The University of Sheffield

Assignment Cover Sheet

Student Number: 210116317

Programme of Study: M.Sc in Data Analytics

Module Name: Statistical Data Analysis in R

Module Code: MAS6024

Title of Project: MAS6024 Assignment

1.0 Introduction

1.1 Objective

To implement a strategy to finish the game in the fewest moves possible. The game should be in compliance with the rules given and the code should be easy to follow through.

1.2 Initial Setup

The first step in the implementation is to place the letters in a 8 by 8 matrix to form a game board. The function **get_start_position** return a random integer representing the index of the starting position.(P.S. We have represented all the squares by a number between 1 to 64.)

```r
lgrid <- matrix(NA, nrow = 8, ncol = 8)
lgrid[1, ] <- c("r", "l", "q", "s", "t", "z", "c", "a")
lgrid[2, ] <- c("i", "v", "d", "z", "h", "l", "t", "p")
lgrid[3, ] <- c("u", "r", "o", "y", "w", "c", "a", "c")
lgrid[4, ] <- c("x", "r", "f", "n", "d", "p", "g", "v")
lgrid[5, ] <- c("h", "j", "f", "f", "k", "h", "g", "m")
lgrid[6, ] <- c("k", "y", "e", "x", "x", "g", "k", "i")
lgrid[7, ] <- c("l", "q", "e", "q", "f", "u", "e", "b")
lgrid[8, ] <- c("l", "s", "d", "h", "i", "k", "y", "n")

edge_squares = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 16, 17, 24, 25, 32, 33, 40, 41, 48, 49,
    56, 57, 58, 59, 60, 61, 62, 63, 64)  # indices of all the edge squares
green_squares = c(14, 23, 42, 51)  # indices of the green squares on the board

# Function to get a staring position randomly
get_start_position <- function() {
    choices <- setdiff(1:64, edge_squares)  # to get a position not on edge
    position <- sample(choices, 1, replace = TRUE)
    return(position)
}
```

2.0 Part-1

2.1 Solution

According to the rules given there are two scenarios:

1. Token is **on edge** square- token should move at random to one of the 64 squares

2. Token is **not on edge** square- token should move to one of the adjacent squares

```r
move_if_on_edge <- function() {
  position <- sample(1:64, 1, replace = TRUE)
  return(position)
}
move_if_not_on_edge <- function(square) {
  dist_from_adj_squares <- c(1, 7, 8, 9)
  # Adding & subtracting these from current square would give all adjacent squares
  adj_squares_1 <- square - dist_from_adj_squares
  adj_squares_2 <- square + dist_from_adj_squares
  adj_squares <- c(adj_squares_1, adj_squares_2)
  position <- sample(adj_squares, 1, replace = TRUE)
  return(position)
}

play_next_turn <- function(position) {
  if (position %in% edge_squares) next_position <- move_if_on_edge() else
    next_position <- move_if_not_on_edge(position)
  return(next_position)
}
play_next_turn(16)
```

```
## [1] 25
```

3.0 Part-2

3.1 Solution

The basic rule that we are applying here is that a five-letter palindrome requires only **three** different letters and **two** of them would be repeated. The following rules are used to decide whether or not the player should pick up the letters:

1. Always pick the letter on the starting square.
2. Check if the letter in next turn is already in the collection.
3. If yes, then add that letter to the collection. Now we need only three more letters out of which two must be identical.
4. If no, then add that letter to collection. Now we need one **new** letter and two same letters that we have in our collection.
5. If we have already picked up three different letters we will not pick any other letter except for two duplicates of our collection.

3.2 Methodology

This strategy would work because it fulfill all the necessary conditions and abide by all the rules. The strategy is tested for different scenarios. The design can be improved if the occurrences of letters in the current square and the adjacent squares can be recorded and the decision is made based on them. This is would certainly increase the complexity of the program but can produce efficient outcomes.

4.0 Part-3

4.1 Solution

Firstly, we define two matrices. **letter_collection** matrix will hold our final palindrome string while **available_letter** matrix will hold the different letters we pick as we progress in the game.

```
letter_collection <- c()
available_letters <- c()
num_letters <- 0
count <- 0
```

After that we define two functions to handle green and white squares respectively.

```
handle_green_squares <- function(square, p){
  prob <- sample(c(p, 1-p), 1, replace = TRUE)

  if(prob == p){
    letter_collection <<- c("f", "f")
    count <<- 1
    available_letters <<- c("h", "k")
    num_letters <<- 3
  }

  else if(prob == 1-p){
    if(lgrid[square] %in% available_letters){
      available_letters <<- available_letters[available_letters != lgrid[square]]
      num_letters <<- num_letters -1
    }

    if(lgrid[square] %in% letter_collection){
      letter_collection <<- letter_collection[letter_collection != lgrid[square]]
      count <<- max(0, length(letter_collection) - 1)
    }
  }
}

handle_white_squares <- function(square){
  if(lgrid[square] %in% available_letters && count < 2){
    letter_collection <<- append(letter_collection, c(lgrid[square], lgrid[square]))
    available_letters <<- available_letters[available_letters != lgrid[square]]
    count <<- count + 1  #Ensure we collect copy of only two letters
  }

  else if(num_letters < 3){
    available_letters <<- append(available_letters, lgrid[square])
    num_letters <<- num_letters + 1
    #Ensure we do not pick more than 3 different letters
  }
}
```

After that we count the number of moves to finish the game using the following function.

```
pick_or_leave_letter <- function(square, p) {
    if (square %in% green_squares)
        handle_green_squares(square, p) else handle_white_squares(square)
}

count_num_moves <- function(start, p) {
    letter_collection <<- c()  #Clear data of last run
    available_letters <<- c()
    num_letters <<- 0
    count <<- 0
    curr_position <- start
    end_game <- FALSE
    moves <- 0

    while (end_game == FALSE) {
        if (count == 2 && length(available_letters) == 1) {
            letter_collection <<- append(letter_collection, available_letters)
            available_letters <<- c()
            end_game <- TRUE
        } else {
            pick_or_leave_letter(curr_position, p)
            next_pos <- play_next_turn(curr_position)
            curr_position <- next_pos
            moves <- moves + 1
        }
    }

    return(moves)
}

(start <- get_start_position())
## [1] 35
count_num_moves(start, 0.46)
## [1] 20
```

5.0 Part-4

5.1 Solution

Square D4 is equal to the square 28. Here, the **count__num__moves** function is called multiple times with random probabilities values. Fig. 5.1.a explains the distribution of number of moves w.r.t probability. We can see in most cases the number of moves are under 40. From the plot it is clear that the number of moves are comparitively less for higher probabilities than lower values of the same.

```
df <- data.frame(p = c(), moves = c())
for (i in 1:20) {
    p <- round(runif(1), 3)  # To get random probability values
    moves <- count_num_moves(28, p)
    df <- rbind(df, data.frame(p = p, moves = moves))
}
ggplot(aes(p, moves), data = df) + geom_line() + geom_point()
```

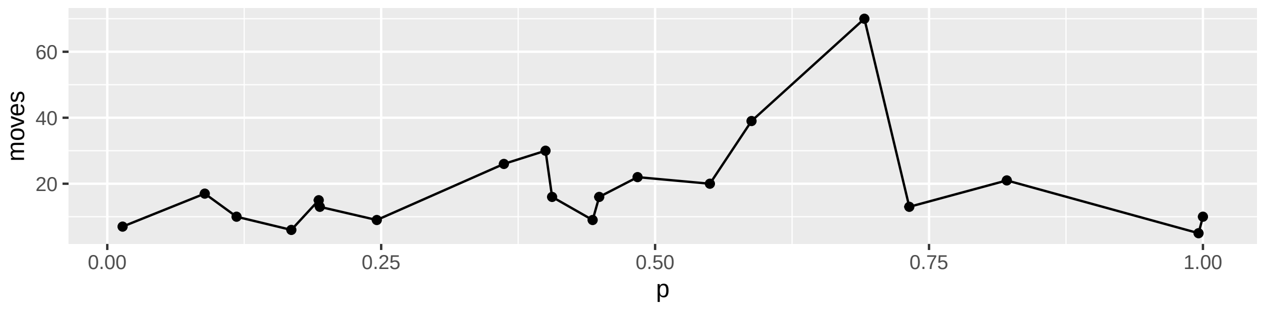

Figure 1: Fig. 5.1.a Probability v/s Number of moves

6.0 Part-5

6.1 Solution

As is clearly depicted from Fig.6.1.a, the probability distribution of both the squares is given by two parallel lines. This means that the number of moves required to finish the game is almost same for both the squares with given probabilities. Fig. 6.1.b shows that number of moves is under 40 in most cases for both squares.

The Summary statistics also show that there is a very small difference between the stats for the squares proving that the number of moves is almost similar for both.

```
sample_results <- replicate(100, count_num_moves(46, 0.05))
sample_results2 <- replicate(100, count_num_moves(28, 0.95))
results1 <- data.frame(p = 0.05, moves = sample_results)
results2 <- data.frame(p = 0.95, moves = sample_results2)
summary(sample_results)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    5.00    7.00   12.50   14.84   19.00   47.00
```

```
summary(sample_results2)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    5.00    8.00   13.00   17.26   24.00   57.00
```

```
ggplot(aes(moves, p, color = "results1"), data = results1) + geom_point() + geom_point(a
    p, color = "results2"), data = results2)
```

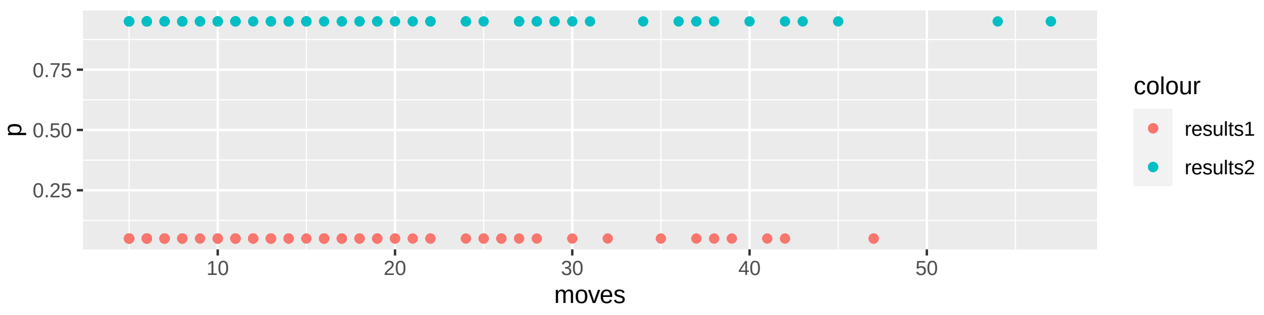

Figure 2: Fig. 6.1.a Number of moves v/s probability for F6 and D4

```
par(mfrow = c(1, 2))
hist(sample_results)
hist(sample_results2)
```

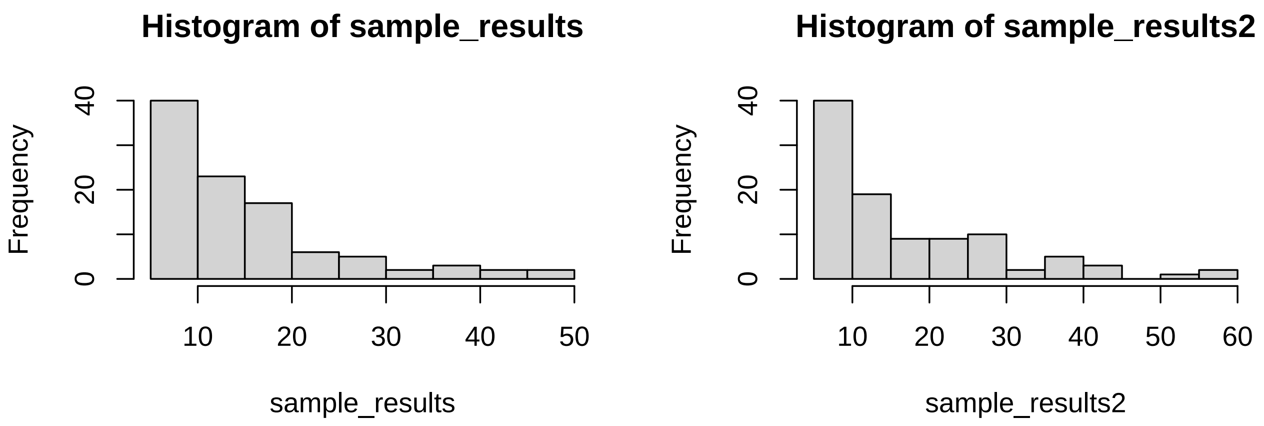

Figure 3: Fig.6.1.b Histogram of number of moves required for F6 and D4

7.0 Part-6

7.1 Solution

As is clearly evident from the results of following code, neither the expected values nor the average moves for square A and square B are not equal.

```
Xa <- c(25, 13, 16, 24, 11, 12, 24, 26, 15, 19, 34)
Xb <- c(35, 41, 23, 26, 18, 15, 33, 42, 18, 47, 21, 26)
p <- 0.5
expected_A <- sum(p * Xa)
expected_B <- sum(p * Xb)
print(c(expected_A, expected_B))
```

```
## [1] 109.5 172.5
```

```
avg_A <- mean(Xa)
avg_B <- mean(Xb)
print(c(avg_A, avg_B))
```

```
## [1] 19.90909 28.75000
```