Pirates!

Dominic Walden

September 12, 2017

1 License

2 Introduction

This is my attempt to solve a problem set for me by James Bach in the style of conjecture and refutation as described by Popper[1] and Lakatos[2]. My interest is to see how useful it is to apply this methodology to software testing.

I do not manage to completely solve the problem in this way, but it does take me most of the way there. I have removed my complete solution because I do not wish to spoil this puzzle for anyone else attempting to solve it.

It includes an example of an automated check used to refute some of my conjectures.

All the code is in the language GNU R[3].

[1]Popper, K. "The Logic Of Scientific Discovery". Routledge. 1959.

[2]Lakatos, I. "Proofs and Refutations". Cambridge University Press. 1976. `https://math.berkeley.edu/~kpmann/Lakatos.pdf`.

[3]`https://www.r-project.org/`

3 My Attempt

The problem:

> Thirteen pirates go on an extended voyage, pillaging and plundering from Africa to Asia. By the end they have quite a stash–too much to take back with them. They decide to lock it in a chest, leave the chest on an island, and come back for it a year later. Of course, not being terribly trusting, they want to ensure that none of them can come back early and claim the treasure for himself. They could just put 13 locks on it and each take a key, but a pirate's life is dangerous–they may not all be around in a year. What they want is for any majority of the original thirteen to be able to open the chest, while any fewer will be locked out. How many locks will it take for them to achieve this? The locks they are using are quite simple. Each key opens only one lock (no master keys), but keys can be duplicated, so multiple pirates can have keys to the same lock.
>
> So, write a program that will calculate the number of keys required total, the number of keys per pirate, and the exact set of keys for each pirate, for any number of pirates total and any number of pirates in the subset that can open the chest.
>
> My program takes this input:
>
> usage: keys.pl select total
>
> select = minimum pirates to open chest total = total pirates

In what follows:

P = Total number of pirates

p = minimum pirates to open chest

T = Total number of locks

t = Number of keys per pirate

In the case that p = 1 then one lock will suffice.

In the case that p = P then each pirate will need a unique key so T = p = P.

Let us explore P = 3, p = 2. If T = 1 then there would only need to be one pirate to open it. If T = 2 then let us call the two keys for the locks A and

B. If one pirate had A and another B then if the third had A he/she could unlock the chest with the pirate who had B but not with the one who had A. Similarly if he/she had B.

It will work if T = 3. Let us call the three keys A, B and C. One pirate could have A and B, another A and C and the third B and C. In symbols: {A, B}, {A, C}, {B, C}. It can be shown that this will fulfil our requirements:

1. $\{A, B\} \cup \{A, C\} = \{A, B, C\}$
2. $\{A, B\} \cup \{B, C\} = \{A, B, C\}$
3. $\{A, C\} \cup \{B, C\} = \{A, B, C\}$.

There are some conjectures we can draw from the above:

1. T = P
2. t = p.

Conjecture 1. is contradicted by the case that p = 1 which we have already established means T = 1. We need only make P > 1 and we have falsified the conjecture.

For 2., if P = 4 and p = 2 then combinations are {A, B}, {A, C}, {B, C} and one of the previous 3 repeated, say {B, C}. t = 2 would no longer work as $\{B, C\} \cup \{B, C\} = \{B, C\}$ and be unable to open all the locks.

For the latter example, T = P = 4 and t = P - 1 = 3 will work. The way the keys are distributed are determined by the binomial combination $\binom{T}{t}$. In the latter example:

```
combn(c("A","B","C","D"), 3)
```

A	A	A	B
B	B	C	C
C	D	D	D

where each pirate gets a column.

T = P and t = P - 1 should generalise to all values of P where p = 2. $\binom{T}{T-1}$ will always produce T combinations, one for each pirate. Each pirate has all but one of the keys. Any pirate, A, will have a unique combination of keys and any other pirate, B, will have a different combination which will contain the one key A does not have.

Might this generalise? Say that for any values of P and p: T = P, t = P - (p - 1), and the distribution of keys is determined by the binomial combination $\binom{T}{t}$.

An experiment (check) which may refute this method would be to take all combinations $\binom{T}{t}$ and find p of these whose union is less than T (where T and t are as above and P and p are arbitrary.)

This is quite simple. For P = 4 and p = 3:

```
combn(c("A","B","C","D"), 2)
```

```
A  A  A  B  B  C
B  C  D  C  D  D
```

Choosing, for example, the first, second and fourth rows:

```
length(union(c("A", "B"), union(c("A", "C"), c("B", "C"))))
```

```
3
```

Binomial combinations make it so that every pirate does not have exactly the same combination of keys as any other pirate. However, a pirate may have all his/her keys in common with another pair of pirates.[4]

What might be more fruitful is to consider designs. Let X be any set, Y a set of k-subsets of X such that each member of X belongs to exactly r of the subsets in Y.

The binomial combination $\binom{T}{t}$ where T = P and t = P - 1 mentioned above is a special example of a design, where the size of X is P and $k = r = t = P-1$. Similar to before r = P - (p - 1).

An interesting conjecture might be: our requirements are satisfied by a design where X = Y = P and r = k = P - (p - 1).

[4]Actually, I have missed a step in the above insofar as I needed to first select 4 out of the 6 columns which will represent the combination of keys for the 4 pirates. I am convinced there is no combination of 4 columns from the above 6 which achieves our purposes, but I do not want to prove that yet as I want to pursue potentially more fruitful paths.

At this point I will introduce the first potentially useful check. It checks that every combination of p - 1 pirates does not have T total keys and that every combination of p pirates does have T total keys.

I will apply this check to one such design for the case P = 7, p = 4. X = Y = 7, k = r = 7 - (4 - 1) = 4.

```
# Convenience function
unionRecursive <- function (thelist) {
    if(length(thelist) == 1) {
        thelist[[1]]
    } else {
        union(thelist[[1]],
              unionRecursive(thelist[-1]))
    }
}

allKeyComb <- function(T, t) {
    data.frame(combn(T, t))
}

checkComb <- function(comb, p) {
    output <- list()
    for(pirates in data.frame(combn(comb, p))) {
        output[[(length(output)+1)]] <- unionRecursive(pirates)
    }
    output
}

checkCombLength <- function(comb, P, p) {
    T <- length(data.frame(combn(P, p - 1)))
    output <- TRUE

    # Every combination of p - 1 pirates should not have T total keys
    for(vect in checkComb(comb, p - 1)) {
        if(length(vect) == T) {
            output <- FALSE
        }
    }
    # Every combinations of p pirates should have T total keys
    for(vect in checkComb(comb, p)) {
        if(length(vect) != T) {
            output <- FALSE
        }
    }

    output
}

design <- data.frame(c(1,2,3,4),c(4,5,6,7),c(1,3,5,7),c(1,2,6,7),c(3,4,5,6),c(1,2,3,5)
checkComb(design, 4)
```

```
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  4  4  4  4  4  4  4  4
 2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  5  5  5  5  5  5  5  5
 3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  6  6  6  6  6  6  6  6
 4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  7  7  7  7  7  7  7  7
 5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  6  6  6  5  1  1  1  1  1  1  1  1
 6  6  6  6  6  6  6  6  6  6  7  7  7  7  7  7  7  7  7  6  3  3  3  3  3  3  2  2
 7  7  7  7  7  7  7  7  7  7  6  6  6  6  6  6  5  5  5  7  2  2  2  2  2  2  3  3
```

Each combination of 4 pirates has all the keys necessary to open the locks. However, when I ran the same code with "checkComb(design, 3)" there were combinations whose union added up to T. In other words, some combinations of 3 pirates could also open all the locks.

It occured to me at this point in time that I already knew the properties of the object I wanted to create, so why not construct it directly.

For any P and p, we associate with each member of binomial combinations $\binom{P}{p-1}$ a unique key and assign it to every other pirate not in that combination. Therefore, each combination of p - 1 pirates has one key which none of its members has but every other pirate does. Note T = $\binom{P}{p-1}$.

```
locksNeeded <- function(P, p) {

    # Initialise list
    pirates <- list()
    for(i in 1:P) {
        pirates[[i]] <- 0
    }

    # First value of the unique key
    uKey <- 1
    # All combinations of p - 1 pirates
    smallComb <- data.frame(combn(P, p - 1))

    # For each of the p - 1 pirate combinations
    for(comb in smallComb) {

        # For each pirate in P
        for(i in 1:P) {

            # If pirate is not in the combination
            if(!is.element(i, comb)) {

                if(all(pirates[[i]] == 0)) {
                    pirates[[i]] <- uKey
                } else {
                    # Give pirate the unique key for that combination
                    pirates[[i]] <- append(pirates[[i]], uKey)
                }

            }

        }

        # New unique key for next combination of pirates
        uKey <- uKey + 1

    }

    pirates

}
```