Chapter 1

Fraud Detection - Dense Subgraph Detection

Contributed by Tong Zhao

1.1 Introduction

Fraud behaviors can be spotted everywhere on online applications such as social networks where the behavior data can be represented as large bipartite graphs which consist of links between followers and followees. Detecting the fraudsters such as bot followers tend to be an unsupervised problem as the size of such social network graphs are huge and labeling even a small portion of the graph will take too much human effort. Luckily, fraudulent actions such as fake followers usually result with creating subgraphs with unexpected high density. For example, as a large number of follower buyers buy followers from one major follower seller, these follower buyers together with the bot followers controlled by the seller will form a subgraph with high density. Therefore, many existing detection methods [17, 29, 27] estimate the suspiciousness of users by identifying whether they are within a dense subgraph.

1.2 The Problem as a Graph

Here we define the definitions of density for graphs according to [6, 14, 20]. Let $G = (V, E)$ be a undirected graph with vertices $V$ and edges $E \subseteq V \times V$. $E(V)$ stands for the set of edges induced by $V$, that is

$$E(V) = \{(i, j) \in E : i \in V, j \in V\}$$

Then the density of subgraph induced by $S \subseteq V$ can be defined as

$$d(S) = \frac{|E(S)|}{|S|}$$

Note that $2d(S)$ is actually the average degree of the subgraph induced by $S$. The Densest Subgraph problem can be defined as

$$DS(G) = \max_{S \subseteq V} \{d(S)\}$$
Dense Subgraph Detection

For directed graphs. Let $G = (V, E)$ be a directed graph with vertices $V$ and edges $E \subseteq V \times V$. $E(S, T)$ stands for the set of edges from vertices in $S \subseteq V$ to vertices in $T \subseteq V$, that is

$$E(S, T) = \{(i, j) \in E : i \in S, j \in T\}$$

Then the density of subgraph induced by $S, T \subseteq V$ can be defined as

$$d(S, T) = \frac{|E(S, T)|}{\sqrt{|S||T|}}$$

The Densest Subgraph problem can be defined as

$$DS(G) = \max_{S, T \subseteq V} \{d(S, T)\}$$

1.3 Some Realistic Data Sets

The data sets that this application encounter can come from social networks (botnet followers.) Thus the graph size can be huge. For example, the Twitter follower-followee data set used in Fraudar [17] contains 41.7 million nodes with 1.47 billion edges. Similar data sets for social networks such as Twitter can be found on SNAP [23] or other platforms. It is intuitive that the size of graphs for such problem in real industry will be growing continuously. Hence it is important for the algorithms to have a linear or near linear run-time or be able to parallelize.

1.4 Dense Subgraph Detection-A Key Graph Kernel

Multiple algorithms exists for detecting the dense subgraphs. One commonly used algorithm is proposed by Charikar in 2000 [6], which is an approximation algorithm by greedy approach. Although Charikar’s algorithm sacrificed quality of the result subgraph for much better time complexity, this algorithm still has a provable 2-approximation guarantee [21]. That is, if the densest existing subgraph $S'$ has edge density of $d(S') = \lambda$, the result subgraph $S$ of Charikar’s algorithm will have edge density of $d(S) \geq \lambda/2$.

The greedy idea of Charikar’s algorithm is to remove the vertex that is least likely in the densest subgraph at each step according to certain rule. In the case of undirected graph, the rule can obviously be to remove the vertex with lowest degree. Then Charikar’s algorithm can be described as following [6].

1: **procedure** DENSEST-SUBGRAPH($G$)
2: **Input:** Undirected graph $G = (V, E)$.
3: **Output:** Dense sugraph $S$ of $G$.
4: $n \leftarrow |V|$
5: $G_n \leftarrow G$
6: **for** $k \leftarrow n$ **down to** 1 **do**
7: $v \leftarrow$ the vertex with smallest degree in $G_k$
8: Delete all edges incident on $v$.
9: Delete all vertices with 0 degree.
10: $G_{k-1} \leftarrow$ the remaining of graph $G_k$
11: **return** The subgraph with maximum density among $G_1, G_2, \ldots, G_n$. 
A detailed proof for this algorithm to achieve a 2-approximation can be found in [21].

For directed graphs, Khuller and Saha proposed a approximation algorithm based on Charikar’s algorithm in 2009 [21] that utilized the same greedy idea. The key point of this algorithm for directed graphs is to first duplicate all the vertices and construct a bipartite graph such that one copy of the vertices have only outgoing edges and the other copy of the vertices have only incoming edges. Then the algorithm can be described as following. [21]

1: procedure DENSEST-SUBGRAPH-DIRECTED(G)
2: Input: Directed graph $G = (V,E)$.
3: Output: Dense sugraph $S$ of $G$.
4: $n \leftarrow |V|$
5: $G_{2n} \leftarrow G$
6: for $k \leftarrow 2n$ down to 1 do
7: $v \leftarrow$ the vertex with smallest degree in $G_k$
8: if $v$ has outgoing edges then
9:   Delete all the outgoing edges incident on $v$.
10: else
11:   Delete all the incoming edges incident on $v$.
12:   Delete all vertices with 0 degree.
13: $G_{k-1} \leftarrow$ the remaining of graph $G_k$
14: return The subgraph with maximum density among $G_1, G_2, \ldots, G_{2n}$.

A detailed proof for this algorithm to achieve a 2-approximation can also be found in [21].

Both algorithms has time complexity of $O(|V| \log |V|)$ and space complexity of $O(|V|^2|E|)$. To evaluate the performance of these two algorithms, both density on the result subgraph and runtime can be used.

1.5 Prior and Related Work

1.5.1 Dense Subgraph Problem

The history for Dense Subgraph problem for static graphs has a rather short history, as the best exact solution was proposed by Goldberg in 1984 [14] and the best approximation algorithm so far were proposed by Charikar in 2000 [6] and Khuller and Saha in 2009 [21] for undirected and directed graphs.

Goldberg’s solution works only for undirected graph. His idea was to interestingly transfer this dense subgraph problem into a well-know min-cut problem by adding two vertices $s$ and $t$. Both $s$ and $t$ are connected with all the vertices in graph $G = (V,E)$. For each vertex $v_i \in V$, edge $(s,v_i)$ has edge weight that is the same as the degree of $v_i$ and edge $(v_i,t)$ has edge weight of a positive constant $c$. All the edges in the original graph has an edge weight of 1. Then buy performing a min-cut call that splits $s$ and $t$ into two subgraphs, one of the subgraphs would be the densest subgraph of $G$ after removing $s$ or $t$.

Since min-cut problem can be solved using the parametric max-flow algorithm, this algorithm has a $O(|V||E|)$ time complexity. Thus Goldberg’s algorithm is not scalable for large graphs; faster approximation algorithms are more preferred in industry situations.

In the year of 2000, Charikar [6] proposed an algorithm for detecting the dense subgraph by a greedy approximation algorithm, which we talked in the last section. In 2009, Khuller, et al. [21] further extended Charikar’s algorithm to directed graphs and proved both algorithms to be 2-approximation, which are so far the best algorithms with fast run-time and theoretically guaranteed
acceptable results.

1.5.2 Fraud Detection

Data-driven approaches have received great success in the field of fraud detection [22, 17]: most methods indentify unexpected dense regions of the bipartite graph, as creating fake reviews/ratings unavoidably generates edges in the graph [18, 8, 25, 38, 31, 3].

Unexpected spectral patterns. Global graph mining methods model the entire graph to find fraud based on singular value decomposition (SVD), latent factor models, and belief propagation (BP). SPOKEN [29] considered the “spokes” pattern produced by pairs of eigenvectors of graphs, and was later generalized for fraud detection. FBOX [30] focuses on mini-scale attacks missed by spectral techniques. BP has been used for fraud classification on eBay [27], link farming on Twitter [11], and fake software review detection [1].

Unexpected high density in subgraphs. Finding dense subgraphs has been studied from a wide array of perspectives such as mining frequent subgraph patterns [24, 39], detecting communities [12, 7, 28], and finding quasi-cliques [13, 35, 10, 32]. Charikar [6] shows that average degree of subgraph can be maximized with approximation guarantees. Tsourakakis, et al. [36] optimize the density of adjacency matrix of subgraph with quality guarantees. Hooi, et al. [17] adopt both node degree and edge density to model suspiciousness of subgraph and further increases accuracy in binary adjacency matrix of bipartite graph.

Unexpected high density in time-series. Typically there are two kinds of representation on density in time-series. One is dense subgraphs in evolving graphs [9]. COPYCATCH [4] uses local search heuristics to find Δt-bipartite cores in which users consistently likes the same Facebook pages at the same short time interval. The other is dense subtensors in high-order tensors of a time dimension [26, 19, 33] or tensor streams [34]. [37, 15] consider fraud detection methods that are robust to camouflage attacks. Hooi, et al. [16] adopt a Bayesian model to find early spikes of outlier ratings in time series. All these methods focus on the time-series domain, observing changes in the behavior from system access logs rather than graph data.

1.6 A Sequential Algorithm

To implement the algorithms we talked in Section 1.4, we use Python 3 with machine learning libraries numpy\(^1\) and scipy\(^2\). The implementation stores all graphs in sparse matrix format which is provide by scipy, so graph libraries were not used in the basic implementation.

1.7 A Reference Sequential Implementation

The source codes of Fraudar [17] is publicly available online\(^3\). With minor modifications on the density calculation functions, the codes of Fraudar can become the exact Python 3 implementation of the algorithm for directed graphs we talked in Section 1.4.

In order to get the algorithm’s best performance, a priority tree data structure must be used to store all the degrees of vertices so that retrieving the vertex with minimum degree and updating the degree of its neighbors can be done in logarithmic time. Following is the Python implementation of priority tree in the source codes of Fraudar [17] with minor modifications.

\(^{1}\)http://www.numpy.org
\(^{2}\)https://www.scipy.org
\(^{3}\)https://www.andrew.cmu.edu/user/bhooi/projects/fraudar/index.html
import math
class MinTree:
    def __init__(self, degrees):
        self.height = int(math.ceil(math.log(len(degrees), 2)))
        self.numLeaves = 2 ** self.height
        self.numBranches = self.numLeaves - 1
        self.n = self.numBranches + self.numLeaves
        self.nodes = [float('inf')] * self.n
        for i in range(len(degrees)):
            self.nodes[self.numBranches + i] = degrees[i]
        for i in reversed(range(self.numBranches)):
            self.nodes[i] = min(self.nodes[2 * i + 1], self.nodes[2 * i + 2])

    def getMin(self):
        cur = 0
        for i in range(self.height):
            if self.nodes[2 * cur + 1] <= self.nodes[2 * cur + 2]:
                cur = (2 * cur + 1)
            else:
                cur = (2 * cur + 2)
        return (cur - self.numBranches, self.nodes[cur])

    def changeVal(self, idx, delta):
        cur = self.numBranches + idx
        self.nodes[cur] += delta
        for i in range(self.height):
            cur = (cur - 1) // 2
            nextParent = min(self.nodes[2 * cur + 1], self.nodes[2 * cur + 2])
            if self.nodes[cur] == nextParent:
                break
            self.nodes[cur] = nextParent

    def dump(self):
        cur = 0
        for i in range(self.height + 1):
            for j in range(2 ** i):
                cur += 1

1.8 Sequential Scaling Results

All experiments are run on a 2.7 GHz Intel Core i7 Macbook Pro, 16 GB RAM, running OS X 10.14.1. The graphs used in experiments are generated by a public available Python graph generator\(^4\), which generates bipartite graphs according to given number of vertices and average degree. The vertices of generated graphs have power-law degree distributions as shown in Figure 1.1.

In the experiments, average degree is fixed as 20 and the results are shown in Table 1.1 and

\(^4\)https://github.com/cooperative-computing-lab/graph-benchmark
plotted in a log-log plot in Figure 1.2. From the results and the plot it is noticeable that this algorithm has a almost linear performance.

1.9 An Enhanced Algorithm

Two ways exist for enhancing the algorithms for dense subgraph detection. The first way is to make the algorithm possible to run on dynamic graphs. As all kinds of platforms have a large number of data coming in every single day, it is crucial for such graph algorithms to be able to run on dynamic graphs. Bhattacharya, et al. [5] proposed a space- and time-efficient algorithm for maintaining dense subgraphs on the one-pass dynamic streams in 2015, which is based on \((\alpha, D, L)\)-decomposition and very complicate. The second way is to make the current algorithms to be able to run on much larger datasets in a acceptable time. In the following few sections, we will focus on the second kind of enhancement.

The current Charikar’s algorithm implementation loads the whole graph into RAM at the beginning. When the graph gets extremely large, it is not possible for almost all computer to do so. Hence the idea is to load only the vertices with their degree information into RAM, because the number of edges in usually far larger than the number of vertices in most of the graphs. However, the negative result of doing so is that the algorithm will have to go through the whole edge list, which is stored in the disk, every time it needs any edge information or the neighbors of one vertex. Therefore, implementing Charikar’s algorithm in this way will result with a \(O(n)\) pass algorithm, where \(n\) is the number of vertices in the graph and 1 pass means the algorithm needs to go through the whole graph in disk one time. [2] Apparently, when the size of the graph is large, going through
### Dense Subgraph Detection

<table>
<thead>
<tr>
<th>Number of vertices</th>
<th>Running time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10}$</td>
<td>0.285</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>0.533</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>0.778</td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>1.394</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>2.860</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>5.992</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>13.259</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>26.932</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>67.042</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>153.725</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>351.798</td>
</tr>
<tr>
<td>$2^{21}$</td>
<td>668.633</td>
</tr>
</tbody>
</table>

Table 1.1: Running time results for the sequential algorithm.

![Graph showing running time results for the sequential algorithm.](image)

Figure 1.2: Running time results for the sequential algorithm.
the whole graph, which is stored in disk, \( n \) times will take an unacceptably long time due to the I/O limitation. Bahmani, et al. [2] proposed a modified version of Charikar’s algorithm, which is able to maintain a \( 2(1 + \epsilon) \) approximation bound as well as only using \( O(\log_{1+\epsilon} n) \) passes for any \( \epsilon > 0 \). The idea of this algorithm is to remove a set of vertices at each iteration instead of only removing only one vertex. Then the algorithm can be described as following [2].

1: \textbf{procedure} DENSEST-SUBGRAPH(\( G \))
2: \textbf{Input:} Undirected graph \( G = (V, E) \).
3: \textbf{Output:} Dense subgraph \( S \) of \( G \).
4: \( S, V' \leftarrow V \)
5: \textbf{while} \( V' \neq \emptyset \) \textbf{do}
6: \( A(V') \leftarrow \{ i \in V' | \deg_S(i) \leq 2(1 + \epsilon)d(V') \} \)
7: \( V' \leftarrow V' \setminus A(V') \)
8: \textbf{if} \( d(V') > d(S) \) \textbf{then}
9: \( S \leftarrow V' \)
10: \textbf{return} \( S \)

This algorithm can also be modified to be used on directed graphs like Charikar’s algorithm.

1.10 A Reference Enhanced Implementation

To implement the algorithm we talked in Section 1.9, we use Python 3 without any outer libraries. In the implementation, \texttt{Counter} is used to store all the vertices with their degree information and \texttt{multiprocessing} is used for parallelization.

1.11 Enhanced Scaling Results

All experiments are run on a 2.7 GHz Intel Core i7 Macbook Pro, 16Gb RAM, running OS X 10.14.1. First of all, we tested the enhanced algorithm with a huge Twitter dataset that contains 41.7 million users (vertices) and 1.47 billion follows (edges). The graph is stored as an edge list in a .txt file which is over 25Gb. For a laptop with 16Gb RAM, it is usually impossible to process such a large graph as the RAM cannot even store the whole graph. We first tried the sequential Charikar’s algorithm on this graph and the program was killed by the system within a few minutes. However, the enhanced algorithm, although very slow, finished the algorithm within three days. The hyperparameter \( \epsilon \) was set as 0.5 in this experiment, which resulted with 44 passes of the whole dataset.

Experiments are also done on the same graphs as the sequential algorithm did, and the results are shown in Table 1.2. However, it is not reasonable to compare this set of results with the results in Table 1.1 because this enhanced algorithm is designed for graphs that are too large to fit in RAM. This enhanced algorithm sacrificed performance for space complexity by not storing any edge information in the RAM. Therefore, it is reasonable for it to have the slow results shown in Table 1.2, which fits my anticipation.

1.12 Conclusion

In conclusion, this chapter discussed about several algorithms of dense subgraph detection for fraud detection. One important enhancement for this kernel is the dense subgraph detection algorithm for dynamic graphs that I mentioned in Section 1.9, which was not implemented due to time
Table 1.2: Running time results for the enhanced algorithm.

<table>
<thead>
<tr>
<th>Number of vertices</th>
<th>Running time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10}$</td>
<td>5.014</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>21.719</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>109.217</td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>311.860</td>
</tr>
</tbody>
</table>

limit. Therefore, any future work of this chapter should start with the algorithm proposed by Bhattacharya, et al. [5].

1.13 Response to Reviews

I made modifications according to each of the advises, specifically:

- Added more information and explain in the introduction section.
- Modified the first algorithm.
- Deleted some irrelevant sentences.
- Added one more paragraph introducing more algorithms in section 1.5.1.
- Corrected a lot of misspellings.
Bibliography


Dense Subgraph Detection


[24] Chao Liu, Xifeng Yan, Hwanjo Yu, Jiawei Han, and Philip S Yu. Mining behavior graphs for “backtrace” of noncrashing bugs. In SDM, pages 286–297, 2005.


Dense Subgraph Detection


