simple_networks_nb

Sid Satya

Description

This is a library of functions used for analysis in the Misinformation Research Project! We have functionality added for the following types of networks:

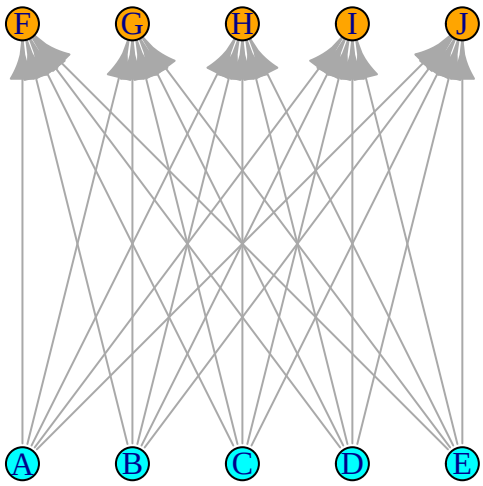
1. Bipartite
2. Diamond
3. Spade
4. Cycle
5. Pair

Furthermore, we have added code that will update the networks above based on the prior and posterior beliefs of each node, as well as a method to simulate the updating over several rounds.

Examples are shown below.

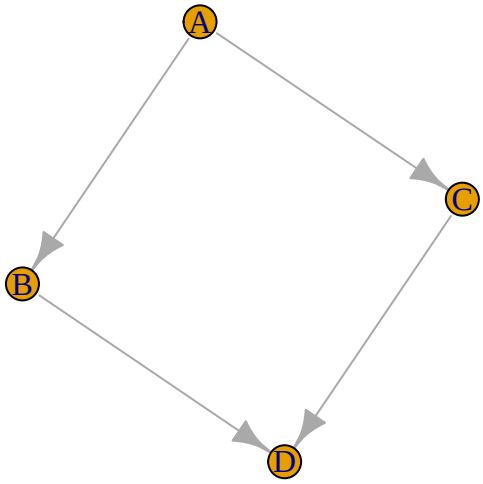
```
#' Creates a bipartite graph out of two distinct sets of nodes
#'
#' @description
#' Takes in two different sets of nodes that do not overlap
#' and creates a directed mapping between them. Provides the option
#' to make a complete bipartite graph out of both sets of nodes
#'
#' @param rd1 The first set of nodes, these nodes will have outgoing edges
#' @param rd2 The second set of nodes, will have incoming edges
#' @param make_complete Boolean. If true, the function will return a complete
#'  bipartite graph between both sets. If false, the mapping will be simple.
#'
#' @return An igraph object containing the nodes and edges.
#'
#' @example
#' rd1 = c("A", "B", "C", "D")
#' rd2 = c("E", "F", "G", "H")
#'
#' g <- bipartite(rd1, rd2)
#' Will return a graph with edges A->E, B->F, C->G, D->H.
#'
#' g <- bipartite(rd1, rd2, make_complete=TRUE)
#' Will return a graph with edges A->E,A->F,A->G,A->H,B->E,B->F,B->G,B->H,C->E,
#' C->F,C->G,C->H,D->E,D->F,D->G,D-H.
```

```
bipartite <- function(rd1, rd2, make_complete=FALSE) {

  # TODO: add functionality for complete bipartite
  if (make_complete) {
    num_nodes = length(rd2)

    from_nodes <- c()
    for (i in 1:length(rd1)) {
      from_nodes <- c(from_nodes, rep(rd1[i], num_nodes))
    }
    edgeList <- data.frame(S1=from_nodes,S2=rd2)
  } else {
    # simple, edges defined as nodes are passed in
    edgeList <- data.frame(S1=rd1, S2=rd2)
  }


  # attributes to add: round, opinion
  g <- graph.empty()


  g <- add.vertices(g,nv=length(rd1),attr=list(name=paste0(rd1),
                                               type=rep(TRUE,length(rd1)),
                                               round=rep('R1', length(rd1)),
                                               prior=DEFAULT_PRIOR,
                                               posterior=DEFAULT_POSTERIOR))

  g <- add.vertices(g,nv=length(rd2),attr=list(name=paste0(rd2),
                                type=rep(FALSE,length(rd2)),
                                round=rep('R2', length(rd1)),
                                prior=DEFAULT_PRIOR,
                                posterior=DEFAULT_POSTERIOR))

  # we need to turn edgeList into a vector (and using names instead of indexes)
  edgeListVec <- as.vector(t(as.matrix(data.frame(S1=paste0(edgeList$S1),
                               S2=paste0(edgeList$S2)))))
  g <- add.edges(g,edgeListVec)
  return(g)
}

rd1 = c("A", "B", "C", "D", "E")
rd2 = c("F", "G", "H", "I", "J")
g <- bipartite(rd1, rd2, make_complete=TRUE)

# set layout
l <- layout_as_bipartite(g)
plot(g, layout=l, vertex.color=c("orange","cyan")[V(g)$type+1])
```



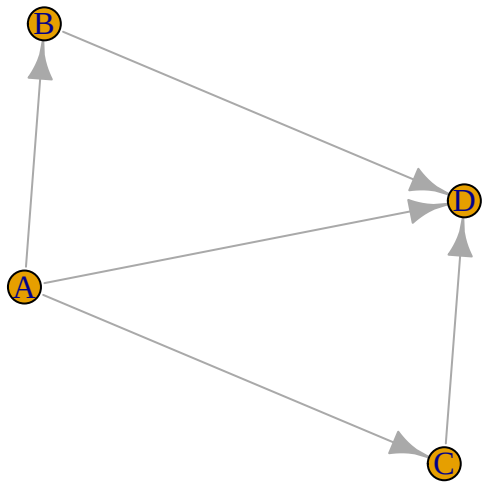

```
#' Creates a diamond graph out of three distinct sets of nodes
#'
#' @description
#' Takes in three different sets of nodes that do not overlap
#' and creates a directed mapping between them in a diamond form. Always
#' has one node as the top node and one node as the bottom node. It can take
#' any number of nodes as the middle layer between the top and bottom node.
#'
#' @param rd1 The first set of nodes, must always have a length of 1. Will have
#'  only outgoing edges to rd2.
#' @param rd2 The second set of nodes, will have incoming edges from rd1 and
#'  outgoing edges to rd3.
#' @param rd3 The third set of nodes, must always have a length of 1. Will have
#'  only incoming edges from rd2.
#'
#' @return An igraph object containing the nodes and edges.
#'
#' @example
#' rd1 = c("A")
#' rd2 = c("B", "C")
#' rd3 = c("D")
#' g <- diamond(rd1, rd2, rd3)
#' Will return a graph with edges A->B, A->C, B->D, C->D.

diamond <- function(rd1, rd2, rd3) {
  stopifnot(length(rd1) == 1)
```

```
  stopifnot(length(rd3) == 1)

  rd2len = length(rd2)

  from_nodes <- c(rep(rd1[1], rd2len), rd2)
  to_nodes <- c(rd2)

  # replicate the round 3 nodes the correct number of times
  for (i in 1:rd2len) {
    to_nodes <- c(to_nodes, rd3)
  }

  edgeList <- data.frame(S1=from_nodes,S2=to_nodes)

  # attributes to add: opinion
  g <- graph.empty()
  g <- add.vertices(g,nv=length(rd1),attr=list(name=paste0(rd1),
                                               type=rep(TRUE,length(rd1)),
                                               round=rep('R1', length(rd1)),
                                               prior=DEFAULT_PRIOR,
                                               posterior=DEFAULT_POSTERIOR))

  g <- add.vertices(g,nv=length(rd2),attr=list(name=paste0(rd2),
                                  type=rep(FALSE,length(rd2)),
                                  round=rep('R2', length(rd1)),
                                  prior=DEFAULT_PRIOR,
                                  posterior=DEFAULT_POSTERIOR))

  g <- add.vertices(g,nv=length(rd3),attr=list(name=paste0(rd3),
                                  type=rep(FALSE,length(rd3)),
                                  round=rep('R3', length(rd3)),
                                  prior=DEFAULT_PRIOR,
                                  posterior=DEFAULT_POSTERIOR))


  # we need to turn edgeList into a vector (and using names instead of indexes)
  edgeListVec <- as.vector(t(as.matrix(data.frame(S1=paste0(edgeList$S1),
                               S2=paste0(edgeList$S2)))))
  g <- add.edges(g,edgeListVec)
  return(g)
}

rd1 = c("A")
rd2 = c("B", "C")
rd3 = c("D")
g <- diamond(c("A"), c("B", "C"), c("D"))
plot(g)
```



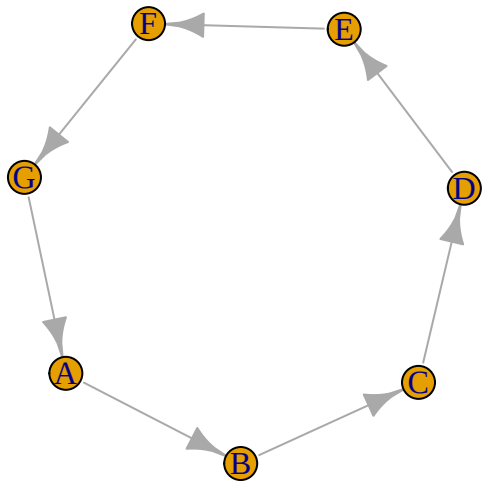

```
#' Creates a spade graph out of three distinct sets of nodes
#'
#' @description
#' Takes in three different sets of nodes that do not overlap
#' and creates a directed mapping between them in a diamond form with a direct
#' edge from the top node to the bottom node. Always has one node as the top
#' node and one node as the bottom node. It can take any number of nodes as the
#' middle layer between the top and bottom node.
#'
#' @param rd1 The first set of nodes, must always have a length of 1. Will have
#'  only outgoing edges to rd2 only and rd3.
#' @param rd2 The second set of nodes, will have incoming edges from rd1 and
#'  outgoing edges to rd3.
#' @param rd3 The third set of nodes, must always have a length of 1. Will have
#'  only incoming edges from rd1 and rd2.
#'
#' @return An igraph object containing the nodes and edges.
#'
#' @example
#' rd1 = c("A")
#' rd2 = c("B", "C")
#' rd3 = c("D")
#' g <- diamond(rd1, rd2, rd3)
#' Will return a graph with edges A->B, A->C, A->D, B->D, C->D.

spade <- function(rd1, rd2, rd3) {
```

```
  stopifnot(length(rd1) == 1)
  stopifnot(length(rd3) == 1)

  # take first node as head, last node as tail, rest as middle
  # connect head and tail to all middle nodes, and edge from head to tail
  rep_nodes = length(rd2) + length(rd3)
  rd2len = length(rd2)

  from_nodes <- c(rep(rd1[1], rep_nodes), rd2)
  to_nodes <- c(rd2)

  # replicate the round 3 nodes the correct number of times
  for (i in 1:rep_nodes) {
    to_nodes <- c(to_nodes, rd3)
  }

  edgeList <- data.frame(S1=from_nodes,S2=to_nodes)

  # attributes to add: opinion
  g <- graph.empty()
  g <- add.vertices(g,nv=length(rd1),attr=list(name=paste0(rd1),
                                               type=rep(TRUE,length(rd1)),
                                               round=rep('R1', length(rd1)),
                                               prior=DEFAULT_PRIOR,
                                               posterior=DEFAULT_POSTERIOR))

  g <- add.vertices(g,nv=length(rd2),attr=list(name=paste0(rd2),
                                  type=rep(FALSE,length(rd2)),
                                  round=rep('R2', length(rd1)),
                                  prior=DEFAULT_PRIOR,
                                  posterior=DEFAULT_POSTERIOR))

  g <- add.vertices(g,nv=length(rd3),attr=list(name=paste0(rd3),
                                  type=rep(FALSE,length(rd3)),
                                  round=rep('R3', length(rd3)),
                                  prior=DEFAULT_PRIOR,
                                  posterior=DEFAULT_POSTERIOR))


  # we need to turn edgeList into a vector (and using names instead of indexes)
  edgeListVec <- as.vector(t(as.matrix(data.frame(S1=paste0(edgeList$S1),
                               S2=paste0(edgeList$S2)))))
  g <- add.edges(g,edgeListVec)
  return(g)
}

rd1 <- c("A")
rd2 <- c("B", "C")
rd3 <- c("D")
g <- spade(rd1, rd2, rd3)
plot(g)
```



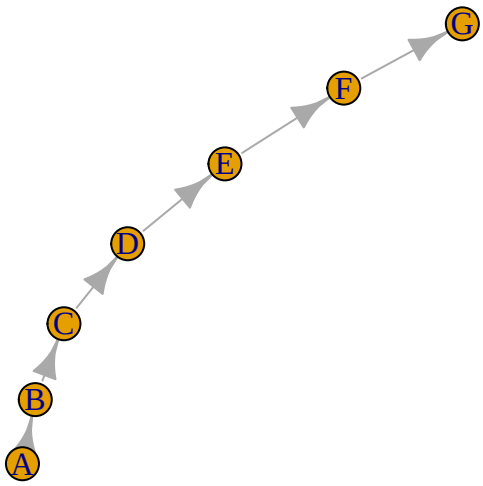

```
#' Creates a large cycle out of a single set of nodes
#'
#' @description
#' Take in a single set of nodes and returns a circular igraph object with each
#' node holding a directed edge to its neighbor. Forms a cycle by holding a
#' directed edge from the last node to the first node.
#'
#' @param nodes The total list of nodes to form the cycle.
#'
#' @return An igraph object containing the nodes and edges.
#'
#' @example
#' nodes <- c("A", "B", "C", "D")
#' g <- cycle(nodes)
#' Will return a graph with edges A->B, B->C, C->D, D->A.


cycle <- function(nodes) {
  # A B C D, A--B, B--C, C--D, D--A,
  from_nodes = nodes
  to_nodes = c(tail(nodes, length(nodes)-1), nodes[1])
  edgeList <- data.frame(S1=from_nodes,S2=to_nodes)


  g <- graph.empty()
```

```
  for (i in 1:length(nodes)) {
    g <- add.vertices(g, nv=length(1), attr=list(name=paste0(nodes[i]),
                                                 type=rep(TRUE, 1),
                                                 round=rep(paste0("R", i)),
                                                 prior=DEFAULT_PRIOR,
                                                 posterior=DEFAULT_POSTERIOR))
  }

   # we need to turn edgeList into a vector (and using names instead of indexes)
  edgeListVec <- as.vector(t(as.matrix(data.frame(S1=paste0(edgeList$S1),
                               S2=paste0(edgeList$S2)))))
  g <- add.edges(g,edgeListVec)
  return(g)
}

# Instead of passing in rounds, you can just pass in the full node list,
# this function assumes that each node is in its own round.
node_list = c("A","B","C","D","E","F","G")
g <- cycle(node_list)
plot(g)
```




```
#' Creates a large chain out of a single set of nodes
#'
#' @description
#' Take in a single set of nodes and returns a chain-like igraph object with
```

```r
#' each node holding a directed edge to its neighbor.
#'
#' @param nodes The total list of nodes to form the chain
#'
#' @return An igraph object containing the nodes and edges.
#'
#' @example
#' nodes <- c("A", "B", "C", "D")
#' g <- cycle(nodes)
#' Will return a graph with edges A->B, B->C, C->D.

chain <- function(nodes) {
  node_df <- data.frame(label=nodes)
  from_nodes = nodes[1:(length(nodes)-1)]
  to_nodes = nodes[2:(length(nodes))]
  edgeList <- data.frame(S1=from_nodes,S2=to_nodes)

  g <- graph.empty()

  for (i in 1:length(nodes)) {
    g <- add.vertices(g, nv=length(1), attr=list(name=paste0(nodes[i]),
                                                 type=rep(TRUE, 1),
                                                 round=rep(paste0("R", i)),
                                                 prior=DEFAULT_PRIOR,
                                                 posterior=DEFAULT_POSTERIOR))
  }

   # we need to turn edgeList into a vector (and using names instead of indexes)
  edgeListVec <- as.vector(t(as.matrix(data.frame(S1=paste0(edgeList$S1),
                               S2=paste0(edgeList$S2)))))
  g <- add.edges(g,edgeListVec)
  return(g)
}

# Instead of passing in rounds, you can just pass in the full node list,
# this function assumes that each node is in its own round.
node_list = c("A","B","C","D","E","F","G")
g <- chain(node_list)
plot(g)
```

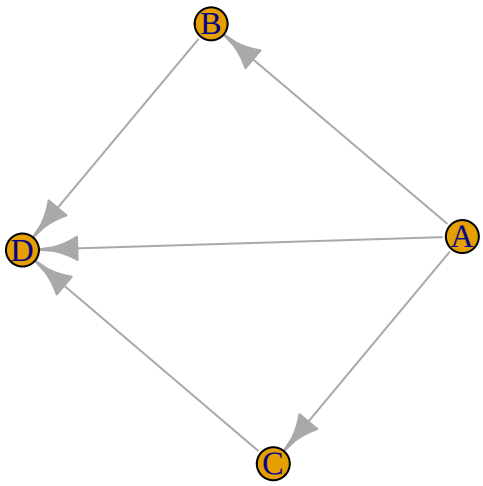



```
# working on this!
funnel <- function(nodes) {
  return(0)
}
```

```
#' Updates each node's posterior to be the average of their neighbors'.
#'
#' @description
#' Take in a graph object and a single node in the graph to update. Will update
#' the node's posterior belief attribute to be equal to the mean of their
#' neighbors' posterior beliefs.
#'
#' @param g An igraph object containing the nodes and edges of the graph of
#'  interest.
#' @param node A single node in the graph.
#'
#' @return An igraph object containing the nodes and edges with updated
#'  posterior values.
#'
#' @example
#' nodes <- c("A", "B", "C", "D")
#' g <- cycle(nodes)
#' g <- update_posterior_pull(g, V(g)["A"])

update_posterior_pull <- function(g, node) {
  # we want to look at incoming edges
```

```
  nbrs <- unlist(neighborhood(g, 1, node, mode="in"))

  # null check
  if (length(nbrs) == 0) {
    return(g)
  }

  # so want to filter out the head node first
  nbrs <- nbrs[2:length(nbrs)]
  # calculate mean posterior of neighbors
  mean_nbr_posterior = 0
  for (i in 1:length(nbrs)) {
    mean_nbr_posterior = mean_nbr_posterior + V(g)[nbrs[i]]$posterior
  }
  mean_nbr_posterior = mean_nbr_posterior/(length(nbrs))

  V(g)[node]$posterior = mean_nbr_posterior
  return(g)
}

#' Finds the willing neighbors of a node given a threshold probability.
#'
#' @description
#' Take in a graph object, a list of nodes that are neighbors of a single node
#' in the graph, and a probability. It will find all willing neighbors by
#' selecting all neighboring nodes with a prior belief that is greater than or
#' equal to the probability passed in.
#'
#' @param g An igraph object containing the nodes and edges of the graph of
#'  interest.
#' @param nbrs A list of nodes in the graph that are neighbors to a single node.
#' @param prob Float, defines the probability threshold for finding willing
#'  neighbors.
#'
#' @return An igraph object containing the nodes and edges with updated
#'  posterior values.

get_willing_neighbors <- function(g, nbrs, cond) {
  willing_nbrs <- V(g)[nbrs][prior >= cond]
  return(willing_nbrs)
}

#' Updates each node's posterior to be the average of their willing neighbors'.
#'
#' @description
#' Take in a graph object, a single node in the graph to update, and a
#' probability. Will update the node's posterior belief attribute to be equal
#' to the mean of all neighbors' posterior beliefs granted that each neighbors'
#' prior belief is greater than or equal to the probability passed in.
#'
#' @param g An igraph object containing the nodes and edges of the graph of
#'  interest.
#' @param node A single node in the graph.
```

```r
#' @param prob Float, defines the probability threshold for finding willing
#'  neighbors.
#'
#' @return An igraph object containing the nodes and edges with updated
#'  posterior values.
#'
#' @example
#' nodes <- c("A", "B", "C", "D")
#' g <- cycle(nodes)
#' prob = 0.8
#' g <- update_posterior_pull(g, V(g)["A"], prob)

update_posterior_push <- function(g, node, prob) {
  # we want to look at incoming edges
  nbrs <- unlist(neighborhood(g, 1, node, mode="in"))

  # null check
  if (length(nbrs) == 0) {
    return(g)
  }

  # filter out the head node
  nbrs <- nbrs[2:length(nbrs)]

  # get all willing neighbors
  willing_nbrs <- get_willing_neighbors(g, nbrs, prob)

  # null check
  if (length(willing_nbrs) == 0) {
    return(g)
  }

  # calculate mean posterior of willing neighbors
  mean_nbr_posterior = 0
  for (i in 1:length(willing_nbrs)) {
    mean_nbr_posterior = mean_nbr_posterior + V(g)[willing_nbrs[i]]$posterior
  }
  mean_nbr_posterior = mean_nbr_posterior/(length(willing_nbrs))

  V(g)[node]$posterior = mean_nbr_posterior
  return(g)
}

rd1 <- c("A")
rd2 <- c("B", "C")
rd3 <- c("D")
g <- spade(rd1, rd2, rd3)
plot(g)
```




```
# test for push
V(g)["A"]$posterior = 0.4
V(g)["B"]$posterior = 0.2
V(g)["C"]$posterior = 0.9
V(g)["A"]$prior = 0.8

g <- update_posterior_push(g, V(g)["D"], 0.8)
V(g)$prior
```

```
## [1] 0.8 0.5 0.5 0.5
```

```
V(g)$posterior
```

```
## [1] 0.4 0.2 0.9 0.4
```

```
# step.2 <- unlist(neighborhood(net, 1, step.1, mode="out"))
```

```
#NOTE: will need to switch to personal or relative path if attempting to replicate code
ani.options(convert="/Users/sidsatya/Desktop/Berkeley/senior year/Misinfo/ImageMagick-7.0.10/convert.exe


#' Simulates and animates nodes within a network updating their beliefs.
#'
#' @description
```

```r
#' Take in a graph object, the number of rounds of updating, and the number of
#' simulations and creates an animation of each node in the network propogating
#' beliefs to each other based on their prior and posterior probabilities.
#'
#' @param net An igraph object containing the nodes and edges of the graph of
#'  interest.
#' @param num_rounds Int, The number of rounds of updating.
#' @param num_sims
#' @param is_bipartite Boolean, true if the igraph object passed in is a
#' bipartite graph and false otherwise. Default values is false.
#'
#' @return An igraph object containing the nodes and edges with updated
#'  posterior values as well as a saved image-by-image GIF of the animation.

simulate_and_animate <- function(net, num_rounds, is_bipartite=FALSE) {
  saveGIF( {
            col <- rep("grey40", vcount(net))

            if (is_bipartite == TRUE) {
              l <- layout_as_bipartite(net)
              plot(net, vertex.color=col, layout=l)
            }
            else {
              plot(net, vertex.color=col)
            }

            # adding colors for up to 7 rounds.
            # will need to update this code to allow for unlimited rounds
            colors = c("#ff5100", "#ff9d00", "#ffdd1f", "#f8859b",
                       "#7add1f", "#3e517f", "#9c517f")

            step.1 <- V(net)[round == "R1"]
            col[step.1] <- colors[1]

            if (is_bipartite == TRUE) {
              l <- layout_as_bipartite(net)
              plot(net, vertex.color=col, layout=l)
            }
            else {
              plot(net, vertex.color=col)
            }

            # add modulo
            for (i in 2:num_rounds) {

              i = i %% num_rounds

              # modulo fix
              if (i == 0) {
                i = num_rounds
              }

              # update colors
```

```
            step.i <- V(net)[round==paste0("R", i)]
            col[step.i] <- colors[i]

            # plot
            if (is_bipartite == TRUE) {
              l <- layout_as_bipartite(net)
              plot(net, vertex.color=col, layout=l)
            }
            else {
              plot(net, vertex.color=col)
            }

            # call update()
            for (j in 1:length(step.i)) {
              net <- update_posterior_pull(net, V(net)[step.i[j]])
            }
          }

  }, interval = .8, movie.name="network_animation.gif" )

  #detach('package:igraph')
  #detach('package:animation')
  return(net)
}
```

```
# example for simulating!
rd1 = c("A", "B", "C", "D", "E")
rd2 = c("F", "G", "H", "I", "J")
net <- bipartite(rd1, rd2, make_complete=TRUE)

# test for push, should converge to 0.1
V(net)["A"]$posterior = 0.1
V(net)["B"]$posterior = 0.2
V(net)["C"]$posterior = 0.3
V(net)["D"]$posterior = 0.4
V(net)$posterior # original posteriors
```

```
##  [1] 0.1 0.2 0.3 0.4 0.5 0.5 0.5 0.5 0.5 0.5
```

```
net <- simulate_and_animate(net, 2, is_bipartite=TRUE)
```

```
## Output at: network_animation.gif
```

```
V(net)$posterior # updated posteriors
```

```
##  [1] 0.1 0.2 0.3 0.4 0.5 0.3 0.3 0.3 0.3 0.3
```