Counting on networks

Jake Hofman

April 7, 2017

Contents


```
library(igraph)
```

```
##
## Attaching package: 'igraph'
```

```
## The following objects are masked from 'package:stats':
##
##     decompose, spectrum
```

```
## The following object is masked from 'package:base':
##
##     union
```

```
library(tidyverse)
```

```
## Loading tidyverse: ggplot2
## Loading tidyverse: tibble
## Loading tidyverse: tidyr
## Loading tidyverse: readr
## Loading tidyverse: purrr
## Loading tidyverse: dplyr
```

```
## Conflicts with tidy packages ----------------------------------------------
```

```
## as_data_frame(): dplyr, tibble, igraph
## compose():       purrr, igraph
## crossing():      tidyr, igraph
## filter():        dplyr, stats
## groups():        dplyr, igraph
## lag():           dplyr, stats
## simplify():      purrr, igraph
```

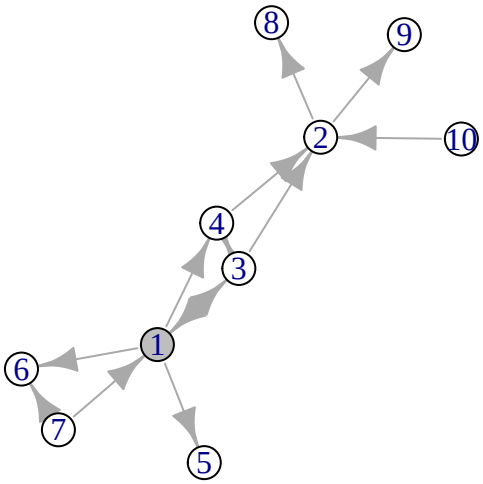
Single source shortest path

```
# function to calculate distance from a single source to all other nodes via BFS
single_source_shortest_path <- function(G, source, plot=F) {
  # initialize all nodes to be inifinitely far away
  # and the source to be at zero
  dist <- rep(Inf, vcount(G))
  names(dist) <- V(G)$name
  dist[source] <- 0

  # initialize the current boundary to be the source node
  curr_boundary <- c(source)

  # explore boundary as long as it's not empty
  while (length(curr_boundary) > 0) {
    if (plot)
      plot_bfs(G, dist, curr_boundary)

    # create empty list for new boundary
    next_boundary <- c()

    # loop over nodes in current boundary
    for (node in curr_boundary)
      # loop over their undiscovered neighbors
      for (neighbor in neighbors(G, node))
        if (!is.finite(dist[neighbor])) {
          # set the neighbor's distance
          dist[neighbor] = dist[node] + 1
          # add the neighbor to the next boundary
          next_boundary <- c(next_boundary, neighbor)
        }

    # update the boundary
    curr_boundary <- unique(next_boundary)
  }

  if (plot)
    plot_bfs(G, dist, curr_boundary)

  dist
}

# helper function to plot bfs iteration
plot_bfs <- function(G, dist, curr_boundary) {
  set.seed(42)
  discovered <- which(is.finite(dist))
  colors <- rep('white', vcount(G))
  colors[discovered] <- 'black'
  colors[curr_boundary] <- 'grey'
  plot.igraph(G, vertex.color=colors)
  print(sprintf('bfs iteration %d', max(dist[discovered])))
  print(sprintf('discovered (black): %s', paste(discovered, collapse=" ")))
  print(sprintf('current boundary (grey): %s', paste(curr_boundary, collapse=" ")))
```

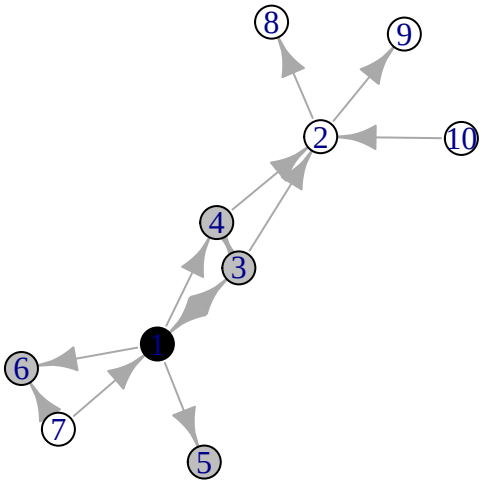
```
  line <- readline('hit enter to continue')
}
```

Example

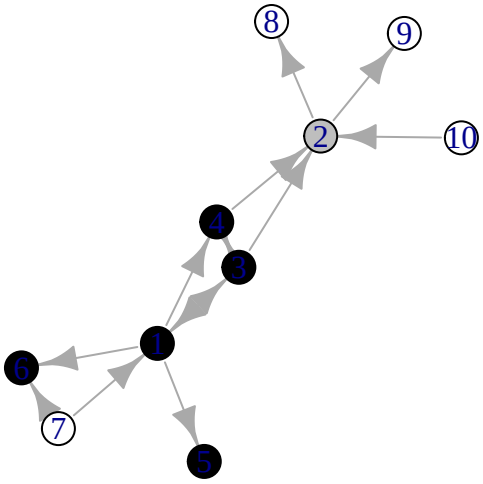
```
# create toy graph
G <- graph(c(1,4,1,5,1,6,7,6,7,1,3,1,1,3,4,2,3,2,2,8,10,2,2,9,3,4,4,3), directed=T)

# find distances from node 1
single_source_shortest_path(G, 1, T)
```



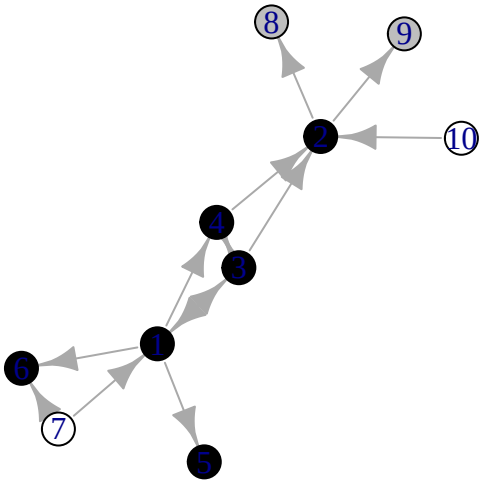

```
## [1] "bfs iteration 0"
## [1] "discovered (black): 1"
## [1] "current boundary (grey): 1"
## hit enter to continue
```



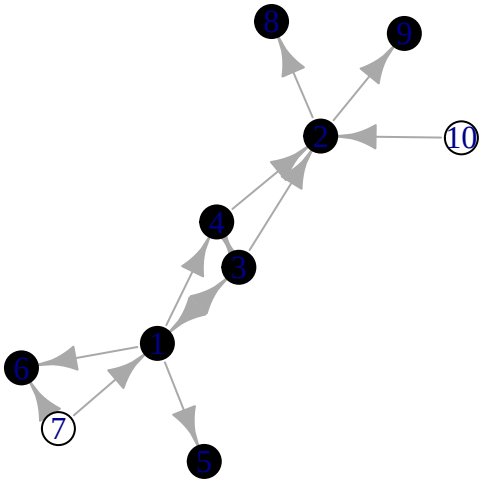

```
## [1] "bfs iteration 1"
## [1] "discovered (black): 1 3 4 5 6"
## [1] "current boundary (grey): 3 4 5 6"
## hit enter to continue
```




```
## [1] "bfs iteration 2"
## [1] "discovered (black): 1 2 3 4 5 6"
## [1] "current boundary (grey): 2"
## hit enter to continue
```



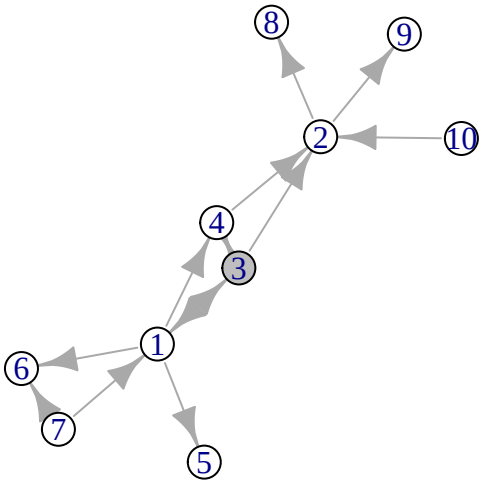

```
## [1] "bfs iteration 3"
## [1] "discovered (black): 1 2 3 4 5 6 8 9"
## [1] "current boundary (grey): 8 9"
## hit enter to continue
```



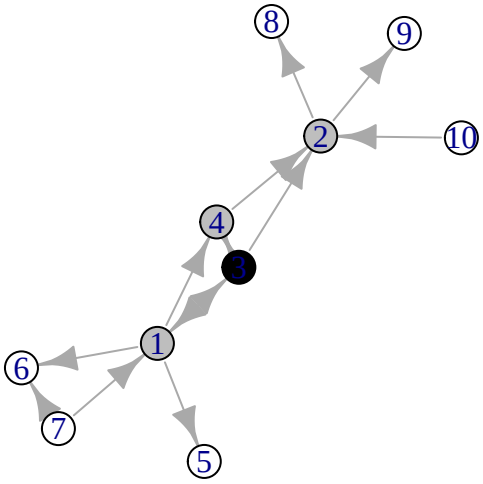

```
## [1] "bfs iteration 3"
## [1] "discovered (black): 1 2 3 4 5 6 8 9"
## [1] "current boundary (grey): "
## hit enter to continue
```

```
##  [1]   0   2   1   1   1   1 Inf   3   3 Inf
```

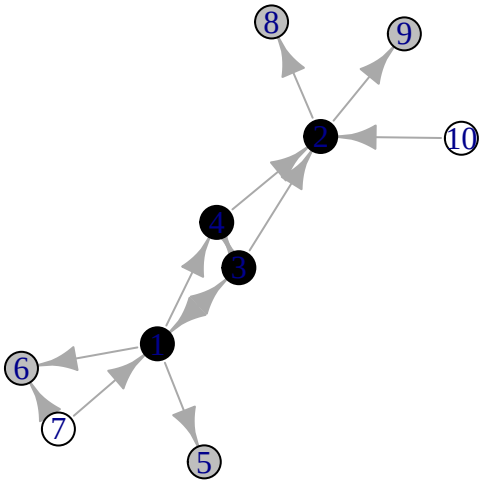
```
# find distances from node 3
single_source_shortest_path(G, 3, T)
```



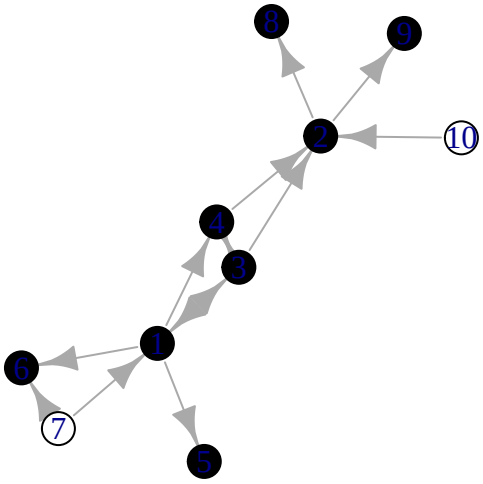

```
## [1] "bfs iteration 0"
## [1] "discovered (black): 3"
## [1] "current boundary (grey): 3"
## hit enter to continue
```




```
## [1] "bfs iteration 1"
## [1] "discovered (black): 1 2 3 4"
## [1] "current boundary (grey): 1 2 4"
## hit enter to continue
```




```
## [1] "bfs iteration 2"
## [1] "discovered (black): 1 2 3 4 5 6 8 9"
## [1] "current boundary (grey): 5 6 8 9"
## hit enter to continue
```




```
## [1] "bfs iteration 2"
## [1] "discovered (black): 1 2 3 4 5 6 8 9"
## [1] "current boundary (grey): "
## hit enter to continue
```

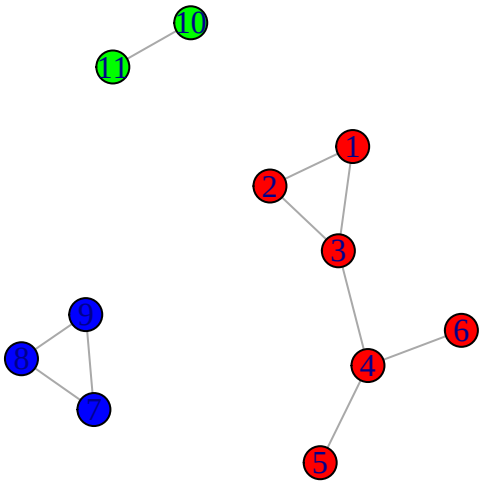
```
##  [1]   1   1   0   1   2   2 Inf   2   2 Inf
```

Connected components

```
# function to compute connected components of a graph via BFS
connected_components <- function(G) {
  components <- rep(NA, vcount(G))

  label <- 1

  # loop until all nodes are assigned to a component
  while (any(is.na(components))) {
    # sample an unassigned node
    source <- sample(which(is.na(components)), 1)

    # do a bfs from this source
    dist <- single_source_shortest_path(G, source)

    # label reachable nodes
    components[is.finite(dist)] <- label

    # increment label
    label <- label + 1
  }
  components
}
```

Example

```r
# create toy graph with multiple connected components
G <- graph(c(1,2,1,3,2,3,3,4,4,5,4,6,7,8,7,9,8,9,10,11), directed=F)

# find and plot connected components
components <- connected_components(G)
colors <- rainbow(max(components))
plot(G, vertex.color=colors[components])
```



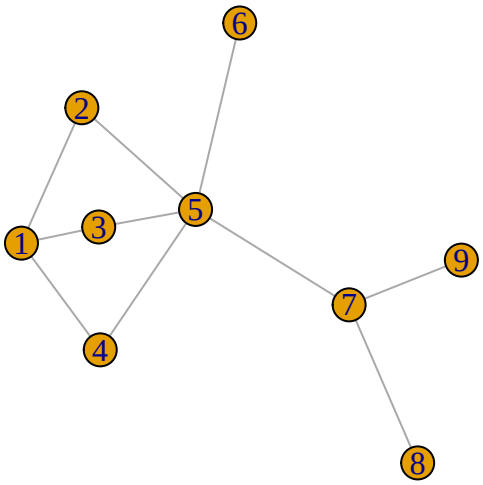
Mutual friends

```r
# function to count the number of mutual friends between every pair of nodes
mutual_friends <- function(G) {
  # initialize an empty matrix to store number of mutual friends between pairs of nodes
  num_nodes <- vcount(G)
  mutual_friends <- matrix(0, nrow=num_nodes, ncol=num_nodes)

  # loop over each node
  for (node in 1:num_nodes) {
    # get this node's list of friends
    friends <- neighbors(G, node)

    # add a count of 1 between all pairs of the node's friends
    for (i in friends)
      for (j in friends)
        mutual_friends[i, j] = mutual_friends[i, j] + 1
  }

  # make the output readable with column names
  dimnames(mutual_friends) <- list(row=V(G)$name, col=V(G)$name)
  diag(mutual_friends) <- NA
  mutual_friends
}

# function to get "people you might know" based on mutual friend counts
people_you_might_know <- function(M, node) {
  recs <- c(which(M[node,] == max(M[node,], na.rm=T)))
```

```
  sprintf('node %d might know node(s) %s', node, paste(recs, collapse=" and "))
}
```

Example

```
# create toy graph with open triads
G <- graph(c(1,2,1,3,1,4,2,5,3,5,4,5,5,6,5,7,7,8,7,9), directed=F)

plot(G)
```




```
M <- mutual_friends(G)
M
```

```
##        col
## row     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
##    [1,]   NA    0    0    0    3    0    0    0    0
##    [2,]    0   NA    2    2    0    1    1    0    0
##    [3,]    0    2   NA    2    0    1    1    0    0
##    [4,]    0    2    2   NA    0    1    1    0    0
##    [5,]    3    0    0    0   NA    0    0    1    1
##    [6,]    0    1    1    1    0   NA    1    0    0
##    [7,]    0    1    1    1    0    1   NA    0    0
##    [8,]    0    0    0    0    1    0    0   NA    1
##    [9,]    0    0    0    0    1    0    0    1   NA
```

```
people_you_might_know(M, 1)
```

```
## [1] "node 1 might know node(s) 5"
```

```
people_you_might_know(M, 2)
```

```
## [1] "node 2 might know node(s) 3 and 4"
```

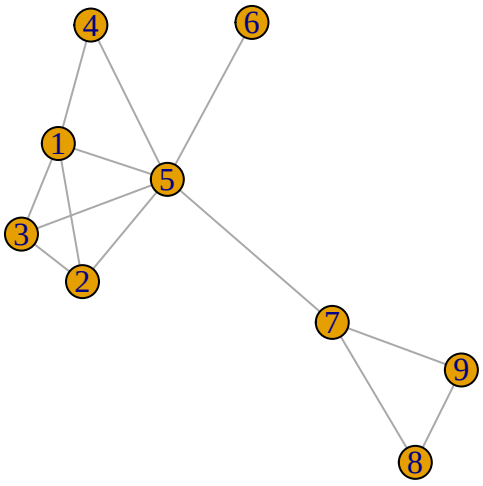
Counting triangles

```
# function to count triangles
count_triangles <- function(G) {
```

```
  num_nodes <- vcount(G)

  # initialize a counter for the number of triangles at each node
  triangles <- rep(0, num_nodes)

  # loop over each node
  for (node in 1:num_nodes) {
    # get this node's list of friends
    friends <- neighbors(G, node)

    # add a count of 1 for each pair of the node's friends that are connected
    for (i in friends)
      for (j in friends)
        if (are.connected(G, i, j))
          triangles[node] = triangles[node] + 1
  }

  # make the output readable with column names
  names(triangles) <- V(G)$name
  triangles / 2.0
}
```

Example

```
# create toy graph with some closed triads
G <- graph(c(1,2,1,3,1,4,2,5,3,5,4,5,5,6,5,7,7,8,7,9,1,5,2,3,8,9), directed=F)

# plot and count triangles
plot(G)
```




```
triangles <- count_triangles(G)
triangles
```

```
## [1] 4 3 3 1 4 0 1 1 1
```

```
# compute clustering coefficient
k <- degree(G)
```

```r
sum(triangles) / sum(k * (k-1) / 2)
```

```
## [1] 0.5454545
```