Supplementary material for: Smith, N.R., Zivich, P.N., Frerichs, L., Moody, J., & Aiello, A.E. A guide for choosing community detection algorithms in social network studies: The Question-Alignment approach.

Contents

- Divisive 2
  - Edge-betweenness 2
- Agglomerative 3
  - Walktrap 3
  - Label Propagation 4
- Optimization based 4
  - Louvain (Multilevel) 4
  - Infomap 5
  - Spinglass 6

References 8

eX-FLU case study 9
- Create Network 9
  - Graph Input Network 11
- Detect Communities 12
- Walktrap Dendrogram (Figure 1) 13
- Describe Communities 16
- Community Overlap 16
- Graph Community Structures to show discrepancies (Figure 2) 17
- Choose communities to intervene (Figure 3) 20

Appendix – QA approach

Mathematical Descriptions of Algorithms

Throughout the following, we will use the following notation. G refers to a network with the adjacency matrix A with n total nodes. Furthermore, we restrict our discussion to unweighted, undirected networks. An edge between node i and node j in the adjacency matrix is defined as

$$A_{ij} = \begin{cases} 1, & \text{if } i \text{ and } j \text{ are connected} \\ 0, & \text{otherwise} \end{cases}$$

The degree of node i is the total number of edges connected to i, written as

$$d_i = \sum_j A_{ij}$$

The total number of edges in the network is defined as

$$m = \frac{1}{2} \sum_{i,j} A_{ij}$$

Modularity is defined as

$$Q = \frac{1}{2m} \sum_{i,j} \left(A_{ij} - \frac{d_i * d_j}{2m} \right) * I(c_i = c_j)$$

where c_i indicates the index of the community that node i belongs to. The indicator function is defined as

$$I(c_i = c_j) = \begin{cases} 1, & \text{if } i \text{ and } j \text{ in same community} \\ 0, & \text{otherwise} \end{cases}$$

Random walks in networks are flows on paths of edges connecting nodes based on successions of random steps. The concept of random walks between nodes is used by several algorithms to define communities. Often this is operationalized through a probability formula rather than directly simulating random walks within the network.

Divisive

Divisive algorithms begin with all nodes being considered as a single community. The process repeatedly divides the network into smaller communities by detecting and removing edges that link disparate communities.

Edge-betweenness

Also referred to as the Girvan-Newman algorithm [1], the edge-betweenness algorithm divides a network into communities by iteratively dividing the network into smaller pieces using single edge deletion. The edge deleted in each iteration is decided by the edge-betweenness values. The following process is used

1. Calculate the edge betweenness for all edges, e. Edge betweenness is the proportion of shortest paths between all nodes in a network that pass through an edge. The formula is
$$B(e) = \sum_{i,j} \frac{P_e(i,j)}{P(i,j)}$$
where $B(e)$ is the edge-betweenness value for edge e, $P(i,j)$ is the shortest path between node i and node j, and $P_e(i,j)$ is the shortest path between node i and node j that include edge e.
2. Remove the edge with the highest betweenness value $B(e)$
3. Recalculate the edge betweenness for all remaining edges
4. Repeat steps 2 and 3 until no edges remain in the network

The result of the above algorithm is a dendrogram, which does not naturally decompose the network into communities [2]. By default, R's `igraph` will return the community structure that results in the greatest modularity value. One advantage of the edge-betweenness algorithm is that the user has greater ability to influence the granularity of communities in the network. Additionally, other functions to measure betweenness can be used to determine edge deletion. One example is the usage of random-walk betweenness. Random-walk betweenness is based on random walks instead of shortest paths.

Agglomerative

Agglomerative community detection methods begin with each node considered as a distinct community and iteratively merges similar nodes into communities. These approaches build communities from independent structures.

Walktrap

Walktrap utilizes the concept of random walks between nodes [3]. Walktrap identifies which nodes are close as defined by the probability of walking from node i to node j. To control the size of detected communities, the distance of the walks can be modified. Pairs of nodes that reduces the overall distance between nodes and communities are iteratively merged together. By focusing on random walks, nodes that have high levels of flow between each other are merged into communities.

The following algorithm is used to determine the set of communities. To start, each node is considered a distinct community.

1. Determine the communities that minimize the change in mean squared distances of nodes and the community, $\Delta\sigma$.
2. Merge the two identified communities into a single community
3. Update distances between communities
4. Repeat until all nodes are combined into one community

To calculate the change in mean squared distances, the following formula is used

$$\Delta\sigma(c_v, c_w) = \frac{1}{n} * \frac{|c_v|\,|c_w|}{|c_v| + |c_w|} * r^2_{c_v,c_w}$$

$$r_{c_v,c_w}^2 = \sqrt{\sum_{k=1}^{n} \frac{\left(P_{c_v,k} - P_{c_w,k}\right)^2}{d_k}}$$

where $|c_v|$ is the number of nodes in community c_v (i.e. cardinality of the set), the transition probability between nodes i and j is $P_{ij} = \frac{A_{ij}}{d_i}$, the probability of going from between nodes i and j in t steps is $P_{ij}^t = (P^t)_{ij}$, and the probability of going from community c_v to node i in t steps is $P_{c_v,i}^t = \frac{1}{|c_v|}\sum_{j\in c_v} P_{ij}^t$

Similar to Edge-betweenness, Walktrap results in a dendrogram and `igraph` will return the community structure with the largest modularity.

Label Propagation

Label propagation defines communities by having nodes adopt the label of the majority of its neighbors, with ties broken at random [4]. The following process is used to propagate labels through the network

1. To start, each node is given a unique label
2. Randomly order the nodes in the network
3. Update each node's label following the order in step 2. The node takes the label of the majority of its neighbors. For tied labels, the new label is chosen randomly from the tied labels
4. Repeat steps 2 and 3 until every node has a label that the maximum number of their neighbors has

Label propagation results in a singular division of the network based on the algorithm. Depending on the random number generator, different community divisions may be returned because different labels could be assigned. For nodes that lie between two cohesive communities, the label adopted by these nodes may be unstable, therefore a seed should be set to obtain consistent results for these nodes.

Optimization based

Optimization-based algorithms define an objective function for the community divisions. The algorithm then searches for the community division that maximizes this objective function. While some algorithms use an agglomerative approach to find the maximum, these methods focus on explicit maximization of an overall objective function.

Louvain (Multilevel)

The Louvain algorithm, also referred to as the Multilevel algorithm, uses the modularity maximization approach [5]. We use Multilevel to refer to this algorithm in the supplementary code.

The Louvain approach uses the following formulation for the change in modularity for adding a single node k into the community v

$$\Delta Q = \left(\frac{\sum_{i,j} A_{ij} * I(c_i, v) * I(c_j, v) + 2 * \sum_j A_{kj} * I(c_j, v)}{2m} - \left(\frac{\sum_{i,j} A_{ij} * I(c_i, v\ ) + \sum_j A_{kj}}{2m}\right)^2\right) - \left(\frac{\sum_{i,j} I(c_i, v) * I(c_j, v)}{2m} - \left(\frac{\sum_{i,j} I(c_i, v\ )}{2m}\right)^2 - \left(\frac{\sum_j A_{kj}}{2m}\right)^2\right)$$

Where $\sum_{i,j} A_{i\,j} * I(c_i, v) * I(c_j, v)$ is the sum of the edges between nodes in community v, $\sum_{i,j} A_{i\,j} * I(c_i, v\ )$ is the total sum of edges of nodes in community v, $\sum_j A_{kj} * I(c_j, v)$ is the sum of edges between node k to edges in community v, and $\sum_j A_{kj}$ is the total sum of edges between node k and nodes in the network. This formula can be generalized to weighted networks.

The Louvain communities are determined by the following iterative process. The process begins with each node belonging to a unique community.

1. Using the previously defined change in modularity equation, merging each pair of communities into a single community is evaluated for ΔQ.
2. The community pair which represents the largest, positive gain in ΔQ is merged.
3. Repeat steps 1 and 2 until there are no additional gains in modularity (i.e. $\Delta Q \leq 0$).

The Louvain algorithm results in a single division of the network into communities, where the single division of the maximum modularity Q. The algorithm can be estimated multiple times (with a different random seed for each run) and the result with the highest overall modularity (Q) can be chosen to avoid a local maxima of modularity.

As a note, there may be differences in how the Louvain algorithm is implemented across software platforms (e.g., *igraph* vs. Pajek).

Infomap

Infomap uses results from information theory to divide the network into communities defined by flow through a network [6]. Flow through a network does not need to be explicitly measured, but relies on random walks within the network. To help conceptualize the procedure, we will describe a coding process based on information theory. Each node is assigned a Huffman codeword. Huffman code is an algorithm for assigning codewords using only 0’s and 1’s with short codes for common objects and longer codes for rarer objects [7]. Our code for a particular network will be a certain length of numbers. To minimize the overall length of our code, we can also consider groups of nodes (communities) to have unique codebooks. There are now two sets of codebooks; the codebook for communities and the codebook for nodes for each community. The codeword length for a community is based the probability of random walks in the network leaving that particular community. The codeword lengths for each node within the community is based on the probability at which random walks visit each node in the module or leaves the module. By minimizing the overall codebook length, we find the ideal division of the network based on random-walk flow.

However, Infomap does not require that we actually assign these codewords. Rather, the map equation defines the lower bound for the code length. Therefore, we only need to minimize the following map equation

$$L(M) = \left(\sum_{\beta=1}^{s} q_\beta\right)\log\left(\sum_{\beta=1}^{s} q_\beta\right) - 2\sum_{\beta=1}^{s} q_\beta \log(q_\beta) - \sum_{\alpha=1}^{n} p_\alpha \log(p_\alpha) + \left(\sum_{\beta=1}^{s} q_\beta + \sum_{\alpha\in\beta} p_\alpha\right)\log\left(\sum_{\beta=1}^{s} q_\beta + \sum_{\alpha\in\beta} p_\alpha\right)$$

where $L(M)$ is the lower bound for the map equation (the quantity we want to minimize), s is the set of current communities, n is the set of all nodes in the network, q_β is the probability of the random walk leaving community β, p_α is the probability of visiting that particular node, and $\alpha \in \beta$ indicates all nodes within model β. To find the minimum, the following iterative process is used. To start, each node is considered to be its own community

1. Calculate the map equation for the current configuration
2. In a random order, nodes are merged with the neighboring community that results in the largest decrease in the map equation from the current configuration. If there is no improvement in the map equation, the node remains in its current community
3. Step 2 is repeated until there is no additional improvement in $L(M)$ for any nodes
4. A new network is built where each community is now considered as a node. Steps 1-3 are repeated using this new network until $L(M)$ for the input network no longer decreases.
5. Step 4 is repeated until $L(M)$ no longer decreases

To improve step 4, sub-groups within each community are moved recursively to determine $L(M)$ instead of the entire community in implementation. The authors recommend that the algorithm be used multiple times (with a different random seed for each run) and use the result that has the smallest $L(M)$ over the repeated runs to avoid local minima of the map equation.

Spinglass

The Spinglass method reframes community detection as a Hamiltonian operator, the sum of kinetic energies and potential energies of all particles in a system [8]. Under this framework, particle spin-state refers to edges between nodes in the same community (same spin state) or in different communities (different spin states). The Hamiltonian for the network is a function of edges between nodes of the same group, edges between nodes of different groups, lack of edges between nodes of the same group, and lack of edges between nodes of different groups. With external versus internal links and non-links equally weighted, the Hamiltonian reduces to

$$\mathcal{H}(\{c\}) = -\sum_{i\neq j}(A_{ij} - \gamma p_{ij}) I(c_i = c_j)$$

where p_{ij} is the probability of a link between node i and node j, and γ is a tuning parameter that balances the importance of present versus missing edges in a community. Values of $\gamma < 1$ places greater

value on existing edges within a community. p_{ij} can be expressed via a variety of different expressions depending on the graph under study. In R's `igraph`, the probability can be based on either a random graph with the same number of edges as the baseline probability or following the same degree distribution as the input graph.

To determine the optimal division of the network into communities, the community division that minimizes the Hamiltonian needs to be found. To find the global minimum, a simulated annealing process is used [9]. The following algorithm is used

1. Randomly select a division of the network into communities, σ
2. Evaluate the change in Hamiltonian relative to an assumed spin-state ψ (similar to another randomly chosen community division)
$$\Delta\mathcal{H}(\psi,\sigma) = \sum_{j\neq l}(A_{lj} - \gamma p_{ij})\, I(\psi = c_j) - \sum_{j\neq l}(A_{lj} - \gamma p_{ij})\, I(\sigma = c_j)$$
3. Decide whether or not to retain the current community division based on the change in Hamiltonian and a probability.
4. As the process continues, the probability of moving to worse conditions goes to zero. This shift in probability is referred to as the annealing schedule and avoids the algorithm becoming stuck in a local optimum.
5. The algorithm terminates once a pre-specified criterion occurs. In keeping with the metallurgical language of annealing, this criterion is referred to as the stopping temperature. The criterion depends on the software.

The Spinglass algorithm allows an upper limit on number of communities to be specified. Additionally, a similar process can be used to determine the community that a single node belongs to. There are also alternative ways to formulate p_{ij}. Lastly, under the specification of p_{ij}, the expected values for communities under a random graph to determine if the network has evidence of a true underlying community structure.

References

1. Girvan, M. and M.E.J. Newman, *Community structure in social and biological networks.* Proceedings of the National Academy of Sciences, 2002. **99**(12): p. 7821-7826.
2. Newman, M., *Networks: An Introduction*. 2010: Oxford University Press, Inc. 720.
3. Pons, P. and M. Latapy. *Computing Communities in Large Networks Using Random Walks*. in *Computer and Information Sciences - ISCIS 2005*. 2005. Berlin, Heidelberg: Springer Berlin Heidelberg.
4. Raghavan, U.N., R. Albert, and S. Kumara, *Near linear time algorithm to detect community structures in large-scale networks.* Physical review E, 2007. **76**(3): p. 036106.
5. Blondel, V.D., et al., *Fast unfolding of communities in large networks.* Journal of Statistical Mechanics: Theory and Experiment, 2008. **2008**(10): p. P10008.
6. Rosvall, M., D. Axelsson, and C.T. Bergstrom, *The map equation.* The European Physical Journal Special Topics, 2009. **178**(1): p. 13-23.
7. Huffman, D.A., *A method for the construction of minimum-redundancy codes.* Proceedings of the IRE, 1952. **40**(9): p. 1098-1101.
8. Reichardt, J. and S. Bornholdt, *Statistical mechanics of community detection.* Physical Review E, 2006. **74**(1): p. 016110.
9. Kirkpatrick, S., C.D. Gelatt, and M.P. Vecchi, *Optimization by Simulated Annealing.* Science, 1983. **220**(4598): p. 671-680.

eX-FLU case study

Last Updated 09/13/2019

Load packages

```
library(igraph)
library(tidyverse)
library(magrittr)
library(readr)
```

Create Network

This code imports the original eX-FLU data and ends with the largest component of the undirected network.

```
  # Load eX-FLU data:

  # load edge file
  edgelist = read_csv("../Shared/case study data/edgelist.csv")


  # load node file
  hh = read_csv("../Shared/case study data/hh.csv")


# Join the nodes file to the edges file to make the initial directed graph
  # n=590 nodes and 2780 edges

    # create from edgelist and simple nodelist
    exflu_hh = igraph::simplify(graph_from_data_frame(edgelist,
                                              directed = TRUE,
                                              vertices = hh), remove.multiple
=TRUE)

  gsize(exflu_hh)

## [1] 2780

  gorder(exflu_hh)

## [1] 590

  summary(exflu_hh)

## IGRAPH dc1448a DN-- 590 2780 --
## + attr: name (v/c), wash_num (v/n), wash_time (v/n), washopt (v/n)

# Adjust other graph options so later graphs will inherit these - make it nic
er for plotting, etc.
  V(exflu_hh)$label = NA
  E(exflu_hh)$arrow.size = 0.25
```

```
  V(exflu_hh)$size = 4
  V(exflu_hh)$label.cex=0.5
  V(exflu_hh)$label.dist=2
  V(exflu_hh)$color = "darkgray"


# Create layout
  #set.seed(432197)
  #n590_layout = layout_with_fr(exflu_hh)
  #plot(exflu_hh, layout=n590_layout)
  #save(n590_layout,file="../Shared/results_graphs/n590-fr-layout.RData")
  load(file="../Shared/results_graphs/n590-fr-layout.RData")

  # Add the layout to the graph
  exflu_hh$layout = n590_layout

  # This layout won't remain when dropping nodes from the graph, so need expl
icit x and y coords also
  V(exflu_hh)$layoutx = n590_layout[,1]


  V(exflu_hh)$layouty = n590_layout[,2]


# Make undirected

  # the "mutual" argument creates one undirected edge for each pair of vertic
es that are connected by a
   #mutual (reciprocal) edge
  new = as.undirected(exflu_hh, mode="mutual")

# Subset to the largest component

  # Get a list of components
  comps = igraph::decompose(new)

  # Get the size of the largest component
  max_size = max(sapply(comps, gorder, simplify=TRUE))

  # Vector that identifies which of the components is that max size (vector==
TRUE)
  subset = sapply(comps, function(z) gorder(z) == max_size)

  # Pick the component that is the max size (TRUE in the subset vector)
    # the [[1]] ensures that it is an igraph object
  exflu_undir_mutual = comps[subset][[1]]
  print(class(exflu_undir_mutual))

## [1] "igraph"
```

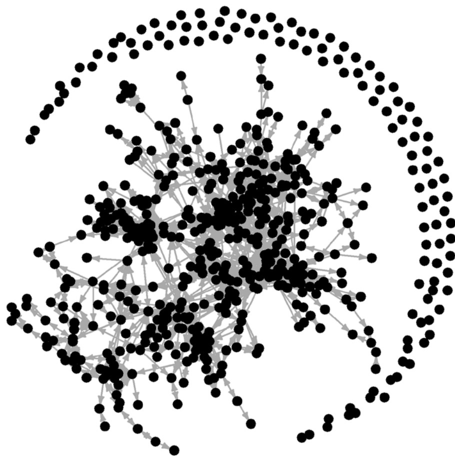
```
  # Re-add the consistent
  exflu_undir_mutual$layout = cbind(V(exflu_undir_mutual)$layoutx, V(exflu_un
dir_mutual)$layouty)

  summary(exflu_undir_mutual)

## IGRAPH dc1e5c9 UN-- 314 880 --
## + attr: layout (g/n), name (v/c), wash_num (v/n), wash_time (v/n),
## | washopt (v/n), label (v/l), size (v/n), label.cex (v/n),
## | label.dist (v/n), color (v/c), layoutx (v/n), layouty (v/n)

  # removing unneeded things
  rm(comps, edgelist, hh, n590_layout, new, max_size, subset)
```

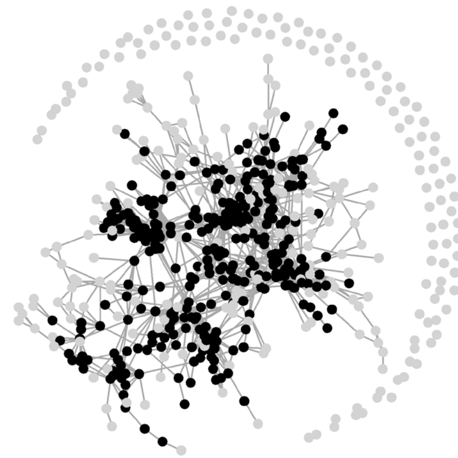
Graph Input Network

This code graphs the largest component of the undirected network compared to the original directed eX-FLU network.

```
# 314 nodes in the mutual network should be colored black
  summary(exflu_undir_mutual)

## IGRAPH dc1e5c9 UN-- 314 880 --
## + attr: layout (g/n), name (v/c), wash_num (v/n), wash_time (v/n),
## | washopt (v/n), label (v/l), size (v/n), label.cex (v/n),
## | label.dist (v/n), color (v/c), layoutx (v/n), layouty (v/n)

  # change node colors based on node NAME (NOT IGRAPH ID - THESE BECOME 1:N A
ND AREN'T INFORMATIVE)
  V(exflu_hh)$mutual = if_else(V(exflu_hh)$name %in% V(exflu_undir_mutual)$na
me,
                                       "black", "lightgray")

  # there are 314 black nodes based on this attribute
  table(V(exflu_hh)$mutual)

##
##     black lightgray
##       314       276

plots = function(GRAPH, TITLE, VCOLOR, ARROW){
  par(mar=c(3.5, 3.5, 2, 1))
  plot(GRAPH, main=TITLE, vertex.color=VCOLOR, vertex.frame.color=VCOLOR, edg
e.color="darkgray", edge.arrow.size=ARROW)
}


png("../Shared/results_graphs/Total vs. Mutual Networks.png", width=8, height
=6, units="in", res=500)
par(mfrow=c(1,2))

  plots(exflu_hh, "eX-FLU: Directed \n 590 nodes", "black", 0.25)
```

```
  plots(exflu_hh, "eX-FLU: Mutual \n 314 nodes", V(exflu_hh)$mutual, 0)

dev.off()

## png
##   2

rm(plots, exflu_hh)

include_graphics("../Shared/results_graphs/Total vs. Mutual Networks.png")
```





Detect Communities

```
mutual_results = list()

set.seed(12101992)

("weight" %in% edge_attr_names(exflu_undir_mutual))

## [1] FALSE

# confirmed that I do not have an edge attribute called 'weight' so none of t
hese algorithms are using it by default
```

```
# NOTE THE DIFFERENCES WITH NA VS. NULL FOR OVER-RIDING EDGEWEIGHTS IF YOU HA
VE A WEIGHT ATTRIBUTE
mutual_results[[1]] = cluster_edge_betweenness(exflu_undir_mutual, directed =
FALSE, weights=NULL)

mutual_results[[2]] = cluster_infomap(exflu_undir_mutual, e.weights=NA)

mutual_results[[3]] = cluster_walktrap(exflu_undir_mutual, weights=NULL)

mutual_results[[4]] = cluster_louvain(exflu_undir_mutual, weights=NA)
mutual_results[[5]] = cluster_label_prop(exflu_undir_mutual, weights=NA)
mutual_results[[6]] = cluster_spinglass(exflu_undir_mutual, weights = NA)

names(mutual_results) = c("edgebetween", "infomap", "walktrap", "multilevello
uvain", "labelprop", "spinglass")



#set_vertex_attr(input, "edgebetween", value=membership(results[[1]]))
# to query the name of the algorithm: names(mutual_results)[1]
# to get the membership of each node: membership(mutual_results[[1]])


  for (i in 1:6){
      exflu_undir_mutual = set_vertex_attr(exflu_undir_mutual, names(mutual_r
esults)[i], value=membership(mutual_results[[i]]))
  }


rm(i)
```

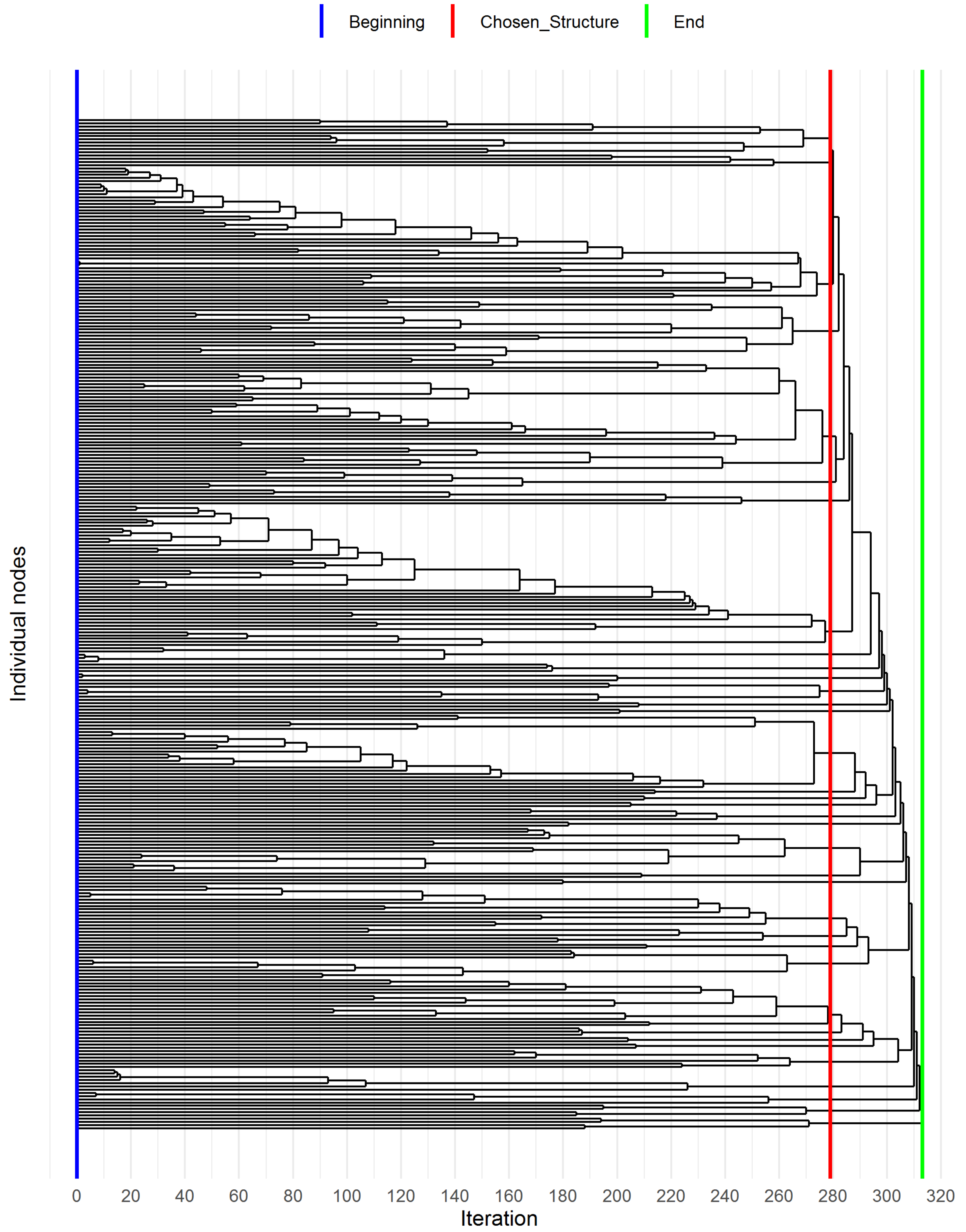
Walktrap Dendrogram (Figure 1)

This code visualizes the walktrap dendrogram and shows where the algorithm begins and ends.

The ‘chosen structure’ (i.e., highest modularity) has 35 communities, which corresponds to the community structure returns on the 279th iteration of the algorithm (if we begin on iteration 0).

```
library(ggdendro)

tree = as.dendrogram(mutual_results[["walktrap"]])
dendata = dendro_data(tree)

#dendata$segments$x
ggplot(data=dendata$segments, horiz=TRUE) +
  geom_segment(aes(x=y, y=x, xend=yend, yend=xend)) +
  geom_vline(aes(xintercept = 279, col="Chosen_Structure"), linetype="solid",
size=1) +
  geom_vline(aes(xintercept = 0, col="Beginning"), linetype="solid", size=1)
+
```

```
  geom_vline(aes(xintercept = 313, col="End"), linetype="solid", size=1) +
  scale_colour_manual(name="", values=c(Chosen_Structure="red", Beginning="bl
ue", End="green")) +
  theme_minimal() +
  scale_y_continuous(breaks=NULL) +
  scale_x_continuous(breaks=c(seq(0, 320, 20))) +
  theme(legend.position = "top") +
  labs(#title="Figure 1: Dendrogram for Walktrap",
       #subtitle="Chosen structure corresponds to the community structure wit
h optimal modularity",
       y="Individual nodes",
       x="Iteration")

ggsave("../Shared/results_graphs/Figure 1 - Walktrap Dendrogram.png", plot=la
st_plot(), device="png", dpi=300, height=9, width=7)

rm(tree, dendata)

include_graphics("../Shared/results_graphs/Figure 1 - Walktrap Dendrogram.png
")
```



Beginning
Chosen_Structure
End
Individual nodes
0
20
40
60
80
100
120
140
160
180
200
220
240
260
280
300
320
Iteration

Describe Communities

This code creates a small table to show descriptive statistics for communities under each algorithm.

```
library(rlist)

# for each community result stored in mutual_results, get the sizes of those
communities
  comm_sizes = lapply(mutual_results, sizes)
  # each result has a list of the 'sizes' of the communities

  # For each list of community sizes, get some descriptive statistics
    n = unlist(lapply(comm_sizes, length)) # how many communities
    mean = unlist(lapply(comm_sizes, mean)) # mean size of communities
    median = unlist(lapply(comm_sizes, median)) # median size of communities
    min = unlist(lapply(comm_sizes, min)) # minimum size of communities
    max = unlist(lapply(comm_sizes, max)) # max size of communities

  # combine those into a dataframe
    desc = cbind(n, mean, median, min, max)

kable(desc)
```

	n	mean	median	min	max
edgebetween	16	19.625000	15.5	4	46
infomap	38	8.263158	6.0	2	35
walktrap	35	8.971429	4.0	2	44
multilevellouvain	13	24.153846	22.0	5	43
labelprop	21	14.952381	8.0	3	81
spinglass	17	18.470588	17.0	5	43

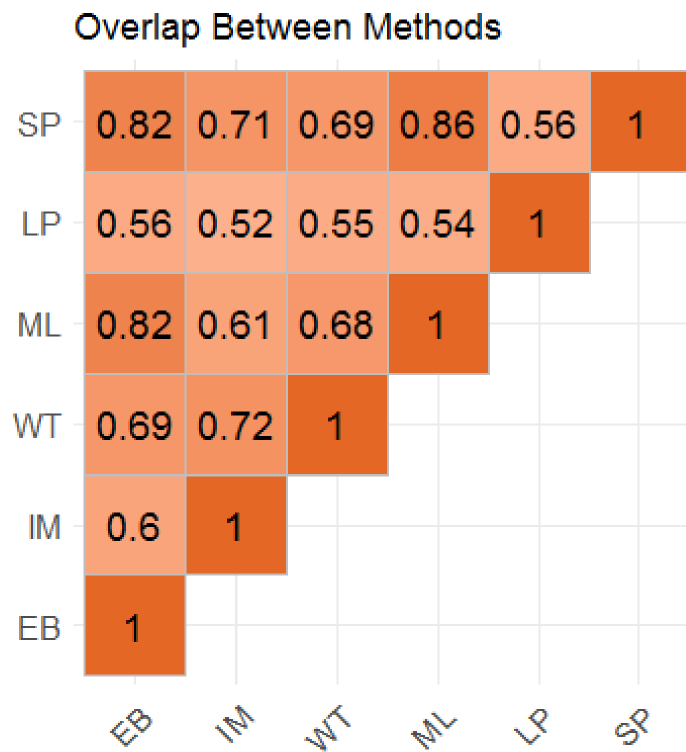
```
write.csv(desc, file="../Shared/results_tables/Community_Descriptives.csv")

rm(comm_sizes, desc, n, mean, median, min, max)
```

Community Overlap

This code calculates the amount of overlap between two community structures using the adjusted rand index, which ranges from 0 to 1.

```
library(ggcorrplot)
library(ggpubr)


tmp = mutual_results
names(tmp) = c("EB", "IM", "WT", "ML", "LP", "SP")

# for each pairwise combination of results, apply the comparison method using
```

```
the adjusted rand index
correlations = sapply(tmp,
                      function(x) sapply(tmp, function(y) compare(x,y, method
='adjusted.rand')))
p = ggcorrplot(correlations, type='upper',
               colors = c("#6D9EC1", "white", "#E46726"),
               show.diag = TRUE, lab=TRUE, show.legend = FALSE, title="Overla
p Between Methods", lab_size = 5)

print(p)
```




```
ggsave("../Shared/results_graphs/Algorithm Overlap - adjusted Rand.png", plot
=last_plot(), device="png", dpi=500)

## Saving 5 x 4 in image

rm(tmp, p, correlations)
```

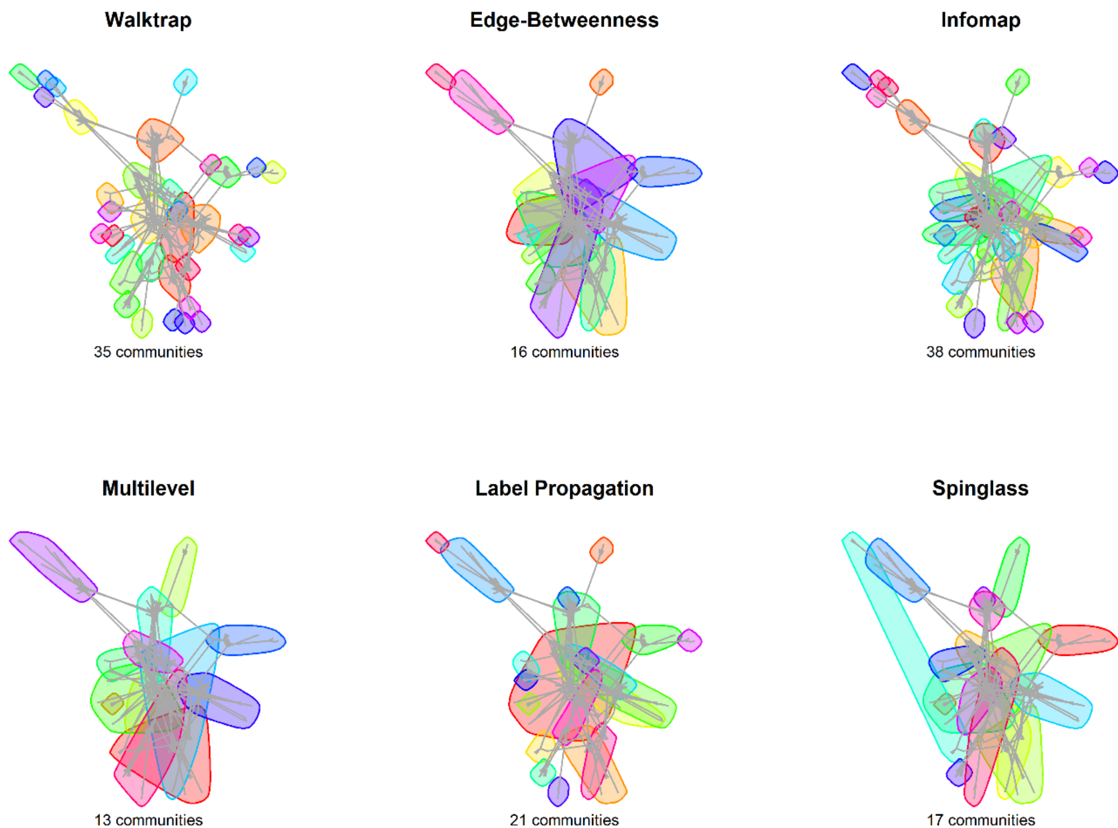
Graph Community Structures to show discrepancies (Figure 2)

This code graphs the community structure returned by each algorithm on the same layout, which can help visualize how the different algorithms return different results.

```
# get edgelist from graph object
  edges_tmp <- igraph::as_data_frame(exflu_undir_mutual, what="edges")

# get nodelist from graph object - individuals
```

```
  nodes_tmp <- igraph::as_data_frame(exflu_undir_mutual, what="vertices")

# keep only the walktrap assignment
  nodes_wt = nodes_tmp %>% select(name, walktrap)


# modify the original edgelist to determine if individuals are in the SAME wa
lktrap community
  # this is done by merging the walktrap community assignments in 'nodes_wt'
onto the edgelist for both nodes (i.e., 'from' and 'to')
  edges_tmp2 =

    # what is the community number of the 'from' ID?
    left_join(edges_tmp, nodes_wt, by=c("from"="name")) %>%
    rename(walktrap_from=walktrap) %>%

    # what is the community number of the 'to' ID?
    left_join(nodes_wt, by=c("to"="name")) %>%
    rename(walktrap_to=walktrap) %>%

    # if they are in the same community, upweight that edgeweight dramaticall
y
    # this will improve the look of the graph -- the weight will be used in t
he layout so that people within
      # the same community are much closer together and make the graph look l
ess messy
    mutate(weight = if_else(walktrap_from==walktrap_to,
                            25, 1))

# Create a graph object
  tmp <- graph_from_data_frame(edges_tmp2, directed=FALSE, nodes_tmp)

# Adjust other graph options so later fraphs will inherit these - make it nic
er for plotting, etc.
  V(tmp)$label = NA
  E(tmp)$arrow.size = 0.25
  V(tmp)$size = 4
  V(tmp)$label.cex=0.5
  V(tmp)$label.dist=2
  V(tmp)$color = "darkgray"


# layout this graph using the edgeweights. calling 'layout consistent' becaus
e this layout will be used to graph all community structures
  layout_consistent=layout_with_fr(tmp, weights=E(tmp)$weight)


# function to graph the communities from each algorithm using the same layout
GRAPH_COMMUNITIES = function(ALGORITHM){
```

```
  # take input argument and make it useful for the function
    algorithm = enquo(ALGORITHM)
    character = deparse(substitute(ALGORITHM))

  # Make the titles
    if (character=="edgebetween"){
      title="Edge-Betweenness"
    } else if (character=="multilevellouvain"){
      title="Multilevel"
    } else if (character=="labelprop"){
      title="Label Propagation"
    } else {
      title = str_to_title(character)
    }

  # Actually plot the communities
    plot(mutual_results[[character]], tmp, layout=layout_consistent,
         vertex.frame.color="darkgray",
         vertex.color="darkgray",
         vertex.size=0,
         edge.color="darkgray",
         main=title)
  # Add text on the number of communities
    mtext(side=1, paste(length(mutual_results[[character]]), "communities"),
cex=0.6)

}



png("../Shared/results_graphs/New Fig 2.png", width=8, height=6, units="in",
res=500)
par(mfrow=c(2,3))

GRAPH_COMMUNITIES(walktrap)
GRAPH_COMMUNITIES(edgebetween)
GRAPH_COMMUNITIES(infomap)
GRAPH_COMMUNITIES(multilevellouvain)
GRAPH_COMMUNITIES(labelprop)
GRAPH_COMMUNITIES(spinglass)

dev.off()

## png
##   2

rm(GRAPH_COMMUNITIES, edges_tmp, edges_tmp2, layout_consistent, nodes_tmp, no
des_wt, tmp)

include_graphics("../Shared/results_graphs/New Fig 2.png")
```



Choose communities to intervene (Figure 3)

This code uses a function to determine which communities to intervene on under each algorithm, and an alternative selection procedure (residence hall).

```
  # Applicable to all algorithms
  # get edgelist from exflu_undir_mutual
  edges_tmp <- igraph::as_data_frame(exflu_undir_mutual, what="edges")

  # get nodelist from exflu_undir_mutual
  nodes_tmp <- igraph::as_data_frame(exflu_undir_mutual, what="vertices")


# Create a function to choose the communities that meet our criteria
CHOOSE_COMMS = function(ALGORITHM){

  # modify input argument to be useful for the function
  algorithm=enquo(ALGORITHM)
  character=deparse(substitute(ALGORITHM))


  # Make a 'community level network' to determine how connected a community i
```

```
s

  # Step 1: make an community-level edgelist
    # Keep just the ID and algorithm assignment within the nodelist
    name_algorithm <- nodes_tmp %>% select(name, !!algorithm)


    # Create new edgelist
    edges_tmp_algorithm <- left_join(edges_tmp, name_algorithm, by=c("from"="
name")) %>%
      dplyr::rename(V1=!!algorithm) %>%
      left_join(name_algorithm, by=c("to"="name")) %>%
      dplyr::rename(V2=!!algorithm) %>%
      select(-from, -to) %>%
      group_by(V1, V2) %>%
      dplyr::summarise(weight=n()) %>%
      ungroup()

  # Step 2: Make an community-level nodelist
    # With the prevalence of optimal/suboptimal HH and community size as attr
ibutes
    nodes_tmp_algorithm <- nodes_tmp %>%
      group_by(!!algorithm) %>%
      dplyr::summarise(n_comm = n(),
                       prop_opt = round(mean(washopt, na.rm=TRUE), digits=2),
                       prop_not_opt = 1-prop_opt)
    #str(nodes_tmp_algorithm)

  # Step 3: Combine into an community-level graph
  graph_tmp <- graph_from_data_frame(edges_tmp_algorithm, directed=FALSE, ver
tices=nodes_tmp_algorithm)
  #summary(graph_tmp)
  #summary(E(graph_tmp)$weight)

  # Simplify to get rid of duplicate edges
  graph_tmp_simplified <- igraph::simplify(graph_tmp)
  #summary(graph_tmp_simplified)
  #summary(E(graph_tmp_simplified)$weight)

  # Get degree for use in decision rule
  V(graph_tmp_simplified)$degree_centrality = degree(graph_tmp_simplified)

  # Output the community-level nodelist
  communities <- igraph::as_data_frame(graph_tmp_simplified, what="vertices")
  #str(communities)

  # Now, we can choose the communities to intervene on according to our decis
ion rule
  chosen <- communities %>%
```

```
    mutate(eligible = if_else(prop_not_opt > 0.80 & # prevalence of suboptima
l HH at least 80%
                                n_comm >= 5 & # at least 5 people in the comm
unity
                                degree_centrality >= 5, # and connected to at 
least 5 other communities
                              1,
                              0)) %>%

    # keep only those communities that are eligible
    filter(eligible==1) %>%

    # arrange dataset by the proportion of suboptimal, the number in eligible 
comms, and degree centrality
    # this is essentially our 'priority' ranking of the decision criteria -- 
it doesn't make a difference for our
    # analyses here, but could be helpful for others
    # the arrangement will matter for the cumulative sum below, which would i
n theory be used to choose communities from the total pool of eligible commun
ities
    # for example... if you had lots of eligible communities and exceeded 50 
people, you'd only pick the communities that came first in this ordering unti
l you got to 50 (or the closest number below it without exceeding)
    dplyr::arrange(desc(prop_not_opt), desc(n_comm), (degree_centrality)) %>%

    # create a running total of the number who are in the eligible communitie
s
    mutate(cum_sum = cumsum(n_comm),

           # choose the communities so that we don't exceed intervening on 50 
people
           chosen = if_else(cum_sum<=50, 1, 0),
           algorithm=character) %>%


    select(algorithm, name, eligible, prop_not_opt, n_comm, degree_centrality
, n_comm, prop_opt, cum_sum, chosen)# %>%

    # don't need to actually choose the chosen ones -- because cumulative sum 
is always below 50
    #filter(chosen==1)
    print(paste("We don't need to subset from all eligible communities becaus
e the total number of people in eligible communities is ", max(chosen$cum_sum
), " which is less than 50, our max number to intervene on", sep=""))

  return(chosen)

}
```

```
WT = CHOOSE_COMMS(walktrap)

## [1] "We don't need to subset from all eligible communities because the tot
al number of people in eligible communities is 39 which is less than 50, our
max number to intervene on"

EB = CHOOSE_COMMS(edgebetween)

## [1] "We don't need to subset from all eligible communities because the tot
al number of people in eligible communities is 42 which is less than 50, our
max number to intervene on"

IM = CHOOSE_COMMS(infomap)

## [1] "We don't need to subset from all eligible communities because the tot
al number of people in eligible communities is 26 which is less than 50, our
max number to intervene on"

ML = CHOOSE_COMMS(multilevellouvain)

## [1] "We don't need to subset from all eligible communities because the tot
al number of people in eligible communities is 21 which is less than 50, our
max number to intervene on"

LP = CHOOSE_COMMS(labelprop)

## [1] "We don't need to subset from all eligible communities because the tot
al number of people in eligible communities is 22 which is less than 50, our
max number to intervene on"

SG = CHOOSE_COMMS(spinglass)

## [1] "We don't need to subset from all eligible communities because the tot
al number of people in eligible communities is 33 which is less than 50, our
max number to intervene on"

    load("../Shared/case study data/res_hall.RData")

    res_hall_mod = res_hall %>% mutate(name=as.character(STUDY_ID)) %>% selec
t(name, RES_HALL)

    res_hall_hh = left_join(nodes_tmp, res_hall_mod, by="name") %>%
      filter(RES_HALL != 555) %>%
      group_by(RES_HALL) %>%
      dplyr::summarise(n_reshall = n(),
                       prop_opt = round(mean(washopt, na.rm=TRUE), digits=2),
                       prop_not_opt = 1-prop_opt) %>%
      ungroup() %>%
      dplyr::mutate(max = max(prop_not_opt),
                    chosen = if_else(prop_not_opt==max, 1, 0)) %>%
      filter(chosen==1) %>%
```

```
      dplyr::mutate(algorithm="reshall",
                    n_comm = n_reshall,
                    degree_centrality=0)




tmp = bind_rows(WT, EB, IM, ML, LP, SG, res_hall_hh) %>%
  dplyr::mutate(Algorithm = factor(algorithm, levels = c("walktrap", "edgebet
ween", "infomap",
                                                        "multilevellouvain",
"labelprop", "spinglass", "reshall"),
                                   labels = c("walktrap" = "Walktrap", "edgeb
etween"="Edge-Betweeness",
                                              "infomap" = "Infomap",
                                              "multilevellouvain" = "Multilev
el",
                                              "labelprop" = "Label Propagatio
n",
                                              "spinglass" = "Spinglass",
                                              "reshall" = "Residence Hall")),


                # MANUAL JITTERING -- there are a few communities with the ex
act same suboptimal proportion and degree centrality
                new_degree_centrality = if_else(# for this one, move the info
map dot up bit. leaving the spinglass dot alone
                                                prop_not_opt==1.00 & degree_c
entrality==5 & algorithm=="infomap",
                                                degree_centrality + 0.2,

                                                # for this pair, move the edg
e-betweenness dot up and the spinglass dot down
                                                if_else(prop_not_opt==0.87 &
degree_centrality==6 & algorithm=="edgebetween",
                                                        degree_centrality + 0
.2,
                                                        if_else(prop_not_opt=
=0.87 & degree_centrality==6 & algorithm=="spinglass",
                                                                degree_centra
lity - 0.2,

                                                                # for all oth
er dots, leave them as they are
                                                                degree_centra
lity))))



colours_outside = c("Walktrap" = "black",
```

```
                    "Edge-Betweeness" = "black",
                    "Infomap" = "black",
                    "Multilevel" = "black",
                    "Label Propagation" = "black",
                    "Spinglass" = "black",
                    "Residence Hall" = "black")

colours_inside = c("Walktrap" = "red",
                      "Edge-Betweeness" = "gray",
                      "Infomap" = "white",
                      "Multilevel" = "gray",
                      "Label Propagation" = "white",
                      "Spinglass" = "gray",
                      "Residence Hall" = "black")

shape_algorithm = c("Walktrap" = 21,
                    "Edge-Betweeness" = 21,
                    "Infomap"=21,
                    "Multilevel" = 25,
                    "Label Propagation"=25,
                    "Spinglass" = 23,
                    "Residence Hall" = 23)

#tmp = tmp %>% filter(Algorithm != "Residence Hall")


ggplot(data=tmp, aes(x=prop_not_opt,
                     y=new_degree_centrality)) +
  geom_point(aes(fill=Algorithm,
                 pch=Algorithm,
                 colour=Algorithm),
             size=sqrt(tmp$n_comm)) +
  scale_shape_manual(values=shape_algorithm) +
  scale_fill_manual(values=colours_inside) +
  scale_colour_manual(values=colours_outside) +

  labs(x = "Prevalence of Suboptimal Hand Hygiene",
       y = "Connectedness (Degree)")+
  theme_bw() +
  theme(legend.position=c(0.85,0.78),
        legend.background = element_rect(size=0.5, linetype="solid",
                                         colour ="black"),
        legend.title = element_blank(),
        legend.key.size = unit(1, 'lines')) +
  scale_y_continuous(breaks=c(seq(0,10,2)), limits=c(0,10))

ggsave("../Shared/results_graphs/Fig 3 Decision Rule.png", plot=last_plot(),
device="png", dpi=500, width=8, height=6, units="in")
```

```
totals = tmp %>% dplyr::group_by(algorithm) %>%
dplyr::summarise(n = sum(n_comm))

rm(EB, edges_tmp, IM, LP, ML, nodes_tmp, res_hall, res_hall_hh, res_hall_mod,
SG, tmp, totals, WT, colours_inside, colours_outside, shape_algorithm, CHOOSE
_COMMS)

include_graphics("../Shared/results_graphs/Fig 3 Decision Rule.png")
```