On Counting Mobile Lock Pattern

Joyanta Basak
Graduate Student, BUET

August 17, 2019

1 What Does Counting Mobile Lock Pattern Means

1.1 Why count them?

Pattern lock is a very common and widely popular security mechanism in modern smartphones. A user can set a definite pattern of nodes in a given grid as shown in fig:2. Then every time the user wakes her phone she has to draw the exact pattern on the screen to unlock. In case of wrong pattern inserted a number of times (usually 3 to 5 times), the screen freezes for a minute. This penalty might discourage most shenanigans, however, a committed larcenist can come around this problem by trying out every possible patterns given (a) The possible number of patterns are low. (b) She is very lucky. If (a) is true (b) will be more likely. Hence, I would attempt to count possible numbers of patterns and see if it is indeed a strong security feature or not.

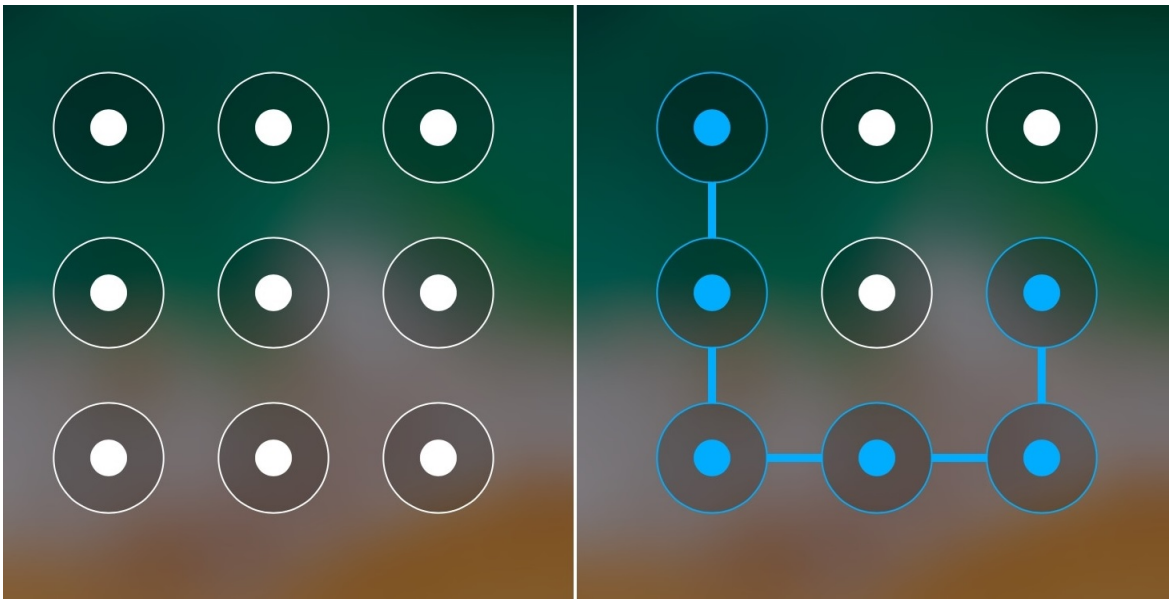

Figure 1: A sample grid (left) and a drawn pattern on the grid (right).

1.2 Formulation to a graph problem.

To count the number of possible patterns, we have to develop some method to generate a set of such unique patterns. To do that, We can formulate the *all possible pattern generation in a grid* problem into a graph problem. We can very conveniently view the grid as a graph, nodes as vertices and possible connection between nodes as edges. Then, a pattern will be a *simple path* in the graph. By definition, a path in a graph is a finite or infinite sequence of distinct edges which joins a sequence of vertices. A *simple path* is a path with no repetitive vertices. So in context of mobile pattern lock, generating every pattern is almost analogous to generating every simple path of a graph with one distinction. A path have starting and ending vertex but a pattern does not require any such orientation. So a simple path containing more that one vertex and that path traversed backwards are two different paths whereas they are the exact same pattern.

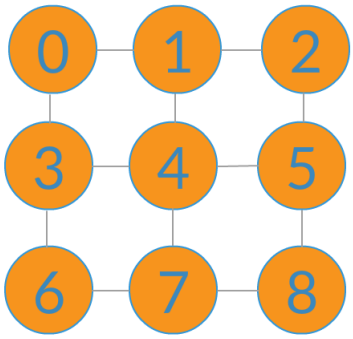

Figure 2: Graph representation of the grid with labeled vertices

2 How to Generate All Possible Simple Path in a Graph

2.1 A Systematic Generation Approach

In first attempt to count them, I have generated all possible simple paths a graph can have. With a grid of only 9 nodes it is very easy to generate all of the paths under a second with a simple algorithm. However, if the grid is much larger, it can be computationally exhaustive. For such cases, I will point out a possible solution in section **??**. A simple outline of the code is provided where *PathSearch(node nd)* procedure recursively finds possible simple paths for source node nd. For full detail, consult the complete code.

```
//Input: n = Number of Nodes, G[n][n]= Adjacency Matrix
int colour[n]=0;    // w/b = Not/present in current path
int Count=0;
class node
{
public:
```

```
    int ID;
    int wentTo[highestNeighbourCount];  // 4

    void UpdatePath()   {
        currentPath.push_back(ID);
        paths.push_back(currentPath);
        Count++;
    }

    void rollbackUpdate()   {
        for (int i=0; i< highestNeighbourCount; i++)    {
            wentTo[i]=-1;
        }
        currentPath.pop_back();
        colour[ID]=white;
    }
    bool haveWent(int id)    {
        for (int i=0; i<highestNeighbourCount; i++)        {
            if (wentTo[i]==id) return true;
        }
        return false;
    }
};

void PathSearch(node nd)
{
    int curID = nd.ID;
    colour[curID] = black;
    nd.UpdatePath();
    for(int j=0; j<n; j++)    {
        if(colour[j]==white && G[curID][j]==1 && !nd.haveWent(j))        {
            PathSearch(Node[j]);
        }
    }
    nd.rollbackUpdate();

}
```

Simple Path Count for each vertex source in 3×3 grid graph									
Label	0	1	2	3	4	5	6	7	8
Count	79	69	79	69	61	69	79	69	79

Table 1

2.2 An Insight

Table 1 shows that, for 9 nodes in a 3×3 grid graph, the number of simple paths starting from a vertex is same for some vertices. In fact, there is only 3 such numbers. It is because grid graphs are symmetric and the vertices can be classified into specific types based on

their neighbourhood relation. Our 3×3 grid has vertices with three different neighbourhood relation –

- Type 1 (Corner Node): Connected to two edge node.
- Type 2 (Edge Node): Connected to two corner node and one inner node.
- Type 3 (Inner Node): Connected to four edge node.

If we expand the grid to 4×4 then we will also find three types of nodes but with different neighbourhood properties. Further expansion into 5×5 grid produces six types of nodes for its 25 nodes. So it would be more efficient if we employed the algorithm in section 2.1 for a single instance of each type and then multiply that with total instances of respective types. Their sum would be the total count of simple paths.

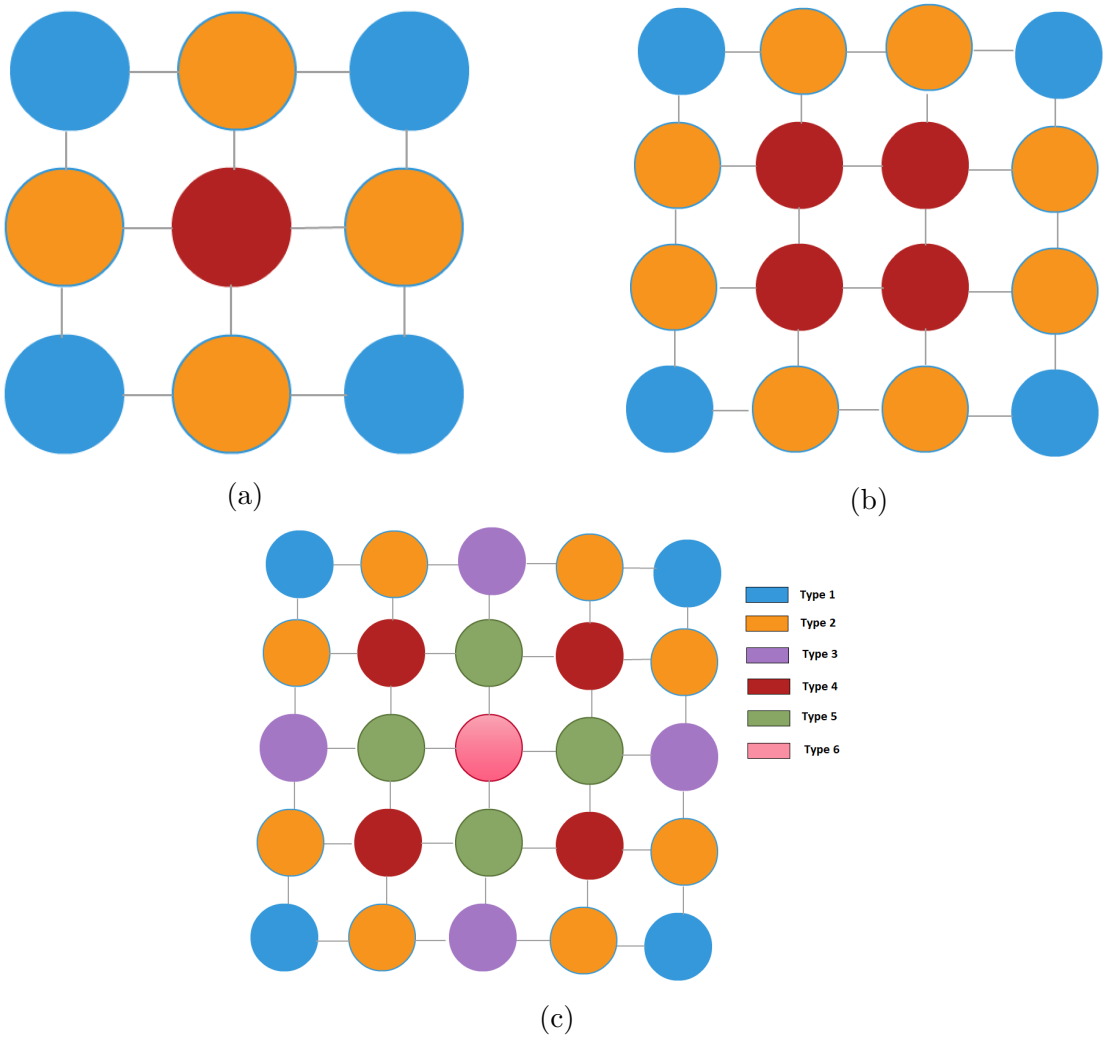

Figure 3: $3 \times 3, 4 \times 4, 5 \times 5$ grids with each type of vertex coloured alike

Paths that originated from vertices of same type can be also produced in a following fashion –

- Label the nodes.
- Produce the paths originated from any arbitrary instance of the target vertex type.
- Make a copy of the grid. Now rotate the copied grid with a combination of horizontal and vertical flips to make the selected vertex of the original grid and target vertex of copied grid to be in same position when both grids are superimposed.
- Relabel the nodes in the path replacing the labels with their respective superimposed node label.

	3×3	4×4	5×5
type 1	79	2111	153745
type 2	69	1728	124190
type 3	61	1561	113087
type 4	-	-	110427
type 5	-	-	110175
type 6	-	-	117161

Table 2: Number of paths originating from each type of vertex for three grids shown in fig 3

2.3 Recursive Vertex Elimination

Table 2 gives us an idea how number of paths can explode with increase of grid dimension. For sufficiently large grid, counting paths can be a time consuming process. Interestingly, it can be speed up if path count for smaller grids are known beforehand. Let us assume, *'a'* is a arbitrary vertex in graph G. Procedure *Count(G,a)* gives the number of simple paths originating from a and $N_i(a)$ gives the i-th neighbour of G where $i = \{x | x \text{ is an integer and x } \in 1, 2, ...Degree(a)\}$. If eliminating a from G results in $G' = G - \{a\}$ then,

$$Count(G, 0) = 1 + \sum_i Count(G', N_i) \tag{1}$$

The idea behind this equation is very simple. Paths originating from vertex a in G must go through its neighbour and must not come back to a again. If we place a before paths generated by neighbours of a in $G' = G - \{a\}$, that would produce all paths originating from a except the trivial path consisting only a itself. Now, $Count(G', N_i)$ can be expanded similarly by eliminating N_i from G'. This process can be continued until we reduce the sub-graph into a sub-graph with known path counts for its nodes.

3 Feasibility of Using Mobile Lock Patterns

From our experimentation, we found 653 simple paths in a 3×3 grid graph. As per discussed in section 1.2, the total number of patterns would be 331 where 9 patterns are of length 1, 12 patterns are of length 2 and rest 310 patterns are of length 3 or more. So a patient person trying one pattern per minute can break this lock in little more than 5 hours at her worst luck. Employing a 4 or 5×5 would make such pattern lock mechanism quite impossible to break under reasonable amount of time. However, adding bigger grid might not be a favourable design considering sreen size of many mobiles. A simple modification of adding diagonal edges to each node in 3×3 grid can alleviate the problem. Such modification boosts the number to 5128 possible patterns of length 3 or more. It would take 85 hours and 28 minutes to break the lock with one attempt per minute. I would not dare certify this as secure with the world full of persistent people.