Cavit Cakir 23657 Homework 3

1 . I created the hash table with linear probing.

```
class HashTable
{
public:
    explicit HashTable(int size = 10000 ); // constructor
    returnType contains( const int & x) const; // tries to find and returns probe number

    void makeEmpty( ); // make the table empty
    returnType insert( const int & x ); // inserting new number to the table
    returnType remove( const int & x ); // removing from table
    int getCurrentSize(){ // returns currentSize
        return currentSize;
    };
    int getTotalSize(){ // returns totalSize
        return totalSize;
    };
    enum EntryType { ACTIVE, EMPTY, DELETED }; // types to check if its active, empty or deleted

private:
    struct HashEntry
    {
        int element;
        EntryType info;

        HashEntry( const int & e = int( ), EntryType i = EMPTY )
        : element( e ), info( i ) { }
    };

    vector<HashEntry> array;
    int currentSize;
    int totalSize;

    bool isActive( int currentPos ) const{ // returns if currentPos is active or not
        return array[currentPos].info == ACTIVE;
    };

    int findPos( const int & x, int & probe ) const; // finds position of x and returns it, increments probe count too.
    int myhash(const int & x) const{ // x mod M
        return x % totalSize;
    };
};
```

2. I created a function for 3 combination(6-1-1, 4-2-2, 2-1-5)

```
int chooseSelect(const int & number , const int & combination){ // 0 for 6-1-1 ~~ 1 for 4-2-2 ~~ 2 for 2-1-5
    if(combination == 0){
        if(number < 6)
            return 0; // insert
        else if(number < 7)
            return 1; // delete
        else
            return 2; // find
    }
    else if(combination == 1){
        if(number < 4)
            return 0; // insert
        else if(number < 6)
            return 1; // delete
        else
            return 2; // find
    }else{
        if(number < 2)
            return 0; // insert
        else if(number < 3)
            return 1; // delete
        else
            return 2; // find
    }
}
```

3. I created 6 [size+1] arrays then fill them up with # of probes and # of transactions.

```
element* insertionSuccess = new element[size+1];
element* insertionFailure = new element[size+1];
element* removeSuccess = new element[size+1];
element* removeFailure = new element[size+1];
element* findSuccess = new element[size+1];
element* findFailure = new element[size+1];
```

Then computed the averages of the transactions after that pushed that information to the 2D arrays.

```
if(insertionSuccess[j].transactions != 0)
    insertionSuccess2D[i][j] = insertionSuccess[j].probe / insertionSuccess[j].transactions;
```

4.I limited the range of the integers to 10M so that most integers have a decent chance of being deleted or searched. I generated such integers by generating a random number and taking mod 10M.

```
int toTransaction = rand() % (size * 10);
```

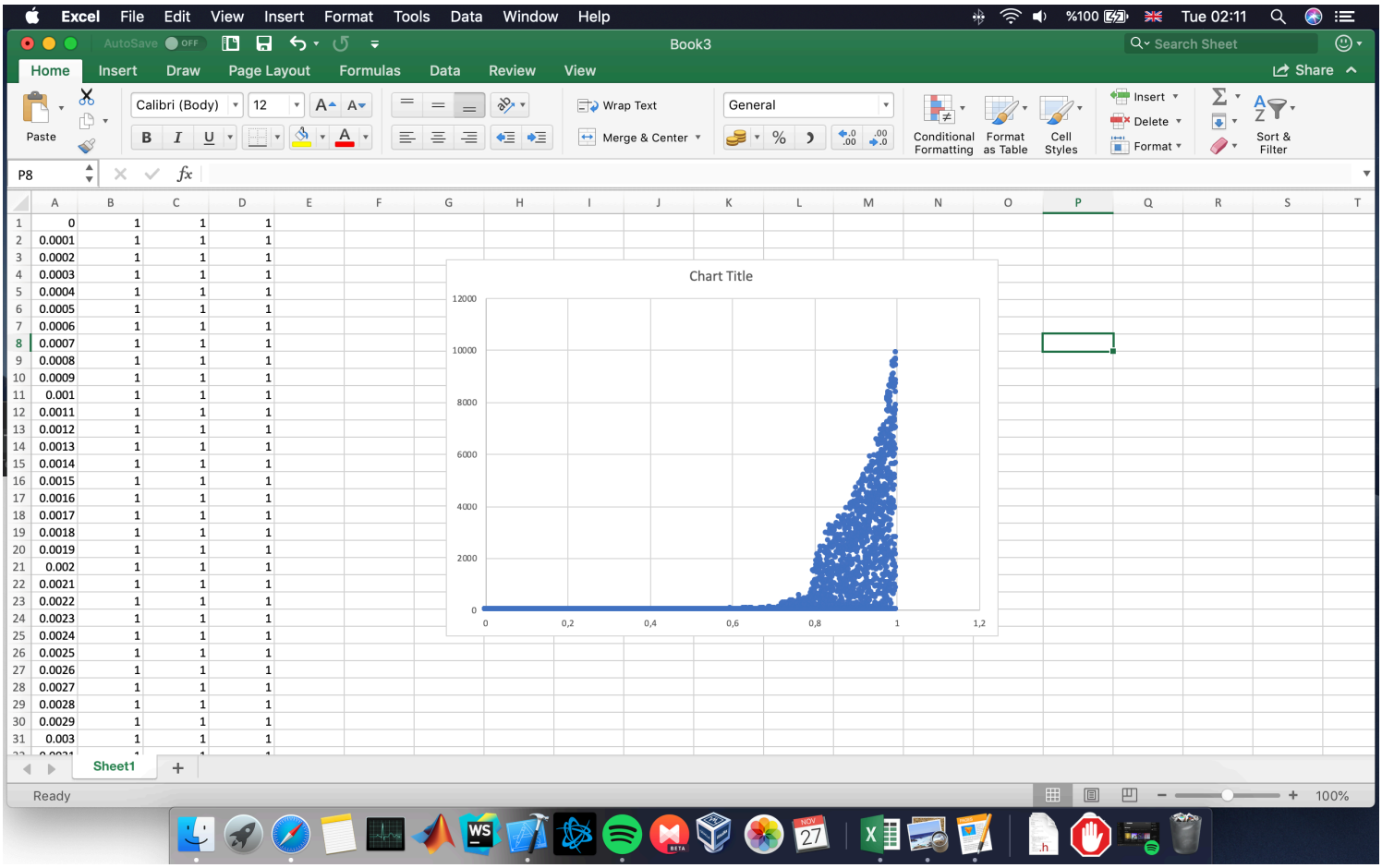
5.I generated transactions until the either the table becomes full or I have a total of 1,000,000 transactions.

```
int transactions = 1000000;
while(transactions && table.getCurrentSize() < size){
```

6. I put comma after each printing to file and i saved file as .csv so excel opens it perfectly and used setprecision(4) to get some observable values.

```
insertSuccessOS.open("insertSuccess.csv");
insertFailureOS.open("insertFailure.csv");
removeSuccessOS.open("removeSuccess.csv");
removeFailureOS.open("removeFailure.csv");
findSuccessOS.open("findSuccess.csv");
findFailureOS.open("findFailure.csv");

insertSuccessOS << fixed << setprecision(4);
insertFailureOS << fixed << setprecision(4);
removeSuccessOS << fixed << setprecision(4);
removeFailureOS << fixed << setprecision(4);
findSuccessOS << fixed << setprecision(4);
findFailureOS << fixed << setprecision(4);
```

7. For example successful insertion excel file looks like this.



8. My hash function is x mod M

```
int myhash(const int & x) const{ // x mod M
    return x % totalSize;
};
```

9. I took all the functions from book and changed couple of things.

=> findPos

```
int HashTable::findPos(const int & x, int & probe ) const{
    int pos = myhash(x);
    while(array[pos].info == ACTIVE && array[pos].element != x){
        pos++;
        probe++;

        if( pos >= array.size( ) )
            pos -= array.size( );
    }
    return pos;
}
```

=> insert

```
124
125 returnType HashTable::insert( const int & x ){
126     int probe = 1;
127     int currentPos = findPos( x , probe );
128     if(isActive( currentPos ) )
129         return returnType(probe,false);
130
131     array[ currentPos ].element = x;
132     array[ currentPos ].info = ACTIVE;
133     currentSize++;
134
135     return returnType(probe,true);
136 }
137
```

=> remove

```
137
138 returnType HashTable::remove(const int & x){
139
140     int probe = 1;
141     int currentPos = findPos( x, probe );
142     if(!isActive(currentPos))
143         return returnType(probe,false);
144
145     array[currentPos].info = DELETED;
146     currentSize--;
147     return returnType(probe,true);
148 }
149
```

=> contains

```
116
117 returnType HashTable::contains( const int & x) const{
118     int probe = 1;
119     if(isActive( findPos( x, probe ) ))
120         return returnType(probe,true);
121     else
122         return returnType(probe,false);
123 }
124
```

=> To choose prime size I implemented some prime functions

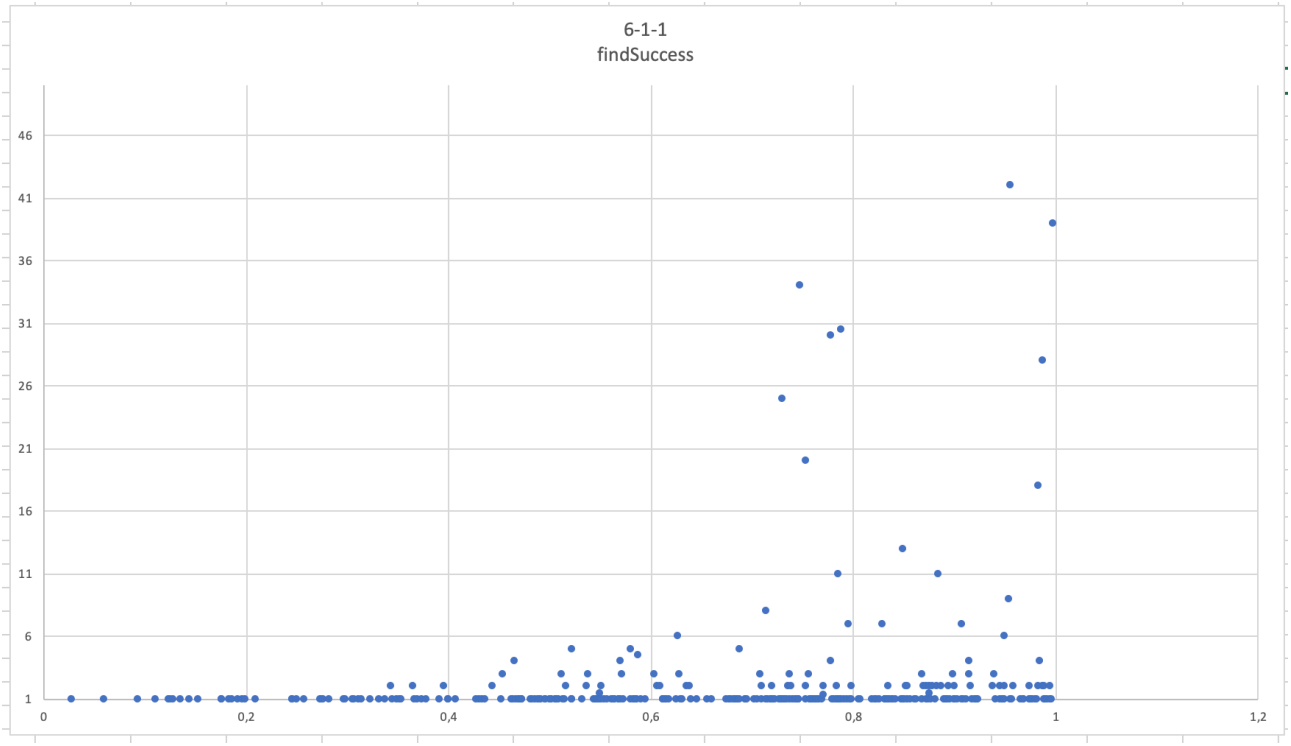
```
bool isPrime ( int x ){ // returns if x is a prime number
    for(int i = 2; i <= x / 2; ++i)
    {
        if(x % i == 0)
        {
            return false;
        }
    }
    return true;
}

int nextPrime( int n ) // finds smallest next prime number
{
    if( n % 2 == 0 )
        n++;

    for( ; !isPrime( n ); n += 2 )
        ;

    return n;
}
```

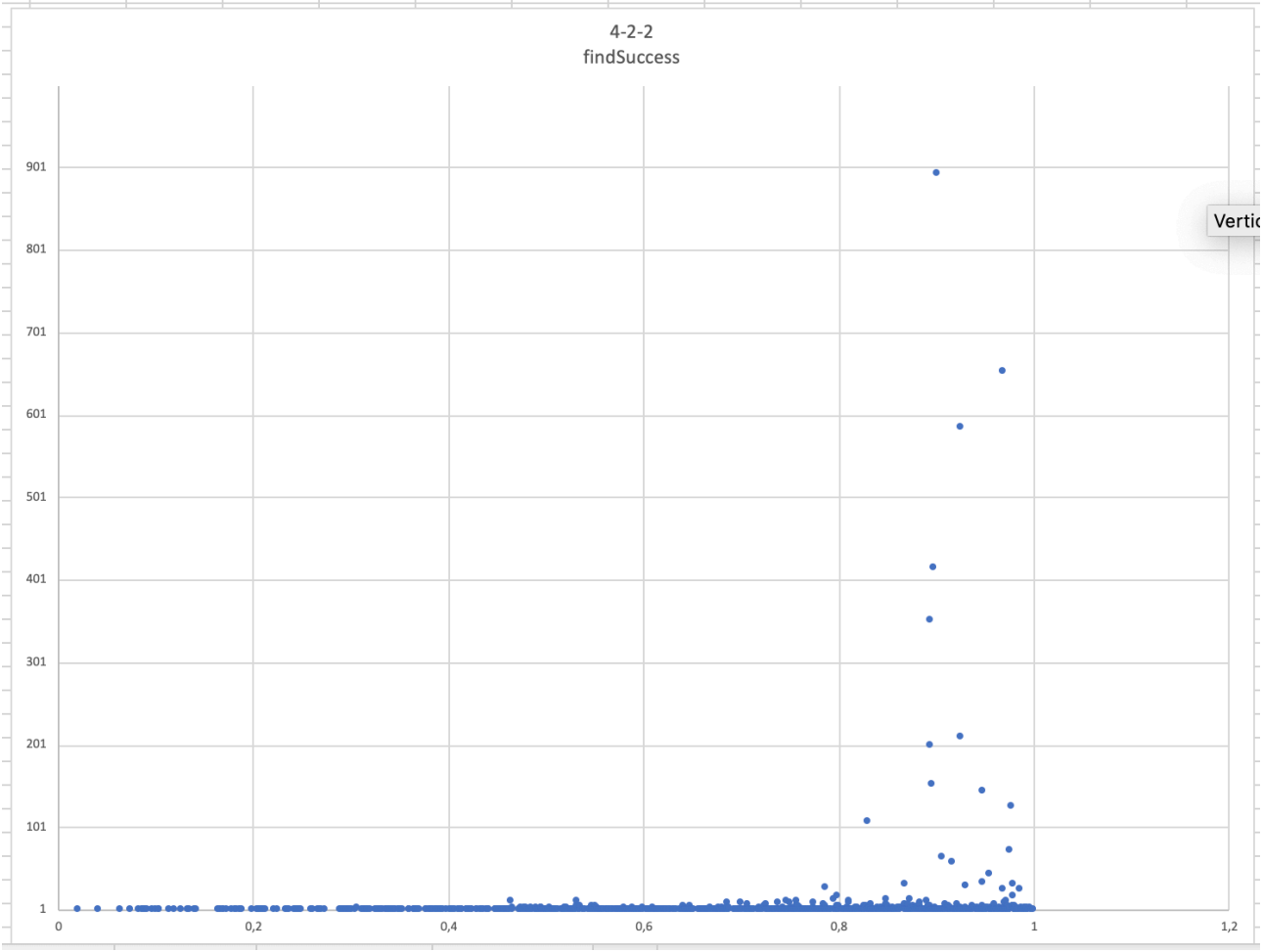
=> and my class looks like this

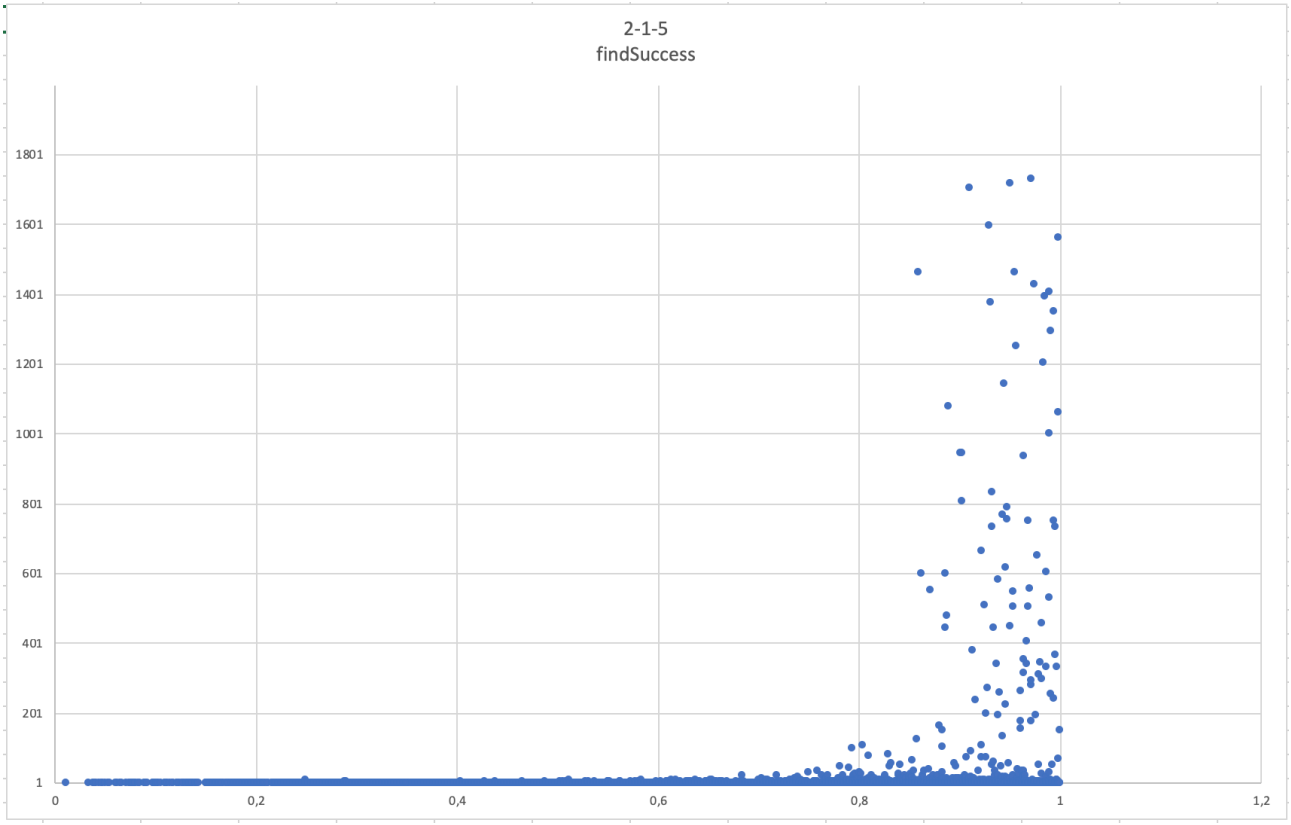
```
class HashTable
{
public:
    explicit HashTable(int size = 10000 ); // constructor
    returnType contains( const int & x) const; // tries to find and returns probe number

    void makeEmpty( ); // make the table empty
    returnType insert( const int & x ); // inserting new number to the table
    returnType remove( const int & x ); // removing from table
    int getCurrentSize(){ // returns currentSize
        return currentSize;
    };
    int getTotalSize(){ // returns totalSize
        return totalSize;
    };
    enum EntryType { ACTIVE, EMPTY, DELETED }; // types to check if its active, empty or deleted

private:
    struct HashEntry
    {
        int element;
        EntryType info;

        HashEntry( const int & e = int( ), EntryType i = EMPTY )
        : element( e ), info( i ) { }
    };

    vector<HashEntry> array;
    int currentSize;
    int totalSize;

    bool isActive( int currentPos ) const{ // returns if currentPos is active or not
        return array[currentPos].info == ACTIVE;
    };

    int findPos( const int & x, int & probe ) const; // finds position of x and returns it, increments probe count too.
    int myhash(const int & x) const{ // x mod M
        return x % totalSize;
    };
};
```

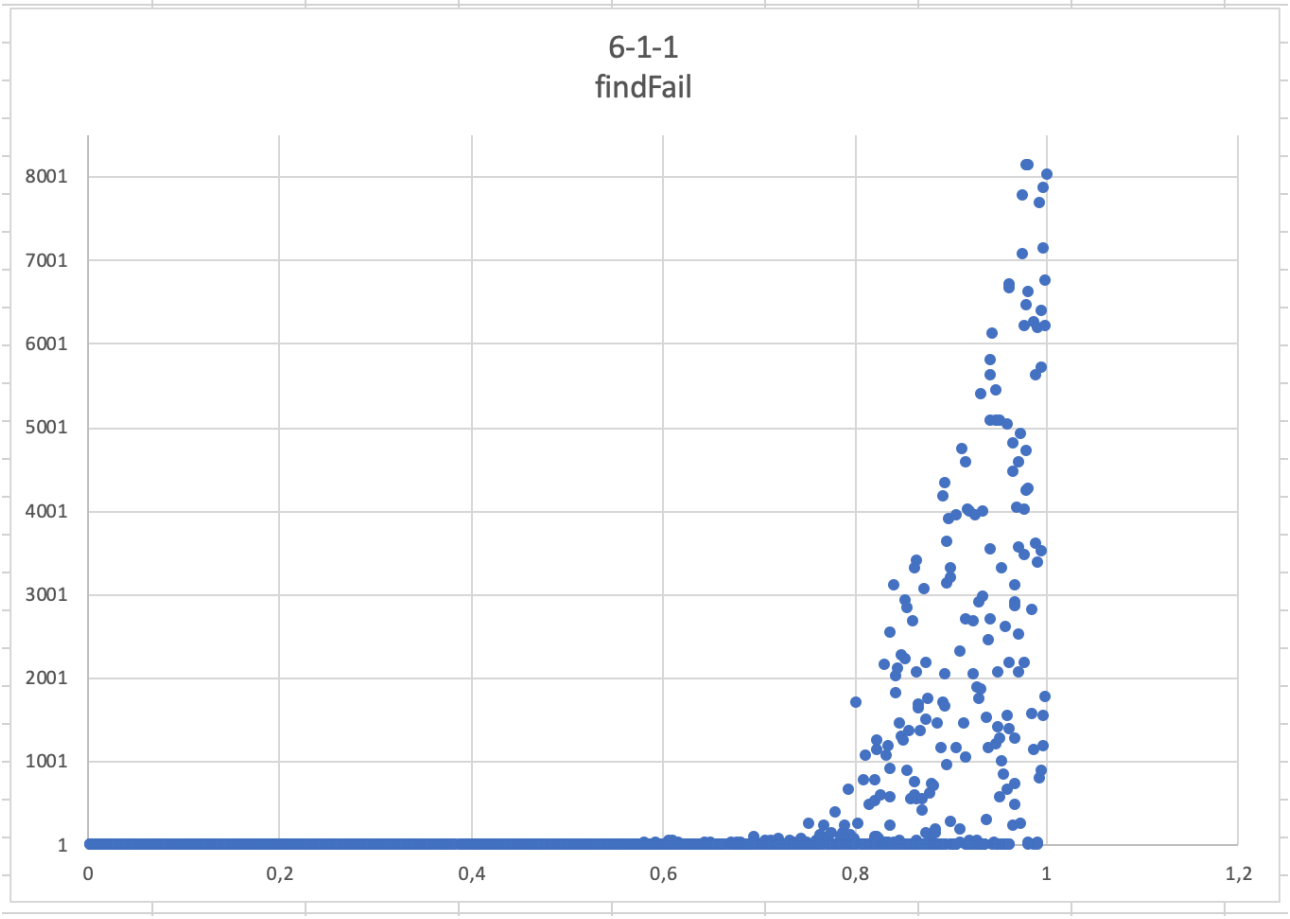
My Drawings

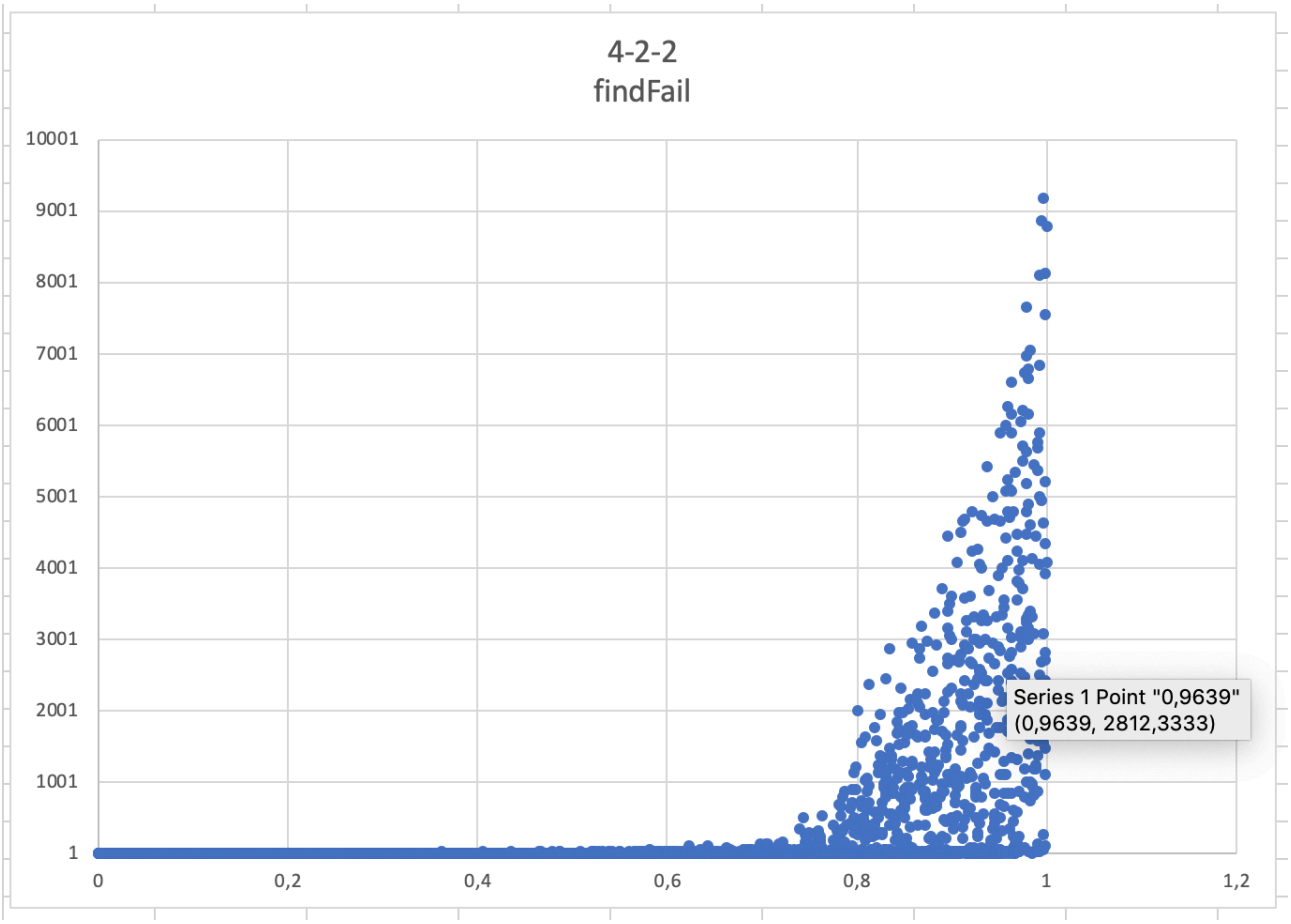
Note that all x-axis = load factor and y-axis =average number of probes

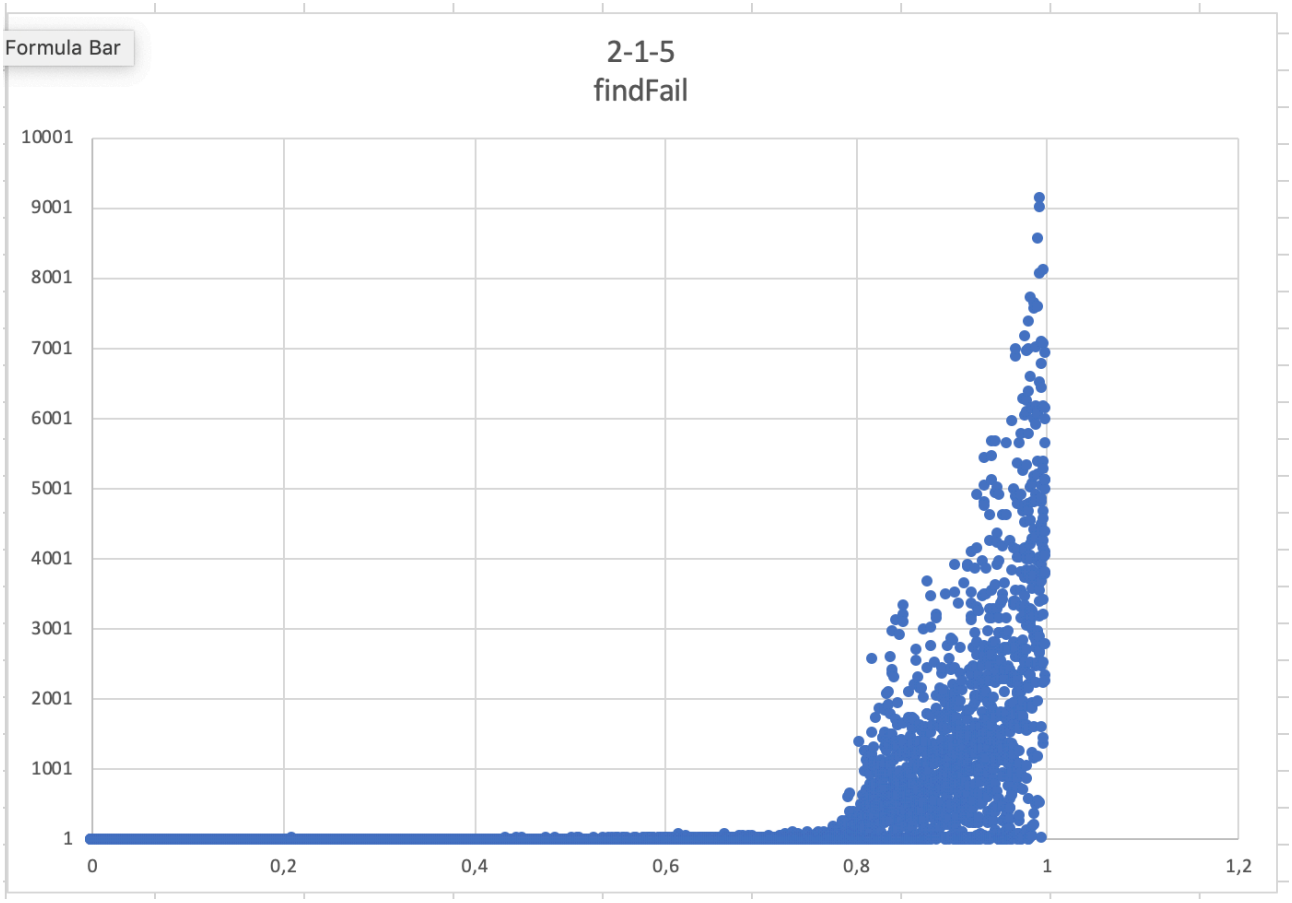


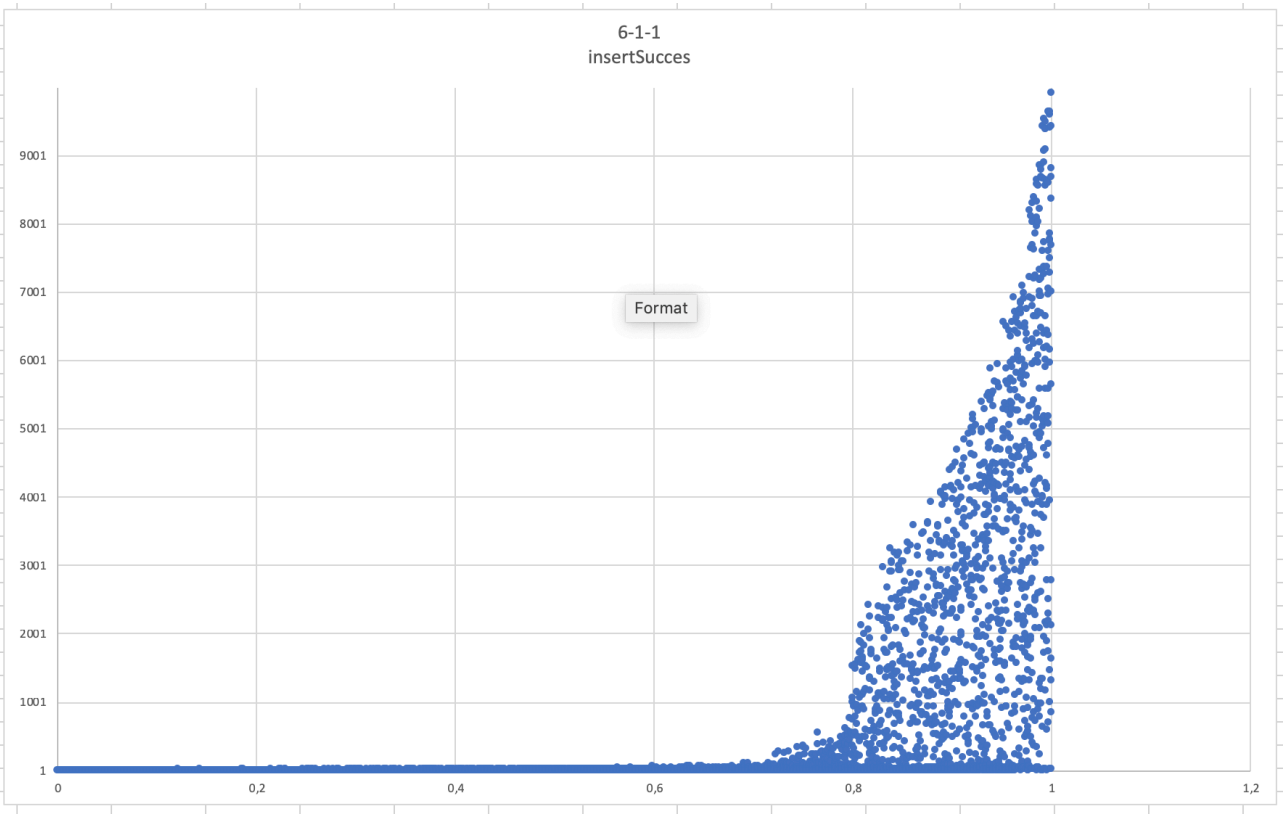


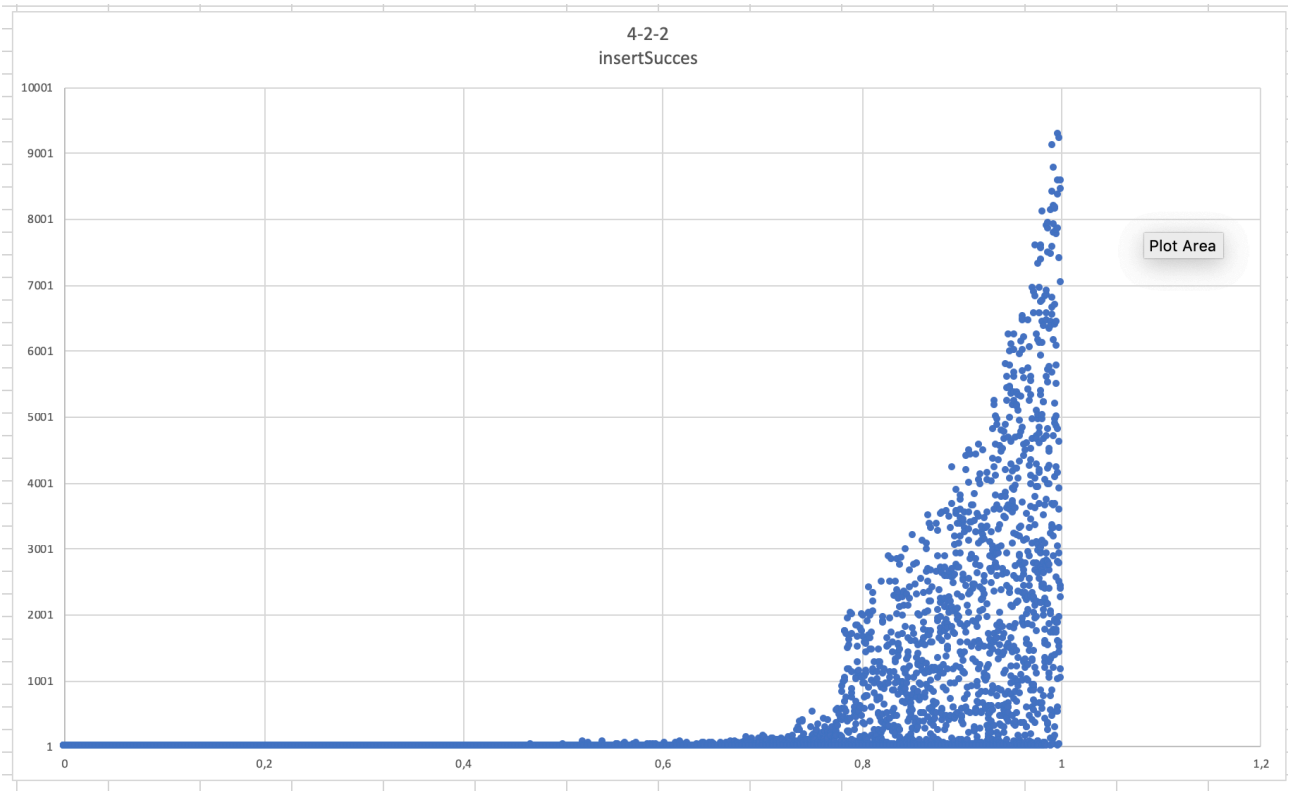

2-1-5
findSuccess
1801
1601
1401
1201
1001
801
601
401
201
1
0
0,2
0,4
0,6
0,8
1
1,2

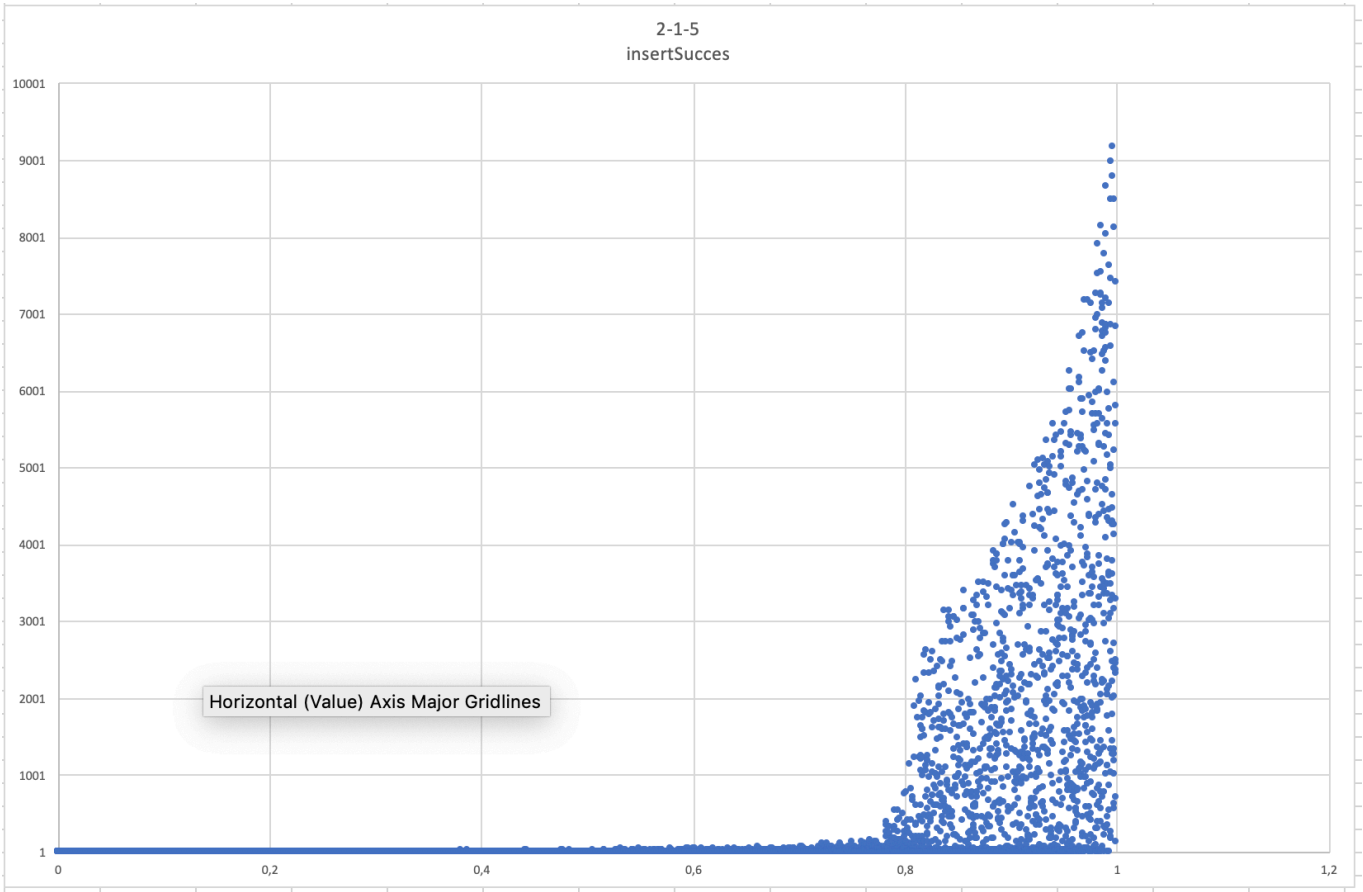

6-1-1
findFail
8001
7001
6001
5001
4001
3001
2001
1001
1
0
0,2
0,4
0,6
0,8
1
1,2

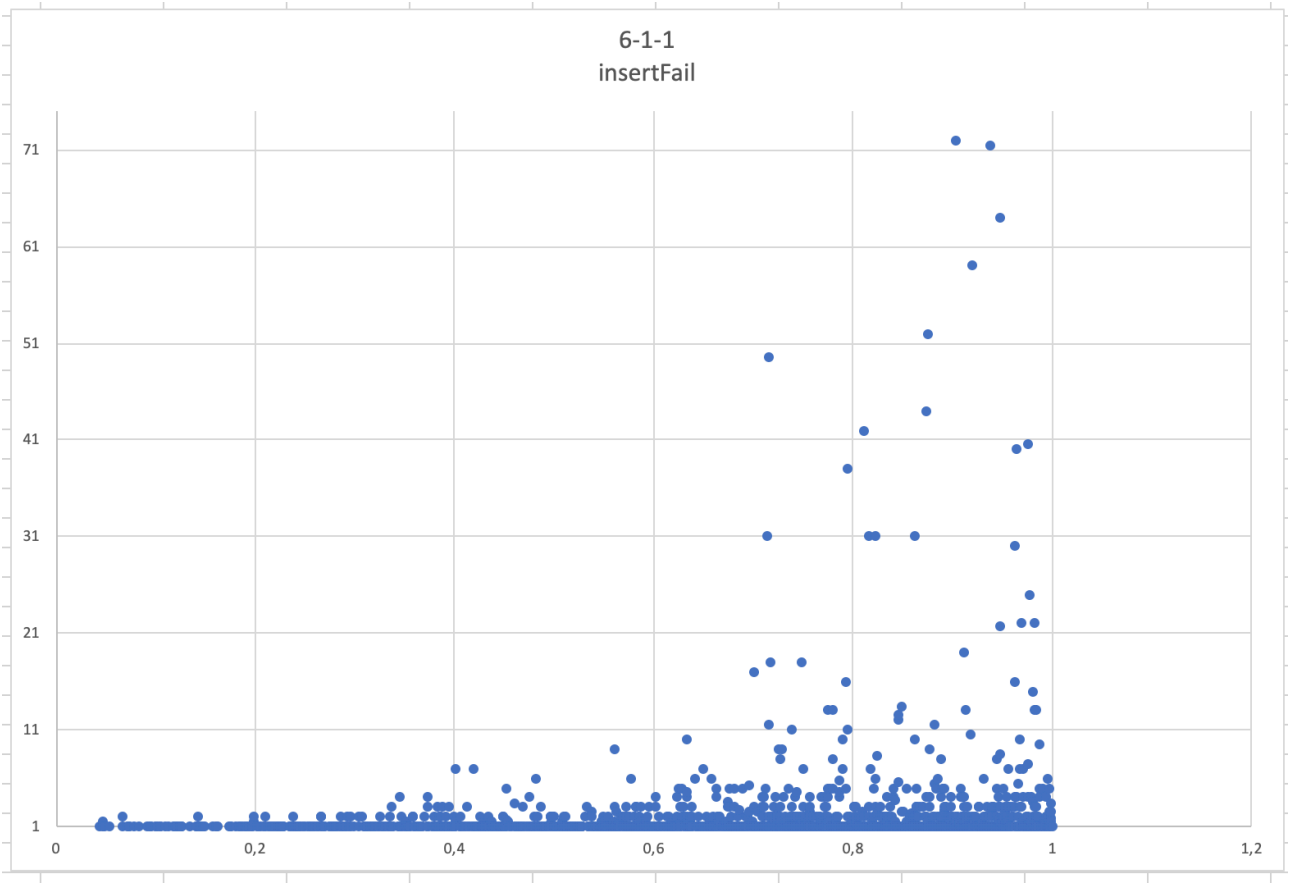

4-2-2
findFail
10001
9001
8001
7001
6001
5001
4001
3001
2001
1001
1
0
0,2
0,4
0,6
0,8
1
1,2
Series 1 Point "0,9639"
(0,9639, 2812,3333)

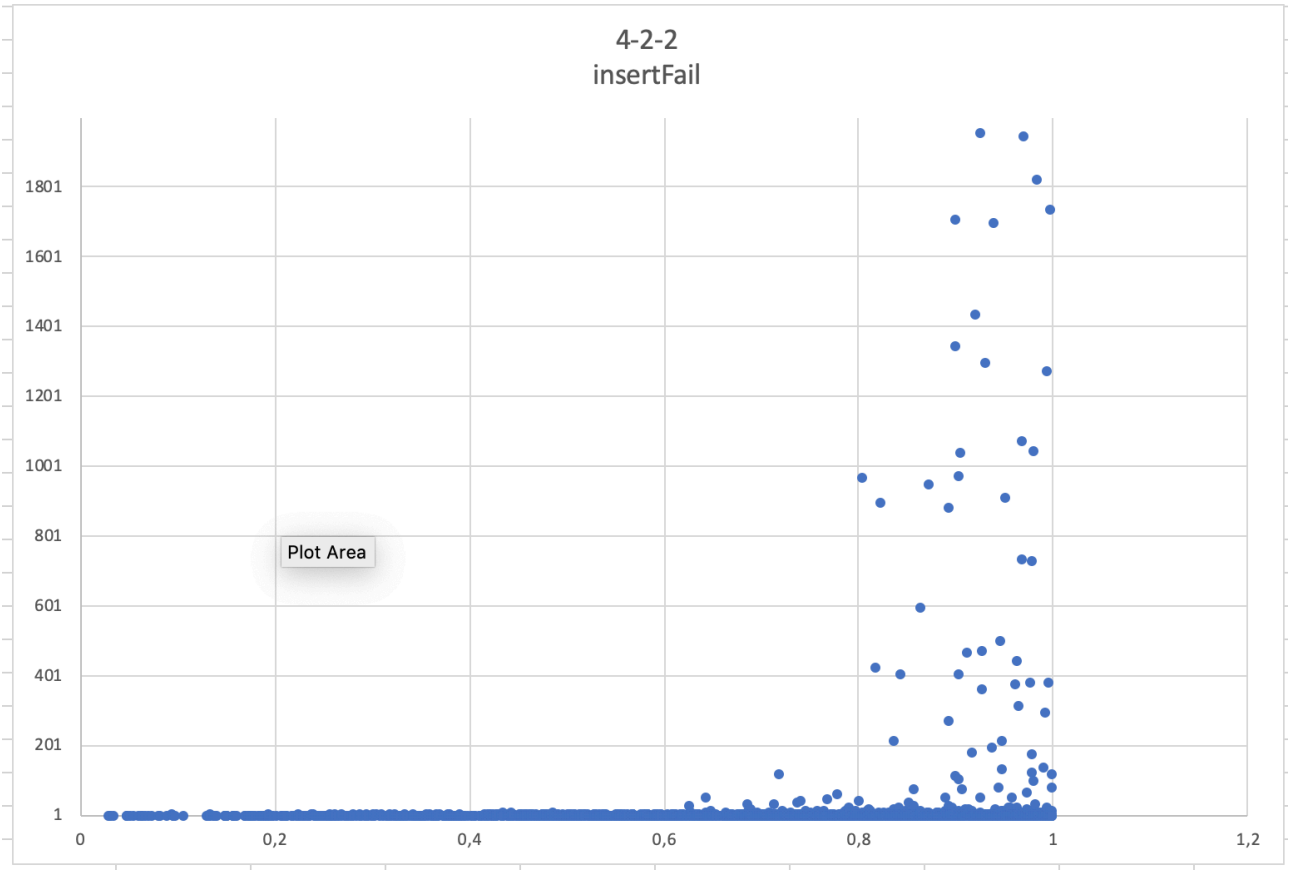

Formula Bar
2-1-5
findFail
10001
9001
8001
7001
6001
5001
4001
3001
2001
1001
1
0
0,2
0,4
0,6
0,8
1
1,2

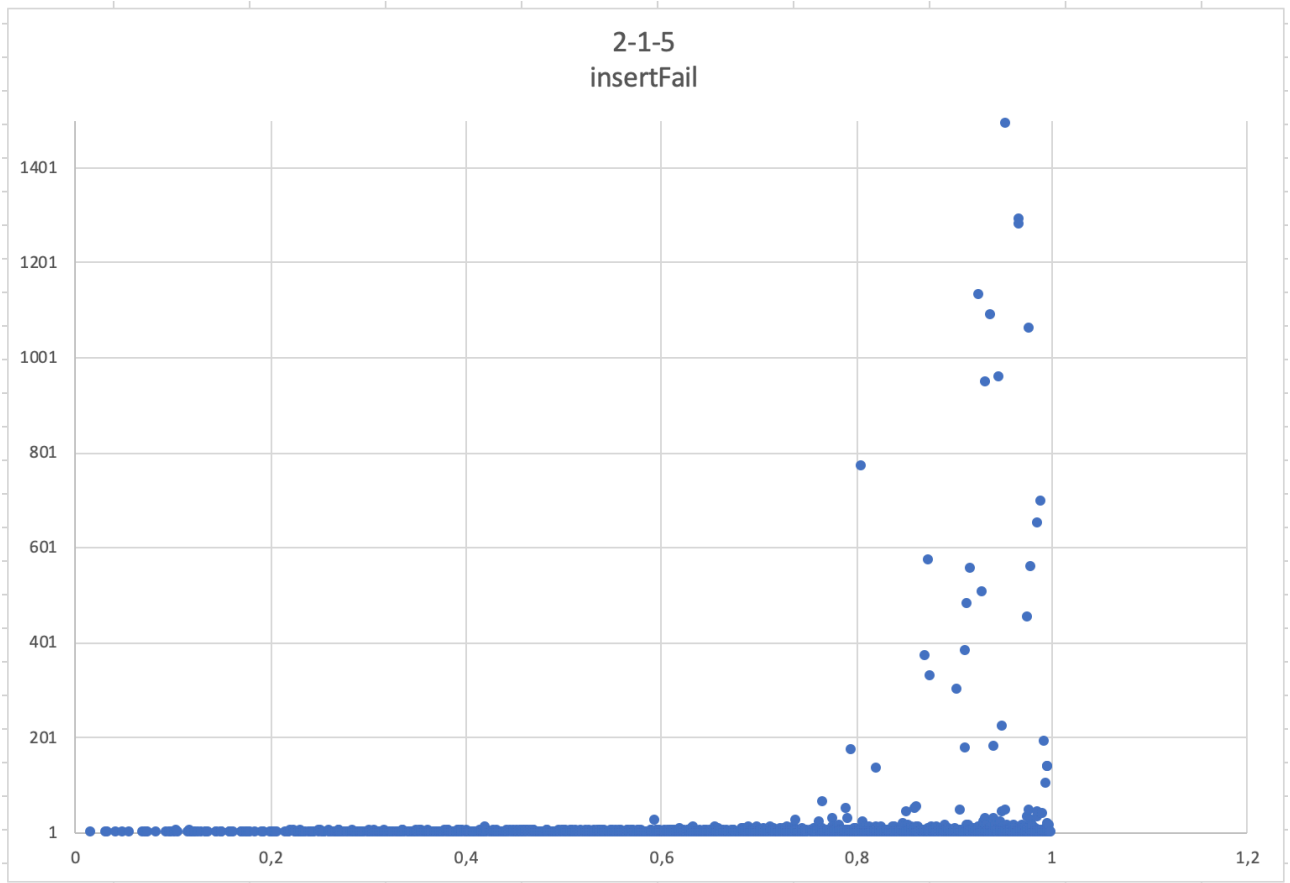

6-1-1
insertSucces
Format
9001
8001
7001
6001
5001
4001
3001
2001
1001
1
0
0,2
0,4
0,6
0,8
1
1,2

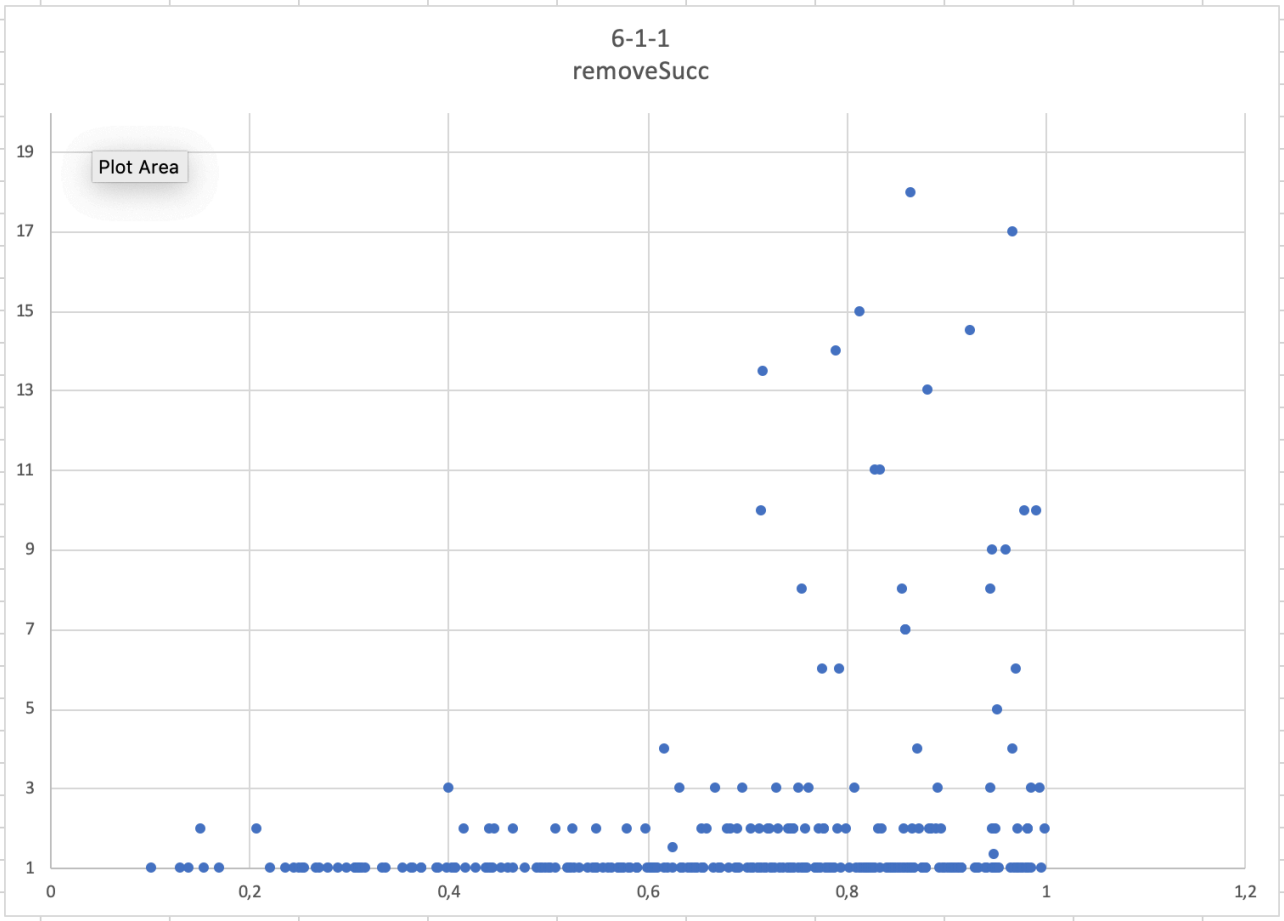

4-2-2
insertSucces
Plot Area
10001
9001
8001
7001
6001
5001
4001
3001
2001
1001
1
0
0,2
0,4
0,6
0,8
1
1,2

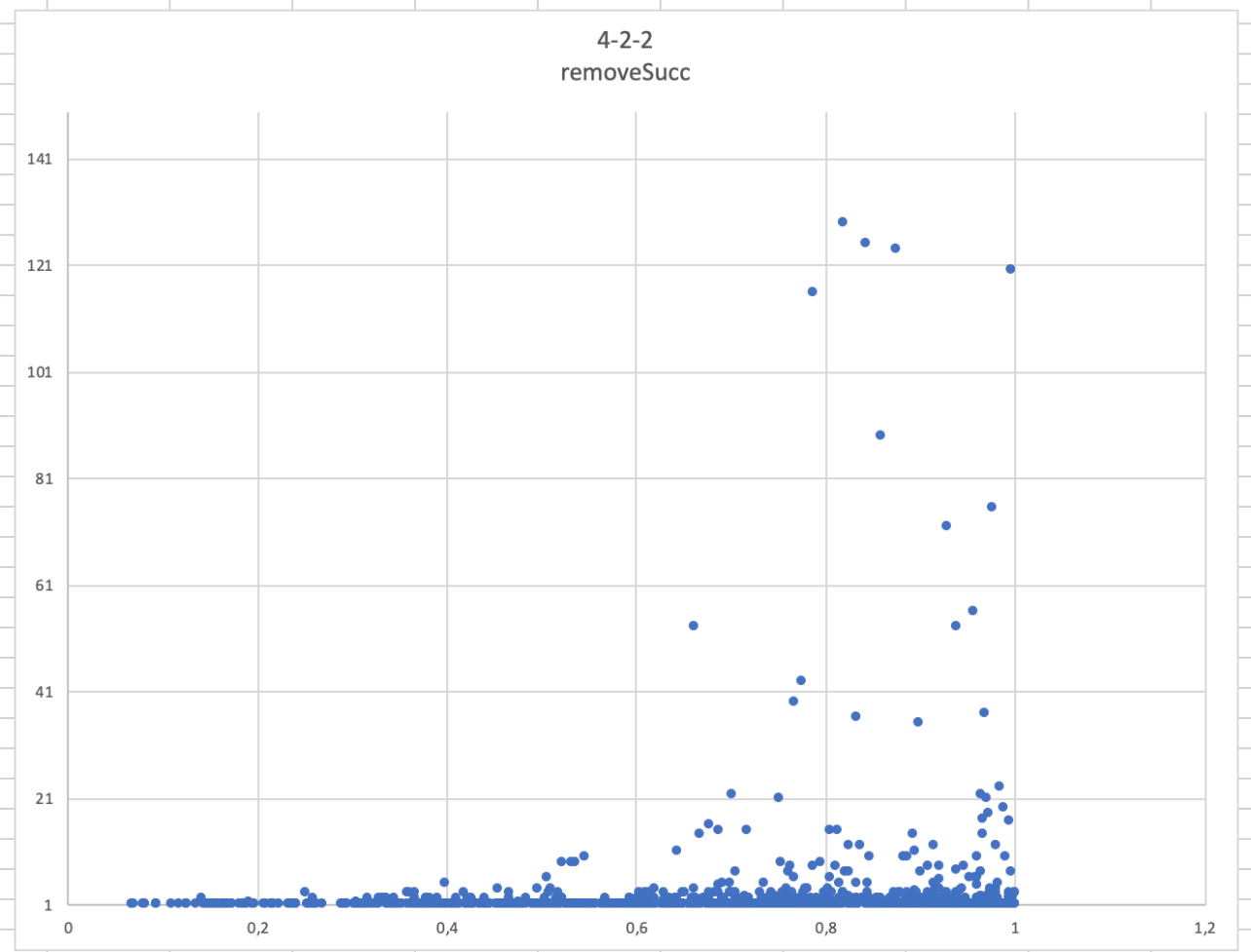

2-1-5
insertSucces
10001
9001
8001
7001
6001
5001
4001
3001
2001
1001
1
0
0,2
0,4
0,6
0,8
1
1,2
Horizontal (Value) Axis Major Gridlines

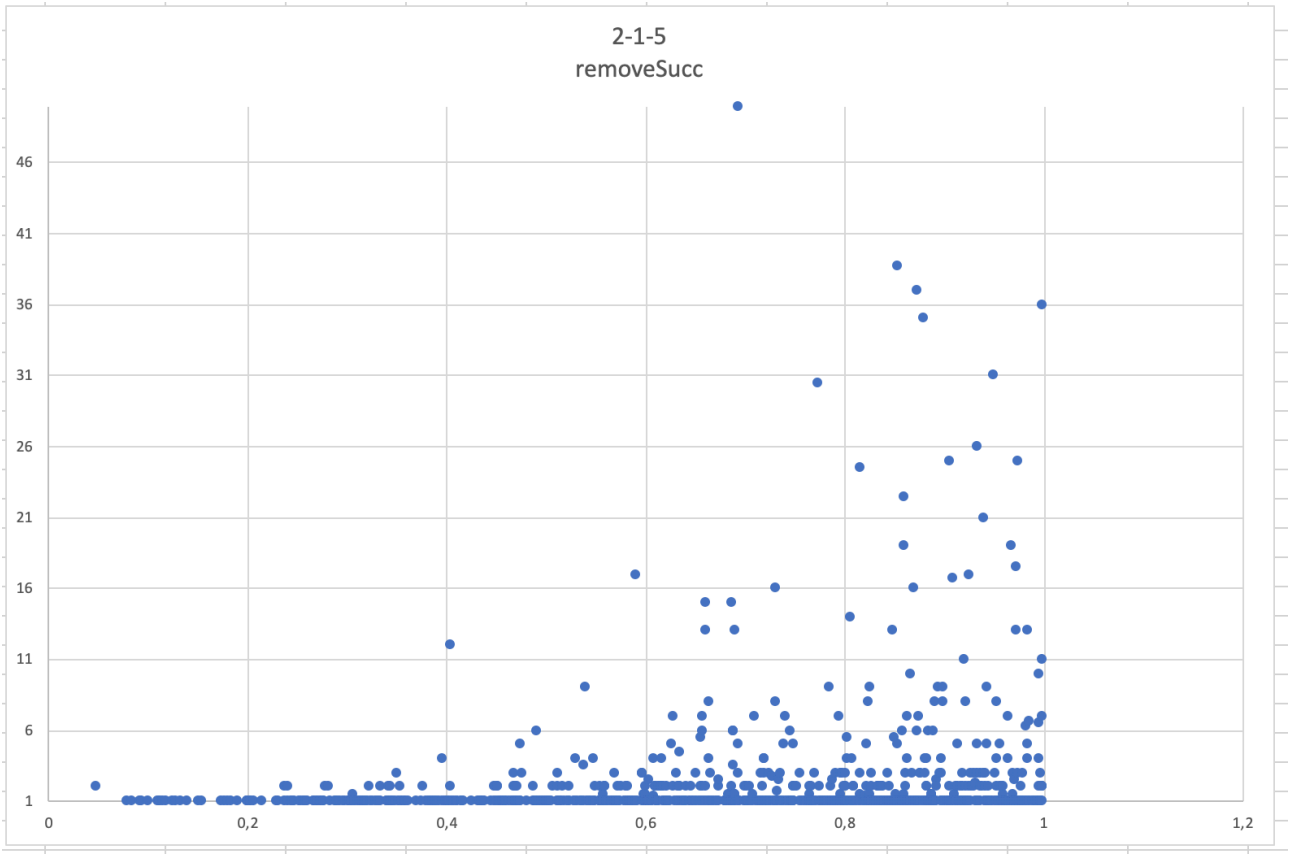

6-1-1
insertFail
71
61
51
41
31
21
11
1
0
0,2
0,4
0,6
0,8
1
1,2

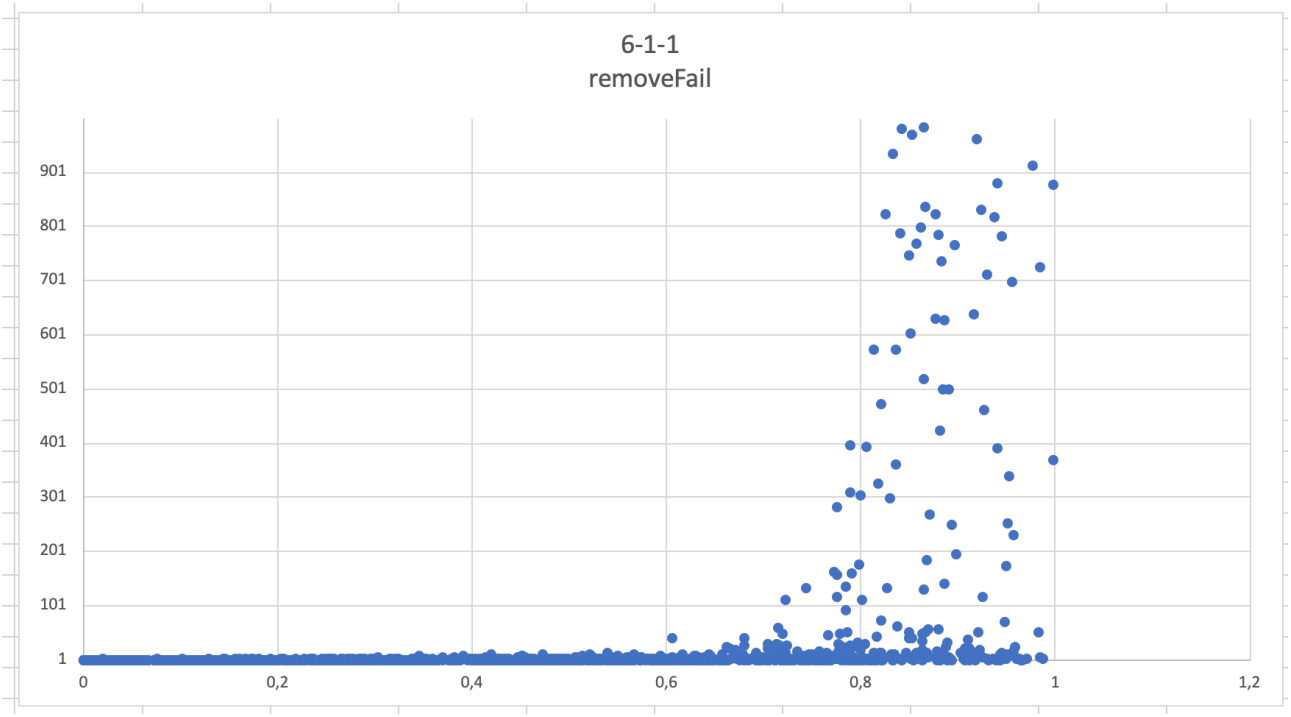

4-2-2
insertFail
1801
1601
1401
1201
1001
801
601
401
201
1
Plot Area
0
0,2
0,4
0,6
0,8
1
1,2

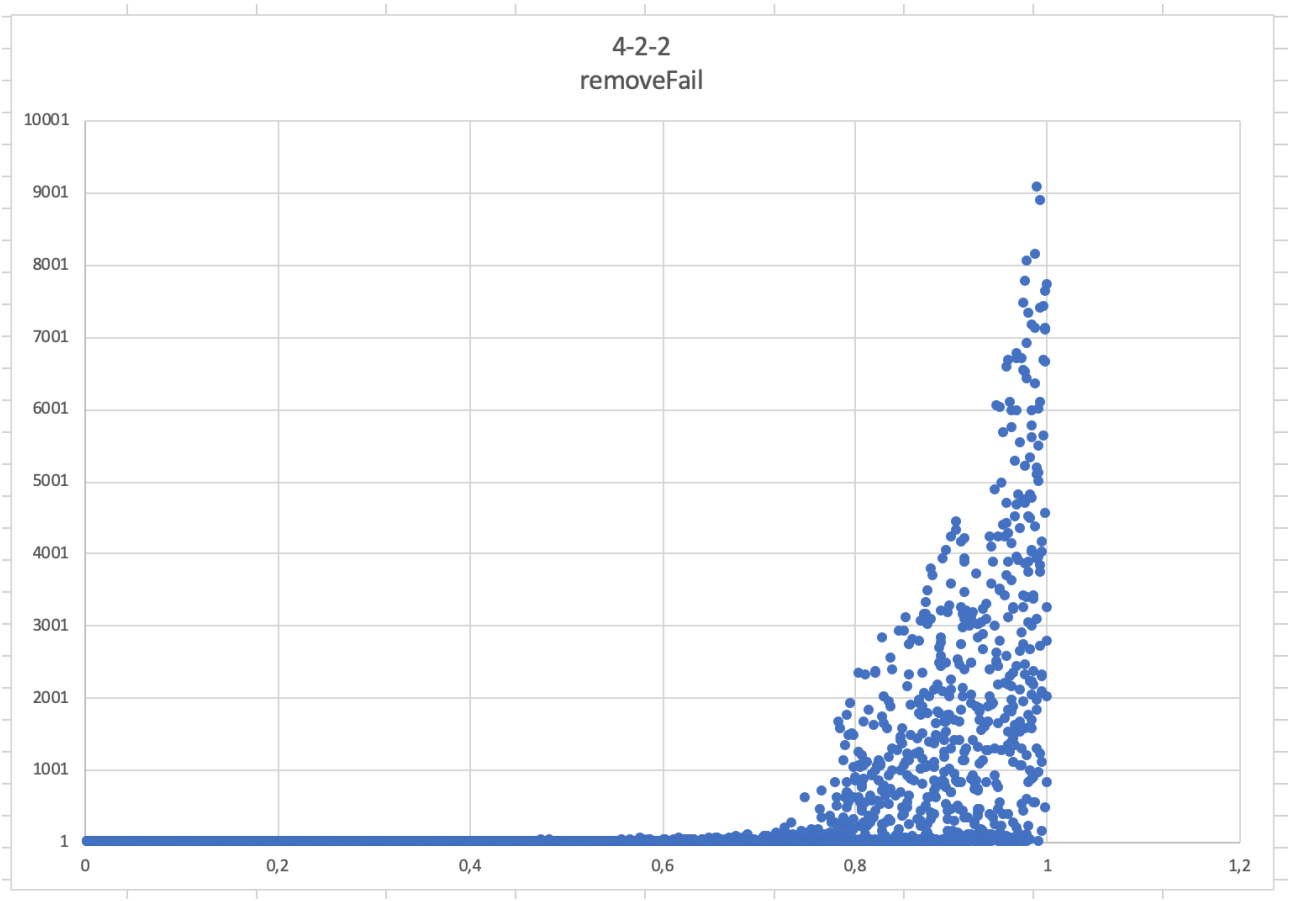

2-1-5
insertFail
1401
1201
1001
801
601
401
201
1
0
0,2
0,4
0,6
0,8
1
1,2

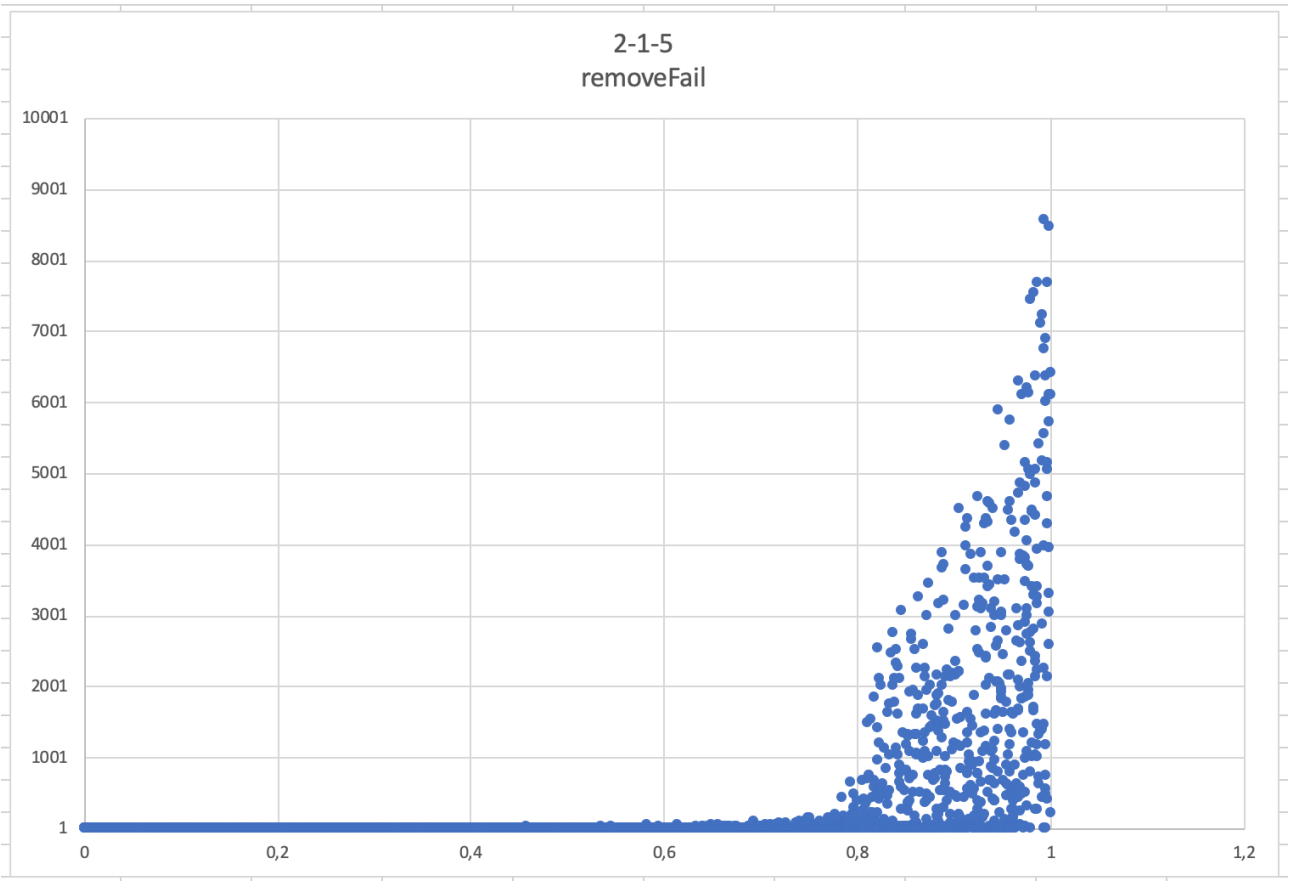

6-1-1
removeSucc
Plot Area
19
17
15
13
11
9
7
5
3
1
0
0,2
0,4
0,6
0,8
1
1,2


4-2-2
removeSucc
141
121
101
81
61
41
21
1
0
0,2
0,4
0,6
0,8
1
1,2


2-1-5
removeSucc
46
41
36
31
26
21
16
11
6
1
0
0,2
0,4
0,6
0,8
1
1,2


6-1-1
removeFail
901
801
701
601
501
401
301
201
101
1
0
0,2
0,4
0,6
0,8
1
1,2


4-2-2
removeFail
10001
9001
8001
7001
6001
5001
4001
3001
2001
1001
1
0
0,2
0,4
0,6
0,8
1
1,2


2-1-5
removeFail
10001
9001
8001
7001
6001
5001
4001
3001
2001
1001
1
0
0,2
0,4
0,6
0,8
1
1,2

To conclude,

The graphs are clearly shows that, as the load factor increases, the average number of probes is increases too. This means that our program runs slower when we don’t resize the hash table.

Furthermore, if we compare the insertion of 6-1-1, 4-2-2 and 2-1-5 we can clearly see that as our table loads up, the traverse of probe is increases.