Creating In Silico Interactomes

Tony Chiang

April 10, 2009

1 Introduction

We explore the problem of estimating *in silico* interactomes: a collection of protein complexes for some cell or tissue under certain specific conditions. We will focus our attentions on Saccharomyces cerevisiae though this need not be the case in general.

This document will detail the various functions of ScISI by creating a running example for an *in silico* interactome. It is recommended that the reader has a working session of R and investigates the examples as they arise.

```
> library("ScISI")
```

2 Obtaining Information

2.1 Gene Ontology

Before an *in silico* interactome can be assembled, we must obtain the necessary information. As remarked earlier, two sources are online databases, Gene Ontology (GO) and the Munich Information Center for Protein Sequences (MIPS), and the third is a catalogue of apComplex estimates from a number of small and large wet-lab experiments. We start by discussing how information is parsed from the GO and MIPS databases.

Parsing through the GO database to collect the required information is done by calling the `getGOInfo` function:

```
> goECodes = c("IEA", "NAS", "ND", "NR")
> go = getGOInfo(wantDefault = TRUE, toGrep = NULL, parseType = NULL,
+     eCode = goECodes, wantAllComplexes = TRUE)
```

All the information that is obtained from the `getGOInfo` function comes from the R packages GO and GOstats. It is important to note that the Gene Ontology repository has periodic updates, so the protein complexes obtained will reflect the version of GO you are using. `getGOInfo` essentially parses through the Cellular Component (CC) of GO and (a)greps for specified key terms within certain boundary conditions. The parsing is carried out on terms (i.e. character vectors) together with perl regular expressions (this accounts for a boundary condition on protein complexes). The parameters of `getGOInfo` have the following characteristics:

1 `wantDefault`: A logical: if passed into the function as TRUE, three default terms will be parsed: complex by agrep, "`\\Base\\b`" by grep, and "`\\Bsome\\b`" by grep. The latter terms will search for protein complexes such as RNA polymerase or Ribosomes which may not contain the term complex.

2 `toGrep`: A named list with only one entry named "pattern" supplied. The pattern entry holds a character vector of terms, with perl regular expressions, supplied by the users. `getGOInfo` will parse for these terms within the CC ontology. If toGrep is not NULL, parseType must not be NULL. This argument lets the user append to the search terms if `wantDefault` is TRUE, or it allows the user to set the search terms if FALSE.

3 `parseType`: A character vector of parse terms, e.g. "grep" or "agrep". This vector must either be of length one or of the same length as the vector contained in `toGrep`. If parseType is of length one, all entries of toGrep will be parsed uniformly. Otherwise, the i^{th} entry from parseType defines the command for the i^{th} entry from toGrep.

To illustrate one example of appending to the search defaults, we have supplied the suffix $-somal$ as a candidate for `toBeParsed`:

```
> toBeParsed = list()
> toBeParsed$pattern = "\\Bsomal\\b"
> goGrep = getGOInfo(wantDefault = TRUE, toGrep = toBeParsed, parseType = "grep",
+     eCode = NULL, wantAllComplexes = TRUE)
> getGOTerm(setdiff(names(goGrep), names(go)))

$CC
                              GO:0000407                                GO:0005762
         "pre-autophagosomal structure" "mitochondrial large ribosomal subunit"
                              GO:0005763                                GO:0005778
"mitochondrial small ribosomal subunit"                "peroxisomal membrane"
                              GO:0005779                                GO:0005782
     "integral to peroxisomal membrane"                  "peroxisomal matrix"
                              GO:0015934                                GO:0015935
               "large ribosomal subunit"              "small ribosomal subunit"
                              GO:0022625                                GO:0022627
    "cytosolic large ribosomal subunit"    "cytosolic small ribosomal subunit"
                              GO:0034045                                GO:0044427
"pre-autophagosomal structure membrane"                    "chromosomal part"
```

Because `wantDefault` is still kept as $TRUE$, the three default search terms were also parsed. The last line of code displays those GO protein complexes found with the addition of the term $-somal$ which were not found with only the default search terms.

4 `eCode`: A character vector of evidence code terms (cite here). The is another search restriction; whereas perl regular expression put restrictions on the protein complexes as a whole, the evidence codes operate on the individual proteins themselves. If a protein is only referenced by evidence codes found within the eCode vector, it is dis-allowed and removed from the protein complex. In the example above, we have chosen the evidence codes IEA (Inferred from Electronic Annotation), NAS (Non-traceable Author Statement), ND (No biological Data available), and NR (Not Recorded).

5 `wantAllComplexes`: A logical. If TRUE, the children of parsed nodes will also be returned (accounting for sub-complexes). In addition, nodes that are children of the node "GO:0043234" (the protein complex node) are returned as well.

```
> class(go)

[1] "list"
```

```
> names(go)[1:5]

[1] "GO:0000015" "GO:0000110" "GO:0000111" "GO:0000112" "GO:0000113"

> go$"GO:0005830"

NULL
```

The return value of `getGOInfo` is a named list (the code and actual output is shown above). The names of this list corresond to the GO ID's of the protein complexes selected, and the entries of the list correspond to the yeast genes (using their systematic names) annotated to the repsective GO ID (node). The list is one way to represent the hyper-graph. Each entry of the return value represents a particular hyper-edge, and the union over all the entries of the list (i.e. over all the hyper-edges) yields the vertex set.

The incidence matrix, M, of a bi-partite graph has its rows indexed by proteins and has its columns indexed by the protein complexes. $M_{i,j} = 1$ if and only if protein i is a member of complex j, and $M_{i,j} = 0$ otherwise. Therefore, M is a $\{0, 1\}$ matrix which is generally sparse.

The function `createGOMatrix` takes the output of `getGOInfo` and creates the bi-partite graph incidence matrix of this hyper-graph list.

```
> goM = list2Matrix(go)
```

We remark that we the proteins names used to index the rows of the incidence matrix are the systematic gene names for yeast. To establish uniformity, we have used the systematic gene names across all data sets. The columns of the incidence matrix are indexed by the GO id's corresponding to certain protein complexes.

Unfortunately, term mining is rarely comprehensive nor exhaustive. There exists GO nodes that neither have the term *complex* within the Go terms nor do they have any terms ending in the suffixes $-ase$ nor $-some$. We have manually found additionally GO nodes of potential interest. The `xtraGO` data set is a character vector of these hand selected GO protein complexes. There are 32 total extra complexes though 6 of these complexes were already obtained by the `getGOInfo` function. From the remaining 26 complexes, only 6 complexes were found to have non-trivial overlap with S. cerevisae in the GO CC ontology. To check possible GO nodes and to add to the GO protein complexes, we call the `xtraGONodes` function (example code shown below). It is important to note that the `xtraGONodes` can only be called after an initial incidence matrix for the bi-partite graph of the GO protein complexes has been built since this function will compare the extra GO nodes with those existing GO protein complexes.

2.2 Munich Information Center for Protein Sequences

Parsing for information from the MIPS database (http://mips.gsf.de/genre/proj/yeast/) is completely analogous to parsing the GO database and is done by calling the `getMipsInfo` function:

```
> mips = getMipsInfo(wantDefault = TRUE, toGrep = NULL, parseType = NULL,
+     eCode = NULL, wantSubComplexes = TRUE, ht = FALSE)
```

Unlike the `getGOInfo` function, `getMipsInfo` parse through three files downloaded from the complex catalogue of the MIPS database. One file contains a hierarchy for the protein complexes (i.e. complexes are built from sub-complexes) while the second file contains the protein composition of each of the levels of the hierarchy from the first. Finally, the third file contains evidence codes which are used to annotate proteins to each protein complex. These files are updated bi-annually, so it is important to note that `getMipsInfo` should be updated periodically to correspond with the newer data files. With the exception of the `wantSubComplexes` argument, the functionality of `getMipsInfo` is identical to that of `getGOInfo` though the implementation is quite different. Thus, the parameters of `getMipsInfo` have the following characteristics:

1 `wantDefault`: A logical: if passed into the function as TRUE, three default terms will be parsed: complex by grep, "`\\Base\\b`" by grep, and "`\\Bsome\\b`" by grep.

2 `toGrep`: A named list with only one entry named "pattern". The entry is a character vector containing terms, with perl regular expressions, supplied by the users. `getMipsInfo` will parse for these terms within the hierarchy. If toGrep is not NULL, parseType must not be NULL neither. The terms supplied will be in addition to those of the default if `wantDefault` is TRUE or will be the only terms parsed if FALSE.

3 `parseType`: A character vector of parse terms, e.g. "grep" or "agrep". This vector must either be of length one or of the same length as the vector toGrep. If parseType is of length one, all entries of toGrep will be parsed uniformly. Otherwise, the i^{th} entry from parseType defines the command for the i^{th} entry from toGrep.

   Exactly like our in example with the `getGOInfo` function, we append to the default search terms with the suffix $-somal$.

```
> toBeParsed = list()
> toBeParsed$pattern = "\\Bsomal\\b"
> mipsGrep = getMipsInfo(wantDefault = TRUE, toGrep = toBeParsed,
+     parseType = "grep", eCode = NULL, wantSubComplexes = TRUE)
```

   Again the output of the code above shows the extra protein clusters obtained with the addition of $-somal$ to the three default search terms.

4 `eCode`: A character vector. This is a character vector of evidence codes. If a protein is only indexed by the evidence codes given in this vector, it is removed from the protein complex it was annotated using only these evidence codes. The default setting of this parameter is set to $NULL$ as the vast majority of the evidence codes pertain to the protein complex estimates obtained from Affinity Purification - Mass Spectrometry (AP-MS) experiments.

5 `wantSubComplexes`: A logical. If FALSE, only the top levels of the hierarchy will be returned (the aggregate protein complex); if TRUE, all the intermediates of the hierarchy will also be returned (the sub-complexes as well). There are certain computational instances when retaining only the top level protein complexes is necessary, but the default setting is to retain all protein complexes, and hence the logical is TRUE.

6 `ht`: A logical. If FALSE, then the protein complexes obtained by high throughput experimentation will not be extracted. Otherwise, if TRUE, those protein complexes obtained by high throughput analysis are extracted.

Another difference between the `getGOInfo` and `getMipsInfo` functions is their respective outputs. We have seen that the output from the `getGOInfo` is a named list that represents the hyper-graph, but, from looking at the output printed below, this is not the case for `getMipsInfo`.

The output of `getMipsInfo` is a list with two entries:

```
> class(mips)

[1] "list"

> names(mips)

  [1] "MIPS-60"              "MIPS-90.10"           "MIPS-90.20"
  [4] "MIPS-90.30"           "MIPS-125.10.10"       "MIPS-130"
  [7] "MIPS-133.30"          "MIPS-133.40"          "MIPS-133.50"
```

```
 [10] "MIPS-140.30.30.20"     "MIPS-140.30.30.30"     "MIPS-160"
 [13] "MIPS-230.10"           "MIPS-230.20.10"        "MIPS-230.20.20"
 [16] "MIPS-230.20.40"        "MIPS-240.20"           "MIPS-260.20"
 [19] "MIPS-260.20.10"        "MIPS-260.20.20"        "MIPS-260.20.30"
 [22] "MIPS-260.20.40"        "MIPS-260.30.30"        "MIPS-260.30.30.10"
 [25] "MIPS-260.30.30.20"     "MIPS-260.40"           "MIPS-260.50.10.10"
 [28] "MIPS-260.60"           "MIPS-260.70"           "MIPS-260.80"
 [31] "MIPS-260.90"           "MIPS-260.100"          "MIPS-265"
 [34] "MIPS-270.10"           "MIPS-270.10.10"        "MIPS-270.20"
 [37] "MIPS-270.20.10"        "MIPS-270.20.20"        "MIPS-270.20.30"
 [40] "MIPS-270.20.40"        "MIPS-270.20.50"        "MIPS-290"
 [43] "MIPS-290.20.10"        "MIPS-290.20.20"        "MIPS-290.20.30"
 [46] "MIPS-295"              "MIPS-300"              "MIPS-310"
 [49] "MIPS-310.10"           "MIPS-310.20"           "MIPS-310.40"
 [52] "MIPS-320"              "MIPS-350.10.10"        "MIPS-400"
 [55] "MIPS-410.10"           "MIPS-410.20"           "MIPS-410.30"
 [58] "MIPS-410.35"           "MIPS-410.40.20"        "MIPS-410.40.30"
 [61] "MIPS-410.40.60"        "MIPS-420.20"           "MIPS-420.30"
 [64] "MIPS-420.40"           "MIPS-420.50"           "MIPS-440.12.10"
 [67] "MIPS-440.12.30"        "MIPS-440.30.10.10"     "MIPS-440.30.10.20"
 [70] "MIPS-445.10"           "MIPS-445.20"           "MIPS-445.30"
 [73] "MIPS-470.10"           "MIPS-470.20"           "MIPS-475.05"
 [76] "MIPS-475.10"           "MIPS-480.10.05"        "MIPS-490"
 [79] "MIPS-500.10.110"       "MIPS-510.30"           "MIPS-510.40.20"
 [82] "MIPS-510.160"          "MIPS-510.170"          "MIPS-510.180.10.10"
 [85] "MIPS-510.180.10.20"    "MIPS-510.180.10.30"    "MIPS-510.180.10.40"
 [88] "MIPS-510.180.30.10"    "MIPS-510.180.30.30"    "MIPS-510.180.50.10"
 [91] "MIPS-510.180.50.20"    "MIPS-510.190.10.10"    "MIPS-510.190.10.20.10"
 [94] "MIPS-510.190.10.20.20" "MIPS-510.190.30"       "MIPS-510.190.40"
 [97] "MIPS-510.190.50"       "MIPS-510.190.60"       "MIPS-510.190.70"
[100] "MIPS-510.190.80"       "MIPS-510.190.90"       "MIPS-510.190.100"
[103] "MIPS-510.190.110"      "MIPS-510.190.120"      "MIPS-510.190.130"
[106] "MIPS-510.190.140"      "MIPS-510.190.150"      "MIPS-510.190.160.10"
[109] "MIPS-510.190.160.20"   "MIPS-510.190.160.30"   "MIPS-510.190.190"
[112] "MIPS-510.190.200"      "MIPS-520.10"           "MIPS-520.10.10"
[115] "MIPS-520.10.20"        "MIPS-20"               "MIPS-70"
[118] "MIPS-80"               "MIPS-110"              "MIPS-120"
[121] "MIPS-120.10"           "MIPS-120.20"           "MIPS-123"
[124] "MIPS-170"              "MIPS-180.10"           "MIPS-180.20"
[127] "MIPS-180.30"           "MIPS-190.10"           "MIPS-190.20"
[130] "MIPS-200"              "MIPS-210"              "MIPS-220"
[133] "MIPS-250"              "MIPS-280"              "MIPS-330"
[136] "MIPS-340"              "MIPS-350.20"           "MIPS-350.30"
[139] "MIPS-370"              "MIPS-390"              "MIPS-380"
[142] "MIPS-410.40.90"        "MIPS-410.40.100"       "MIPS-410.40.110"
[145] "MIPS-410.50"           "MIPS-430"              "MIPS-440.12.20"
[148] "MIPS-440.14.10"        "MIPS-440.40.10"        "MIPS-450"
[151] "MIPS-485"              "MIPS-510.10"           "MIPS-510.40"
[154] "MIPS-510.40.10"        "MIPS-510.110"          "MIPS-510.120"
[157] "MIPS-510.180.30.20"    "MIPS-520.20"           "MIPS-520.30"
[160] "MIPS-360"              "MIPS-360.10"           "MIPS-360.10.10"
```

```
[163] "MIPS-510.180.10"
```

The first, *Mips*, is a named list of character vectors where the names are the MIPS IDs and the vectors contain the constituent members to the respective protein complexes (i.e. it is also a hyper-graph representation exactly like the `getGOInfo` output):

```
> class(mips)

[1] "list"

> names(mips)[1:10]

 [1] "MIPS-60"           "MIPS-90.10"        "MIPS-90.20"
 [4] "MIPS-90.30"        "MIPS-125.10.10"    "MIPS-130"
 [7] "MIPS-133.30"       "MIPS-133.40"       "MIPS-133.50"
[10] "MIPS-140.30.30.20"

> mips$Mips$"MIPS-510.40"

NULL
```

And the second is a named character vector where the names, again, correspond to MIPS ID and the entries contain a description of its respective protein complex:

```
> names(mips$DESC)[1:5]

NULL

> mips$DESC["510.40"]

NULL
```

The GO package has methods of returning descriptions of any GO ID, and, therefore, it is redundant to create such a character vector containing the descriptions. Since there is no such methods for the MIPS dataset, it is necessary to record this vector as we parse the MIPS files.

The MIPS bi-partite graph incidence matrix is created exactly as the GO bipartite graph incidence matrix. The argument for `createMipsMatrix` is the output from `getMipsInfo`.

```
> mipsM = list2Matrix(mips)
```

Again we use the systematic gene names for yeast to index the rows of the MIPS incidence matrix and the the MIPS ID's corresponding to protein complexes index the columns. It is important to note that `createMipsMatrix` has prefixed each of the MIPS id's with the term "MIPS-". This is useful when we merge different incidence matrices, for it allows us to know where each protein complex orginates.

2.3 Purified Data Set Complexes estimated by apComplex

Protein complexes obtained from the both the GO and MIPS repository haven generally come from small-scale experiments, and the resulting protein complex estimates have been curated. These estimates, therefore, have a stronger likelihood to correctly represent true protein complexes found in vivo. Within the last five years, high throughput techniques such as Tandem Affinity Purification followed by Mass Spectroscopy have assayed protein complex co-membership relationships in large scale. We use data derived from such AP-MS experiments and complex estimation algorithms on such data to generate novel protein complex estimates (which have not been curated).

```
> library(ppiData)
> get("Gavin2006BPGraph")

A graphNEL graph with directed edges
Number of Nodes = 2551
Number of Edges = 19105
```

Presently, three small-scale and two large-scale AP-MS experimental datasets are publically available. The R-data package ppiData contain these five datasets and store them as a directed graph object (as seen from the example "Gavin2006BPGraph"). These directed graphs represent bait to prey co-membership data assayed from the experiment rather than explicit protein complex composition. In fact, protein complex estimation is a difficult computational problem that is in itself an active area of research.

From the bait/prey data obtained from ppiData, we applied a quality assessment on the protein interactions (cite); those proteins which are likely affected by a systematic bias of the experimental assay are removed from further analysis. Once the quality assessment is completed, we applied the penalized likelihood method of protein complex estimation given in apComplex (cite) to obtain protein complex estimates from the data.

The methods and discussion for conducting the quality assessment on the AP-MS protein interaction data can be found within the R-package ppiStats, and we defer discussion of this process to that package. Likewise, a discussion concerning the estimation of protein complexes from the bait to prey data can be found within apComplex. For the purposes of generating high quality protein complexes to be added to the in silico interactome, we generated protein complexes via apComplex by removing all proteins likely affected by a systematic bias at the p-value threshold of 0.01 and then setting the `commonFrac` parameter of the `findComplexes` function of apComplex to $\frac{1}{2}$. One further step taken to produce higher quality complexes is to use statistically significant derived data, i.e. only those estimates which fall under the multi-bait (more than one protein in the complex was used as a bait protein in the AP-MS techonology) and multi-edge (the directed graph showed higher instances of reciprocity) were taken to be high quality protein complex estimates.

Again the bi-partite graph incidence matrix are indexed in the rows by the yeast standard gene names, but since there are no complex id's for the estimates reported by apComplex, the `getAPMSData` function denominates ad hoc id's for these complexes. The columns are indexed by such ad hoc ids which reference the wet-lab experiments from where the were derived.

3 Comparing Protein Complexes Within and Across Databases

Now that we have obtained the various bi-partite graph incidence matrices, we need to evaluate and cross reference the protein complexes within a single database as well as across databases and eliminate redundancy. There are two items for which we must find when cross referencing protein complexes: equality or inclusion. To compare two bi-partite graph incidence matrices, we use the `runCompareComplex` function:

```
> mips2mips = runCompareComplex(mipsM, mipsM)
> go2go = runCompareComplex(goM, goM)
> mips2go = runCompareComplex(mipsM, goM)
> names(mips2go)

[1] "JC"         "equal"      "toBeRm"     "subcomplex" "toBeRmSubC"
```

The `runCompareComplex` function takes two bi-partite graph incidence matrices as arguments, and compares each complex from the first bipartite graph to the each complex from the second bi-partite graph. The argument *byWhich* shall be elucidated in the following paragraphs. Note that the bi-partite

graphs need not be different as we can run this function on a single bi-partite graph. The output of this function is a named list whose entries carry various statistical information. An explanation of each entry follows:

3.1 Jaccard Coefficients

The JC entry of the output from `runCompareComplex` contains a matrix of the Jaccard similarity coefficient between the two incidence matrices that are compaired - in our running example, the complexes of mips and the complexes of go. The $(i,j)^{th}$ entry of this matrix corresponds to the Jaccard similarity between the i^{th} complex of mips and the j^{th} complex of go. The coefficient is calculated by taking the quotient of the cardinality of the intersection of these complexes $mips_i \cap go_j$ by the cardinality of the union of the same complexes $mips_i \cup go_j$:

$$JC_{i,j} = \frac{|mips_i \cap go_j|}{|mips_i \cup go_j|}. \tag{1}$$

```
> all(diag(mips2mips[["JC"]]) == 1)
```

```
[1] TRUE
```

Equality of complexes returns a Jaccard index of unity while disjoint complexes return zero. So when we compared the Incidence matrix of MIPS against itself, we would have gotten 1's along the main diaganol. We remark that if one complex completely contains another, e.g. $go_j \subset mips_i$, the Jaccard index will be the cardinality of go_j divided by the cardinality of $mips_i$:

$$\frac{|go_j|}{|mips_i|}; \tag{2}$$

it is important to realize that containment is not easily ascertained without the cardinality of each complex at hand.

3.2 Alignment of Protein Complexes

The $maxIntersect$ entry of mips2go aligns either the complexes of go to complexes of $mips$ or complexes of $mips$ to complexes of go or both. We define alignment in this context to mean for a given protein complex of one bipartite graph which complex(es) in the second bipartite graph will return a maximal Jaccard coefficient. This operation is not symmetric, and therefore the $byWhich$ argument in `runCompareComplex` instructs the function as to which alignment to produce: "ROW" compares all the complexes of the second bi-partite graph with each complex to the first; "COL"compares all the complexes of the first bi-partite graph with each complex of the second; and "BOTH" will produce both the the outputs. This particular alignment will often not be one to one, but rather, one to many if either the "ROW" or the "COL" parameters are set. The mapping will be many to many if the "BOTH" parameter is set.

The $maxIntersect$ entry is itself a named list with two entries:

```
> names(mips2go$maxIntersect)
```

```
NULL
```

The maximize entry records the maximum Jaccard indices for each of the complex aligned either by "row" or by "column" or both if they are both called. The maxComp entry contains a list of named vectors also either by "row" or "column"; the names of the vectors correspond to the complex aligned and the entries are the names of those complexes for which produce the highest Jaccard index.

```
> mips2go$maxIntersect$maximize$row[1:3]

NULL

> mips2go$maxIntersect$maxComp$row[1:3]

NULL
```

The information obtained by the *maxIntersect* entry allows us to interpret how similar (or close) each complex is with respect to other complexes. From this information, we can decide whether to allow or dis-allow any protein complex within the `in silico` interactome if they are deemed to be too similar. For the creation of the working example interactome, we only dis-allow one complex from a pair of complexes if and only if that pair returns Jaccard index 1 which implies equality between the two complexes.

3.3 Equality of Protein Complexes

The *equal* entry records protein complexes of the first bi-partite graph matrix equal to those in the second and this is easily done by selecting those complexes which have a Jaccard index of unity. To produce an *in silico* interactome, we will eventually merge the different bi-partite graph matrices, and if two different bi-partite graphs have common protein complexes, we need to determine those complexes and eliminate redundancy.

```
> length(mips2go$equal)

[1] 46

> names(mips2go$equal[[1]])

[1] "BG1Comp"      "BG2Comp"      "orderBG1Comp" "orderBG2Comp" "intersect"

> mips2go$equal[[1]]

$BG1Comp
[1] "MIPS-90.10"

$BG2Comp
[1] "GO:0033186"

$orderBG1Comp
[1] 3

$orderBG2Comp
[1] 3

$intersect
[1] 3
```

The *equal* entry is a list of list which in turn has five entries: "BG1Comp" records the complex of the first bi-partite graph; "BG2Comp" records the complex of the second bi-partite graph; "orderBG1Comp" records the cardinality of "BG1Comp"; analogously for "orderBG2Comp"; and finally, "intersect" records the cardinality of $BG1Comp \cap BG2Comp$. Rather than just recording the complex names, this data structure allows the user to inspect relative size of those common complexes.

We remark here that there can be redendancy even within a single database as well, and it is necessary that we find and eliminate those protein complexes that may have occurred more than once in a database:

```
> ind = which(sapply(mips2mips$equal, function(w) w$BG1Comp !=
+     w$BG2Comp))
> mips2mips$equal[ind]
```

The code above shows the redundancy within the Mips data-base. Though we have hidden the outputs, we encourage the user to run these commands to see the redundancies found within Mips and within any other data repository.

3.4 Eliminating Redundancy

The "toBeRm" entry takes only those protein complexes of the first bi-partite graph (in our running example, this would be mips or equivalently mipsM) which are equal to some protein complexes of the second bi-partite graph. The result is a character vector of protein complexes that will need to be deleted in order to avoid redundancy in the *in silico* interactome.

3.5 Protein Sub-Complexes

The "subcomplex" looks for any complex from one bi-partite graph which is completely contained in a complex from the other bi-partite graph. The outputs in this entry are identical to those of the *equal* and defer the reader to the explanation written previously.

While the storage of the data does not explicitly state which complex contains the other complex, it is pretty clear that the sub-complex must have the smaller cardinality. It is quite evident that if two complexes are equal, then one of the two complexes must be removed to eliminate redundancy; it is not clear, however, whether or not to remove sub-complexes. For the purpose of simulating AP-MS technology, sub-complexes will be masked by the aggregate complex, and thus, removal is needed to avoid redundancy. There may be other computational tools that require the availability of the sub-complexes, so there is no consistent framework calling for the removal of sub-complexes, and hence, we leave that decision to the users. For the construction of the working example of the `in silico` interactome, we do not remove protein sub-complexes, and so our estimates will carry through with their inclusion.

3.6 Removing Protein Sub-Complexes From the Interactome

If removal of sub-complexes is necessary, we have built a simple component into the `runCompareComplex` function that makes this process fairly simple. The *toBeRmSubC* entry records all those protein clusters which a proper sub-complexes of another protein complex, either within a single database or across databases.

```
> mips2mips$toBeRmSubC[1:3]

[1] "MIPS-230.20.10" "MIPS-260.20.10" "MIPS-260.20.20"

> mips2go$toBeRmSubC[1:3]

[1] "MIPS-60"        "MIPS-90.20"     "MIPS-125.10.10"
```

Much like the *toBeRm* entry of the output from `runCompareComplex`, the *toBeRmSubC* entry is a a character vector of protein complexes that should be removed if sub-complexes are irrelevant.

As an aside, we remark that it is necessary to record what we have removed, either redundant protein complexes or protein sub-complexes, so we advise users who are building these `in silico` interactomes to save the pair-wise comparison data from the output of `runCompareComplex`.

```
> save(mips2go, file = "mips2go.rda", compress = TRUE)
```

4 Combining Bi-Partite Graphs - Building In Silico Interactomes

After we have called the `runCompareComplex` function on two different bi-partite graph matrices, we can merge these repsective graph incidence matrices to form an aggregate incidence matrix and thus creating an aggregate bi-partite graph. The aggregate bi-partite graph, A, has certain important characteristics: the rows of A are indexed by the union of the proteins of the two constituent bi-partite graphs and the columns are indexed by the union of the protein complexes of the two constituent bi-partite graphs.

To merge two incidence matrices, we simply call the `mergeBGMat` function:

```
> dim(mipsM)

[1] 814 163

> dim(goM)

[1] 2244  323

> ISI = mergeBGMat(mipsM, goM, toBeRm = c(mips2go$toBeRm, mips2mips$toBeRm,
+     go2go$toBeRm))
```

The first two arguments of the `mergeBGMat` function is two bi-partite graph incidence matrices; the last argument is a character vector of those protein complexes which should be deleted in order to avoid redundancy or to eliminate certain protein complexes that should be dis-allowed from the *in silico* interactome. We have chosen to only remove redundant protein complexes in our working example; the end user has the freedom to choose whichever protein complexes to remove from the in silico interactome. A character vector consisting of the identification code of the unwanted protein complexes should be passed into the `toBeRm` parameter so that these protein complexes are disallowed.

Once two bi-partite graphs are merged, we can now call the `runCompareComplex` on the *merged* incidence matrix and a new bi-partite graph matrix (one which is not any of the constituent matrices of *merged*). It is also important to call the `runCompareComplex` function on each bi-partite graph from within one database in addition to comparing across databases. In this way, redundancy can be eliminated in all avenues, and we can construct the *in silico* interactome iteratively by merging a new incidence matrix in each subsequent iteration.

5 The estimated Saccharomyces cerevisiae In Silico Interactome

Up until now, we have demonstrated how to use the various functions of the ScISI, but we diverge from that endeavor to explore the characteristics of our working example `in silico` interactome for budding yeast. By obtaining the dimension tag of incidence matrix representing the interactome,

```
> dim(ISI)

[1] 2507  435
```

we can determine the number of uniquely expressed genes of the yeast genome (i.e. the number of rows of the ISI which is the first entry of the output above), and the number of distinct multi-protein complexes derived from the five different data sources (respectively the number of columns of the ISI which corresponds to the second entry).

There are two summary statistics that are of some importance. The first figure gives an indication of the distribution of the cardinality of the protein complexes within the *in silico* interactome. We can see

from this figure that the majority of the protein complexes have less than ten constituent proteins. This statistic verfies what many have conjectured - that most multi-protein complexes are have relatively few proteins. It is also widely conjectured that the larger multi-protein complexes are built from the union of sub-complexes (e.g. Mitochondrial translocase complex) which can be verified in our working interactome by exploring the relationships between aggregate complexes and sub-complexes. We have touched upon this analysis in Table 2, but a more thorough and exhaustive analysis is conducted in (cite).

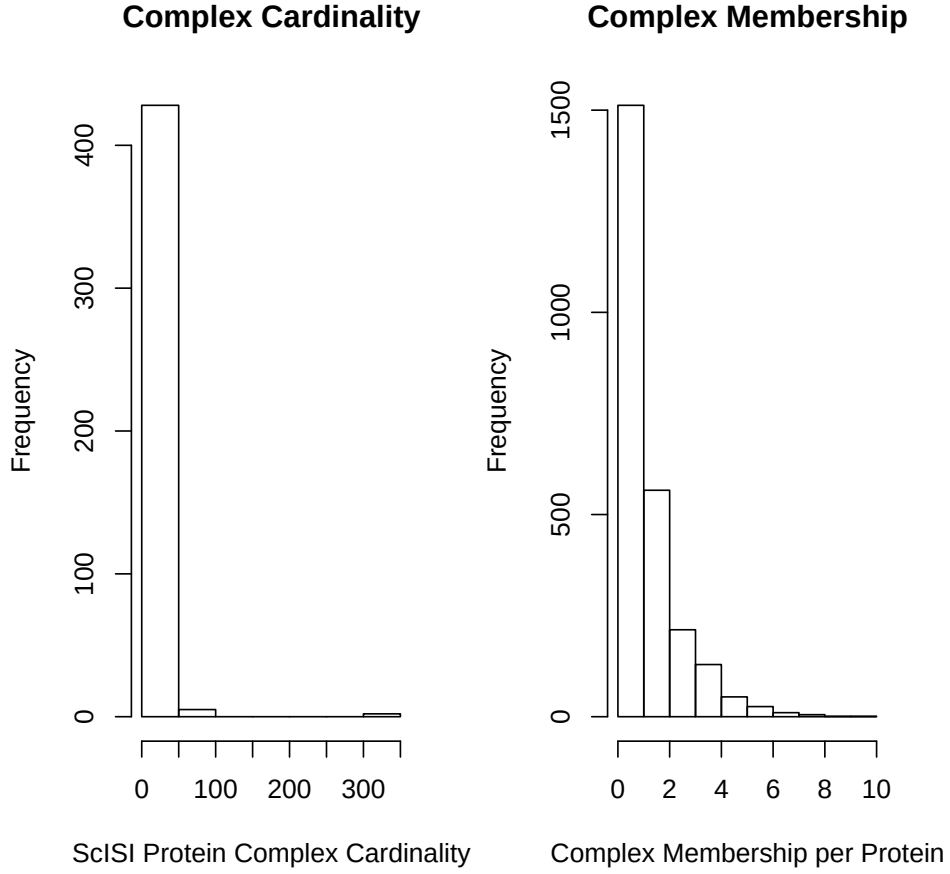

Figure 1:

The second summary statistic gives the distribution of the number of complexes for which each protein may have membership. This statistic is important because we can use this distribution (along with the summary statistic above concerning protein complex size) to give us a sense of the nodal degree of a generic protein in this rather large co-membership network.

The second histogram shows that an overwhelming majority of proteins participate in fewer that five distinct protein complexes. Because the majority of complexes have fewer than ten members and because the majority of proteins are members of fewer than 5 protein complexes, the number of neighbors of a generic protein in the protein-protein co-membership graph should also be relatively small.

We also discuss some simple summary statistic concerning the pairwise comparisons between the five data-sets in which we have obtained our protein complexes.

Table 1 reports the number of protein complexes that were redundant by comparing the repositories pairwise including self comparisons. Table 1 shows that the only data-set with self-redundancy is derived from the MIPS repository; and upon investigation, MIPS has categorized protein complexes by known functionality, and three protein complexes were placed in two different categories each. Table 1 directly shows that the no two repositories has extremely large overlap.

	MIPS	GO	Gavin	Ho	Krogan
MIPS	4	38	1	2	0
GO	38	0	5	2	1
Gavin	1	5	0	0	1
Ho	2	2	0	0	0
Krogan	0	1	1	0	0

Table 1: Number of repetitive Protein Complexes

Not limited to redundant protein complexes, we also process each data-set independently to find and to remove all protein sub-complexes. The results are reported in Table 2 where the row names indicate the location of the sub-complex and the column names indicate the location of the aggregate protein complex. Reading across row two of Table 2 we see that GO contains protein complexes that were protein sub-complexes in each of the other data repositories including itself. Table 2 is much less sparse relative to Table 1, and it is this relative comparison that makes all the estimates more credible since each data repository is based on unique experiments coupled with unique estimation algorithms.

	MIPS	GO	Gavin	Ho	Krogan
MIPS	39	52	21	0	2
GO	34	14	19	1	7
Gavin	10	26	0	5	1
Ho	12	14	9	0	1
Krogan	11	8	14	4	0

Table 2: Number of Protein Sub-Complexes

While these four summary statistics give a good description of the *in silico* interactome, these statistics cannot give a comprehensive description; that would take a much more considerable dissection of the interactome (for which computational experiments should be conducted). It is also necessary to state that this interactome represents Saccharomyces cerevisiae for some set of environmental conditions, and so is one of the many estimates that can be created by this package.

We remark that the interactome can only give predicted members of the protein complexes of interest, but it cannot ascertain the multiplicity of each individual constituent protein member. Therefore, it is important to be mindful that the *in silico* interactome is a good model and representation of known biology, and so it is limited in its presentation.

6 Customizing the Interactome

We have successfully built an *in silico* interactome, and for some computational methodologies, this interactome is the best interactome to use. In this section, we present some other functions within ScISI that allows the users to refine the interactome from the coarser one built.

6.1 Generating an Instance of Class yeastData

One of the functionalities of the SclSI package is to create an instance of the class *yeastData*. Before we can instantiate this class, some preliminary functions need to be called.

The class *yeastData* will contain all the available information of some bi-partite graph, and so we must collate these sets of information into one usable format. We can do this by calling either the `createGODataFrame` or `createMipsDataFrame` function.

```
> goDF = createGODataFrame(go, goM)
> mipsDesc <- sapply(mips, function(x) {
+     attr(x, "desc")
+ })
> mipsDF = createMipsDataFrame(mipsDesc, mipsM)
```

The names of these two functions describe the output quite explicitly, but the arguments need some clarification. The arguments for the both functions are essentially the same. The arguments for `createGODataFrame` are:

1 `go` - A named list whose entries are character vectors. This is essentially the hyper-graph of the GO protein complexes. The GO id's are obtained by calling the `names` function on the list, and the description for the protein complexes is obtained by calling the `getGOTerm` function from GO.

2 `goM` - The incidence matrix representing the GO hyper-graph. The names of the GO protein complexes are obtained by calling the `colnames` on this matrix.

The arguments for the function `createMipsDataFrame` are:

1 mips$DESC - A named character vector. The MIPS' id's are obtained by calling `names` on the vector, and the description is obtained by accessing the vector values.

2 mipsM - The incidence matrix representing the MIPS hyper-graph. The names of the MIPS' protein complexes are obtained by calling the `colnames` on this matrix.

The generated database holds three distinct pieces of information for each protein complex: its name relative to the incidence matrix; its id (GO or MIPS); and its description.

Once we have created either the GO dataframe or the MIPS dataframe, we can proceed to call an instance of the *yeastData* class. Instantiating an *yeastData* object is done by calling the `createYeastDataObj` function:

```
> mipsOb = createYeastDataObj(mipsDF)
> goOb = createYeastDataObj(goDF)
```

With the instance of the *yeastData* class, we have access to three generic methods: `ID`, `Desc`, and `getURL`.

```
> ID(mipsOb, "MIPS-510.40")

[1] "MIPS-510.40"

> Desc(mipsOb, "MIPS-510.40")

                   MIPS-510.40
"RNA polymerase II holoenzyme"

> getURL(mipsOb, "MIPS-510.40")
```

```
[1] "http://mips.gsf.de/genre/proj/yeast/searchCatalogListAction.do?style=cataloglist.xslt&table=CELLU
```

The arguments for each of the methods are identical: the first argument (`mipsOb` in our running example) is the instance of *yeastData*; the second argument ("MIPS-510.40" in the running example) is the name of the protein complex as defined in the incidence matrix `mipsM`, i.e. the column names. A good exercise is too create an instance of *yeastData* with the GO dataframe and work with the generic methods.

The return values the `ID` and `Desc` are self-explanatory. The output for `getURL` returns the url for the web-page containing a detailed description of the MIPS or GO protein complex. This allows the user to parse through the derived protein complexes manually and refine the *in silico* interactome. Simply call the `browseURL` function with the output of `getURL` as the argument to open a browser with the specified web-page.

6.2 Generating an Informational .html File

Once we have created an instance of *yeastData*, we can call the `ScISI2html` function to generate a .html file in the user's home directory.

```
> mipsNames = colnames(mipsM)
> mipsCompOrder = colSums(mipsM)
> url = vector(length = length(mipsNames))
> linkNames = vector(length = length(mipsNames))
> for (i in 1:length(mipsNames)) {
+     url[i] = getURL(mipsOb, mipsNames[i])
+     linkNames[i] = Desc(mipsOb, mipsNames[i])
+ }
> ScISI2html(urlVect = url, linkName = linkNames, filename = "mips.html",
+     title = "MIPS Protein Complex", compSize = mipsCompOrder)
> goNames = as.character(goDF[, 1])
> goCompOrder = colSums(goM[, goNames])
> url = vector(length = length(goNames))
> linkNames = vector(length = length(goNames))
> for (i in 1:length(goNames)) {
+     print(i)
+     url[i] = getURL(goOb, goNames[i])
+     linkNames[i] = Desc(goOb, goNames[i])
+ }
> ScISI2html(urlVect = url, linkName = linkNames, filename = "go.html",
+     title = "GO Protein Complex", compSize = goCompOrder)
```

The arguments for the `ScISI2html` are somewhat self explanatory, but we will provide a terse description of each argument:

1 `urlVect` is a character vector consisting of url address for the protein complexes.

2 `linkName` is a character vector taht will be anchored by `url`.

3 `filename` is a character that will name the file, e.g. in our running example, the output file will be named "mips.html" and "go.html".

4 `title` will be the title of the .html file.

The purpose for this .html file is a quick and efficient way for users to manually parse the protein complexes derived from ScISi either for quality control and for refinement.

```
> data(ScISIC)
> identical(ISI, ScISIC)

[1] FALSE
```

The function has two arguments, the bi-partite graph incidence matrix and a character vector. Each entry of the character vector is an protein complex ID which will be deleted. After the deletion of these protein complexes, the function also removes any proteins which are no longer apart of any protein complex.

7 Conclusion

The process given above has created one example on an *in silico* interactome. For the purposes of simulating AP-MS data and generating estimates of via estimation algorithms, this interactome is a sound place from where to begin. It is clearly not the most suitable interactome for every computational experiment, and we concede that refinements will inevitably be made. The *in silico* interactome will also change and refine itself as the data from the GO and MIPS databases are refined. One of the most important aspect to the R package ScISI is its dynamic nature, for the basic principal in which it has been built is that the *in silico* interactome must be amenable to change due to a number of outside circumstances.