Graph Partitioning Exploration

Kernel k-means and ratio cut equivalence proposition

We would like to show that these two statements are equivalent:

$$\text{argmin}_H \left\{ \text{Tr}(N^{-1/2} H L H^T N^{-1/2}) \right\}$$

$$\text{argmax}_H \left\{ \text{Tr}(N^{-1/2} H L^\dagger H^T N^{-1/2}) \right\}$$

constrainted to H being of a specific form (cluster assignment matrix).

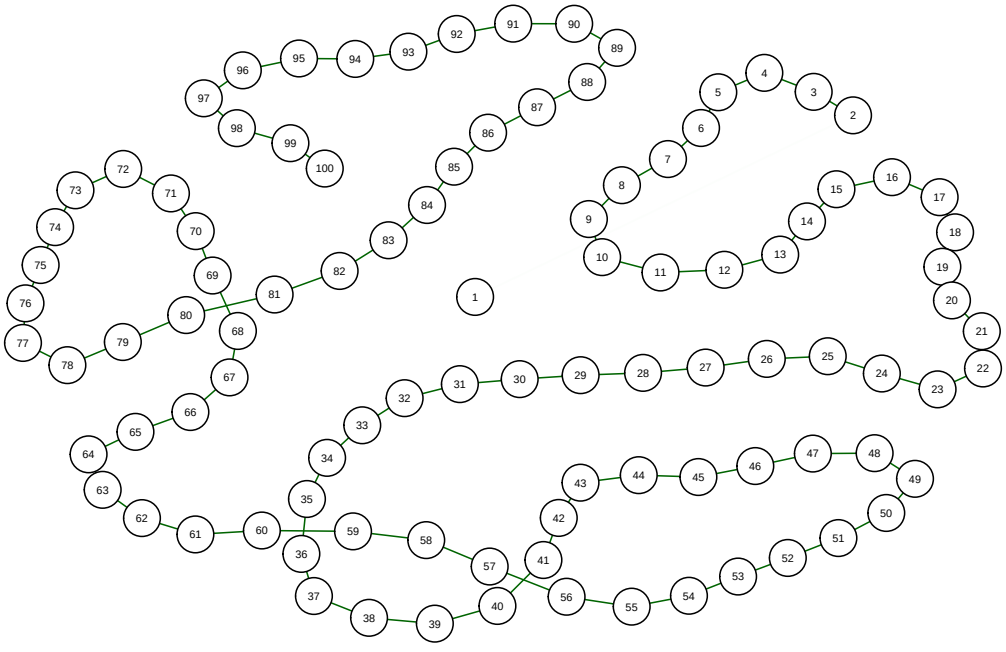
The motivation behind this proposition is that if we were to solve the *relaxed* versions of each of these problems, where H is free, we arrive at the same results.

Counterexample

Construct a graph of 100 vertices and 99 edges such that there is an edge between vertices 1 and 2, an edge between vertices 2 and 3, etc. Let the weight between vertices 1 and 2 be 1, with the rest of the edges having weight 100.

```
# packages, etc.
library(ggplot2)
import::from(magrittr, `%>%`, `%<>%`)
import::from(psych, tr)
import::from(qgraph, qgraph)
source('http://pages.iu.edu/~mtrosset/Courses/675/manifold.r')

theme_set(theme_bw())

#' @title construct H function
#' @description H is constructed based on a vector that designates the clusters
#' @param clustering (numeric) A vector of cluster assignments
#' @return (matrix) H based on the cluster assignments
construct.H <- function(clustering, cluster.max = max(clustering)) {
  clusters <- seq(max(cluster.max))

  if (min(clustering) < 1) {
    stop(simpleError('cluster indexing starts at 1'))
  }

  # find |A_k|
  cluster.sizes <- sapply(clusters, function(i) {
    length(clustering[clustering == i])
  })

  # construct H
  H <- sapply(clustering, function(i) {
    h <- rep(0, length(clusters))
    h[i] <- 1 / sqrt(cluster.sizes[i])
```

```
    return(h)
  })

  return(H)
}

kernel.kmeans.obj <- function(K, clustering, cluster.max = max(clustering)) {
  # compute the objective for kernel k-means
  H <- construct.H(clustering, cluster.max)
  psych::tr(H %*% K %*% t(H))
}

ratio.cut.obj <- function(L, clustering, cluster.max = max(clustering)) {
  # compute the objective for ratio cut
  kernel.kmeans.obj(L, clustering, cluster.max)
}

n <- 100
size <- 100

W <- matrix(rep(0, n ** 2), nrow = n)
for (i in seq(2, n - 1)) W[i, i + 1] <- W[i + 1, i] <- size
W[1, 2] <- W[2, 1] <- 1

qgraph(W, layout = 'spring')
```



Then we can see that there are 99 possible cuts. This is a small enough problem to iterate through.

```r
# construct graph laplacian
L <- graph.laplacian(W)

# compute pseudoinverse
L.dagger <- MASS::ginv(L)

# iterate through all n - 1 cuts
out.df <- lapply(seq(n - 1), function(i) {
  # make the cut
  clustering <- c(rep(1, i), rep(2, n - i))

  # construct cluster assignment matrix
  H <- construct.H(clustering)

  # compute objectives
  ratio.cut <- tr(H %*% L %*% t(H))
  k.means <- tr(H %*% L.dagger %*% t(H))

  # output
  dplyr::data_frame(n = n, k = i,
                    w = W[i, i + 1],
                    ratio.cut = ratio.cut,
                    k.means = k.means)
}) %>%
  dplyr::bind_rows()

# compare ratio cut and k-means
which.min(out.df$ratio.cut)
```

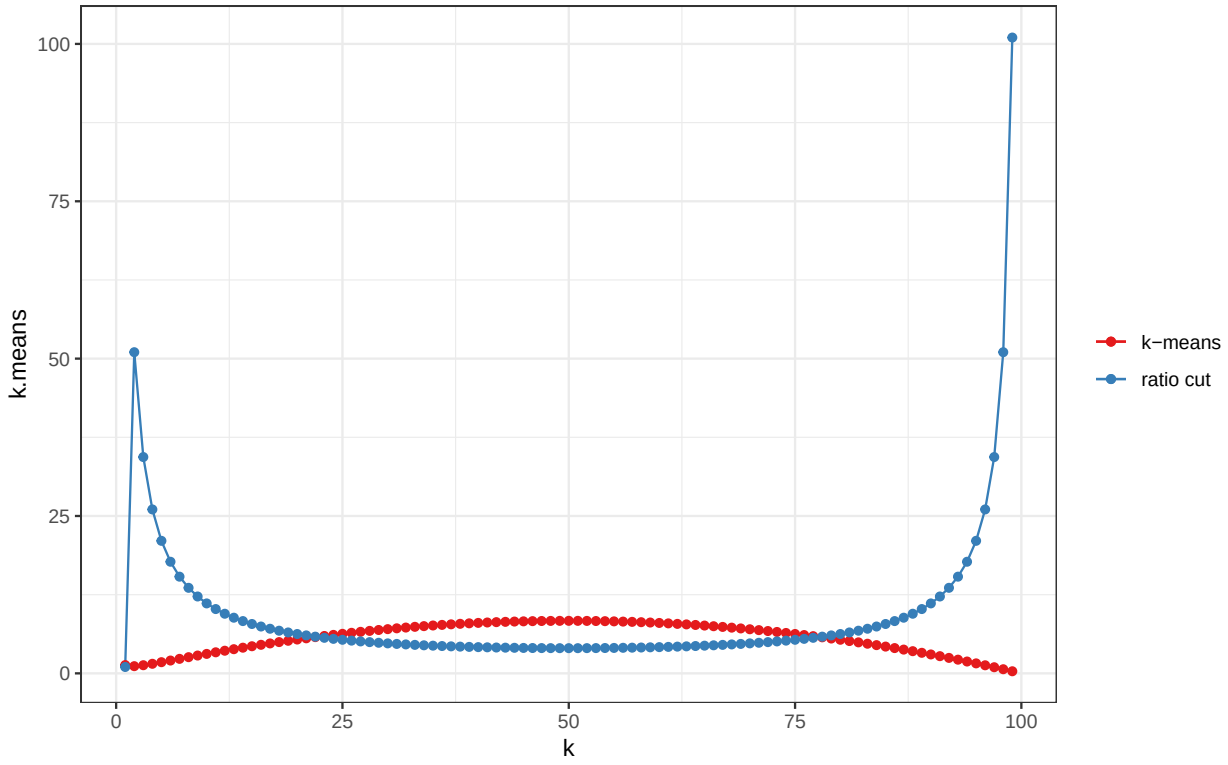
```
[1] 1
```

```r
which.max(out.df$k.means)
```

```
[1] 50
```

The ratio cut objective is minimized when we cut the edge connecting vertices 1 and 2, while the k-means objective is maximized when we cut the edge connecting vertices 50 and 51. However, it could be that this is some degenerate case.

```r
ggplot(out.df) +
  geom_line(aes(x = k, y = k.means, colour = 'k-means')) +
  geom_line(aes(x = k, y = ratio.cut, colour = 'ratio cut')) +
  geom_point(aes(x = k, y = k.means, colour = 'k-means')) +
  geom_point(aes(x = k, y = ratio.cut, colour = 'ratio cut')) +
  labs(colour = NULL) +
  scale_colour_brewer(palette = 'Set1')
```



From previous explorations, we saw that kernel k-means behaves a bit more nicely than ratio cut. In particular, the ratio cut objective tends to have many local minima, while the k-means objective tends to be smoother.

Solving ratio cut and kernel k-means

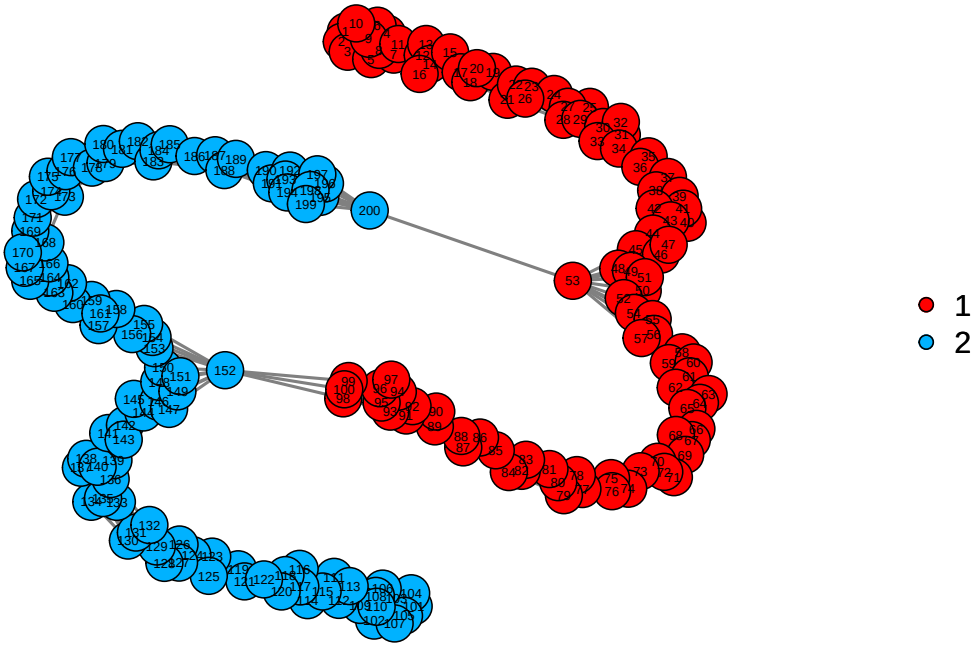
Previously, we saw that solving the relaxed forms of the ratio cut and kernel k-means optimization problems via spectral embedding and then using k-means clustering on the embedding doesn't always produce what we want it to produce. In particular, solving regular (Euclidean) k-means on the embedding results in a different optimum than solving kernel k-means or ratio cut directly. So we might as well try to solve them directly. However, this is a difficult problem since they are discrete optimization functions without gradients.

We will again use the double spiral graph to test our algorithms. Note that we are not particularly interested in the spirals themselves—what we really want is a "best" partitioning of the resultant graph.

```
# parameters
set.seed(112358)
eps <- 2 ** -2
k <- 10  # for constructing the knn graph
rad.max <- 10
ang.max <- 2 * pi
angles <- seq(0, ang.max, length.out = 100)
radii <- seq(1, sqrt(rad.max), length.out = 100) ** 2

# data
spiral.df <- dplyr::data_frame(X = radii * cos(angles),
                               Y = radii * sin(angles))
spiral.df <- dplyr::data_frame(X = radii * cos(angles),
                               Y = radii * sin(angles))
```

```
neg.spiral.df <- dplyr::mutate(spiral.df,
                               X = -X, Y = -Y,
                               id = '2')
spiral.df %<>%
  dplyr::mutate(id = '1') %>%
  dplyr::bind_rows(neg.spiral.df) %>%
  dplyr::mutate(X = X + rnorm(n = n(), sd = eps),
                Y = Y + rnorm(n = n(), sd = eps))

# construct W
W <- spiral.df %>%
  dplyr::select(X, Y) %>%
  as.matrix() %>%
  mds.edm1() %>%
  graph.knn(k) %>%
  graph.adj()

# viz
qgraph(W, groups = spiral.df$id,
       title = 'kNN graph of the double spiral',
       node.width = 2)
```

kNN graph of the double spiral



Exchange methods

This is inspired by k-means exchange methods (which are rather inefficient) and is also very similar to an algorithm outlined by Dhillon et al.[1] The algorithm is as follows (for kernel k-means:

[1] https://pdfs.semanticscholar.org/0f28/993f09606a3e3f6c5c9b6d17138f27d85069.pdf

Input: K kernel matrix, k number of clusters, $(C_1^0, ..., C_k^0)$ initial clustering

Output: $(C_1, ..., C_k)$ final clustering

Algorithm: Until convergence:

1. Compute the k-means objective with the current clustering.
2. For each vertex v_i:
    a. For each cluster C_j, compute the k-means objective assigning v_i to C_j.
    b. Choose the cluster assignment C_j^* for v_i that maximizes the k-means objective

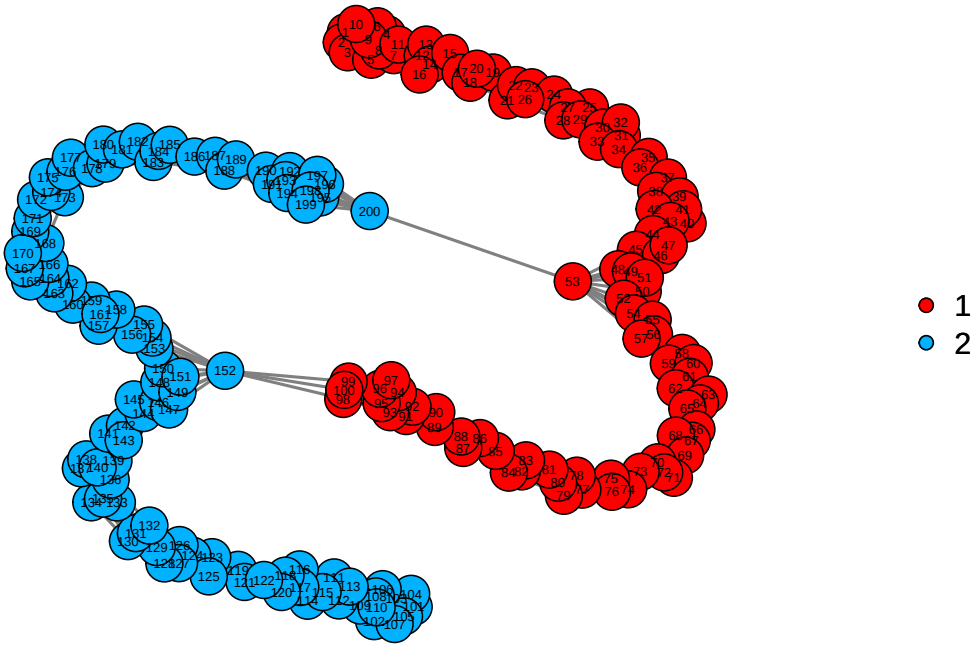
The algorithm is the same for ratio cut, except instead of computing the k-means objective we compute the ratio cut objective, and instead of maximizing the objective we minimize it.

```
kernel.kmeans.exchange <- function(K, k = 2,
                                   initial.clust = NULL,
                                   randomize = FALSE,
                                   max.iter = 1e3,
                                   verbose = FALSE) {
  # number of indices
  n <- nrow(K)
  ind <- seq(n)

  # if no initial clustering is specified, create one
  # else make sure the initial clustering is valid
  if (is.null(initial.clust)) {
    clustering <- c(seq(k), sample(seq(k), n - k, replace = TRUE))
  } else {
    assertthat::assert_that(max(initial.clust) <= k)
    assertthat::assert_that(all(sort(unique(initial.clust)) == seq(k)))
    clustering <- initial.clust
  }

  # kmeans criterion
  k.means <- kernel.kmeans.obj(K, clustering, cluster.max = k)
  k.means.prev <- 0

  # iteration counter
  iter <- 0

  while (k.means.prev < k.means) {
    # update kmeans criterion
    k.means.prev <- k.means

    # break if too many iterations
    if (iter > max.iter) {
      warning('failed to converge')
      break
    }

    # shuffle the indices if specified
    if (randomize) {
      ind <- sample(ind)
    }
```

```
    # cycle through the objects
    for (i in ind) {
      # figure out the cluster assignment for this index
      cur.clust.assign <- clustering[i]
      clust.assign.to.try <- seq(k)[-cur.clust.assign]
      k.means.try <- sapply(clust.assign.to.try, function(j) {
        temp.clustering <- clustering
        temp.clustering[i] <- j
        kernel.kmeans.obj(K, temp.clustering, cluster.max = k)
      })
      if (max(k.means.try) > k.means) {
        k.means <- max(k.means.try)
        clustering[i] <- clust.assign.to.try[which.max(k.means.try)]
      }
    }

    # update iteration counter
    iter <- iter + 1
    if (verbose) print(k.means)
  }

  return(list(k.means = k.means,
              clustering = clustering,
              n.iter = iter))
}

ratio.cut.exchange <- function(L, k = 2,
                               initial.clust = NULL,
                               randomize = FALSE,
                               max.iter = 1e3,
                               verbose = FALSE) {
  # number of indices
  n <- nrow(L)
  ind <- seq(n)

  # if no initial clustering is specified, create one
  # else make sure the initial clustering is valid
  if (is.null(initial.clust)) {
    clustering <- c(seq(k), sample(seq(k), n - k, replace = TRUE))
  } else {
    assertthat::assert_that(max(initial.clust) <= k)
    assertthat::assert_that(all(sort(unique(initial.clust)) == seq(k)))
    clustering <- initial.clust
  }

  # ratio cut criterion
  ratio.cut <- ratio.cut.obj(L, clustering, cluster.max = k)
  ratio.cut.prev <- ratio.cut + 1

  # iteration counter
  iter <- 0

  while (ratio.cut.prev > ratio.cut) {
```

```
    # update kmeans criterion
    ratio.cut.prev <- ratio.cut

    # break if too many iterations
    if (iter > max.iter) {
      warning('failed to converge')
      break
    }

    # shuffle the indices if specified
    if (randomize) {
      ind <- sample(ind)
    }

    # cycle through the objects
    for (i in ind) {
      # figure out the cluster assignment for this index
      cur.clust.assign <- clustering[i]
      clust.assign.to.try <- seq(k)[-cur.clust.assign]
      ratio.cut.try <- sapply(clust.assign.to.try, function(j) {
        temp.clustering <- clustering
        temp.clustering[i] <- j
        ratio.cut.obj(L, temp.clustering, cluster.max = k)
      })
      if (min(ratio.cut.try) <= ratio.cut) {
        ratio.cut <- min(ratio.cut.try)
        clustering[i] <- clust.assign.to.try[which.min(ratio.cut.try)]
      }
    }

    # update iteration counter
    iter <- iter + 1
    if (verbose) print(ratio.cut)
  }

  return(list(ratio.cut = ratio.cut,
              clustering = clustering,
              n.iter = iter))
}

# compute laplacian
L <- graph.laplacian(W)

# compute MP inverse
L.dagger <- MASS::ginv(L)

# try out kernel k-means
kkm.exchange <- kernel.kmeans.exchange(L.dagger, k = 2)
qgraph(W, groups = factor(kkm.exchange$clustering),
       node.width = 2,
       title = 'kernel k-means clustering via exchange method')
```

kernel k−means clustering via exchange method



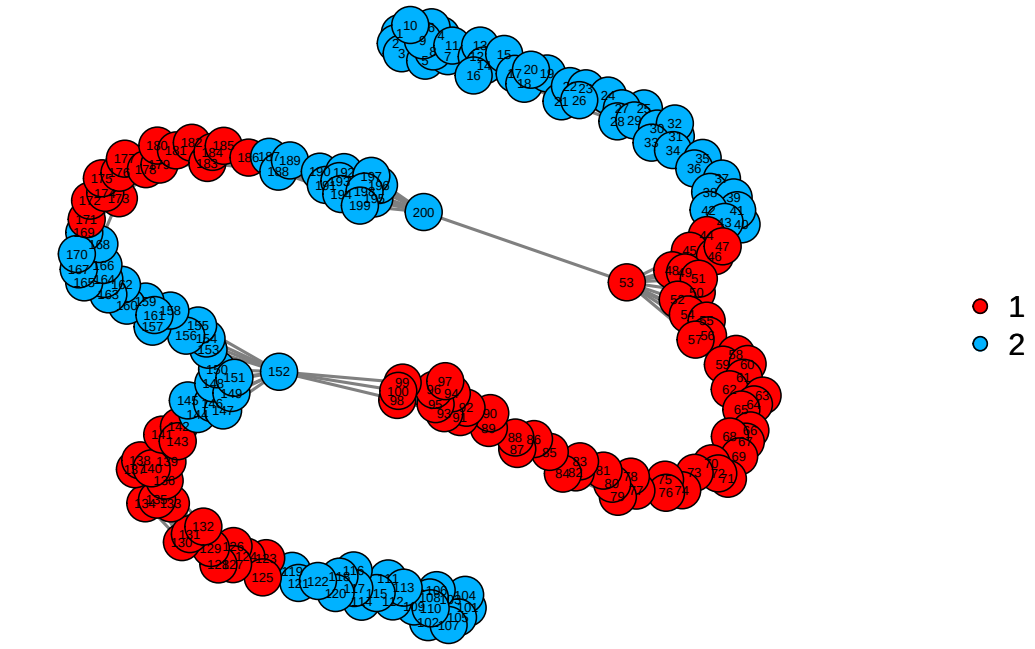

```
# final objective value
kkm.exchange$k.means
```

```
[1] 28.73899
```

```
# best objective value
kernel.kmeans.obj(L.dagger, c(rep(1, 100), rep(2, 100)))
```

```
[1] 28.73899
```

```
# try out ratio cut
rc.exchange <- ratio.cut.exchange(L, k = 2)
qgraph(W, groups = factor(rc.exchange$clustering),
       node.width = 2,
       title = 'ratio cut clustering via exchange method')
```




```
# final objective value
rc.exchange$ratio.cut
```

```
[1] 1.585709
```

```
# best objective value
ratio.cut.obj(L, c(rep(1, 100), rep(2, 100)))
```

```
[1] 0.08
```

Here, we can see that while kernel k-means found the global optimum, ratio cut did not. We saw previously that ratio cut tends to have many more local optima than kernel k-means, so this is somewhat expected. It should also be noted that kernel k-means also has local optima, and so sometimes this method fails.

Finally, we should note that exchange methods are very slow. With $n = 200$ and $k = 2$, it did not take long, but the complexity of the problem grows very quickly as we add more vertices or more clusters.

Lloyd's algorithm

Due to the inefficiency of exchange methods, it is common to use Lloyd's algorithm to solve Euclidean k-means. However, Lloyd's algorithm requires the computation of cluster centroids, which don't exist in our problem. Instead, we can try the next best thing: approximating one of the vertices as a "centroid". The center vertex of a cluster would be one that has the highest degree within that cluster.

Input K kernel matrix, k number of clusters, $(C_1^0, ..., C_k^0)$ initial clustering

Output $(C_1, ..., C_k)$ final clustering

Algorithm Until convergence:

1. For each cluster C_j:
    i. Compute the row-sums of K_j, the sub-matrix of K pertaining to the j^{th} cluster.

ii. Take the argmax and assign it as the most central point.

2. For each vertex v_i:

   i. Compute K_{ij} for each cluster C_j.
   ii. Assign v_i to C_j^* that maximizes K_{ij}

The algorithm is the same for ratio cut, except instead of finding maxima, we look for minima.

```
kernel.kmeans.lloyd <- function(K, k = 2,
                                initial.clust = NULL,
                                max.iter = 1e2,
                                verbose = FALSE) {
  # number of indices
  n <- nrow(L)

  # if no initial clustering is specified, create one
  # else make sure the initial clustering is valid
  if (is.null(initial.clust)) {
    clustering <- c(seq(k), sample(seq(k), n - k, replace = TRUE))
  } else {
    assertthat::assert_that(max(initial.clust) <= k)
    assertthat::assert_that(all(sort(unique(initial.clust)) == seq(k)))
    clustering <- initial.clust
  }

  # randomly initialize the central points
  centers <- sapply(seq(k), function(i) clustering[clustering == i][1])

  # randomly initialize the nonexistent "previous" iteration
  clustering.prev <- sample(clustering)
  centers.prev <- seq(k + 1, k + k)

  # iteration counter in case we need to exit early
  iter <- 0

  # initialize kmeans objective value
  kmeans.obj <- kernel.kmeans.obj(K, clustering, cluster.max = k)
  kmeans.obj.prev <- 0

  # while the previous iteration is not the same as the current iteration
  while (!(all(centers == centers.prev)) &
         !(all(clustering == clustering.prev))) {
    # update iteration counter
    iter <- iter + 1
    # and check to see if we've been trying too long
    if (iter > max.iter) {
      warning('failed to converge')
      break
    }

    # update previous iteration
    clustering.prev <- clustering
    centers.prev <- centers

    # update centers
```

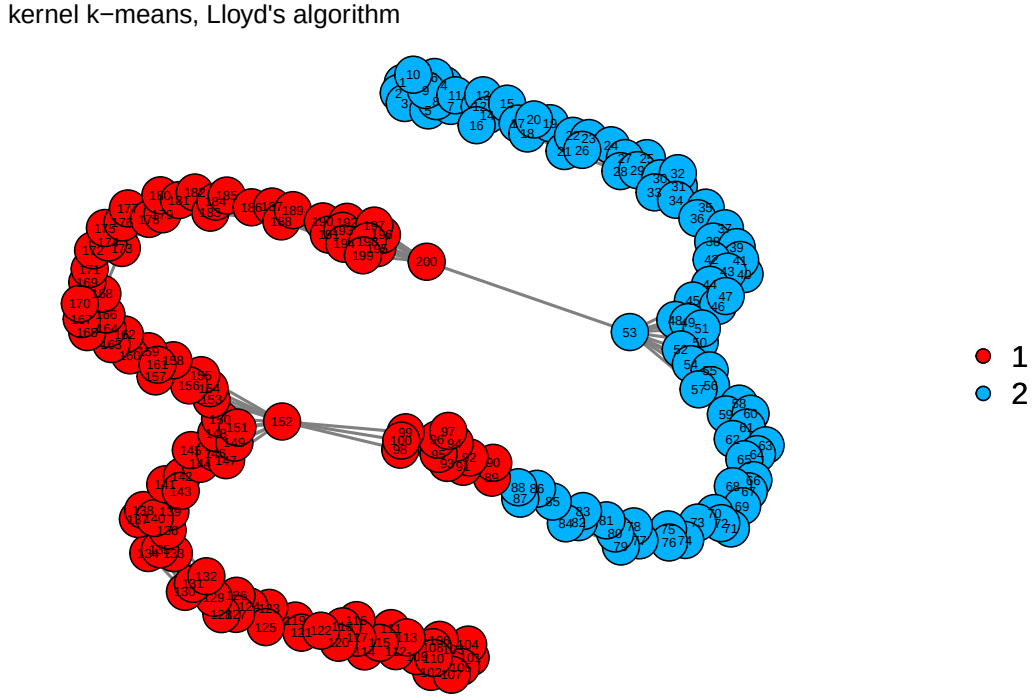
```
    centers <- sapply(seq(k), function(i) {
      sub.K <- K
      sub.K[clustering != i, ] <- 0
      sub.K[, clustering != i] <- 0
      sub.D <- D <- sapply(seq(nrow(sub.K)), function(j) {
        sum(sub.K[j, ])
      })
      which.max(sub.D)
    })

    # update clusters
    clustering <- sapply(seq_along(clustering), function(i) {
      sapply(centers, function(j) {
        K[i, j]
      }) %>%
        which.max()
    })

    # update kernel kmeans objective
    kmeans.obj.prev <- kmeans.obj
    kmeans.obj <- kernel.kmeans.obj(K, clustering, cluster.max = k)
    # report kernel kmeans objective
    if (verbose) print(kmeans.obj)
    # if kmeans objective decreased, exit
    if (kmeans.obj.prev > kmeans.obj) {
      kmeans.obj <- kmeans.obj.prev
      clustering <- clustering.prev
      centers <- centers.prev
      break
    }
  }

  return(list(
    centers = centers,
    clustering = clustering,
    kmeans.obj = kmeans.obj
  ))
}

ratio.cut.lloyd <- function(L, k = 2,
                            initial.clust = NULL,
                            max.iter = 1e2,
                            verbose = FALSE) {
  # number of indices
  n <- nrow(L)

  # if no initial clustering is specified, create one
  # else make sure the initial clustering is valid
  if (is.null(initial.clust)) {
    clustering <- c(seq(k), sample(seq(k), n - k, replace = TRUE))
  } else {
    assertthat::assert_that(max(initial.clust) <= k)
    assertthat::assert_that(all(sort(unique(initial.clust)) == seq(k)))
```

```
  clustering <- initial.clust
}

# randomly initialize the central points
centers <- sapply(seq(k), function(i) clustering[clustering == i][1])

# randomly initialize the nonexistent "previous" iteration
clustering.prev <- sample(clustering)
centers.prev <- seq(k + 1, k + k)

# iteration counter in case we need to exit early
iter <- 0

# initialize kmeans objective value
ratio.cut <- ratio.cut.obj(L, clustering)
ratio.cut.prev <- 0

# while the previous iteration is not the same as the current iteration
while (!(all(centers == centers.prev)) &
       !(all(clustering == clustering.prev))) {
  # update iteration counter
  iter <- iter + 1
  # and check to see if we've been trying too long
  if (iter > max.iter) {
    warning('failed to converge')
    break
  }

  # update previous iteration
  clustering.prev <- clustering
  centers.prev <- centers

  # update centers
  centers <- sapply(seq(k), function(i) {
    sub.L <- L
    sub.L[clustering != i, ] <- 1e6
    sub.L[, clustering != i] <- 1e6
    sub.D <- D <- sapply(seq(nrow(sub.L)), function(j) {
      sum(sub.L[j, ])
    })
    which.min(sub.D)
  })

  # update clusters
  clustering <- sapply(seq_along(clustering), function(i) {
    sapply(centers, function(j) {
      L[i, j]
    }) %>%
      which.min()
  })

  # update ratio cut objective
  ratio.cut.prev <- ratio.cut
```

```
    ratio.cut <- ratio.cut.obj(L, clustering)
    # report ratio cut objective
    if (verbose) print(ratio.cut)
    # if ratio cut objective decreased, exit
    if (ratio.cut.prev < ratio.cut) {
      ratio.cut <- ratio.cut.prev
      clustering <- clustering.prev
      centers <- centers.prev
      break
    }
  }

  return(list(
    centers = centers,
    clustering = clustering,
    ratio.cut = ratio.cut
  ))
}
```

Since we are approximating the cluster centroids, we are not guaranteed to improve on the objective function at each iteration. So there is an additional step where we break out of the loop if we notice that the objective function decreases (or incrases in the case of ratio cut).

```
# try out kernel k-means
kkm.lloyd <- kernel.kmeans.lloyd(L.dagger, k = 2)
qgraph(W, groups = factor(kkm.lloyd$clustering), node.width = 2,
       title = 'kernel k-means, Lloyd\'s algorithm')
```



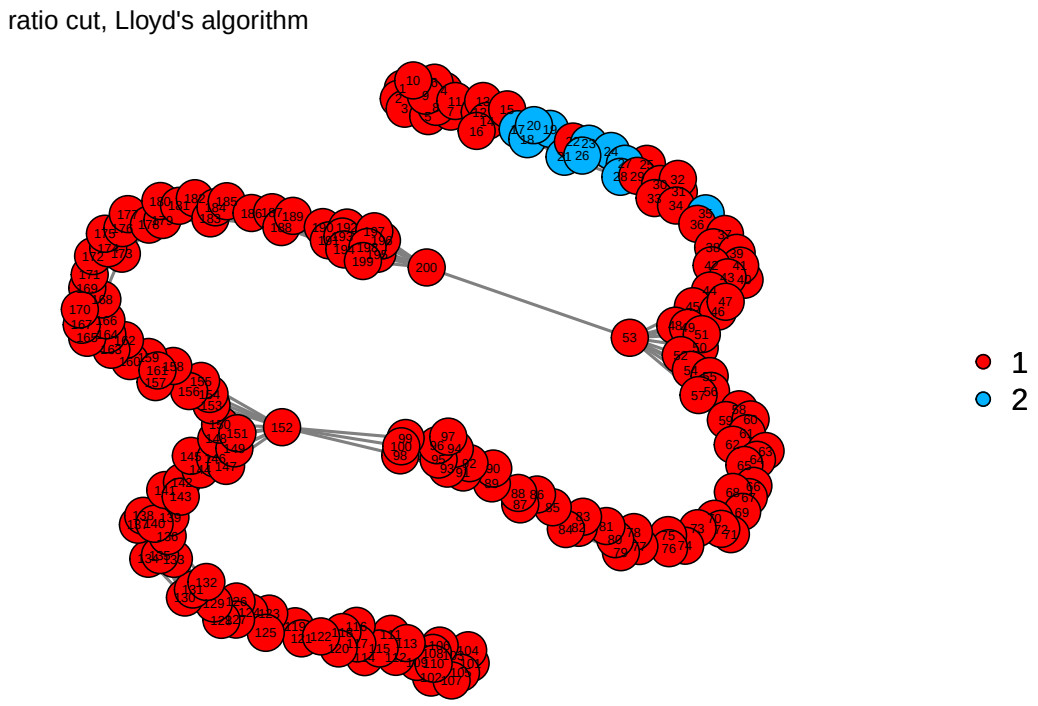
```
# final objective value
kkm.lloyd$kmeans.obj
```

```
[1] 28.63588
```

```
# best objective value
kernel.kmeans.obj(L.dagger, c(rep(1, 100), rep(2, 100)))
```

```
[1] 28.73899
```

```
# try out ratio cut
rc.lloyd <- ratio.cut.lloyd(L, k = 2)
qgraph(W, groups = factor(rc.lloyd$clustering), node.width = 2,
       title = 'ratio cut, Lloyd\'s algorithm')
```




```
# final objective value
rc.lloyd$ratio.cut
```

```
[1] 5.772006
```

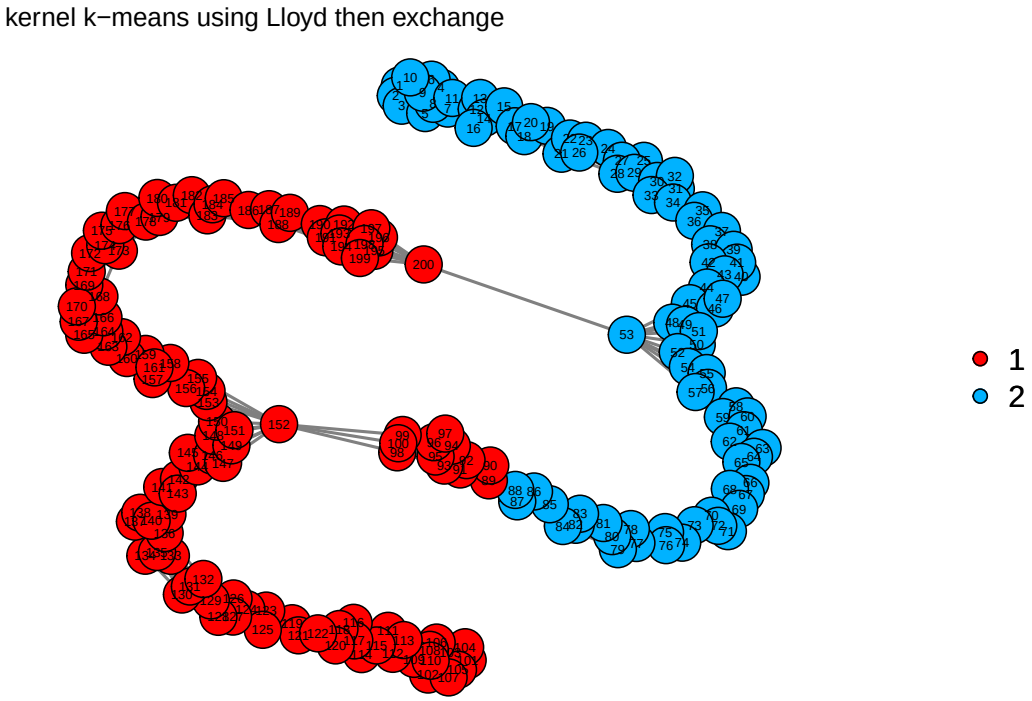
```
# best objective value
ratio.cut.obj(L, c(rep(1, 100), rep(2, 100)))
```

```
[1] 0.08
```

We can see that neither found the global optimum, although k-means came close. We can try taking the output of k-means via Lloyd and putting that into k-means via exchange. The motivation here is that if Lloyd's algorithm doesn't get us to the global solution, perhaps it gets us close enough and the slower exchange method can finish things up.

```
kkm.lloyd.exchange <-
  kernel.kmeans.exchange(L.dagger, initial.clust = kkm.lloyd$clustering)
```

```
qgraph(W, groups = factor(kkm.lloyd.exchange$clustering), node.width = 2,
       title = 'kernel k-means using Lloyd then exchange')
```




```
# final objective value
kkm.lloyd.exchange$k.means
```

```
[1] 28.63588
```

Unfortunately that did not happen. It appears that Lloyd's algorithm found a local optimum. Successive trials show that it almost always converges to this local solution.