



OpenEye
Scientific Software

GraphSim TK – C++
Release 2.0.1

OpenEye Scientific Software, Inc.

January 11, 2012

CONTENTS

1	Front Matter	1
2	Introduction	3
2.1	Getting Started	3
3	GraphSim Theory	5
3.1	Fingerprint Generation	5
3.2	Fingerprint Types	7
3.3	Storage and Retrieval	11
3.4	Similarity Measures	15
3.5	Fingerprint Database	19
3.6	User-defined Fingerprint	24
3.7	Fingerprint Coverage	31
3.8	Fingerprint Overlap	34
4	API	39
4.1	OESystem Classes	39
4.2	OEGraphSim Classes	44
4.3	OEGraphSim Constants	58
4.4	OEGraphSim Functions	67
5	Glossary and Bibliography	77
5.1	Glossary	77
5.2	Bibliography	78
6	Release Notes	79
6.1	GraphSimTK 2.0.1	79
6.2	GraphSimTK 2.0.0	79
6.3	GraphSimTK 1.0.1	80
6.4	GraphSimTK 1.0.0	80
7	Indices	83
	Bibliography	85
	Index	87

FRONT MATTER

Copyright 1997-2012 OpenEye Scientific Software, Santa Fe, New Mexico. All rights reserved.

All rights reserved. This material contains proprietary information of OpenEye Scientific Software. Use of copyright notice is precautionary only and does not imply publication or disclosure.

The information supplied in this document is believed to be true but no liability is assumed for its use or the infringement of the rights of others resulting from its use. Information in this document is subject to change without notice and does not represent a commitment on the part of OpenEye Scientific Software.

This package is sold/licensed/distributed subject to the condition that it shall not, by way of trade or otherwise, be lent, re-sold, hired out or otherwise circulated without OpenEye Scientific Software's prior consent, in any form of packaging or cover other than that in which it was produced. No part of this manual or accompanying documentation, may be reproduced, stored in a retrieval system on optical or magnetic disk, tape, CD, DVD or other medium, or transmitted in any form or by any means, electronic, mechanical, photocopying recording or otherwise for any purpose other than for the purchaser's personal use without a legal agreement or other written permission granted by OpenEye.

This product should not be used in the planning, construction, maintenance, operation or use of any nuclear facility nor the flight, navigation or communication of aircraft or ground support equipment. OpenEye Scientific Software, shall not be liable, in whole or in part, for any claims arising from such use, including death, bankruptcy or outbreak of war.

Windows is a registered trademark of Microsoft Corporation. Apple, OS X, and Macintosh are registered trademarks of Apple Computer, Inc. AIX and IBM are registered trademarks of International Business Machines Corporation. UNIX is a registered trademark of the Open Group. RedHat is a registered trademark of RedHat, Inc. Linux is a registered trademark of Linus Torvalds. SPARC is a registered trademark of SPARC International Inc.

SYBYL is a registered trademark of TRIPOS, Inc. MDL is a registered trademark and ISIS is a trademark of Accelrys, Inc. SMILES, SMARTS, and SMIRKS may be trademarks of Daylight Chemical Information Systems. Macromodel is a trademark of Schrodinger, Inc.

Python is a trademark of the Python Software Foundation. Java is a trademark or registered trademark of Sun Microsystems, Inc. in the U.S. and other countries.

Other products and software packages referenced in this document are trademarks and registered trademarks of their respective vendors or manufacturers.

INTRODUCTION

This manual is to familiarize the user with the *GraphSimTK* functionalities. It does not provide explanations of basic *OEChem* classes and functions. Therefore reading the *OEChem* manual beforehand is highly recommended.

The *OEGraphSim* library provides a facility for encoding 2D molecular graph information into fingerprints. The following basic fingerprint functionalities are available:

- Generation (see *Fingerprint Generation* chapter)
- Storage (see *Storage and Retrieval* chapter)
- Comparison (see *Similarity Measures* chapter)

Emphasis has been placed on speed, with attention to rapid in-memory fingerprint search (see *Fingerprint Database* chapter).

2.1 Getting Started

Please read the “OpenEye C++ Quick Start” manual

GRAPHSIM THEORY

3.1 Fingerprint Generation

The fingerprint types implemented in the *GraphSimTK* encode the 2D graph features of molecules. Fingerprints can be used in applications such as similarity searches, molecular characterization, molecular diversity and chemical database clustering.

The following five types of fingerprints are implemented:

1. *MACCS* (`OEFPType::MACCS166`)
2. *LINGO* (`OEFPType::Lingo`)
3. *Circular* (`OEFPType::Circular`)
4. *Path* (`OEFPType::Path`)
5. *Tree* (`OEFPType::Tree`)

3.1.1 MACCS

MACCS keys are 166 bit *structural key* descriptors in which each bit is associated with a *SMARTS* pattern.

The following code snippets demonstrate two **separate** ways to create a MACCS keys fingerprint:

```
OEFPType fp;  
OEMakeMACCS166FP(fp, mol);  
  
OEMakeFP(fp, mol, OEFPType::MACCS166);
```

3.1.2 LINGO

The *GraphSimTK* provides fingerprint API for the *LINGO* similarity search method implemented in *OEChem*.

The following code snippets demonstrate two **separate** ways to create a *LINGO* fingerprint:

```
OEFPType fp;  
OEMakeLingoFP(fp, mol);  
  
OEMakeFP(fp, mol, OEFPType::Lingo);
```

3.1.3 Circular

A circular fingerprint is generated by **exhaustively** enumerating **all** circular fragments grown radially from each heavy atom of the molecule up to the given radius and then hashing these fragments into a fixed-length bitvector. See *Figure: Example of enumerating circular fragments*.

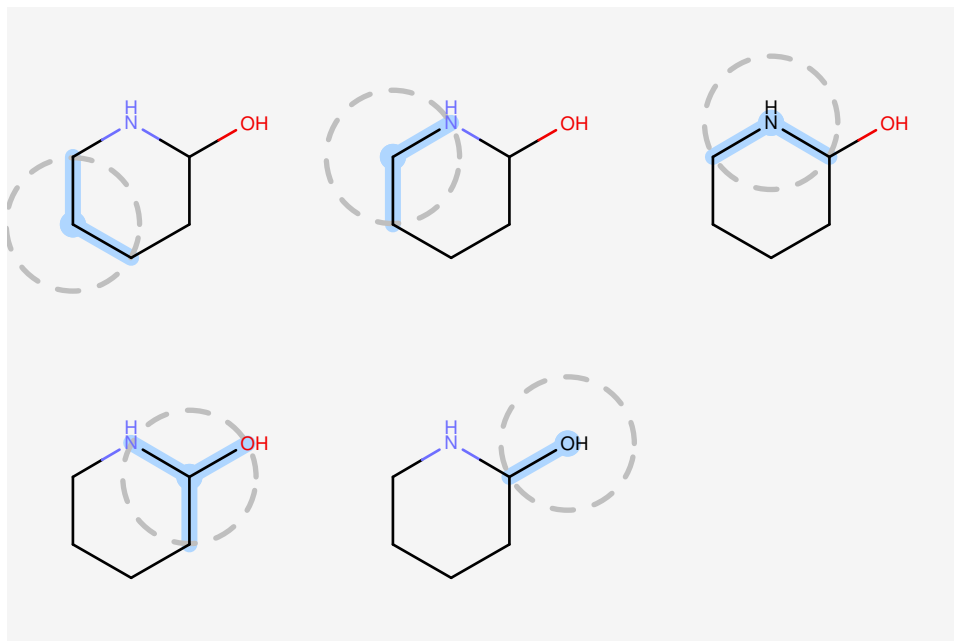


Figure 3.1: **Example of enumerating all unique circular fragments with one radius**

The following code snippets demonstrate two **separate** ways to create a circular fingerprint with default parameters:

```

OEFingerPrint fp;
OEMakeCircularFP(fp, mol);

OEMakeFP(fp, mol, OEFPType::Circular);

```

3.1.4 Path

A path fingerprint is generated by **exhaustively** enumerating **all** linear fragments of a molecular graph up to a given size and then hashing these fragments into a fixed-length bitvector. See *Figure: Example of enumerating path fragments*.

The following code snippets demonstrate two **separate** ways to create a path fingerprint with default parameters:

```

OEFingerPrint fp;
OEMakePathFP(fp, mol);

OEMakeFP(fp, mol, OEFPType::Path);

```

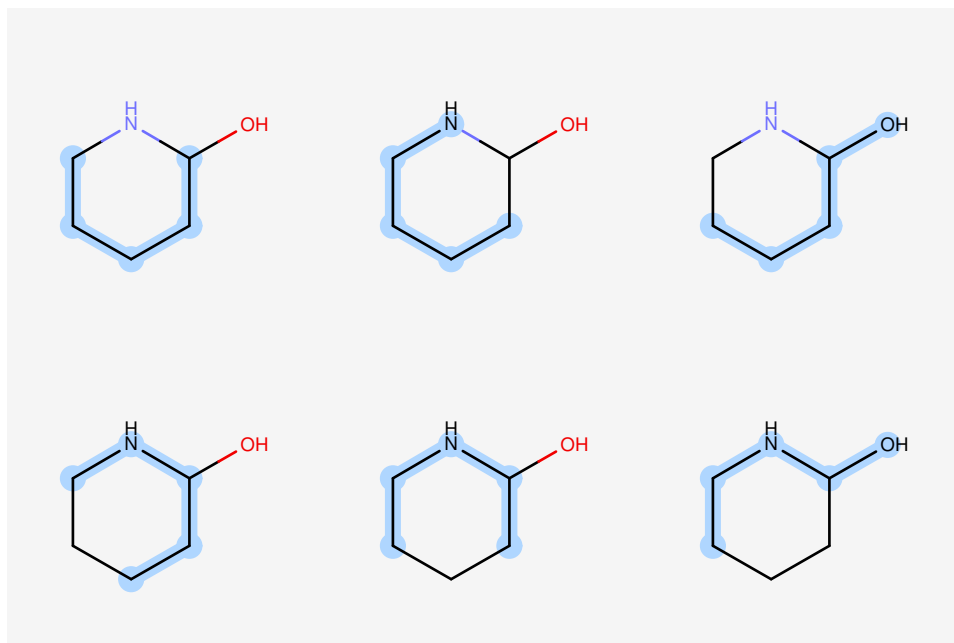


Figure 3.2: Example of enumerating all unique tree fragments with four bonds

3.1.5 Tree

A tree fingerprint is generated by **exhaustively** enumerating **all** tree fragments of a molecular graph up to a given size and then hashing these fragments into a fixed-length bitvector. See *Figure: Example of enumerating tree fragments*.

The following code snippets demonstrate two **separate** ways to create a tree fingerprint with default parameters:

```

OEFingerprint fp;
OEMakeTreeFP(fp, mol);

OEMakeFP(fp, mol, OEFPType::Tree);

```

See Also:

GraphSimTK also provides the ability to parameterize the circular, path and tree fingerprint generation with arbitrary sets of properties. For more details see the *User-defined Fingerprint* chapter.

3.2 Fingerprint Types

A fingerprint is a bitvector. To reflect this the `OEFingerprint` class derives from the `OEBitVector` class. The difference is that `OEFingerprint` has a type that represents how the fingerprint is generated. Fingerprints may only be compared if they are generated in the same way. Therefore, the following restriction is introduced:

Warning: When two fingerprints are subjected to similarity calculation their type has to be identical.

Listing 1 shows how to create different fingerprint objects (`OEFingerprint`) and identify or compare their types.

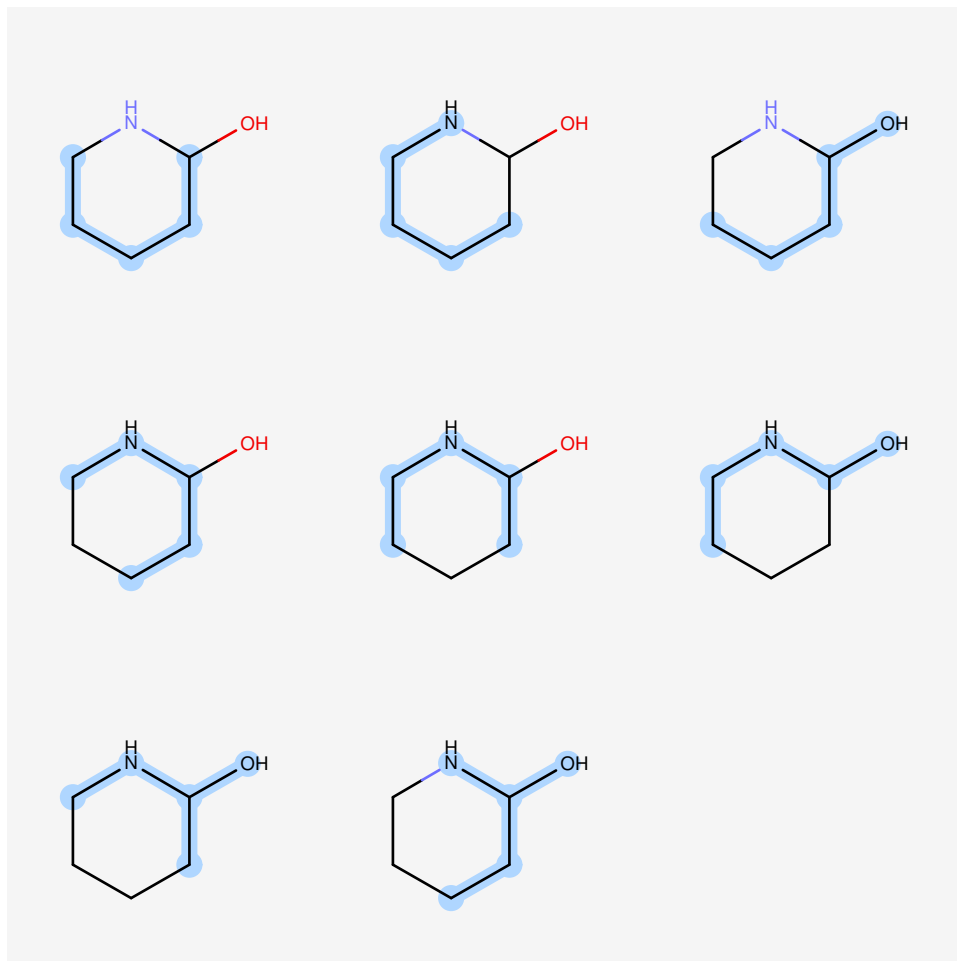


Figure 3.3: Example of enumerating all unique linear fragments with four bonds

Listing 1: Fingerprint type

```

#include <openeye.h>
#include <oechem.h>
#include <oegraphs.h>

using namespace OEChem;
using namespace OEGraphSim;

int main(int, char* [])
{
    OEFingerPrint fpA;
    OEFingerPrint fpB;
    if ( !fpA )
        std::cout << "uninitialized fingerprint" << std::endl;

    OEGraphMol mol;
    OEParseSmiles(mol, "ClCCCCCl");

    OEMakeFP(fpA, mol, OEFPType::Path);
    OEMakeFP(fpB, mol, OEFPType::Lingo);

    if (OEIsFPType(fpA, OEFPType::Lingo))
        std::cout << "Lingo" << std::endl;
    if (OEIsFPType(fpA, OEFPType::Path))
        std::cout << "Path" << std::endl;

    if (OEIsSameFPType(fpA, fpB))
        std::cout << "same fingerprint types" << std::endl;
    else
        std::cout << "different fingerprint types" << std::endl;

    return 0;
}

```

The output of Listing 1 is the following:

```

uninitialized fingerprint
Path
different fingerprint types

```

Two fingerprints are considered to be equivalent only if they have the same fingerprint type (`OEFPTypeBase`) and have identical bit-vectors (`OEBitVector`). The following code snippet shows how to compare two `OEFingerPrint` objects.

```

if (fpA == fpB)
    std::cout << "same fingerprints" << std::endl;
else
    std::cout << "different fingerprints" << std::endl;

```

The following code snippet shows how to initialize a `OEFingerPrint` object by using the type of another fingerprint. The type of a fingerprint is accessed by the `OEFingerPrint::GetFPTypeBase` method.

```

OEFingerPrint fpA;
OEMakePathFP(fpA, mol);

```

```
OEFingerPrint fpB;  
OEMakeFP(fpB, mol, fpA.GetFPTypeBase());
```

3.2.1 Fingerprint parameters

The *User-defined Fingerprint* chapter gives examples of how user defined fingerprints can be generated by defining, for example, the atom and bond properties that will be encoded into the fingerprints.

In order to ensure that only equivalent fingerprints can be compared, the fingerprint type stores the parameters being used in the generation process. The `OEFPTypeBase::GetFPTypeString` method returns the string representation of the fingerprint type that includes information about the parameters being used.

```
OEFingerPrint fpA;  
OEMakeFP(fpA, mol, OEFPType::Path);  
std::cout << fpA.GetFPTypeBase()->GetFPTypeString() << std::endl;
```

The output of the preceding snippet is the following:

```
Path,ver=2.0.0,size=4096,bonds=0-5,atype=AtmNum|Arom|Chiral|FCharge|HvyDeg|Hyb|EqHalo,  
btype=Order|Chiral
```

Note: The returned string does not include newline characters, the string was broken into two separate lines here only for better readability.

The following [Listing 2](#) shows how to extract the parameters of a fingerprint from a string representation by using the `OEFPTypeParams` class.

Listing 2: Fingerprint parameters

```
#include <openeye.h>  
#include <oechem.h>  
#include <oegraphsim.h>  
  
using namespace std;  
using namespace OEChem;  
using namespace OEGraphSim;  
  
int main()  
{  
    const OEFPTypeBase* fptype = OEGetFPType(OEFPType::Path);  
    OEFPTypeParams prms(fptype->GetFPTypeString());  
    cout << "version = " << OEGetFingerprintVersionString(prms.GetVersion()) << endl;  
    cout << "number of bits = " << prms.GetNumBits() << endl;  
    cout << "min bonds = " << prms.GetMinDistance() << endl;  
    cout << "max bonds = " << prms.GetMaxDistance() << endl;  
    cout << "atom types = " << OEGetFPAtomType(prms.GetAtomTypes()) << endl;  
    cout << "bond types = " << OEGetFPBondType(prms.GetBondTypes()) << endl;  
    return 0;  
}
```

The output of [Listing 2](#) is the following:

```

version = 2.0.0
number of bits = 4096
min bonds = 0
max bonds = 5
atom types = AtmNum|Arom|Chiral|FCharge|HvyDeg|Hyb|EqHalo
bond types = Order|Chiral

```

See Also:

- *User-defined Fingerprint* chapter
- `OEIsValidFPTypeString` function
- `OEGetFPType` function
- `OEFPAtomType` namespace
- `OEGetFPAtomType` function
- `OEFPBondType` namespace
- `OEGetFPBondType` function

3.2.2 Fingerprint version number

Each fingerprint type additionally has a version number. Version numbers are introduced in order to keep track of changes in the fingerprint generation algorithm itself. The `OEFPTypeBase::GetFPVersionString` method returns the string representation of the fingerprint version.

```

OEFingerprint fpB;
OEMakeFP(fpB, mol, OEFPType::Circular);
std::cout << fpB.GetFPTypeBase()->GetFPVersionString() << std::endl;

```

The output of the preceding snippet is the following:

```
2.0.0
```

The version number of the fingerprints will not be changed with each release. It will be incremented only if modifications or bug fixes to the corresponding algorithm would result in generating a different bit-vector for the same molecules.

Fingerprints with an old version number will be still readable and comparable with each other but not with fingerprints which have different version number.

3.3 Storage and Retrieval

The `OEFingerprint` does not store any reference to the molecule from which it was generated. The user has to keep track of which fingerprint corresponds to which molecule. One way to do this is to attach the fingerprint, as generic data, to the molecule. [Listing 3](#) shows how store and retrieve fingerprints as generic data.

Listing 3: Storing and retrieving fingerprint as generic data

```
#include <openeye.h>
#include <oechem.h>
#include <oegraphsimsim.h>

using namespace OEChem;
using namespace OEGraphSim;

int main(int, char* [])
{
    const char* tag = "FP_DATA";
    OEGraphMol mol;
    OEParseSmiles(mol, "ClCCCCCl");

    OEFingerPrint fp;
    OEMakeLingoFP(fp, mol);
    mol.SetData<OEFingerPrint>(tag, fp);

    if (mol.HasData(tag))
    {
        OEFingerPrint f = mol.GetData<OEFingerPrint>(tag);
        if ( f )
        {
            std::string fptype = f.GetFPTypeBase()->GetFPTypeString();
            std::cout << fptype << " fingerprint with `" << tag
                << "` identifier" << std::endl;
        }
    }

    return 0;
}
```

It is good practice to check the validity of the fingerprint after retrieving it. The `OEFingerPrint::operator bool` method returns true if the fingerprint was successfully initialized.

Listing 4 demonstrates how to create an OEB binary file that stores molecules along with their fingerprints. When reading the OEB file that was generated by this program, the pre-calculated fingerprints can be accessed rapidly with the `PATH_FP` tag. This eliminates the *on-the-fly* generation of the fingerprints.

See Also:

Additional examples in Listing 10 and Listing 12 of the *Fingerprint Database* chapter.

Listing 4: Fingerprint generation and storage in OEB

```
#include <openeye.h>
#include <oechem.h>
#include <oegraphsimsim.h>

using namespace OESystem;
using namespace OEChem;
using namespace OEGraphSim;

int main(int argc, char* argv[])
{
    if (argc != 3)
        OEThrow.Usage("%s <infile> <outfile>", argv[0]);
```

```

oemolistream ifs;
if (!ifs.open(argv[1]))
    OThrow.Fatal("Unable to open %s for reading", argv[1]);

oemolostream ofs;
if (!ofs.open(argv[2]))
    OThrow.Fatal("Unable to open %s for writing", argv[2]);
if (ofs.GetFormat() != OEFormat::OEB)
    OThrow.Fatal("%s output file has to be an OEBinary file", argv[2]);

OEFingerprint fp;
OEGraphMol mol;
while (OEReadMolecule(ifs, mol))
{
    OEMakeFP(fp, mol, OEFPTType::Path);
    mol.SetData<OEFingerprint>("PATH_FP", fp);
    OEWwriteMolecule(ofs, mol);
}
return 0;
}

```

The following code snippets shows how to generate a bitstring from an `OEFingerprint` object.

Listing 5: Accessing a fingerprint as a bitstring

```

string GetBitString(const OEFingerprint& fp)
{
    string bitstring;
    bitstring.resize(fp.GetSize(), '0');
    for(unsigned int b = 0; b < fp.GetSize(); ++b)
    {
        if (fp.IsBitOn(b))
            bitstring[b] = '1';
    }
    return bitstring;
}

```

See Also:

- `OEBitVector` class

Warning: Even though the *OEGraphSim* library provides a fingerprint API for the *LINGO* similarity search method, it is not implemented as a *real fingerprint*, so bitstrings that are generated from LINGO fingerprints are meaningless.

Fingerprints can also be stored in SDF files. The Listing 6 demonstrates how to create an SDF and store fingerprints as hexadecimal strings.

After the fingerprint is generated, it is attached to the molecule as an SD data tag. The identifier of the fingerprint in the SDF file will be the string representation of the fingerprint type. The bitvector of the fingerprint is converted to a hexadecimal string with the `OEBitVector::ToHexString` method.

Listing 6: Storing fingerprint in SDF

```
#include <openeye.h>
#include <oechem.h>
#include <oegraphsimsim.h>

using namespace std;
using namespace OESystem;
using namespace OEChem;
using namespace OEGraphSim;

int main(int argc, char* argv[])
{
    if (argc != 3)
        OEThrow.Usage("%s <infile> <outfile>", argv[0]);

    oemolistream ifs;
    if (!ifs.open(argv[1]))
        OEThrow.Fatal("Unable to open %s for reading", argv[1]);

    oemolostream ofs;
    if (!ofs.open(argv[2]))
        OEThrow.Fatal("Unable to open %s for writing", argv[2]);
    if (ofs.GetFormat() != OEFormat::SDF)
        OEThrow.Fatal("%s output file has to be an SDF file", argv[2]);

    OEFingerprint fp;
    OEGraphMol mol;
    while (OEReadMolecule(ifs, mol))
    {
        OEMakeFP(fp, mol, OEFPTType::Circular);
        string fptypestr = fp.GetFPTTypeBase()->GetFPTTypeString();
        string fphexdata;
        fp.ToHexString(fphexdata);
        OESetSDData(mol, fptypestr, fphexdata);
        OEWriteMolecule(ofs, mol);
    }
    return 0;
}
```

The following example (Listing 7) shows how to retrieve fingerprints from an SDF file. When looping over the SD data the `OEIsValidFPTTypeString` functions can be used to identify SD data that stores a fingerprint. The tag of the data is the string representation of the fingerprint type. This string representation can be used to generate the corresponding `OEFPTTypeBase` object by using the `OEGetFPTType` function. The type of the `OEFingerprint` object then has to be set by the `OEFingerprint::SetFPTTypeBase` method. Finally, the bitvector of the fingerprint then can be initialized from the hexadecimal string by using the `OEBitVector::FromHexString` method.

Listing 7: Retrieving fingerprint from SDF

```
#include <openeye.h>
#include <oesystem.h>
#include <oechem.h>
#include <oeographsimsim.h>
```

```

using namespace std;
using namespace OESystem;
using namespace OEChem;
using namespace OEGraphSim;

int main(int argc, char* argv[])
{
    if (argc != 2)
        OEThrow.Usage("%s <infile>", argv[0]);

    oemolistream ifs;
    if (!ifs.open(argv[1]))
        OEThrow.Fatal("Unable to open %s for reading", argv[1]);
    if (ifs.GetFormat() != OEFormat::SDF)
        OEThrow.Fatal("%s input file has to be an SDF file", argv[1]);

    unsigned int molcounter = 0;
    unsigned int fpcounter = 0;
    OEGraphMol mol;
    while (OEReadMolecule(ifs, mol))
    {
        molcounter += 1;
        for (OEIter<OESDDDataPair> dp = OEGetSDDDataPairs(mol); dp; ++dp)
        {
            if (OEIsValidFPTypeString(dp->GetTag()))
            {
                fpcounter += 1;
                string fptypestr = dp->GetTag();
                string fphexdata = dp->GetValue();
                OEFingerprint fp;
                const OEFPTypeBase* fptype = OEGetFPType(fptypestr);
                fp.SetFPTypeBase(fptype);
                fp.FromHexString(fphexdata);
            }
        }
    }

    cout << "Number of molecules = " << molcounter << endl;
    cout << "Number of fingerprints = " << fpcounter << endl;

    return 0;
}

```

See Also:

- `OEFPTypeBase::GetFPTypeString` method
- `OEBitVector` class
- **SD Tagged Data Manipulation** chapter in the *OEChemTk* manual

3.4 Similarity Measures

The basic idea underlying similarity-based measures is that molecules that are structurally similar are likely to have similar properties. In a fingerprint the presence or absence of a structural fragment is represented by the presence or absence of a set bit. This means that two molecules are judged as being similar if they have a large number of bits in common.

Measuring molecular similarity or dissimilarity has two basic components: the representation of molecular characteristics (such as fingerprints) and the similarity coefficient that is used to quantify the degree of resemblance between two such representations.

3.4.1 Built-in Similarity Measures

Since different similarity coefficients quantify different types of structural resemblance, several built-in similarity measures are available in the *GraphSimTK* (see [Table: Built-in similarity indices](#)) The table below defines the four basic bit count terms that are used in fingerprint-based similarity calculations:

Table 3.1: Basic bit count terms of similarity calculation

Symbol	Description
<i>onlyA</i>	number of bits set “on” in fingerprint A but not in B
<i>onlyB</i>	number of bits set “on” in fingerprint B but not in A
<i>bothAB</i>	number of bits set “on” in both fingerprints
<i>neitherAB</i>	number of bits set “off” in both fingerprints

Table 3.2: Built-in similarity indices

Similarity measure	Range	OEGraphSim Function
Cosine	[0.0 – 1.0]	OECosine
Dice	[0.0 – 1.0]	OEDice
Euclidean	[0.0 – 1.0]	OEEuclid
Manhattan	[1.0 – 0.0]	OEManhattan
Tanimoto	[0.0 – 1.0]	OETanimoto
Tversky	variable	OETversky

Cosine

$$\text{Formula: } \text{Sim}_{\text{Cosine}}(A, B) = \frac{\text{bothAB}}{\sqrt{(\text{onlyA} + \text{bothAB}) * (\text{onlyB} + \text{bothAB})}}$$

Calculates the ratio of the bits in common to the geometric mean of the number of “on” bits in the two fingerprints.

Dice

$$\text{Formula: } \text{Sim}_{\text{Dice}}(A, B) = \frac{2 * \text{bothAB}}{\text{onlyA} + \text{onlyB} + 2 * \text{bothAB}}$$

Calculates the ratio of the bits in common to the arithmetic mean of the number of “on” bits in the two fingerprints.

Euclidean

$$\text{Formula: } \text{Sim}_{\text{Euclid}}(A, B) = \sqrt{\frac{\text{bothAB} + \text{neitherAB}}{\text{onlyA} + \text{onlyB} + \text{bothAB} + \text{neitherAB}}}$$

Manhattan

$$\text{Formula: } \text{Sim}_{\text{Manhattan}}(A, B) = \frac{\text{onlyA} + \text{onlyB}}{\text{onlyA} + \text{onlyB} + \text{bothAB} + \text{neitherAB}}$$

Tanimoto

$$\text{Formula: } \text{Sim}_{\text{Tanimoto}}(A, B) = \frac{\text{both}AB}{\text{only}A + \text{only}B + \text{both}AB}$$

The number of bits set in both molecules divided by the number of bits set in either molecule. The more sparsely bits are set “on”, the smaller $\text{Sim}_{\text{Tanimoto}}$ values generally become.

Tversky

$$\text{Formula: } \text{Sim}_{\text{Tversky}}(A, B) = \frac{\text{both}AB}{\alpha * \text{only}A + \beta * \text{only}B + \text{both}AB}$$

The Tversky similarity measure is asymmetric. Setting the parameters $\alpha = \beta = 1.0$ is identical to using the *Tanimoto* measure.

The factor α weights the contribution of the first ‘reference’ molecule. The larger α becomes, the more weight is put on the bit setting of the reference molecule.

Note: The calculation of the `OEFPType::Lingo` fingerprint is based on fragmenting canonical isomeric *SMILES* into overlapping four character long substrings. If any of the two *SMILES* being compared is shorter than four characters, then their Tanimoto score will be:

- 1.0, if the two *SMILES* are identical
- 0.0, otherwise.

3.4.2 Similarity Calculation

The following example demonstrates how to calculate `Tanimoto` similarity scores for the molecules depicted in *Figure: Example molecules*.

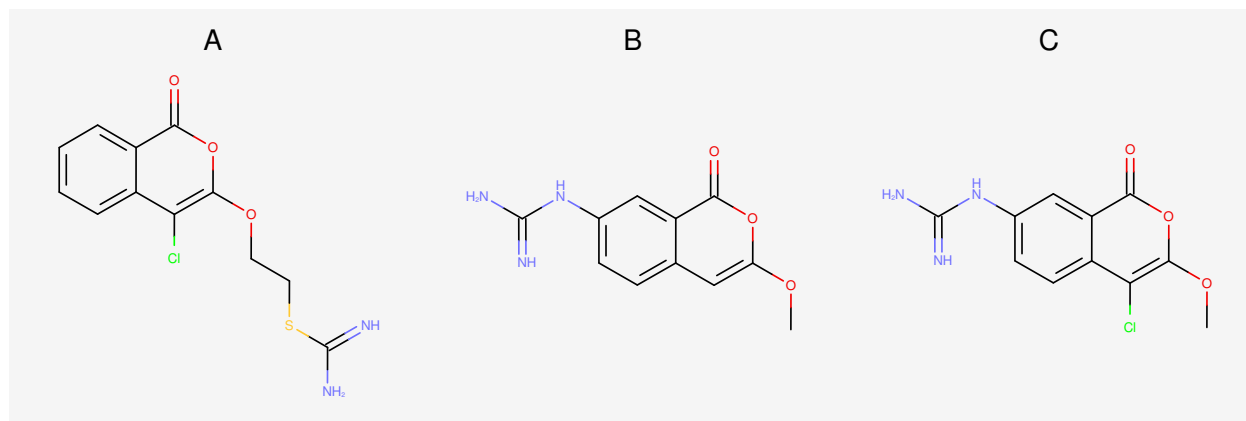


Figure 3.4: Example molecules

Listing 8: Calculating Tanimoto index

```
#include <openeye.h>
#include <oechem.h>
#include <oegraphs.h>
```

```

using namespace OEChem;
using namespace OEGraphSim;

int main(int, char* [])
{
    OEGraphMol molA;
    OEParseSmiles(molA, "c1ccc2c(c1)c(c(oc2=O)OCCSC(=N)N)Cl");
    OEFingerPrint fpA;
    OEMakeFP(fpA, molA, OEFPTType::MACCS166);

    OEGraphMol molB;
    OEParseSmiles(molB, "COc1cc2ccc(cc2c(=O)o1)NC(=N)N");
    OEFingerPrint fpB;
    OEMakeFP(fpB, molB, OEFPTType::MACCS166);

    OEGraphMol molC;
    OEParseSmiles(molC, "COc1c(c2ccc(cc2c(=O)o1)NC(=N)N)Cl");
    OEFingerPrint fpC;
    OEMakeFP(fpC, molC, OEFPTType::MACCS166);

    std::cout.precision(3);
    std::cout << "Tanimoto(A,B) = " << OETanimoto(fpA, fpB) << std::endl;
    std::cout << "Tanimoto(A,C) = " << OETanimoto(fpA, fpC) << std::endl;
    std::cout << "Tanimoto(B,C) = " << OETanimoto(fpB, fpC) << std::endl;
    return 0;
}

```

Molecules B and C (shown in *Figure: Example Molecules*) have the largest Tanimoto value since they share the largest number of common structural features.

For these example molecule the output of [Listing 8](#) is the following:

```

Tanimoto(A,B) = 0.618
Tanimoto(A,C) = 0.709
Tanimoto(B,C) = 0.889

```

3.4.3 User-defined Similarity Measures

The following code snippet demonstrates how implement the Yule similarity measure with the following formula:

$$Sim_{Yule}(A, B) = \sqrt{\frac{(bothAB * neitherAB) - (onlyA * onlyB)}{(bothAB * neitherAB) + (onlyA * onlyB)}}$$

```

float CalculateYule(const OEFingerPrint& fpA, const OEFingerPrint& fpB)
{
    unsigned int onlyA, onlyB, bothAB, neitherAB;
    OEGetBitCounts(fpA, fpB, &onlyA, &onlyB, &bothAB, &neitherAB);
    float yule = (float)(bothAB * neitherAB - onlyA * onlyB);
    yule /= (float)(bothAB * neitherAB + onlyA * onlyB);
    return yule;
}

```

The `OEGetBitCounts` function returns the four basic values (namely *onlyA*, *onlyB*, *bothAB* and *neitherAB*) from which any similarity measures can be calculated. For the definition of these values see [Table: Basic bit count terms](#)

```
OEMakeFP(fpA, molA, OEFPTType::Path);
OEMakeFP(fpB, molB, OEFPTType::Path);
OEMakeFP(fpC, molC, OEFPTType::Path);

std::cout << "Yule(A,B) = " << CalculateYule(fpA, fpB) << std::endl;
std::cout << "Yule(A,C) = " << CalculateYule(fpA, fpC) << std::endl;
std::cout << "Yule(B,C) = " << CalculateYule(fpB, fpC) << std::endl;
```

Warning: User-defined similarity measures can only be used with circular, path, tree, and MACCS key fingerprints but **not** with LINGO (`OEFPTType::Lingo`).

3.5 Fingerprint Database

The following four examples perform the same task, detailed below:

1. reading a query structure
2. printing out the similarity score between the fingerprint of this query and the fingerprint generated for each molecule read from a database file.

In [Listing 9](#), after importing the query structure and generating its path fingerprint, the program loops over the database file creating a path fingerprint for each structure. Then the program calculates the Tanimoto similarity between the fingerprint of the query and the database entry by calling the `OETanimoto` function.

Listing 9: Similarity calculation from file

```
#include <iostream>

#include <openeye.h>
#include <oechem.h>
#include <oegraphs.h>

using namespace OESystem;
using namespace OEChem;
using namespace OEGraphSim;

int main(int argc, char* argv[])
{
    if (argc != 3)
        OThrow.Usage("%s <queryfile> <targetfile>", argv[0]);

    oemolistream ifs;
    if (!ifs.open(argv[1]))
        OThrow.Fatal("Unable to open %s for reading", argv[1]);

    OEGraphMol qmol;
    if (!OEReadMolecule(ifs, qmol))
        OThrow.Fatal("Unable to read query molecule");

    OEFingerPrint qfp;
    OEMakeFP(qfp, qmol, OEFPTType::Path);
```

```
if (!ifs.open(argv[2]))
    OThrow.Fatal("Unable to open %s for reading", argv[2]);

std::cout.setf(std::ios::fixed, std::ios::floatfield);
std::cout.precision(3);

OEFingerprint tfp;
OEGraphMol tmol;
while (OEReadMolecule(ifs, tmol))
{
    OMakeFP(tfp, tmol, OEFPType::Path);
    std::cout << OETanimoto(qfp, tfp) << std::endl;
}

return 0;
}
```

In Listing 10 only the code block that is different from Listing 9 is shown.

In this example, it is assumed that the fingerprints are pre-calculated and stored in an OEB binary file as generic data attached to the corresponding molecules. The program loops over the file and accesses the pre-generated fingerprints or calculates them if they are not available.

The obvious advantage of this process is that the fingerprints one have to be generated once when the binary file is created. This can be significantly faster, than generating the fingerprints on-the-fly every time the program is executed.

See Also:

The *Storage and Retrieval* section shows an example of how to generate an OEB binary file which stores molecule along with their corresponding fingerprints.

Listing 10: Similarity calculation from OEB file

```
OEFingerprint tfp;
OEGraphMol tmol;
while (OEReadMolecule(ifs, tmol))
{
    if (tmol.HasData("PATH_FP"))
    {
        tfp = tmol.GetData<OEFingerprint>("PATH_FP");
    }
    else
    {
        OThrow.Warning("Unable to access fingerprint for %s", tmol.GetTitle());
        OMakeFP(tfp, tmol, OEFPType::Path);
    }
    std::cout << OETanimoto(qfp, tfp) << std::endl;
}
```

Listing 11 differs from Listing 9 in that it uses an `OEFPDatabase` object to store the generated fingerprints. The `OEFPDatabase` class is designed to perform in-memory fingerprint searches.

Listing 11: Similarity calculation with fingerprint database from file

```

OEFPPDatabase fpdb(qfp.GetFPTypeBase());

OEGraphMol tmol;
while (OEReadMolecule(ifs, tmol))
    fpdb.AddFP(tmol);

for (OEIter<OESimScore> si = fpdb.GetScores(qfp); si; ++si)
    std::cout << si->GetScore() << std::endl;

```

After building the fingerprint database, the scores can be accessed by the `OEFPPDatabase::GetScores` method. This will return an iterator over the similarity scores calculated.

Note: The `OEFPPDatabase` only stores fingerprints and not the molecules from which they are generated. A correspondence between a molecule and its fingerprint stored in the database can be established by using the index returned by the `OEFPPDatabase::AddFP` method.

See Also:

Listing 13 shows how to keep track of the correspondence between a fingerprint added to a `OEFPPDatabase` object and a molecule from which it is calculated.

In the last example (Listing 12), `OEFPPDatabase` is used again to store the fingerprints. If the fingerprint is read from the OEB input binary file, then it is directly added to the database, otherwise the fingerprint is generated on-the-fly when passing the `OEMolBase` molecule itself to the `OEFPPDatabase::AddFP` method.

Listing 12: Similarity calculation with fingerprint database from OEB

```

OEFPPDatabase fpdb(qfp.GetFPTypeBase());

OEFingerPrint tfp;
OEGraphMol tmol;
while (OEReadMolecule(ifs, tmol))
{
    if (tmol.HasData("PATH_FP"))
    {
        tfp = tmol.GetData<OEFingerPrint>("PATH_FP");
        fpdb.AddFP(tfp);
    }
    else
    {
        OEThrow.Warning("Unable to access fingerprint for %s", tmol.GetTitle());
        fpdb.AddFP(tmol);
    }
}

for (OEIter<OESimScore> si = fpdb.GetScores(qfp); si; ++si)
    std::cout << si->GetScore() << std::endl;

```

3.5.1 Sorted Search

Similarity searching based on a 2D representation of molecular structure (such as fingerprints) is one of the most common approaches for virtual screening. A molecule that is structurally similar to an active molecule is more likely to be active.

A virtual screening strategy involves going through a molecule database and calculating the similarity between a reference structure and each of the molecules, followed by ranking the similarity scores in descending order to identify molecules that are the most similar to the reference structure.

Listing 13 shows how to search a molecule database using the `OEFPDatabase::GetSortedScores` method to identify analogs based on their fingerprint similarity. After importing molecules and inserting their fingerprints into an `OEFPDatabase` database, the program reads reference molecules (in the *SMILES* format) from standard input. After parsing the *SMILES* string, the fingerprint database is searched to identify structures with the highest similarity scores. Finally, the *SMILES* string of the best hits are written to standard output.

Listing 13: Similarity search in memory

```
#include <vector>
#include <iostream>
#include "openeye.h"
#include "oechem.h"
#include "oegraphs.h"

using namespace OESystem;
using namespace OEChem;
using namespace OEGraphSim;

void LoadDatabase(const char *fname, OEFPDatabase& fpdb,
                 std::vector<OEGraphMol>& mvec)
{
    oemolistream ifs(fname);
    OEGraphMol mol;
    while (OEReadMolecule(ifs, mol))
    {
        fpdb.AddFP(mol);
        mvec.push_back(mol);
    }
}

int main(int argc, const char **argv)
{
    if (argc != 2)
        OEThrow.Usage("%s <database>", argv[0]);

    OEFPDatabase fpdb(OEFPType::Path);
    std::vector<OEGraphMol> mvec;
    LoadDatabase(argv[1], fpdb, mvec);

    // Read SMILES from stdin
    OEWallTimer sw;
    std::string line;
    OEGraphMol query;
    while (true)
    {
        std::cerr << "Enter SMILES> ";
        std::cerr.flush();
        getline(std::cin, line);

        if (line.size() == 0)
            return 0;
    }
}
```

```

query.Clear();
if (!OEParseSmiles(query, line))
{
    OThrow.Warning("Invalid SMILES string");
    continue;
}
sw.Start();
OEIter<OESimScore> si = fpdb.GetSortedScores(query, 5);
OThrow.Info("%f seconds to search %i fingerprints", sw.Elapsed(), mvec.size());
for (; si; ++si)
{
    std::string smiles;
    const OEGraphMol& hit = mvec[si->GetIdx()];
    OCreateIsoSmiString(smiles, hit);
    OThrow.Info("Tanimoto score %4.3f %s", si->GetScore(), smiles.c_str());
}
}

return 0;
}

```

As mentioned before, `OEFPDatabase` is a fingerprint container and does not store the corresponding molecule. Listing 13 therefore stores molecules in a separate container. When a fingerprint is added to the `OEFPDatabase`, its corresponding molecule is inserted into this container. When the `OEFPDatabase::GetSortedScores` returns the iterator over the best similarity scores, the associated index can be utilized to access the corresponding structure.

In the above example, the entire database was searched to identify structurally similar molecules. However, the user can also specify a segment of the database to be searched by providing a begin and end index.

See Also:

Examples of fingerprint searches in the API section:

- `OEFPDatabase::GetScores` method
- `OEFPDatabase::GetSortedScores` method

3.5.2 Searching with User-defined Similarity Measures

By default, the `Tanimoto` similarity is used when calling either the `OEFPDatabase::GetScores` method or the `OEFPDatabase::GetSortedScores` method. The user can set other types of similarity measures to be applied by calling the `OEFPDatabase::SetSimFunc` method with a value from the `OESimMeasure` namespace. Each of the constants from this namespace corresponds to one of the built-in similarity calculation methods.

There is also a facility to use user-defined similarity measures when searching a fingerprint database. The following example shows how a similarity calculation can be implemented by deriving from the `OESimFuncBase` class.

Formula: $Sim_{Simpson}(A, B) = \sqrt{\frac{bothAB}{\min(onlyA+bothAB, onlyB+bothAB)}}$

```

class SimpsonSimFunc : public OESimFuncBase
{
public:

    SimpsonSimFunc() : OESimFuncBase() {}

    float operator()(const OEFingerPrint& fpA, const OEFingerPrint& fpB) const

```

```

{
    unsigned int onlyA, onlyB, bothAB;
    OEGetBitCounts(fpA, fpB, &onlyA, &onlyB, &bothAB);
    if (onlyA + onlyB == 0)
        return 1.0;
    if (bothAB == 0)
        return 0.0;
    float sim = (float)bothAB;
    sim /= std::min((float)(onlyA+bothAB), (float)(onlyB+bothAB));
    return sim;
}

std::string GetSimTypeString() const { return std::string("Simpson"); }
OESimFuncBase *CreateCopy() const { return new SimpsonSimFunc(); }
};

```

After implementing the similarity calculation, it can be added to an `OEFPDatabase` object, henceforth this new similarity calculation will be used.

```

OEFPDatabase fpdb(OEFPType::Path);
fpdb.SetSimFunc(SimpsonSimFunc());

```

See Also:

- The *User-defined Similarity Measures* section

3.6 User-defined Fingerprint

The previous *Fingerprint Generation* chapter showed how to create circular, path and tree fingerprints with default parameters. These default parameters are calibrated on the Briem-Lessel [Briem-Lessel-2000], Hert-Willett [Hert-Willett-2004] and Grant [Grant-2006] benchmarks.

However, the *GraphSimTK* also provides facilities to construct user-defined fingerprints. When constructing a user-defined fingerprint, the following parameters have to be considered:

1. Atom and bond typing that define which atom and bond properties are encoded into the fingerprints (see the *Atom and Bond Typing* section)
2. Size of the fragments that are exhaustively enumerated during the fingerprint generation (see the *Fragment Size* section)
3. Size of the generated fingerprint (in bits) (see the *Fingerprint Size* section)

The following code snippet shows how to generate a 1024 bit long fingerprint that encodes paths from 0 up to 5 bonds in length with default atom and bond properties defined by the `OEFPAtomType::DefaultAtom` and `OEFPBondType::DefaultBond` constants, respectively.

```

unsigned int numbits = 1024;
unsigned int minbonds = 0;
unsigned int maxbonds = 5;
OEMakePathFP(fp, mol, numbits, minbonds, maxbonds,
             OEFPAtomType::DefaultAtom, OEFPBondType::DefaultBond);

```

Warning: Two fingerprints which are generated with different parameters will have different fingerprint types!

In Listing 14, two fingerprints are generated with different parameters, namely they have a different number of bits. This means that they also have different types, therefore, no similarity value can be calculated between them.

Listing 14: Example of different path fingerprint types

```
#include <openeye.h>
#include <oechem.h>
#include <oegraphs.h>

using namespace OEChem;
using namespace OEGraphSim;

int main(int, char* [])
{
    OEGraphMol mol;
    OEParseSmiles(mol, "c1cccc1");

    OEFingerprint fpA;
    unsigned int numbits = 1024;
    unsigned int minbonds = 0;
    unsigned int maxbonds = 5;
    OEMakePathFP(fpA, mol, numbits, minbonds, maxbonds,
                 OEFAtomType::DefaultAtom, OEFBondType::DefaultBond);

    numbits = 2048;
    OEFingerprint fpB;
    OEMakePathFP(fpB, mol, numbits, minbonds, maxbonds,
                 OEFAtomType::DefaultAtom, OEFBondType::DefaultBond);
    std::cout << "same fingerprint types = " << OEIsSameFPType(fpA, fpB) << std::endl;
    std::cout << OETanimoto(fpA, fpB) << std::endl;

    return 0;
}
```

The output of Listing 14 is the following:

```
same fingerprint types = False
Fatal: fingerprint type mismatch!
```

3.6.1 Atom and Bond Typing

Listing 15 shows how to generate fingerprints for two molecules with various atom and bond types (depicted in *Example molecules*). Reducing the number of atom and bond properties increases the similarity between the two molecules (*i.e.* their `Tanimoto` similarity). At the end, when only the topology of two molecules is considered, *i.e.*, whether or not their atoms and bonds belong to any ring system, the fingerprints of the two molecules become identical.

Listing 15: Similarity calculation with various atom/bond typing

```
#include <openeye.h>
#include <oechem.h>
#include <oegraphs.h>
```

```

using namespace OEChem;
using namespace OEGraphSim;

void PrintTanimoto(const OEMolBase& molA, const OEMolBase& molB,
                  unsigned int atype, unsigned int btype)
{
    OEFingerPrint fpA;
    OEFingerPrint fpB;
    unsigned int numbits = 2048, minb = 0, maxb = 5;
    OEMakePathFP(fpA, molA, numbits, minb, maxb, atype, btype);
    OEMakePathFP(fpB, molB, numbits, minb, maxb, atype, btype);
    std::cout << "Tanimoto(A,B) = " << OETanimoto(fpA, fpB) << std::endl;
}

int main(int, char*[])
{
    OEGraphMol molA;
    OEParseSmiles(molA, "Oc1c2c(cc(c1)CF)CCCC2");

    OEGraphMol molB;
    OEParseSmiles(molB, "c1ccc2c(c1)c(cc(n2)CCl)N");

    std::cout.setf(std::ios::fixed, std::ios::floatfield);
    std::cout.precision(3);

    PrintTanimoto(molA, molB, OEFPAAtomType::DefaultAtom, OEFPBondType::DefaultBond);
    PrintTanimoto(molA, molB, OEFPAAtomType::DefaultAtom|OEFPAAtomType::EqAromatic,
                  OEFPBondType::DefaultBond);
    PrintTanimoto(molA, molB, OEFPAAtomType::Aromaticity, OEFPBondType::DefaultBond);
    PrintTanimoto(molA, molB, OEFPAAtomType::InRing, OEFPBondType::InRing);

    return 0;
}

```

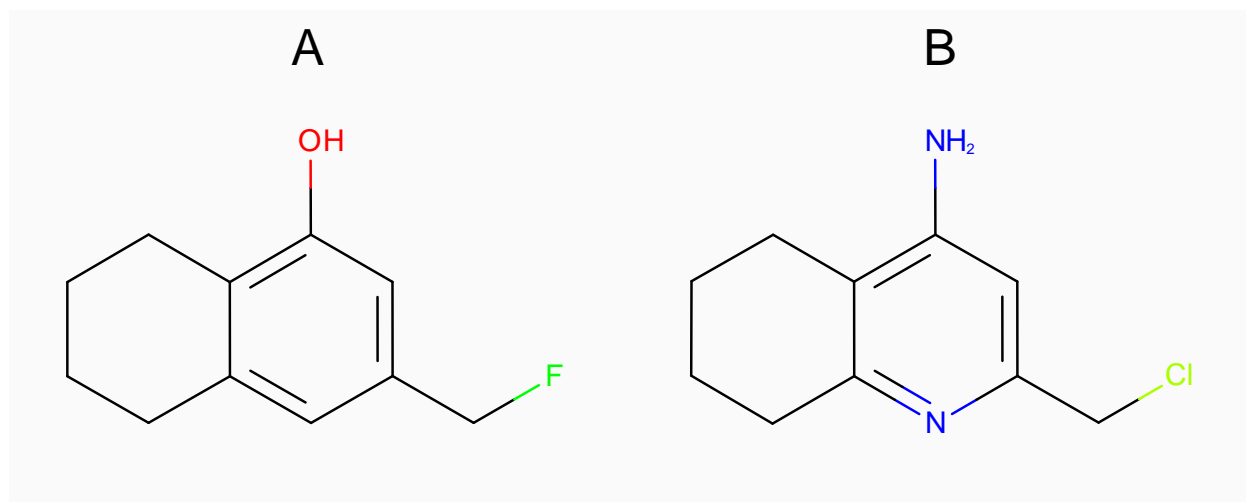


Figure 3.5: Example molecules

The output of Listing 15 is the following:

Tanimoto(A, B) = 0.166
 Tanimoto(A, B) = 0.241
 Tanimoto(A, B) = 0.592
 Tanimoto(A, B) = 1.000

Table: Atom typing options and Table: Bond typing options list the currently available typing options.

Table 3.3: Atom typing options

atom typing constant	encoded atom property
<code>OEFPAtomType::Aromaticity</code>	<code>OEAtomBase::IsAromatic</code>
<code>OEFPAtomType::AtomicNumber</code>	<code>OEAtomBase::GetAtomicNum</code>
<code>OEFPAtomType::Chiral</code>	<code>OEAtomBase::IsChiral</code>
<code>OEFPAtomType::FormalCharge</code>	<code>OEAtomBase::GetFormalCharge</code>
<code>OEFPAtomType::HCount</code>	<code>OEAtomBase::GetTotalHCount</code>
<code>OEFPAtomType::HvyDegree</code>	<code>OEAtomBase::GetHvyDegree</code>
<code>OEFPAtomType::Hybridization</code>	<code>OEAtomBase::GetHyb</code>
<code>OEFPAtomType::InRing</code>	<code>OEAtomBase::IsInRing</code>

Table 3.4: Atomic number modifiers

<code>OEFPAtomType::EqAromatic</code>	
<code>OEFPAtomType::EqHalogen</code>	
<code>OEFPAtomType::EqHBondAcceptor</code>	
<code>OEFPAtomType::EqHBondDonor</code>	

Table 3.5: Bond typing options

bond typing constant	encoded bond property
<code>OEFPBondType::BondOrder</code>	<code>OEAtomBase::GetOrder</code>
<code>OEFPBondType::Chiral</code>	<code>OEAtomBase::IsChiral</code>
<code>OEFPBondType::InRing</code>	<code>OEAtomBase::IsInRing</code>

See Also:

- `OEFPAtomType` namespace
- `OEFPBondType` namespace

3.6.2 Fragment Size

Circular, path and tree-based fingerprint generation involves molecular graph traversal to identify all unique radial, linear or branched fragments, respectively. When a path or tree fingerprint is initialized, the minimum and maximum number of bonds of the fragments that are encoded into the fingerprint can be specified.

Table 3.6: Example of enumerated path fragments with increasing number of bonds

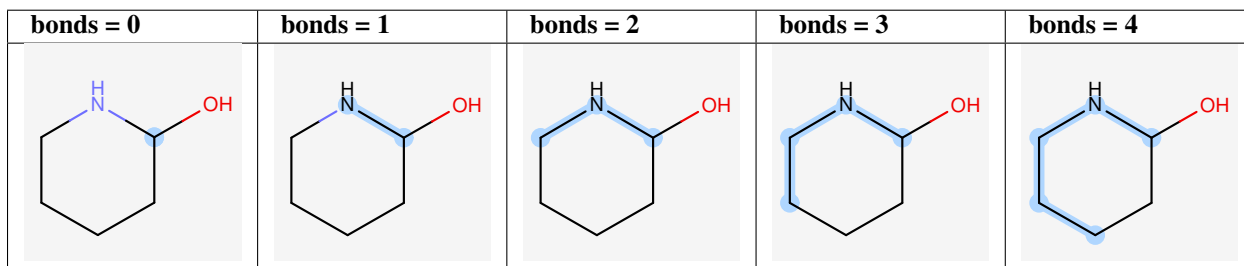
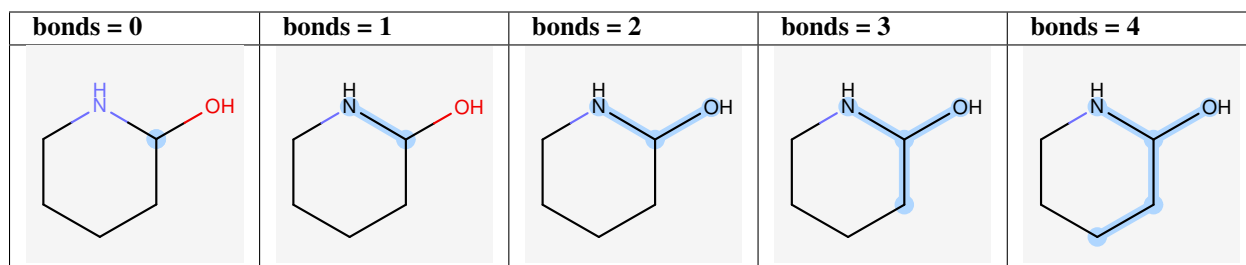
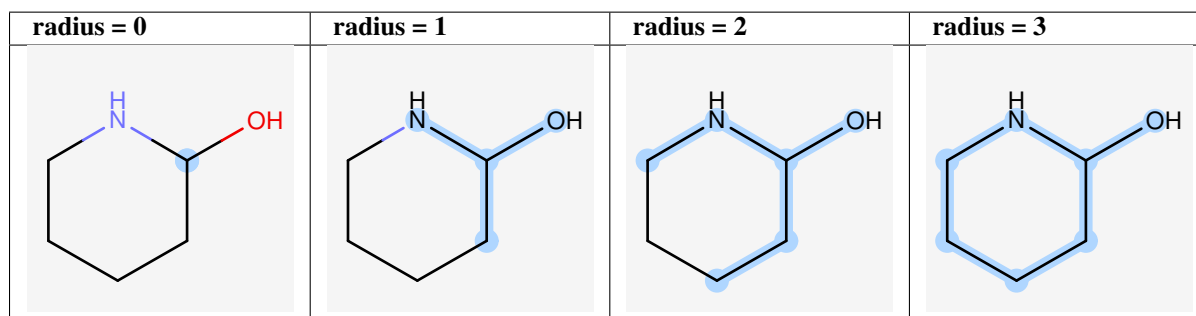


Table 3.7: Example of enumerated tree fragments with increasing number of bonds



In case of a circular fingerprint, the minimum and maximum radius of the enumerated fragments can be specified.

Table 3.8: Example of enumerated circular fragments with increasing radius



For example, when generating a fingerprint of the molecule shown in *Figure: Example Molecule* with minimum and maximum length set to 0 and 3, respectively, only paths listed in the first four rows in *Table: Enumerated Paths*, are encoded into the fingerprint.

Table 3.9: Enumerated paths

Path length (in bonds)	Generated Unique Paths
0	C, N, O
1	C-C, C-N, C-O
2	C-C-C, C-C-N, C-C-O, C-N-C, N-C-O
3	C-C-C-C, C-C-C-N, C-C-C-O, C-C-N-C, C-N-C-O,
4	C-C-C-C-C, C-C-C-C-N, C-C-C-C-O, C-C-C-N-C, C-C-N-C-C, O-C-N-C-C
5	C-C-C-C-C-N, C-C-C-C-C-O, C-C-C-C-N-C, C-C-C-N-C-C, C-C-C-N-C-O

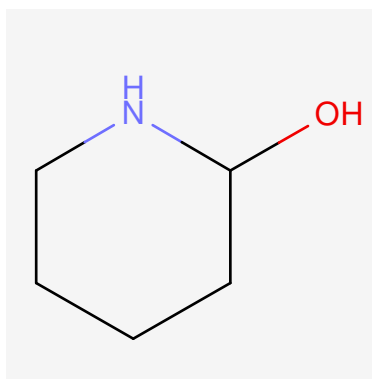


Figure 3.6: Example molecule

Figure: Example of enumerated paths depicts the six unique paths of length four that are generated for the example molecule. Each unique path is encoded only once without considering its frequency.

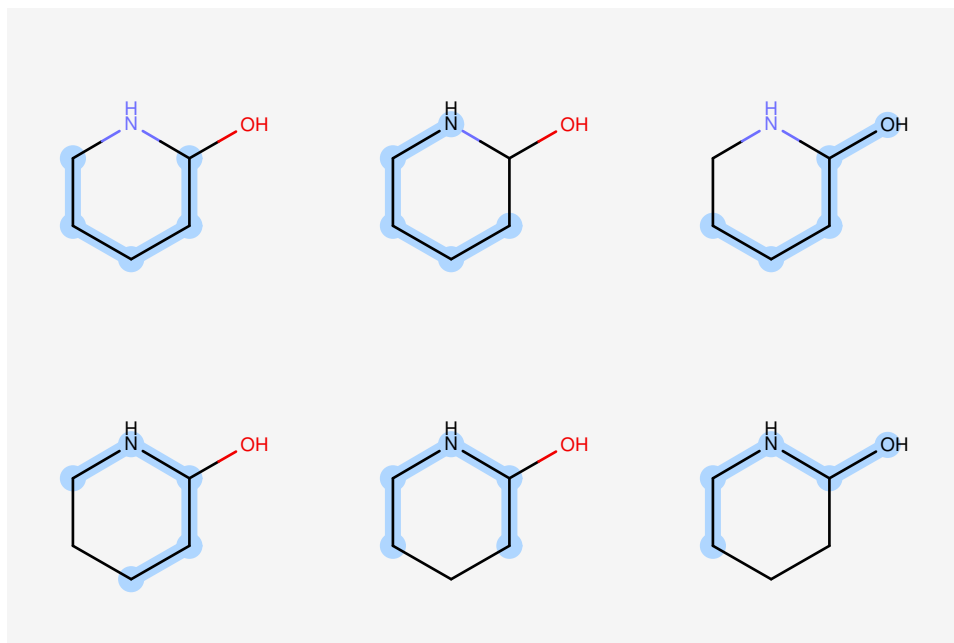


Figure 3.7: Example of enumerated paths

In the example shown in Listing 16, fingerprints with various minimum and maximum path length are generated for pyrrole and pyridine. When enumerating only paths that are shorter than four bonds, the fingerprints generated for the two molecules are identical. Since the four bond-length pattern `CCCC` is present in pyridine but not in pyrrole, the fingerprints become different, resulting in a smaller Tanimoto similarity score.

Listing 16: Similarity calculation with various path lengths

```
#include <openeye.h>
#include <oechem.h>
#include <oegraphs.h>

using namespace OEChem;
using namespace OEGraphSim;

void PrintTanimoto(const OEMolBase& molA, const OEMolBase& molB,
                  unsigned int minb, unsigned int maxb)
{
    OEFingerprint fpA;
    OEFingerprint fpB;
    unsigned int numbits = 2048;
    unsigned int atype = OEFPAAtomType::DefaultAtom;
    unsigned int btype = OEFPBondType::DefaultBond;
    OEMakePathFP(fpA, molA, numbits, minb, maxb, atype, btype);
    OEMakePathFP(fpB, molB, numbits, minb, maxb, atype, btype);
    std::cout << "Tanimoto(A,B) = " << OETanimoto(fpA, fpB) << std::endl;
}

int main(int, char*[])
{
    OEGraphMol molA;
```

```

OEParseSmiles(molA, "c1ccncc1");
OEGraphMol molB;
OEParseSmiles(molB, "c1cc[nH]c1");

std::cout.setf(std::ios::fixed, std::ios::floatfield);
std::cout.precision(3);

PrintTanimoto(molA, molB, 0, 3);
PrintTanimoto(molA, molB, 1, 3);
PrintTanimoto(molA, molB, 0, 4);
PrintTanimoto(molA, molB, 0, 5);

return 0;
}

```

The output of Listing 16 is the following:

```

Tanimoto(A,B) = 1.000
Tanimoto(A,B) = 1.000
Tanimoto(A,B) = 0.950
Tanimoto(A,B) = 0.731

```

3.6.3 Fingerprint Size

The previous sections explain how the atom and bond typing and encoded fragment size can effect the similarity scores. Selecting an adequate fingerprint size is also very crucial. The number of unique circular, path or tree fragments present in molecular structures can be extremely large, therefore the generated fragments have to be hashed into the fixed-length fingerprint. This means that a bit in a fingerprint does not correspond to a unique pattern exclusively (as it does in *structural key*). Also a bit has no particular structural meaning, *i.e.*, each bit represents the presence of a number of structural patterns.

The smaller the size of the fingerprints, the more dense they become, raising the probability of collisions. A collision occurs when different fragments are mapped to the same bit. This will inherently result in information loss and weaken the power to discriminate between structurally similar and dissimilar molecules. On the other hand, when the size of the fingerprints is too large they become very sparse, which will reduce information loss. However, the time spent to calculate similarity scores will increase.

The following table shows the number of unique paths generated for benzylpenicillin (depicted in *Figure: Benzylpenicillin*).

Note: The more atom and bond properties that are taken into account and the larger the size of paths to enumerate, the larger the size of the fingerprint has to be in order to encode the enumerated fragments without a significant number of bit collisions.

Table 3.10: Number of unique paths generated for Benzylpenicillin

Atom/Bond typing	path 0-3	path 0-5	path 0-7
AtomicNumber, BondOrder	56	149	297
AtomicNumber HvyDegree, BondOrder	111	265	453
AtomicNumber HvyDegree Aromaticity, BondOrder InRing	126	297	499
DefaultAtom, DefaultBond	147	362	617

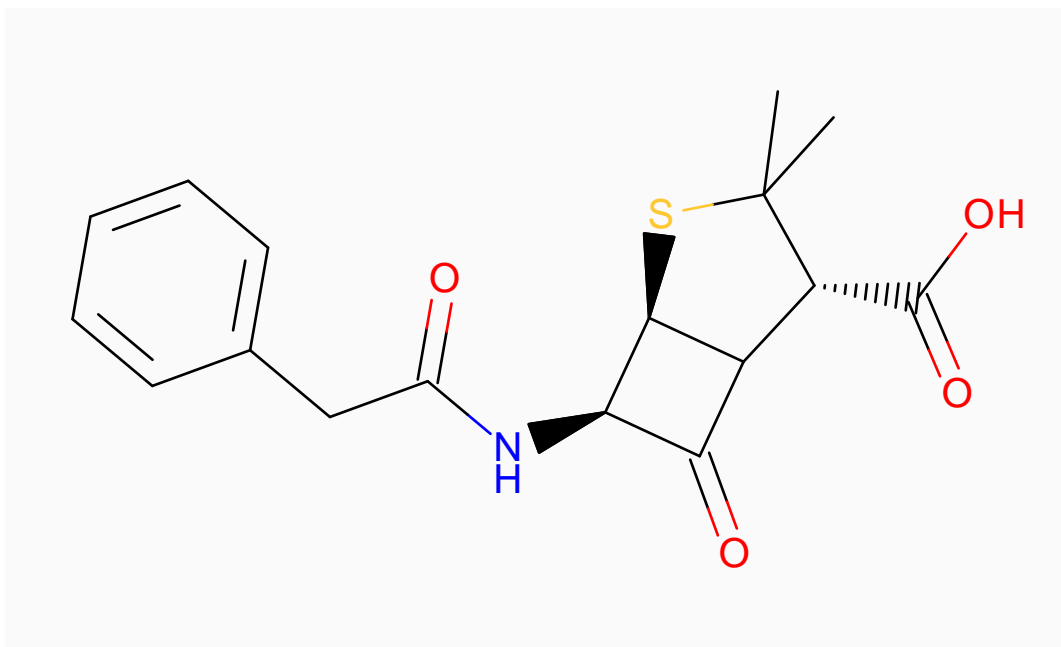


Figure 3.8: Benzylpenicillin

3.7 Fingerprint Coverage

Fingerprints are usually generated by enumerating various fragments of a molecule and then hashing them into a fixed-length bitvector. The `OEGetFPCoverage` function provides access to these fragments by returning an iterator over `OEAtomBondSet` objects, each of which storing the atoms and bonds of a specific fragment.

The following example shows how to retrieve the unique fragments that are enumerated when generating a path fingerprint. The obtained fragments are depicted in *Table: Example of path fragments*

Listing 17: Example of accessing patterns encoded into a fingerprint

```
#include <openeye.h>
#include <oesystem.h>
#include <oechem.h>
#include <oegraphs.h>

using namespace OESystem;
using namespace OEChem;
using namespace OEGraphSim;

int main()
{
    OEGraphMol mol;
    OEParseSmiles(mol, "CCNCC");

    const OEFPTTypeBase* fptype = OEGetFPTType(OEFPTType::Path);

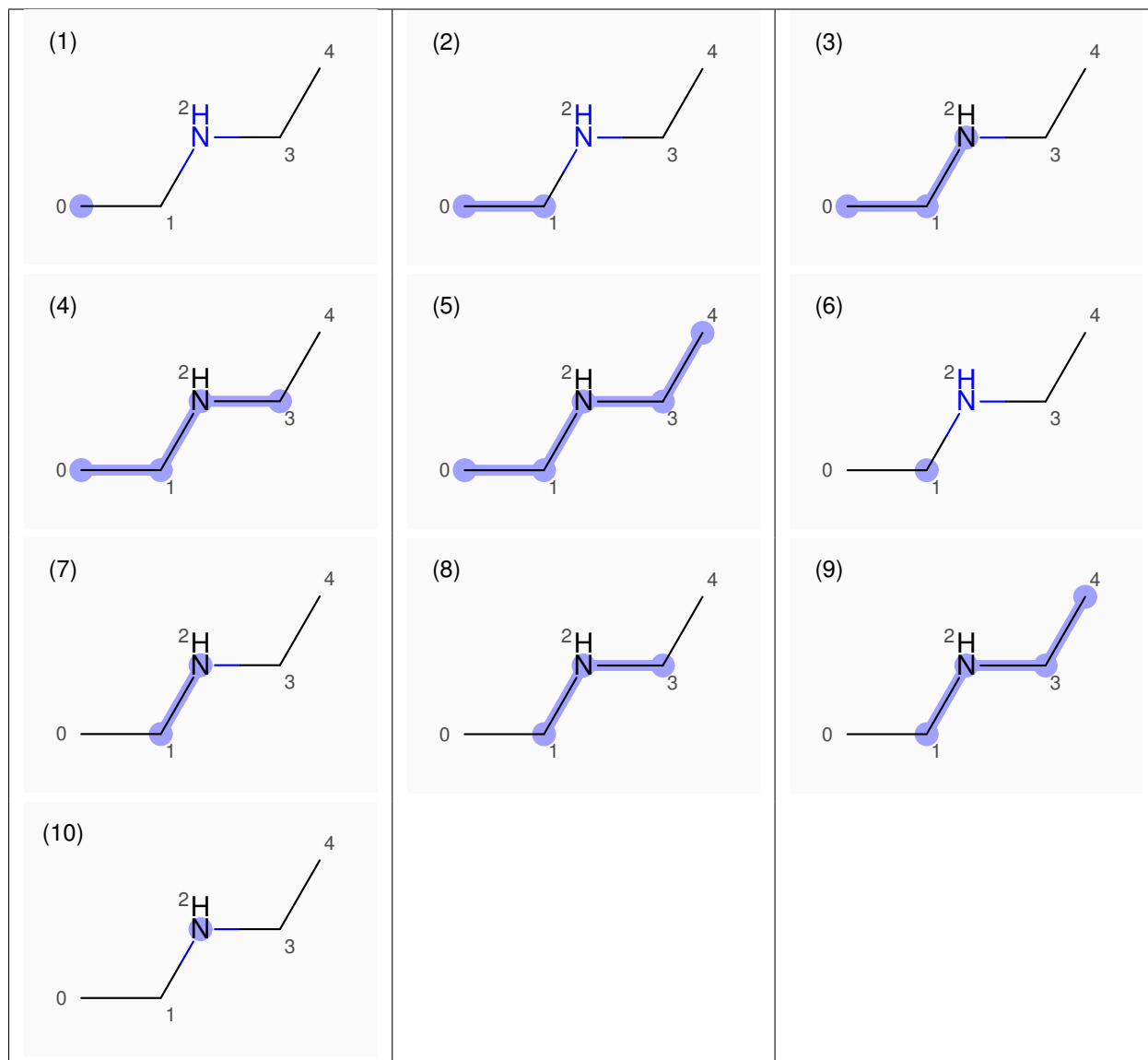
    bool unique = true;
    unsigned int idx = 0;
    for (OEIter<OEAtomBondSet> abset = OEGetFPCoverage(mol, fptype, unique); abset; ++abset)
```

```
{
  idx++;
  printf("%2d ", idx);
  for (OEIter<OEAtomBase> a = abset->GetAtoms(); a; ++a)
  {
    printf(" %d %s", a->GetIdx(), OEGetAtomicSymbol(a->GetAtomicNum()));
  }
  printf("\n");
}
return 0;
}
```

The output of Listing 17 is the following:

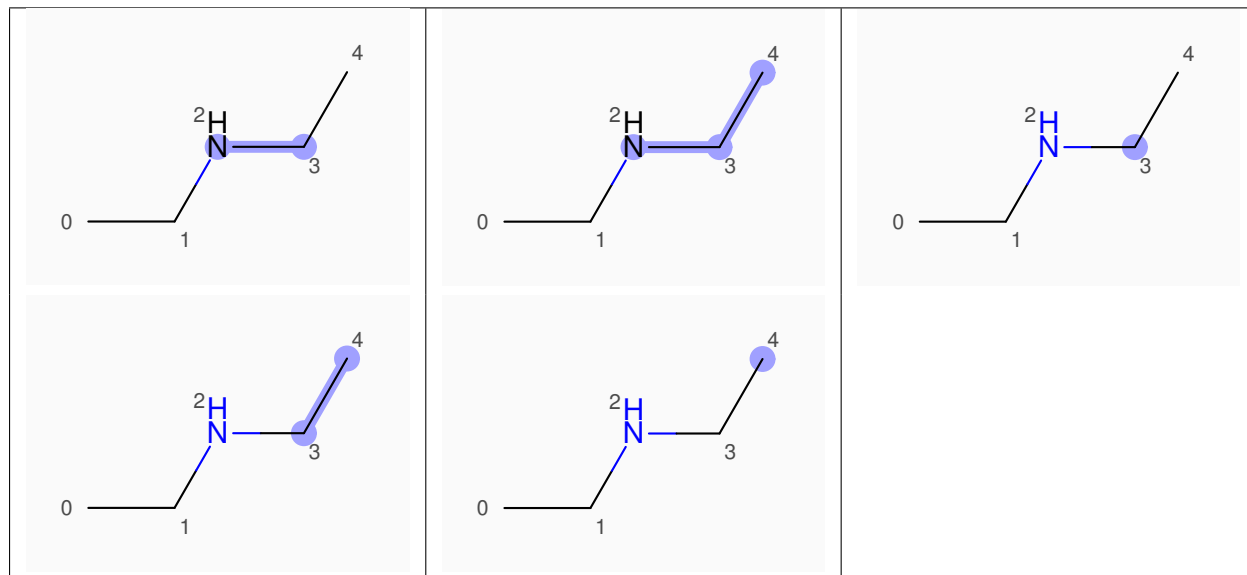
```
1  0 C
2  0 C  1 C
3  0 C  1 C  2 N
4  0 C  1 C  2 N  3 C
5  0 C  1 C  2 N  3 C  4 C
6  1 C
7  1 C  2 N
8  1 C  2 N  3 C
9  1 C  2 N  3 C  4 C
10 2 N
```

Table 3.11: Example of unique path fragments. The numbers displayed next to atoms are the atom indices.



The `OEGetFPCoverage` function in the Listing 17 example is called with a unique options. This means that it returns only unique fragments, where a fragment (*i.e.* subgraph) is considered unique, if it differs from all other subgraphs identified previously by at least one atom or bond. For example, executing the same code with a non-unique option would generate five additional paths depicted in *Table: Example of additional non-unique path fragments*

Table 3.12: Example of additional non-unique path fragments. The numbers displayed next to atoms are the atom indices.



See Also:

- `OEGetFPOverlap` function
- *Fingerprint Overlap* chapter

3.8 Fingerprint Overlap

The `OEGetFPOverlap` function provides access to the fragments of two molecules that are considered equivalent based on a specific fingerprint type. This means that the returned fragment-pairs set the same bit “on” when fingerprints are generated. The following example shows how to retrieve the common five bond-length patterns of two molecules.

Listing 18: Example of accessing common patterns based on a fingerprint

```
#include <openeye.h>
#include <oesystem.h>
#include <oechem.h>
#include <oegraphs.h>

using namespace OESystem;
using namespace OEChem;
using namespace OEGraphSim;

int main()
{
    OEGraphMol pmol;
    OEParseSmiles(pmol, "c1cnc2c(c1)CC(CC2O)CF");

    OEGraphMol tmol;
    OEParseSmiles(tmol, "c1cc2c(cc1)CC(CC1)CC2N");
```

```

const OEFPTypeBase* fptype = OEGetFPType("Tree,ver=2.0.0,size=4096,bonds=5-5,atype=AtmNum|HvyDeg|E

unsigned int idx = 0;
for (OEIter<OEMatchBase> match = OEGetFPOverlap(pmol, tmol, fptype); match; ++match)
{
  idx++;
  printf("match %2d:", idx);
  for (OEIter<OEMatchPair<OEAtomBase> > mpair = match->GetAtoms(); mpair; ++mpair)
  {
    printf(" %d%s-%d%s", mpair->pattern->GetIdx(), OEGetAtomicSymbol(mpair->pattern->GetAtomicNum(
      mpair->target->GetIdx(), OEGetAtomicSymbol(mpair->target->GetAtomicNum()));
  }
  printf("\n");
}
return 0;
}

```

The first three matches returned by the Listing 18 are depicted in the next table. The output of code is the following:

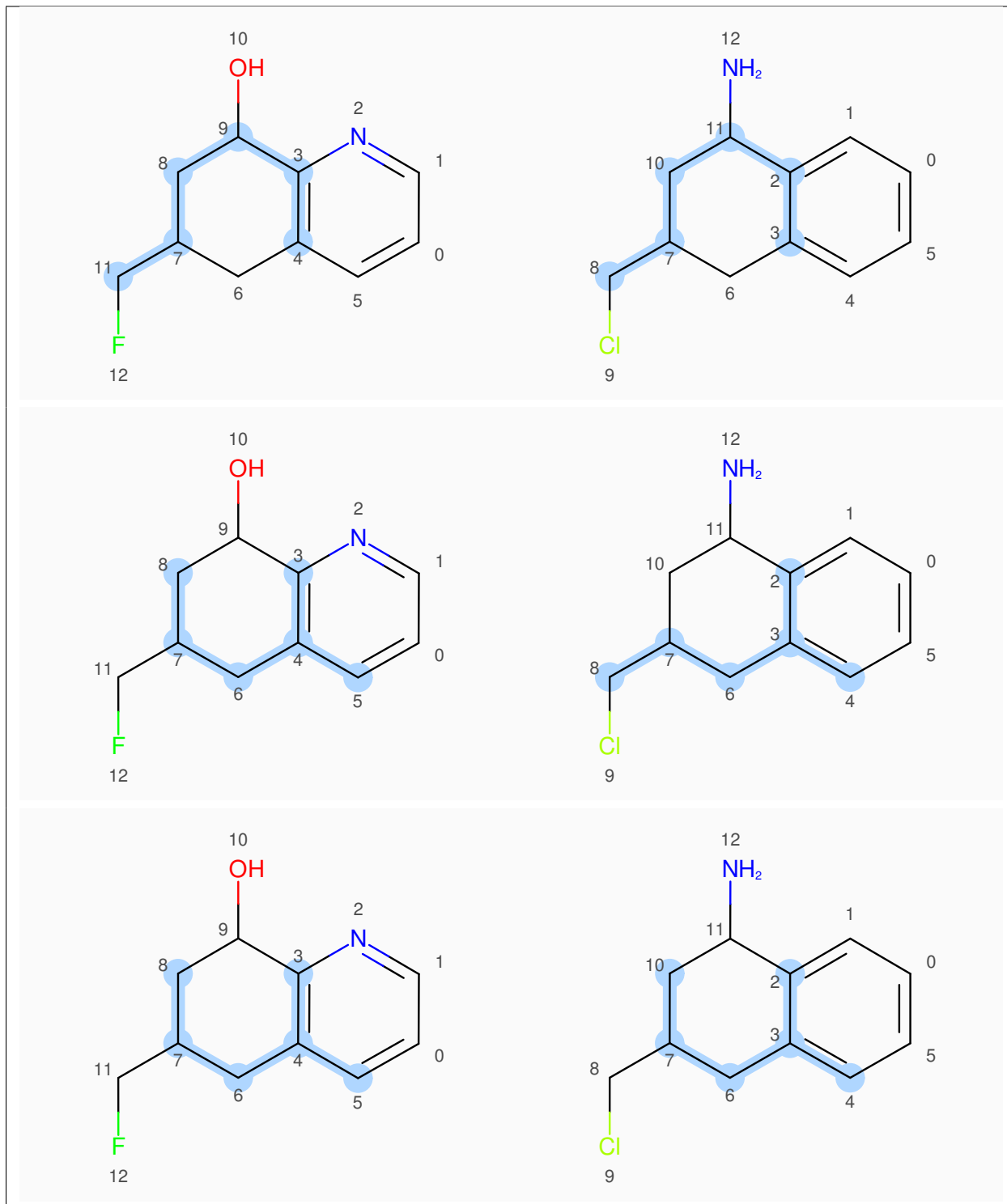
```

match 1: 3C-2C  9C-11C  4C-3C  8C-10C  7C-7C  11C-8C
match 2: 3C-2C  4C-3C  5C-4C  6C-6C  7C-7C  8C-8C
match 3: 3C-2C  4C-3C  5C-4C  6C-6C  7C-7C  8C-10C
match 4: 3C-2C  4C-3C  5C-4C  6C-6C  7C-7C  11C-8C
match 5: 3C-2C  4C-3C  5C-4C  6C-6C  7C-7C  11C-10C
match 6: 3C-2C  9C-11C  4C-3C  6C-6C  7C-7C  11C-8C

```

... truncated ...

Table 3.13: Example of matches returned by the OEGetFPOverlap function. The numbers depicted next to the atoms are the atom indices.



Warning: Even though the `OEGetFPOverlap` function returns an iterator of `OEMatchBase` objects, they are not matches in the traditional sense, i.e. the atom-pair and bond-pair correspondences between the pattern and the target atoms and bonds are not guaranteed. See example depicted below.

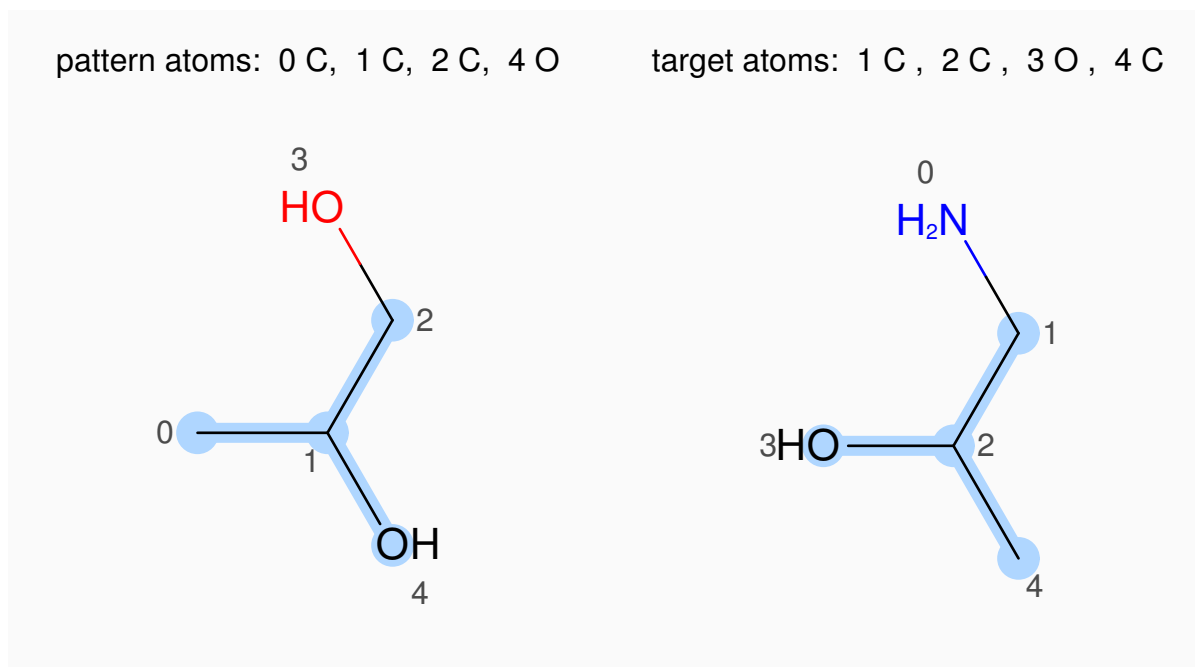


Figure 3.9: The two highlighted patterns set the same bit when fingerprints are generated, but the returned match is not pair-wise. The numbers depicted next to the atoms are the atom indices.

See Also:

- `OEGetFPCoverage` function
- *Fingerprint Coverage* chapter

4.1 OESystem Classes

4.1.1 OEBitVector

`class OEBitVector`

This class represents *OEBitVector*.

The `OEBitVector` class is used to represent a resizable bitmap. These are commonly used to store *fingerprints* in chemoinformatics.

Constructors

```
OEBitVector()  
OEBitVector(unsigned int size)  
OEBitVector(const OEBitVector &src)
```

When constructed with a `size` argument, this specifies the initial number of bits in the `OEBitVector`. When constructed without an argument, the default number of bits is architecture dependent, 32 bits on 32-bit hosts and 64 bits on 64-bit hosts. Initially, all the bits are set to zero.

```
OEBitVector(OERandom &rand, unsigned int size)
```

Create a `OEBitVector` with `size` bits randomly initialized from the `rand` object.

```
OEBitVector(const unsigned char *data, unsigned int size)
```

Using this constructor is equivalent to creating an default constructed `OEBitVector` and then immediately calling `OEBitVector::SetData` with the `data` and `size` arguments.

operator<

```
bool operator<(const OEBitVector &other) const
```

operator=

```
OEBitVector &operator=(const OEBitVector &src)
```

operator&=

```
OEBitVector &operator&=(const OEBitVector &)
```

operator-=

```
OEBitVector &operator-=(const OEBitVector &)
```

operator[]

```
bool operator[](unsigned int bit) const
```

operator^=

```
OEBitVector &operator^=(unsigned int bit)  
OEBitVector &operator^=(const OEBitVector &)
```

operator|=

```
OEBitVector &operator|=(unsigned int bit)  
OEBitVector &operator|=(const OEBitVector &)
```

ClearBits

```
void ClearBits()
```

Clears all of the bits of the `OEBitVector` to zero.

CountBits

```
unsigned int CountBits() const
```

Counts the number of bits that are set to one, in the `OEBitVector` object.

FirstBit

```
int FirstBit() const
```

Returns the index of the first bit, *i.e.* the set bit with the lowest index, in the `OEBitVector`. Bit position indices are numbered from zero, and have a maximum of `OEBitVector::GetSize - 1`. If the `OEBitVector` has no bits set, *i.e.* `OEBitVector::IsEmpty` returns `true`, this method return will return the value `-1`.

FromHexString

```
void FromHexString(const char *bvs)
void FromHexString(const std::string &bvs)
```

Converts a hexadecimal string, in either upper, lower or mixed case, into an `OEBitVector`. The `OEBitVector` is resized to four times the length of the specified string, each character or hex digit corresponding to four bits. All characters outside the ranges `'0' .. '9'`, `'A' .. 'F'` and `'a' .. 'f'` are treated as zero.

GetData

```
const unsigned char *GetData() const
```

Returns a pointer to the internal storage of the `OEBitVector`'s bitmap. This pointer to a sequence of at least $(\text{OEBitVector::GetSize} + 7) / 8$ consecutive bytes. Bit index zero corresponds to the least significant bit of the first byte. All bits beyond `OEBitVector::GetSize` in the last byte are guaranteed to be zero. The `OEBitVector::GetData` method always returns a valid non-NULL pointer even if the number of bits is zero.

GetSize

```
unsigned int GetSize() const
```

Returns the size, in bits, of the `OEBitVector`. An `OEBitVector` may potentially contain zero bits.

IsBitOn

```
bool IsBitOn(unsigned int bit) const
```

Tests whether the bit as the specified position/index of an `OEBitVector` is set. Bit position indices are numbered from zero, and have a maximum of `OEBitVector::GetSize - 1`. This method returns `false` for all indices greater than or equal to `OEBitVector::GetSize`.

IsEmpty

```
bool IsEmpty() const
```

Returns `true` if all the bits of an `OEBitVector` are zero. This is equivalent to, but much more efficient than, testing that `OEBitVector::CountBits == 0`. An `OEBitVector` of zero size is considered empty.

LastBit

```
int LastBit() const
```

Returns the index of the last bit, *i.e.* the set bit with the highest index, in the `OEBitVector`. Bit position indices are numbered from zero, and have a maximum of `OEBitVector::GetSize - 1`. If the `OEBitVector` has no bits set, *i.e.* `OEBitVector::IsEmpty` returns `true`, this method return will return the value `-1`.

NegateBits

```
void NegateBits()
```

Inverts all of the bits of an `OEBitVector`.

NextBit

```
int NextBit(unsigned int) const
```

Returns the next set bit after the specified bit position, *i.e.* the set bit with the lowest index greater than the argument. If there are no such set bits, or the value of bit is greater than or equal to `OEBitVector::GetSize`, this method returns the value `-1`.

PrevBit

```
int PrevBit(unsigned int) const
```

Returns the previous set bit before the specified bit position, *i.e.* the set bit with the highest index less than the argument. If there are no such set bits, this method returns the value `-1`. If the specified value of bit is greater than or equal to `OEBitVector::GetSize`, this method returns the same values as `OEBitVector::LastBit`.

SetBitOff

```
void SetBitOff(unsigned int bit)
```

Clears the bit at the specified bit position/index of an `OEBitVector`. Bit position indices are numbered from zero, and have a maximum of `OEBitVector::GetSize - 1`. If the value of 'bit' is greater than or equal to `OEBitVector::GetSize`, the bitmap is resized to 'bit'+1 bits. All of the new bits are initialized to zero.

SetBitOn

```
void SetBitOn(unsigned int bit)
```

Tests whether the bit as the specified position/index of an `OEBitVector` is set. Bit position indices are numbered from zero, and have a maximum of `OEBitVector::GetSize - 1`. This method returns `false` for all indices greater than or equal to `OEBitVector::GetSize`.

SetData

```
void SetData(const unsigned char *data, unsigned int nbits)
```

Initialize the `OEBitVector` by the binary data pointed to by `data`, possibly resizing the `OEBitVector` to accommodate the `nbits` of data. `data` should point to at least `nbits/8` bytes of data. If a non-multiple of 8 bits is to be copied it is the users responsibility to make sure the remainder of bits in the last byte are zero'd out.

SetRangeOff

```
void SetRangeOff(unsigned int start, unsigned int end)
```

SetRangeOn

```
void SetRangeOn(unsigned int start, unsigned int end)
```

SetSize

```
void SetSize(unsigned int bits)
```

Resizes an `OEBitVector` to the specified number of bits. A size argument of zero is allowed. If this method increases the size of an `OEBitVector` all of the new bit positions are initialized to zero.

ToHexString

```
void ToHexString(std::string &bvs) const
```

Generates a hexadecimal string representation of an `OEBitVector`. Each “nibble” of four bits is converted into an uppercase hexadecimal character. Hexadecimal strings can only represent bitmaps that have a multiple of four bits. For `OEBitVectors` that are not a multiple of four, the remaining bits of the most significant nibble, *i.e.* the first character, are assumed to be zero.

ToggleBit

```
void ToggleBit(unsigned int bit)
```

Inverts the bit at the specified bit position/index of an `OEBitVector`. Bit position indices are numbered from zero, and have a maximum of `OEBitVector::GetSize - 1`. If the value of ‘bit’ is greater than or equal to `OEBitVector::GetSize`, the bitmap is resized to ‘bit’+1 bits. All of the new bits other than the one specified in the argument, *i.e.* the last, are initialized to zero.

4.2 OEGraphSim Classes

4.2.1 OEFPDatabase

```
class OEFPDatabase
```

The `OEFPDatabase` class is designed to perform rapid in-memory fingerprint searches. Each `OEFPDatabase` object is associated with a fingerprint type (`OEFPTypeBase`) that is specified when the database is constructed. An `OEFPDatabase` object can only store fingerprints (`OEFPFingerprint`) with this specified type.

Constructors

```
OEFPDatabase(unsigned int fptype)
```

Creates an `OEFPDatabase` object that can store `OEFPFingerprint` objects with a given type.

fptype The type of the `OEFPFingerprint` object stored in the `OEFPDatabase`. This value has to be from the `OEFPType` namespace.

```
OEFPDatabase(const OEGraphSim::OEFPTypeBase *)
```

Creates an `OEFPDatabase` object that can store `OEFPFingerprint` objects with `OEFPTypeBase` type.

Note: By default, an `OEFPDatabase` object is constructed with:

- no cutoff value
- `Tanimoto` similarity measure
- descending order to return scores when calling the `OEFPDatabase::GetSortedScores` method

These default values can be altered by the following methods:

- `OEFPDatabase::SetCutoff`
- `OEFPDatabase::SetSimFunc`

AddFP

```
unsigned int AddFP(const OEChem::OEMolBase &mol)
```

Generates an `OEFPFingerprint` object from the `OEMolBase` molecule with the fingerprint type of the `OEFPDatabase`. The generated `OEFPFingerprint` object is then inserted into the database returning its index. This method will return `-1`, if the fingerprint generation was unsuccessful.

```
unsigned int AddFP(const OEGraphSim::OEFPFingerprint &fp)
```

Creates a copy of the `OEFPFingerprint` object, inserts it into the database, and then returns its index. If the type of the passed fingerprint was different from the type of the database, than the insertion is unsuccessful and this method will return `-1`.

Note: The index returned by the `OEFPDatabase::AddFP` method is a unique number starting from zero. This index can be used as a reference number to associate the fingerprint with the molecule from which is it generated.

ClearCutoff

```
void ClearCutoff()
```

Removes the cutoff value previously set by the `OEFPDatabase::SetCutoff` method. After clearing the cutoff value `OEFPDatabase::HasCutoff` method will return false.

See Also:

- `OEFPDatabase::GetCutoff`
- `OEFPDatabase::HasCutoff`
- `OEFPDatabase::SetCutoff`

GetCutoff

```
float GetCutoff() const
```

Returns the cutoff value previously set by the `OEFPDatabase::SetCutoff` method.

See Also:

- `OEFPDatabase::ClearCutoff`
- `OEFPDatabase::HasCutoff`
- `OEFPDatabase::SetCutoff`

GetFPTypeBase

```
const OEGraphSim::OEFPTypeBase *GetFPTypeBase() const
```

Returns the fingerprint type of the `OEFPDatabase` object.

GetFingerPrints

```
OESystem::OEIterBase<const OEGraphSim::OEFingerPrint> *GetFingerPrints() const
```

Returns a iterator pointer over fingerprints (`OEFingerPrint`) stored in the `OEFPDatabase` object. The returned `OEFingerPrint` objects can only be accessed by *const* methods or functions, *i.e.*, they can not be modified.

GetScores

```
OESystem::OEIterBase<OEGraphSim::OESimScore> *
  GetScores(const OEChem::OEMolBase &mol, unsigned int bgn=0,
            unsigned int end=0) const
```

```
OESystem::OEIterBase<OEGraphSim::OESimScore> *
  GetScores(const OEGraphSim::OEFingerPrint &fp, unsigned int bgn=0,
            unsigned int end=0) const
```

Performs a similarity calculation between a given molecule or a fingerprint and the fingerprints stored in the `OEFPDatabase` object. It returns an iterator over the calculated similarity scores (`OESimScore`). Each `OESimScore` holds a similarity score and index of the corresponding fingerprint of the database.

If the method is called with an `OEMolBase` object, then a fingerprint is generated from this molecule before looping over the fingerprints of the database and calculating similarities. If it is called with an `OEFPDatabase` object, then its type has to match with the type of the `OEFPDatabase`.

The `bgn` and `end` arguments define the segment of the database on which the similarity calculation will take place. If both of these parameters are omitted (or set to zero), then the similarity calculation is performed on the entire fingerprint database.

Note: By default, the `OEFPDatabase::GetScores` methods calculates `Tanimoto` similarity scores and will always return these scores in the order in which the corresponding `OEFPDatabase` objects are added to the database.

By using the `OEFPDatabase::SetSimFunc` method, similarity measures other than the default can be performed. The behavior of `OEFPDatabase::GetScores` is also influenced by the cutoff value (`OEFPDatabase::SetCutoff`) of the database and the order of the scores specified by the `OEFPDatabase::SetSimFunc` method. If a cutoff value was set, then the `OEFPDatabase::GetScores` method will

- return scores that are equal to or larger than the specified cutoff value if the order is descending
- return scores that are equal to or smaller than the specified cutoff value if the order is ascending

See Also:

- `OEFPDatabase::SetSimFunc` method
- `OEFPDatabase::SetCutoff` method

Examples:

- Calculates the `Tanimoto` similarity on the first 100 entries of the database and returns scores that are equal to or larger than 0.1.

```
bool descending = true;
fpdb.SetSimFunc(OESimMeasure::Tanimoto, descending);
fpdb.SetCutoff(0.1);
for (OEIter<OESimScore> si = fpdb.GetScores(qfp, 0, 100); si; ++si)
    std::cout << si->GetScore() << std::endl;
```

- Calculates the `Tversky` similarity (with $\alpha = 0.9$ and $\beta = 0.1$) on the entire database and returns all scores.

```
fpdb.SetSimFunc(OETverskySim(0.9));
for (OEIter<OESimScore> si = fpdb.GetScores(qfp); si; ++si)
    std::cout << si->GetScore() << std::endl;
```

- Calculates the `Dice` similarity beginning at the 100th entry of the database and returns scores that are equal to or smaller than 0.5.

```
descending = true;
fpdb.SetSimFunc(OESimMeasure::Dice, !descending);
fpdb.SetCutoff(0.5);
for (OEIter<OESimScore> si = fpdb.GetScores(qfp, 100); si; ++si)
    std::cout << si->GetScore() << std::endl;
```

GetSortedScores

```
OESystem::OESimScore *
  GetSortedScores(const OEChem::OEMolBase &mol, unsigned int limit=0,
                 unsigned int bgn=0, unsigned int end=0) const
```

```
OESystem::OESimScore *
  GetSortedScores(const OEGraphSim::OEFingerprint &fp, unsigned int limit=0,
                 unsigned int bgn=0, unsigned int end=0) const
```

Performs similarity calculations between a molecule or a fingerprint and the fingerprints stored in the `OEFPPDatabase` object. It returns an iterator over the calculated similarity scores (`OESimScore`) in sorted order. Each `OESimScore` holds a similarity score and index of the corresponding fingerprint of the database.

If the method is called with an `OEMolBase` object, then a fingerprint is generated from this molecule before looping over the fingerprints of the database and calculating similarities. If it is called with an `OEFingerprint` object, then its type has to match with the type of the `OEFPPDatabase`.

The `bgn` and `end` arguments define the segment of the database on which the similarity calculation will take place. If both of these parameters are omitted (or set to zero), then the similarity calculation is performed on the entire fingerprint database.

The `limit` argument defines the number of similarity scores returned by the `OEFPPDatabase::GetSortedScores` method. If it is omitted (or set to zero) then all of the similarity scores are returned.

Note: By default, the `OEFPPDatabase::GetSortedScores` method calculates `Tanimoto` similarity scores and returns them in descending order.

The `OEFPPDatabase::SetSimFunc` method allows other similarity measures to be used. The behavior of `OEFPPDatabase::GetSortedScores` is also influenced by the cutoff value (`OEFPPDatabase::SetCutoff`) of the database and the order of the scores specified by the `OEFPPDatabase::SetSimFunc` method.

Examples:

- Calculates the `Tanimoto` similarity on the entire database and returns the 10 best scores in descending order.

```
for (OESimScore * si = fpdb.GetSortedScores(qfp, 10); si; ++si)
  std::cout << si->GetScore() << std::endl;
```

- Calculates the `Dice` similarity on the first 100 entries of the database and returns scores that are equal to or larger than 0.5 in descending order.

```
descending = true;
fpdb.SetSimFunc(OESimMeasure::Dice, descending);
fpdb.SetCutoff(0.5);
for (OESimScore * si = fpdb.GetSortedScores(qfp, 0, 0, 100); si; ++si)
  std::cout << si->GetScore() << std::endl;
```

- Calculates `Manhattan` similarity beginning at the 100th entry of the database and returns the “worst” 5 scores that are equal to or smaller than 0.3 in ascending order.

```
descending = true;
fpdb.SetSimFunc(OESimMeasure::Manhattan, !descending);
fpdb.SetCutoff(0.3);
for (OESimScore * si = fpdb.GetSortedScores(qfp, 5, 100); si; ++si)
  std::cout << si->GetScore() << std::endl;
```

HasCutoff

```
bool HasCutoff() const
```

Returns whether the cutoff value of the `OEFPDatabase` object has been set by the `OEFPDatabase::SetCutoff` method.

See Also:

- `OEFPDatabase::ClearCutoff`
- `OEFPDatabase::GetCutoff`
- `OEFPDatabase::SetCutoff`

NumFingerPrints

```
unsigned int NumFingerPrints() const
```

Returns the number of `OEFPDatabase` objects stored in the database.

SetCutoff

```
void SetCutoff(float)
```

Sets the cutoff value of the `OEFPDatabase` object. The cutoff value influences the behavior of both the `OEFPDatabase::GetScores` and the `OEFPDatabase::GetSortedScores` methods.

See Also:

- `OEFPDatabase::ClearCutoff`
- `OEFPDatabase::HasCutoff`
- `OEFPDatabase::SetCutoff`

SetSimFunc

Sets the method used to evaluate fingerprint similarity when calling either the `OEFPDatabase::GetScores` or the `OEFPDatabase::GetSortedScores` methods.

```
void SetSimFunc(unsigned int simtype, bool descending=true)
```

Sets the similarity calculation by specifying a similarity method with a constant from the `OESimMeasure` namespace. The second argument defines the order in which the calculated scores are returned by the `OEFPDatabase::GetSortedScores` method.

```
void SetSimFunc(const OESimFuncBase &, bool descending=true)
```

Creates a copy of the `OESimFuncBase` object and uses its `OESimFuncBase::operator()` method to evaluate similarity between two `OEFPDatabase` objects. The second argument defines the order in which the calculated scores are returned by the `OEFPDatabase::GetSortedScores` method.

Note: By default, both the `OEFPDatabase::GetScores` and the `OEFPDatabase::GetSortedScores` methods calculate Tanimoto similarity scores. While the `OEFPDatabase::GetScores` always returns these

scores in the order in which the corresponding `OEFingerprint` objects are added to the database, the latter method returns them in descending order, by default.

See Also:

- *Searching with User-defined Similarity Measures* section
- *User-defined Similarity Measures* section
- `OESimMeasure` namespace
- `OETanimotoSim` class
- `OETverskySim` class

4.2.2 OEFTypeBase

`class OEFTypeBase`

The `OEFTypeBase` class is the abstract interface for representing `OEFingerprint` types.

Note: When a `OEFingerprint` object is initialized, its type is set to a type that is derived from this abstract base class.

Constructors

`OEFTypeBase()`

Default constructor.

GetFPType

`unsigned int GetFPType() const =0`

Returns the type of the `OEFTypeBase` object from the `OEFType` namespace.

GetFPTypeString

`std::string GetFPTypeString() const =0`

Returns a string representation of `OEFTypeBase`. This string representation encodes information about the fingerprinting method and the parameters being used to generate the fingerprint.

Note: The string returned by the `OEFTypeBase::GetFPTypeString` method does not include new line characters. The strings presented here were broken into separate lines only for better readability.

The string representation of the default `OEFType::Circular` fingerprint is the following:

```
Circular, ver=2.0.0, size=4096, radius=0-5, atype=AtmNum|Arom|Chiral|FCharge|HCount|EqHalo,
btype=Order
```

The string representation of the default `OEFType::Lingo` fingerprint is the following:

Lingo, ver=2.0.0

The string representation of the default `OEFPType::MACCS166` fingerprint is the following:

```
MACCS166, ver=2.0.0
```

The string representation of the default `OEFPType::Path` fingerprint is the following:

```
Path, ver=2.0.0, size=4096, bonds=0-5, atype=AtmNum|Arom|Chiral|FCharge|HvyDeg|Hyb|EqHalo,
btype=Order|Chiral
```

The string representation of the default `OEFPType::Tree` fingerprint is the following:

```
Tree, ver=2.0.0, size=4096, bonds=0-4, atype=AtmNum|Arom|Chiral|FCharge|HvyDeg|Hyb,
btype=Order
```

See Also:

- `OEFPTypeParams` class

GetFPVersion

```
unsigned short GetFPVersion() const
```

Returns the version number of the fingerprint.

GetFPVersionString

```
std::string GetFPVersionString() const
```

Returns the string format of the version of the fingerprint.

4.2.3 OEFPTypeParams

```
class OEFPTypeParams
```

This class represents `OEFPTypeParams`.

See Also:

- *Fingerprint parameters* section

Constructors

```
OEFPTypeParams(const std::string &typestr)
```

Default constructor that parses the given string as a representation of a fingerprint and extracts the parameters that are used to generate the fingerprint.

See Also:

- `OEFPTypeBase::GetFPTypeString` method

GetAtomTypes

`unsigned int` GetAtomTypes() `const`

Returns

- the atom types extracted from the string for `OEFPType::Path`, `OEFPType::Circular` and `OEFPType::Tree` fingerprint types
- zero for `OEFPType::Lingo` and `OEFPType::MACCS166` fingerprint types

The return value is taken from the `OEFPAtomType` namespace.

See Also:

- `OEGetFPAtomType` function

GetBondTypes

`unsigned int` GetBondTypes() `const`

Returns

- the bond types extracted from the string for `OEFPType::Path`, `OEFPType::Circular` and `OEFPType::Tree` fingerprint types
- zero for `OEFPType::Lingo` and `OEFPType::MACCS166` fingerprint types

The return value is taken from the `OEFPBondType` namespace.

See Also:

- `OEGetFPBondType` function

GetFPType

`unsigned int` GetFPType() `const`

Returns the fingerprint type extracted from the string. The return value is taken from the `OEFPType` namespace.

GetMaxDistance

`unsigned int` GetMaxDistance() `const`

Returns:

- the size of the largest path fragments (in bonds), if the fingerprint type is `OEFPType::Path`
- the size of the largest circular fragments (in radius), if the fingerprint type is `OEFPType::Circular`
- the size of the largest tree fragments (in bonds), if the fingerprint type is `OEFPType::Tree`
- zero for `OEFPType::Lingo` and `OEFPType::MACCS166` fingerprint types

GetMinDistance

```
unsigned int GetMinDistance() const
```

Returns:

- the size of the smallest path fragments (in bonds), if the fingerprint type is `OEFPType::Path`
- the size of the smallest circular fragments (in radius), if the fingerprint type is `OEFPType::Circular`
- the size of the smallest tree fragments (in bonds), if the fingerprint type is `OEFPType::Tree`
- zero for `OEFPType::Lingo` and `OEFPType::MACCS166` fingerprint types

GetNumBits

```
unsigned int GetNumBits() const
```

Returns:

- the size of the fingerprint in bits for `OEFPType::Path`, `OEFPType::Circular` and `OEFPType::Tree` fingerprint types
- 166 for `OEFPType::MACCS166` fingerprint type
- zero for `OEFPType::Lingo` fingerprint type

GetVersion

```
unsigned short GetVersion() const
```

Returns the version number.

See Also:

- `OEGetFingerprintVersionString` function

IsValid

```
bool IsValid() const
```

Returns whether the string was valid from which the `OEFPTypeParams` object was initialized.

See Also:

- `OEIsValidFPTypeString` function

4.2.4 OEFingerprint

```
class OEFingerprint : public OESystem::OEBitVector
```

`OEFingerprint` class is used to encode molecular properties. An `OEFingerprint` object is a *typed* bitvector (`OEBitVector`). The type of an `OEFingerprint` object is set when it is initialized.

See Also:

- `OEfingerprint::GetFPTypeBase`
- `OEfingerprint::SetFPTypeBase`

The following methods are publicly inherited from `OEBitVector`:

<code>operator</code>	<code>FirstBit</code>	<code>PrevBit</code>
<code>operator=</code>	<code>FromHexString</code>	<code>SetBitOff</code>
<code>operator--</code>	<code>GetData</code>	<code>SetBitOn</code>
<code>operator[]</code>	<code>GetSize</code>	<code>SetData</code>
<code>operator^=</code>	<code>IsBitOn</code>	<code>SetRangeOff</code>
<code>operator =</code>	<code>IsEmpty</code>	<code>SetRangeOn</code>
<code>operator &=</code>	<code>LastBit</code>	<code>SetSize</code>
<code>ClearBits</code>	<code>NegateBits</code>	<code>ToHexString</code>
<code>CountBits</code>	<code>NextBit</code>	<code>ToggleBit</code>

Constructors

```
OEfingerprint()
```

Default constructor that creates an `OEfingerprint` object with an uninitialized type, zero pointer (`OEFPTypeBase`).

```
OEfingerprint(const OEfingerprint &rhs)
```

Copy constructor.

operator=

```
OEfingerprint &operator=(const OEfingerprint &rhs)
```

Assignment operator that copies the data of the `rhs OEfingerprint` object into the left-hand side `OEfingerprint` object.

operator==

```
bool operator==(const OEfingerprint& rhs) const
```

Two `OEfingerprint` objects are considered to be equivalent only if they have the same fingerprint type (`OEFPTypeBase`) and have identical bit-vectors (`OEBitVector`).

operator!=

```
bool operator!=(const OEfingerprint& rhs) const
```

Two `OEfingerprint` objects are considered to be different if either they have different fingerprint types (`OEFPTypeBase`) or they have different bit-vectors (`OEBitVector`).

operator bool

```
operator bool() const
```

Returns whether the `OEFPingPrint` has been initialized, *i.e.*, has a valid type.

GetFPTypeBase

```
const OEFPTypeBase *GetFPTypeBase() const
```

Returns a *const* pointer to the fingerprint type (`OEFPTypeBase`) of the `OEFPingPrint` object. This method will return 0 if the `OEFPingPrint` object has not been initialized.

SetFPTypeBase

```
void SetFPTypeBase(const OEFPTypeBase *t)
```

Sets the fingerprint type of a `OEFPingPrint` object.

The following functions set the type of the fingerprint when an `OEFPingPrint` object is initialized:

- `OEMakeFP`
- `OEMakeMACCS166FP`
- `OEMakeLingoFP`
- `OEMakePathFP`
- `OEMakeCircularFP`
- `OEMakeTreeFP`

Warning: Use this method with caution.

The type of a `OEFPingPrint` object (that is an `OEFPTypeBase`) encodes how the fingerprint is generated. Changing this type can mean that the information represented by bitvector of the fingerprint will be misinterpreted.

4.2.5 OESimFuncBase

```
class OESimFuncBase
```

`OESimFuncBase` is an abstract base class which defines the interface necessary to perform similarity calculations of `OEFPingPrint` objects.

See Also:

- *User-defined Similarity Measures* section

The following classes derive from this class:

- `OETanimotoSim`
- `OETverskySim`

Constructors

`OESimFuncBase()`

Default constructor.

operator()

```
float operator() (const OEGraphSim::OEFingerprint &fpA,
                 const OEGraphSim::OEFingerprint &fpB) const =0
```

Virtual *const* operator that takes two `OEFingerprint` objects and returns their evaluated similarity value.

CreateCopy

```
OESimFuncBase *CreateCopy() const =0
```

`OESimFuncBase::CreateCopy` is a virtual constructor which allows copying of concrete derived objects using a reference to this base class.

GetSimTypeString

```
std::string GetSimTypeString() const =0
```

Returns a string representation of the concrete derived class.

4.2.6 OESimScore

```
class OESimScore
```

This class represents `OESimScore` that holds a similarity value with an index that identifies the fingerprint in the `OEFPPDatabase` object from which the score is calculated.

See Also:

- `OEFPPDatabase::GetScores`
- `OEFPPDatabase::GetSortedScores`

Constructors

```
OESimScore(unsigned int i, float s)
```

Initializes an `OESimScore` from an index `i` and a score `s`.

GetIdx

```
unsigned int GetIdx() const
```

Returns the index of the fingerprint corresponding to the similarity score.

GetScore

```
float GetScore() const
```

Returns the similarity score.

4.2.7 OETanimotoSim

```
class OETanimotoSim : public OEGraphSim::OESimFuncBase
```

This class represents `OETanimotoSim` that calculates the `Tanimoto` similarity score.

Formula:

$$Sim_{Tanimoto}(A, B) = \frac{bothAB}{onlyA+onlyB+bothAB}$$

See Also:

- `Tanimoto` section
- `OETanimoto` function

The following methods are publicly inherited from `OESimFuncBase`:

<code>operator()</code>	<code>CreateCopy</code>	<code>GetSimTypeString</code>
-------------------------	-------------------------	-------------------------------

Constructors

```
OETanimotoSim()
```

Default constructor.

operator()

```
float operator() (const OEGraphSim::OEFingerprint &fpA,  
                 const OEGraphSim::OEFingerprint &fpB) const
```

Returns the `Tanimoto` similarity coefficient of the two given `OEFingerprint` objects.

CreateCopy

```
OEGraphSim::OESimFuncBase *CreateCopy() const
```

Deep copy constructor that returns a copy of the object. The memory for the returned `OETanimotoSim` object is dynamically allocated and owned by the caller.

GetSimTypeString

```
std::string GetSimTypeString() const
```

Returns a string representation of the `OETanimotoSim` class.

4.2.8 OETverskySim

```
class OETverskySim : public OEGraphSim::OESimFuncBase
```

This class represents `OETverskySim` that calculates the `Tversky` similarity score.

Formula:

$$Sim_{Tversky}(A, B) = \frac{bothAB}{\alpha * onlyA + \beta * onlyB + bothAB}$$

See Also:

- `Tversky` section
- `OETversky` function

The following methods are publicly inherited from `OESimFuncBase`:

<code>operator()</code>	<code>CreateCopy</code>	<code>GetSimTypeString</code>
-------------------------	-------------------------	-------------------------------

Constructors

```
OETverskySim(float a=0.95f)
```

Default constructor ($\alpha = 0.95$, $\beta = 1.0 - \alpha = 0.05$)

```
OETverskySim(float a, float b)
```

Constructor with arbitrary α and β parameters.

operator()

```
float operator() (const OEGraphSim::OEFingerPrint &fpA,  
                 const OEGraphSim::OEFingerPrint &fpB) const
```

Returns the `Tversky` similarity coefficient of the two given `OEFingerPrint` objects.

CreateCopy

```
OEGraphSim::OESimFuncBase *CreateCopy() const
```

Deep copy constructor that returns a copy of the object. The memory for the returned `OETverskySim` object is dynamically allocated and owned by the caller.

GetAlpha

```
float GetAlpha() const
```

Returns the α parameter of the *Tversky* similarity calculation.

GetBeta

```
float GetBeta() const
```

Returns the β parameter of the *Tversky* similarity calculation.

GetSimTypeString

```
std::string GetSimTypeString() const
```

Returns a string representation of the *OETverskySim* class.

4.3 OEGraphSim Constants

4.3.1 OEFPAtomType

This namespace contains atom typing options that can be used when generating *Circular*, *Path* or *Tree* fingerprints. Atom type options control how the atoms of the enumerated circular, path or tree fragments are encoded during the fingerprint generation.

Note: The constants of the *OEFPAtomType* namespace can be **combined** using the bitwise OR operation.

See Also:

- *OEFPBondType* namespace
- *OEMakeCircularFP* function
- *OEMakePathFP* function
- *OEMakeTreeFP* function
- *Atom and Bond Typing* section

Note: All explicit hydrogens are suppressed of the molecule before generating any fingerprints. (See example in *Example of molecules that are considered to be equivalent due to suppressing their explicit hydrogens*).

AtomicNumber

This flag indicates that atomic number information is encoded into the generated fingerprint, *i.e.*, if two fragments (either circular, paths or tree) are composed of atoms with different atomic numbers (the value returned by the *OEAtomBase::GetAtomicNum* method), then the two fragments will be mapped to different bits of the fingerprint.

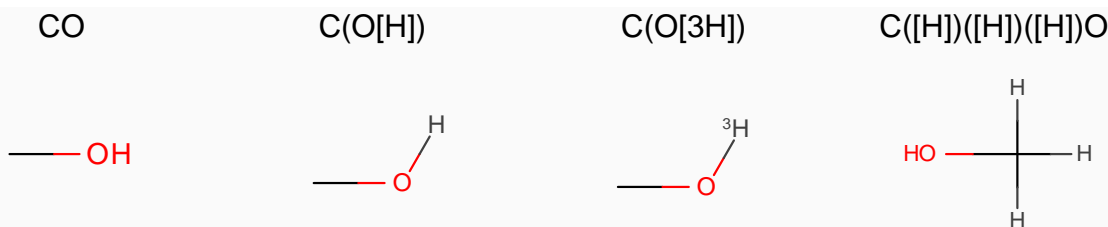


Figure 4.1: Example of molecules that are considered to be equivalent due to suppressing their explicit hydrogens

Aromaticity

This flag indicates that atom aromaticity information (returned by the `OEAtomBase::IsAromatic` method) is encoded into the generated fingerprint, *i.e.*, an aromatic and an aliphatic fragment will be mapped to different bits of the fingerprint.

Chiral

This flag indicates that chiral and non-chiral atoms (value returned by the `OEAtomBase::IsChiral` method) are distinguished during the fingerprint generation.

Note: Different stereoisomers of molecules can not be distinguished when the `OEFPAtomType::Chiral` flag is set.

FormalCharge

This flag indicates that formal charge information (the value returned by the `OEAtomBase::GetFormalCharge` method) is encoded into the generated fingerprint.

HvyDegree

This flag indicates that heavy degree information (the value returned by the `OEAtomBase::GetHvyDegree` method) is encoded into the generated fingerprint.

Hybridization

This flag indicates that formal charge information (the value returned by the `OEAtomBase::GetHyb` method) is encoded into the generated fingerprint.

InRing

This flag indicates that atom topology information is encoded into the generated fingerprint, *i.e.*, if two fragments (either circular, path or tree) are composed of atoms with different atom topology (the value returned by the `OEAtomBase::IsInRing` method) then the two fragments will be mapped to different bits of the fingerprint.

HCount

This flag indicates that number of hydrogens (the value returned by the `OEAtomBase::GetTotalHCount` method) is encoded into the generated fingerprint.

EqAromatic

This flag modifies the meaning of the `OEFPAtomType::AtomicNumber` flag. If the `OEFPAtomType::EqAromatic` flag is set then aromatic atoms are considered equivalent during the fingerprint generation. *Figure: *Examples for the EqAromatic option* with the related table demonstrate the effect of using the `OEFPAtomType::EqAromatic` flag.

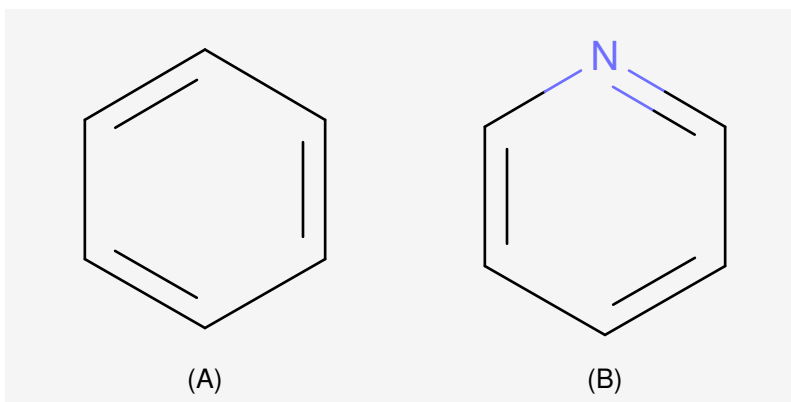


Figure 4.2: Examples for the EqAromatic option

Table 4.1: Example of using the `OEFPAtomType::EqAromatic` option (Circular, numbits=4096, minrad=0, maxrad=3).

atom typing	bond typing	OETanimoto(A,B)
<code>AtomicNumber</code>	<code>BondOrder</code>	0.214
<code>AtomicNumber EqAromatic</code>	<code>BondOrder</code>	1.000

EqHalogen

This flag modifies the meaning of `OEFPAtomType::AtomicNumber` flag. If the `OEFPAtomType::EqHalogen` flag is set then halide atoms (`OEElemNo::F`, `OEElemNo::Cl`, `OEElemNo::Br`, and `OEElemNo::I`) are considered equivalent during the fingerprint generation.

Figure: Examples for the EqHalogen option with the related table demonstrate the effect of using the `OEFPAtomType::EqHalogen` flag.

Table 4.2: Example of using the `OEFPAtomType::EqHalogen` option (Circular, numbits=4096, minrad=0, maxrad=3).

atom typing	bond typing	OETanimoto(A,B)
<code>AtomicNumber</code>	<code>BondOrder</code>	0.095
<code>AtomicNumber EqHalogen</code>	<code>BondOrder</code>	1.000

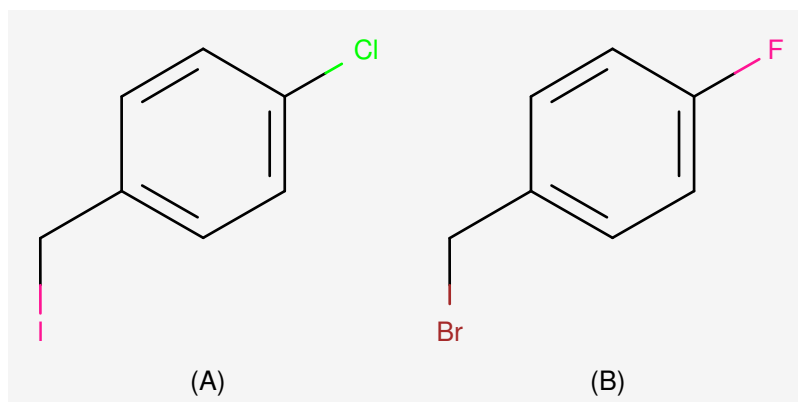


Figure 4.3: Examples for the EqHalogen option

EqHBondAcceptor

This flag modifies the meaning of the `OEFPAtomType::AtomicNumber` flag. If the `OEFPAtomType::EqHBondAcceptor` flag is set then atoms that are perceived as hydrogen bonding acceptors are considered equivalent during the fingerprint generation.

EqHBondDonor

This flag modifies the meaning of the `OEFPAtomType::AtomicNumber` flag. If the `OEFPAtomType::EqHBondDonor` flag is set then atoms that are perceived as hydrogen bonding donors are considered equivalent during the fingerprint generation.

DefaultAtom

The combination of the following *primitive* atom typing flags:

- `OEFPAtomType::AtomicNumber`
- `OEFPAtomType::Aromaticity`
- `OEFPAtomType::Chiral`
- `OEFPAtomType::FormalCharge`
- `OEFPAtomType::HvyDegree`
- `OEFPAtomType::Hybridization`
- `OEFPAtomType::EqHalogen`

DefaultCircularAtom

The bitwise OR'd value of the following atom typing options:

- `OEFPAtomType::AtomicNumber`
- `OEFPAtomType::Aromaticity`
- `OEFPAtomType::Chiral`

- `OEFPAtomType::FormalCharge`
- `OEFPAtomType::HCount`
- `OEFPAtomType::EqHalogen`

This constant is used as atom typing parameter when a default `Circular` fingerprint is generated by the following functions:

- `OEMakeFP (OEFingerPrint &, const OEMolBase &, OEFPType::Circular)`
- `OEMakeCircularFP`

DefaultPathAtom

The bitwise OR'd value of the following atom typing options:

- `OEFPAtomType::AtomicNumber`
- `OEFPAtomType::Aromaticity`
- `OEFPAtomType::Chiral`
- `OEFPAtomType::FormalCharge`
- `OEFPAtomType::HvyDegree`
- `OEFPAtomType::Hybridization`
- `OEFPAtomType::EqHalogen`

This constant is used as atom typing parameter when a default `Path` fingerprint is generated by the following functions:

- `OEMakeFP (OEFingerPrint &, const OEMolBase &, OEFPType::Path)`
- `OEMakePathFP`

DefaultTreeAtom

The bitwise OR'd value of the following atom typing options:

- `OEFPAtomType::AtomicNumber`
- `OEFPAtomType::Aromaticity`
- `OEFPAtomType::Chiral`
- `OEFPAtomType::FormalCharge`
- `OEFPAtomType::HvyDegree`
- `OEFPAtomType::Hybridization`

This constant is used as atom typing parameter when a default `Tree` fingerprint is generated by the following functions:

- `OEMakeFP (OEFingerPrint &, const OEMolBase &, OEFPType::Tree)`
- `OEMakeTreeFP`

4.3.2 OEFPBondType

This namespace contains bond typing options that can be used when generating *Circular*, *Path* or *Tree* fingerprints. Bond type options control how the bonds of the enumerated circular, path or tree fragments are encoded during the fingerprint generation.

Note: The constants of the `OEFPBondType` namespace can be **combined** using the bitwise OR operation.

See Also:

- `OEFPAtomType` namespace
- `OEMakeCircularFP` function
- `OEMakePathFP` function
- `OEMakeTreeFP` function
- *Atom and Bond Typing* section

BondOrder

This flag indicates that bond order information is encoded into the generated fingerprint, *i.e.*, if two fragments (either circular, path or tree) are composed of bonds with different bond orders (the value returned by the `OEBondBase::GetOrder` method) then the two fragments will be mapped to different bits of the fingerprint.

Chiral

This flag indicates that chiral and non-chiral bonds (the value returned by the `OEBondBase::IsChiral` method) are distinguished during the circular, path, the tree fingerprint generation. For example, the path fingerprint of molecules CC=CC and C/C=C/C will only be different if the `OEFPBondType::Chiral` flag is set.

Note: *Cis* and *trans* chiral bonds can not be distinguished when the `OEFPBondType::Chiral` flag is set. Fingerprints generated for molecules C\C=C/C and C/C=C/C will always be identical.

InRing

This flag indicates that bond topology information is encoded into the generated fingerprint, *i.e.*, if two fragments (either circular, path or tree) are composed of bonds with different bond topology (the value returned by the `OEBondBase::IsInRing` method) then the two fragments will be mapped to different bits of the fingerprint.

DefaultBond

The bitwise OR'd value of the following bond typing options:

- `OEFPBondType::BondOrder`
- `OEFPBondType::Chiral`

DefaultCircularBond

Combination of `OEFPBondType::BondOrder` and `OEFPBondType::Chiral` bond typing flags.

This constant is used as bond typing parameter when a default `Circular` fingerprint is generated by the following functions:

- `OEMakeFP (OEFingerPrint &, const OEMolBase &, OEFPType::Circular)`
- `OEMakeCircularFP`

DefaultPathBond

The bitwise OR'd value of the following bond typing options:

- `OEFPBondType::BondOrder`
- `OEFPBondType::Chiral`

This constant is used as bond typing parameter when a default `Path` fingerprint is generated by the following functions:

- `OEMakeFP (OEFingerPrint &, const OEMolBase &, OEFPType::Path)`
- `OEMakePathFP`

DefaultTreeBond

Same as the `OEFPBondType::BondOrder` constant.

This constant is used as bond typing parameter when a default `Tree` fingerprint is generated by the following functions:

- `OEMakeFP (OEFingerPrint &, const OEMolBase &, OEFPType::Tree)`
- `OEMakeTreeFP`

4.3.3 OEFPType

This namespace contains constants representing various `OEFingerPrint` types available in the *GraphSimTK*.

See Also:

- `OEMakeFP`

Circular

This constant represents the circular fingerprint type.

Circular fingerprints are generated by **exhaustively** enumerating **all** circular fragments grown radially from each heavy atom of the molecule up to the given radius and then hashing these fragments into a fixed-length bitvector.

See Also:

- *Circular* section

Lingo

This constant represents the LINGO fingerprint type.

See Also:

- *LINGO* section
- *LINGO* definition in the *Glossary* chapter

MACCS166

This constant represents the MACCS key fingerprint type.

See Also:

- *MACCS* section

Path

This constant represents the path fingerprint type.

Path fingerprints are generated by **exhaustively** enumerating **all** linear paths of a molecular graph up to a given size and then hashing these fragments into a fixed-length bitvector.

See Also:

- *Path* section

Tree

This constant represents the tree fingerprint type.

A tree fingerprint is generated by **exhaustively** enumerating **all** trees of a molecular graph up to a given size and then hashing these fragments into a fixed-length bitvector.

See Also:

- *Tree* section

4.3.4 OESimMeasure

This namespace contains constants representing the built-in similarity indices of the *GraphSimTK*.

See Also:

- `OEFPDatabase::SetSimFunc` method
- *Searching with User-defined Similarity Measures* section

Cosine

This constant represents Cosine similarity index.

See Also:

- *Cosine* section
- `OECosine` function

Dice

This constant represents Dice similarity index.

See Also:

- *Dice* section
- `OEDice` function

Euclid

This constant represents the Euclidean similarity index.

See Also:

- *Euclidean* section
- `OEEuclid` function

Manhattan

This constant represents the Manhattan similarity index.

See Also:

- *Manhattan* section
- `OEManhattan` function

Tanimoto

This constant represents the Tanimoto similarity index.

See Also:

- *Tanimoto* section
- `OETanimoto` function
- `OETanimotoSim` class

Tversky

This constant represents the Tversky similarity index.

See Also:

- *Tversky* section
- `OETversky` function
- `OETverskySim` class

4.4 OEGraphSim Functions

4.4.1 OECosine

float OECosine(**const** OEFingerprint &fpA, **const** OEFingerprint &fpB)

Calculates the Cosine similarity value of two OEFingerprint objects.

Formula: $Sim_{Cosine}(A, B) = \frac{bothAB}{\sqrt{(onlyA+bothAB)*(onlyB+bothAB)}}$

See Also:

- *Cosine* section

4.4.2 OEDice

float OEDice(**const** OEFingerprint &fpA, **const** OEFingerprint &fpB)

Calculates the Dice similarity value of two OEFingerprint objects.

Formula: $Sim_{Dice}(A, B) = \frac{2*bothAB}{onlyA+onlyB+2*bothAB}$

See Also:

- *Dice* section

4.4.3 OEEuclid

float OEEuclid(**const** OEFingerprint &fpA, **const** OEFingerprint &fpB)

Calculates the Euclidean similarity value of two OEFingerprint objects.

Formula: $Sim_{Euclid}(A, B) = \sqrt{\frac{bothAB+neitherAB}{onlyA+onlyB+bothAB+neitherAB}}$

See Also:

- *Euclidean* section

4.4.4 OEGetBitCounts

bool OEGetBitCounts(**const** OEFingerprint &fpA, **const** OEFingerprint &fpB,
unsigned int *onlyA, **unsigned int** *onlyB,
unsigned int *bothAB, **unsigned int** *neitherAB)

fpA, fpB The two OEFingerprint objects being compared.

onlyA The number of bits set “on” in fingerprint *fpA* but not in *fpB*.

onlyB The number of bits set “on” in fingerprint *fpB* but not in *fpA*.

bothAB The number of bits set “on” in both OEFingerprint objects.

neitherAB The number of bits unset in both OEFingerprint objects.

See Also:

- *User-defined Similarity Measures* section

4.4.5 OEGetFPAtomType

```
std::string OEGetFPAtomType(unsigned int value)
```

Returns the string representation of a value that is from the `OEGFPAtomType` namespace. For example, the string representation of `OEGFPAtomType::AtomicNumber` | `OEGFPAtomType::Aromaticity` value is the "AtmNum|Arom" string.

Table 4.3: The string representation of various fingerprint atom types

Atom type	String representation
<code>OEGFPAtomType::AtomicNumber</code>	"AtmNum"
<code>OEGFPAtomType::Aromaticity</code>	"Arom"
<code>OEGFPAtomType::Chiral</code>	"Chiral"
<code>OEGFPAtomType::FormalCharge</code>	"FCharge"
<code>OEGFPAtomType::HvyDegree</code>	"HvyDeg"
<code>OEGFPAtomType::Hybridization</code>	"Hyb"
<code>OEGFPAtomType::InRing</code>	"InRing"
<code>OEGFPAtomType::HCount</code>	"HCount"
<code>OEGFPAtomType::EqAromatic</code>	"EqArom"
<code>OEGFPAtomType::EqHalogen</code>	"EqHalo"
<code>OEGFPAtomType::EqHBondAcceptor</code>	"EqHBAcc"
<code>OEGFPAtomType::EqHBondDonor</code>	"EqHBDon"

```
unsigned int OEGetFPAtomType(const std::string &expression)
```

This function is the inverse of the previous `OEGetFPAtomType` function *i.e.* it converts the string representation into a value from the `OEGFPAtomType` namespace.

4.4.6 OEGetFPBondType

```
std::string OEGetFPBondType(unsigned int value)
```

Returns the string representation of a value that is from the `OEGFPBondType` namespace. For example, the string representation of `OEGFPBondType::BondOrder` | `OEGFPBondType::Chiral` value is the: "Order|Chiral" string.

Table 4.4: The string representation of various fingerprint bond types

Bond type	String representation
<code>OEGFPBondType::BondOrder</code>	"Order"
<code>OEGFPBondType::Chiral</code>	"Chiral"
<code>OEGFPBondType::InRing</code>	"InRing"

```
unsigned int OEGetFPBondType(const std::string &expression)
```

This function is the inverse of the previous `OEGetFPBondType` function *i.e.* it converts the string representation into a value from the `OEGFPBondType` namespace.

4.4.7 OEGetFPCoverage

```
OESystem::OEIterBase<OEChem::OEAtomBondSet> *
  OEGetFPCoverage(const OEChem::OEMolBase &mol, const OEFPTTypeBase *fptype,
                 bool unique=false)
```

Returns an iterator over the fragments that are generated when a fingerprint is constructed.

mol The OEMolBase objects for from which the fingerprint overlap is generated.

fptype The type of the fingerprint that also stores the parameters that are used when the fingerprint is generated.

unique If true, then the OEGetFPCoverage function returns only unique fragments. A fragment (i.e. a subgraph) is considered unique if it differs from all other subgraphs identified previously by at least one atom or bond.

Note: The OEGetFPCoverage function is not available for the OEFPTType::Lingo fingerprint type.

See Also:

- [Fingerprint Coverage](#) chapter

4.4.8 OEGetFPOverlap

```
OESystem::OEIterBase<OEChem::OEMatchBase> *
  OEGetFPOverlap(const OEChem::OEMolBase &qmol, const OEChem::OEMolBase &tmol,
                 const OEFPTTypeBase *fptype)
```

Returns an iterator over all **unique** matches (common fragments) found between two molecules based on the given fingerprint type. This means that these common molecular fragments are mapped into the same bit when fingerprints are generated with the given type.

qmol, tmol The two OEMolBase objects from which the fingerprint overlap is generated.

fptype The type of the fingerprint that also stores the parameters that are used when the fingerprint is generated.

Note: The OEGetFPOverlap function is not available for the OEFPTType::Lingo fingerprint type.

Note: Two matches are considered to be equivalent only if they store the same sets of target and pattern atoms and bonds. The correspondence between the pattern and target atoms and bonds are not considered.

See Also:

- [Fingerprint Overlap](#) chapter

4.4.9 OEGetFPTType

```
const OEFPTTypeBase *OEGetFPTType(unsigned int fptype)
```

Returns a *const* pointer of a default fingerprint type.

fptype The type of the fingerprint. This value has to be from the OEFPTType namespace.

```
const OEFPTTypeBase *OEGetFPTType(const std::string &fptypestring)
```

Returns a *const* pointer of a fingerprint type of the given valid string representation. If the string representation is not valid, it will return a NULL pointer.

See Also:

- `OEIsValidFPTypeString` function
- `OEFPTypeBase::GetFPTypeString` method
- `OEFPTypeParams` class

4.4.10 OEGetFingerPrintVersion

```
unsigned short OEGetFingerPrintVersion(unsigned int fptype)
```

See Also:

- `OEFPTypeBase::GetFPVersion` method
- `OEFPTypeBase::GetFPVersionString` method

4.4.11 OEGetFingerPrintVersionString

```
std::string OEGetFingerPrintVersionString(unsigned short version)
```

Returns the string representation of a fingerprint version number.

4.4.12 OEGraphSimGetArch

```
const char *OEGraphSimGetArch()
```

4.4.13 OEGraphSimGetLicensee

```
bool OEGraphSimGetLicensee(std::string &licensee)
```

4.4.14 OEGraphSimGetPlatform

```
const char *OEGraphSimGetPlatform()
```

Returns the internal build string used by OpenEye Scientific Software to identify a platform. The format of these strings may change over time, and future distributions may contain different values even when using the same operating system, compiler and processor. For example, on an `x86_64` Red Hat Enterprise Linux box this would return `redhat-RHEL5-g++4.1-x64`.

4.4.15 OEGraphSimGetRelease

```
const char *OEGraphSimGetRelease()
```

Returns the release name of the *GraphSimTK* library being used. This returns a value similar to `1.0` for production versions of the library, and `1.0 debug` for the checking version of the library.

4.4.16 OEGraphSimGetSite

```
bool OEGraphSimGetSite(std::string &site)
```

4.4.17 OEGraphSimGetVersion

```
unsigned int OEGraphSimGetVersion()
```

4.4.18 OEGraphSimIsLicensed

```
bool OEGraphSimIsLicensed(const char *feature=0, unsigned int *expdate=0)
```

4.4.19 OEIsFPType

```
bool OEIsFPType(const OEFingerprint &fp, unsigned int fptype)
```

Returns true if the given `OEFingerprint` object's fingerprint type is identical to `fptype`. `fptype` has to be a value from the `OEFPType` namespace.

See Also:

- `OEIsSameFPType` function

4.4.20 OEIsSameFPType

```
bool OEIsSameFPType(const OEFingerprint &fpA, const OEFingerprint &fpB)
```

Returns whether or not the types (`OEFPTypeBase`) of the two `OEFingerprint` objects are the same.

Note: Two user-defined path fingerprints have the same type only if they have been generated with the same parameters (*i.e.* number of bits, minimum and maximum path size and atom and bond typing options) and have the same version number.

Warning: Similarity measure functions (such as `OETanimoto`) only work on fingerprints with identical type (`OEFPTypeBase`).

4.4.21 OEIsValidFPTypeString

```
bool OEIsValidFPTypeString(std::string ftypestring);
```

Returns true if the given string is a valid representation of any available fingerprint types.

See Also:

- `OEFPTypeBase::GetFPTypeString` method

4.4.22 OEMakeCircularFP

```
bool OEMakeCircularFP(OEFingerPrint &fp, const OEChem::OEMolBase &mol)
```

Generates the `OEFingerPrint` object as a *Circular* fingerprint using default parameters.

fp The `OEFingerPrint` object that is being initialized.

mol The `OEMolBase` object from which the fingerprint is generated.

```
bool OEMakeCircularFP(OEFingerPrint &fp, const OEChem::OEMolBase &mol,
    unsigned int numbits,
    unsigned int minradius, unsigned int maxradius,
    unsigned int atype, unsigned int btype)
```

Generates the `OEFingerPrint` object as a *Circular* fingerprint using the given parameters.

fp The `OEFingerPrint` object that is being initialized.

mol The `OEMolBase` object from which the fingerprint is generated.

numbits The size of the fingerprint in bits. This number has to be larger than or equal to 2^4 and smaller than 2^{16} .

minradius, maxradius The smallest and largest circular fragments (in radius) that are enumerated during the fingerprint generation. All enumerated circular fragments are hashed into the `OEFingerPrint` object.

atype Defines which atom properties are encoded during the fingerprint generation. This value has to be either a value or a set of bitwise OR'd values from the `OEFPAtomType` namespace.

btype Defines which bond properties are encoded during the fingerprint generation. This value has to be either a value or a set of bitwise OR'd values from the `OEFPBondType` namespace.

Note: For an empty molecule, both `OEMakeCircularFP` function will return `false` and the type of the fingerprint will be set to 0 (`OEFingerPrint::GetFPTypeBase`).

See Also:

- *User-defined Fingerprint* chapter
- `OEFPAtomType` namespace
- `OEFPBondType` namespace

4.4.23 OEMakeFP

```
bool OEMakeFP(OEFingerPrint &fp, const OEChem::OEMolBase &mol,
    unsigned int fptype)
```

Generates a fingerprint from the given molecule.

fp The `OEFingerPrint` object that is being initialized.

mol The `OEMolBase` object from which the fingerprint is generated.

fptype The type of the fingerprint. This value has to be from the `OEFPType` namespace.

```
bool OEMakeFP(OEFingerPrint &fp, const OEChem::OEMolBase &mol,
    const OEFPTypeBase *fptype)
```

Generates a fingerprint from the given molecule.

fp The `OEFPTypeBase` object that is being initialized.

mol The `OEChem::OEMolBase` object from which the fingerprint is generated.

fpType The type of the fingerprint specified by an `OEFPTypeBase` pointer.

Note: For an empty molecule, both `OEMakeFP` function will return `false` and the type of the fingerprint will be set to 0 (`OEFPTypeBase::GetFPTypeBase`).

4.4.24 OEMakeLingoFP

```
bool OEMakeLingoFP(OEFPTypeBase &fp, const OEChem::OEMolBase &mol)
```

Generates the `OEFPTypeBase` object as a *LINGO* fingerprint.

fp The `OEFPTypeBase` object that is being initialized.

mol The `OEChem::OEMolBase` object from which the fingerprint is generated.

Note: For an empty molecule, the `OEMakeLingoFP` function will return `false` and the type of the fingerprint will be set to 0 (`OEFPTypeBase::GetFPTypeBase`).

See Also:

- *LINGO* section

4.4.25 OEMakeMACCS166FP

```
bool OEMakeMACCS166FP(OEFPTypeBase &fp, const OEChem::OEMolBase &mol)
```

Generates the `OEFPTypeBase` object as a *MACCS* fingerprint.

fp The `OEFPTypeBase` object that is being initialized.

mol The `OEChem::OEMolBase` object from which the fingerprint is generated.

Note: For an empty molecule, the `OEMakeMACCS166FP` function will return `false` and the type of the fingerprint will be set to 0 (`OEFPTypeBase::GetFPTypeBase`).

See Also:

- *MACCS* section

4.4.26 OEMakePathFP

```
bool OEMakePathFP(OEFPTypeBase &fp, const OEChem::OEMolBase &mol,
```

Generates the `OEFPTypeBase` object as a *Path* fingerprint using default parameters.

fp The `OEFPTypeBase` object that is being initialized.

mol The `OEChem::OEMolBase` object from which the fingerprint is generated.

```
    unsigned int numbits,
    unsigned int minbonds, unsigned int maxbonds,
    unsigned int atype, unsigned int btype)
```

Generates the `OEFP` object as a *Path* fingerprint using the given parameters.

fp The `OEFP` object that is being initialized.

mol The `OEChem::OEMolBase` object from which the fingerprint is generated.

numbits The size of the fingerprint in bits. This number has to be larger than or equal to 2^4 and smaller than 2^{16} .

minbonds, maxbonds The smallest and largest paths (in bonds) that are enumerated during the fingerprint generation. All enumerated paths are hashed into the `OEFP` object.

atype Defines which atom properties are encoded during the fingerprint generation. This value has to be either a value or a set of bitwise OR'd values from the `OEFPAtomType` namespace.

btype Defines which bond properties are encoded during the fingerprint generation. This value has to be either a value or a set of bitwise OR'd values from the `OEFPBondType` namespace.

Note: For an empty molecule, both `OEMakePathFP` function will return `false` and the type of the fingerprint will be set to 0 (`OEFP::GetFPTypeBase`).

See Also:

- *User-defined Fingerprint* chapter
- `OEFPAtomType` namespace
- `OEFPBondType` namespace

4.4.27 OEMakeTreeFP

```
bool OEMakeTreeFP(OEFP &fp, const OEChem::OEMolBase &mol)
```

Generates the `OEFP` object as a *Tree* fingerprint using default parameters.

fp The `OEFP` object that is being initialized.

mol The `OEChem::OEMolBase` object from which the fingerprint is generated.

```
bool OEMakeTreeFP(OEFP &fp, const OEChem::OEMolBase &mol,
                 unsigned int numbits,
                 unsigned int minbond, unsigned int maxbonds,
                 unsigned int atype, unsigned int btype)
```

fp The `OEFP` object that is being initialized.

mol The `OEChem::OEMolBase` object from which the fingerprint is generated.

numbits The size of the fingerprint in bits. This number has to be larger than or equal to 2^4 and smaller than 2^{16} .

minbonds, maxbonds The smallest and largest tree fragments (in bonds) that are enumerated during the fingerprint generation. All enumerated tree fragments are hashed into the `OEFP` object.

atype Defines which atom properties are encoded during the fingerprint generation. This value has to be either a value or a set of bitwise OR'd values from the `OEFPAtomType` namespace.

btype Defines which bond properties are encoded during the fingerprint generation. This value has to be either a value or a set of bitwise OR'd values from the `OEFPBondType` namespace.

Note: For an empty molecule, both `OEMakeTreeFP` function will return `false` and the type of the fingerprint will be set to 0 (`OEFP::GetFPTypeBase`).

See Also:

- *User-defined Fingerprint* chapter
- `OEFPPAtomType` namespace
- `OEFPPBondType` namespace

4.4.28 OEmanhattan

`float` `OEmanhattan(const OEFingerprint &fpA, const OEFingerprint &fpB)`

Calculates the Manhattan similarity value of two `OEFingerprint` objects.

Formula: $Sim_{Manhattan}(A, B) = \frac{onlyA+onlyB}{onlyA+onlyB+bothAB+neitherAB}$

See Also:

- *Manhattan* section

4.4.29 OETanimoto

`float` `OETanimoto(const OEFingerprint &fpA, const OEFingerprint &fpB)`

Calculates the `Tanimoto` similarity value of two `OEFingerprint` objects.

Formula: $Sim_{Tanimoto}(A, B) = \frac{bothAB}{onlyA+onlyB+bothAB}$

See Also:

- *Tanimoto* section

4.4.30 OETversky

`float` `OETversky(const OEFingerprint &fpA, const OEFingerprint &fpB, float a, float b)`

Calculates the `Tversky` similarity value of two `OEFingerprint` objects.

Formula: $Sim_{Tversky}(A, B) = \frac{bothAB}{\alpha*onlyA+\beta*onlyB+bothAB}$

See Also:

- *Tversky* section

GLOSSARY AND BIBLIOGRAPHY

5.1 Glossary

canonical SMILES In *OEChem*, the term *canonical SMILES* is used for a unique SMILES string that encodes the connection table of a molecule, but no chiral or isotopic information. Consequently, two stereoisomers always share the same canonical SMILES, since their stereo information is ignored during the canonicalization process. For generating a canonical SMILES, use the `OECreatCanSmIString` function.

Note: *OEChem's canonical SMILES* terminology corresponds to *Daylight's 'unique'* SMILES definition.

canonical isomeric SMILES In *OEChem*, the name *canonical isomeric SMILES* is used for a unique SMILES string that also encodes isotopic and stereo information. Due to the unambiguity of canonical isomeric SMILES, they can be used as a universal identifier for a specific chemical structure. For generating a canonical isomeric SMILES, use the `OECreatIsoSmIString` function.

Note: *OEChem's canonical isomeric SMILES* terminology corresponds to *Daylight's 'absolute'* SMILES definition.

LINGO LINGO is a very fast text-based molecular similarity search method. It is based on fragmentation of *canonical isomeric SMILES* strings into overlapping substrings.

See Also:

- [Grant-2006]
- [Vidal-2005]
- [Vidal-2006]

SMARTS SMARTS is a language that allows specifying substructures by providing a number of primitive symbols describing atomic and bond properties. Atom and bond primitive specifications may be combined to form expressions by using logical operators. For more information go to <http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html>

SMILES A SMILES string represents a molecule by describing only its molecular graph (*i.e.* atoms and bonds in the connection table, but no chiral or isotopic information). There are usually a large number of valid SMILES which represent a given structure. For example, `CCO`, `OCC` and `C(O)C` all specify the structure of ethanol. For generating an arbitrary SMILES string, use the `OECreatAbsSmIString` function. For more information go to <http://www.daylight.com/smiles/>.

structural key A structural key is a fixed-length bitstring in which each bit is associated with a specific molecular pattern. When a structural key is generated for a molecule, the bitstring encodes whether or not these specific molecular patterns are present or absent in the molecule. The performance of such keys depends on the choice of the fragments used for constructing the keys and the probability of their presence in the searched molecule databases.

fingerprint Fingerprints do not use a predefined pattern dictionary, the encoded fragments are enumerated exhaustively. Since the number of possible patterns present in molecular structures is extremely large, it is impractical to assign a particular bit to each unique pattern, as in the case of *structural key* method. Instead, each pattern is subjected to a hashing function that logically *OR* into the fingerprint. The usage of hashing inherently results in overlap of some structural patterns.

5.2 Bibliography

RELEASE NOTES

6.1 GraphSimTK 2.0.1

6.1.1 Minor bug fixes

- If a molecule being fingerprinted has only hydrogen atom(s), for example molecule [H], then an empty fingerprint is generated, that is, all bits of the fingerprint are set to zero. All hydrogens of the molecule (polar, stereo, isotope, charge) are suppressed before generating its fingerprint.
- If the shell of the circular fingerprint can not be extended, for example, all atoms of the molecule are already being considered, then the search is terminated. This modification does not effect the generated circular fingerprints, it only makes the generation process faster.
- Some of the algorithms generating fingerprints can return the same fragment of a molecule more than once. For example the path fingerprint enumerates both OC and CO. These duplicates are now filtered out when calling the `OEGetFPOverlap` function. Two overlaps are considered equivalent only if they store the same target and pattern atoms and bonds, not considering the correspondence between the pattern and target atoms and bonds.

6.2 GraphSimTK 2.0.0

6.2.1 New features

- Adding two new fingerprint types:
 - The `OEFPTypE::Circular` fingerprint is generated by exhaustively enumerating all circular fragments grown radially from each heavy atom of the input molecule up to the given radius. (See more details in *Circular Fingerprint* section and `OEMakeCircularFP` function)
 - The `OEFPTypE::Tree` fingerprint is generated by exhaustively enumerating all unique trees of a molecular graph. (See more details in *Tree Fingerprint* section and `OEMakeTreeFP` function)
- Adding the `OEGetFPCoverage` function that allows access to the patterns that are encoded into a fingerprint. (See example in *Fingerprint Coverage* chapter)
- Adding the `OEGetFPOverlap` function that returns the common patterns of two molecules based on their fingerprint. (See example in *Fingerprint Overlap* chapter)
- Adding version numbers for fingerprints that prevents the comparison of fingerprints that are generated by different versions of the algorithm. (See more details in *Fingerprint version number* section and `OEGetFingerprintVersion` and `OEGetFingerprintVersionString` functions)
- Adding three more atom typing options that can be used when generating path, circular or tree fingerprints:

- `OEFPAtomType::HCount`
- `OEFPAtomType::EqHBondAcceptor`
- `OEFPAtomType::EqHBondDonor`
- Adding the `OEGetFPType` function that generates a fingerprint type from a valid string representation.
- Adding the `OEFPTypeParams` that allow to extract the parameters of the fingerprint types from their string representation:
- Adding 9 code examples in four different languages (C++, Python, Java, C#) that demonstrate the usage of fingerprints and functionalities.

6.2.2 Documentation changes

- Adding figures to the *Fingerprint* chapter that demonstrate how the `OEFPType::Circular`, `OEFPType::Path` and `OEFPType::Tree` fingerprints are generated.
- Adding a new chapter *Fingerprint Types*, that gives more details about fingerprint types.
- Adding two new examples to the *Storage and Retrieval* chapter that demonstrate how to store and retrieve fingerprints in SDF files.
- Rewriting part of the *User-defined Fingerprint* chapter, since the two new fingerprint types can also be customized.
- Adding two new chapters:
 - *Fingerprint Coverage* chapter
 - *Fingerprint Overlap* chapter

6.3 GraphSimTK 1.0.1

6.3.1 Documentation changes

- Adding an example to the *Storage and Retrieval* section that demonstrate how to generate a bitstring from an `OEFPFingerprint` object.

6.4 GraphSimTK 1.0.0

- Provides the following features:
 - Generate three basic fingerprint types (MACCS key, LINGO and path-based)
 - Parameterize the path-based fingerprint generation
 - * allowing various atom/bond typing
 - * specifying the minimum and maximum length of the enumerated paths
 - * setting the size of fingerprint
 - Store and retrieve fingerprints as generic data
 - Save/import fingerprints to/from OEB binary file format
- Implementation of six common similarity indices (Cosine, Dice, Euclidean, Manhattan, Tanimoto, Tversky)

- Implementation of fingerprint database that provides the following methods for rapid fingerprint-based similarity search:
 - apply user-defined similarity measures
 - set similarity cutoff value
 - access similarity in sorted order

INDICES

- *Index*
- *Search Page*

BIBLIOGRAPHY

- [Briem-Lessel-2000] H. Briem and U. F. Lessel, **In Vitro and in Silico Affinity Fingerprints: Finding Similarities Beyond Structural Classes**, *Perspectives in Drug Discovery and Design*, Vol. 20, pp. 231–244, **2000**, (online: <http://www.springerlink.com/content/m7j34558107q3748/?p=bb33f4fc2ac64df8b47314e96619a86d&pi=13>)
- [Grant-2006] J. A. Grant, J. A. Haigh, B. T. Pickup, A. Nicholls and R. A. Sayle, **Lingos, Finite State Machines and Fast Similarity Searching**, *Journal of Chemical Information and Modeling*, Vol. 45, pp. 1912–1918, **2006**, (online: <http://pubs.acs.org/doi/abs/10.1021/ci6002152>)
- [Hert-Willett-2004] J. Hert, P. Willett and D. J. Wilton **Comparison of Fingerprint-Based Methods for Virtual Screening Using Multiple Bioactive Reference Structures**, *Journal of Chemical Information and Computer Science*, Vol. 44, pp. 1177–1185 **2004**, (online: <http://pubs.acs.org/doi/abs/10.1021/ci034231b>)
- [Holliday-2002] J.D. Holliday, C-Y Hu. and P. Willett, **Grouping of Coefficients for the Calculation of Inter-Molecular Similarity and Dissimilarity using 2D Fragment Bit-Strings**, *Combinatorial Chemistry & High Throughput Screening*, Vol. 5, pp. 155–166, **2002**, (online: <http://www.ingentaconnect.com/content/ben/cchts/2002/00000005/00000002/art00007>)
- [Rogers-Hahn-2010] D. Rogers and M. Hahn, **Extended-Connectivity Fingerprints**, *Journal of Chemical Information and Modeling*, Vol. 50, pp. 742–754, **2010**, (online: <http://pubs.acs.org/doi/abs/10.1021/ci100050t>)
- [Vidal-2005] D. Vidal, M. Thormann, and M. Pons, **LINGO, an Efficient Holographic Text Based Method To Calculate Biophysical Properties and Intermolecular Similarities**, *Journal of Chemical Information and Modeling*, Vol. 45, pp. 386–393, **2005**, (online: <http://pubs.acs.org/doi/abs/10.1021/ci0496797>)
- [Vidal-2006] D. Vidal, M. Thormann, and M. Pons, **A Novel Search Engine for Virtual Screening of Very Large Databases**, *Journal of Chemical Information and Modeling*, Vol. 46, pp. 836–843, **2006**, (online: <http://pubs.acs.org/doi/abs/10.1021/ci050458q>)
- [Willett-1998] P. Willett, **Chemical Similarity Searching**, *Journal of Chemical Information and Computer Science*, Vol. 38, pp. 983–996, **1998**, (online: <http://pubs.acs.org/doi/abs/10.1021/ci9800211>)

INDEX

A

AddFP

OEGraphSim::OEFPDatabase, 44

C

canonical isomeric SMILES, 77

canonical SMILES, 77

ClearBits

OESystem::OEBitVector, 40

ClearCutoff

OEGraphSim::OEFPDatabase, 45

Constructors

OEGraphSim::OEFingerPrint, 53

OEGraphSim::OEFPDatabase, 44

OEGraphSim::OEFPTTypeBase, 49

OEGraphSim::OEFPTTypeParams, 50

OEGraphSim::OESimFuncBase, 55

OEGraphSim::OESimScore, 55

OEGraphSim::OETanimotoSim, 56

OEGraphSim::OETverskySim, 57

OESystem::OEBitVector, 39

Cosine

similarity measure, 16

CountBits

OESystem::OEBitVector, 40

CreateCopy

OEGraphSim::OESimFuncBase, 55

OEGraphSim::OETanimotoSim, 56

OEGraphSim::OETverskySim, 57

D

Dice

similarity measure, 16

E

Euclidean

similarity measure, 16

Example Code

FP2OEB.cpp, 12

FP2SDF.cpp, 13

FPAtomTyping.cpp, 25

FPCoverage.cpp, 31

FPData.cpp, 11

FPOverlap.cpp, 34

FPPathLength.cpp, 29

FPTanimoto.cpp, 17

FPTType.cpp, 7

FPTTypeParams.cpp, 10

MemSimSearch.cpp, 22

PathFPTType.cpp, 25

SDF2FP.cpp, 14

SimCalcFromFile.cpp, 19

F

fingerprint, 77

FirstBit

OESystem::OEBitVector, 41

FP2OEB.cpp

Example Code, 12

FP2SDF.cpp

Example Code, 13

FPAtomTyping.cpp

Example Code, 25

FPCoverage.cpp

Example Code, 31

FPData.cpp

Example Code, 11

FPOverlap.cpp

Example Code, 34

FPPathLength.cpp

Example Code, 29

FPTanimoto.cpp

Example Code, 17

FPTType.cpp

Example Code, 7

FPTTypeParams.cpp

Example Code, 10

FromHexString

OESystem::OEBitVector, 41

G

GetAlpha

OEGraphSim::OETverskySim, 58
 GetAtomTypes
 OEGraphSim::OEFPTYPEParams, 51
 GetBeta
 OEGraphSim::OETverskySim, 58
 GetBondTypes
 OEGraphSim::OEFPTYPEParams, 51
 GetCutoff
 OEGraphSim::OEFPDATABASE, 45
 GetData
 OESystem::OEBitVector, 41
 GetFingerPrints
 OEGraphSim::OEFPDATABASE, 45
 GetFPType
 OEGraphSim::OEFPTYPEBase, 49
 OEGraphSim::OEFPTYPEParams, 51
 GetFPTypeBase
 OEGraphSim::OEFingerPrint, 54
 OEGraphSim::OEFPDATABASE, 45
 GetFPTypeString
 OEGraphSim::OEFPTYPEBase, 49
 GetFPVersion
 OEGraphSim::OEFPTYPEBase, 50
 GetFPVersionString
 OEGraphSim::OEFPTYPEBase, 50
 GetIdx
 OEGraphSim::OESimScore, 56
 GetMaxDistance
 OEGraphSim::OEFPTYPEParams, 51
 GetMinDistance
 OEGraphSim::OEFPTYPEParams, 52
 GetNumBits
 OEGraphSim::OEFPTYPEParams, 52
 GetScore
 OEGraphSim::OESimScore, 56
 GetScores
 OEGraphSim::OEFPDATABASE, 45
 GetSimTypeString
 OEGraphSim::OESimFuncBase, 55
 OEGraphSim::OETanimotoSim, 57
 OEGraphSim::OETverskySim, 58
 GetSize
 OESystem::OEBitVector, 41
 GetSortedScores
 OEGraphSim::OEFPDATABASE, 47
 GetVersion
 OEGraphSim::OEFPTYPEParams, 52

H

HasCutoff
 OEGraphSim::OEFPDATABASE, 48

I

IsBitOn

OESystem::OEBitVector, 41
 IsEmpty
 OESystem::OEBitVector, 41
 IsValid
 OEGraphSim::OEFPTYPEParams, 52

L

LastBit
 OESystem::OEBitVector, 42
 LINGO, 77

M

Manhattan
 similarity measure, 16
 MemSimSearch.cpp
 Example Code, 22

N

NegateBits
 OESystem::OEBitVector, 42
 NextBit
 OESystem::OEBitVector, 42
 NumFingerPrints
 OEGraphSim::OEFPDATABASE, 48

O

OEGraphSim::OECosine, 67
 OEGraphSim::OEDice, 67
 OEGraphSim::OEEuclid, 67
 OEGraphSim::OEFingerPrint, 52
 Constructors, 53
 GetFPTypeBase, 54
 operator
 =, 53
 operator bool, 54
 operator=, 53
 operator==, 53
 SetFPTypeBase, 54
 OEGraphSim::OEFPAAtomType, 58
 OEGraphSim::OEFPAAtomType::Aromaticity, 59
 OEGraphSim::OEFPAAtomType::AtomicNumber, 58
 OEGraphSim::OEFPAAtomType::Chiral, 59
 OEGraphSim::OEFPAAtomType::DefaultAtom, 61
 OEGraphSim::OEFPAAtomType::DefaultCircularAtom,
 61
 OEGraphSim::OEFPAAtomType::DefaultPathAtom, 62
 OEGraphSim::OEFPAAtomType::DefaultTreeAtom, 62
 OEGraphSim::OEFPAAtomType::EqAromatic, 60
 OEGraphSim::OEFPAAtomType::EqHalogen, 60
 OEGraphSim::OEFPAAtomType::EqHBondAcceptor, 61
 OEGraphSim::OEFPAAtomType::EqHBondDonor, 61
 OEGraphSim::OEFPAAtomType::FormalCharge, 59
 OEGraphSim::OEFPAAtomType::HCount, 60

- OEGraphSim::OEFPAtomType::HvyDegree, 59
- OEGraphSim::OEFPAtomType::Hybridization, 59
- OEGraphSim::OEFPAtomType::InRing, 59
- OEGraphSim::OEFPBondType, 63
- OEGraphSim::OEFPBondType::BondOrder, 63
- OEGraphSim::OEFPBondType::Chiral, 63
- OEGraphSim::OEFPBondType::DefaultBond, 63
- OEGraphSim::OEFPBondType::DefaultCircularBond, 64
- OEGraphSim::OEFPBondType::DefaultPathBond, 64
- OEGraphSim::OEFPBondType::DefaultTreeBond, 64
- OEGraphSim::OEFPBondType::InRing, 63
- OEGraphSim::OEFPDatabase, 44
 - AddFP, 44
 - ClearCutoff, 45
 - Constructors, 44
 - GetCutoff, 45
 - GetFingerPrints, 45
 - GetFPTypeBase, 45
 - GetScores, 45
 - GetSortedScores, 47
 - HasCutoff, 48
 - NumFingerPrints, 48
 - SetCutoff, 48
 - SetSimFunc, 48
- OEGraphSim::OEFPTType, 64
- OEGraphSim::OEFPTType::Circular, 64
- OEGraphSim::OEFPTType::Lingo, 65
- OEGraphSim::OEFPTType::MACCS166, 65
- OEGraphSim::OEFPTType::Path, 65
- OEGraphSim::OEFPTType::Tree, 65
- OEGraphSim::OEFPTTypeBase, 49
 - Constructors, 49
 - GetFPType, 49
 - GetFPTypeString, 49
 - GetFPVersion, 50
 - GetFPVersionString, 50
- OEGraphSim::OEFPTTypeParams, 50
 - Constructors, 50
 - GetAtomTypes, 51
 - GetBondTypes, 51
 - GetFPType, 51
 - GetMaxDistance, 51
 - GetMinDistance, 52
 - GetNumBits, 52
 - GetVersion, 52
 - IsValid, 52
- OEGraphSim::OEGetBitCounts, 67
- OEGraphSim::OEGetFingerPrintVersion, 70
- OEGraphSim::OEGetFingerPrintVersionString, 70
- OEGraphSim::OEGetFPAAtomType, 68
- OEGraphSim::OEGetFPBondType, 68
- OEGraphSim::OEGetFPCoverage, 69
- OEGraphSim::OEGetFPOverlap, 69
- OEGraphSim::OEGetFPType, 69
- OEGraphSim::OEGraphSimGetArch, 70
- OEGraphSim::OEGraphSimGetLicensee, 70
- OEGraphSim::OEGraphSimGetPlatform, 70
- OEGraphSim::OEGraphSimGetRelease, 70
- OEGraphSim::OEGraphSimGetSite, 71
- OEGraphSim::OEGraphSimGetVersion, 71
- OEGraphSim::OEGraphSimIsLicensed, 71
- OEGraphSim::OEIsFPType, 71
- OEGraphSim::OEIsSameFPType, 71
- OEGraphSim::OEIsValidFPTypeString, 71
- OEGraphSim::OEMakeCircularFP, 72
- OEGraphSim::OEMakeFP, 72
- OEGraphSim::OEMakeLingoFP, 73
- OEGraphSim::OEMakeMACCS166FP, 73
- OEGraphSim::OEMakePathFP, 73
- OEGraphSim::OEMakeTreeFP, 74
- OEGraphSim::OEManhattan, 75
- OEGraphSim::OESimFuncBase, 54
 - Constructors, 55
 - CreateCopy, 55
 - GetSimTypeString, 55
 - operator(), 55
- OEGraphSim::OESimMeasure, 65
- OEGraphSim::OESimMeasure::Cosine, 65
- OEGraphSim::OESimMeasure::Dice, 66
- OEGraphSim::OESimMeasure::Euclid, 66
- OEGraphSim::OESimMeasure::Manhattan, 66
- OEGraphSim::OESimMeasure::Tanimoto, 66
- OEGraphSim::OESimMeasure::Tversky, 66
- OEGraphSim::OESimScore, 55
 - Constructors, 55
 - GetIdx, 56
 - GetScore, 56
- OEGraphSim::OETanimoto, 75
- OEGraphSim::OETanimotoSim, 56
 - Constructors, 56
 - CreateCopy, 56
 - GetSimTypeString, 57
 - operator(), 56
- OEGraphSim::OETversky, 75
- OEGraphSim::OETverskySim, 57
 - Constructors, 57
 - CreateCopy, 57
 - GetAlpha, 58
 - GetBeta, 58
 - GetSimTypeString, 58
 - operator(), 57
- OESystem::OEBitVector, 39
 - ClearBits, 40
 - Constructors, 39
 - CountBits, 40
 - FirstBit, 41
 - FromHexString, 41

GetData, 41
 GetSize, 41
 IsBitOn, 41
 IsEmpty, 41
 LastBit, 42
 NegateBits, 42
 NextBit, 42
 operator=, 40
 operator=, 40
 operator&=, 40
 operator^=, 40
 operator|=, 40
 operator<, 39
 operator[], 40
 PrevBit, 42
 SetBitOff, 42
 SetBitOn, 42
 SetData, 43
 SetRangeOff, 43
 SetRangeOn, 43
 SetSize, 43
 ToggleBit, 43
 ToHexString, 43
 operator
 =
 OEGraphSim::OEFingerPrint, 53
 operator bool
 OEGraphSim::OEFingerPrint, 54
 operator()
 OEGraphSim::OESimFuncBase, 55
 OEGraphSim::OETanimotoSim, 56
 OEGraphSim::OETverskySim, 57
 operator==
 OESystem::OEBitVector, 40
 operator=
 OEGraphSim::OEFingerPrint, 53
 OESystem::OEBitVector, 40
 operator==
 OEGraphSim::OEFingerPrint, 53
 operator&=
 OESystem::OEBitVector, 40
 operator^=
 OESystem::OEBitVector, 40
 operator|=
 OESystem::OEBitVector, 40
 operator<
 OESystem::OEBitVector, 39
 operator[]
 OESystem::OEBitVector, 40

P

PathFPTType.cpp
 Example Code, 25
 PrevBit

OESystem::OEBitVector, 42

S

SDF2FP.cpp
 Example Code, 14
 SetBitOff
 OESystem::OEBitVector, 42
 SetBitOn
 OESystem::OEBitVector, 42
 SetCutoff
 OEGraphSim::OEFPDatabase, 48
 SetData
 OESystem::OEBitVector, 43
 SetFPTTypeBase
 OEGraphSim::OEFingerPrint, 54
 SetRangeOff
 OESystem::OEBitVector, 43
 SetRangeOn
 OESystem::OEBitVector, 43
 SetSimFunc
 OEGraphSim::OEFPDatabase, 48
 SetSize
 OESystem::OEBitVector, 43
 SimCalcFromFile.cpp
 Example Code, 19
 similarity measure
 Cosine, 16
 Dice, 16
 Euclidean, 16
 Manhattan, 16
 Tanimoto, 17
 Tversky, 17
 SMARTS, 77
 SMILES, 77
 structural key, 77

T

Tanimoto
 similarity measure, 17
 ToggleBit
 OESystem::OEBitVector, 43
 ToHexString
 OESystem::OEBitVector, 43
 Tversky
 similarity measure, 17