Implementation of DFS and BFS

1. Create a function that performs Depth First Search (DFS) from a specified node of the graph.
   - The function should provide an implementation of the iterative DFS algorithm.
   - Visualize the depth first search tree and the implied preorder, by highlighting the edges in a different color.

Note 1: You may want to think of creating a tree visualization function which you can re-use in questions 1 and 2.

Note 2: Develop the code on paper before starting the implementation. This will save time and reduce the number of possible errors.

Hints:

procedure dfs2(v)

$P \leftarrow empty\_stack$

$mark[v] \leftarrow 1$

push v unto P

while P is not empty

while there exists a node w adjacent to **top**(P)

such that $mark[w] \neq 1$ **do**

$mark[w] \leftarrow 1$

push w unto P #w is the new top of P

pop P

Looking at the procedure, we see that we need:

- implementation of a stack structure

- an array $mark[]$, initialized with numbers different from 1 (say, 0)

- ways to identify the neighbors of a specified node in a given graph

The stack data structure is already implemented in R, see package `dequer`

```
install.packages("dequer")
```

```
library(dequer)
```

The three functions of interest are `stack()`,`pop()`, and `push()`

```
s <- stack() #creates a stack s
push(s, 1) #pushes 1 onto stack s
push(s, 2) #pushes 2 onto stack s
pop(s) #pops the top element of the stacks
for (i in 4:7){
push(s, i)
}
str(s) #displays the internal structure of an R object
```

Implementing top

```
top <- function(x){
     t <- pop(x)
     push(x, t)
     return(t)
}
```

Let the graph be stored in an igraph object named g

```
g <- make_chordal_ring(15, matrix(c(3, 12, 4, 7, 8, 11), nrow =
2))
```

To create an array $mark[]$ with as many elements as there are nodes in the graph

```
num_nodes <- length(V(g))
mark <- rep(0, num_nodes)
```

Implementing the function

```
#put the pieces above together
#first on paper
```

Tracking the edges included in the tree

```
h <- make_ring(10)
ei <- get.edge.ids(h, c(1,2, 4,5))
E(h)[ei]

## non-existant edge
get.edge.ids(h, c(2,1, 1,4, 5,4))
```

Trace the edges by updating the code we provided above

Insert the following lines of code appropriately

```
edges <- c() # empty vector of edges
edges <- c(edges, top(p), w)
return(edges)
```

2. Create a function that performs Breadth First Search (BFS) from a given node of the graph.

    - Visualize the breadth first search tree by highlighting the edges in a different color.

Hint:

procedure bfs(v)

$Q \leftarrow empty\ queue$

$mark[v] \leftarrow 1\#\ 1\ means\ visited$

enqueue v into Q

while Q is not empty **do**

$u \leftarrow first(Q)$

dequeue u from Q

for each node w adjacent to u **do**

if $mark[w] \neq 1$ **then**

$mark[w] \leftarrow 1$

enqueue w into Q

Note that `queue()` can be used with the operations `pushback()` and `pop()` for enqueue and dequeue.

3. Use the DFS or BFS procedures to count the number of connected components.

Note: The pseudocode provided created DFS and BFS for a component which contains the node v provided as input. Therefore, you should also return the marked nodes and check if there are any left unvisited.

Homework

1. Implement an algorithm that determines the lengths of the shortest paths for any pair of nodes in a given graph based on BFS.
2. Implement an algorithm that determines the center of the graph based on your answer to homework question 2. The eccentricity of a node is given by the longest shortest path to any other node. The center of the graph is composed of all nodes of minimum eccentricity. Visualize the center by highlighting its nodes in a different color.