HitWalker Vignette

Daniel Bottomly

November 29, 2012

1 An initial example

We will begin by replicating Figure 1 in Bottomly et al. (submitted) using a basic workflow. For most users, a workflow of this kind will be used frequently once the necessary database is setup. This example assumes that the `RSQLite` and `HitWalkerData` packages are installed and uses a very simple database that is bundled with `HitWalkerData`. The bundled graph is from STRING (http://string-db.org/newstring_download/protein.links.detailed.v9.0.txt.gz; 1-30-2012) with the edge weights thresholded at 400 and limited to human proteins. Creation of this database is covered in the Hitwalker_Create_DB vignette.

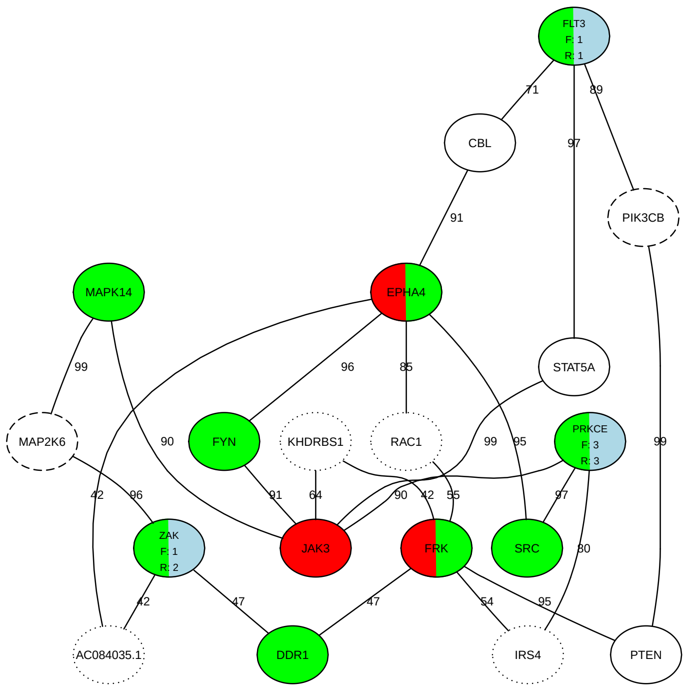
```
> stopifnot(require(HitWalker))
> stopifnot(require(RSQLite))
> stopifnot(require(HitWalkerData))
> data(params)
> db.con <- dbConnect("SQLite", hitwalker.db.path())
> graph.obj <- loadGraph(graph.file.path(), examp.prior.param)
> test.out <- run.prioritization.patient(db.con, "08-00102", graph.obj, examp.prior.param)
> stopifnot(dbDisconnect(db.con))
>
```

The `variantPriorResult` object returned from `run.prioritization.patient` contains all of the information retrieved from the database for that sample. To create a graphical representation, we first create a `graphDispParams` object which controls many of the aspects of the plotting. Examples of creation of this object can be found in the Hitwalker_Add_Metadata vignette. For now we can use the saved parameters in the `HitWalkerData` package:

```
> plot(test.out, examp.graph.param)
```

```
Found 3 target nodes and 3 mut nodes
Finding target--mutation distances
Finding target--target distances
Finding mut--mut distances
Making subgraph
Converting to graphNEL
```

```
>
```



Aspects of `graphDispParams` objects will be examined in the following sections. As the main purpose of this package is to prioritize variants, of greater interest might be the ranked variants themselves along with accompanying metadata.

```
> sum.dta <- summary(test.out)
> disp.cols <- c("symbol", "protein", "chr", "pos", "ref", "alt", "type.rank")
> head(sum.dta[,disp.cols])

    symbol         protein chr       pos ref alt type.rank
10    FLT3 ENSP00000241453  13  28610138   G   A         1
24     ZAK ENSP00000364361   2 174128561   A   T         2
33   PRKCE ENSP00000306124   2  46372327   C   T         3
48 RPS6KA2 ENSP00000386050   6 167271710   G  GC         4
23   MYO3B ENSP00000386213   2 171070907   G   T         5
39     MYC ENSP00000367207   8 128750527   T   C         6

>
```

2 A closer look

The parameter classes, `priorDbParams` and `graphDispParams` are the most important part of `HitWalker` and work with the specified database to provide the desired output. Next we examine some of the considerations if using an inhouse database. These considerations are explored below. Those interested in adopting our MySQL schema are referred to www.biodevlab.org/software.html for documentation. Queries that will work with our schema are provided in the `HitWalker:::defaultPriorDbParams()`.

2.1 Graph and annotation

The simplest way to maintain a graph structure associated with the variant and functional assay data is by storing it in a database table. If it is stored with a row indicating an edge, with a column indicating the 'from' node, another indicating the 'to' edge and the weight then it can easily be retrieved using `get.protein.protein.graph` with the exact retrieval procedure being controled by a function supplied to the `matrix.query` slot of `priorDbParams` that returns the requisite SQL. Accompanying annotation will be retrieved by SQL derived from a function supplied to the `annot.query.func` slot. Currently `get.protein.protein.graph` subsets the graph and annotation to only the nodes that are common to both. In addition there is potential for local upweighting of edges based on scores supplied to nodes however, it should be considered highly experimental at this time.

Currently `get.protein.protein.graph` assumes that the graph database table contains columns named 'protein1', 'protein2' and 'weight' and that the annotation `data.frame` contains a column named 'protein'. For the other functions more flexibility is allowed. In this implementation both 'core' and 'summary' (e.g. protein and gene symbol respectively) column names need to be specified. Each function typically only operates on the data using either the 'summary' or 'core' IDs and this is controlled by the named character vector supplied to the `func.summary.map` slot of the `priorDbParams` class. In order to enable this functionality any of the retrieved `data.frame` objects should have columns corresponding to either of the 'core' or 'summary' columns.

2.2 Sample Names

The next component of `HitWalker` is resolution of the sample name. The current implementation is very simple and assumes that there will be little ambiguity. There are four relevant slots in the `priorDbParams` object: `id.table`, `possible.id.cols`, `reconciled.id.col` and `sample.id.col`. Essentially the specified columns are searched in the table for a partial match to the specified sample identifier. If a unique match is found, the values in `reconciled.id.col` and `sample.id.col` will be passed on to downstream functions. This procedure is carried out using the `reconcile.sample.name` function. Further information can be found in the documentation and source code.

2.3 Variants, functional hits and metadata

After determining the sample name, the variants, functional hits and relevant metadata are retrieved. This is carried out by specified functions that return SQL queries in text form placed in the `variant.query`, `hit.query` and `patient.overlay.func` slots accordingly. These slots will be accessed using methods defined for the `priorDbParams` class and the queries will be dispatched using a common mechanism, namely the `retrieve.param.query` function.

2.3.1 Variants and text-based filters

In addition to variant retrieval, a basic filtering mechanism, `filter.variant.annots`, is provided for text-based filters created from the `variantFilter` class. This is provided to facilitate hard filtering that has to be performed prior to any analysis. By specifying the criteria in the program as opposed to the database, a researcher can easily determine the effect of the chosen criteria. Also of importance are the `pat.var.id` and `variant.type.col` slots of `priorDbParams`. These specify the columns that uniquely identifies a variant and the type of variant respectively.

2.3.2 Functional hit scores

Multiple types of functional hit data can be retrieved. However, for prioritization the scores have to be summarized into a single value for each unit. The manner in which this occurs needs to be specified using the `score.summary.func` slot with `hit.comb.score.name` indicating the resulting column.

2.4 Metadata

Metadata relavant to all the nodes in the graph should be retrieved along with the rest of the annotation (See the Graph and Annotation subsection). If samples have specific metadata applicable to a subset of their nodes then that is retrieved though the function specified in the `patient.overlay.func` slot of the `priorDbParams` object.

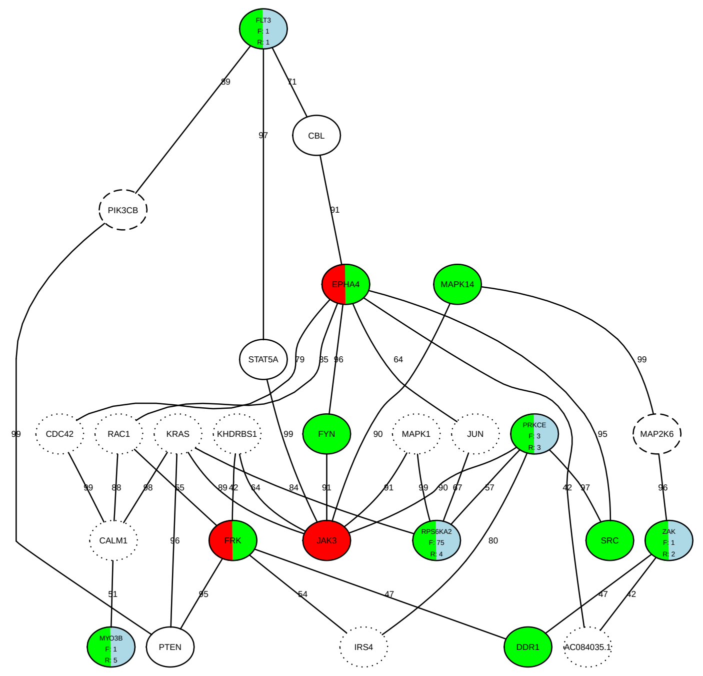
2.5 Plotting considerations

Plotting is carried out using by first instantiating a `graphDispParams` class using the `makeGraphDispParams` function. Of universal relevance are the `file.name`, `width` and `height` arguments which specify the PDF to which the plot is drawn and its dimensions. On screen display is provided by supplying `character()` as the `file.name`. The size of the plotted graph (in terms of nodes and edges) as well as the time it takes to create is controlled by the `max.plot.vars` and `max.plot.hits` arguments. For example, increasing the number of variants from the default 3 to 5 in the example above results in the figure below when supplied to the `plot` method:

```
> maxPlotVars(examp.graph.param) <- 5
> plot(test.out, examp.graph.param)

Found 3 target nodes and 5 mut nodes
Finding target--mutation distances
Finding target--target distances
Finding mut--mut distances
Making subgraph
Converting to graphNEL

>
```

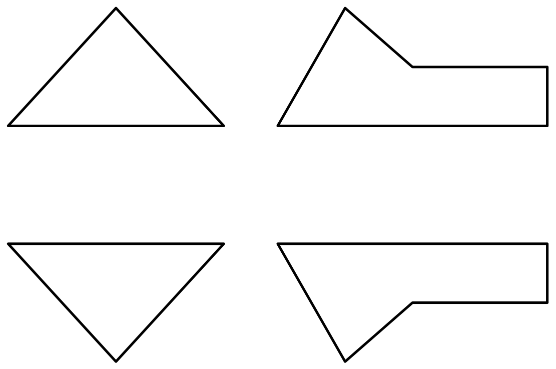


Other arguments to `makeGraphDispParams` can be used to control how the retrieved metadata are mapped to the nodes of the graph. Currently there are three different manners of displaying metadata. Nodes can take different shapes, have different colors, as well as have different borders styles.

2.5.1 Node shapes

In this implementation of `HitWalker`, all of the nodes shapes available from `RGraphviz` could potentially be used in addition to up and down triangles as well as other shapes. For example, we use triangles to indicate sample-specific expression differences and 'half' triangles to indicate alternative splicing events (the direction of the point indicating direction of differences). The standard ellipses indicate no expression differences and/or no information and a rectangle indicates annotation issues or other ambiguity and is drawn with a warning. This is controlled using the `shape.mapping.func` function supplied to `graphDispParams`. Examples of the triangles are shown below:

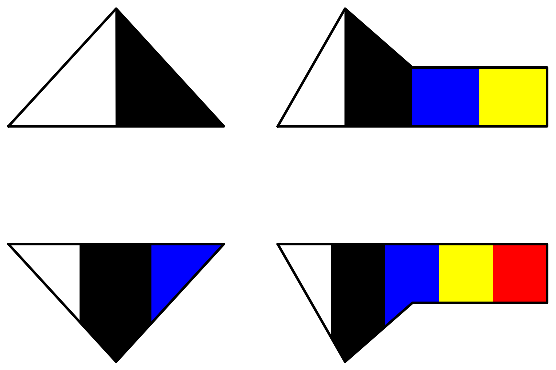
```
> plot.new()
> HitWalker:::draw.triangle(.2,.75,.25,.2,.2,"white","black",2,1,"up_std")
> HitWalker:::draw.triangle(.2,.25,.25,.2,.2,"white","black",2,1,"down_std")
> HitWalker:::draw.triangle(.75,.75,.25,.25,.25,"white","black",2,1,"up_half")
> HitWalker:::draw.triangle(.75,.25,.25,.25,.25,"white","black",2,1,"down_half")
>
```



2.5.2 Node colors

In addition to shapes, colors also provide an effective way to communicate node relationships. In this implementation they refer to variant or functional hit status. Colors are mapped to nodes using the character vectors specified to the `hit.colors` and `query.color` arguments of the `makeGraphDispParams` function in conjunction with the `getFillColors` method of `variantPriorResult`. When multiple colors overlap (e.g. a node is both a functional assay hit and has a variant), the node is divided into equal pieces and colored accordingly. Some examples follow:

```
> plot.new()
> use.color <- c("white", "black")
> HitWalker:::draw.triangle(.2,.75,.25,.2,.2,use.color,"black",2,1,"up_std")
> use.color <- append(use.color, "blue")
> HitWalker:::draw.triangle(.2,.25,.25,.2,.2,use.color,"black",2,1,"down_std")
> use.color <- append(use.color, "yellow")
> HitWalker:::draw.triangle(.75,.75,.25,.25,.25,use.color,"black",2,1,"up_half")
> use.color <- append(use.color, "red")
> HitWalker:::draw.triangle(.75,.25,.25,.25,.25,use.color,"black",2,1,"down_half")
>
```



2.5.3 Node borders

Changing node borders is the last mechanism for displaying relevant metadata on the plots. It is controlled by a function supplied to the `style.mapping.func` argument of the `makeGraphDispParams` function. Currently appropriate return values are defined as in the `lty` subsection of `par`.