Minimal Free Resolutions Package

David J. Marchette

1 Introduction

A graph is a set of vertices and a set of edges, $G = (V, E)$. In the `mfr` package all graphs are assumed undirected and simple: there are no self-loops (edges from a vertex v to itself) and no multiple edges.

The neighborhood of a vertex v is the set of vertices connected to v by an edge, union the vertex v. We write this as $N[v]$. A splitting edge is an edge between vertices u and v, such that $N[u] \subset N[v]$.

A graph is chordal if every induced cycle has a chord (an edge between two vertices that are not adjacent in the cycle). The complement of a graph is a graph on the same vertices with an edge in the new graph between vertices u and v if and only if there is no such edge in the original graph.

A graph invariant is a function that is constant on isomorphism classes. The Betti numbers of a graph correspond to a matrix-valued graph invariant. The details of what these are and some information about how they can be computed is in the vignette `edgeideals` and the references therein. For now, we will simply treat these as a "black box" of graph invariants, and look at the various functions available in the `mfr` package that allows us to compute them (at least for small graphs).

The `mfr` package extends the `igraph` package and makes use of the functionality inherent in it. There are a number of fundamental limitations that are the result of the computational complexity of the minimal free resolution (MFR).

1. The algorithms work primarily on chordal graphs. The algorithm is recursive, so the size of the graph for which the MFR can be calculated depends on the size of the recursion stack, and on the size of memory.

2. The recursive algorithm can be applied to non-chordal graphs, so long as there exist splitting edges in the graph. If the graph is not chordal, at some point in the recursion a graph will occur which has no splitting edges. At this point, either one of the special cases will be used, the Singular algebraic geometry package will be called if available, or an approximation to the MFR will be returned.

3. It is unlikely that the MFR for a graph with more than a couple dozen vertices can be computed on a reasonable computer in a reasonable amount of time if the graph contains no splitting edges and is not one of the special cases.

```
> set.seed(1234)
> g <- rtree(20)
> plot(g, layout = layout.fruchterman.reingold, vertex.label = "",
+     vertex.size = 1)
```

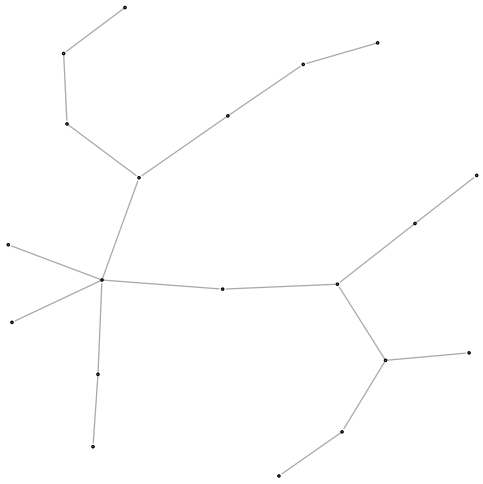

Figure 1: A random tree.

The following sections detail the main functionality of the package. Section 2 describes some new random graphs, and Section 3 describe some new graphs added to the ones available in the `igraph` package, while the later sections cover the minimal free resolutions themselves.

2 Some Random Graphs

The `mfr` package has several functions for creating random graphs. Because of the recursive algorithm for chordal graphs (see the vignette `edgeideals`) there is code to generate trees and chordal graphs.

By default, `rchordal` starts with a random tree and adds edges (keeping the graph chordal) until the desired number of edges is obtained.

There is also a slight generalization of the `erdos.renyi.game` random graph, a block model with two blocks called the `kidney.egg.game`. The basic idea is

```
> set.seed(2322)
> h <- rchordal(10, 20)
> plot(h, layout = layout.fruchterman.reingold, vertex.label = "",
+     vertex.size = 1)
```

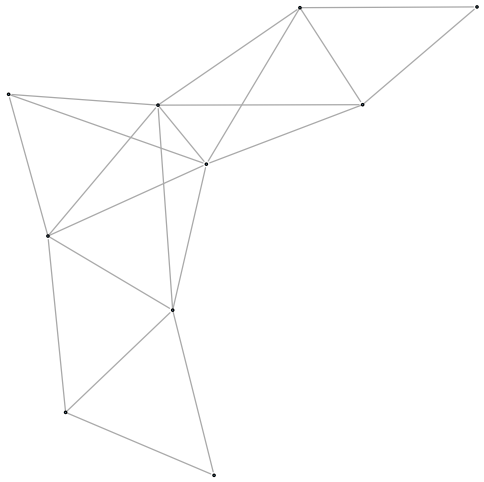

Figure 2: A random chordal graph.

```
> set.seed(2452)
> g <- kidney.egg.game(n = 30, p = 0.1, m = 10, q = 0.6)
> plot(g, vertex.label = "", vertex.size = 2)
```

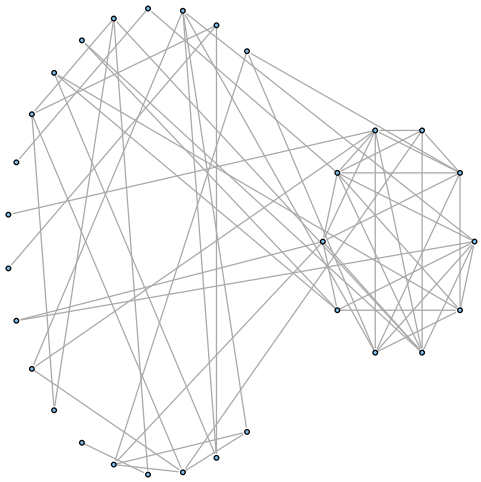

Figure 3: A kidney-egg graph.

that there is a large subset of vertices (the "kidney") and a small set (the "egg") with a higher edge probability in the egg.

Note that kidney-egg graphs have their own plot function, putting the egg on the right. Kidney-egg graphs are a special case of `product.game` graphs. This function takes two graphs and creates a new graph with these two as subgraphs, randomly connected by edges between the subgraphs. The use provides either a probability p or a number n, and either n edges are randomly selected between the subgraphs, or each possible edge is chosen with probability p. The argument p takes precedent, so if it is given n is ignored.

Another class of random graphs are the random dot product graphs (RDPG). A random dot product graph is a random graph in which each vertex v has associated with it a vector x_v, and the probability of an edge between two vertices u and v is the dot product of their vectors, $P[uv] = x_u^t x_v$. Conditional on the x vectors, the edges are independent, just as is the case with `erdos.renyi.game`. The function `rdpg.game` will generate an RDPG. If it is called with the argu-

ment x, it uses the matrix x to generate the graph, with one vertex for each row of x. If x is not given, it is generated uniformly inside the simplex.

Given an RDPG graph g, we may estimate the x vectors using `rdpg.estimate`. For this one mush posit the dimension of the vectors. One RDPG graph, with the estimated x points, is shown in Figure 4.

The picture in Figure 4 does not seem to be very impressive. The estimate is not terribly good. The estimates will improve as n increases. The average Frobenious error between the true probability matrix and the estimated one, in this case, is 0.11. Increasing n to 1000, a similar experiment results in the average Frobenious error of 0.02.

The estimates are not guaranteed to have valid dot products (values in [0,1]) or to be in the first quadrant. They are only unique up to rotations and flips.

3 Some New Graph Types

We have added some new graph types to the package to supplement those in `igraph`.

We implement a version of barbell graphs in `graph.barbell`. The graph consists of two "bells" – either two complete graphs or two cycles – connected by a path. See the top two graphs in Figure 5.

Two other graphs that have a particular coloring property (the vertices and edges can be colored so that every neighborhood has exactly the same number of each color) are depicted in the bottom two graphs of Figure 5.

One interesting way of creating new graphs from old is via clique expansion. Given a d-regular graph, we can replace very vertex with a d-clique. This can be done with the function `clique.expand`.

■ g <- graph.full(4) h <- clique.expand(g) k <- graph.truncated.tetrahedron() graph.isomorphic(h,k)

4 Minimal Free Resolutions

See the vignette `edgeideals`, and the references therein, for information about minimal free resolutions. For now we will simply consider an MFR to be a matrix of graph invariants.

The code uses the following logic:

1. Check to see if the graph is of a particular form that allows fast calculation. See Section 4.2
2. Check to see if the graph is in the pre-computed database. See Section 4.1
3. Determine if there is a splitting edge (an edge that allows the recursive algorithm to progress). If so, recurse.
4. If no splitting edges exist:

```
> set.seed(2452)
> g <- rdpg.game(n = 30, d = 2)
> est <- rdpg.estimate(g, d = 2)
> X <- procrustes(est$x, g$x)$X.new
> plot(g, layout = g$x, vertex.label = "", vertex.size = 2, rescale = FALSE,
+     xlim = c(min(X[, 1], 0), max(X[, 1], 1)), ylim = c(min(X[,
+         2], 0), max(X[, 2], 1)))
> points(X, pch = 20, col = 2)
> segments(X[, 1], X[, 2], g$x[, 1], g$x[, 2])
```

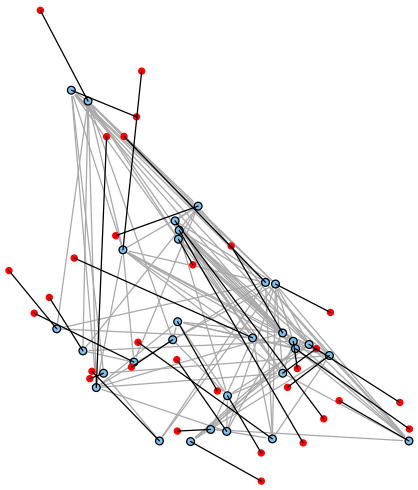

Figure 4: A random dot product graph, with the estimated points shown in red. The black segments link the estimated point to the vertex.

```
> par(mfrow = c(2, 2))
> g <- graph.barbell(7, 5, 2)
> x <- layout.circle(graph.full(7))
> y <- scale(layout.circle(graph.full(5)), center = c(-5, 0), scale = FALSE)
> z <- rbind(c(2, 0), c(3, 0))
> plot(g, layout = rbind(x, y, z), vertex.label = "")
> g <- graph.barbell(7, 5, 2, full = FALSE)
> x <- layout.circle(graph.full(7))
> y <- scale(layout.circle(graph.full(5)), center = c(-5, 0), scale = FALSE)
> z <- rbind(c(2, 0), c(3, 0))
> plot(g, layout = rbind(x, y, z), vertex.label = "")
> g <- graph.truncated.tetrahedron()
> plot(g, vertex.label = "")
> g <- graph.martini.glasses()
> plot(g, vertex.label = "")
> par(mfrow = c(1, 1))
```

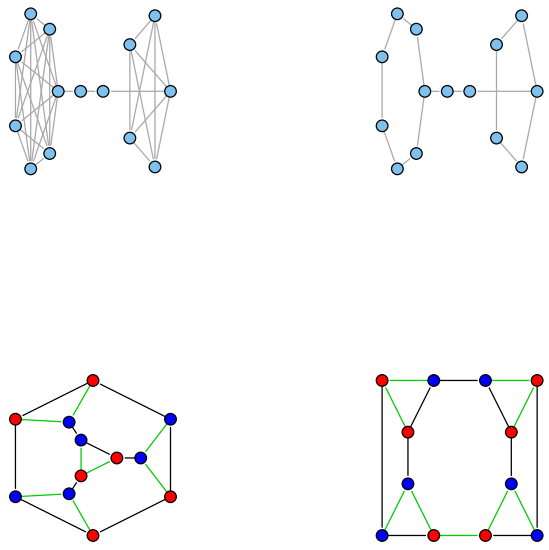

Figure 5: Two barbell graphs (top), a truncated tetrahedron (bottom left) and a "martinis glasses" graph (bottom right).

(a) If Singular is available, call it to compute the MFR.

(b) Else "fake it".

The last case, "fake it" means to try to approximate the MFR (see the `edgeideals` vignette).

Let's consider a typical result of calling `mfr`:

```
> g <- graph.famous("petersen")
> mfr(g)

$bettis
[1]   1  15  45  87 100  60  20   4

$graded
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    0    0    0    0    0    0    0
[2,]    0   15   30   10    0    0    0    0
[3,]    0    0   15   72   80   30    0    0
[4,]    0    0    0    5   20   30   20    4

$reg
[1] 4

$pd
[1] 7

$punted
[1] 0

attr(,"class")
[1] "mfr"   "exact"
```

The attribute `graded` is the graded Betti number matrix. `bettis` are the total Betti numbers, corresponding to the column sum of `graded`. The value `pd` is the projective dimension of the edge ideal, which is one less than the number of columns (it is the length of the non-trivial part of the resolution), and `reg` is the regularity (Mumford-Castelnuovo regularity) of the ideal, the number of rows of the graded resolution, and corresponds to a measure of how "complicated" the graph is. Finally, the attribute `punted` indicates whether the code had to call Singular, or, if Singular is unavailable, whether the code had to use an approximation (and in this case if it is non-zero the resolution is not guaranteed to be exact). Note that for the Petersen graph above (see Figure 6) we would have to call Singular to compute the resolution if it wasn't one of the graphs in the database for which the MFR is pre-computed. See Section 4.1.

The MFR can be computed separately on the connected components of a graph, and then the overall resolution can be computed from these. For example,

```
> x <- layout.concentric.circles(g, list(nodes = 5:9))
> plot(g, vertex.label = "", vertex.size = 2, layout = x)
```

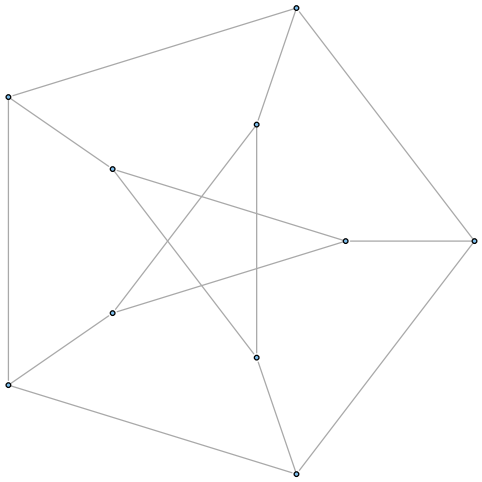

Figure 6: The Petersen graph.

for two components with resolutions β^1 and β^2 we have:

$$\beta_{i,j} = \sum_{p+q=i, r+s=j} \beta^1_{p,r} * \beta^2_{q,s}.$$

```
> g <- graph.famous("petersen")
> g1 <- graph(c(0, 1), directed = FALSE)
> g2 <- graph.disjoint.union(g1, g1)
> g3 <- graph.disjoint.union(g, g1)
> mfr(g1)

$bettis
[1] 1 1

$graded
     [,1] [,2]
[1,]    1    0
[2,]    0    1

$reg
[1] 2

$pd
[1] 1

$punted
[1] 0

attr(,"class")
[1] "mfr"   "exact"

> mfr(g2)

$bettis
[1] 1 2 1

$graded
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    1

$reg
[1] 3

$pd
[1] 2
```

```
$punted
[1] 0

attr(,"class")
[1] "mfr"   "exact"
```

```
> mfr(g3)
```

```
$bettis
[1]   1  16  60 132 187 160  80  24   4

$graded
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
[1,]    1    0    0    0    0    0    0    0    0
[2,]    0   16   30   10    0    0    0    0    0
[3,]    0    0   30  102   90   30    0    0    0
[4,]    0    0    0   20   92  110   50    4    0
[5,]    0    0    0    0    5   20   30   20    4

$reg
[1] 5

$pd
[1] 8

$punted
[1] 0

attr(,"class")
[1] "mfr"   "exact"
```

It is a fact that $\beta_{2,2}$ is always the size of the graph (the number of edges) and that the other Betti numbers compute the number of certain types of induced subgraphs, but it is not trivial to figure out which subgraphs are counted for each Betti number.

Define an angle to be a graph on three vertices with two edges, and a triangle is K_3, the graph on three vertices with three edges. Finally, define a *bars* graph to be a graph isomorphic to `g2` above: a two component graph on 4 vertices in which each component is a two vertex graph with one edge. Write $\#\rangle$ for the number of induced angles in a graph, $\#\Delta$ for the number of induced triangles, and $\#||$ for the number of induced bars. It turns out that: $\beta_{2,3} = \#\rangle + 2\#\Delta$ and $\beta 3, 3 = \#||$.

```
> g <- graph.famous("petersen")
> g1 <- graph(c(0, 1), directed = FALSE)
> h <- graph.disjoint.union(g, g1)
> mfr(g)
```

```
$bettis
[1]   1  15  45  87 100  60  20   4

$graded
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    0    0    0    0    0    0    0
[2,]    0   15   30   10    0    0    0    0
[3,]    0    0   15   72   80   30    0    0
[4,]    0    0    0    5   20   30   20    4

$reg
[1] 4

$pd
[1] 7

$punted
[1] 0

attr(,"class")
[1] "mfr"   "exact"
```

```
> mfr(h)
```

```
$bettis
[1]   1  16  60 132 187 160  80  24   4

$graded
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
[1,]    1    0    0    0    0    0    0    0    0
[2,]    0   16   30   10    0    0    0    0    0
[3,]    0    0   30  102   90   30    0    0    0
[4,]    0    0    0   20   92  110   50    4    0
[5,]    0    0    0    0    5   20   30   20    4

$reg
[1] 5

$pd
[1] 8

$punted
[1] 0

attr(,"class")
[1] "mfr"   "exact"
```

Note that we have doubled the number of bars without effecting the number of angles or triangles in going from the Petersen graph to `h`. It is clear that we have also added quite a bit of structure to the rest of the matrix, indicating that the other Betti numbers are counting complicated combinations of induced subgraphs some of which are not connected (like "bars").

4.1 The Database `mfrDB`

The package comes with a database with a number of MFRs precomputed. This makes it easier to calculate the MFR of small graphs. It is also useful for larger graphs: when the recursive algorithm reaches a graph with no splitting edges, it first checks the database to see if the graph has already been pre-computed, allowing it to avoid a call to Singular (or a "punt" to an approximation).

The database contains (mostly) graphs which require Singular for the computation of the minimal free resolution. There are 1951 graphs in the database, of which 1943 required Singular (the rest are just there for convenience). See `create.mfr.database` for more detail.

For graphs in the database, the attribute `punted` is set to 0, even if Singular had to be called in order to compute the MFR for the graph originally. As we saw above, the Petersen graph is in the database and the MFR indicates that Singular was not called. However, we can check the database directly and see that Singular had to be called in order to compute the MFR:

```
> g <- graph.famous("petersen")
> mfrDB$mfrs[which.DB(g)]

[[1]]
$bettis
[1]   1  15  45  87 100  60  20   4

$graded
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    0    0    0    0    0    0    0
[2,]    0   15   30   10    0    0    0    0
[3,]    0    0   15   72   80   30    0    0
[4,]    0    0    0    5   20   30   20    4

$reg
[1] 4

$pd
[1] 7

$punted
[1] 1
```

```
attr(,"class")
[1] "mfr"   "exact"
```

Thus, Singular was called once on the graph in order to compute the MFR. It is easy to see that the Petersen graph contains no splitting edges, and is not one of the special cases (see Section 4.2), which is why it immediately fails all the tests and must resort to a call to Singular.

4.2 Special Cases

There are a number of special cases that can be computed quickly. Most of these take the parameter(s) that defines the graph and return the MFR for that graph.

- Complete graphs.

```
> MFRComplete(11)

$bettis
 [1]    1   55  330  990 1848 2310 1980 1155  440   99   10

$graded
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
[1,]    1    0    0    0    0    0    0    0    0     0     0
[2,]    0   55  330  990 1848 2310 1980 1155  440    99    10

$reg
[1] 2

$pd
[1] 10

$punted
[1] 0

attr(,"class")
[1] "mfr"   "exact"
```

- Complete bipartite graphs.

```
> MFRCompleteBipartite(6, 5)

$bettis
 [1]   1  30 135 310 455 461 330 165  55  11   1

$graded
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
[1,]    1    0    0    0    0    0    0    0    0     0     0
```

```
[2,]    0   30  135  310  455  461  330  165   55    11     1

$reg
[1] 2

$pd
[1] 10

$punted
[1] 0

attr(,"class")
[1] "mfr"   "exact"
```

- Cycles.

```
> MFRCycle(11)

$bettis
[1]  1 11 44 88 99 66 22  1

$graded
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    0    0    0    0    0    0    0
[2,]    0   11   11    0    0    0    0    0
[3,]    0    0   33   66   33    0    0    0
[4,]    0    0    0   22   66   66   22    0
[5,]    0    0    0    0    0    0    0    1

$reg
[1] 5

$pd
[1] 7

$punted
[1] 0

attr(,"class")
[1] "mfr"   "exact"
```

- Paths.

```
> MFRPath(11)

$bettis
[1]  1 10 37 69 72 43 13  1
```

```
$graded
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    0    0    0    0    0    0    0
[2,]    0   10    9    0    0    0    0    0
[3,]    0    0   28   49   21    0    0    0
[4,]    0    0    0   20   50   40   10    0
[5,]    0    0    0    0    1    3    3    1

$reg
[1] 5

$pd
[1] 7

$punted
[1] 0

attr(,"class")
[1] "mfr"   "exact"
```

- Stars.

```
> MFRStar(11)

$bettis
 [1]   1  10  45 120 210 252 210 120  45  10   1

$graded
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
[1,]    1    0    0    0    0    0    0    0    0     0     0
[2,]    0   10   45  120  210  252  210  120   45    10     1

$reg
[1] 2

$pd
[1] 10

$punted
[1] 0

attr(,"class")
[1] "mfr"   "exact"
```

- Wheels with an odd number of vertices.

```
> MFRWheel(11)

$bettis
 [1]   1  20  90 215 325 337 250 131  46  10   1

$graded
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
[1,]    1    0    0    0    0    0    0    0    0     0     0
[2,]    0   20   65  130  210  252  210  120   45    10     1
[3,]    0    0   25   75   75   25    0    0    0     0     0
[4,]    0    0    0   10   40   60   40   11    1     0     0

$reg
[1] 4

$pd
[1] 10

$punted
[1] 0

attr(,"class")
[1] "mfr"   "exact"
```

- Graphs whose complement is chordal. Note that complete graphs are a special case of these.

```
> set.seed(5234)
> g <- rchordal(7, 12)
> h <- graph.complementer(g)
> mfr(h)

$bettis
[1]  1  9 17 12  3

$graded
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    9   17   12    3

$reg
[1] 2

$pd
[1] 4

$punted
```

```
[1] 0

attr(,"class")
[1] "mfr"   "exact"
```

It is important to note that `chordal.comp.mfr` *does not check that the complement of the graph is chordal*! It assumes that you have done this check prior to calling the function. Thus it is best to let `mfr` take care of things rather than calling `chordal.comp.mfr` yourself.

```
> mfr(g, check.database = FALSE)

$bettis
[1]  1 12 30 33 18  4

$graded
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    0    0    0    0    0
[2,]    0   12   25   20    7    1
[3,]    0    0    5   13   11    3

$reg
[1] 3

$pd
[1] 5

$punted
[1] 0

attr(,"class")
[1] "mfr"   "exact"

> chordal.comp.mfr(g)

$bettis
[1]  1 12 25 20  7  1

$graded
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    0    0    0    0    0
[2,]    0   12   25   20    7    1

$reg
[1] 2

$pd
```

```
[1] 5

$punted
[1] 0

attr(,"class")
[1] "mfr"   "exact"
```

This last is *wrong*. You must always check, using `is.chordal.comp(g)` or equivalently `is.chordal(graph.complementer(g))`, to ensure that `chordal.comp.mfr` is appropriate. The `mfr` code does this for you.

5 Partial Results

There is a small amount of code for producing partial Betti number results. In particular, the functions `betti2`, `betti3` and `linearStrand`. `betti2` just the size of the graph ($\beta_{2,2}$), and `betti3` is the third column of the graded Betti numbers, corresponding to the triangle/angle and bars calculations discussed above.

The linear strand is the second (first non-trivial) row of the graded Betti numbers. There are two versions: an exact calculation, and a lower bound. The code always produces the exact result if the graph does not contain an induced 4-cycle. If it does, then the result depends on whether `linearStrand` was called with `exact=TRUE`. If so, it runs a (very slow) algorithm to compute the exact linear strand. If not, it runs a (slightly less slow, but still very slow) algorithm to obtain a lower bound. It is almost always going to be faster than computing the full MFR, though.

```
> set.seed(3452356)
> g <- rchordal(10, 20)
> h <- graph.complementer(g)
> mfr(g)

$bettis
[1]   1  20  80 163 200 152  70  18   2

$graded
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
[1,]    1    0    0    0    0    0    0    0    0
[2,]    0   20   60   83   67   33    9    1    0
[3,]    0    0   20   80  133  119   61   17    2

$reg
[1] 3

$pd
```

```
[1] 8

$punted
[1] 0

attr(,"class")
[1] "mfr"   "exact"
```

```
> linearStrand(g)
```

```
Linear Strand of a Minimal Free Resolution:
1 0 0 0 0 20 0 60 0 83 0 67 0 33 0 9 0 1
Betti numbers are exact
```

```
> mfr(h)
```

```
$bettis
[1]   1  25  94 165 164  95  30   4

$graded
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    0    0    0    0    0    0    0
[2,]    0   25   94  165  164   95   30    4

$reg
[1] 2

$pd
[1] 7

$punted
[1] 0

attr(,"class")
[1] "mfr"   "exact"
```

```
> linearStrand(h)
```

```
Linear Strand of a Minimal Free Resolution:
1 0 0 25 0 94 0 165 0 164 0 95 0 30 0 4
Betti numbers are exact
```

6 Scan Statistics

In spite of all the effort to implement good algorithms to compute the MFR (both in the R/C code here and the Singular package) the fact is, the calculation of the MFR for graphs with more than a few tens of vertices is impractical.

One way around this, is to give up on finding the MFR of the full graph, and instead think locally. A scan statistic on a graph is a graph invariant applied to a local region (in our case, the 1-neighborhood of a vertex). The local region is chosen to have some maximal property (such as size of the induced subgraph), and `scanMFR` returns the the MFR of this induced subgraph.

There's good news and bad news about scan statistics. First, the induced subgraph is the star of the induced subgraph of the open neighborhood, and so we get a little savings there (the MFR of the star of a graph g can be computed easily from the MFR of g). The bad news is that unless these subgraphs are "small" we are still doomed.

```
> set.seed(5383)
> g <- erdos.renyi.game(n = 100, p = 0.1)
> m <- scanMFR(g)
> m

Scan method: size
Minimal Free Resolution:
Total betti numbers:
         1         36         271          1186          3601          8215          14770

Graded:         1         0         0         0         0         0         0         0
         0         36         190          742          2423          6210          12383
         0         0         81          309          485          425          239          89
         0         0         0          135          595          1082          1070          638
         0         0         0          0          98          472          945          1023
         0         0         0          0          0          26          133          286
```

Note that in the above example, $\beta_{2,2} = 36 < 100$. Even here there are limits. For example, in the above example, if we increase n from 100 to 1000, Singular must be called, and the estimated time to complete is sometime after the heat-death of the universe. We can attempt to compute just a few terms in the linear strand, but even here we may be biting off far more than we can chew.