Lab1 - Rummikub First Set

Bryce Samwel

4/26/2020

Rummikub Explanation

Rummikub is a game in which players draw from a pool of 106 tiles and attempt to be the first player to place all of their tiles on the table/play area. 104 tiles are numbered and 2 are wild. The numbered tiles have a face value between 1 and 13 as well as a color. A tile can have a black, blue, red, or orange number displayed. There are 2 total combinations of all colors and numbers, representing the 104 numbered tiles. The 2 wild tiles can represent any possible tile.

The game begins with every player randomly selecting 14 tiles from the pool to place in their rack. In order to make their first move, a player must be able to play a set from their rack totaling at least 30 points. A set is a combination of at least three tiles which can include a run of tiles of the same color, or matching numbers with differing colors (i.e. three black tiles numbered 1, 2, & 3, or a blue 12, a black 12, and a red 12). Once a player has entered the game, they may place their other tiles onto existing sets, and alter existing sets in play in order to remove all of their tiles from their rack.

For the purposes of this lab, additional rules do not need to be explained.

Lab Introduction

For this Lab, there are several questions that we want to answer about how the beginning of a *Rummikub* game might play out.

1. What is the probability of being able to go down on the initial draw of 14 tiles?
2. How many times must a player draw, on average, before they are able to go down?
3. What's the largest number of times a player can draw before they can go down?
4. Do the results of the previous questions depend on how many players there are?

In all honesty, the final question is almost impossible because in order to test the effect of additional players, a full version of the game would need to be simulated in order for the other players to know when they need to draw. On that note, however, we should talk about some of our simplifying assumptions!

Simplifying Assumptions and Modeling Choices

Being able to find every single possible combination of tiles that makes a set with a given rack is very challenging, bordering on a waste of time. For example, it's difficult to account for a player that has the perfect tiles where they have the choice between making a run with three black tiles, or using one of those black tiles in a set of the same number. Instead, I took a different approach. I chose to first look for any instances of number sets in the player's rack without using wilds (first looking for sets of four, then of three).

Next, I looked for runs without wilds. This is where my second simplifying assumption comes in. Technically, it's possible to draw a run of 13. However, the odds of getting a run greater than 4 are fleeting. So, I chose

to only look for runs of length 5, 4, and 3, in that order. I also indexed backwards in order to find the larger valued runs first.

Once I had exhausted all of my options without wilds, I did the exact same search while including wilds. The only difference here is that I indexed the search for number sets backwards as well, to ensure that wild cards would be used on larger numbers. The previous assumptions will lead to rare occasions where overlapping sets might not be used optimally. This would effectively increase the number of draws taken by the player, but likely by a negligible amount.

The final assumption I made strictly applies to the final question of interest, which I mentioned in the previous section. Instead of simulating the entire game in order to determine how often opponents may need to draw, I will assume that on every turn, each player must draw. This assumption will cause the number of tiles in the pool to decrease faster, reducing our player's chances of acquiring the tiles they need. However, we also assume that our player gets to pick their 14 tiles before everyone else. These two assumptions combined likely produce negligible effects on the results of the simulation.

The Code

The code is rather lengthy, but I will take a moment to explain certain chunks of code that I had to get creative with in order to do these simulations.

To begin, I intialized all of my variables and matrices. I then sampled 14 numbers from the values 1:106 in order to simulate the player selecting their first 14 tiles. Obviously, the numbers 1:106 are not the numbers that are present on the real tiles in the game. The method for converting them is explained in the next code block. However, you can see at the bottom of this section of code, I assign the numbers 105 and 106 to be wild tiles.

```
N <- 10000 #Number of simulated games until the rack is playable
opponents <- 1 #Max number of players is 4 according to the game
sim_draws <- matrix(0,1,N)

for (a in 1:N){

  Pool <- c(1:106)
  Rack <- sample(x = Pool, size = 14)
  Pool <- Pool[!Pool %in% Rack] #Remove Rack from Pool

  oppDraws <- sample(x=Pool,size = 14*opponents)
  Pool <- Pool[!Pool %in% oppDraws] #Remove oppDraws from Pool

  mRack <- matrix(0, 4, 13)
  wild <- 0
  total <- 0
  draws <- 0

  #Count the number of wilds in rack
  for (i in 1:length(Rack)){
    if ( Rack[i] > 104 ){
      wild <- wild + 1
      }
  }

#The first thing to note in the following chunk of code is how I chose to assign the numbers 1:106 to t

  #Count number of each tile in the player's rack
```

```
  for (i in 1:length(Rack)){
    if ( Rack[i] < 27 ){
      mRack[1,ceiling(Rack[i]/2)] <- mRack[1,ceiling(Rack[i]/2)] + 1
    }
    else if ( Rack[i] < 53 ){
      mRack[2,ceiling((Rack[i]-26)/2)] <- mRack[2,ceiling((Rack[i]-26)/2)] + 1
    }
    else if ( Rack[i] < 79 ){
      mRack[3,ceiling((Rack[i]-52)/2)] <- mRack[3,ceiling((Rack[i]-52)/2)] + 1
    }
    else if ( Rack[i] < 105 ){
      mRack[4,ceiling((Rack[i]-78)/2)] <- mRack[4,ceiling((Rack[i]-78)/2)] + 1
    }
  }

#This snippet of code will emphasise the method I used throughout the entire simulation in order to sea

  #Check to see if starting rack is playable

  #Check for Number Sets without Wilds
  for (i in 13:1){
    if (sum(mRack[,i]>0)==4){
      total <- total + 4*i
      mRack[,i]<-mRack[,i] - (mRack[,i]>0)
    }
    else if (sum(mRack[,i]>0)==3){
      total <- total + 3*i
      mRack[,i]<-mRack[,i] - (mRack[,i]>0)
    }
  }

#The same methodology we used to look at number sets is used to look for runs. However, since there are

  #Check for runs of 5 without wilds
  for (j in 1:4){
    for(i in 13:5){
      if (sum(mRack[j,(i-4):i]>0)==5){
        total <- total + 5*i - 10
        mRack[j,(i-4):i] <- mRack[j,(i-4):i] - (mRack[j,(i-4):i]>0)
      }
    }
  }

  #Check for runs of 4 without wilds
  for (j in 1:4){
    for(i in 13:4){
      if (sum(mRack[j,(i-3):i]>0)==4){
        total <- total + 4*i - 6
        mRack[j,(i-3):i] <- mRack[j,(i-3):i] - (mRack[j,(i-3):i]>0)
      }
    }
  }
```

```
  #Check for runs of 3 without wilds
  for (j in 1:4){
    for(i in 13:3){
      if (sum(mRack[j,(i-2):i]>0)==3){
        total <- total + 3*i - 3
        mRack[j,(i-2):i] <- mRack[j,(i-2):i] - (mRack[j,(i-2):i]>0)
      }
    }
  }

#I also want to take a quick moment to show the next for-loop which looks for number sets wile using wi

  #Check for Number sets with wilds
  for (i in 13:1){ #Index backwards to use wilds on larger sets
    if (sum(mRack[,i]>0)+wild>2){
      total <- total + 3*i
      wild <- wild - (3 - sum(mRack[,i]>0))
      mRack[,i]<-mRack[,i] - (mRack[,i]>0)
    }
  }

  #Check for runs of 5 with wilds
  for (j in 1:4){
    for(i in 13:5){
      if (sum(mRack[j,(i-4):i]>0)+wild > 4){
        total <- total + 5*i - 10
        wild <- wild - (5 - sum(mRack[j,(i-4):i]>0))
        mRack[j,(i-4):i] <- mRack[j,(i-4):i] - (mRack[j,(i-4):i]>0)
      }
    }
  }

  #Check for runs of 4 with wilds
  for (j in 1:4){
    for(i in 13:4){
      if (sum(mRack[j,(i-3):i]>0)+wild > 3){
        total <- total + 4*i - 6
        wild <- wild - (4 - sum(mRack[j,(i-3):i]>0))
        mRack[j,(i-3):i] <- mRack[j,(i-3):i] - (mRack[j,(i-3):i]>0)
      }
    }
  }

  #Check for runs of 3 with wilds
  for (j in 1:4){
    for(i in 13:3){
      if (sum(mRack[j,(i-2):i]>0)+wild > 2){
        total <- total + 3*i - 3
        wild <- wild - (3 - sum(mRack[j,(i-2):i]>0))
        mRack[j,(i-2):i] <- mRack[j,(i-2):i] - (mRack[j,(i-2):i]>0)
      }
    }
  }
```

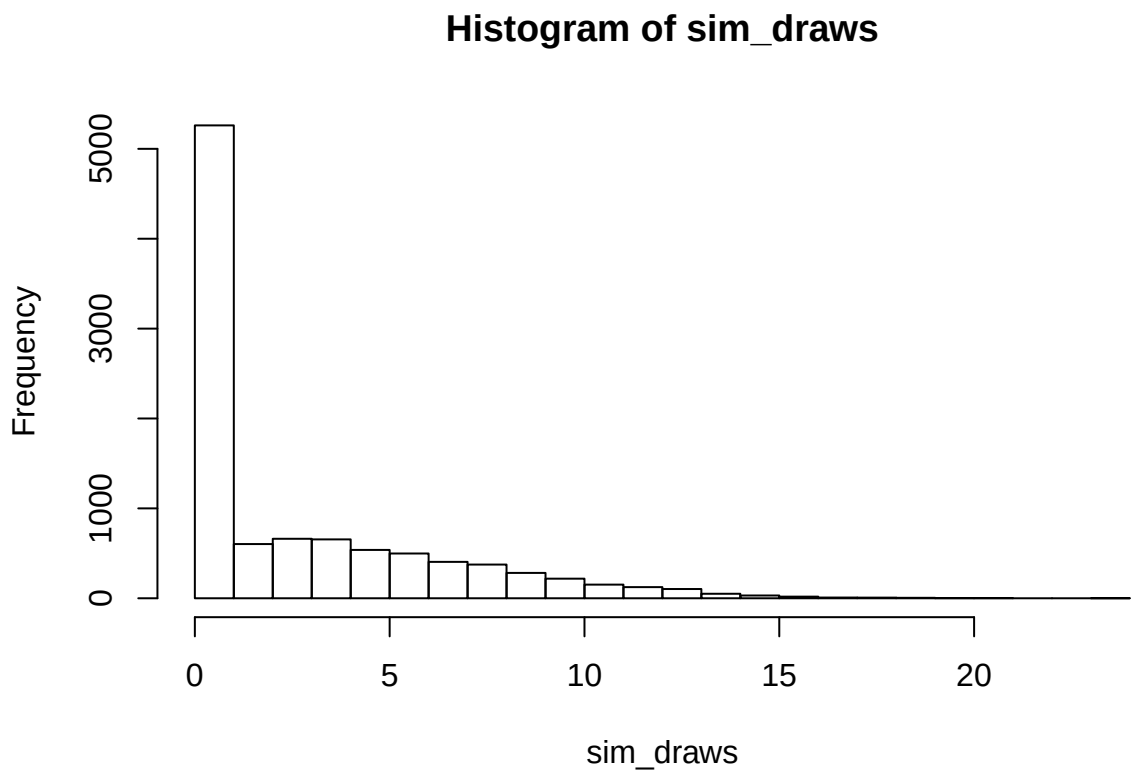
```r
#Once the first searches have been completed, we can now enter the loop if our total isn't greater than

  #Loop while total < 30 & length(Pool)>0
  while(total<30){

    #Draw another tile
    draws <- draws + 1
    newTile <- sample(x = Pool, size = 1)
    Pool <- Pool[!Pool %in% newTile] #Remove newTile from Pool

    newOppTiles <- sample(x = Pool, size = opponents)
    Pool <- Pool[!Pool %in% newOppTiles] #Remove newOppTiles from Pool

    #Add a wild if the newTile is one

    if ( newTile > 104 ){
      wild <- wild + 1
    }


    #Add newTile to the player's rack
    if ( newTile < 27 ){
      mRack[1,ceiling(newTile/2)] <- mRack[1,ceiling(newTile/2)] + 1
    }
    else if ( newTile < 53 ){
      mRack[2,ceiling((newTile-26)/2)] <- mRack[2,ceiling((newTile-26)/2)] + 1
    }
    else if ( newTile < 79 ){
      mRack[3,ceiling((newTile-52)/2)] <- mRack[3,ceiling((newTile-52)/2)] + 1
    }
    else if ( newTile < 105 ){
      mRack[4,ceiling((newTile-78)/2)] <- mRack[4,ceiling((newTile-78)/2)] + 1
    }


    #Check for Number Sets without Wilds
    for (i in 13:1){
      if (sum(mRack[,i]>0)==4){
        total <- total + 4*i
        mRack[,i]<-mRack[,i] - (mRack[,i]>0)
      }
      else if (sum(mRack[,i]>0)==3){
        total <- total + 3*i
        mRack[,i]<-mRack[,i] - (mRack[,i]>0)
      }
    }

    #Check for runs of 5 without wilds
    for (j in 1:4){
      for(i in 13:5){
        if (sum(mRack[j,(i-4):i]>0)==5){
          total <- total + 5*i - 10
          mRack[j,(i-4):i] <- mRack[j,(i-4):i] - (mRack[j,(i-4):i]>0)
```

```
    }
  }
}

#Check for runs of 4 without wilds
for (j in 1:4){
  for(i in 13:4){
    if (sum(mRack[j,(i-3):i]>0)==4){
      total <- total + 4*i - 6
      mRack[j,(i-3):i] <- mRack[j,(i-3):i] - (mRack[j,(i-3):i]>0)
    }
  }
}

#Check for runs of 3 without wilds
for (j in 1:4){
  for(i in 13:3){
    if (sum(mRack[j,(i-2):i]>0)==3){
      total <- total + 3*i - 3
      mRack[j,(i-2):i] <- mRack[j,(i-2):i] - (mRack[j,(i-2):i]>0)
    }
  }
}

#Check for Number sets with wilds
for (i in 13:1){ #Index backwards to use wilds on larger sets
  if (sum(mRack[,i]>0)+wild>2){
    total <- total + 3*i
    wild <- wild - (3 - sum(mRack[,i]>0))
    mRack[,i]<-mRack[,i] - (mRack[,i]>0)
  }
}

#Check for runs of 5 with wilds
for (j in 1:4){
  for(i in 13:5){
    if (sum(mRack[j,(i-4):i]>0)+wild > 4){
      total <- total + 5*i - 10
      wild <- wild - (5 - sum(mRack[j,(i-4):i]>0))
      mRack[j,(i-4):i] <- mRack[j,(i-4):i] - (mRack[j,(i-4):i]>0)
    }
  }
}

#Check for runs of 4 with wilds
for (j in 1:4){
  for(i in 13:4){
    if (sum(mRack[j,(i-3):i]>0)+wild > 3){
      total <- total + 4*i - 6
      wild <- wild - (4 - sum(mRack[j,(i-3):i]>0))
      mRack[j,(i-3):i] <- mRack[j,(i-3):i] - (mRack[j,(i-3):i]>0)
    }
  }
```

```
    }

    #Check for runs of 3 with wilds
    for (j in 1:4){
      for(i in 13:3){
        if (sum(mRack[j,(i-2):i]>0)+wild > 2){
          total <- total + 3*i - 3
          wild <- wild - (3 - sum(mRack[j,(i-2):i]>0))
          mRack[j,(i-2):i] <- mRack[j,(i-2):i] - (mRack[j,(i-2):i]>0)
        }
      }
    }
    if (length(Pool)==0){
      break
    }
  }
  sim_draws[a] <- draws
}
```

The results in the following histogram are for a game between two people (or one opponent to our player), simulated 10,000 times.

```
hist(sim_draws, breaks = (max(sim_draws)-1))
```



The average and maximum number of draws taken in this simulation are:

```r
mean(sim_draws)
```

```
## [1] 2.9111
```

```r
max(sim_draws)
```

```
## [1] 24
```

The probability of being able to play on the first turn is:

```r
sum(sim_draws==0)/N
```

```
## [1] 0.4608
```

Analysis and Answers

The answers to the first three questions are visible above. When adding another player to the game, the results don't really change. During some simulations, the average drops by as much as 0.1 turns on average. However, it gets a little more interesting when we add a fourth player. This is the maximum number of recommended players for the game. What we find is that the average number of draws doesn't change, but our maximum number of draws hits a natural boundary. With four players in the game, there can be no more than 12 additional draws by each player. This forces our maximum to be 12 and approximately 3% of the simulated games end at twelve draws for our player. Obviously, this wouldn't happen in a real game. We made the assumption that every player always drew, essentially assuming that no player is ever able to play a set. To account for this, we could instead assume that opponents need to draw only a fraction of the time. With three opponents, each needing to draw an additional card every three turns, we find that the average number of draws still doesn't change much, but the maimum number of draws shoots back up near 20.

Resources, Challenges, and Conclusions

Whenever I encountered a situation where I wasn't sure what to do, Google was a fabulous asset! Most of my problems were simply with not being too familiar with R. For a while I couldn't figure out why the simulation almost always had the player going down on their first turn. I learned that these two lines of code don't do the same thing! It seems rather obvious in hindsight...

```r
z <- 5
z-2:5
```

```
## [1] 3 2 1 0
```

```r
(z-2):5
```

```
## [1] 3 4 5
```

Besides that, it took me quite a while to simply get the lab started. I was really wrestling with how I wanted to define the pool of tiles and how I could define the player's rack, all while being able to reference between the two in order to properly add and sutract tiles from both. Ultimately, I'm happy with the solution I came up with. It was a little tricky to wrap my mind around at the beginnning, but I figured it out eventually!

The results of the simulations actually surprised me a litle bit. I chose this lab because I had played *Rummikub* quite recently and had a game where I almost went the entire duration of the game without being able to lay down! When the results of the simulation showed that players can go down on their first move almost 50% of the time, I was pissed! Bad luck, I suppose! I hope you enjoyed this lab and found the results interesting yourself!