abhik1368 / **netpredictor**

Branch: master ▾ netpredictor / R / **netpredictor.R** Find file Copy path

abhik1368 remove prints ebf022d on Feb 10, 2016

1 contributor

999 lines (853 sloc) 37.5 KB

```
#' @title Perform Random walk on a Unipartite Network
#' @description Peforms random walk with restart with preferred seed sets. If seed sets are not given then the adjacencny
#' matrix is taken as the input as the input seed sets. THe restart parameter controls the random walk probability . This can be
#' changed default is set to 0.8. Normalization of the matrix can be done by row,column,laplacian. For faster computation
#' Parallalization is implemented with multicores. Parallization is done using foreach package.
#' @param ig igraph object
#' @param normalise normalise method
#' @param dataSeed vector or dataframe
#' @param restart restart probability parameter
#' @param parallel to execute in parallel either TRUE or FALSE
#' @param multicores Number of cores to be used when running in parallel
#' @param verbose Verbose output
#' @name uNetwalk
#' @references
#' \itemize{
#'   \item Kohler S, et al. Walking the Interactome for Prioritization of Candidate Disease Genes. American Journal of Human Genetics. 2008
#'   \item Can, T., Camoglu, O., and Singh, A.K. (2005). Analysis of protein-protein interaction networks using random walks. In BIOKDD '05
#' }
#' @export
#' @examples
#' \donttest{
#' # generate a random graph according to the ER model
#' library(igraph)
#' library(netpredictor)
#' g1 <- upgrade_graph(erdos.renyi.game(100, 1/100))
#' V(g1)$name <- seq(1,100,1)
#' ## Computing RWR
#' pM <- uNetwalk(g1,normalise="laplacian", restart=0.75, parallel=FALSE)
#' ## Settin the seed nodes.
#' d1 <- c(1,0,1,0,1)
#' d2 <- c(0,0,1,0,1)
#' dataSeed <- data.frame(d1,d2)
#' rownames(dataSeed) <- 1:5
#' pM <- uNetwalk(g1, normalise="laplacian", dataSeed=dataSeed, restart=0.8,
#'                parallel=FALSE,multicores=NULL, verbose=T)
#' }

uNetwalk <- function(ig, normalise=c("row","column","laplacian","none"), dataSeed=NULL, restart=0.8, parallel=TRUE, multicores=NULL, verbos
{

    startT <- Sys.time()
    stop_delta <- 1e-7
    if (class(ig) != "igraph"){
        stop("The function must apply to either 'igraph' or 'matrix' object.\n")
```

```r

    }

    if(verbose){
        now <- Sys.time()
        message(sprintf("First, get the adjacency matrix of the input graph (%s) ...", as.character(now)), appendLF=T)
    }

    if(is.null(restart) || is.na(restart) || restart<0 || restart>100){
        c <- 0.8
    }
    else{
        c <- restart
    }
    normalise <- match.arg(normalise)

    if ("weight" %in% list.edge.attributes(ig)){
        adjM <- get.adjacency(ig, type="both", attr="weight", edges=F, names=T, sparse=getIgraphOpt("sparsematrices"))
        if(verbose){
            message(sprintf("\tNotes: using weighted graph!"), appendLF=T)
        }
    }else{
        adjM <- get.adjacency(ig, type="both", attr=NULL, edges=F, names=T, sparse=getIgraphOpt("sparsematrices"))
        if(verbose){
            message(sprintf("\tNotes: using unweighted graph!"), appendLF=T)
        }
    }

    if(verbose){
        now <- Sys.time()
        message(sprintf("Normalising the adjacency matrix using %s normalisation (%s) ...", normalise, as.character(now)), appendLF=T)
    }

    A <- adjM!=0
    if(normalise == "row"){
        D <- Matrix::Diagonal(x=(Matrix::rowSums(A))^(-1))
        nadjM <- adjM %*% D
    }else if(normalise == "column"){
        D <- Matrix::Diagonal(x=(Matrix::colSums(A))^(-1))
        nadjM <- D %*% adjM
    }else if(normalise == "laplacian"){
        D <- Matrix::Diagonal(x=(Matrix::colSums(A))^(-0.5))
        nadjM <- D %*% adjM %*% D
    }else{
        nadjM <- adjM
    }

    ##  A function to make elements in each steady probability vector is one column normalize
    colNorm<- function(m){
        #res <- t(t(m)/colSums(m))

        col_sum <- apply(m, 2, sum)
        col_sum_matrix <- matrix(rep(col_sum, nrow(m)), ncol=ncol(m), nrow=nrow(m), byrow =T)
        res <- as.matrix(m)/col_sum_matrix
        res[is.na(res)] <- 0
        return(res)
    }

    if(is.null(dataSeed)){

        P0matrix <- Matrix::Matrix(diag(vcount(ig)), sparse=T)
        rownames(P0matrix) <- V(ig)$name
        colnames(P0matrix) <- V(ig)$name

    }else{
        ## check input data
        if(is.matrix(dataSeed) | is.data.frame(dataSeed)){
            data <- as.matrix(dataSeed)
        }else if(is.vector(dataSeed)){
            data <- as.matrix(dataSeed, ncol=1)
        }

        if(is.null(rownames(dataSeed))) {
            stop("The function must require the row names of the input dataSeed.\n")
        }else if(any(is.na(rownames(data)))){
            warning("dataSeed with NA as row names will be removed")
            data <- data[!is.na(rownames(data)),]
        }

        cnames <- colnames(data)
        if(is.null(cnames)){
            cnames <- seq(1,ncol(data))
        }

        if (class(ig) == "igraph"){
            ind <- match(rownames(data), V(ig)$name)
            nodes_mapped <- V(ig)$name[ind[!is.na(ind)]]
            if(length(nodes_mapped)==0){
                stop("The row names of input dataSeed do not contain all those in the input graph.\n")
            }
            P0matrix <- matrix(0,nrow=nrow(nadjM),ncol=ncol(data))
            P0matrix[ind[!is.na(ind)],] <- as.matrix(data[!is.na(ind),])

            ## make sure the sum of elements in each steady probability vector is one
            P0matrix <- colNorm(P0matrix)

            ## Assign row and colnames
            rownames(P0matrix) <- V(ig)$name
            colnames(P0matrix) <- cnames
        }
    }
    if(restart==1){
        ## just seeds themselves
        PTmatrix <- P0matrix
    }else{
        ###### Run in parallel
        flag_parallel <- F
        if(parallel==TRUE){

            flag_parallel <- dCheckParallel(multicores=multicores, verbose=verbose)
            if(flag_parallel){
                j <- 1
                PTmatrix <- foreach::`%dopar%` (foreach::foreach(j=1:ncol(P0matrix), .inorder=T, .combine="cbind"), {
                    P0 <- P0matrix[,j]
                    ## Initializing variables
                    delta <- 1
                    PT <- P0
                    ## Iterative update till convergence (delta<=1e-10)
                    while (delta>stop_delta){
                        PX <- (1-c) * nadjM %*% PT + c * P0
                        # p-norm of v: sum((abs(v).p)^(1/p))
                        delta <- sum(abs(PX-PT))
                        PT <- PX
                    }
                    as.matrix(PT)
                })

                PTmatrix[PTmatrix<1e-6] <- 0
                #PTmatrix <- Matrix::Matrix(PTmatrix, sparse=T)
            }
        }

        if(flag_parallel==F){
            PTmatrix <- Matrix::Matrix(0, nrow=nrow(P0matrix), ncol=ncol(P0matrix), sparse=T)
            for(j in 1:ncol(P0matrix)){
                #P0 <- as.matrix(P0matrix[,j],ncol=1)
                P0 <- P0matrix[,j]

                ## Initializing variables
                delta <- 1

                PT <- P0
                ## Iterative update till convergence (delta<=1e-10)
                while (delta>stop_delta){
                    PX <- (1-c) * nadjM %*% PT + c * P0

                    # p-norm of v: sum((abs(v).p)^(1/p))
                    delta <- sum(abs(PX-PT))

                    PT <- PX
                    #step <- step+1
                }
                #PTmatrix[,j] <- as.matrix(PT, ncol=1)
                PT[PT<1e-6] <- 0
                PTmatrix[,j] <- Matrix::Matrix(PT, sparse=T)
            }
        }
    }
    if(verbose){
        now <- Sys.time()
        message(sprintf("Rescaling steady probability vector (%s) ...", as.character(now)), appendLF=T)
    }
    PTmatrix <- colNorm(PTmatrix)
    PTmatrix <- Matrix::Matrix(PTmatrix, sparse=T)
    rownames(PTmatrix) <- rownames(P0matrix)
    colnames(PTmatrix) <- colnames(P0matrix)

    endT <- Sys.time()
    runTime <- as.numeric(difftime(strptime(endT, "%Y-%m-%d %H:%M:%S"), strptime(startT, "%Y-%m-%d %H:%M:%S"), units="secs"))
    message(paste(c("Runtime in total is: ",runTime," secs\n"), collapse=""), appendLF=T)

    invisible(PTmatrix)

}


#' Network based inference on Bipartite networks
#' @title Network Based Inference
#' @description Given a bipartite graph , a two phase resource transfer Information from  X(x,y,z) set of nodes gets distributed to Y set o
#' This process allows us to define a technique for the calculation of the weight matrix W. if the similarity matrices are not provided it
#' bipartite graph to compute netowrk based inference .
#' @name nbiNet
#' @param A Adjacency Matrix
#' @param lamda Tuning parameter (value between 0 and 1) which determines how the distribution of
#'               resources takes place in thesecond phase
#' @param alpha Tuning parameter (value between 0 and 1) to adjust the performance of the algorithm.
#' @param s1 Target Similarity matrix
#' @param s2 Chemical Similarity Matrix
#' @param format type of file as Adjacnency file
#' @name nbiNet
#' @references
#' \itemize{
#' \item Cheng F, et al. Prediction of drug target interactions and drug repositioning via network-based inference. PLoS Comput. Biol. 2012
#' \item Zhou T, et al. Solving the apparent diversity-accuracy dilemma of recommender systems. Proc. Natl Acad. Sci. USA 2010;107:4511-451
#' \item Zhou T, et al. Bipartite network projection and personal recommendation. Phys. Rev. E Stat. Nonlin. Soft Matter Phys. 2007;76:0461
#' \item Blog post from Abhik Seal \url{http://data2quest.blogspot.com/2015/02/link-prediction-using-network-based.html}
#' }
#' @examples

#' \donttest{
```

```r
#' data(Enzyme)
#' A <- t(enzyme_ADJ)
#' S1 = as.matrix(enzyme_Csim)
#' S2 = as.matrix(enzyme_Gsim)
#' g1 = upgrade_graph(graph.incidence(A))
#' ## other format available \code{format = c("igraph","matrix","pairs")}
#' M2 <- nbiNet(A,alpha=0.5, lamda=0.5,  s1=S1, s2=S2,format = "matrix")
#' M3 <- nbiNet(A,alpha=0.5,lamda=0.5,format="matrix")
#' }
#' @export

## Edit the code to include HeatS code to only predict if we have adjacency matrix
nbiNet <- function (A, alpha=0.5, lamda=0.5, s1=NA, s2=NA,format = c("igraph","matrix","pairs")) {

    startT <- Sys.time()
    format <- match.arg(format)
    now <- Sys.time()
    message(sprintf("Running computation of the input graph (%s) ...", as.character(startT)), appendLF=T)
    if (format == "igraph"){
        adjM = get.incidence(A)
    }
    else if (format == "matrix"){

        adjM <- as.matrix(A)
    }
    else if(format == "pairs") {
        d<- graph.data.frame(A) ## only accepts pairs file
        V(d)$type <- V(d)$name %in% A[,1]
        data <-  get.incidence(d)
        adjM <- transpose(data)
    }
    else stop ("Adjacency matrix should be 'igraph','matrix' or 'pairs' file type \n.")


    n = nrow(adjM)
    m = ncol(adjM)
    if (is.na(s1) && is.na(s2)){

        Ky <- diag(1/colSums(adjM))
        Ky[is.infinite(Ky) | is.na(Ky)] <- 0

        kx <- rowSums(adjM)
        kx[is.infinite(kx) | is.na(kx)] <- 0
        Nx <- 1/(matrix(kx, nrow=n, ncol=n, byrow=TRUE)^(lamda) *
                     matrix(kx, nrow=n, ncol=n, byrow=FALSE)^(1-lamda))
        Nx[is.infinite(Nx) | is.na(Nx)] <- 0
        cl <- makeCluster(detectCores())
        W <- suppressWarnings(t(snow::parMM(cl,adjM,Ky)))
        W <- suppressWarnings(snow::parMM(cl, adjM, W))
        #W <- t(adjM %*% Ky)
        W <- Nx * W
        rownames(W) <- rownames(adjM)
        colnames(W) <- rownames(adjM)
        rM <-  suppressWarnings(snow::parMM(cl,W,adjM))
        endT <- Sys.time()
        stopCluster(cl)
        runTime <- as.numeric(difftime(strptime(endT, "%Y-%m-%d %H:%M:%S"), strptime(startT, "%Y-%m-%d %H:%M:%S"), units="secs"))
        message(sprintf("Done computation of the input graph (%s) ...", as.character(endT)), appendLF=T)
        message(paste(c("Runtime in total is: ",runTime," secs\n"), collapse=""), appendLF=T)
        invisible (rM)

    } else {

        if (nrow(s2) != m || ncol(s2) != m) {
            stop("The matrix s2 should be an m by m matrix with same number of columns as A.")
        }
        if (nrow(s1) != n || ncol(s1) != n) {
            stop("The matrix s1 should be an n by n matrix with same number of rows as A")
        }

        Ky <- diag(1/colSums(adjM))
        Ky[is.infinite(Ky) | is.na(Ky)] <- 0

        kx <- rowSums(adjM)
        kx[is.infinite(kx) | is.na(kx)] <- 0
        Nx <- 1/(matrix(kx, nrow=n, ncol=n, byrow=TRUE)^(lamda) *
                     matrix(kx, nrow=n, ncol=n, byrow=FALSE)^(1-lamda))
        Nx[is.infinite(Nx) | is.na(Nx)] <- 0
        cl <- makeCluster(detectCores())
        W <- suppressWarnings(t(snow::parMM(cl,adjM,Ky)))
        W <- suppressWarnings(snow::parMM(cl, adjM, W))
        #W <- t(adjM %*% Ky)
        W <- Nx * W
        rownames(W) <- rownames(adjM)
        colnames(W) <- rownames(adjM)
        X5 <- suppressWarnings(snow::parMM(cl, adjM, s2))
        X6 <- suppressWarnings(snow::parMM(cl, X5, t(adjM)))
        X7 <- suppressWarnings(snow::parMM(cl, adjM, matrix(1, nrow=m, ncol=m)))
        X8 <- suppressWarnings(snow::parMM(cl, X7, t(adjM)))
        S3 <- X6 / X8

        W  <- W * ((alpha * s1) + ((1-alpha) * S3))

        W[is.nan(W)] <- 0
        rM <-  suppressWarnings(snow::parMM(cl,W,adjM))

        endT <- Sys.time()
        stopCluster(cl)
        runTime <- as.numeric(difftime(strptime(endT, "%Y-%m-%d %H:%M:%S"), strptime(startT, "%Y-%m-%d %H:%M:%S"), units="secs"))
        message(sprintf("Done computation of the input graph (%s) ...", as.character(endT)), appendLF=T)
        message(paste(c("Runtime in total is: ",runTime," secs\n"), collapse=""), appendLF=T)
        invisible (rM)
    }
}


#' Randomm walk with restart on Bipartite networks
#' @title Bipartite Random Walk
#' @name biNetwalk
#' @param g1 Bipartite graph igraph object.
#' @param s1 Accepts a matrix object of similarity scores for targets.
#' @param s2 Accepts a matrix object similarity scores for compounds.
#' @param normalise Normalisation of matrix using laplacian, Chen, None(the transition matrix will be column normalized)
#' @param dataSeed seeds file
#' @param restart restart value
#' @param weight if we want to use a weighted network . Options are either TRUE or FALSE.
#' @references
#' \itemize{
#' \item {Chen X, et al. Drug target interaction prediction by random walk on the heterogeneous network. Mol. BioSyst 2012;8:1970-1978.}
#' \item {Vanunu O, Sharan R. Proceedings of the German Conference on Bioinformatics. Germany: GI; 2008. A propagation-based algorithm for
#' }
#' @examples
#' \dontrun{
#' data(Enzyme)
#' A <- enzyme_ADJ
#' S2 = enzyme_Csim
#' S1 = enzyme_Gsim
#' g1 = graph.incidence(A)
#' M3 <- biNetwalk(g1,s1=S1,s2=S2,normalise="laplace", dataSeed=NULL,restart=0.8,

#'                 parallel=FALSE, verbose=T,weight=FALSE)
#' dataF<- read.csv("seedFile.csv",header=FALSE)
#' Mat <- biNetwalk(g1,s1=S1,s2=S2,normalise="laplace", dataSeed=dataF,restart=0.8,
#'                  parallel=TRUE,verbose=T,weight=FALSE)
#' }
#' @export

biNetwalk <- function(g1,s1,s2,normalise=c("laplace","none","chen"), dataSeed=NULL,restart=0.8,verbose=T,weight=FALSE) {

    startT <- Sys.time()

    if (!exists('s1') || !exists('s2')){
        stop("You must submit s1 and s2 matrices.\n")
    }

    if (class(g1) != "igraph"){
        stop("The function applies to 'igraph' object.\n")
    }

    if (!bipartite.mapping(g1)$res){
        stop("The function applies to bipartite graphs.\n")
    }

    if(verbose){
        now <- Sys.time()
        message(sprintf("First, get the adjacency matrix of the input graph (%s) ...", as.character(now)), appendLF=T)
    }
    if(is.null(restart) || is.na(restart) || restart<0 || restart>100){
        c <- 0.8
    }
    else{
        c <- restart
    }
    normalise <- match.arg(normalise)
    if (weight){
        if ("weight" %in% list.edge.attributes(g1)){
            adjM <- get.incidence(g1, attr="weight", names=T)
            if(verbose){
                message(sprintf("Notes: using weighted graph!"), appendLF=T)
            }
        }
    }else{
        adjM <- get.incidence(g1, attr=NULL, names=T)
        if(verbose){
            message(sprintf("Note: using unweighted graph!"), appendLF=T)
        }
    }
    adjM <- as.matrix(adjM)
    # get the transition matrix
    W = tMat(adjM,s1,s2,normalise=normalise)
    message(sprintf("got the transition matrix for RWR"))
    if(is.null(dataSeed)){

        M<-Matrix(adjM)
        M2<-0.99*M
        d<-Matrix(0.01*diag(nrow(s2)))
        P0matrix<-rBind(M2,d)

    }else{

        # part of the section for input file name
        drug.names <- as.character(unique(dataSeed$V2))
        P0matrix <- matrix(0,nrow=nrow(W),ncol=length(drug.names))

        for (i in 1:length(drug.names)){
            sub.fr <- dataSeed[dataSeed$V2==drug.names[i],]

            proteins <- as.character(sub.fr$V1)
            ind1 <- match(proteins, rownames(W))
            ind2 <- match(drug.names[i],rownames(W))
            ind <- append(ind1,ind2)
            nodes_mapped <- rownames(W)[ind[!is.na(ind)]]
            if(length(nodes_mapped)!=length(ind)){
                warning("The row names of input dataSeed do not contain all those in the input graph.\n")
            }

            P0matrix[ind[!is.na(ind)],i] <- 1
        }
        P0matrix <- colNorm(P0matrix)

    }


    if (exists("W")){
        rmat <- rwr(W,P0matrix,r=c)
    } else{
        stop("Transition matrix couldnt be generated..")
    }

    if (!exists("rmat")){
        stop("Couldn't return the RWR matrix. \n")
    }else{
        if(verbose){
            now <- Sys.time()
            message(sprintf("Rescaling steady probability vector (%s) ...", as.character(now)), appendLF=T)
        }
        rmat[rmat < 1e-06] <- 0
        rmat <- rmat[1:nrow(adjM),]

        rmat <- colNorm(as.matrix(rmat))
        rownames(rmat)<- rownames(adjM)
        if(!is.null(dataSeed)){
            colnames(rmat)<- drug.names
            invisible(rmat)
        } else {
            colnames(rmat)<- colnames(adjM)
            invisible(rmat)
        }
        endT <- Sys.time()
        runTime <- as.numeric(difftime(strptime(endT, "%Y-%m-%d %H:%M:%S"), strptime(startT, "%Y-%m-%d %H:%M:%S"), units="secs"))
        message(paste(c("Runtime in total is: ",runTime," secs\n"), collapse=""), appendLF=T)
        invisible(rmat)

    }

}

#' Significance of Bipartite networks
#' @title Significant Network
#' @param data n x m Adjancency matrix of seeds or dataframe of pairs.
#' @param g igrah object of bipartite indcident adjacency matrix.
#' @param Amatrix Affinity matrix computed from biNetWalk or uNetWalk.
#' @param num.permutation number of permutation of Affinity matrix needed to performed.
#' @param adjp.cutoff pvalue cutoff 0.05
#' @param p.adjust.method Adjusting the pvalue by diiferent method.It uses method from the stats package.
#' @param parallel Using parallization either True or False
#' @param multicores If parallisation is set TRUE number of cores to perform parallel computaion.
#' @param verbose For verbose output.
#' @name sig.net
#' @examples
#' \donttest{
#' data(Enzyme)
#' A <- enzyme_ADJ
#' S1 = enzyme_Gsim
#' S2 = enzyme_Csim
#' g1 = graph.incidence(A)
#' Q = biNetwalk(g1,s1=S1,s2=S2,normalise="laplace", dataSeed=NULL, file=NULL,restart=0.8,verbose=T)
```

```r
#' Z = sig.net(data=A,g=g1,Amatrix=Q,num.permutation=100,adjp.cutoff=0.01,p.adjust.method="BH",parallel=FALSE,verbose=T)
#' }
#' @export

sig.net <- function(data, g, Amatrix, num.permutation=10, adjp.cutoff=0.05, p.adjust.method=c("holm", "hochberg", "hommel", "bonferroni", "
{
    library(igraph)
    startT <- Sys.time()

    ######################################################################################
    permutation <- "random"
    p.adjust.method <- match.arg(p.adjust.method)
    ## check input data
    if(is.matrix(data) | is.data.frame(data)){

        data <- as.matrix(data)
        if (ncol(data)==2){
            g.data <- graph.data.frame(data) ## only accepts pairs file
            V(g.data)$type <- V(g.data)$name %in% data[,1]
        } else if (ncol(data) > 2){ ## Needs check seeds can be adjacency matrix
            data <- t(data)
            g.data <- graph.incidence(data)
        } else if(ncol(data) < 2){
            stop("The input data must be matrix or dataframe with at least two columns.\n")
        }



    }else if(is.null(data)){
        stop("The input data must be matrix.\n")
    }

    ## check input graph
    if (class(g) != "igraph"){
        stop("The function must apply to 'igraph' object.\n")
    }

    if (!bipartite.mapping(g.data)$res){
        stop("The function applies to igrah bipartite graphs.\n")
    }

    ## check bipartite mapping between input seed data and graph
    p <- bipartite.projection(g.data)
    rnames <- V(p[[1]])$name # rownames of the seed matrix
    cnames <- V(p[[2]])$name # colnames of the seed matrix

    # Get the adjacency matrix from igraph object
    g.incident <- get.incidence(g)

    ind <- match(cnames, rownames(g.incident))
    nodes_mapped <- rownames(Amatrix)[ind[!is.na(ind)]]
    P0matrix <- matrix(0, nrow=length(rownames(Amatrix)),ncol=length(rnames))
    P0matrix[ind[!is.na(ind)],] <- 1
    rownames(P0matrix) <- rownames(Amatrix)
    colnames(P0matrix) <- rnames

    ## check mapping between input Affinity matrix and graph
    ind1 <- match(rownames(Amatrix), rownames(g.incident))
    ind2 <- match(colnames(Amatrix), rnames)
    if(length(ind1[!is.na(ind1)])!=length(rownames(g.incident)) && length(ind2[!is.na(ind2)])!=length(colnames(g.incident))) {
        stop("The function must require input Affinity matrix (Amatrix) has the same names (both columns and rows) as the input graph.\n")

    }


    PTmatrix <- Amatrix[ind1[!is.na(ind1)],ind2]
    PTmatrix <- colNorm(as.matrix(PTmatrix))


    ####################################################

    obs <- as.matrix(t(PTmatrix) %*% PTmatrix)
    B <- num.permutation
    if(verbose){
        message(sprintf("Third, generate the distribution of association scores based on %d permutations on nodes respecting %s (%s)...", B
    }


    f <- function(){
        pb <- txtProgressBar(min=1, max=num.permutation-1,style=3)
        count <- 0
        function(...) {
            count <<- count + length(list(...)) - 1
            setTxtProgressBar(pb,count)
            Sys.sleep(0.01)
            flush.console()
            c(...)
        }
    }
    ###### parallel computing
    flag_parallel <- F
    if(parallel==TRUE){

        flag_parallel <- dCheckParallel(multicores=multicores, verbose=verbose)
        if(flag_parallel){
            b <- 1
            exp_b <- foreach::`%dopar%` (foreach::foreach(b=1:B, .inorder=T,.combine=f()), {
                PT_random <- PTmatrix[sample(nrow(PTmatrix)),sample(ncol(PTmatrix))]
                ## make sure the sum of elements in each steady probability vector is one
                PT_random <- colNorm(as.matrix(PT_random))
                as.matrix(t(as.matrix(PT_random)) %*% PT_random)
            })
        }
    }

    ## non-parallel computing
    if(flag_parallel==F){
        exp_b <- lapply(1:B, function(b){
            PT_random <- PTmatrix[sample(nrow(PTmatrix)),sample(ncol(PTmatrix))]
            ## make sure the sum of elements in each steady probability vector is one
            PT_random <- colNorm(as.matrix(PT_random))
            as.matrix(t(as.matrix(PT_random)) %*% PT_random)
        })
    }

    n <- ncol(obs)
    ## for zscore
    exp_mean <- matrix(0, ncol=n, nrow=n)
    exp_square <- matrix(0, ncol=n, nrow=n)
    for(b in 1:B){
        exp <- exp_b[[b]]
        exp_mean <- exp_mean + exp
        exp_square <- exp_square + exp^2
    }
    exp_mean <- exp_mean/B
    exp_square <- exp_square/B
    exp_std <- sqrt(exp_square-exp_mean^2)
    zscore <- (obs-exp_mean)/exp_std
    zscore[is.na(zscore)] <- 0
    zscore[is.infinite(zscore)] <- 0

    ## for pvalue
    num <- matrix(0, ncol=n, nrow=n)
    for(b in 1:B){
        num <- num + (obs < exp_b[[b]])
    }
    pval <- num/B
    colnames(pval) <- colnames(obs)
    rownames(pval) <- rownames(obs)

    ## for adjusted pvalue
    adjpval <- pval
    ## lower part
    flag_lower <- lower.tri(pval, diag=F)
    adjpval[flag_lower] <- stats::p.adjust(pval[flag_lower], method=p.adjust.method)
    ## upper part
    flag_upper <- upper.tri(pval, diag=F)
    adjpval[flag_upper] <- stats::p.adjust(pval[flag_upper], method=p.adjust.method)

    if(verbose){
        message(sprintf("Also, construct the association graph under the cutoff %1.1e of adjusted-pvalue (%s)...", adjp.cutoff, as.characte
    }
    flag <- adjpval < adjp.cutoff
    adjmatrix <- flag
    adjmatrix[flag] <- zscore[flag]
    cgraph <- igraph::graph.adjacency(adjmatrix, mode="undirected", weighted=T, diag=F, add.colnames=NULL, add.rownames=NA)

    ##########################################
    endT <- Sys.time()
    runTime <- as.numeric(difftime(strptime(endT, "%Y-%m-%d %H:%M:%S"), strptime(startT, "%Y-%m-%d %H:%M:%S"), units="secs"))
    message(paste(c("Runtime in total is: ",runTime," secs\n"), collapse=""), appendLF=T)

    result <- list(pval = pval,
                   adjpval = adjpval,
                   cgraph  = cgraph)
    invisible(result)
}

#' NetCombo
#' @title  NetCombo
#' @description Peforms computation three different algorithms like random walk, network based inference and heterogenous based inference a
#' @param g1 igraph object
#' @param s1 Accepts a matrix object of similarity scores for targets.
#' @param s2 Accepts a matrix object similarity scores for compounds.
#' @param nbi.alpha alpha value for network based inference.
#' @param nbi.lamda lamda value for network based inference.
#' @param norm normalization of matrices options are "laplace" or "none".
#' @param restart restart parameter for RWR
#' @param par parallel execution for RWR.
#' @return Matrix object with sum score values.
#' @name netCombo
#' @examples
#' \donttest{
#' data(Enzyme)
#' A = enzyme_ADJ
#' S1 = as.matrix(enzyme_Gsim)
#' S2 = as.matrix(enzyme_Csim)
#' g1 = graph.incidence(A)
#' P <- netCombo(g1,s1=S1,s2=S2,nbi.alpha=0.5,nbi.lamda=0.5,par=TRUE)
#' ## With a different restart
#' P <- netCombo(g1,s1=S1,s2=S2,nbi.alpha=0.5,nbi.lamda=0.5,restart=0.7,par=TRUE)
#' }
#' @export



netCombo <- function(g1,s1,s2,nbi.alpha=0.4,nbi.lamda=0.5,norm="laplace",restart=0.8,par=TRUE) {

    startT <- Sys.time()
    now <- Sys.time()
    if (!exists('s1') || !exists('s2')){
        stop("You must submit s1 and s2 matrices.\n")
    }

    if (class(g1) != "igraph"){
        stop("The function must apply to either 'igraph' object.\n")
    }
    if (!bipartite.mapping(g1)$res){
        stop("The function applies to bipartite graphs only.\n")
    }

    A <- as.matrix(get.incidence(g1))
    message(sprintf("Running computation of the input graph (%s) ...", as.character(startT)), appendLF=T)
    message(sprintf("Running computation for RWR..\n"))
    Q1 = biNetwalk(g1,s1=s1,s2=s2,normalise="laplace",verbose=T,restart = restart)


    message(sprintf("Running computation for network based inference..\n"))
    Q2 = nbiNet(A,lamda=nbi.lamda,alpha=nbi.alpha,s1=as.matrix(s1),s2=as.matrix(s2),format = "matrix")

    if (exists("Q1") && exists("Q2")){
        M <- (Q1+Q2)/2
        return (M)
    }

}

#' get the performance of the link Prediction algorithms.
#' @title  Link Prediction Performance
#' @description This function samples links and removies links from the adjacency matrix and predicts them and calculates  area under accum
#' @param S1 Sequence similarity matrix object
#' @param A Drug target association matrix
#' @param S2 Accepts a matrix object similarity scores for compounds.
#' @param relinks Number of links to remove randomly from the input matrix.
#' @param numT Frequency of the number of targets.
#' @param restart restart value if using rwr or netcombo
#' @param alpha alpha value if using nbi or netcombo
#' @param lamda lamda value if using nbi or netcombo
#' @param Calgo Algorithm to use for Bipartite link prediction options are "rwr","nbi" & "netcombo".
#' @param norm normalization of matrices options are "laplace" or "none".
#' @name net.perf
#' @return it returns a list of aucc,auc, bedorc,enrichment factor and auc (top 10%)
#' \itemize{
#'   \item {Truchon et al. Evaluating Virtual Screening Methods: Good and Bad Metrics for the "Early Recognition" Problem. J. Chem. Inf. Mo
#'   \item {Sheridan RP et al. Protocols for bridging the peptide to nonpeptide gap in topological similarity searches. J. Chem. Inf. Compu
#' }
#' @examples
#' \dontrun{
#' data(Enzyme)
#' A = enzyme_ADJ
#' S1 = enzyme_Gsim
#' S2= enzyme_Csim
#' m = net.perf(A,S1,S2,alpha=0.5,lamda=0.5,relinks = 50,numT=2,norm="laplace",Calgo="nbi")
#' }
#' @export

net.perf<- function(A,S1,S2,restart=0.8,alpha=0.5,lamda=0.5,relinks=100,numT=2,norm="laplace",Calgo = c("rwr","nbi","netcombo","all")){

    auctop = numeric()
    aucc = numeric()
    bdr  = numeric()
    efc   = numeric()
    ranks = numeric()
    totallinks = sum(A)

    m = dim(A)[1] ## rows for targets
```

```r
    n = dim(A)[2] ## columns for drugs

    if (!exists('S1') || !exists('S2')){
        stop("You must submit s1 and s2 matrices.\n")
    }

    if (nrow(S1)!=m | ncol(S1) != m){
        stop("Your number of targets does not match with target similarity matrix.\n")
    }

    if (nrow(S2)!=n | ncol(S2) != n){
        stop("Your number of targets does not match with target similarity matrix.\n")
    }


    ## Get the name of the algorithm.
    algo <- match.arg(Calgo)
    g1 <- graph.incidence(A)
    eg <- get.edgelist(g1)
    c <- data.frame(table(eg[,2]))
    c <- c[c$Freq>numT,]

    drugnames <- unique(as.character(c$Var1))

    ids <- which(eg[,2] %in% drugnames)
    re <- eg[sample(ids,size = relinks,replace=FALSE),]


    if (totallinks <= relinks){
        stop("Total links removed is less than equal given links to be removes. Give a sensible value.")
    }

    SampledGraph <- g1
    for (i in 1:dim(re)[1])
    {
        if (are.connected(SampledGraph, re[i,1], re[i,2]))
            SampledGraph <- delete.edges(SampledGraph, E(SampledGraph, P=c(re[i,1], re[i,2])))
    }
    g1 = SampledGraph
    Sg_t <- get.incidence(SampledGraph)

    #Sg_t <- randomizeMatrix(Sg_t,null.model = "frequency",iterations = 1000)

    #mat<-tMat(Sg_t,as.matrix(S1),as.matrix(S2),normalise="laplace")

    drugs <- re[,2]

    message(sprintf("Detected (%s) drugs & (%s) proteins with (%s) interactions...",n,m,totallinks))
    message(sprintf("Running prediction for (%s) links removed using (%s) .. ",as.character(relinks),as.character(algo)))

    performances <- function(predictR,m,re){

        s1<-predictR[1:m,]
        s1<- scale(s1, center=FALSE, scale=colSums(s1,na.rm=TRUE))
        s1[is.na(s1)] <- 0
        test <- data.frame(re)
        for (dis in 1:dim(s1)[2]) {

            drugname = colnames(s1)[dis]
            subfr <- test[test$X2==drugname,]

            p1name<-as.character(subfr$X1)
            id = which(rownames(s1) %in% p1name)
            clabel <- rep(0,m)
            clabel[id] <- 1
            res = cbind(s1[,dis],clabel)
            colnames(res)[1] <- "score"

            d <- res[order(-res[,1]),]
            ac <- auac(d[,1], d[,2])
            au <- auc(d[,1], d[,2])
            at <-  auc(d[,1], d[,2],top=0.1)
            bd <- bedroc(d[,1], d[,2])
            ef <- enrichment_factor(d[,1], d[,2],top=0.1)
            aucc <- c(aucc, ac)
            bdr <- c(bdr,bd)
            efc <- c(efc,ef)
            auctop <- c(auctop,at)

        }

        scores = c(list(auac = mean(aucc),auc= mean(au),auctop = mean(auctop),bdr = mean(bdr),efc = mean(efc)))
        return (scores)
    }

    if (algo == "rwr"){
        #par="True"
        message(sprintf("Running RWR Algorithm"))
        mat = biNetwalk(g1,s1=S1,s2=S2,restart=restart,normalise=norm,verbose=T)
        predictR <- mat[,colnames(mat) %in% drugs]
        scores <- performances(predictR,m,re)
        return (scores)
    }
    else if (algo == "nbi"){
        message(sprintf("Running NBI Algorithm"))
        #S1 = S1[rownames(S1) %in% rownames(N_M),colnames(S1) %in% rownames(N_M)]
        #S2 = S2[rownames(S2) %in% colnames(N_M),colnames(S2) %in% colnames(N_M)]
        mat <- nbiNet(Sg_t, lamda=lamda, alpha=alpha, s1=as.matrix(S1), s2=as.matrix(S2),format = "matrix")
        predictR <- mat[,colnames(mat) %in% drugs]
        scores <- performances(predictR,m,re)
        return (scores)
    }

    else if(algo == "netcombo"){
        message(sprintf("Running NetCombo Algorithm"))
        #par="True"
        mat1 = biNetwalk(g1,s1=S1,s2=S2,normalise=norm,verbose=T,restart=restart)
        mat2 <- nbiNet(Sg_t,lamda=lamda, alpha=alpha, s1=as.matrix(S1), s2=as.matrix(S2),format = "matrix")
        mat = (mat1+mat2)/2
        predictR <- mat[,colnames(mat) %in% drugs]
        scores <- performances(predictR,m,re)
        return (scores)
    } else if (algo == "all"){

        message(sprintf("Running all the algorithms ..."))
        #par="True"
        mat1 <- biNetwalk(g1,s1=S1,s2=S2,normalise=norm,verbose=T)
        mat2 <- nbiNet(Sg_t, lamda=0.5, alpha=0.5, s1=as.matrix(S1), s2=as.matrix(S2),format = "matrix")
        mat3 <- (mat1+mat2)/2
        predictR1 <- mat1[,colnames(mat1) %in% drugs]
        predictR2 <- mat2[,colnames(mat2) %in% drugs]
        predictR3 <- mat3[,colnames(mat3) %in% drugs]

        scores1 <- performances(predictR1,m,re)
        scores2 <- performances(predictR2,m,re)
        scores3 <- performances(predictR3,m,re)

        list1 = list(type = 'rwr',score=scores1)
        list2 = list(type = 'nbi',score=scores2)
        list3 = list(type = 'netcombo',score=scores3)
        scoreList = list(list1,list2,list3)
        return (scoreList)

```

```r
    }
}

#' Get top predicted results.
#' @title  Get Top Results
#' @description The function returns the given top number of predicted results along with true interactions.
#' @param A Drug target association matrix.
#' @param P Drug target predicted matrix.
#' @param top top number of predicted targets.
#' @param druglist It accepts a vector of drugnames for which results will return
#' @name getTopresults
#' @return it returns a list of aucc,auc, bedorc,enrichment factor and auc (top 10%)
#' @examples
#' \donttest{
#' data(Enzyme)
#' A = enzyme_ADJ
#' S1 = enzyme_Gsim
#' S2= enzyme_Csim
#' ## Running the netcombo algorithm.
#' P <- netCombo(g1,s1=S1,s2=S2,nbi.alpha=0.5,nbi.lamda=0.5,par=TRUE)
#' result = getTopresults(A,P,top=10,druglist=NULL)
#' ## Getting result from a drug list.
#' drugs = c("D00014","D00018", "D00029", "D00036","D00045","D00049")
#' result = getTopresults(A,P,top=10,druglist=drugs)
#' }
#' @export

getTopresults <- function(A,P,top=10,druglist=NULL){

    startT <- Sys.time()
    now <- Sys.time()

    `%not in%` <- function (x, table) is.na(match(x, table, nomatch=NA_integer_))

    A <- A[,colnames(A) %in% colnames(P)]

    if (length(rownames(A)) <=0 ){
        stop("Drugs names doesnt match for Predicted matrix and Original Matrix")
    }

    g1 <- graph.incidence(A)

    el = data.frame(get.edgelist(g1))

    if (is.null(druglist)){
        drugnames = colnames(P)
        fr <- data.frame()
        for (i in 1:length(drugnames)){
            lt = el[el$X2==drugnames[i],]
            tproteins = as.character(lt$X1)
            if (length(tproteins) > 0 ){
                d <- P[order(-P[,i]),]
                pnames = rownames(d)
                score <- as.numeric(d[,i])
                drug <- drugnames[i]
                result <- data.frame(cbind(drug,pnames,score))
                tp <- result[result$pnames %in% tproteins,]
                tp$type <- "True Interactions"
                pi = result[result$pnames %not in% tproteins,]

                pi = pi[1:top,]
                pi$type = "Predicted Interactions"
                r <- rbind(tp,pi)
                fr <- rbind(fr,r)
            }

        }

    }

    else {
        drugnames = druglist
        fr <- data.frame()
        for (i in 1:length(drugnames)){

            lt = el[el$X2==drugnames[i],]
            tproteins = as.character(lt$X1)
            if (length(tproteins) > 0 ){
                d <- P[order(-P[,colnames(P) %in% drugnames[i]]),]
                pnames = rownames(d)
                score <- as.numeric(d[,colnames(P) %in% drugnames[i]])
                drug <- drugnames[i]
                result <- data.frame(cbind(drug,pnames,score))
                tp <- result[result$pnames %in% tproteins,]
                tp$type <- "True Interactions"
                pi = result[result$pnames %not in% tproteins,]
                pi = pi[1:top,]
                pi$type = "Predicted Interactions"
                r <- rbind(tp,pi)
                fr <- rbind(fr,r)
            }
        }
    }

    invisible(fr)
}
```