Android Passcode Brute-Force Enumeration

Eskender (SID: 2018280304)

akd18@mails.tsinghua.edu.cn

Tsinghua University

2018-10-10

1 Problem

Android phone passcode entry area has nine dots which can be used to draw a pattern. A valid pattern consists at least of four dots (3 line segments) and has to satisfy the following criteria:

1. At least four interconnected points.
2. If the line segment between two points passes through a third point, then the third point must have been previously used.
3. The passcode must be fully connected.
4. The passcode cannot use the same dot twice.

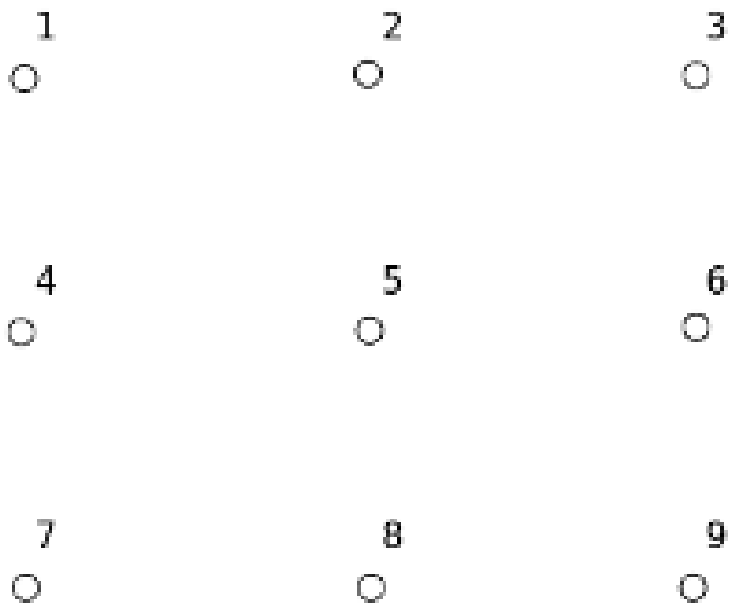

Figure 1: Passcode area.

2 Algorithm

Here a recursive algorithm is designed to enumerate all possibilites. That is to count the number patterns of a given length l, from a given starting dot, first the same problem of length $l-1$ is first solved for all valid subsequent dots coming after the original dot with an added restriction that the original dot can not be used again while solving the subproblems. (A pattern of length l contain $l+1$ dots)

For example to count all possible passcodes starting from the first dot the following subproblems must be solved first.

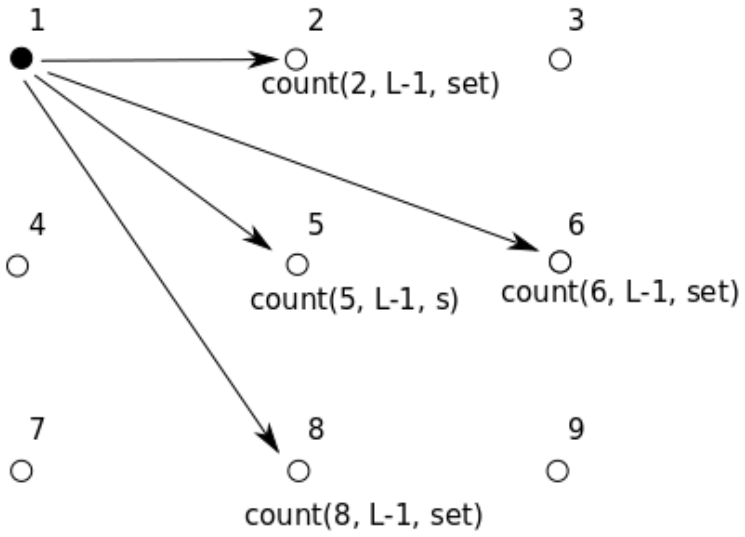

Figure 2: For patterns with initial dot=1.

The number of occupied dots increase as problems of lower level are solved. If the second dot is selected next for example, then all previous dots coming before it can not be considered again. The occupied dots are shown in black in the following figure.

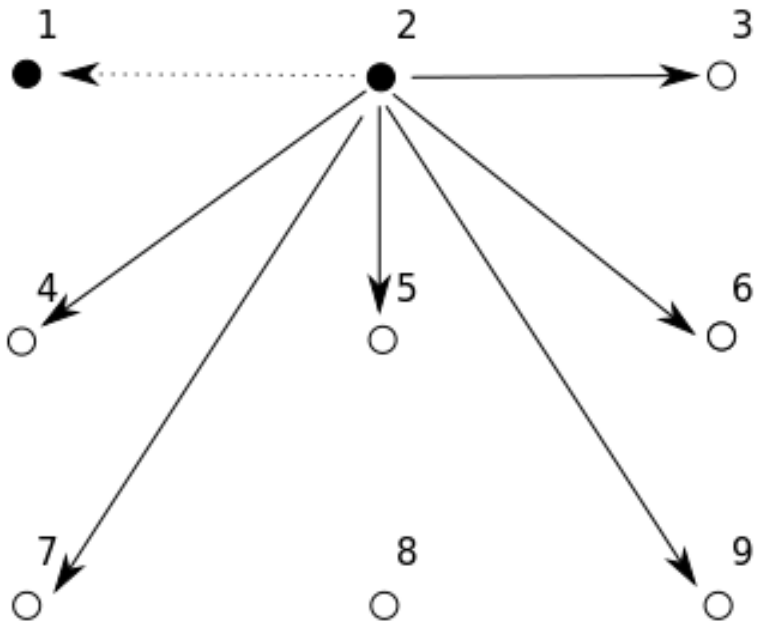

Figure 3: Number of occupied dots increase for lower level subproblems.

Stoping criteria for the recursion would be the case when $l = 1$. In this case the number of possible steps would be equal to the number of valid next dots.

When the recursion function starts the first step is to calculate the next valid dots. To handle the case of varying possible next valid dots a dictionary is maintained as shown in the implementation section.

Total number for a given length is found by adding the recursion result for each 9 dots set as the initial. But using the symmetry of the problem it is enough to run recursion for example on dots 1, 2, 5 and multiply each results according to the symmetry – recursion results for 1 and 2 are multiplied by 4.

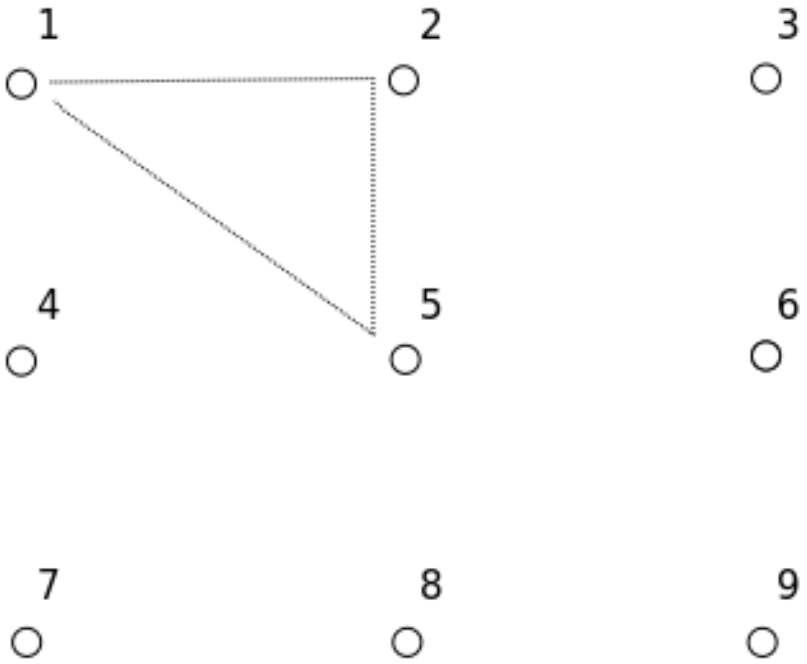

Figure 4: Symmetry of the problem.

3 Implementation

A python implementation of the alogrithm is shown below. The number of all possible patterns of more than 4 dots is calculated to be 389112 which is greater than the number for iphones which is 10000.

```
dic = { 1: [[4,7], [2,3], [5,9]],
    2: [[5,8]],
    3: [[2,1],[6,9], [5,7]],
    4: [[5,6]],
    5: [],
    6: [[5,4]],
    7: [[4,1], [5,3],[8,9]],
    8: [[5,2]],
    9: [[5,1], [8,7], [6,3]],
}
def count(init, length, occupied_dots):
    #Cross off the initial dot.
    occupied_dots.add(init)

    ## Calculate possible next dots.
    # Start from all dots.
    dots = set(range(1,10))

    # Discard elements by consulting the dictionary and current occupied list.
    for el in dic[init]:
        if not occupied_dots.__contains__(el[0]):
            dots.discard(el[1])

    # Remove all occupied dots.
    dots = dots.difference(occupied_dots)

    # Base case
    if length == 1:
        return len(dots)
    # Non-base cases
    cnt = 0
    for dot in dots:
        cnt = cnt + count(dot, length-1, occupied_dots.copy())
    return cnt

def countFromUpto(minn, maxx):
    cnt = 0
    for l in range(minn,maxx+1):
        cnt += 4*count(1, l, set([])) + 4*count(2, l, set([])) + count(5, l, set([]))

    return cnt

#Enumerate all patterns consisting of more than 4 dots (3 segments)
print(countFromUpto(3, 8))
```

4 Optimization

If all that is needed is the total number of possible passcodes of length l in the range $min\_len \leq l \leq max\_len$ rather than results for each single length, then the previous algorithm with for-loop is inefficient. This is because the algorithm is calculating possible next valid dots again and again for each different l. The following code addresses this problem by adding possible 1-length segments for each subproblem.

```
dic = { 1: [[4,7], [2,3], [5,9]],
    2: [[5,8]],
    3: [[2,1],[6,9], [5,7]],
    4: [[5,6]],
    5: [],
    6: [[5,4]],
    7: [[4,1], [5,3],[8,9]],
    8: [[5,2]],
    9: [[5,1], [8,7], [6,3]],
}
def count2(init, min_len, max_len, occupied_dots):
    #Cross off the initial dot.
    occupied_dots.add(init)

    ## Calculate possible next dots.
    # Start from all dots.
    dots = set(range(1,10))

    # Discard elements by consulting the dictionary and current occupied list.
    for el in dic[init]:
        if not occupied_dots.__contains__(el[0]):
            dots.discard(el[1])
    # Remove all occupied dots.
    dots = dots.difference(occupied_dots)

    # Base case
    if max_len == 1:
        return len(dots)
    # Non-base case
    cnt = 0
    for dot in dots:
        cnt = cnt + count2(dot, min_len-1, max_len-1, occupied_dots.copy())
    # Add the number of possible entry at every recurssion stage
    # provided min_len constraint
    if min_len <= 1:
        cnt = cnt + len(dots)
    return cnt

ans = 4*count2(1,3, 8, set([])) + 4*count2(4, 3, 8, set([])) + count2(5, 3, 8, set([]))
print(ans)
```

5 Test

A test of 100 iteration was run for both algorithms and the time results are compared. For the specific test machine the time taken by the first and second algorithm respectively are 14.77 and 9.27 seconds.