ChemmineR: Cheminformatics of Small Molecule Data

Yiqun Cao, Tyler Backman, Yan Wang, Thomas Girke
Email contact: thomas.girke@ucr.edu

August 28, 2013

1 Introduction

ChemmineR is a cheminformatics package for analyzing drug-like small molecule data in R. Its latest version contains functions for efficient processing of large numbers of small molecules, physicochemical/structural property predictions, structural similarity searching, classification and clustering of compound libraries with a wide spectrum of algorithms.

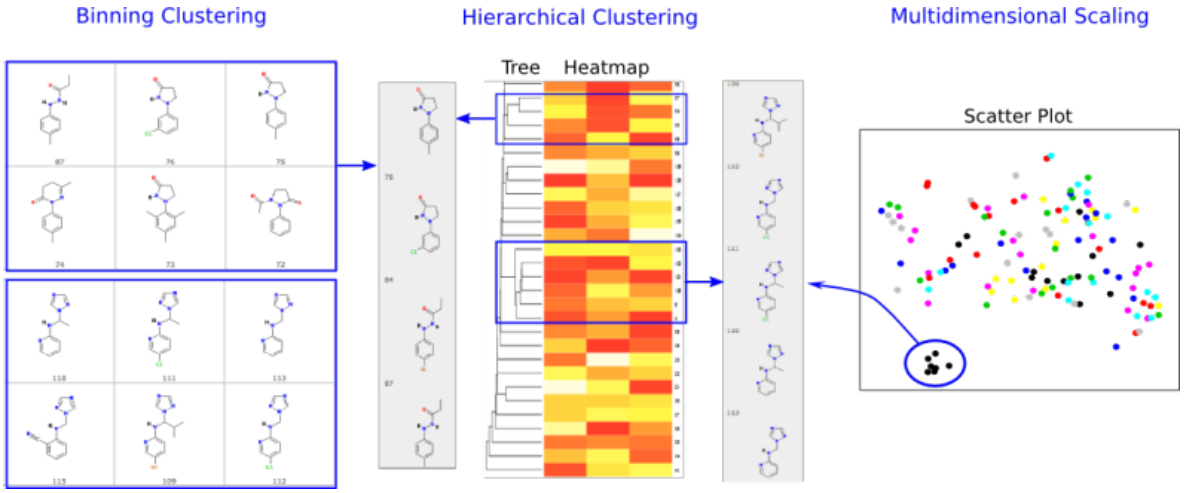


Figure 1: Selected functionalities provided by the *ChemmineR* package.

In addition, *ChemmineR* offers visualization functions for compound clustering results and chemical structures. The integration of chemoinformatic tools with the R programming environment has many advantages, such as easy access to a wide spectrum of statistical methods, machine learning algorithms and graphic utilities. The first version of this package was published in Cao et al. (2008). Since then many additional utilities have been added to the package and many more are under development for future releases (Backman et al., 2011; Wang et al., 2013).

2 Recently Added Features

- Streaming functionality for processing millions of molecules on a laptop
- Maximum common substructure (MCS) search algorithm

- Fast and memory efficient fingerprint search support using atom pair or PubChem fingerprints

3 Getting Started

3.1 Installation

The R software for running ChemmineR can be downloaded from CRAN (`http://cran.at.r-project.org/`). The ChemmineR package can be installed from R using the `bioLite` install command.

```
> source("http://bioconductor.org/biocLite.R") # Sources the biocLite.R installation script.
> biocLite("ChemmineR") # Installs the package.
```

3.2 Loading the Package and Documentation

```
> library("ChemmineR") # Loads the package

> library(help="ChemmineR") # Lists all functions and classes
> vignette("ChemmineR") # Opens this PDF manual from R
```

3.3 Five Minute Tutorial

The following code gives an overview of the most important functionalities provided by *ChemmineR*. Copy and paste of the commands into the R console will demonstrate their utilities.

Create Instances of *SDFset* class:

```
> data(sdfsample)
> sdfset <- sdfsample
> sdfset # Returns summary of SDFset

An instance of "SDFset" with 100 molecules

> sdfset[1:4] # Subsetting of object

An instance of "SDFset" with 4 molecules

> sdfset[[1]] # Returns summarized content of one SDF

An instance of "SDF"

<<header>>
                          Molecule_Name
                               "650001"
                                 Source
                "  -OEChem-07071010512D"
                                Comment
                                     ""
```

```
                              Counts_Line
" 61 64  0     0  0  0  0  0  0999 V2000"

<<atomblock>>
          C1      C2  C3  C5  C6  C7  C8  C9 C10 C11 C12 C13 C14 C15 C16
O_1   7.0468  0.0839   0   0   0   0   0   0   0   0   0   0   0   0   0
O_2  12.2708  1.0492   0   0   0   0   0   0   0   0   0   0   0   0   0
...       ...     ... ... ... ... ... ... ... ... ... ... ... ... ... ...
H_60  1.8411 -1.5985   0   0   0   0   0   0   0   0   0   0   0   0   0
H_61  2.6597 -1.2843   0   0   0   0   0   0   0   0   0   0   0   0   0

<<bondblock>>
     C1  C2  C3  C4  C5  C6  C7
1     1  16   2   0   0   0   0
2     2  23   1   0   0   0   0
... ... ... ... ... ... ... ...
63   33  60   1   0   0   0   0
64   33  61   1   0   0   0   0

<<datablock>> (33 data items)
          PUBCHEM_COMPOUND_CID PUBCHEM_COMPOUND_CANONICALIZED
                      "650001"                             "1"
     PUBCHEM_CACTVS_COMPLEXITY  PUBCHEM_CACTVS_HBOND_ACCEPTOR
                         "700"                             "7"

                         "..."
```

```
> view(sdfset[1:4]) # Returns summarized content of many SDFs, not printed here
> as(sdfset[1:4], "list") # Returns complete content of many SDFs, not printed here
```

An *SDFset* is created during the import of an SD file:

```
> sdfset <- read.SDFset("http://faculty.ucr.edu/~tgirke/Documents/
+                       R_BioCond/Samples/sdfsample.sf")
```

Miscellaneous accessor methods for *SDFset* container:

```
> header(sdfset[1:4]) # Not printed here
```

```
> header(sdfset[[1]])
```

```
                           Molecule_Name
                                "650001"
                                  Source
                  "  -OEChem-07071010512D"
                                 Comment
                                      ""
                              Counts_Line
" 61 64  0     0  0  0  0  0  0999 V2000"
```

```
> atomblock(sdfset[1:4]) # Not printed here

> atomblock(sdfset[[1]])[1:4,]

         C1     C2 C3 C5 C6 C7 C8 C9 C10 C11 C12 C13 C14 C15 C16
O_1  7.0468 0.0839  0  0  0  0  0  0   0   0   0   0   0   0   0
O_2 12.2708 1.0492  0  0  0  0  0  0   0   0   0   0   0   0   0
O_3 12.2708 3.1186  0  0  0  0  0  0   0   0   0   0   0   0   0
O_4  7.9128 2.5839  0  0  0  0  0  0   0   0   0   0   0   0   0

> bondblock(sdfset[1:4]) # Not printed here

> bondblock(sdfset[[1]])[1:4,]

  C1 C2 C3 C4 C5 C6 C7
1  1 16  2  0  0  0  0
2  2 23  1  0  0  0  0
3  2 27  1  0  0  0  0
4  3 25  1  0  0  0  0

> datablock(sdfset[1:4]) # Not printed here

> datablock(sdfset[[1]])[1:4]

          PUBCHEM_COMPOUND_CID PUBCHEM_COMPOUND_CANONICALIZED
                      "650001"                             "1"
     PUBCHEM_CACTVS_COMPLEXITY  PUBCHEM_CACTVS_HBOND_ACCEPTOR
                         "700"                             "7"
```

Assigning compound IDs and keeping them unique:

```
> cid(sdfset)[1:4] # Returns IDs from SDFset object

[1] "CMP1" "CMP2" "CMP3" "CMP4"

> sdfid(sdfset)[1:4] # Returns IDs from SD file header block

[1] "650001" "650002" "650003" "650004"

> unique_ids <- makeUnique(sdfid(sdfset))

[1] "No duplicates detected!"

> cid(sdfset) <- unique_ids
```

Converting the data blocks in an *SDFset* to a matrix:

```
> blockmatrix <- datablock2ma(datablocklist=datablock(sdfset))
>    # Converts data block to matrix
> numchar <- splitNumChar(blockmatrix=blockmatrix)
>    # Splits to numeric and character matrix
> numchar[[1]][1:2,1:2] # Slice of numeric matrix
```

```
       PUBCHEM_COMPOUND_CID PUBCHEM_COMPOUND_CANONICALIZED
650001               650001                             1
650002               650002                             1

> numchar[[2]][1:2,10:11] # Slice of character matrix

       PUBCHEM_MOLECULAR_FORMULA
650001 "C23H28N4O6"
650002 "C18H23N5O3"
       PUBCHEM_OPENEYE_CAN_SMILES
650001 "CC1=CC(=NO1)NC(=O)CCC(=O)N(CC(=O)NC2CCCC2)C3=CC4=C(C=C3)OCCO4"
650002 "CN1C2=C(C(=O)NC1=O)N(C(=N2)NCCCO)CCCC3=CC=CC=C3"
```

Compute atom frequency matrix, molecular weight and formula:

```
> propma <- data.frame(MF=MF(sdfset), MW=MW(sdfset), atomcountMA(sdfset))
> propma[1:4, ]

                MF       MW  C  H N O S F Cl
650001  C23H28N4O6 456.4916 23 28 4 6 0 0  0
650002  C18H23N5O3 357.4069 18 23 5 3 0 0  0
650003 C18H18N4O3S 370.4255 18 18 4 3 1 0  0
650004 C21H27N5O5S 461.5346 21 27 5 5 1 0  0
```

Assign matrix data to data block:

```
> datablock(sdfset) <- propma
> datablock(sdfset[1])

$`650001`
          MF           MW            C            H            N            O
"C23H28N4O6"   "456.4916"         "23"         "28"          "4"          "6"
           S            F           Cl
         "0"          "0"          "0"
```

String searching in *SDFset* ():

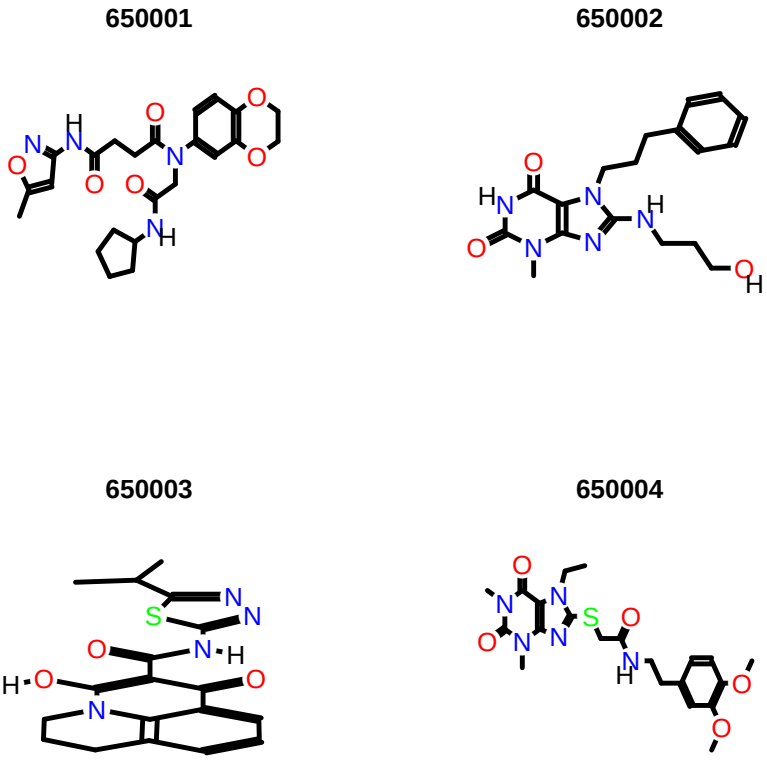
```
> grepSDFset("650001", sdfset, field="datablock", mode="subset")
>    # Returns summary view of matches. Not printed here.
> .

> grepSDFset("650001", sdfset, field="datablock", mode="index")

1 1 1 1 1 1 1 1 1
1 2 3 4 5 6 7 8 9
```

Export SDFset to SD file:

```
> write.SDF(sdfset[1:4], file="sub.sdf", sig=TRUE)
```

Plot molecule structure of one or many SDFs:

```
> plot(sdfset[1:4], print=FALSE) # Plots structures to R graphics device
```




```
> sdf.visualize(sdfset[1:4]) # Compound viewing in web browser
```

Structure similarity searching and clustering:

```
> apset <- sdf2ap(sdfset)
>    # Generate atom pair descriptor database for searching
> .

> data(apset)
>    # Load sample apset data provided by library.
> cmp.search(apset, apset[1], type=3, cutoff = 0.3, quiet=TRUE)

  index    cid    scores
1     1 650001 1.0000000
2    96 650102 0.3516643
3    67 650072 0.3117569
4    88 650094 0.3094629
5    15 650015 0.3010753
```

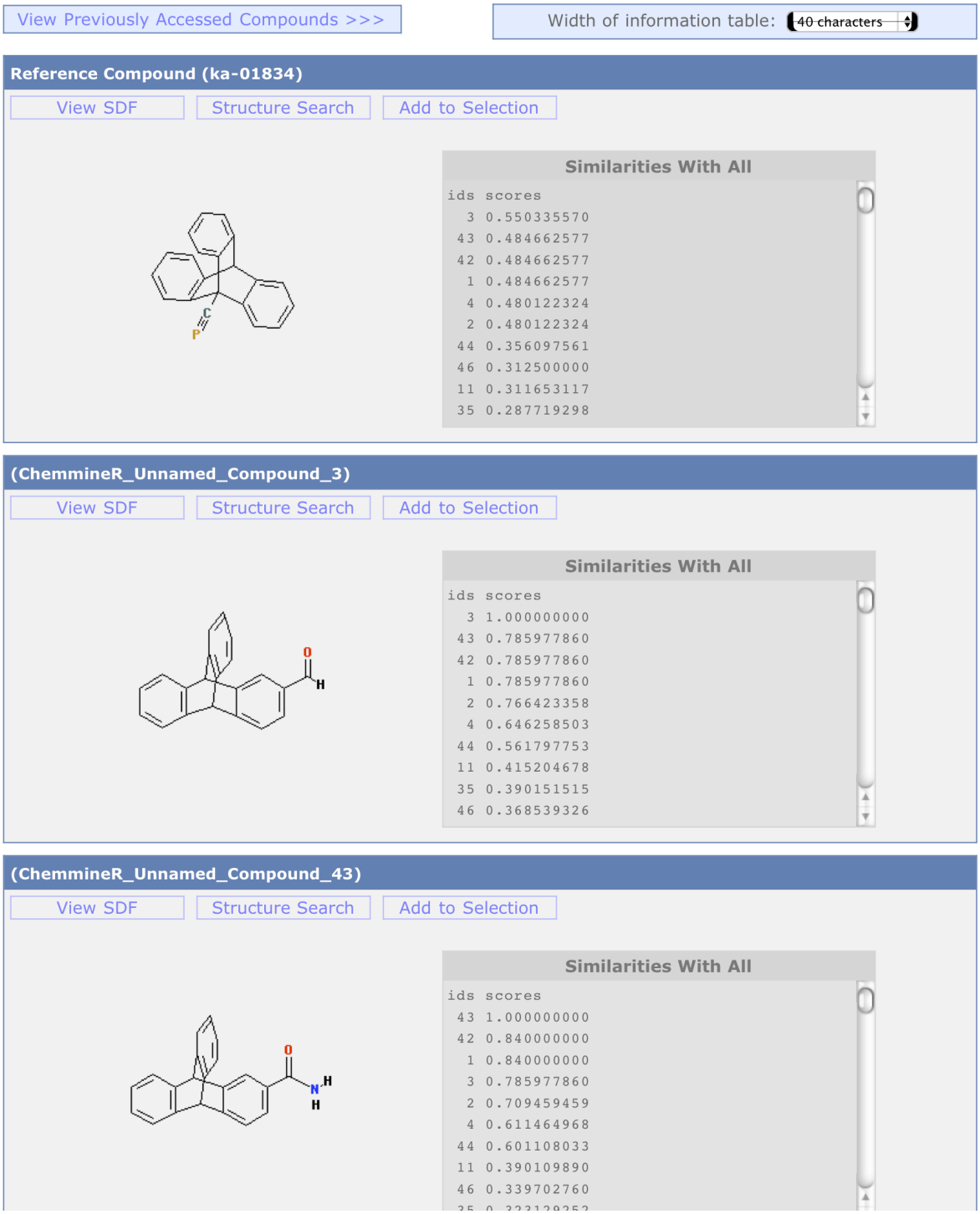


Figure 2: Visualization webpage created by calling `sdf.visualize`.

```
>    # Search apset database with single compound.
> cmp.cluster(db=apset, cutoff = c(0.65, 0.5), quiet=TRUE)[1:4,]

sorting result...
      ids CLSZ_0.65 CLID_0.65 CLSZ_0.5 CLID_0.5
48 650049         2        48        2       48
49 650050         2        48        2       48
54 650059         2        54        2       54
55 650060         2        54        2       54

>    # Binning clustering using variable similarity cutoffs.
```

4 Overview of Classes and Functions

The following list gives an overview of the most important S4 classes, methods and functions available in the ChemmineR package. The help documents of the package provide much more detailed information on each utility. The standard R help documents for these utilities can be accessed with this syntax: `?function_name` (*e.g.* `?cid`) and `?class_name-class` (*e.g.* `?"SDFset-class"`).

4.1 Molecular Structure Data

Classes

- *SDFstr*: intermediate string class to facilitate SD file import; not important for end user
- *SDF*: container for single molecule imported from an SD file
- *SDFset*: container for many SDF objects; most important structure container for end user

Functions/Methods

- Accessor methods for *SDF/SDFset*
  - Object slots: `cid, header, atomblock, bondblock, datablock (sdfid, datablocktag)`
  - Summary of *SDFset*: `view`
  - Matrix conversion of data block: `datablock2ma, splitNumChar`
  - String search in SDFset: `grepSDFset`
- Coerce one class to another
  - Standard syntax `as(..., "...")` works in most cases. For details see R help with *?"SDFset-class"*.
- Utilities
  - Atom frequencies: `atomcountMA, atomcount`
  - Molecular weight: `MW`
  - Molecular formula: `MF`
  - ...
- Compound structure depictions
  - R graphics device: `plot, plotStruc`
  - Online: `cmp.visualize`

4.2 Structure Descriptor Data

Classes

- *AP*: container for atom pair descriptors of a single molecule
- *APset*: container for many AP objects; most important structure descriptor container for end user
- *FP*: container for fingerprint of a single molecule
- *FPset*: container for fingerprints of many molecules, most important structure descriptor container for end user

Functions/Methods

- Create *AP/APset* instances
  - From *SDFset*: `sdf2ap`
  - From SD file: `cmp.parse`
  - Summary of *AP/APset*: `view, db.explain`
- Accessor methods for AP/APset
  - Object slots: `ap, cid`
- Coerce one class to another
  - Standard syntax `as(..., "...")` works in most cases. For details see R help with *?"APset-class"*.
- Structure Similarity comparisons and Searching
  - Compute pairwise similarities : `cmp.similarity, fpSim`
  - Search APset database: `cmp.search, fpSim`
- AP-based Structure Similarity Clustering
  - Single-linkage binning clustering: `cmp.cluster`
  - Visualize clustering result with MDS: `cluster.visualize`
  - Size distribution of clusters: `cluster.sizestat`

5 Importing Compounds

The following code gives an overview of the most important import/export functionalities provided by *ChemmineR*. The example creates an instance of the *SDFset* class using as sample data set the first 100 compounds from this PubChem SD file (SDF): Compound_00650001_00675000.sdf.gz (`ftp://ftp.ncbi.nih.gov/pubchem/Compound/CURRENT-Full/SDF/`).

SDFs can be imported with the `read.SDFset` function:

```
> sdfset <- read.SDFset("http://faculty.ucr.edu/~tgirke/Documents/
+                        R_BioCond/Samples/sdfsample.sdf")

> data(sdfsample) # Loads the same SDFset provided by the library
> sdfset <- sdfsample
> valid <- validSDF(sdfset) # Identifies invalid SDFs in SDFset objects
> sdfset <- sdfset[valid] # Removes invalid SDFs, if there are any
```

Import SD file into *SDFstr* container:

```
> sdfstr <- read.SDFstr("http://faculty.ucr.edu/~tgirke/Documents/
+                        R_BioCond/Samples/sdfsample.sdf")
```

Create *SDFset* from *SDFstr* class:

```
> sdfstr <- as(sdfset, "SDFstr")
> sdfstr

An instance of "SDFstr" with 100 molecules

> as(sdfstr, "SDFset")

An instance of "SDFset" with 100 molecules
```

6 Export of Compounds

Write objects of classes *SDFset/SDFstr/SDF* to SD file:

```
> write.SDF(sdfset[1:4], file="sub.sdf")
```

Writing customized *SDFset* to file containing *ChemmineR* signature, IDs from *SDFset* and no data block:

```
> write.SDF(sdfset[1:4], file="sub.sdf", sig=TRUE, cid=TRUE, db=NULL)
```

Example for injecting a custom matrix/data frame into the data block of an *SDFset* and then writing it to an SD file:

```
> props <- data.frame(MF=MF(sdfset), MW=MW(sdfset), atomcountMA(sdfset))
> datablock(sdfset) <- props
> write.SDF(sdfset[1:4], file="sub.sdf", sig=TRUE, cid=TRUE)
```

Indirect export via *SDFstr* object:

```
> sdf2str(sdf=sdfset[[1]], sig=TRUE, cid=TRUE)
>    # Uses default components
> sdf2str(sdf=sdfset[[1]], head=letters[1:4], db=NULL)
>    # Uses custom components for header and data block
```

Write *SDF*, *SDFset* or *SDFstr* classes to file:

```
> write.SDF(sdfset[1:4], file="sub.sdf", sig=TRUE, cid=TRUE, db=NULL)
> write.SDF(sdfstr[1:4], file="sub.sdf")
> cat(unlist(as(sdfstr[1:4], "list")), file="sub.sdf", sep="\n")
```

7 Splitting SD Files

The following `write.SDFsplit` function allows to split SD Files into any number of smaller SD Files. This can become important when working with very big SD Files. Users should note that this function can output many files, thus one should run it in a dedicated directory!

Create sample SD File with 100 molecules:

```
> write.SDF(sdfset, "test.sdf")
```

Read in sample SD File. Note: reading file into SDFstr is much faster than into SDFset:

```
> sdfstr <- read.SDFstr("test.sdf")
```

Run export on *SDFstr* object:

```
> write.SDFsplit(x=sdfstr, filetag="myfile", nmol=10)
>    # 'nmol' defines the number of molecules to write to each file
```

Run export on *SDFset* object:

```
> write.SDFsplit(x=sdfset, filetag="myfile", nmol=10)
```

8 Streaming Through Large SD Files

The `sdfStream` function allows to stream through SD Files with millions of molecules without consuming much memory. During this process any set of descriptors, supported by *ChemmineR*, can be computed (*e.g.* atom pairs, molecular properties, etc.), as long as they can be returned in tabular format. In addition to descriptor values, the function returns a line index that gives the start and end positions of each molecule in the source SD File. This line index can be used by the downstream `read.SDFindex` function to retrieve specific molecules of interest from the source SD File without reading the entire file into R. The following outlines the typical workflow of this streaming functionality in *ChemmineR*.

Create sample SD File with 100 molecules:

```
> write.SDF(sdfset, "test.sdf")
```

Define descriptor set in a simple function:

```
> desc <- function(sdfset) {
+         cbind(SDFID=sdfid(sdfset),
+               # datablock2ma(datablocklist=datablock(sdfset)),
+               MW=MW(sdfset),
+               groups(sdfset),
+               APFP=desc2fp(x=sdf2ap(sdfset), descnames=1024, type="character"),
+               AP=sdf2ap(sdfset, type="character"),
+               rings(sdfset, type="count", upper=6, arom=TRUE)
+         )
+ }
```

Run `sdfStream` with `desc` function and write results to a file called `matrix.xls`:

```
> sdfStream(input="test.sdf", output="matrix.xls", fct=desc, Nlines=1000)
>    # 'Nlines': number of lines to read from input SD File at a time
```

One can also start reading from a specific line number in the SD file. The following example starts at line number 950. This is useful for restarting and debugging the process. With `append=TRUE` the result can be appended to an existing file.

```
> sdfStream(input="test.sdf", output="matrix2.xls", append=FALSE, fct=desc,
+           Nlines=1000, startline=950)
```

Select molecules meeting certain property criteria from SD File using line index generated by previous `sdfStream` step:

```
> indexDF <- read.delim("matrix.xls", row.names=1)[,1:4]
> indexDFsub <- indexDF[indexDF$MW < 400, ]
>    # Selects molecules with MW < 400
> sdfset <- read.SDFindex(file="test.sdf", index=indexDFsub, type="SDFset")
>    # Collects results in 'SDFset' container
```

Write results directly to SD file without storing larger numbers of molecules in memory:

```
> read.SDFindex(file="test.sdf", index=indexDFsub, type="file", outfile="sub.sdf")
```

Read AP/APFP strings from file into `APset` or `FP` object:

```
> apset <- read.AP(x="matrix.xls", type="ap", colid="AP")
> apfp <- read.AP(x="matrix.xls", type="fp", colid="APFP")
```

Alternatively, one can provide the AP/APFP strings in a named character vector:

```
> apset <- read.AP(x=sdf2ap(sdfset[1:20], type="character"), type="ap")
> fpchar <- desc2fp(sdf2ap(sdfset[1:20]), descnames=1024, type="character")
> fpset <- as(fpchar, "FPset")
```

9 Storing Compounds in an SQL Database

As an alternative to sdfStream, there is now also an option to store data in an SQL database, which then allows for fast queries and compound retrieval. This is still an experimental feature. The default database is SQLite, but any other SQL database should work with some minor modifications to the table definitions, which are stored in schema/compounds.SQLite under the ChemmineR package directory. Compounds are stored in their entirety in the databases so there is no need to keep any original data files.

Users can define their own set of compound features to compute and store when loading new compounds. Each of these features will be stored in its own, indexed table. Searches can then be performed using these features to quickly find specific compounds. Compounds can always be retrieved quickly because of the database index, no need to scan a large compound file. In addition to user defined features, descriptors can also be computed and stored for each compound.

A new database can be created with the `initDb` function. This takes either an existing database connection, or a filename. If a filename is given then an SQLite database connection is created. It then ensures that the required tables exist and creates them if not. The connection object is then returned. This function can be called safely on the same connection or database many times and will not delete any data.

9.1 Loading Data

The function `loadSdf` can be used to load `SDF` data, either from a file or *SDFset* object. The `fct` parameter should be a function to extract features from the data. It will be handed an *SDFset* generated from the data being loaded. This may be done in batches, so there is no guarantee that the given SDFSset will contain the whole dataset. This function should return a data frame with a column for each feature and a row for each compound given. The order of the final data frame should be the same as that of the *SDFset*. The column names will become the feature names. Each of these features will become a new, indexed, table in the database which can be used later to search for compounds.

The `descriptors` parameter can be a function which computes descriptors. This function will also be given an *SDFset* object, which may be done in batches. It should return a data frame with the following two columns: "descriptor" and "descriptor_type". The "descriptor" column should contain a string representation of the descriptor, and "descriptor_type" is the type of the descriptor. Our convention for atom pair is "ap" and "fp" for finger print. The order should also be maintained.

When the data has been loaded, `loadSdf` will return the compound id numbers of each compound loaded. These compound id numbers are computed by the database and are not extracted from the compound data itself. They can be used to quickly retrieve compounds later.

New features can also be added using this function. However, all compounds must have all features so if new features are added to a new set of compounds, all existing features must be computable by the `fct` function given. If new features are detected, all existing compounds will be run through `fct` in order to compute the new features for them as well.

For example, if dataset X is loaded with features F1 and F2, and then at a later time we load dataset Y with new feature F3, the `fct` function used to load dataset Y must compute and return features F1, F2, and F3. `loadSdf` will call `fct` with both datasets X and Y so

that all features are available for all compounds. If any features are missing an error will be raised. If just new features are being added, but no new compounds, use the `addNewFeatures` function.

In this example, we create a new database called "test.db" and load it with data from and *SDFset*. We also define `fct` to compute the molecular weight, "MW", and the number of rings and aromatic rings. The rings function actually returns a data frame with columns "RINGS" and "AROMATIC", which will be merged into the data frame being created which will also contain the "MW" column. These will be the names used for these features and must be used when searching with them. Finally, the new compound ids are returned and stored in the "ids" variable.

```
> data(sdfsample)
> #create and initialize a new SQLite database
> conn = initDb("test.db")

[1] "createing db"

> # load data and compute 3 features: molecular weight, with the MW function,
> # and counts for RINGS and AROMATIC, as computed by rings, which
> # returns a data frame itself.
> ids=loadSdf(conn,sdfsample,
+       function(sdfset)
+        data.frame(MW = MW(sdfset),
+                   rings(sdfset,type="count",upper=6, arom=TRUE))
+       )
```

9.2 Searching

Compounds can be searched for using the `findCompounds` function. This function takes a connection object, a vector of feature names used in the tests, and finally, a vector of tests that must all pass for a compound to be included in the result set. Each test should be a boolean expression. For example: "c("MW <= 400","RINGS > 3")" would return all compounds with a molecular weight of 400 or less and more than 3 rings, assuming these features exist in the database. The syntax for each test is "<feature name> <SQL operator> <value>". If you know SQL you can go beyond this basic syntax. These tests will simply be concatenated together with "AND" in-between them and tacked on the end of a WHERE clause of an SQL statement. So any SQL that will work in that context is fine. The function will return a list of compound ids, the actual compounds can be fetched with `getCompounds`. If just the names are needed, the `getCompoundNames` function can be used. Compounds can also be fetched by name using the `findCompoundsByName` function.

In this example we search for compounds with molecular weight less than 300. We then fetch the matching compounds and show their molecular weight.

```
> lightIds = findCompounds(conn,"MW",c("MW < 300"))
> MW(getCompounds(conn,lightIds))

     206      214      217      222      224      228      229      230
262.3043 240.2541 265.3050 287.3999 266.7465 278.2789 282.3171 270.3479
```

```
     233      234      236      237      242      246      250      255
223.2252 265.3098 249.2162 285.2581 229.3159 248.3042 266.3427 275.3859
     256      257      268      273      276      282      289      295
299.8361 263.3752 296.3205 275.3693 294.3510 140.1399 294.3726 276.3357

> #names of matching compounds:
> getCompoundNames(conn,lightIds)

     282      233      242      214      246      236      206      257
"650088" "650034" "650043" "650014" "650047" "650037" "650006" "650062"
     217      234      250      224      230      273      255      295
"650017" "650035" "650052" "650025" "650031" "650078" "650060" "650101"
     228      229      237      222      276      289      268      256
"650029" "650030" "650038" "650023" "650081" "650095" "650073" "650061"
```

10 Working with SDF/SDFset Classes

Several methods are available to return the different data components of *SDF/SDFset* containers in batches. The following examples list the most important ones. To save space their content is not printed in the manual.

```
> view(sdfset[1:4]) # Summary view of several molecules
> length(sdfset) # Returns number of molecules
> sdfset[[1]] # Returns single molecule from SDFset as SDF object
> sdfset[[1]][[2]] # Returns atom block from first compound as matrix
> sdfset[[1]][[2]][1:4,]
> c(sdfset[1:4], sdfset[5:8]) # Concatenation of several SDFsets
```

The `grepSDFset` function allows string matching/searching on the different data components in *SDFset*. By default the function returns a SDF summary of the matching entries. Alternatively, an index of the matches can be returned with the setting *mode="index"*.

```
> grepSDFset("650001", sdfset, field="datablock", mode="subset")
>    # To return index, set mode="index")
> .
```

Utilities to maintain unique compound IDs:

```
> sdfid(sdfset[1:4])
>    # Retrieves CMP IDs from Molecule Name field in header block.
> cid(sdfset[1:4])
>    # Retrieves CMP IDs from ID slot in SDFset.
> unique_ids <- makeUnique(sdfid(sdfset))
>    # Creates unique IDs by appending a counter to duplicates.
> cid(sdfset) <- unique_ids # Assigns uniquified IDs to ID slot
```

Subsetting by character, index and logical vectors:

```
> view(sdfset[c("650001", "650012")])
> view(sdfset[4:1])
> mylog <- cid(sdfset) %in% c("650001", "650012")
> view(sdfset[mylog])
```

Accessing *SDF/SDFset* components: header, atom, bond and data blocks:

```
> atomblock(sdf); sdf[[2]]; sdf[["atomblock"]]
>    # All three methods return the same component
> header(sdfset[1:4])
> atomblock(sdfset[1:4])
> bondblock(sdfset[1:4])
> datablock(sdfset[1:4])
> header(sdfset[[1]])
> atomblock(sdfset[[1]])
> bondblock(sdfset[[1]])
> datablock(sdfset[[1]])
```

Replacement Methods:

```
> sdfset[[1]][[2]][1,1] <- 999
> atomblock(sdfset)[1] <- atomblock(sdfset)[2]
> datablock(sdfset)[1] <- datablock(sdfset)[2]
```

Assign matrix data to data block:

```
> datablock(sdfset) <- as.matrix(iris[1:100,])
> view(sdfset[1:4])
```

Class coercions from *SDFstr* to *list*, *SDF* and *SDFset*:

```
> as(sdfstr[1:2], "list")
> as(sdfstr[[1]], "SDF")
> as(sdfstr[1:2], "SDFset")
```

Class coercions from *SDF* to *SDFstr*, *SDFset*, list with SDF sub-components:

```
> sdfcomplist <- as(sdf, "list")
> sdfcomplist <- as(sdfset[1:4], "list"); as(sdfcomplist[[1]], "SDF")
> sdflist <- as(sdfset[1:4], "SDF"); as(sdflist, "SDFset")
> as(sdfset[[1]], "SDFstr")
> as(sdfset[[1]], "SDFset")
```

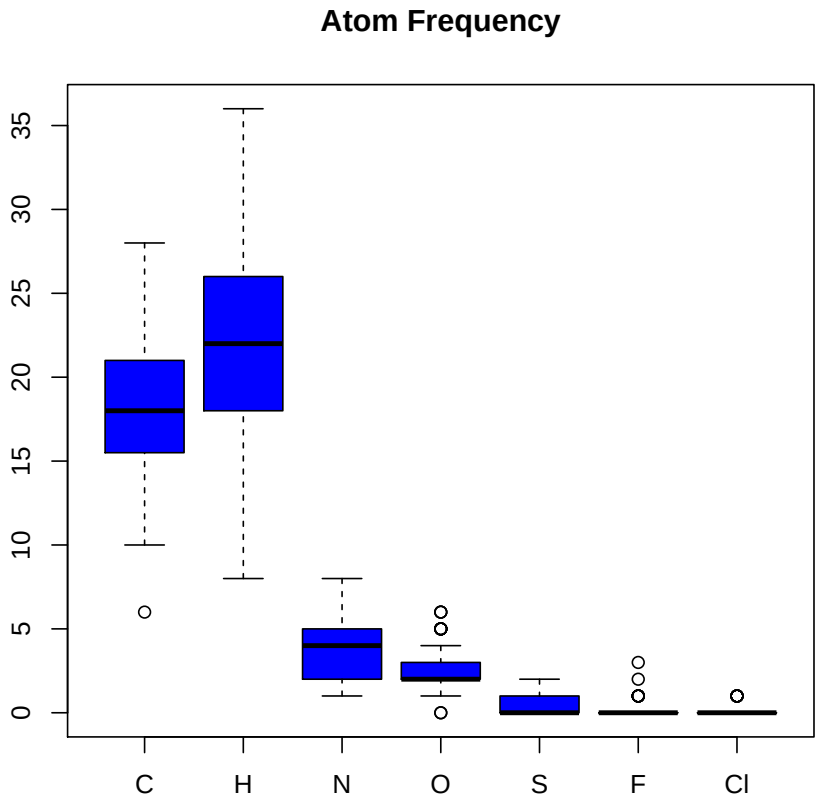
Class coercions from *SDFset* to lists with components consisting of SDF or sub-components:

```
> as(sdfset[1:4], "SDF")
> as(sdfset[1:4], "list")
> as(sdfset[1:4], "SDFstr")
```

11 Molecular Property Functions (Physicochemical Descriptors)

Several methods and functions are available to compute basic compound descriptors, such as molecular formula (MF), molecular weight (MW), and frequencies of atoms and functional groups. In many of these functions, it is important to set `addH=TRUE` in order to include/add hydrogens that are often not specified in an SD file.

```
> propma <- atomcountMA(sdfset, addH=FALSE)
> boxplot(propma, col="blue", main="Atom Frequency")
```




```
> boxplot(rowSums(propma), main="All Atom Frequency")
```

Data frame provided by library containing atom names, atom symbols, standard atomic weights, group and period numbers:

```
> data(atomprop)
> atomprop[1:4,]
```

```
  Number      Name Symbol Atomic_weight Group Period
1      1  hydrogen      H      1.007940     1      1
```

```
2      2    helium     He      4.002602    18      1
3      3   lithium     Li      6.941000     1      2
4      4 beryllium     Be      9.012182     2      2
```

Compute MW and formula:

```
> MW(sdfset[1:4], addH=FALSE)

    CMP1     CMP2     CMP3     CMP4
456.4916 357.4069 370.4255 461.5346

> MF(sdfset[1:4], addH=FALSE)

         CMP1          CMP2          CMP3          CMP4
 "C23H28N4O6"  "C18H23N5O3" "C18H18N4O3S" "C21H27N5O5S"
```

Enumerate functional groups:

```
> groups(sdfset[1:4], groups="fctgroup", type="countMA")

     RNH2 R2NH R3N ROPO3 ROH RCHO RCOR RCOOH RCOOR ROR RCCH RCN
CMP1    0    2   1     0   0    0    0     0     0   2    0   0
CMP2    0    2   2     0   1    0    0     0     0   0    0   0
CMP3    0    1   1     0   1    0    1     0     0   0    0   0
CMP4    0    1   3     0   0    0    0     0     0   2    0   0
```

Combine MW, MF, charges, atom counts, functional group counts and ring counts in one data frame:

```
> propma <- data.frame(MF=MF(sdfset, addH=FALSE), MW=MW(sdfset, addH=FALSE),
+                      Ncharges=sapply(bonds(sdfset, type="charge"), length),
+                      atomcountMA(sdfset, addH=FALSE), groups(sdfset,
+                      type="countMA"), rings(sdfset, upper=6, type="count",
+                      arom=TRUE))
> propma[1:4,]

              MF       MW Ncharges  C  H N O S F Cl RNH2 R2NH R3N ROPO3 ROH
CMP1  C23H28N4O6 456.4916        0 23 28 4 6 0 0  0    0    2   1     0   0
CMP2  C18H23N5O3 357.4069        0 18 23 5 3 0 0  0    0    2   2     0   1
CMP3 C18H18N4O3S 370.4255        0 18 18 4 3 1 0  0    0    1   1     0   1
CMP4 C21H27N5O5S 461.5346        0 21 27 5 5 1 0  0    0    1   3     0   0
     RCHO RCOR RCOOH RCOOR ROR RCCH RCN RINGS AROMATIC
CMP1    0    0     0     0   2    0   0     4        2
CMP2    0    0     0     0   0    0   0     3        3
CMP3    0    1     0     0   0    0   0     4        2
CMP4    0    0     0     0   2    0   0     3        3
```

The following shows an example for assigning the values stored in a matrix (*e.g.* property descriptors) to the data block components in an *SDFset*. Each matrix row will be assigned to the corresponding slot position in the *SDFset*.

```
> datablock(sdfset) <- propma # Works with all SDF components
> datablock(sdfset)[1:4]
> test <- apply(propma[1:4,], 1, function(x) data.frame(col=colnames(propma), value=x))
> sdf.visualize(sdfset[1:4], extra = test)
```

The data blocks in SDFs contain often important annotation information about compounds. The `datablock2ma` function returns this information as matrix for all compounds stored in an *SDFset* container. The `splitNumChar` function can then be used to organize all numeric columns in a *numeric matrix* and the character columns in a *character matrix* as components of a *list* object.

```
> datablocktag(sdfset, tag="PUBCHEM_NIST_INCHI")
> datablocktag(sdfset, tag="PUBCHEM_OPENEYE_CAN_SMILES")
```

Convert entire data block to matrix:

```
> blockmatrix <- datablock2ma(datablocklist=datablock(sdfset))
>    # Converts data block to matrix
> numchar <- splitNumChar(blockmatrix=blockmatrix)
>    # Splits matrix to numeric matrix and character matrix
> numchar[[1]][1:4,]; numchar[[2]][1:4,]
>    # Splits matrix to numeric matrix and character matrix
> .
```

12 Bond Matrices

Bond matrices provide an efficient data structure for many basic computations on small molecules. The function `conMA` creates this data structure from *SDF* and *SDFset* objects. The resulting bond matrix contains the atom labels in the row/column titles and the bond types in the data part. The labels are defined as follows: 0 is no connection, 1 is a single bond, 2 is a double bond and 3 is a triple bond.

```
> conMA(sdfset[1:2], exclude=c("H"))
>    # Create bond matrix for first two molecules in sdfset
> conMA(sdfset[[1]], exclude=c("H"))
>    # Return bond matrix for first molecule
> plot(sdfset[1], atomnum = TRUE, noHbonds=FALSE , no_print_atoms = "", atomcex=0.8)
>    # Plot its structure with atom numbering
> rowSums(conMA(sdfset[[1]], exclude=c("H")))
>    # Return number of non-H bonds for each atom
> .
```

13 Charges and Missing Hydrogens

The function `bonds` returns information about the number of bonds, charges and missing hydrogens in *SDF* and *SDFset* objects. It is used by many other functions (*e.g.* `MW`, `MF`, `atomcount`, `atomcuntMA` and `plot`) to correct for missing hydrogens that are often not specified in SD files.

```
> bonds(sdfset[[1]], type="bonds")[1:4,]
```

```
  atom Nbondcount Nbondrule charge
1    O          2         2      0
2    O          2         2      0
3    O          2         2      0
4    O          2         2      0
```

```
> bonds(sdfset[1:2], type="charge")
```

```
$CMP1
NULL

$CMP2
NULL
```

```
> bonds(sdfset[1:2], type="addNH")
```

```
CMP1 CMP2
   0    0
```

14 Ring Perception and Aromaticity Assignment

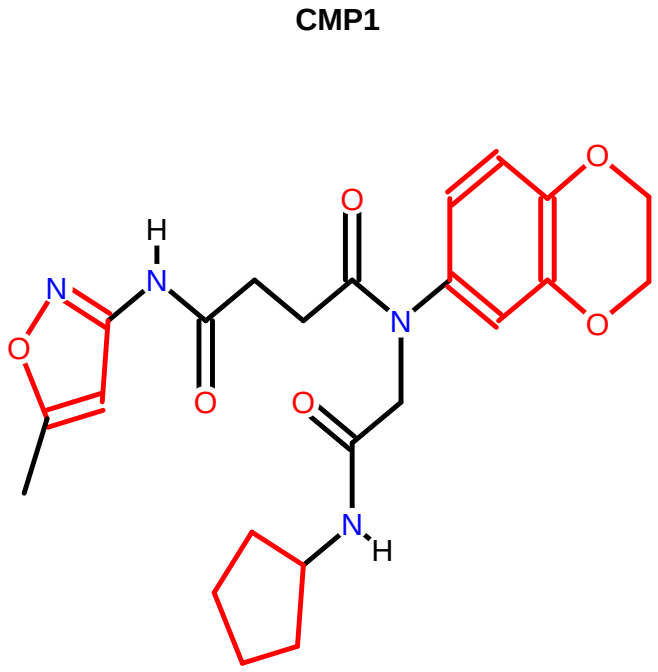
The function `rings` identifies all possible rings in one or many molecules (here *sdfset[1]*) using the exhaustive ring perception algorithm from Hanser et al. (1996). In addition, the function can return all smallest possible rings as well as aromaticity information.

The following example returns all possible rings in a *list*. The argument `upper` allows to specify an upper length limit for rings. Choosing smaller length limits will reduce the search space resulting in shortened compute times. Note: each ring is represented by a character vector of atom symbols that are numbered by their position in the atom block of the corresponding *SDF/SDFset* object.

```
> (ringatoms <- rings(sdfset[1], upper=Inf, type="all", arom=FALSE, inner=FALSE))

$ring1
[1] "N_10" "O_6"  "C_32" "C_31" "C_30"

$ring2
[1] "C_12" "C_14" "C_15" "C_13" "C_11"

$ring3
[1] "C_23" "O_2"  "C_27" "C_28" "O_3"  "C_25"

$ring4
[1] "C_23" "C_21" "C_18" "C_22" "C_26" "C_25"

$ring5
 [1] "O_3"  "C_28" "C_27" "O_2"  "C_23" "C_21" "C_18" "C_22" "C_26" "C_25"
```

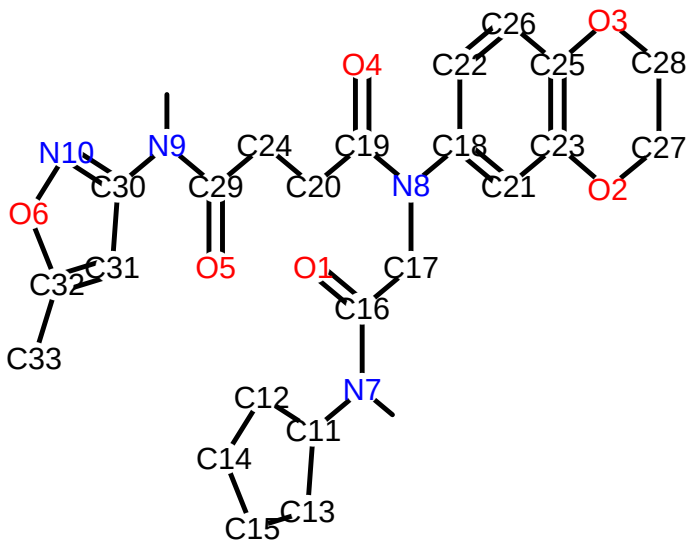
For visual inspection, the corresponding compound structure can be plotted with the ring bonds highlighted in color:

```
> atomindex <- as.numeric(gsub(".*_", "", unique(unlist(ringatoms))))
> plot(sdfset[1], print=FALSE, colbonds=atomindex)
```



Alternatively, one can include the atom numbers in the plot:

```
> plot(sdfset[1], print=FALSE, atomnum=TRUE, no_print_atoms="H")
```



Aromaticity information of the rings can be returned in a logical vector by setting `arom=TRUE`:

```
> rings(sdfset[1], upper=Inf, type="all", arom=TRUE, inner=FALSE)

$RINGS
$RINGS$ring1
[1] "N_10" "O_6"  "C_32" "C_31" "C_30"

$RINGS$ring2
[1] "C_12" "C_14" "C_15" "C_13" "C_11"

$RINGS$ring3
[1] "C_23" "O_2"  "C_27" "C_28" "O_3"  "C_25"

$RINGS$ring4
[1] "C_23" "C_21" "C_18" "C_22" "C_26" "C_25"

$RINGS$ring5
 [1] "O_3"  "C_28" "C_27" "O_2"  "C_23" "C_21" "C_18" "C_22" "C_26" "C_25"
```

```
$AROMATIC
ring1 ring2 ring3 ring4 ring5
 TRUE FALSE FALSE  TRUE FALSE
```

Return rings with no more than 6 atoms that are also aromatic:

```
> rings(sdfset[1], upper=6, type="arom", arom=TRUE, inner=FALSE)

$AROMATIC_RINGS
$AROMATIC_RINGS$ring1
[1] "N_10" "O_6"  "C_32" "C_31" "C_30"

$AROMATIC_RINGS$ring4
[1] "C_23" "C_21" "C_18" "C_22" "C_26" "C_25"
```

Count shortest possible rings and their aromaticity assignments by setting `type=count` and `inner=TRUE`. The inner (smallest possible) rings are identified by first computing all possible rings and then selecting only the inner rings. For more details, consult the help documentation with `?rings`.

```
> rings(sdfset[1:4], upper=Inf, type="count", arom=TRUE, inner=TRUE)

     RINGS AROMATIC
CMP1     4        2
CMP2     3        3
CMP3     4        2
CMP4     3        3
```

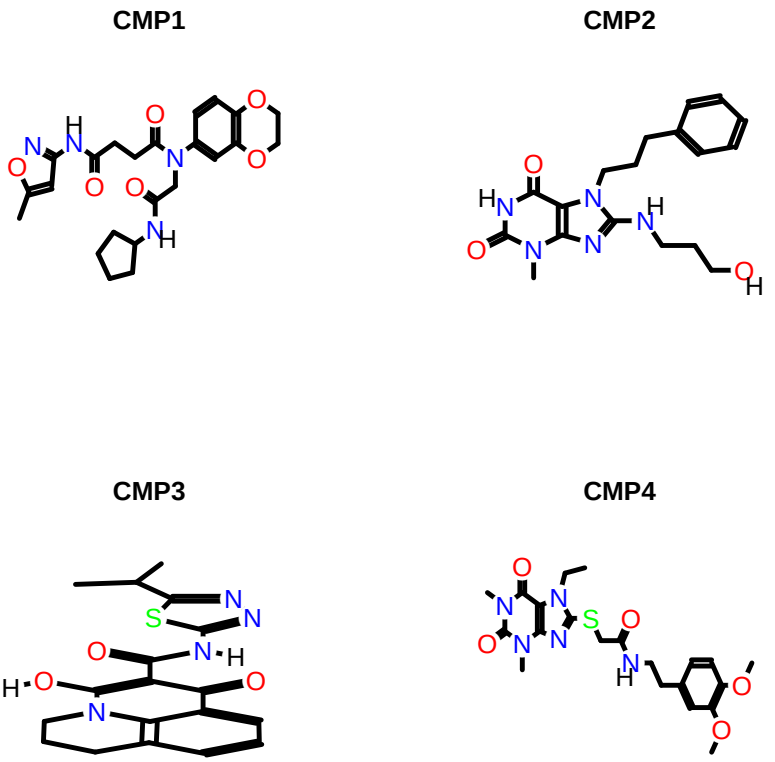
15 Rendering Chemical Structure Images

15.1 R Graphics Device

A new plotting function for compound structures has been added to the package recently. This function uses the native R graphics device for generating compound depictions. At this point this function is still in an experimental developmental stage but should become stable soon.

Plot compound Structures with R's graphics device:

```
> data(sdfsample)
> sdfset <- sdfsample
> plot(sdfset[1:4], print=FALSE) # 'print=TRUE' returns SDF summaries
```

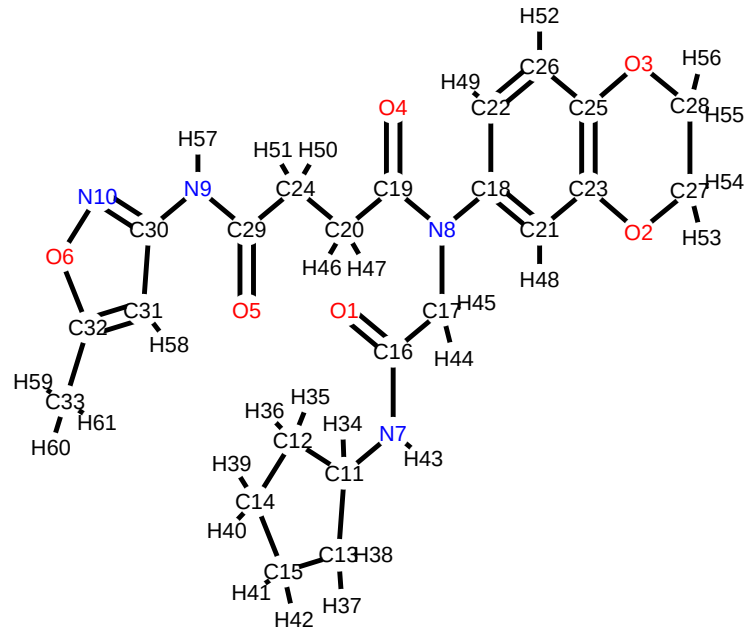


Customized plots:

```
> plot(sdfset[1:4], griddim=c(2,2), print_cid=letters[1:4], print=FALSE,
+      noHbonds=FALSE)
```

In the following plot, the atom block position numbers in the SDF are printed next to the atom symbols (`atomnum = TRUE`). For more details, consult help documentation with `?plotStruc` or `?plot`.

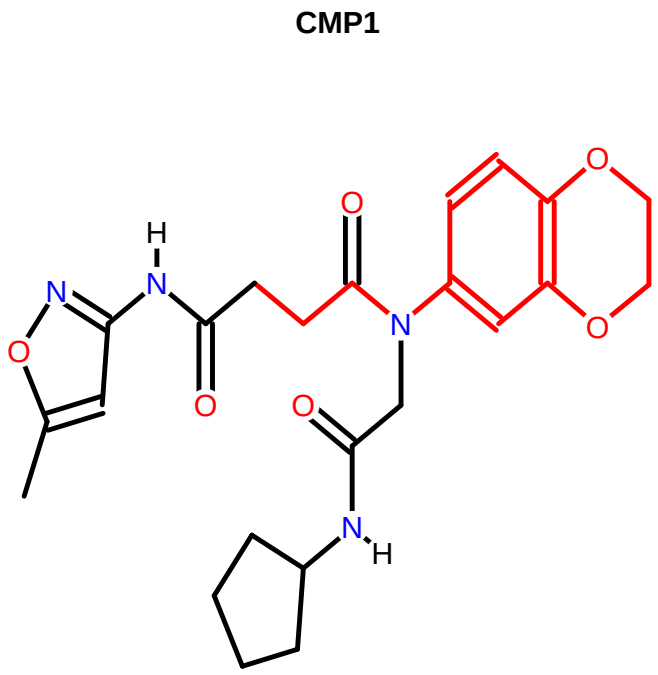
```
> plot(sdfset["CMP1"], atomnum = TRUE,  noHbonds=F , no_print_atoms = "",
+      atomcex=0.8, sub=paste("MW:", MW(sdfsample["CMP1"])), print=FALSE)
```



MW: 456.49162

Substructure highlighting by atom numbers:

```
> plot(sdfset[1], print=FALSE, colbonds=c(22,26,25,3,28,27,2,23,21,18,8,19,20,24))
```



15.2 Online with ChemMine Tools

Alternatively, one can visualize compound structures with a standard web browser using the online ChemMine Tools service. The service allows to display other information next to the structures using the extra argument of the `sdf.visualize` function. The following examples demonstrate, how one can plot and annotate structures by passing on extra data as vector of character strings, matrices or lists.

Plot structures using web service ChemMine Tools:

```
> sdf.visualize(sdfset[1:4])
```

Add extra annotation as *vector*:

```
> sdf.visualize(sdfset[1:4], extra=month.name[1:4])
```

Add extra annotation as *matrix*:

```
> extra <- apply(propma[1:4,], 1, function(x)
+          data.frame(Property=colnames(propma), Value=x))
> sdf.visualize(sdfset[1:4], extra=extra)
```

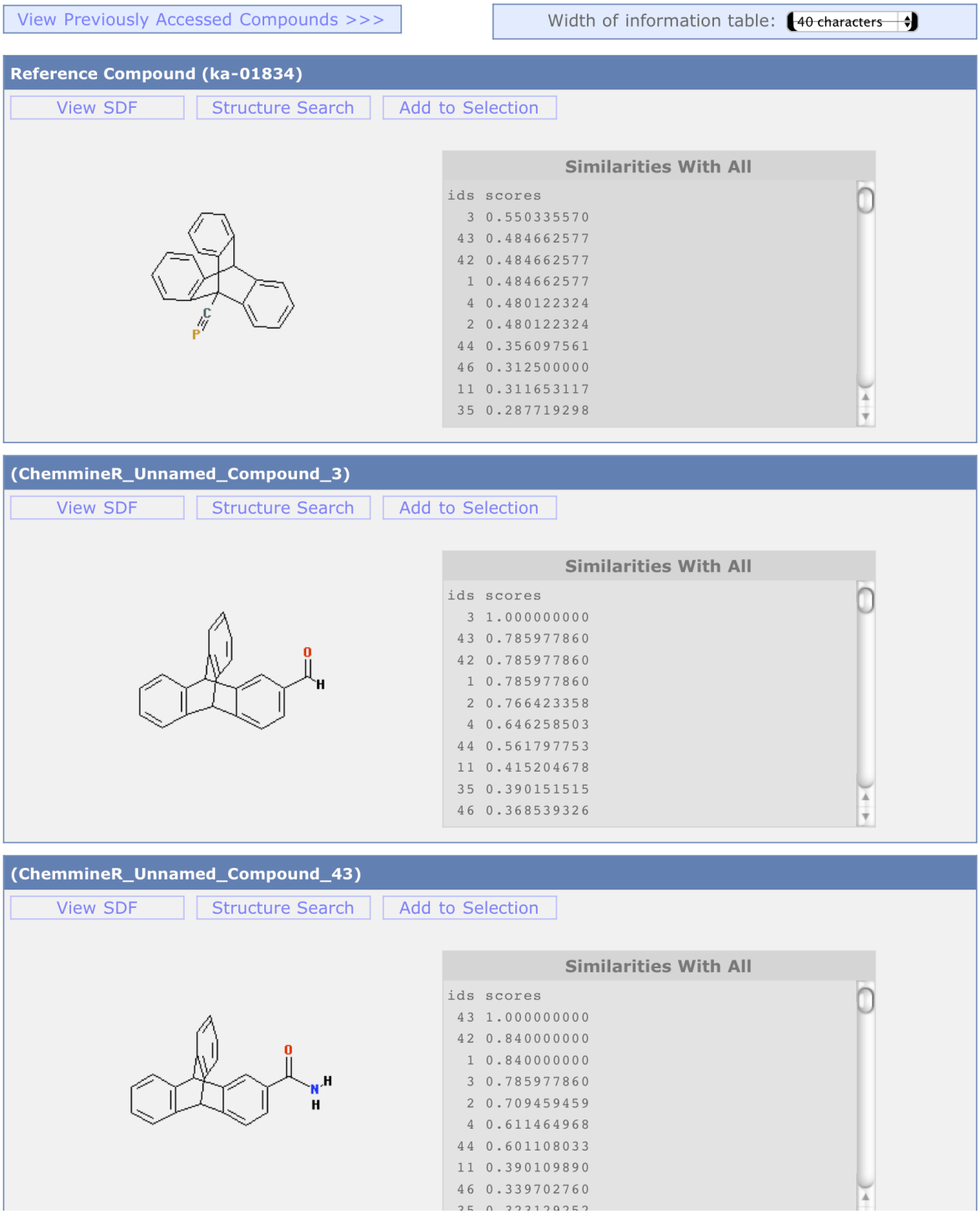


Figure 3: Visualization webpage created by calling `sdf.visualize`.

Add extra annotation as *list*:

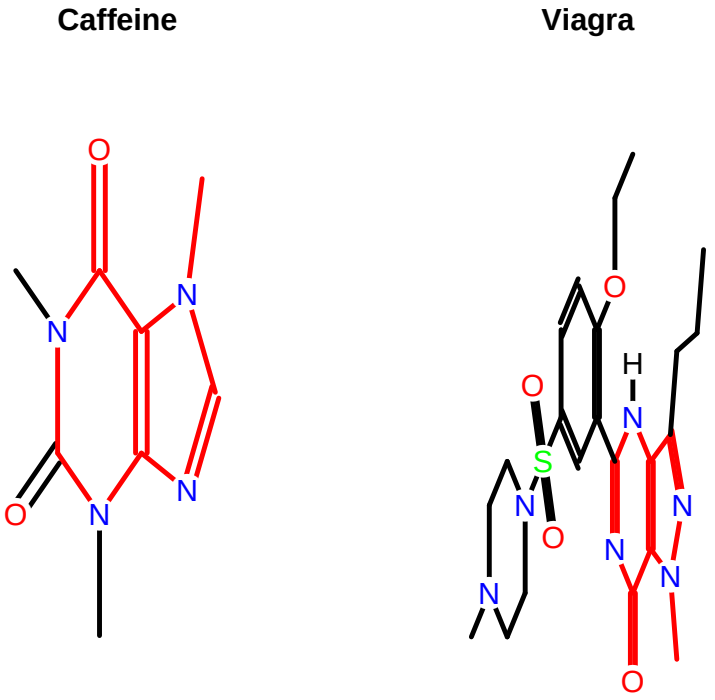
```
> sdf.visualize(sdfset[1:4], extra=bondblock(sdfset[1:4]))
```

16 Similarity Comparisons and Searching

16.1 Maximum Common Substructure (MCS) Searching

The ChemmineR add-on package fmcsR provides support for identifying maximum common substructures (MCSs) and flexible MCSs among compounds. The algorithm can be used for pairwise compound comparisons, structure similarity searching and clustering. The manual describing this functionality is available here and the associated publication is available in Wang et al. (2013). The following gives a short preview of some functionalities provided by the *fmcsR* package.

```
> library(fmcsR)
> data(fmcstest) # Loads test sdfset object
> test <- fmcs(fmcstest[1], fmcstest[2], au=2, bu=1) # Searches for MCS with mismatches
> plotMCS(test) # Plots both query compounds with MCS in color
```



16.2 AP/APset Classes for Storing Atom Pair Descriptors

The function `sdf2ap` computes atom pair descriptors for one or many compounds (Carhart et al., 1985; Chen and Reynolds, 2002). It returns a searchable atom pair database stored in a

container of class *APset*, which can be used for structural similarity searching and clustering. As similarity measure, the Tanimoto coefficient or related coefficients can be used. An *APset* object consists of one or many *AP* entries each storing the atom pairs of a single compound. Note: the deprecated `cmp.parse` function is still available which also generates atom pair descriptor databases, but directly from an SD file. Since the latter function is less flexible it may be discontinued in the future.

Generate atom pair descriptor database for searching:

```
> ap <- sdf2ap(sdfset[[1]]) # For single compound
> ap

An instance of "AP"
<<atom pairs>>
52614450304 52615497856 52615514112 52616547456 52616554624 ... length: 528

> apset <- sdf2ap(sdfset) # For many compounds.

> view(apset[1:4])

$`650001`
An instance of "AP"
<<atom pairs>>
53688190976 53688190977 53688190978 53688190979 53688190980 ... length: 528

$`650002`
An instance of "AP"
<<atom pairs>>
53688190976 53688190977 53688190978 53688190979 53689239552 ... length: 325

$`650003`
An instance of "AP"
<<atom pairs>>
52615496704 53688190976 53688190977 53689239552 53697627136 ... length: 325

$`650004`
An instance of "AP"
<<atom pairs>>
52617593856 52618642432 52619691008 52619691009 52628079616 ... length: 496
```

Return main components of APset objects:

```
> cid(apset[1:4]) # Compound IDs
> ap(apset[1:4]) # Atom pair descriptors
> db.explain(apset[1]) # Return atom pairs in human readable format
```

Coerce APset to other objects:

```
> apset2descdb(apset) # Returns old list-style AP database
> tmp <- as(apset, "list") # Returns list
> as(tmp, "APset") # Converts list back to APset
```

16.3 Large SDF and Atom Pair Databases

When working with large data sets it is often desirable to save the *SDFset* and *APset* containers as binary R objects to files for later use. This way they can be loaded very quickly into a new R session without recreating them every time from scratch.

Save and load of *SDFset* and *APset* containers:

```
> save(sdfset, file = "sdfset.rda", compress = TRUE)
> load("sdfset.rda")
> save(apset, file = "apset.rda", compress = TRUE)
> load("apset.rda")
```

16.4 Pairwise Compound Comparisons with Atom Pairs

The `cmp.similarity` function computes the atom pair similarity between two compounds using the Tanimoto coefficient as similarity measure. The coefficient is defined as $c/(a+b+c)$, which is the proportion of the atom pairs shared among two compounds divided by their union. The variable c is the number of atom pairs common in both compounds, while a and b are the numbers of their unique atom pairs.

```
> cmp.similarity(apset[1], apset[2])

[1] 0.2637037

> cmp.similarity(apset[1], apset[1])

[1] 1
```

16.5 Similarity Searching with Atom Pairs

The `cmp.search` function searches an atom pair database for compounds that are similar to a query compound. The following example returns a data frame where the rows are sorted by the Tanimoto similarity score (best to worst). The first column contains the indices of the matching compounds in the database. The argument cutoff can be a similarity cutoff, meaning only compounds with a similarity value larger than this cutoff will be returned; or it can be an integer value restricting how many compounds will be returned. When supplying a cutoff of 0, the function will return the similarity values for every compound in the database.

```
> cmp.search(apset, apset["650065"], type=3, cutoff = 0.3, quiet=TRUE)

  index    cid    scores
1    61 650066 1.0000000
2    60 650065 1.0000000
3    67 650072 0.3389831
4    11 650011 0.3190608
5    15 650015 0.3184524
6    86 650092 0.3154270
7    64 650069 0.3010279
```

Alternatively, the function can return the matches in form of an index or a named vector if the `type` argument is set to `1` or `2`, respectively.

```
> cmp.search(apset, apset["650065"], type=1, cutoff = 0.3, quiet=TRUE)

[1] 61 60 67 11 15 86 64

> cmp.search(apset, apset["650065"], type=2, cutoff = 0.3, quiet=TRUE)

   650066    650065    650072    650011    650015    650092    650069
1.0000000 1.0000000 0.3389831 0.3190608 0.3184524 0.3154270 0.3010279
```

16.6 FP/FPset Classes for Storing Fingerprints

The *FPset* class stores fingerprints of small molecules in a matrix-like representation where every molecule is encoded as a fingerprint of the same type and length. The *FPset* container acts as a searchable database that contains the fingerprints of many molecules. The *FP* container holds only one fingerprint. Several constructor and coerce methods are provided to populate *FP/FPset* containers with fingerprints, while supporting any type and length of fingerprints. For instance, the function `desc2fp` generates fingerprints from an atom pair database stored in an *APset*, and `as(matrix, "FPset")` and `as(character, "FPset")` construct an *FPset* database from objects where the fingerprints are represented as *matrix* or *character* objects, respectively.

Show slots of *FPset* class:

```
> showClass("FPset")

Class "FPset" [package "ChemmineR"]

Slots:

Name:    fpma
Class: matrix
```

Instance of *FPset* class:

```
> data(apset)
> (fpset <- desc2fp(apset))

An instance of a 1024 bit "FPset" with 100 molecules

> view(fpset[1:2])

$`650001`
An instance of "FP"
<<fingerprint>>
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ... length: 1024

$`650002`
An instance of "FP"
<<fingerprint>>
1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 0 0 1 1 ... length: 1024
```

FPset class usage:

```
> fpset[1:4] # behaves like a list

An instance of a 1024 bit "FPset" with 4 molecules

> fpset[[1]] # returns FP object

An instance of "FP"
<<fingerprint>>
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ... length: 1024

> length(fpset) # number of compounds

[1] 100

> cid(fpset) # returns compound ids

  [1] "650001" "650002" "650003" "650004" "650005" "650006" "650007" "650008"
  [9] "650009" "650010" "650011" "650012" "650013" "650014" "650015" "650016"
 [17] "650017" "650019" "650020" "650021" "650022" "650023" "650024" "650025"
 [25] "650026" "650027" "650028" "650029" "650030" "650031" "650032" "650033"
 [33] "650034" "650035" "650036" "650037" "650038" "650039" "650040" "650041"
 [41] "650042" "650043" "650044" "650045" "650046" "650047" "650048" "650049"
 [49] "650050" "650052" "650054" "650056" "650058" "650059" "650060" "650061"
 [57] "650062" "650063" "650064" "650065" "650066" "650067" "650068" "650069"
 [65] "650070" "650071" "650072" "650073" "650074" "650075" "650076" "650077"
 [73] "650078" "650079" "650080" "650081" "650082" "650083" "650085" "650086"
 [81] "650087" "650088" "650089" "650090" "650091" "650092" "650093" "650094"
 [89] "650095" "650096" "650097" "650098" "650099" "650100" "650101" "650102"
 [97] "650103" "650104" "650105" "650106"

> fpset[10] <- 0 # replacement of 10th fingerprint to all zeros
> cid(fpset) <- 1:length(fpset) # replaces compound ids
> c(fpset[1:4], fpset[11:14]) # concatenation of several FPset objects

An instance of a 1024 bit "FPset" with 8 molecules
```

Construct *FPset* class form *matrix*:

```
> fpma <- as.matrix(fpset) # coerces FPset to matrix
> as(fpma, "FPset")

An instance of a 1024 bit "FPset" with 100 molecules
```

Construct *FPset* class form *character vector*:

```
> fpchar <- as.character(fpset) # coerces FPset to character strings
> as(fpchar, "FPset") # construction of FPset class from character vector

An instance of a 1024 bit "FPset" with 100 molecules
```

Compound similarity searching with *FPset*:

```
> fpSim(fpset[1], fpset, method="Tanimoto", cutoff=0.4, top=4)

        1        96        67        15
1.0000000 0.4719101 0.4288499 0.4275229
```

16.7 Atom Pair Fingerprints

Atom pairs can be converted into binary atom pair fingerprints of fixed length. Computations on this compact data structure are more time and memory efficient than on their relatively complex atom pair counterparts. The function `desc2fp` generates fingerprints from descriptor vectors of variable length such as atom pairs stored in *APset* or *list* containers. The obtained fingerprints can be used for structure similarity comparisons, searching and clustering.

Create atom pair sample data set:

```
> data(sdfsample)
> sdfset <- sdfsample[1:10]
> apset <- sdf2ap(sdfset)
```

Compute atom pair fingerprint database using internal atom pair selection containing the 4096 most common atom pairs identified in DrugBank's compound collection. For details see *?apfp*. The following example uses from this set the 1024 most frequent atom pairs:

```
> fpset <- desc2fp(apset, descnames=1024, type="FPset")
```

Alternatively, one can provide any custom atom pair selection. Here, the 1024 most common ones in `apset`:

```
> fpset1024 <- names(rev(sort(table(unlist(as(apset, "list"))))))[1:1024])
> fpset <- desc2fp(apset, descnames=fpset1024, type="FPset")
```

A more compact way of storing fingerprints is as character values:

```
> fpchar <- desc2fp(x=apset, descnames=1024, type="character")
> fpchar <- as.character(fpset)
```

Converting a fingerprint database to a matrix and vice versa:

```
> fpma <- as.matrix(fpset)
> fpset <- as(fpma, "FPset")
```

Similarity searching and returning Tanimoto similarity coefficients:

```
> fpSim(fpset[1], fpset, method="Tanimoto")
```

Under `method` one can choose from several predefined similarity measures including `Tanimoto` (default), `Euclidean`, `Tversky` or `Dice`. Alternatively, one can pass on custom similarity functions.

```
> fpSim(fpset[1], fpset, method="Tversky", cutoff=0.4, top=4, alpha=0.5, beta=1)
```

Example for using a custom similarity function:

```
> myfct <- function(a, b, c, d) c/(a+b+c+d)
> fpSim(fpset[1], fpset, method=myfct)
```

Clustering example:

```
> simMAap <- sapply(cid(apfpset), function(x) fpSim(x=apfpset[x], apfpset, sorted=FALSE))
> hc <- hclust(as.dist(1-simMAap), method="single")
> plot(as.dendrogram(hc), edgePar=list(col=4, lwd=2), horiz=TRUE)
```

16.8 Pairwise Compound Comparisons with PubChem Fingerprints

The `fpSim` function computes the similarity coefficients (*e.g.* Tanimoto) for pairwise comparisons of binary fingerprints. For this data type, c is the number of "on-bits" common in both compounds, and a and b are the numbers of their unique "on-bits". Currently, the PubChem fingerprints need to be provided (here PubChem's SD files) and cannot be computed from scratch in *ChemmineR*. The PubChem fingerprint specifications can be loaded with `data(pubchemFPencoding)`.

Convert base 64 encoded PubChem fingerprints to *character* vector, *matrix* or *FPset* object:

```
> cid(sdfset) <- sdfid(sdfset)
> fpset <- fp2bit(sdfset, type=1)
> fpset <- fp2bit(sdfset, type=2)
> fpset <- fp2bit(sdfset, type=3)
> fpset

An instance of a 881 bit "FPset" with 100 molecules
```

Pairwise compound structure comparisons:

```
> fpSim(fpset[1], fpset[2])

   650002
0.5364807
```

16.9 Similarity Searching with PubChem Fingerprints

Similarly, the `fpSim` function provides search functionality for PubChem fingerprints:

```
> fpSim(fpset["650065"], fpset, method="Tanimoto", cutoff=0.6, top=6)

   650065    650066    650035    650019    650012    650046
1.0000000 0.9944751 0.7435897 0.7432432 0.7230047 0.7142857
```

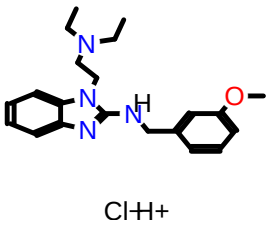
16.10 Visualize Similarity Search Results

The `cmp.search` function allows to visualize the chemical structures for the search results. Similar but more flexible chemical structure rendering functions are `plot` and `sdf.visualize` described above. By setting the visualize argument in `cmp.search` to `TRUE`, the matching compounds and their scores can be visualized with a standard web browser. Depending on the `visualize.browse` argument, an URL will be printed or a webpage will be opened showing the structures of the matching compounds along with their scores.

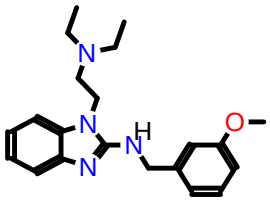
View similarity search results in R's graphics device:

```
> cid(sdfset) <- cid(apset) # Assure compound name consistency among objects.
> plot(sdfset[names(cmp.search(apset, apset["650065"], type=2, cutoff=4,
+      quiet=TRUE))], print=FALSE)
```

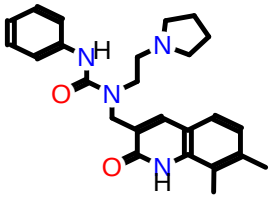
650065



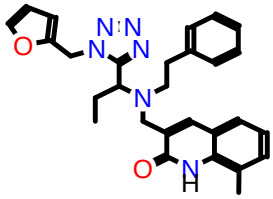
650066



650072



650011



View results online with Chemmine Tools:

```
> similarities <- cmp.search(apset, apset[1], type=3, cutoff = 10)
> sdf.visualize(sdfset[similarities[,1]], extra=similarities[,3])
```

17 Clustering

17.1 Clustering Identical or Very Similar Compounds

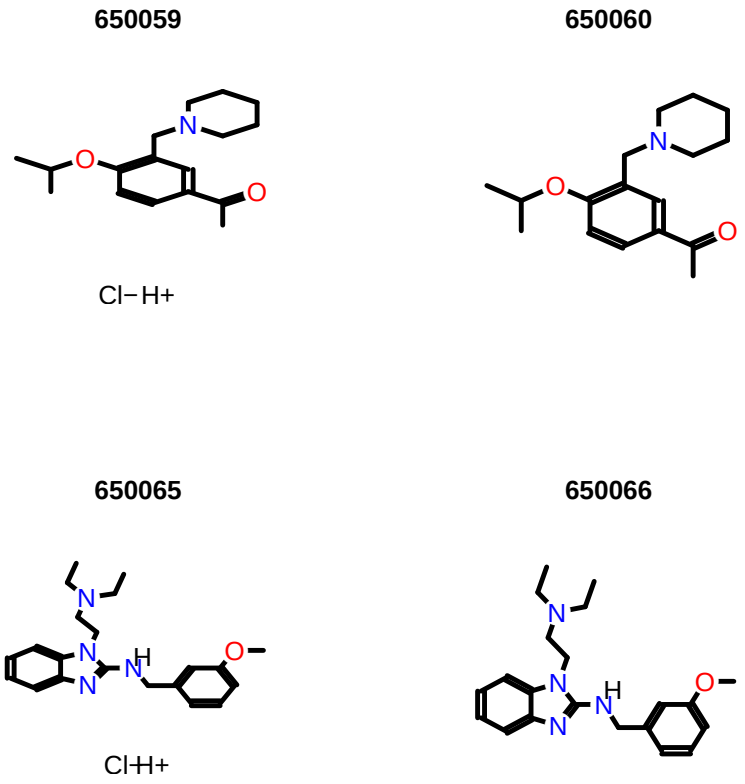
Often it is of interest to identify very similar or identical compounds in a compound set. The `cmp.duplicated` function can be used to quickly identify very similar compounds in atom pair sets, which will be frequently, but not necessarily, identical compounds.

Identify compounds with identical AP sets:

```
> cmp.duplicated(apset, type=1)[1:4] # Returns AP duplicates as logical vector

[1] FALSE FALSE FALSE FALSE

> cmp.duplicated(apset, type=2)[1:4,] # Returns AP duplicates as data frame

     ids CLSZ_100 CLID_100
1 650082        1        1
2 650059        2        2
3 650060        2        2
4 650010        1        3
```

Plot the structure of two pairs of duplicates:

```
> plot(sdfset[c("650059","650060", "650065", "650066")], print=FALSE)
```



Remove AP duplicates from SDFset and APset objects:

```
> apdups <- cmp.duplicated(apset, type=1)
> sdfset[which(!apdups)]; apset[which(!apdups)]

An instance of "SDFset" with 96 molecules

An instance of "APset" with 96 molecules
```

Alternatively, one can identify duplicates via other descriptor types if they are provided in the data block of an imported SD file. For instance, one can use here fingerprints, InChI, SMILES or other molecular representations. The following examples show how to enumerate by identical InChI strings, SMILES strings and molecular formula, respectively.

```
> count <- table(datablocktag(sdfset, tag="PUBCHEM_NIST_INCHI"))
> count <- table(datablocktag(sdfset, tag="PUBCHEM_OPENEYE_CAN_SMILES"))
> count <- table(datablocktag(sdfset, tag="PUBCHEM_MOLECULAR_FORMULA"))
> count[1:4]

 C10H9FN2O2S   C11H12N4OS    C11H13NO4 C12H11ClN2OS
           1            1            1            1
```

17.2 Binning Clustering

Compound libraries can be clustered into discrete similarity groups with the binning clustering function `cmp.cluster`. The function accepts as input an atom pair (`APset`) or a fingerprint (`FPset`) descriptor database as well as a similarity threshold. The binning clustering result is returned in form of a data frame. Single linkage is used for cluster joining. The function calculates the required compound-to-compound distance information on the fly, while a memory-intensive distance matrix is only created upon user request via the `save.distances` argument (see below).

Because an optimum similarity threshold is often not known, the `cmp.cluster` function can calculate cluster results for multiple cutoffs in one step with almost the same speed as for a single cutoff. This can be achieved by providing several cutoffs under the cutoff argument. The clustering results for the different cutoffs will be stored in one data frame.

One may force the `cmp.cluster` function to calculate and store the distance matrix by supplying a file name to the `save.distances` argument. The generated distance matrix can be loaded and passed on to many other clustering methods available in R, such as the hierarchical clustering function `hclust` (see below).

If a distance matrix is available, it may also be supplied to `cmp.cluster` via the `use.distances` argument. This is useful when one has a pre-computed distance matrix either from a previous call to `cmp.cluster` or from other distance calculation subroutines.

Single-linkage binning clustering with one or multiple cutoffs:

```
> clusters <- cmp.cluster(db=apset, cutoff = c(0.7, 0.8, 0.9), quiet = TRUE)

sorting result...

> clusters[1:12,]

      ids CLSZ_0.7 CLID_0.7 CLSZ_0.8 CLID_0.8 CLSZ_0.9 CLID_0.9
48 650049        2       48        2       48        2       48
49 650050        2       48        2       48        2       48
54 650059        2       54        2       54        2       54
55 650060        2       54        2       54        2       54
56 650061        2       56        2       56        2       56
57 650062        2       56        2       56        2       56
58 650063        2       58        2       58        2       58
59 650064        2       58        2       58        2       58
60 650065        2       60        2       60        2       60
61 650066        2       60        2       60        2       60
1  650001        1        1        1        1        1        1
2  650002        1        2        1        2        1        2
```

Clustering of `FPset` objects with multiple cutoffs. This method allows to call various similarity methods provided by the `fpSim` function. For details consult `?fpSim`.

```
> fpset <- desc2fp(apset)
> clusters2 <- cmp.cluster(fpset, cutoff=c(0.5, 0.7, 0.9), method="Tanimoto",
+                          quiet=TRUE)
```

```
sorting result...
```

```
> clusters2[1:12,]
```

```
      ids CLSZ_0.5 CLID_0.5 CLSZ_0.7 CLID_0.7 CLSZ_0.9 CLID_0.9
69 650074       14       11        2       69        1       69
79 650085       14       11        2       69        1       79
11 650011       14       11        1       11        1       11
15 650015       14       11        1       15        1       15
45 650046       14       11        1       45        1       45
47 650048       14       11        1       47        1       47
51 650054       14       11        1       51        1       51
53 650058       14       11        1       53        1       53
64 650069       14       11        1       64        1       64
65 650070       14       11        1       65        1       65
67 650072       14       11        1       67        1       67
86 650092       14       11        1       86        1       86
```

Sames as above, but using Tversky similarity measure:

```
> clusters3 <- cmp.cluster(fpset, cutoff=c(0.5, 0.7, 0.9), method="Tversky",
+                          alpha=0.3, beta=0.7, quiet=TRUE)
```

```
sorting result...
```

Return cluster size distributions for each cutoff:

```
> cluster.sizestat(clusters, cluster.result=1)
```

```
  cluster size count
1            1    90
2            2     5
```

```
> cluster.sizestat(clusters, cluster.result=2)
```

```
  cluster size count
1            1    90
2            2     5
```

```
> cluster.sizestat(clusters, cluster.result=3)
```

```
  cluster size count
1            1    90
2            2     5
```

Enforce calculation of distance matrix:

```
> clusters <- cmp.cluster(db=apset, cutoff = c(0.65, 0.5, 0.3),
+                         save.distances="distmat.rda")
>    # Saves distance matrix to file "distmat.rda" in current working directory.
> load("distmat.rda") # Loads distance matrix.
```

17.3 Jarvis-Patrick Clustering

The Jarvis-Patrick clustering algorithm is widely used in cheminformatics (Jarvis and Patrick, 1973). It requires a nearest neighbor table, which consists of `j` nearest neighbors for each item (*e.g.* compound). The nearest neighbor table is then used to join items into clusters when they meet the following requirements: (a) they are contained in each other's neighbor list and (b) they share at least `k` nearest neighbors. The values for `j` and `k` are user-defined parameters. The `jarvisPatrick` function implemented in *ChemmineR* takes a nearest neighbor table generated by `nearestNeighbors`, which works for *APset* and *FPset* objects. This function takes either the standard Jarvis-Patrick `j` parameter (as the `numNbrs` parameter), or else a `cutoff` value, which is an extension to the basic algorithm that we have added. Given a cutoff value, the nearest neighbor table returned contains every neighbor with a similarity greater than the cutoff value, for each item. This allows one to generate tighter clusters and to minimize certain limitations of this method, such as false joins of completely unrelated items when operating on small data sets. The `trimNeighbors` function can also be used to take an existing nearest neighbor table and remove all neighbors whose similarity value is below a given cutoff value. This allows one to compute a very relaxed nearest neighbor table initially, and then quickly try different refinements later.

In case an existing nearest neighbor matrix needs to be used, the `fromNNMatrix` function can be used to transform it into the list structure that `jarvisPatrick` requires. The input matrix must have a row for each compound, and each row should be the index values of the neighbors of compound represented by that row. The names of each compound can also be given through the `names` argument. If not given, it will attempt to use the `rownames` of the given matrix.

The `jarvisPatrick` function also allows one to relax some of the requirements of the algorithm through the `mode` parameter. When set to "a1a2b", then all requirements are used. If set to "a1b", then (a) is relaxed to a unidirectional requirement. Lastly, if `mode` is set to "b", then only requirement (b) is used, which means that all pairs of items will be checked to see if (b) is satisfied between them. The size of the clusters generated by the different methods increases in this order: "a1a2b" < "a1b" < "b". The run time of method "a1a2b" follows a close to linear relationship, while it is nearly quadratic for the much more exhaustive method "b". Only methods "a1a2b" and "a1b" are suitable for clustering very large data sets (e.g. >50,000 items) in a reasonable amount of time.

An additional extension to the algorithm is the ability to set the linkage mode. The `linkage` parameter can be one of "single", "average", or "complete", for single linkage, average linkage and complete linkage merge requirements, respectively. In the context of Jarvis-Patrick, average linkage means that at least half of the pairs between the clusters under consideration must meet requirement (b). Similarly, for complete linkage, all pairs must requirement (b). Single linkage is the normal case for Jarvis-Patrick and just means that at least one pair must meet requirement (b).

The output is a cluster *vector* with the item labels in the name slot and the cluster IDs in the data slot. There is a utility function called `byCluster`, which takes out cluster vector output by `jarvisPatrick` and transforms it into a list of vectors. Each slot of the list is named with a cluster id and the vector contains the cluster members. By default the function excludes singletons from the output, but they can be included by setting `excludeSingletons`=FALSE. Load/create sample *APset* and *FPset*:

```
> data(apset)
```

```
> fpset <- desc2fp(apset)
```

Standard Jarvis-Patrick clustering on *APset* and *FPset* objects:

```
> jarvisPatrick(nearestNeighbors(apset,numNbrs=6), k=5, mode="a1a2b") #Using "APset"

650001 650002 650003 650004 650005 650006 650007 650008 650009 650010 650011
     1      2      3      4      5      6      7      8      9     10     11
650012 650013 650014 650015 650016 650017 650019 650020 650021 650022 650023
    12     13     14     11     15     16     17     18     19     20     21
650024 650025 650026 650027 650028 650029 650030 650031 650032 650033 650034
    22     23     24     25     26     27     28     29     30     31     32
650035 650036 650037 650038 650039 650040 650041 650042 650043 650044 650045
    33     34     35     36     37     38     39     40     41     42     43
650046 650047 650048 650049 650050 650052 650054 650056 650058 650059 650060
    44     45     46     47     48     49     50     51     52     53     54
650061 650062 650063 650064 650065 650066 650067 650068 650069 650070 650071
    55     56     57     58     59     60     61     62     63     64     65
650072 650073 650074 650075 650076 650077 650078 650079 650080 650081 650082
    66     67     68     69     70     71     72     73     74     75     76
650083 650085 650086 650087 650088 650089 650090 650091 650092 650093 650094
    77     78     79     80     81     82     83     84     85     86     87
650095 650096 650097 650098 650099 650100 650101 650102 650103 650104 650105
    88     89     90     91     92     93     94     95     96     97     98
650106
    99

> jarvisPatrick(nearestNeighbors(fpset,numNbrs=6), k=5, mode="a1a2b") #Using "FPset"

650001 650002 650003 650004 650005 650006 650007 650008 650009 650010 650011
     1      2      3      4      5      6      7      8      9     10     11
650012 650013 650014 650015 650016 650017 650019 650020 650021 650022 650023
    12     13     14     11     15     16     17     18     19     20     21
650024 650025 650026 650027 650028 650029 650030 650031 650032 650033 650034
    22     23     24     25     26     27     28     29     30     31     32
650035 650036 650037 650038 650039 650040 650041 650042 650043 650044 650045
    33     34     35     36     37     38     39     40     41     42     43
650046 650047 650048 650049 650050 650052 650054 650056 650058 650059 650060
    44     45     46     47     48     49     50     51     52     53     54
650061 650062 650063 650064 650065 650066 650067 650068 650069 650070 650071
    55     56     57     58     59     60     61     62     63     64     65
650072 650073 650074 650075 650076 650077 650078 650079 650080 650081 650082
    66     67     68     69     70     71     72     73     74     75     76
650083 650085 650086 650087 650088 650089 650090 650091 650092 650093 650094
    77     78     79     80     81     82     83     84     85     86     87
650095 650096 650097 650098 650099 650100 650101 650102 650103 650104 650105
    88     89     90     91     92     93     94      1     95     96     97
650106
    98
```

The following example runs Jarvis-Patrick clustering with a minimum similarity `cutoff` value (here Tanimoto coefficient). In addition, it uses the much more exhaustive `"b"` method that generates larger cluster sizes, but significantly increased the run time. For more details, consult the corresponding help file with `?jarvisPatrick`.

```
> cl=jarvisPatrick(nearestNeighbors(fpset,cutoff=0.6, method="Tanimoto"), k=2 ,mode="b")
> byCluster(cl)

$`11`
[1] "650011" "650092"

$`15`
[1] "650015" "650069"

$`45`
[1] "650046" "650054"

$`48`
[1] "650049" "650050"

$`52`
[1] "650059" "650060"

$`53`
[1] "650061" "650062"

$`54`
[1] "650063" "650064"

$`55`
[1] "650065" "650066"

$`62`
[1] "650074" "650085"
```

Output nearest neighbor table (*matrix*):

```
> nnm <- nearestNeighbors(fpset,numNbrs=6)
> nnm$names[1:4]

[1] "650001" "650002" "650003" "650004"

> nnm$ids[1:4,]

NULL

> nnm$similarities[1:4,]
```

```
    650001    650102    650072    650015    650094    650069
sim      1 0.4719101 0.4288499 0.4275229 0.4247423 0.4187380
sim      1 0.4343891 0.4246575 0.4216867 0.3939394 0.3922078
sim      1 0.4152249 0.3619303 0.3610315 0.3424242 0.3367089
sim      1 0.5791045 0.4973958 0.4192708 0.4166667 0.4104683
```

Trim nearest neighbor table:

```
> nnm <- trimNeighbors(nnm,cutoff=0.4)
> nnm$similarities[1:4,]

    650001    650102    650072    650015    650094    650069
sim      1 0.4719101 0.4288499 0.4275229 0.4247423 0.4187380
sim      1 0.4343891 0.4246575 0.4216867        NA        NA
sim      1 0.4152249        NA        NA        NA        NA
sim      1 0.5791045 0.4973958 0.4192708 0.4166667 0.4104683

>
```

Perform clustering on precomputed nearest neighbor table:

```
> jarvisPatrick(nnm, k=5,mode="b")

650001 650002 650003 650004 650005 650006 650007 650008 650009 650010 650011
     1      2      3      4      5      6      7      8      9     10     11
650012 650013 650014 650015 650016 650017 650019 650020 650021 650022 650023
    12     13     14     11     15     16     17     18     19     20     21
650024 650025 650026 650027 650028 650029 650030 650031 650032 650033 650034
    22     23     24     25     26     27     28     29     30     31     32
650035 650036 650037 650038 650039 650040 650041 650042 650043 650044 650045
    33     34     35     36     37     38     39     40     41     42     43
650046 650047 650048 650049 650050 650052 650054 650056 650058 650059 650060
    11     44     11     45     46     47     48     49     50     51     52
650061 650062 650063 650064 650065 650066 650067 650068 650069 650070 650071
    53     54     55     56     57     57     58     59     11     60     61
650072 650073 650074 650075 650076 650077 650078 650079 650080 650081 650082
    62     63     64     65     66     67     68     69     37     70     71
650083 650085 650086 650087 650088 650089 650090 650091 650092 650093 650094
    72     64     73     74     75     76     77     78     11     79     80
650095 650096 650097 650098 650099 650100 650101 650102 650103 650104 650105
    81     82     83     84     85     86     87      1     88     89     90
650106
    91
```

Using a user defined nearest neighbor matrix:

```
> nn = matrix(c(1,2,2,1),2,2,dimnames=list(c('one','two')))
> nn
```

```
    [,1] [,2]
one    1    2
two    2    1
```

```
> byCluster(jarvisPatrick(fromNNMatrix(nn),k=1))
```

```
$`1`
[1] "one" "two"
```

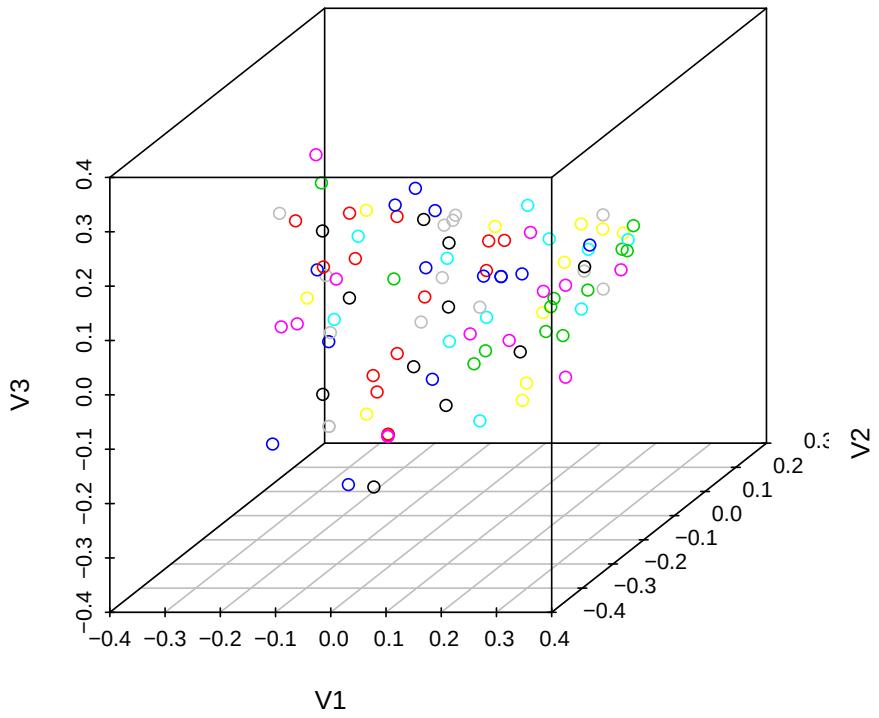
17.4 Multi-Dimensional Scaling (MDS)

To visualize and compare clustering results, the `cluster.visualize` function can be used. The function performs Multi-Dimensional Scaling (MDS) and visualizes the results in form of a scatter plot. It requires as input an *APset*, a clustering result from `cmp.cluster`, and a cutoff for the minimum cluster size to consider in the plot. To help determining a proper cutoff size, the `cluster.sizestat` function is provided to generate cluster size statistics.
MDS clustering and scatter plot:

```
> cluster.visualize(apset, clusters, size.cutoff=2, quiet = TRUE)
>    # Color codes clusters with at least two members.
> cluster.visualize(apset, clusters, quiet = TRUE) # Plots all items.
```

Create a 3D scatter plot of MDS result:

```
> library(scatterplot3d)
> coord <- cluster.visualize(apset, clusters, size.cutoff=1, dimensions=3, quiet=TRUE)
> scatterplot3d(coord)
```



Interactive 3D scatter plot with Open GL (graphics not evaluated here):

```
> library(rgl)
> rgl.open(); offset <- 50; par3d(windowRect=c(offset, offset, 640+offset, 640+offset))
> rm(offset); rgl.clear(); rgl.viewpoint(theta=45, phi=30, fov=60, zoom=1)
> spheres3d(coord[,1], coord[,2], coord[,3], radius=0.03, color=coord[,4], alpha=1,
+ shininess=20); aspect3d(1, 1, 1)
> axes3d(col='black'); title3d("", "", "", "", "", col='black'); bg3d("white")
>    # To save a snapshot of the graph, one can use the command rgl.snapshot("test.png").
```

17.5 Clustering with Other Algorithms

ChemmineR allows the user to take advantage of the wide spectrum of clustering utilities available in R. An example on how to perform hierarchical clustering with the hclust function is given below.

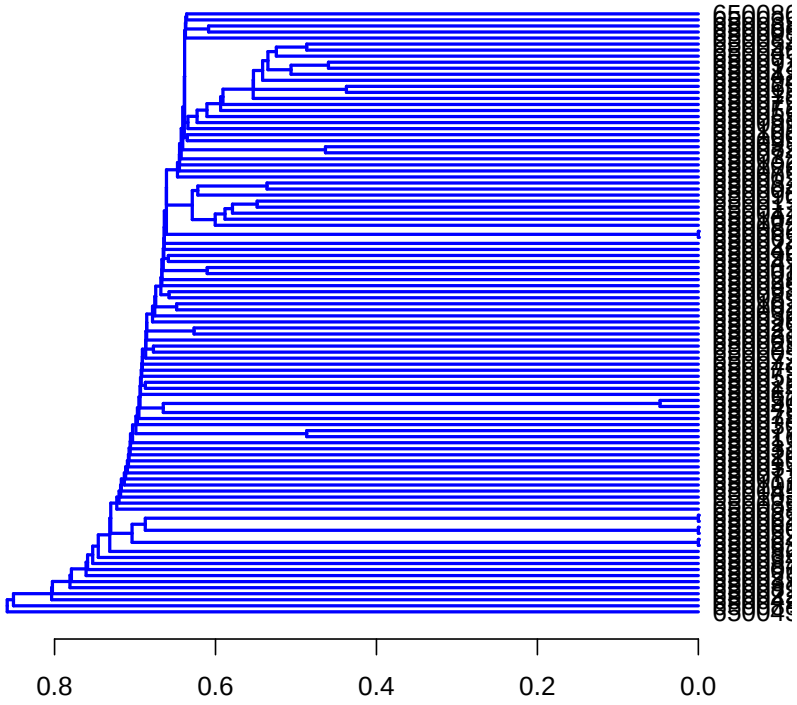
Create atom pair distance matrix:

```
> dummy <- cmp.cluster(db=apset, cutoff=0, save.distances="distmat.rda", quiet=TRUE)

sorting result...
```

```
> load("distmat.rda")
```

Hierarchical clustering with `hclust`:

```
> hc <- hclust(as.dist(distmat), method="single")
> hc[["labels"]] <- cid(apset) # Assign correct item labels
> plot(as.dendrogram(hc), edgePar=list(col=4, lwd=2), horiz=T)
```

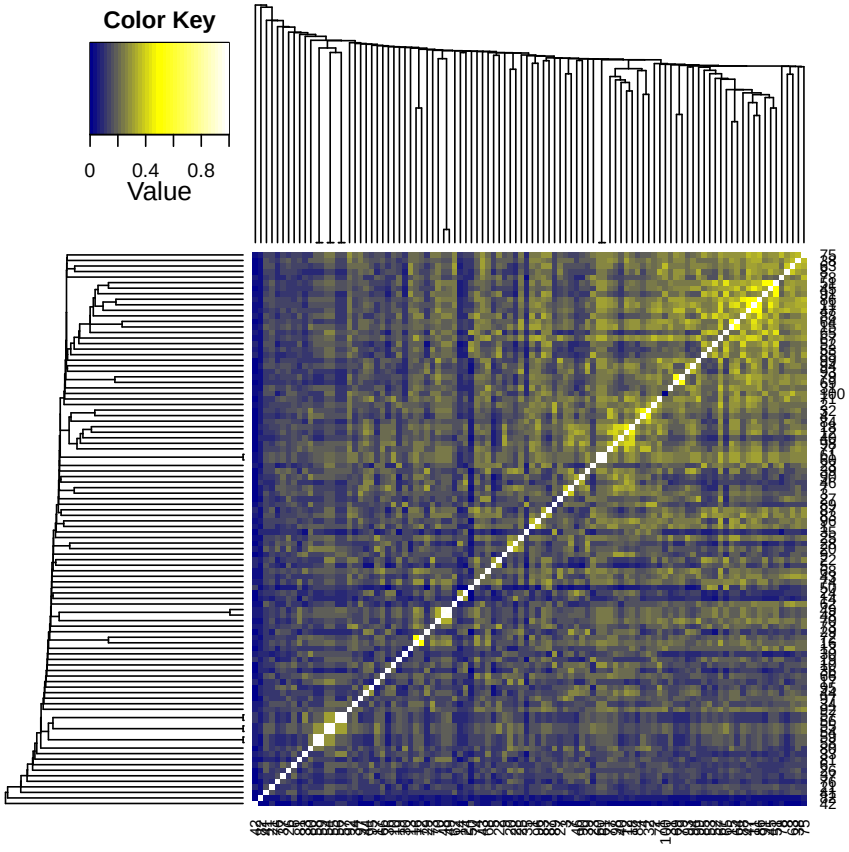


Instead of atom pairs one can use PubChem's fingerprints for clustering:

```
> simMA <- sapply(cid(fpset), function(x) fpSim(fpset[x], fpset, sorted=FALSE))
> hc <- hclust(as.dist(1-simMA), method="single")
> plot(as.dendrogram(hc), edgePar=list(col=4, lwd=2), horiz=TRUE)
```

Plot dendrogram with heatmap (here similarity matrix):

```
> library(gplots)
> heatmap.2(1-distmat, Rowv=as.dendrogram(hc), Colv=as.dendrogram(hc),
+           col=colorpanel(40, "darkblue", "yellow", "white"),
+           density.info="none", trace="none")
```


Color Key
0
0.4
0.8
Value

18 Searching PubChem

18.1 Get Compounds from PubChem by Id

The function `getIds` accepts one or more numeric PubChem compound ids and downloads the corresponding compounds from PubChem Power User Gateway (PUG) returning results in an *SDFset* container. The ChemMine Tools web service is used as an intermediate, to translate queries from plain HTTP POST to a PUG SOAP query.

Fetch 2 compounds from PubChem:

```
> compounds <- getIds(c(111,123))
> compounds
```

18.2 Search a SMILES Query in PubChem

The function `searchString` accepts one SMILES string (Simplified Molecular Input Line Entry Specification) and performs a >0.95 similarity PubChem fingerprint search, returning the hits in an *SDFset* container. The ChemMine Tools web service is used as an intermediate, to translate queries from plain HTTP POST to a PubChem Power User Gateway (PUG) query.

Search a SMILES string on PubChem:

```
> compounds <- searchString("CC(=O)OC1=CC=CC=C1C(=O)O")
> compounds
```

18.3 Search an SDF Query in PubChem

The function `searchSim` performs a PubChem similarity search just like `searchString`, but accepts a query in an *SDFset* container. If the query contains more than one compound, only the first is searched.

Search an *SDFset* container on PubChem:

```
> data(sdfsample); sdfset <- sdfsample[1]
> compounds <- searchSim(sdfset)
> compounds
```

19 Format Interconversions

The `sdf2smiles` and `smiles2sdf` functions provide format interconversion between SMILES strings (Simplified Molecular Input Line Entry Specification) and *SDFset* containers. Currently these functions are limited to a single compound at a time.

Convert an *SDFset* container to a SMILES *character* string:

```
> data(sdfsample); sdfset <- sdfsample[1]
> smiles <- sdf2smiles(sdfset)
> smiles
```

Convert a SMILES *character* string to an *SDFset* container:

```
> sdf <- smiles2sdf("CC(=O)OC1=CC=CC=C1C(=O)O\tname")
> view(sdf)
```

These functions require internet connectivity, as they rely on the ChemMine Tools web service for conversion with the Open Babel Open Source Chemistry Toolbox.

20 Version Information

```
> sessionInfo()

R version 3.0.1 (2013-05-16)
Platform: i386-w64-mingw32/i386 (32-bit)

locale:
[1] LC_COLLATE=C
[2] LC_CTYPE=English_United States.1252
[3] LC_MONETARY=English_United States.1252
[4] LC_NUMERIC=C
[5] LC_TIME=English_United States.1252

attached base packages:
[1] grid      stats     graphics  grDevices utils     datasets  methods
[8] base

other attached packages:
 [1] gplots_2.11.3        MASS_7.3-28          KernSmooth_2.23-10
 [4] caTools_1.14         gdata_2.13.2         gtools_3.0.0
 [7] scatterplot3d_0.3-33 fmcsR_1.2.1          RSQLite_0.11.4
[10] DBI_0.2-7            ChemmineR_2.12.3

loaded via a namespace (and not attached):
[1] RCurl_1.95-4.1 bitops_1.0-6   digest_0.6.3   tools_3.0.1
```

21 Funding

This software was developed with funding from the National Science Foundation: ABI-0957099, 2010-0520325 and IGERT-0504249.

References

T W Backman, Y Cao, and T Girke. ChemMine tools: an online service for analyzing and clustering small molecules. *Nucleic Acids Res*, 39(Web Server issue):486–491, Jul 2011. doi: 10.1093/nar/gkr320. URL `http://www.hubmed.org/display.cgi?uids=21576229`.

Y Cao, A Charisi, L C Cheng, T Jiang, and T Girke. ChemmineR: a compound mining framework for R. *Bioinformatics*, 24(15):1733–1734, Aug 2008. doi: 10.1093/bioinformatics/btn307. URL `http://www.hubmed.org/display.cgi?uids=18596077`.

R.E. Carhart, D.H. Smith, and R. Venkataraghavan. Atom pairs as molecular features in structure-activity studies: definition and applications. *Journal of Chemical Information and Computer Sciences*, 25(2):64–73, 1985.

X. Chen and C.H. Reynolds. Performance of Similarity Measures in 2D Fragment-Based Similarity Searching: Comparison of Structural Descriptors and Similarity Coefficients. *Journal of Chemical Information and Computer Sciences*, 42(6):1407–1414, 2002.

Th. Hanser, Ph. Jauffret, and G. Kaufmann. A new algorithm for exhaustive ring perception in a molecular graph. *Journal of Chemical Information and Computer Sciences*, 36(6):1146–1152, 1996. doi: 10.1021/ci960322f. URL `http://pubs.acs.org/doi/abs/10.1021/ci960322f`.

R.A. Jarvis and E.A. Patrick. Clustering using a similarity measure based on shared near neighbors. *IEEE Transactions on Computers*, 22(11):1025–1034, 1973. URL `http://davide.eynard.it/teaching/2012_PAMI/JP.pdf`.

Y Wang, T W Backman, K Horan, and T Girke. fmcsR: Mismatch Tolerant Maximum Common Substructure Searching in R. *Bioinformatics*, Aug 2013. doi: 10.1093/bioinformatics/btt475. URL `http://www.hubmed.org/display.cgi?uids=23962615`.