many times each fragment was found in the ChEMBL_Subset-8018.smi database. The molecular weight was computing with "*cxcalc mass*" from ChemAxon. The names are given in weight-order: we have ranked fragments by increasing molecular weight and then named them starting by 1. Technical detail: the files were formatted with a text editor where "Tab" is 8-character long.

11.3. Mono-rings and 2-fused rings

For mono-rings and 2-fused-rings, we have prepared a SMARTS pattern which should correspond exactly to each fragment (it is written in the files Ring1.smi and Ring2.smi). For example, when counting for the pattern of cyclopentane in a database, cyclopentanone fragments should not be counted because the two are considered to be different fragments. We have prepared all the fragments (mono-rings and 2-fused rings) that were found more than 10 times in the ChEMBL library (respectively 58 and 44 fragments). If you want to do it by yourself on a lot of substructures, note that this process can be time-consuming and you should first start with a simple case.

The easiest is to work with an example. Let us consider pyrrolidine, which is represented here. Analysis of the structure of pyrrolidine shows us that three types of hydrogens can be found (labelled (a), (b) and (c)); thus, a fragment can be bound to pyrrolidine at three different positions. In Ring1.smi, pyrrolidine is in the 6th line with the data:



C1CCCN1 Ring1_31 71.121 329 [C;X4;R1]1[C;X4;R1][C;X4;R1][C;X4;R1][N;R1]1

In our definition, this SMILES string corresponds to the position (a) since the connection point (the nitrogen) is the last atom in the string. The provided SMARTS matches the SMILES string regarding atom order. This description of pyrrolidine will be called Ring1_31_1. We must then prepare the descriptions for the two other possible connections ((b) and (c)) that will be called Ring1_31_2 and Ring1_31_3. We need to make a permutation of the atoms: Ring1_31_2/(b) can be described as C1CCNC1 and Ring1_31_3/(c) can be described as C1CNCC1. Similarly, the SMARTS patterns can be made by permutation. Thus, we have:

C1CCCN1 Ring1_31_1 71.1210 [C;X4;R1]1[C;X4;R1][C;X4;R1][C;X4;R1][N;R1]1
C1CCNC1 Ring1_31_2 71.1210 [C;X4;R1]1[C;X4;R1][C;X4;R1][N;R1][C;X4;R1]1
C1CNCC1 Ring1_31_3 71.1210 [C;X4;R1]1[C;X4;R1][N;R1][C;X4;R1][C;X4;R1]1

These SMARTS patterns are the ones we call "with linked atom on the right". We now need to prepare the ones called "with linked atom on the left". The reason for this is explained in the Appendix (Operation details of OpenGrowthGUI/3-mer List) and is linked to the preparation of the 3mers. We will only give the example for the SMARTS pattern of Ring1_31_1 and we will color each atom to make it easier to understand: [C;X4;R1]1[C;X4;R1][C;X4;R1][C;X4;R1][N;R1]1. To prepare the SMARTS pattern "with linked atom on the left", we only need to mirror it:

[N;R1]1[C;X4;R1][C;X4;R1][C;X4;R1][C;X4;R1]1. The nitrogen is now the first atom. This is a simple case, you will sometimes have to change things a little bit more, especially when atoms are "outside" the ring (for example the oxygen of cyclopentanone). The best for someone who wants to do it by himself/herself is probably to try to reproduce patterns from fragments that we have already prepared. To avoid any problems, we have preferred to label the rings with indices 1 and 2 for the SMARTS patterns "with linked atom on the right" and with indices 3 and 4 for SMARTS patterns "with linked atom on the left": [N;R1]3[C;X4;R1][C;X4;R1][C;X4;R1][C;X4;R1]3. Since we have only one ring here, we only use the indices 1 ("on the right") and 3 ("on the left"). For 2-fused rings, the other indices will be used. If we want to make a dimer of pyrrolidine where the connection is made through the two nitrogen atoms, we only need to join the two SMARTS pattern "with linked atom on the right" and "with linked atom on the left". The preparation of these patterns has to be done for each fragment.

Finally, we must prepare a last pattern for the creation of 3Mers, and we will edit the SMARTS pattern "with linked atom on the left": [N;R1]5[C;X4;R1][C;X4;R1][C;X4;R1][C;X4;R1]5 (we use indices 5 and 6 for this last pattern). We need to mark where other fragments can be added after the connection has been made *via* the nitrogen. Here, one fragment can be put at the carbon atom labelled (b) and one can put at the carbon atom labelled (c). We label these positions with (y): [N;R1]5[C;X4;R1][C;X4;R1](y)[C;X4;R1][C;X4;R1]5(y). This is the SMARTS pattern "with linked atom on the left and with marked positions for the 3Mer". All these data can be then merged in the file called "*PrepareFragments.dat*".

11.4. 3D structures

We must now construct the 3D structure of each fragment. We start by creating a first guess from the SMILES string and then we use a quantum chemistry program (such as Gaussian or GAMESS for example) to optimize the geometry of the fragment. We have prepared the script "*Run_01-Create-3D.sh*" that performs the following (you should adapt it to your needs):

- Extract the SMILES string of all the fragments in *PrepareFragments.dat*.
- Use OpenBabel to convert them to 3D (you must have installed OpenBabel before).
- Prepare the following input files for Gaussian:

```
%Nprocshared=16
%mem=10000MB
#P M062X/aug-cc-pVTZ Opt SCF=(MaxCycle=500)
# SCRF Test Units(Ang,Deg)

Name

0 1
(XYZ)
```

  (Note that we have tried to automatize the calculations of the charge, but it may be wrong. Similarly, OpenBabel may fail in the protonation state. So you should check each fragment structure.)
- Clean the files to only optimize the fragments called "_1": the others, such as "_2"/"_3"/… are different only by where the connection is made. Thus, the geometries are the same.
- Start Gaussian09 and extract the optimized geometry.

For the example of pyrrolidine, the optimized geometry is:

```
15
Ring1_31_1
 C    -0.804605    -0.969417    -0.183108
 C    -1.177783     0.427922     0.277618
```

```
C    -0.003515     1.267259    -0.218816
C     1.208559     0.407518     0.112822
N     0.696184    -1.022056     0.019861
H    -0.983460    -1.099182    -1.246444
H    -1.262495    -1.783214     0.368151
H    -1.244621     0.464190     1.364927
H    -2.131890     0.736590    -0.139731
H     0.063863     2.237207     0.265317
H    -0.077383     1.420860    -1.294594
H     1.544020     0.556811     1.133894
H     2.046442     0.519892    -0.565340
H     0.921075    -1.528858     0.873272
H     1.145949    -1.527632    -0.739907
```

Once the geometries have been optimized, the fragments need to be manually processed. What we first do is group all the optimized .xyz files in a folder "*XYZ-Optimized*". Then, we change their names in the corresponding files. This is not mandatory. To find the names, a good resource is at the following address: http://zinc.docking.org/browse/rings/?page=1&type=simple. If you can't find the names, the SMILES string is usually more interesting to have than the name of the fragment such as "Ring1_31_1". You then need to open the .xyz files and change the second line. Naming a fragment can be useful since when a fragment is added to a ligand by OpenGrowth, its name is displayed. However, as said before, it is not mandatory.

We then group all the files in a folder "*XYZ-Renamed*" and use the program "*ProcessFragments.exe*" (available when you have compiled OpenGrowth). This program will place the first atom at (0,0,0), the second atom in the x axis, the third atom in the xy plan. It will also right a 5th column for all atoms with a "0" in it. Moving and reorienting the fragments is not mandatory (OpenGrowth moves the fragments by itself where they need to be moved) but we prefer to do it. Adding the 5th column could also be done by hand. To use the program, type "*./ProcessFragments.exe 1 File1.xyz"*. It will output *File1_New.xyz*. This program can perform two tasks and the first one is moving molecules, hence the first parameter ("*1*"). You can process as many files as you want at the same time with "*./ProcessFragments.exe 1 File1.xyz File2.xyz File3.xyz"* (for example). The file *Ring1_31_1_New.xyz* (for pyrrolidine) is shown below:

```
15
Pyrrolidine
C     0.000000   0.000000   0.000000   0
C     1.517922   -0.000000  0.000000   0
C     1.851211   1.489549   0.000000   0
C     0.862442   2.071334   -1.001061  0
N     -0.355817  1.165437   -0.900825  0
H     -0.398234  0.229373   0.984027   0
H     -0.469258  -0.898259  -0.385673  0
H     1.897765   -0.479622  -0.902057  0
H     1.909959   -0.526855  0.865163   0
H     2.874489   1.698905   -0.297989  0
H     1.684246   1.913760   0.989200   0
H     1.227323   1.997972   -2.020321  0
H     0.554058   3.090208   -0.797651  0
H     -0.618617  0.831532   -1.825583  0
H     -1.162413  1.668811   -0.538662  0
```

Finally, you need to edit the "0" of the last column by a number for each fragment. For heavy atoms, keep "0". If there is only one type of hydrogen, add "1" for all the hydrogens where a growth can happen. Please note that if an amine is protonated, one of the hydrogens connected to the nitrogen

must always have a “0” in the last column even if the connection point is the nitrogen: we don’t want to grow quaternary ammonium. If you have different types of hydrogens, use “1” for the first type, “2” for the second type, “3” for the third, etc… Then save the file and duplicate it if needed. For example for pyrrolidine, you should have only *Ring1_31_1_New.xyz* for now. Copy it to *Ring1_31_1.xyz, Ring1_31_2.xyz* and *Ring1_31_3.xyz* and place them in a “*XYZ*” folder. They all look like this:

```
15
Pyrrolidine
C     0.000000   0.000000   0.000000   0
C     1.517922   -0.000000  0.000000   0
C     1.851211   1.489549   0.000000   0
C     0.862442   2.071334   -1.001061  0
N     -0.355817  1.165437   -0.900825  0
H     -0.398234  0.229373   0.984027   2
H     -0.469258  -0.898259  -0.385673  2
H     1.897765   -0.479622  -0.902057  3
H     1.909959   -0.526855  0.865163   3
H     2.874489   1.698905   -0.297989  3
H     1.684246   1.913760   0.989200   3
H     1.227323   1.997972   -2.020321  2
H     0.554058   3.090208   -0.797651  2
H     -0.618617  0.831532   -1.825583  1
H     -1.162413  1.668811   -0.538662  1
```

To say it again, you can directly open the optimized geometry and add manually the 5th column with “0”/“1”/”2”…, but we have preferred to rename and move the fragments. We let you check the files in the Fragments-GUI folder.

Once the SMARTS patterns and the .xyz files have been prepared, you can use the file *Run_02-PrepareFragments.sh* as explained at the beginning. You will need *ProcessFragments.exe* in the same folder. The script will check that the two patterns “with linked atom on the right” and “with linked atom on the left” return the same number of occurrence in the library. It also checks the same thing for the patterns “with linked atom on the left” and “with linked atom on the left with connection point”. If it is the case, a folder “*Fragments-GUI*” will be created with as many subfolders as fragments. In each subfolder the files needed by the GUI will be created.

11.5. Tweaking fragments

It is also possible to tweak the growth. For example, OpenGrowth doesn’t try to avoid building *cis* conformation for amides. If you want to do so, you can open the amide fragment and put a “0” in the last column of the corresponding hydrogen. Similarly, you can forbid the growth from axial hydrogens in cyclohexane.

11.6. List of prepared fragments

We present below the list of fragments that have already been prepared and that are available in the folder “*Fragments-GUI-413*” from the *Resources* file. There are 58 mono-rings and 41 2-fused rings fragments. Probability files created with all these fragments and with the database ChEMBL_Subset-8018.smi are available in the folder “*Fragments-OpenGrowth-413*”.

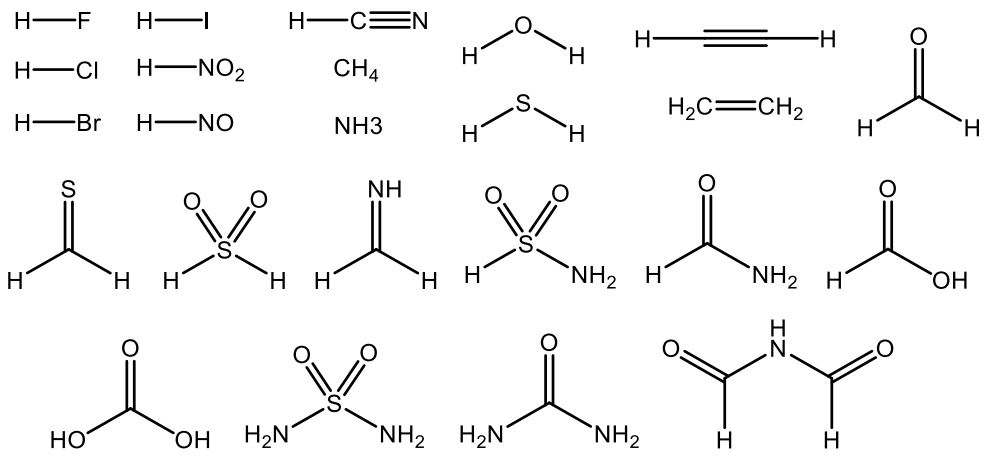


Figure 8. List of prepared non-ring fragments.

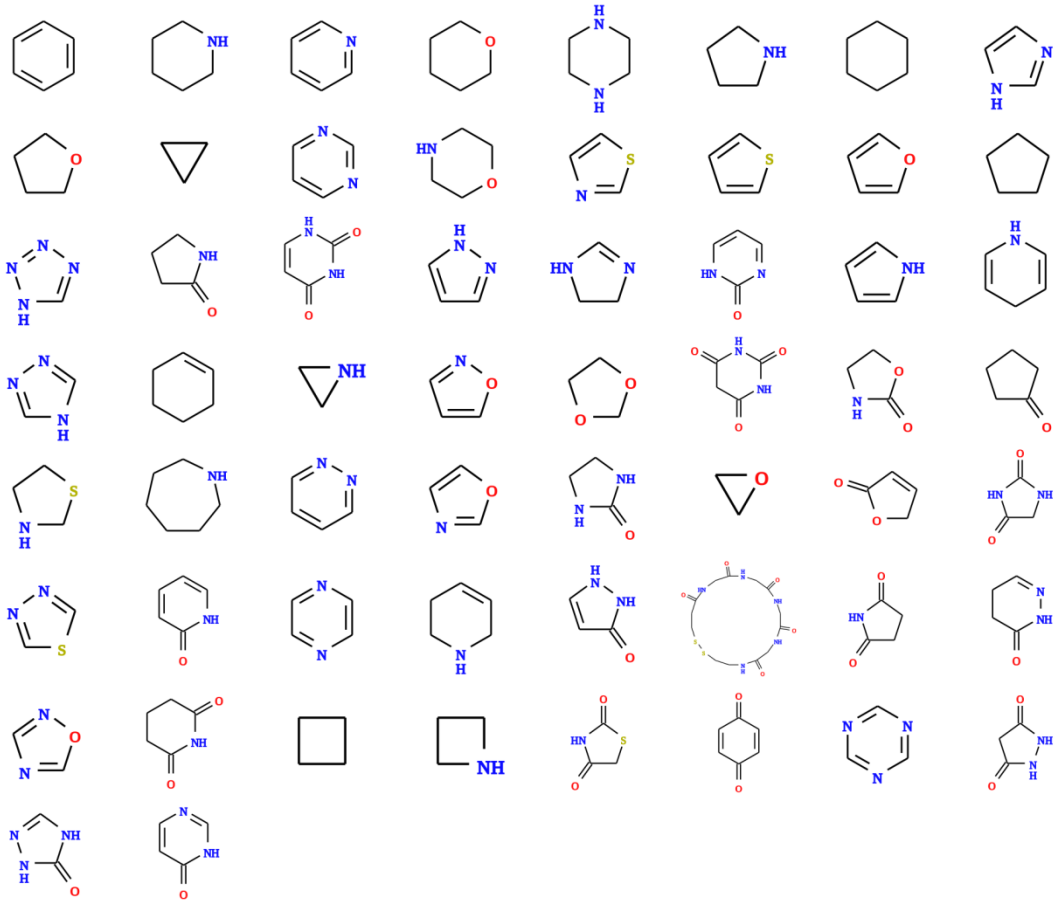


Figure 9. List of prepared mono-ring fragments.

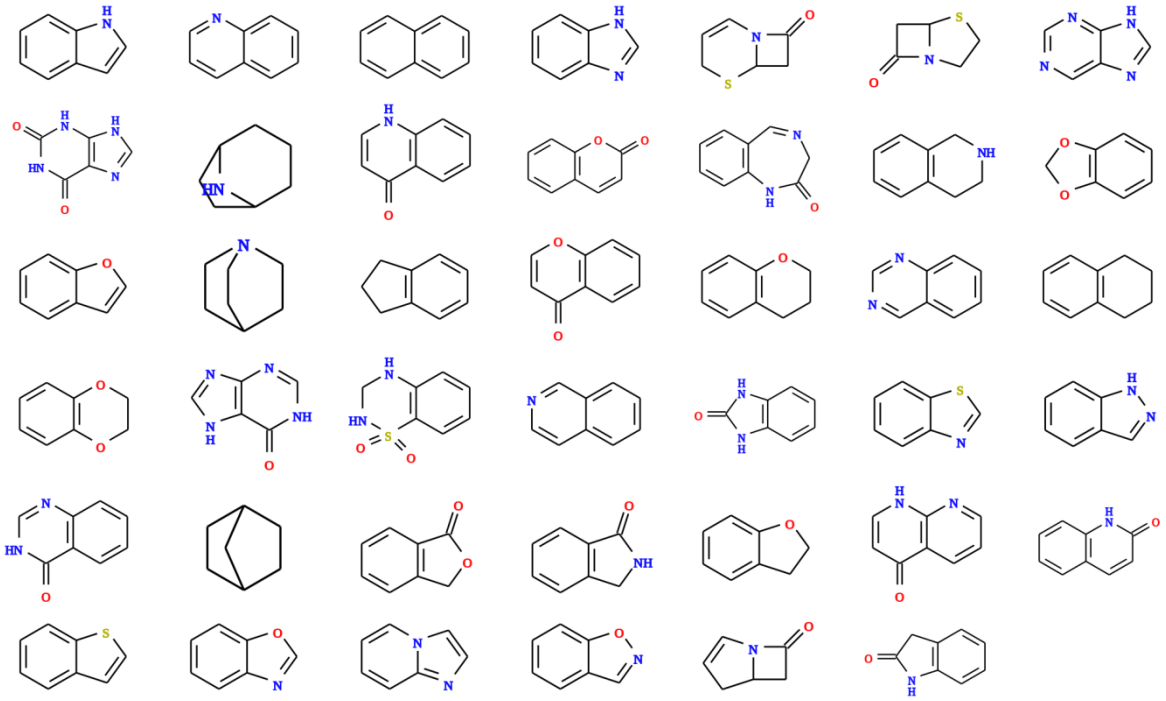


Figure 10. List of prepared 2-fused rings fragments.

12. Adding a new scoring function

If you want to add your own scoring function in the program, we explain below how to do it. If the function can be shared, please send it to us and we will add it to the next official release of the program.

12.1. Parse.cpp

If you want to add a new scoring function, you need first to change Parse.cpp to allow the use of your new scoring function. The easiest is to search for "SMOG2016" in the file and adapt it to your needs. The most important is the condition block around the line 328, with:

if (parameters.scoringFunction!="SMOG2001" && parameters.scoringFunction!="SMOG2016") {…}

You have to change it and add the name of your new function. Depending on the need of your function, you can also change the default value of some parameters in Parse.cpp.

12.2. PrepareProtein.cpp

When OpenGrowth starts, it "prepares" the protein files. It reads each protein structure provided as input, and creates an array of Atom (a structure defined in *OpenGrowth.h*). We store in this array only the protein atoms which are in the `PROTEIN_RANGE` to the center of the `BINDING_SITE`, and we assign to them the coordinates, the type of atom for the scoring function, the Lennard-Jones parameters, etc… If you have a function to assign the protein atom types needed by your scoring function, you need to add a line similar to the one around line 43:

else if (parameters.scoringFunction == "SMOG2016") {…}

12.3. Energy.cpp

The Energy function is very simple. If the scoring function is SMOG2016, we first assign the atom types for the ligand (we do that every time because they can change when a new fragment is added), and then call the function SMOG2016 which actually calculates the interaction energy.

12.4. Energy_SMOG2016.cpp

Here we are still using the example of the SMOG2016 scoring function. In this file, we have gathered everything that is needed. There are four functions: SMOG2016, KBP2016, LJP, LigandTypeSMOG2016 and ProteinTypeSMOG2016. The first one calculates the energy, using KBP2016 and LJP. The last two functions assign the atom types. So this is the kind of file that you must create with whatever you need.

12.5. OpenBabel objects

Proteins and ligands are stored as OpenBabel objects which are part of a Molecule or a Protein structure. For the protein, if you want to know how to access the coordinates, the residue or the atomID, you can have a look at what we did in PrepareProtein.cpp. For the ligand, you can check the function LigandTypeSMOG2016 in Energy_SMOG2016.cpp.

13. The Regrow option

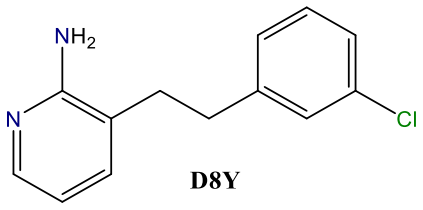
13.1. From grown ligands

After the growth of a good candidate, it can be interesting to regrow it with a directed mode in another receptor. Cross-reactivity can then be studied by using the `REGROW` option of `GROWTH_MODE`. The idea of this mode is to give to the program the numbers which would have been randomly picked otherwise. Only the choice of fragments and hydrogens are given. The position and orientation are still randomly chosen. With an infinite amount of time, every ligand which is grown with this mode would have been made otherwise.

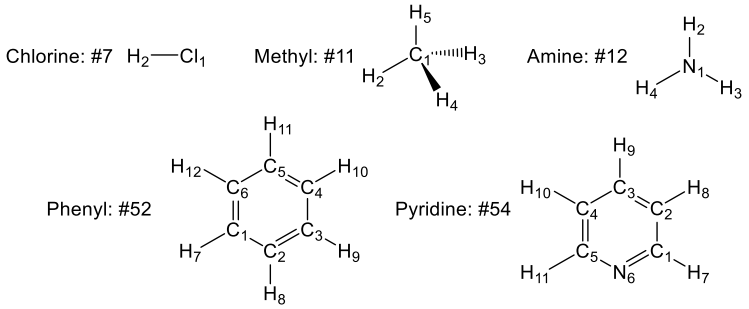
When the parameter `WRITEDESCRIPTION` is set to 1 during a growth, the list of randomly selected numbers is saved. They can then be copied in a text file which is given as an input of `REGROW_FILE`. The first number in this file is the index of the first fragment (from the list of fragments in `FRAGMENT_LIST`). Then, for every new fragment 3 numbers are given: the index of the hydrogen in the current ligand, the index of the new fragment from the list of fragments, the index of the hydrogen in the chosen fragment. If you want to regrow ligands grown by OpenGrowth, you only need to copy the list in a new file (made by a single line with all the numbers separated by a space).

13.2. From known ligands

It is also possible to manually create the list for known ligands. We will present how to build one ligand of the BACE-1 protein called D8Y (found in PDB ID 3KMY) whose structure is presented here. To build it, five fragments are used: pyridine, amine, methyl, phenyl, chlorine. For each one, we need to know where each fragment is in the `FRAGMENT_LIST` file (at which line) and the numbering of the atoms in the fragments. We display the structures of these fragments below. Note that for pyridine, three fragments are possible; which one you choose should not matter.

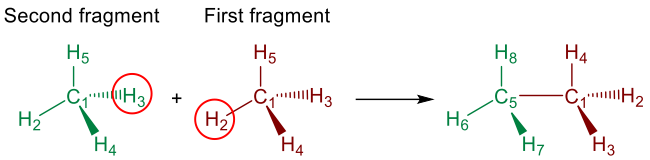


D8Y

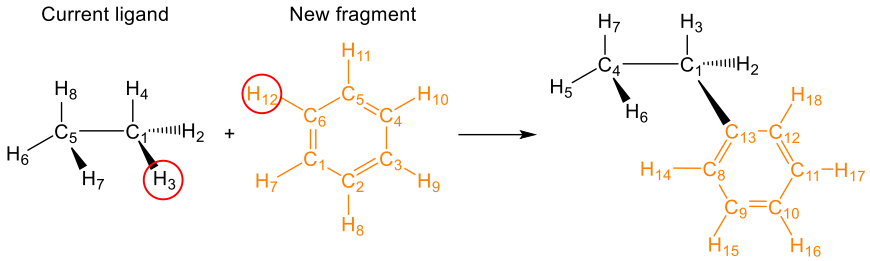


We will now describe how to generate the list of numbers needed for the `REGROW_FILE`. We first provide the list, and use colors to separate the fragments: 11 2 11 3 3 52 12 5 54 8 16 7 2 23 12 2. We will start by choosing a methyl, which is the 11th fragment in the `FRAGMENT_LIST` list. Hence, the first number is 11. Then, for every new fragment we provide: the index of the hydrogen in the current ligand, the index of the new fragment from the list of fragments, the index of the hydrogen in the chosen fragment.

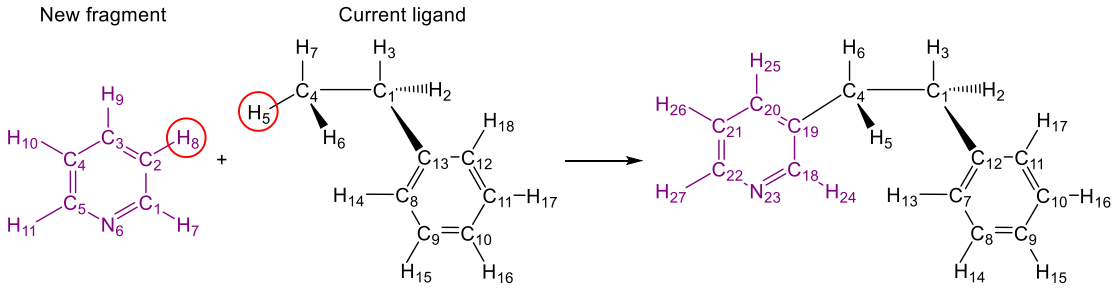
- In the current ligand (only methyl for now), we choose the hydrogen H_2 to connect a new fragment: we provide the number 2. We then choose methyl again (fragment #11) and we choose H_3 from the new methyl to connect to the current ligand. Thus, we provide "2 11 3". In the current ligand, $H_3/H_4/H_5$ become $H_2/H_3/H_4$ because the former H_2 have been removed (to create a bond to the new fragment) and thus the index of these three hydrogens decreases by one unit. The atoms of new fragments are stored after atoms of the current ligand; thus, their indices also change. The new ligand is displayed below with the ordering of atoms. We highly recommend to write all the structures for the preparation:



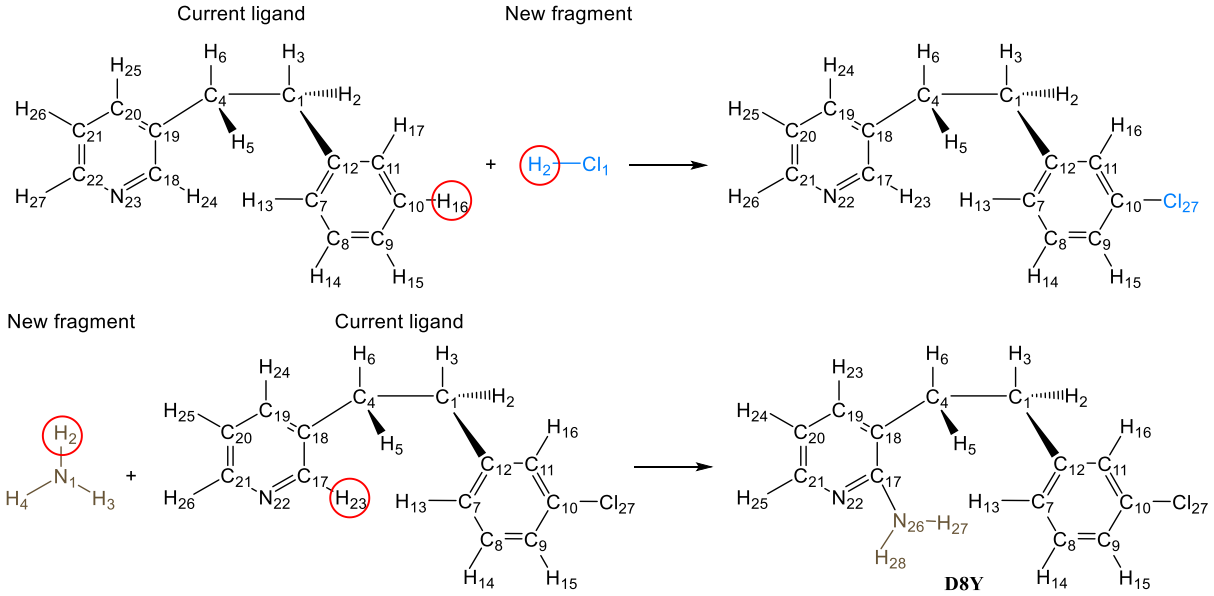
- We will now select the hydrogen H_3 from this ligand (displayed above), attach a phenyl to it (fragment #52), and select the 12th atom of phenyl. The numbers are "3 52 12", and the ligand is displayed below (note how the numbering of atoms change):



- We now select the 5th atom from this ligand, attach to it pyridine (fragment #54) from the 8th atom of pyridine. Hence, the sequence is "5 54 8". The ligand is displayed:



- Having detailed the previous steps, we assume that the process for choosing numbers is clear. The two following sequences are "16 7 2" and "23 12 2". See the numbering of atoms in the ligand below:



Finally, we have been able to create D8Y. To check that no mistakes were made, it is best to try to grow ligands with this sequence of numbers and analyze the output ligands. Of course, another sequence could have been used to grow the same ligand. We recommend starting from a fragment in the middle of the molecule and avoiding growing molecules linearly from one side to the other.

14. Molecular dynamics simulations

After molecules have been grown by OpenGrowth, it is important to compute more accurate binding free energies. We describe in our publication the workflow that we have used [5]; it is highly inspired by the one described by Wright *et al.* [9]. You can find in the file "*MD-Scripts_1.0.1.zip*" the scripts that we have actually used to perform these simulations. A Readme.txt file is available there. A more accurate solution would be to perform Free Energy Perturbations.

15. Roadmap

15.1. Energy

- Implementation of other scoring functions .
- One scoring function for growth, one for the final prediction (such as MM-PBSA).
- Coenzyme, inclusion of water (need to compute energy between ligand and small molecule).
- With protein flexibility, read conformers energies from a file and use them (ΔG = score + ligand constraints + protein constraints). The protein energy can come from:
  - Molecular mechanics: only compute the energy at a force field level.
  - MD simulations: after clustering, weight according to how many snapshots are in each clusters.
  - B factor from crystallography.

15.2. Growth

- Adapt the protonation states for aliphatic and aromatic amines.
- For ester to be different than carboxylic, once the fragment has been added we must force the addition of something to the H, or use other SMARTS patterns. The same applies for ether/hydroxyl and carbonyl/ketone.
- Covalent docking.
- Growing peptides (new GROWTH_MODE + prepare fragments).
- Use the Metropolis criterion every *n* fragments.

15.3. Miscellaneous

- Parallelize with openCL/CUDA.
- Post-process: synthetic rules, toxicity, ADME properties, stability, solubility.
- Fully integrate FOG2.0 in OpenGrowth.
- Identify conformational strain (gauche-gauche conformation, axial connection, cis-amide…).

16. References

[1] R. S. DeWitte and E. I. Shakhnovich, *J. Am. Chem. Soc.*, Vol. *118*, pp. 11733–11744, **1996**.

[2] R. S. DeWitte, A. V. Ishchenko, and E. I. Shakhnovich, *J. Am. Chem. Soc.*, Vol. *119*, pp. 4608–4617, **1997**.

[3] P. S. Kutchukian, D. Lou, and E. I. Shakhnovich, *J. Chem. Inf. Model.*, Vol. *49*, pp. 1630–1642, **2009**.

[4] N. O'Boyle, M. Banck, C. James, C. Morley, T. Vandermeersch, and G. Hutchison, *J. Cheminformatics*, Vol. *3*, p. 33, **2011**.

[5] N. Chéron, N. Jasty, and E. I. Shakhnovich, *J. Med. Chem.*, Vol. *59*, p. 4171−4188, **2016**.

[6] T. Debroise, E. I. Shakhnovich, and N. Chéron, *J. Chem. Inf. Model.*, Vol. *57*, pp. 584–593, **2017**.

[7] A. V. Ishchenko and E. I. Shakhnovich, *J. Med. Chem.*, Vol. *45*, pp. 2770–2780, **2002**.

[8] Y. Li, L. Han, Z. Liu, and R. Wang, *J. Chem. Inf. Model.*, Vol. *54*, pp. 1717–1736, **2014**.

[9] D. W. Wright, B. A. Hall, O. A. Kenway, S. Jha, and P. V. Coveney, *J. Chem. Theory Comput.*, Vol. *10*, pp. 1228–1241, **2014**.

17. Appendix

17.1. Compile OpenGrowthGUI

Linux

Start by installing qmake and the Qt libraries. For example, on a Debian-based distribution: "*sudo apt-get install qt4-qmake libqt4-dev*". Then go to the SourceGUI folder, and type "*qmake*", then "*make*". It should produce a OpenGrowthGUI.exe file. You can also use the Qt Creator: install "*qtcreator*" on OpenSuse/Debian/LinuxMint/Ubuntu and "*qt-creator.x86_64*" on Fedora. Then, start "*qtcreator*", open the project "*gui.pro*" from the SourceGUI folder and compile by clicking on the green arrow. More details are given below in the MacOS part.

MacOS

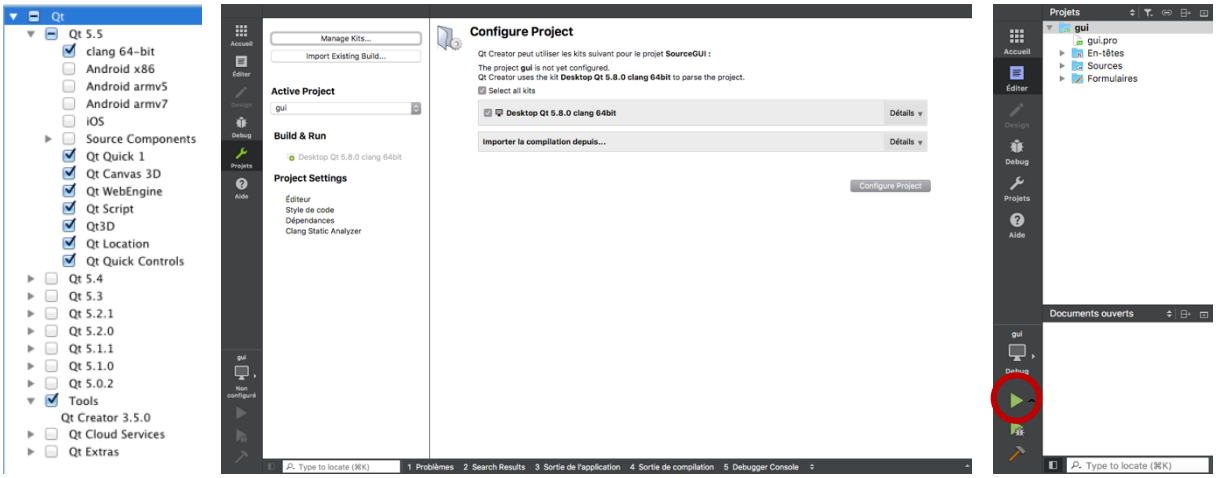
Here are the instructions if you need to compile OpenGrowthGUI on MacOS (either because you want to change it or because the provided files don't work):

- Download Qt Creator (http://www.qt.io/download-open-source/). During the installation, you will need to create an account for Qt. I have selected the following path: /Applications/Qt. You don't need to install all the components (for example you can remove those for iOS and Android, see Figure 11-a; the choice you have depends on the version you have downloaded, the screenshot was made for an old version of Qt Creator).
- Once the program is installed, open Qt Creator and select "Open Project" on the right. Open the *gui.pro* file from the SourceGUI folder.
- On the next screen, Qt will propose you to select a kit. Select the one by default ("Desktop QT 5.5.0 clang 64bit") and click on "Configure project" (see Figure 11-b).
- Then, click on the green arrow (see Figure 11-c) (not the one with the little bug on it, the plain one). After 10seconds, the GUI should open.

Another option, once you have installed Qt Creator, is to compile the GUI from command-line.

- Navigate to the SourceGUI folder in a terminal.

- Type "/Applications/Qt/5.8/clang_64/bin/qmake -config release" or something similar depending on where you have installed Qt.
- Type "make". In the same folder you should get a OpenGrowthGUI.app file that can be opened.
- To create a .dmg file, type "/Applications/Qt/5.5/clang_64/bin/macdeployqt OpenGrowthGUI.app/ -dmg -no-strip".



(a) Options. (b) Selection of a kit. (c) Compilation.

Figure 11. Installation of Qt Creator and compilation of OpenGrowthGUI.

Windows

If you want to change the GUI under Windows, it requires more steps than on Linux. We provide instructions below but we don't provide any support. First, start by compiling OpenBabel 2.3.2 with Visual C++ 2010 (and not Visual Studio 2013 or above). Note that we have only tested the 2.3.2 version and not yet the 2.4.1. The instructions are provided here: http://open-babel.readthedocs.org/en/latest/Installation/install.html#windows-msvc and are copied here:

- Install Visual C++ 2010 Express (free)
- Install CMake (http://www.cmake.org/download/, choose cmake-3.8.0-win32-x86.exe for example). During the installation, choose "Add CMake to the system PATH for all users".
- Dowload the OpenBabel sources and extract them. You need the v2.3.2 that you can get here: https://sourceforge.net/projects/openbabel/files/openbabel/2.3.2/.
- Go to the *windows-vc2008* folder (in the OpenBabel folder) and double-click on *default_build.bat*.
- Then go to the *build* folder, and double-click on openbabel.sln. This should open Visual C++ 2010. Choose *Release* in the menu bar. Then right-click on ALL_BUILD on the left panel and choose Build. When it is done, right-click on INSTALL (still on the left panel).
- In the "openbabel-2.3.2\windows-vc2008\install" directory, you will find three folders: bin, include, share.

Now that OpenBabel is installed, you can then go to the "*OpenGrowthGUI/Windows*" folder, and extract *SearchGUI_Project.zip*. Double-click on SearchUI.sln: this should open the project in Visual C++ 2010. To compile, click on the green arrow. If it complains that it can't find some files from OpenBabel, you will have to change some settings:

1. On the solution explorer (the left panel), right-click on the SearchGUI project and open "*Properties*".
2. Under "*Configuration properties*", find and click on "*VC++ Directories*".

3. Add the OpenBabel include folder to "*Include Directories*": it is the folder "*openbabel-2.3.2**windows-vc2008**install**include**openbabel-2.0*".
4. Add the source folder to "*Source Directories*". It is the folder "*openbabel-2.3.2**src*".
5. Under *Linker->General*, add to "*Additional Library Directories*" the folder containing "*openbabel-2.lib*". Very likely: "*openbabel-2.3.2**windows-vc2008**install**bin*".
6. Under *Linker->Input*, write "*openbabel-2.lib*" to "*Additional Dependencies*".

At this point, you should be able to compile SearchGUI. If you are trying to modify the GUI, the easiest way is to install Qt Creator (https://www.qt.io/download/). Then extract *SourceGUI_Project.zip* and load it in Qt Creator by opening the .gro file. It will then lead you through the configuration process. Once you have compiled the GUI, you can use the *windeployqt* utility from QT on the *OpenGrowthGUI.exe* file to gather all the needed libraries.

17.2. Operation details of OpenGrowthGUI

This section describes the overall working of the GUI in more details; this is useful especially if one wishes to modify the code in any way or if you want to create new fragments since it will be easier to understand what is needed. It is a technical section and is not needed for the simple use of the program. As explained before, OpenGrowthGUI is made of two parts: the GUI made with Qt and a standalone C++ executable (*SearchGUI.exe*) that needs to be compiled against the OpenBabel library. The reason for this design is to allow users to use the standalone executable on external CPU clusters, which permits parallelization of the tasks that would otherwise take a fairly long time.

Update

When the "*Update*" button is clicked, the GUI checks whether a Resource Directory and database are given. If not, an error message will be displayed in the message label in the bottom left corner. If they have been given, the GUI makes sure that the fragment button panels are empty, deleting any existing buttons. This allows for the clearing of the previous set of fragment buttons if a new database is selected within the same user session. Upon deletion, the GUI looks for *Fragments.dat* in the "*Fragments*" sub-folder of the Resource Directory, and begins to load the fragments listed into memory—categorizing them into non-ring fragments, one-ring fragments, and two-ring fragments as it goes. Global variables are set to store the number of each fragment type found. Next, the program starts the fragment searching in the database, calling the external executable *SearchGUI.exe* and passing the first fragment and database path as arguments.

The external executable is called as a QProcess, and signals and slots are used to pass the standard output and standard error of this executable to the GUI. The signal is the output of *SearchGUI.exe*, and the slot is a function in the GUI. In this case, the stdout of *SearchGUI.exe* is the number of times that the fragment is found in the database, giving us a way to store this value (along with the fragment name) in a map. Once the signal is received, the slot receiving the stdout of *SearchGUI.exe* also begins the search for the next fragment in the same manner. In this way, we are able to keep the load process sequential and non-overlapping. We continue in this way until all fragments are searched, and then create buttons for each fragment. Icons for these buttons are searched in the fragment folder for the given fragment; if no .svg icon is found, then the fragment name is painted on the button instead. The button label is set as the number of matches for that fragment found in the database, taken from the fragment map (which itself is set via *SearchGUI.exe*'s cout).

Fragment Files

When the "*Fragment Files*" button is clicked, the GUI makes sure that:

1. The Resource Directory is set.
2. The database is chosen.

3. The global variables numNoRings, numOneRings, and numTwoRings are not all set to zero (in other words, the fragments have been loaded).

It then opens the "*Fragments.dat*" file found in the Resource Directory. For each line in this file, it checks whether the corresponding button is clicked. If it is, the fragment line is copied to the file "*selected_fragments.dat*" located in the output directory. In addition, the xyz files for the selected fragments are copied to the output directory, and an index of these xyz files is written to the output folder, labeled "*Fragments.dat*".

Probabilities

When the "*Probabilities*" button is clicked, the GUI makes sure that the Resource Directory is set, the database has been chosen, and that fragments have been loaded (see *Fragment Files* section for details). Next, it calls *SearchGUI.exe* with the proper arguments—in this case, we pass the character '*p*' to signal the probability calculation is required, along with the database path, resource folder path, output folder path (an optional argument), and the final argument '*q*' to signal that we are calling the program from our GUI.

Adding the final argument '*q*' signals to *SearchGUI.exe* that it must maintain communication with the GUI. When *SearchGUI.exe* is calculating transition probabilities, it looks at one fragment and calculates all the transition probabilities from that fragment to all the others (by counting the number of times that the resulting dimer appears in the database, and normalizing the result). After one fragment's probabilities are computed, *SearchGUI.exe* contacts the GUI by sending the message "ping" to its stdout. The GUI normally replies '0' by writing data to the QProcess, signaling *SearchGUI.exe* to continue to the next fragment. If the user has chosen to "*Pause"* the program, the global integer "flag" is set to 1, signaling that the QProcess should be paused. The next time that *SearchGUI.exe* pings the GUI, the GUI will send back a '1' instead of '0', and the *SearchGUI.exe* QProcess will be killed. If the final argument is not set to '*q*', no pinging will occur—the program will continue until completion. Therefore, if using *SearchGUI.exe* outside the GUI, simply don't pass this parameter.

3-mer List

When the "*3-mer List*" button is clicked, the GUI makes sure that the Resource Directory is set, the database has been chosen, and that fragments have been loaded (see *Fragment Files* section for details). Next, it reads in the fragments data from the "*Fragments*" folder in the Resource Directory. The data for each fragment includes three distinct patterns (more details are provided in the section on how to add new fragments):

1. The SMARTS pattern for the given fragment, with connection point to the right.
2. The SMARTS pattern for the given fragment, with connection point to the left.
3. The same SMARTS pattern as 2, but with an extra placeholder "(y)" at every other connection point in the fragment.

For example, *Fragments/Ring1_8_1/ Ring1_8_1.smarts* (pyrazole) contains the following, formatted here on separate lines for clarity:

- Pattern1 (-->) = [c;X3;R1;!$(c=O)]1[c;X3;R1][c;X3;R1;!$(c=O)][n;X2;R1][n;X3;R1]1
- Pattern2 (<--) = [n;X3;R1]3[n;X2;R1][c;X3;R1;!$(c=O)][c;X3;R1][c;X3;R1;!$(c=O)]3
- Pattern3 (<--) = [n;X3;R1]5[n;X2;R1][c;X3;R1;!$(c=O)](y)[c;X3;R1](y)[c;X3;R1;!$(c=O)]5(y)

The first pattern has the connection atom (a nitrogen atom) on the right, the second pattern has it on the left (the 2nd pattern is the mirror of the 1st pattern), and the third has it on the left as well. This allows us to combine the three patterns by simply ordering the patterns and combining them. To make a two fragment molecules, we can simply add the 1st pattern of the desired first fragment to the 2nd pattern of the desired 2nd fragment. Pyrazole has four types of hydrogens and we have thus

four fragments (Ring1_8_1, Ring1_8_2, Ring1_8_3 and Ring1_8_4) with connection points for each one. Listing out each fragment orientation makes it faster and easier to form all possible connections between two fragments.

However, to make 3-mers we have to take more into account. For each first fragment (A), we can easily add a second fragment (B) in a manner similar to the one described above. However, the 3rd fragment can be added in a number of places. The easiest solution we found to solve this problem was to mark all these positions where a 3rd fragment could be added, and then replace one of these markers with the desired 3rd pattern. All the other markers must also be deleted. The character "y" was chosen as marker, as it wouldn't appear in any of our fragments. When combining SMARTS patterns in this manner, it is also important that ring connection indices are not confused. Since we are dealing with 2-fused ring fragments, it was easiest to use indices 1 and 2 for first patterns, indices 3 and 4 for second patterns, and indices 5 and 6 for third patterns (which will eventually become middle patterns). As an example, here is how we could join three of the pyrazole fragments to an A-B-C structure (using the patterns described above):

- First, we will create the A-B structure by merging Pattern1 and Pattern3: Pattern1 + "-" + Pattern3 = [c;X3;R1;!$(c=O)]1[c;X3;R1][c;X3;R1;!$(c=O)][n;X2;R1][n;X3;R1]1-[n;X3;R1]5[n;X2;R1][c;X3;R1;!$(c=O)](y)[c;X3;R1](y)[c;X3;R1;!$(c=O)]5(y).
- Next, we add structure C (Pattern2) in the intermediate pattern where the (y) are and delete the remaining (y). Three structures will be created since three (y) can be found. One of them is (by inserting the fragment C into the 1st insertion point of the middle fragment): Final 3-mer = [c;X3;R1;!$(c=O)]1[c;X3;R1][c;X3;R1;!$(c=O)][n;X2;R1][n;X3;R1]1-[n;X3;R1]5[n;X2;R1][c;X3;R1;!$(c=O)]([n;X3;R1]3[n;X2;R1][c;X3;R1;!$(c=O)][c;X3;R1][c;X3;R1;!$(c=O)]3)[c;X3;R1][c;X3;R1;!$(c=O)]5

However, we will not be testing all possible 3-mers. First, the GUI loads *Proba_Transition.dat* into memory. Then, as 3-mers are being made, the GUI checks whether there is a non-zero transition probability between the first fragment and the second fragment (A to B). If this value is zero, the 3-mer is not made (because the 3-mer will obviously not be present in the database). In this way, we can avoid searching in the database for many of the possible 3-mers. However, even if the B-C transition probability is 0 the 3-mer will be made: we have not yet found a way to avoid this. As each 3-mer is constructed, the GUI writes this 3-mer to the file "*3mers_to_search.dat*".

3-mers

When the "*3-mers*" button is clicked, the GUI makes sure that the Resource Directory is set, the database has been chosen, and that fragments have been loaded (see *Fragment Files* section for details). Then, it calls *SearchGUI.exe* with the proper arguments; in this case, we use the character '*t*' to signify that we are looking for 3mers. Again, we also need to include the database path, resource folder path, output folder path (an optional argument), and the final argument '*q*' to signal that we are calling the program from our GUI.

Upon starting, *SearchGUI.exe* will check the output folder for the file "*3mers_left.dat*". This file will be present if the user has paused the search process, and is attempting to restart it. If so, the program will search through the 3mers in "*3mers_left.dat*", outputting unfound ones to the file "*unfound_3mers.dat*". If this file doesn't exist, the program will check the output folder to determine whether a "*3mers_to_search.dat*" file is present in the output directory. If not, it will perform the same process as "*3-mer List*" above. Then, it will begin searching through the database for the 3mers found in "*3mers_to_search.dat*", outputting unfound ones to the file "*unfound 3mers.dat*".

After searching for each 3mer, *SearchGUI.exe* (with the '*q*' option enabled) will ping the GUI in a manner similar to the one described in the "*Probabilities*" section. If the program is paused, *SearchGUI.exe* will output the file "*3mers_left.dat*" before closing, which will contain all the 3mers

left to search. Upon restarting, these 3mers will be loaded into memory, and "*3mers_left.dat*" will be deleted.