GEM Assignment - Source Code

Steffie van Poppel, Mike Weltevrede, Joost Westland (Group 7)

9/21/2019

Question a

```
#### Initialise document ####
#'Assignment Games and Economic Models - Question A
#'@author Steffie van Poppel, Mike Weltevrede, Joost Westland (Group 7)

# Clean environment
rm(list = ls())

# Activating packages
library(readxl)
library(bazar)
```

```
#### Defining functions ####

#'Import and process data
#'
#'\code{import_data} reads data and does some basic cleaning and computations.
#'
#'@param \code{file_location} The path to the data to read in.
#'
#'@return A list \code{f} containing the priority ordering (which is computed as
#'  the ascending order of patient numbers), the numeric encoding \code{w}
#'  for the waiting list  (which is computed as the maximum patient number plus
#'  1), the matrix with \code{preferences}, a list \code{current_assignment}
#'  with currently favourite kidneys, an empty list \code{final_assignment}
#'  with patient numbers as names to store the final assignment, an empty vector
#'  \code{assigned} to store the patients that are assigned, and an empty vector
#'  \code{available_kidneys} to store the kidneys that become available due to
#'  assigning w-chains
import_data <- function(file_location) {

  data <- readxl::read_excel(file_location, .name_repair = "minimal")

  colnames(data)[1] <- "Patient"

  # Define the priority ordering f (ascending order of patient number)
  f <- sort(data$Patient)
  number_of_patients <- max(f)

  # Define which patient ID codes for the waiting list (maximum patient ID + 1)
  w <- number_of_patients + 1

  # Define patient names as the character variant of their patient number
  colnames(data) <- append("Patient", as.character(1:w))
  patient_names <- as.character(data$Patient)
```

```
  # Define preference profiles; i.e. all data except for patient ID
  preferences <- data[, -1]

  # Initialise kidneys and living donors

  # Make patients point to kidneys
  current_assignment <- t(preferences[, 1]) # Initialise to first choice
  names(current_assignment) <- patient_names
  current_assignment <- as.list(current_assignment)

  # Initialise the final_assignment to 0
  final_assignment <- t(as.matrix(rep(0, number_of_patients)))
  names(final_assignment) <- patient_names
  final_assignment <- as.list(final_assignment)

  return(list("f" = f,
              "w" = w,
              "preferences" = preferences,
              "current_assignment" = current_assignment,
              "final_assignment" = final_assignment,
              "assigned" = c(),
              "available_kidneys" = c()))
}
```

```
#'Search for cycles
#'
#'\code{circle_finder} finds circles in the current preferences of patients,
#'assigns these, and updates the variables accordingly. Notice that we use the
#'term circle instead of cycle, since \code{cycle()} is a function in R.
#'
#'@param \code{data} List with structure like the output of \code{import_data}.
#'
#'@return A list \code{f} containing the priority ordering, the numeric encoding
#'  \code{w} for the waiting list, the matrix with \code{preferences}, a list
#'  \code{current_assignment} with currently favourite kidneys, a list
#'  \code{final_assignment} the currently final assignment, a vector
#'  \code{assigned} with patients that are already assigned, and a vector
#'  \code{available_kidneys} with kidneys that become available due to
#'  assigning w-chains.
circle_finder <- function(data){

  # Unpack the list
  f <- data$f
  w <- data$w
  preferences <- data$preferences
  current_assignment <- data$current_assignment
  final_assignment <- data$final_assignment
  assigned <- data$assigned
  available_kidneys <- data$available_kidneys

  rm(data) # Clean up; `data` is not needed in the rest of this function

  circle_found <- TRUE # Initialise
```

```
# As long as cycles exist in the current assignment...
while (circle_found) {

  # Initialise
  new_assigned <- c()
  no_circle_found_so_far <- c(w)

  # For every patient in the order of priority list...
  for (i in 1:length(f)) {

    if (!f[i] %in% no_circle_found_so_far) {
      # Then patient is not yet assigned

      current_chain <- c() # Create an empty chain
      j <- f[i] # Becomes the current step

      # As long as j is not already assigned to something...
      while (!j %in% no_circle_found_so_far && !j %in% assigned) {

        current_chain <- append(current_chain, j)
        j <- current_assignment[[as.character(j)]]

        if (j == w || j %in% no_circle_found_so_far) {
          # If j is in a w-chain or already assigned, stop

          # Assign to no_circle_found_so_far, when a new patient point to
          # something that's already checked, stop
          no_circle_found_so_far <- append(no_circle_found_so_far,
                                           current_chain)
        }

        if (j %in% current_chain) {
          # When j is in the current chain then we have found a cycle
          circle <- current_chain[
            which(current_chain == j):length(current_chain)]

          for (k in 1:length(circle)) {
            # Update the final result
            final_assignment[[as.character(circle[k])]] <-
              current_assignment[[as.character(circle[k])]]
          }

          # Append new_assigned, we need to update this in current_assignment
          new_assigned <- append(new_assigned, circle)

          # When a patient points to the cycle, we don't need to look further
          # for this chain.
          no_circle_found_so_far <- append(no_circle_found_so_far,
                                           current_chain)
        }
      }
    }
  }
```

```
    # If circles exist, assign all circles
    if (!is.null(new_assigned)) {
      # Check if we have assigned anything
      assigned <- append(assigned, new_assigned)
      selection <- c()

      for (k in 1:length(new_assigned)) {
        # Select patients that are assigned in this loop
        selection <- append(selection,
                            which(as.numeric(
                              names(current_assignment)) == new_assigned[k]))
      }

      # Select all patients not assigned in this loop (with the minus sign)
      selection <- -selection

      # Drop all assigned patients from the graph
      current_assignment <- current_assignment[selection]
      f <- f[selection]

      # Check if their are any patients left
      if (!bazar::is.empty(current_assignment)) {

        # Reassign arrows and recheck circles
        for (k in 1:length(current_assignment)) {
          # Only look to the remaining patients
          index.X <- as.numeric(names(current_assignment)[k])

          # Not allowed to point to the assigned patients, except the kidneys
          # that remain after the w chain
          index.Y <- !preferences[index.X, ] %in%
            assigned[which(!assigned %in% available_kidneys)]

          current_assignment[k] <- preferences[index.X, index.Y][1]
        }
      } else {
        # If no patients are left, exit the loop
        circle_found <- FALSE
      }
    } else {
      # If no cycles are found, exit the loop
      circle_found <- FALSE
    }
  }
  # Return updated data points
  return(list("f" = f,
              "w" = w,
              "preferences" = preferences,
              "current_assignment" = current_assignment,
              "final_assignment" = final_assignment,
              "assigned" = assigned,
              "available_kidneys" = available_kidneys))
}
```

```r
#'Find w-chains
#'
#'\code{w_finder} finds w-chains according to chain selection rule (e) in the
#'current preferences of patients, assigns these, and updates the variables
#'accordingly.
#'
#'@param \code{data} List with structure like the output of \code{import_data}.
#'
#'@return A list \code{f} containing the priority ordering, the numeric encoding
#'  \code{w} for the waiting list, the matrix with \code{preferences}, a list
#'  \code{current_assignment} with currently favourite kidneys, a list
#'  \code{final_assignment} the currently final assignment, a vector
#'  \code{assigned} with patients that are already assigned, and a vector
#'  \code{available_kidneys} with kidneys that become available due to
#'  assigning w-chains.
w_finder <- function(data){

  # Unpack the list
  f <- data$f
  w <- data$w
  preferences <- data$preferences
  current_assignment <- data$current_assignment
  final_assignment <- data$final_assignment
  assigned <- data$assigned
  available_kidneys <- data$available_kidneys

  rm(data) # Clean up; `data` is not needed in the rest of this function

  # Initialize temporary data
  w_chain <- c()
  new_assigned <- c()
  already_checked <- c(w)
  first_of_w_chain <- w

  # Search the w_chain for all patients in the order of priority list
  for (i in 1:length(f)) {

    # Check if we have not yet found the current patient
    if (!f[i] %in% already_checked) {
      current_chain <- c()

      # Similar as in circle_finder, j is the current step in the chain
      j <- f[i]

      # As long as j is available...
      while (!j %in% already_checked && !j %in% assigned) {
        current_chain <- append(current_chain, j)
        j <- current_assignment[[as.character(j)]]

        # Check if j is still avalable, otherwise update already checked and
        # stop the loop
        if (j == w || j %in% already_checked) {
          already_checked <- append(already_checked, current_chain)
```

```
      }

      # Check if j is a remaining kidney, patient is already assigned, but
      # kidney not yet (first entry of w-chain with highest priority)
      if (j %in% available_kidneys) {

        index <- which(available_kidneys == j)
        available_kidneys[index] <- current_chain[1]

        # Update final_assignment
        for (k in 1:length(current_chain)) {
          final_assignment[[as.character(current_chain[k])]] <-
            current_assignment[[as.character(current_chain[k])]]
        }

        # Update the new_assigned entries
        new_assigned <- append(new_assigned, current_chain)
        already_checked <- append(already_checked, current_chain)
      }

      # When j is equal to the entry of the current w_chain with the highest
      # priority, then we expand the w-chain
      if (j == first_of_w_chain) {

        w_chain <- append(current_chain, w_chain)
        if (w != first_of_w_chain) {
        already_checked <- append(already_checked, current_chain)
        }
        first_of_w_chain <- w_chain[1]
      }
    }
  }
}

# Check if we have found a w-chain
if (!is.null(w_chain)) {

  # Update final_assignment
  for (k in 1:length(w_chain)) {
    final_assignment[[as.character(w_chain[k])]] <-
      current_assignment[[as.character(w_chain[k])]]
  }

  # If w is not equal to the first_of_w_chain...
  if (w != first_of_w_chain) {
    available_kidneys <- append(available_kidneys, first_of_w_chain)
  }
}

new_assigned <- append(new_assigned, w_chain)

# Check if the new_assigned are not empty
if (!is.null(new_assigned)) {
```

```
    assigned <- append(assigned, new_assigned)
    selection <- c()

    # Throw out all already assigned patients
    for (k in 1:length(new_assigned)) {
      selection <- append(selection, which(as.numeric(
        names(current_assignment)) == new_assigned[k]))
    }

    selection <- -selection
    current_assignment <- current_assignment[selection]
    f <- f[selection]
  }

  # Update preferences, patients can only point to the first kidney of the
  # w-chain(s), not to the other patients
  for (k in 1:length(current_assignment)) {

    index.X <- as.numeric(names(current_assignment)[k])
    index.Y <- !preferences[index.X, ] %in%
      assigned[which(!assigned %in% available_kidneys)]
    current_assignment[k] <- preferences[index.X, index.Y][1]
  }

  # Return updated data
  return(list("f" = f,
              "w" = w,
              "preferences" = preferences,
              "current_assignment" = current_assignment,
              "final_assignment" = final_assignment,
              "assigned" = assigned,
              "available_kidneys" = available_kidneys))
}
```

```
#'Runs the TTCC algorithm with chain selection rule (e)
#'
#'\code{exercise_a} executes the TTCC algorithm with chain selection rule (e) by
#'repeating the process of finding cycles, followed by finding w-chains until
#'all patients are assigned
#'
#'@param \code{file_location} The path to the data to read in.
#'
#'@return A \code{data.frame} \code{final_assignment} with the final assignment
#'for all patients and a list \code{remaining_kidneys} with the kidneys that are
#'not assigned to a patient (as a result of their respective patient being the
#'first patient in an assigned w-chain).
exercise_a <- function(file_location){
  # Import data, using the function that is written above
  iterate_data <- import_data(file_location = file_location)

  # All patients that are currently not yet assigned
  f <- iterate_data$f

  # As long as patients remain unassigned
```

```
  while (!bazar::is.empty(f)) {

    # Search for cycles...
    iterate_data <- circle_finder(iterate_data)

    # And update f accordingly
    f <- iterate_data$f

    # Check if patients are unassigned
    if (!bazar::is.empty(f)) {
      # Then there is a w-chain left, since all cycles were removed by
      # circle_finder
      iterate_data <- w_finder(iterate_data)
      f <- iterate_data$f #update f
    }
  }

  # Update the final output for the waiting list
  iterate_data$final_assignment[
    which(iterate_data$final_assignment == iterate_data$w)] <- "w"

  # Create tidy table
  df.final <- data.frame(iterate_data$final_assignment)
  colnames(df.final) <- names(iterate_data$final_assignment)

  # Return final result
  return(list("final_assignment" = df.final,
              "remaining_kidneys" = iterate_data$available_kidneys))
}
```

```
#### Run the algorithm ####
result <- exercise_a(file_location = "data/dataset7.xlsx")
```

Question b

```
#### Initialise document ####
#'Assignment Games and Economic Models - Question B
#'@author Steffie van Poppel, Mike Weltevrede, Joost Westland (Group 7)

# Clean environment
rm(list = ls())

# Activating packages
library(readxl)
```

```
#### Defining functions ####

#'Import and process data
#'
#'\code{import_data} reads data and does some basic cleaning and computations.
#'
#'@param \code{file_location} The path to the data to read in.
#'
#'@return The matrix with \code{preferences}, the list of \code{patient_names},
```

```r
#'  a list \code{f} containing the priority ordering (which is computed as
#'  the ascending order of patient numbers), and the numeric encoding \code{w}
#'  for the waiting list (which is computed as the maximum patient number plus
#'  1).
import_data = function(file_location) {
  data = readxl::read_excel(file_location, .name_repair = "minimal")
  colnames(data)[1] = "Patient"

  # Define the priority ordering f
  f = sort(as.numeric(data$Patient))

  # Define which patient ID codes for the waiting list (maximum patient ID + 1)
  number_of_patients = max(f)
  w = number_of_patients + 1

  # Clean up column names
  patient_names = as.character(data$Patient)
  colnames(data) = append("Patient", append(patient_names, w))

  # Define preference profiles; i.e. all data except for patient ID
  preferences = data[, -1]

  return(list("preferences" = preferences,
              "patient_names" = patient_names,
              "f" = f,
              "w" = w))
}
```

```r
#'Retains the unique entries in a named list
#'
#'\code{uniquefy_list}, given a named list, retains the unique entries. This is
#'different from the \code{unique()} function since that function only considers
#'(unnamed) values of a list when seaching for unique entries.
#'
#'@param \code{lst} The list to "uniquefy".
#'
#'@return The "uniquefied" list \code{list_unique}.
uniquefy_list = function(lst) {

  # Initialise
  list_unique = c()

  # Loop over the input list
  for (i in 1:length(lst)) {

    # Check if this entry is already in the list and, if not, append it
    if (!lst[i] %in% list_unique) {

      list_unique = c(list_unique, lst[i])
    }
  }

  return(list_unique)
}
```

```r
#'Search for cycles
#'
#'\code{circle_chain_finder} finds circles and w-chains in the current
#'preferences of patients. Notice that we use the term circle instead of cycle,
#'since \code{cycle()} is a function in R.
#'
#'@param \code{current_assignment} List with the currently favourite preferences
#'@param \code{available_kidneys} List with available kidneys
#'@param \code{w} The numeric encoding for the waiting list
#'
#'@return A list of cycles in \code{circles} and w-chains in \code{chains}.
circle_chain_finder = function(current_assignment, available_kidneys, w){

  # Initialise sets of all circles and chains
  circles = list()
  chains = list()

  # Initialise an empty cycle and w-chain
  circle = c()
  w_chain = c()

  # Initialise patients that are already checked (so that we avoid finding a
  # cycle or chain that we have already found before)
  already_checked = c()

  for (i in 1:length(current_assignment)) {

    # Initialise boolean to avoid assignment later in this function
    already_found = FALSE

    if (names(current_assignment)[i] %in% already_checked) {
      # This patient belongs to a cycle or chain that we already explored

      already_checked = append(already_checked, names(current_assignment)[i])
      next
    }

    # Start a chain
    current_chain = c(current_assignment[i])

    # Find the preferred kidney by this patient
    points_to = as.character(current_assignment[i])

    if (points_to == names(current_assignment)[i]) {
      # This patient points to themselves, assign this

      circles = c(circles, list(current_chain))
      already_checked = append(already_checked, unique(names(circle)))
      next
    }

    if (points_to == w) {
      # w-chain found
```

```
    chains = c(chains, list(current_chain))
    already_checked = append(already_checked, unique(names(current_chain)))
    next
  }

  # Set the next patient to the one corresponding to the preferred kidney...
  next_patient = current_assignment[points_to]

  # And make the list unique, just to be sure (cheap operation)
  current_chain = uniquefy_list(append(current_chain, next_patient))

  # Continue this loop "indefinitely" (if-statements with break calls make
  # sure that the loop gets broken)
  while (TRUE) {

    if (next_patient %in% names(unlist(circles))) {
      # This will lead to a circle already found before: skip

      checked = append(current_chain, next_patient)
      already_checked = append(already_checked, unique(names(checked)))
      already_found = TRUE
      break

    } else if (next_patient %in% names(unlist(chains))) {
      # This will extend an existing chain or will be a new branch.

      for (i in 1:length(chains)) {
        chain = chains[[i]]

        if (next_patient %in% names(chain)) {
          selected_chain = chain
          which_chain = i
          break
        }
      }

      ind = which(next_patient == names(selected_chain))
      new_chain = uniquefy_list(append(next_patient,
                                       selected_chain[
                                         ind:length(selected_chain)]))

      if (ind == 1) {
        # This extends an existing chain. Since we want to keep the longest
        # chain (containing the highest-priority patient), we delete the
        # original chain and add this one instead

        chains[[which_chain]] = new_chain
      } else {
        chains = append(chains, list(new_chain))
      }

      checked = append(current_chain, next_patient)
      already_checked = append(already_checked, unique(names(checked)))
```

```
      already_found = TRUE
      break

    } else if (next_patient %in% names(current_chain)) {
      # Circle found

      # Check where it is linked to, don't keep the patients before this
      # (we are only interested in cycles, and not loose chains appended to
      # a cycle - we know that these will not form a cycle anyway)
      starting_index = which(next_patient == names(current_chain))

      circle = append(current_chain[starting_index:length(current_chain)],
        next_patient)

      already_checked = append(already_checked, unique(names(circle)))
      break
    } else if (next_patient == w) {
      # w-chain found
      w_chain = append(current_chain, next_patient)

      already_checked = append(already_checked, unique(names(w_chain)))
      break
    } else if (next_patient %in% available_kidneys) {
      # This is a path that can safely be assigned, treat this as a w-chain
      w_chain = append(current_chain, next_patient)

      already_checked = append(already_checked, unique(names(w_chain)))
      break
    } else {
      # No circle or w-chain is found with this iteration, so continue by
      # extending the chain with the patient that is being pointed to.
      current_chain = uniquefy_list(append(current_chain, next_patient))

      # Find patient that this next patient prefers
      next_patient = current_assignment[as.character(next_patient)]
    }
  }

  if (already_found) {
    # Don't append this circle or chain!
    next
  }

  if (length(circle) > 0) {
    # A cycle was found, append this to the list of cycles
    circles = c(circles, list(circle))
  }

  if (length(w_chain) > 0) {
    # A w-chain was found, append this to the list of w-chains
    chains = c(chains, list(w_chain))
  }
}
```

```
  # Return the lists of cycles and chains
  return(list("circles" = circles,
              "chains" = chains))
}
```

```
#'Assigns cycles
#'
#'\code{circle_assigner} assigns cycles from \code{circles} by placing the
#'corresponding assignments in \code{final_assignment}
#'
#'@param \code{final_assignment} List with currently final assignment
#'@param \code{circles} List of circles
#'
#'@return Updated \code{final_assignment} list
circle_assigner = function(final_assignment = list(), circles) {

  for (circle in circles) {
    final_assignment = append(final_assignment, circle)
  }

  return(final_assignment)
}
```

```
#'Assigns chains using chain selection rule (e)
#'
#'\code{chain_assigner} assigns chains from \code{chains} by placing the
#'corresponding assignments in \code{final_assignment}. It takes into account
#'chain selection rule (e), which states that the chain containing the highest
#'priority patient should be assigned. When that patient appears in multiple
#'chains, we take the longest chain, breaking the tie with the first one found.
#'
#'@param \code{final_assignment} List with currently final assignment
#'@param \code{chains} List of chains
#'@param \code{f} List with the priority ordering
#'@param \code{available_kidneys} List with available kidneys
#'@param \code{assigned_patients} List with already assigned patients
#'
#'@return A list with updated \code{final_assignment}, \code{available_kidneys},
#'  and \code{assigned_patients}.
chain_assigner = function(final_assignment = list(), chains, f,
                          available_kidneys, assigned_patients) {

  in_chain = names(unlist(chains))

  # Find the person highest on the priority list
  index = which.min(match(in_chain, f))
  highest_priority = in_chain[index]

  # Find chains which have highest_priority in them
  candidate_chains = which(sapply(chains,
                                  function(x) {highest_priority %in% names(x)}))

  # Find which of these chains is the longest. If there is a tie, we take the
  # first one found
```

```
  longest_candidates = which.max(sapply(chains[candidate_chains], length))
  selected_chain = chains[longest_candidates]

  # Add first kidney in chain to available kidneys
  available_kidneys = c(available_kidneys, names(selected_chain)[1])

  # Update final_assignment and assigned_patients accordingly
  final_assignment = append(final_assignment, selected_chain)
  assigned_patients = c(assigned_patients, names(unlist(selected_chain)))

  return(list("final_assignment" = final_assignment,
              "available_kidneys" = available_kidneys,
              "assigned_patients" = assigned_patients))
}
```

```
#'Updates preferences
#'
#'\code{preference_updater} updates preferences by not allowing patients to have
#'an already assigned patient as their current preference. If \code{HPBM} is
#'\code{TRUE}, then only the preference for the \code{hpbm_patient} should be
#'updated.
#'
#'@param \code{preferences} Matrix with all preferences
#'@param \code{assigned_patients} List with already assigned patients
#'@param \code{patient_names} List with patient names (for generality)
#'@param \code{available_kidneys} List with available kidneys
#'@param \code{hpbm} Boolean, will the function be run in the HPBM algorithm?
#'@param \code{hpbm_patient} String specifying which patient the HPBM is
#'  currently considering
#'
#'@return A list with updated \code{preferences} and corresponding
#'  \code{current_assignment}.
preference_updater = function(preferences, assigned_patients, patient_names,
                              available_kidneys, hpbm = FALSE,
                              hpbm_patient=c()) {

  if (hpbm) {

    current_assignment = t(preferences[, 1])
    names(current_assignment) = patient_names
    current_assignment = as.list(current_assignment)
    current_assignment[[as.character(hpbm_patient)]] =
      as.numeric(preferences[hpbm_patient, which(
        !preferences[hpbm_patient, ] %in% assigned_patients)][1])
  } else {
    for (i in 1:dim(preferences)[1]) {
      if (i %in% assigned_patients) {
        # Don't update preferences for assigned patients
        next
      }

      # Start with their 2nd highest preference
      j = 2
      while (preferences[i, 1] %in% assigned_patients) {
```

```
        # Patient cannot have their next best preference be an assigned
        # patient...

        if (preferences[i, 1] %in% available_kidneys) {
          # Unless that kidney is available!
          break
        } else {
          preferences[i, 1] = preferences[i, j]
          j = j + 1
        }
      }
    }

    # Initialise to current first choice
    current_assignment = t(preferences[, 1])
    names(current_assignment) = patient_names
    current_assignment = as.list(current_assignment)

    # Only take the current preference of non-assigned patients
    current_assignment = current_assignment[
      !names(current_assignment) %in% assigned_patients]

  }

  return(list("preferences" = preferences,
              "current_assignment" = current_assignment))
}
```

```
#'Runs the HPBM algorithm
#'
#'\code{hpbm} runs the Highest-Priority Breaking Method (HPBM) for TTCC. In
#'light of practical reasons, it can be so that cycles cannot be larger than
#'some capacity. For example, there are not enough surgeons available. To
#'circumvent this problem, we propose this method. In short, when all found
#'cycles are too long, the HPBM explores the preferences of the patient with the
#'highest priority in those cycles until a proper match is found. For more
#'details, please read our report.
#'
#'@param \code{circles} List of circles, all exceeding some capacity q
#'@param \code{preferences} Matrix with all preferences
#'@param \code{assigned_patients} List with already assigned patients
#'@param \code{patient_names} List with patient names (for generality)
#'@param \code{current_assignment} List with the currently favourite preferences
#'@param \code{final_assignment} List with currently final assignment
#'@param \code{f} List with the priority ordering
#'@param \code{w} The numeric encoding for the waiting list
#'@param \code{available_kidneys} List with available kidneys
#'@param \code{q} The capacity constraint for length of a cycle
#'
#'@return A list with updated \code{final_assignment},\code{available_kidneys},
#'  \code{assigned_patients}, \code{preferences}, and \code{current_assignment}.
hpbm = function(circles, preferences, assigned_patients, patient_names,
                current_assignment, final_assignment, f, w, available_kidneys,
                q) {
```

```
# Save preferences in a different variable to retrieve original preferences
# later
preferences_original = preferences

# Initialise list of all patients in these cycles
candidates = unique(names(unlist(circles)))

# Find highest-priority patient in cycles and their currently favourite kidney
t = f[min(match(candidates, f), na.rm = TRUE)]
p_star = current_assignment[as.character(t)]

# Initialise vector of kidneys that this patient is not allowed to receive
not_allowed = c(unlist(assigned_patients), p_star[[1]])

while (TRUE) {
  # Update preferences, taking into account that the patient is not allowed
  # to receive the kidneys in not_allowed
  p = preference_updater(preferences_original, not_allowed, patient_names,
                         available_kidneys, hpbm = TRUE, hpbm_patient = t)
  preferences = p$preferences
  current_assignment = p$current_assignment

  # Find the next favourite kidney of t
  p_prime = current_assignment[as.character(t)]

  if (p_prime == w) {
    # We do not want to assign someone to the waiting list via the HPBM method

    # Reset preferences to original...
    preferences = preferences_original

    # and continue with the highest-priority patient apart from t (so we need
    # to remove t from the possible candidates)
    candidates = candidates[-1]

    # Find next patient and their currently favourite kidney
    t = f[min(match(as.numeric(candidates), f), na.rm = TRUE)]
    p_star = current_assignment[as.character(t)]
    not_allowed = c(unlist(assigned_patients), p_star[[1]])

  } else if (p_prime %in% available_kidneys) {
    # This is allowed: we will assign this kidney to this person

    # Remove p_prime from kidneys and add t
    available_kidneys = available_kidneys[-which(
      available_kidneys == p_prime)]

    final_assignment = append(final_assignment, p_prime)

    return(list("final_assignment" = final_assignment,
                "available_kidneys" = c(available_kidneys, t),
                "assigned_patients" = c(assigned_patients, t),
                "preferences" = preferences,
```

```
                "current_assignment" = current_assignment))

  } else {
    # Try to find circles

    result = circle_chain_finder(current_assignment, available_kidneys, w)
    circles = result$circles

    if (t %in% unique(names(unlist(circles)))) {
      # Only consider cycles when t is in them
      t_circles = circles[sapply(circles, function(x) {t %in% names(x)})]
      circle_lengths = sapply(t_circles, length)

      # Find circles that obey the capacity constraint
      correct_index = which(circle_lengths <= q)
      correct_circles = circles[correct_index]

      if (length(correct_circles) > 0) {
        final_assignment = circle_assigner(final_assignment, correct_circles)

        # And kick them out / update preferences
        assigned_patients = c(assigned_patients, names(unlist(correct_circles)))

        update = preference_updater(preferences, assigned_patients,
                                    patient_names, available_kidneys)
        preferences = update$preferences
        current_assignment = update$current_assignment

        return(list("final_assignment" = final_assignment,
                    "available_kidneys" = available_kidneys,
                    "assigned_patients" = assigned_patients,
                    "preferences" = preferences,
                    "current_assignment" = current_assignment))
      } else {
        # Go to step Y; namely stay with t and update preferences
        # I.e. go to beginning of while loop

        not_allowed = c(not_allowed, p_prime[[as.character(t)]])
        next
      }
    } else {
      # Go to step Y; namely stay with t and update preferences
      # I.e. go to beginning of while loop

      not_allowed = c(not_allowed, p_prime[[as.character(t)]])
      next
    }
  }
}

# Return variables
return(list("final_assignment" = final_assignment,
            "available_kidneys" = available_kidneys,
```

```
              "assigned_patients" = assigned_patients,
              "preferences" = preferences,
              "current_assignment" = current_assignment))
}
```

```
#'Runs the TTCC algorithm with HPBM method
#'
#'\code{combined} runs the TTCC algorithm with Highest-Priority Breaking Method
#'(HPBM) when all available cycles exceed the capacity constraint \code{q}.
#'
#'@param \code{current_assignment} List with the currently favourite preferences
#'@param \code{preferences} Matrix with all preferences
#'@param \code{final_assignment} List with currently final assignment
#'@param \code{assigned_patients} List with already assigned patients
#'@param \code{patient_names} List with patient names (for generality)
#'@param \code{available_kidneys} List with available kidneys
#'@param \code{f} List with the priority ordering
#'@param \code{w} The numeric encoding for the waiting list
#'@param \code{q} The capacity constraint for length of a cycle
#'
#'@return A list with updated \code{current_assignment}, \code{preferences},
#'  \code{final_assignment}, \code{assigned_patients},
#'  and \code{available_kidneys}.
combined = function(current_assignment, preferences, final_assignment,
                    assigned_patients, patient_names, available_kidneys, f, w,
                    q) {

  result = circle_chain_finder(current_assignment, available_kidneys, w)
  circles = result$circles
  chains = result$chains

  while (length(circles) > 0) {
    # Then circles exist; calculate lengths of these circles...
    circle_lengths = sapply(circles, length)

    # And find circles that obey the capacity constraint
    correct_index = which(circle_lengths <= q)
    correct_circles = circles[correct_index]

    if (length(correct_circles) > 0) {
      # There are correct circles, assign these

      final_assignment = circle_assigner(final_assignment, correct_circles)

      # And kick them out / update preferences
      assigned_patients = c(assigned_patients, names(unlist(correct_circles)))
      update = preference_updater(preferences, assigned_patients, patient_names,
                                  available_kidneys)
      preferences = update$preferences
      current_assignment = update$current_assignment

      # Update circles variable; remove assigned circles
      circles = circles[-correct_index]
```

```
    } else {
      # Then there are circles, but these are all too long -> apply HPBM
      hpbm_out = hpbm(circles, preferences, assigned_patients, patient_names,
            current_assignment, final_assignment, f, w, available_kidneys, q)

      return(list("current_assignment" = hpbm_out$current_assignment,
                  "preferences" = hpbm_out$preferences,
                  "final_assignment" = hpbm_out$final_assignment,
                  "assigned_patients" = hpbm_out$assigned_patients,
                  "available_kidneys" = hpbm_out$available_kidneys))
    }
  }

  # No cycles exist, so a w-chain must be used
  result = chain_assigner(final_assignment, chains, f, available_kidneys,
                          assigned_patients)
  final_assignment = result$final_assignment
  available_kidneys = result$available_kidneys
  assigned_patients = result$assigned_patients

  update = preference_updater(preferences, assigned_patients, patient_names,
                              available_kidneys)
  current_assignment = update$current_assignment

  return(list("current_assignment" = current_assignment,
              "preferences" = preferences,
              "final_assignment" = final_assignment,
              "assigned_patients" = assigned_patients,
              "available_kidneys" = available_kidneys))
}
```

```
#### RUN FUNCTIONS ####
data = import_data(file_location = "data/dataset7.xlsx")

# Initialise kidneys: make patients "point" to currently "favourite" kidneys
p = preference_updater(data$preferences,
                       assigned_patients = c(),
                       patient_names = data$patient_names,
                       available_kidneys = c())
current_assignment = p$current_assignment

# Clean up; `p` is not needed anymore in the rest of this function and will only
# take up workspace memort
rm(p)

# Initialise list of input
outcome = list("current_assignment" = current_assignment,
               "preferences" = data$preferences,
               "final_assignment" = list(),
               "assigned_patients" = c(),
               "available_kidneys" = c(),
               "patient_names" = data$patient_names,
               "f" = data$f,
               "w" = data$w)
```

```r
number_of_runs = 1
while (length(outcome$final_assignment) != length(data$f)) {
  # Continue running the algorithm until everyone has been assigned

  print(paste("Current run:", number_of_runs))

  outcome = combined(current_assignment = outcome$current_assignment,
                     preferences = outcome$preferences,
                     final_assignment = outcome$final_assignment,
                     assigned_patients = outcome$assigned_patients,
                     available_kidneys = outcome$available_kidneys,
                     patient_names = data$patient_names,
                     f = data$f,
                     w = data$w,
                     q = 3)

  # Remove duplicates from the final_assignment
  outcome$final_assignment = uniquefy_list(outcome$final_assignment)

  # Print the final assignment to retain output when the code, unfortunately,
  # does not finish
  print("Current assignment:")
  print(outcome$final_assignment)
  print("----------")

  number_of_runs = number_of_runs + 1
}

assignment = uniquefy_list(outcome$final_assignment)
print(assignment)
```