Chapter 1

Introduction

1.1 Motivation

The problem of identifying significant subgraphs is widely studied in many applications as social network, chemo-informatics, bioinformatics. These application deals with data in form of a graph. Research in finding various properties of these graphs and finding patterns on these graphs is highly encouraged. Many of these applications rely on exploiting significant substructures as paths [1], trees [2], cliques [3], or subgraphs [4–7].

There exists several techniques for finding statistically significant subgraphs. One such technique is to find those subgraphs, whose frequency in original graph exceeds a particular threshold. These frequent subgraphs are considered as statistically significant. But these frequent subgraphs might not always be statistically significant [4]. Thus, a new technique was proposed to find statistically significant subgraphs by transforming graphs into feature vectors. To mine significant patterns with low p-value was proposed in [8].

There are many applications where each vertex of a graph can be assigned a label. Vertex labels can be continuous [9, 10] or discrete [11, 12]. In this thesis we focus on the graphs having discrete vertex labels. Discrete labeled graphs are graphs where each vertex of a graph is labeled and labels come from some finite set. For social graphs, vertices are labeled as user, page, group etc. Edges among those vertices
represents relationship. For e.g. in social graph edges might be there from user to user or from user to page.

A new algorithm for mining statistically significant connected subgraphs in a vertex labeled graph was given by Arora et al. [13]. In this approach chi-square statistic [14] is used to quantify statistical significance. The algorithm first reduces the graph to a super-graph. If the super-graph is small enough then all the connected subgraphs of this super-graph are enumerated and most significant subgraphs are given. If the super-graph is large, then reduction algorithm is applied on the super-graph till it gets reduced to small size such that trivial algorithm to find all connected subgraphs can be applied.

In this thesis, we identified the steps of algorithm given by Arora et al. [13] for the graphs having discrete vertex labels that are parallelizable. We used GraphX library of Apache Spark [15, 16] to implement our parallel algorithm. Apache Spark is a distributed cluster computing framework. Size of the graphs of aforementioned applications is substantially large. Practically to mine statistically significant subgraphs, sequential algorithm is slow. The main motivation is to mine statistically significant subgraphs efficiently such that a practical solution exists.

1.2 Contribution

In this work, we setup the Apache Spark cluster consisting of 3 worker nodes each having 8 Vcores and 16 GB of memory. We also setup the Hadoop Filesystem (HDFS) consisting of 3 datanodes. We developed and implemented the parallel algorithm for statistically significant subgraph mining. We have shown comparison of parallel and sequential algorithm.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows.

Chapter 2: This chapter explains the algorithm given by Arora et al. [13] in detail.
We will also explain Apache spark architecture and challenges faced during the implementation in this chapter.

**Chapter 3:** In this chapter, we discussed the parallel algorithm for statistical significant subgraph mining.

**Chapter 4:** In this chapter, we have shown our experimental results in detail. Experiments were performed on data sets from SNAP [17] and on random graphs.

**Chapter 5:** We conclude the thesis and have discussed some future work.
Chapter 2

Background and Related Work

In this chapter we describe what statistical significance means. We also describe the sequential algorithm proposed by Arora et al. [13] in detail. Later we describe the Apache Spark GraphX framework [16] and the challenges to implement the parallel algorithm on this framework.

2.1 Statistical Significance

We now describe what null hypothesis, p-value and statistical significance means.

**Null Hypothesis** : Null hypothesis states that the occurrence of any observed events is purely due to chance.

**p-value** : p-value is a metric, which is used to test the null hypothesis. Some threshold value is selected before experiment, called as significance level. If p-value is less than or equal to significance level, then null hypothesis can be rejected.

**Statistical Significance** : Statistical significance is used as a measure to claim that outcome of an experiment is due to some external factor and not due to randomness. It is shown by Arora et al. [13] that if occurrence of a subgraph is dependent on some factor then it can be considered as statistically significant. p-value is used to measure the statistical significance, but the calculation of p-value is not practical as there are exponential number of outcomes [13]. Thus, chi-square
Algorithm 1 Construction of Super-graph for Discrete Labels

Require: Input Graph $G = (V, E)$
Ensure: Super-graph $G_s = (V_s, E_s)$
1: Make a copy of original graph $G_c(V_c, E_c) \leftarrow G(V, E)$
2: for all edges $e = (u, v) \in E_c$ do
3:   if $e$ is non-contracting then
4:     Delete $e$ from Graph $G_c$
5:   end if
6: end for
7: $V_s \leftarrow$ Run BFS and find connected components of Graph $G_c$
8: for all edges $e = (u, v) \in E$ do
9:   if $e$ is non-contracting then
10:      if $u$ belong to super-vertex $u_s$ and $v$ belong to super-vertex $v_s$ then
11:         Add super-edge $e_s = (u_s, v_s)$ to $G_s$
12:      end if
13:   end if
14: end for
15: return $G_s(V_s, E_s)$

value is used as a measure for statistical significance [18]. Chi-square is denoted as $X^2$. Lower the p-value, higher is the $X^2$.

2.2 Sequential Algorithm - Discrete labels

The algorithm takes graph $G(V, E)$ having discrete vertex label as a input. It constructs a super-graph by first removing non-contracting edges i.e. edges having different labelled vertices at their end. It then runs Breadth-First-Search (BFS) on this new graph to find the connected components. Each component in this graph is treated as super-vertex. It then iterates over all the edges of the original graph and adds an edge between two super vertices if and only if there is an edge between vertices belonging to different component. The pseudo code for construction of super-graph is given in Algorithm 1.

After constructing a super-graph, size of the graph is calculated. If the size of a super-graph is small then the naïve algorithm is run on this super-graph to get most statistically significant subgraphs (MSCS). If size of super-graph is large than expected, then the reduction algorithm is applied on a super-graph to reduce it further. The reduction algorithm takes the super-graph $G_s(V_s, E_s)$ as a input graph.
Algorithm 2 Reduction Step

Require: Super-graph $G_s = (V_s, E_s)$ and Threshold $N_\theta$
Ensure: Reduced super-graph $G_r = (V_r, E_r)$

1: while $G_s$.Num_Vertices $> N_\theta$ do
2:   Get $e_s = (u_s, v_s)$ with minimum chi-square value
3:   Create new vertex $v_r$
4:   $v_r = \text{Merge}(u_s, v_s)$
5:   Add $v_r$ to $V_s$
6:   Add all edges of form $(u_s, x)$ and $(v_s, x)$ as $(v_r, x)$ to $E_s$
7:   Remove $u_s, v_s$ and $e_s$ from $G_s$
8: end while
9: return $G_r(V_r, E_r)$

Algorithm 3 Naïve algorithm

Require: Graph $G = (V, E)$
Ensure: MSCS of $G$, Max chi-square value

1: $\text{MAX} \leftarrow 0$
2: for all possible connected subgraphs $C \in G(V, E)$ do
3:   $X \leftarrow X^2$ value of subgraph $C$
4:   if $X > \text{MAX}$ then
5:     $\text{MAX} \leftarrow X$
6:     MSCS $\leftarrow C$
7: end if
8: end for
9: return MSCS, MAX

and threshold $N_\theta$. It iterates over all super-edges and selects the edge with minimum chi-square value. It then merges the vertices of this super-edge. All the neighbours of the merged super-vertices become the neighbours of new vertex. This process repeats till size of graph of becomes less than $N_\theta$. The pseudo code for reduction algorithm is given in Algorithm 2.

After reduction step, size of graph is small enough to run the naïve algorithm. The naïve algorithm finds all the connected subgraphs. As this step takes exponential time, first the original graph is reduced to a smaller size graph such that naïve algorithm can run in practical time. This step takes the graph $G(V, E)$ as a input and gives the MSCS of the graph. The pseudo code for naïve algorithm is given in Algorithm 3.
2.3 Apache Spark GraphX

Apache Spark is a platform for distributed computing [15]. Spark is fast as compared to other distributed frameworks, as it performs its computation in memory. The task of spark is to schedule, distribute and monitor applications on its worker nodes.

Spark performs its computation on Resilient Distributed Data (RDDs). RDDs are immutable datasets. Spark creates RDD mainly from two sources 1. Scala collections 2. Files stored Hadoop File System (HDFS). RDDs are processed in parallel. There are two operations supported by RDDs 1. Transformation and 2. Actions. Transformations transform current RDDs to new RDDs whereas actions do calculations and returns new value from RDDs. Architecture of Apache spark is shown in Figure 2.1.

![Architecture of Apache Spark](image)

**Figure 2.1:** Architecture of Apache Spark

Spark creates a spark context object, which serves as an entry point to all the functionality of the spark. With spark context, it is possible connect to a spark cluster and create new RDDs on that cluster. Files can also be accessed through spark context. Cluster manager manages the cluster. It schedules and distributes
the tasks on worker nodes. It is also responsible to check the status of live working nodes. Worker nodes are the nodes where actual computations occur. They get tasks from cluster manager and perform actions given to them. They report the results back when required, to the driver.

**GraphX**

GraphX is a spark library for Graph data structure which creates directed graphs [16]. Property Graph can be created with the help of this library. Property graphs are the graphs where vertices and edges possess different properties. For example in a social network graph, *user* is a vertex which can have properties like *name, age etc.* Edge which might represent *friendship* between them can also have properties, in this case the *date* when they became friends.

Graph created from this library consists of VertexRDD and EdgeRDD. VertexRDD represents the vertices of a graph and EdgeRDD represents edges. As graph is a combination of VertexRDD and EdgeRDD, parallel computations and algorithms can be run on this graph structure. GraphX has library of common graph algorithms like *pagerank, connected components, triangle counting etc.* GraphX also provides operators to manipulate graphs. In our thesis, we have used GraphX to run our parallel algorithm to mine statistically significant subgraphs.

### 2.4 Challenges

In this section we will explain the challenges that we faced during the implementation of our algorithm. We started with Titan Graph Database [19]. Titan Graph Database is a vertex centric graph database. Vertices are indexed and thus it makes suitable for the queries which are more focused on nearby node data. It saves the graph on disk. So there is a cost of disk read and write when performing operations on the graph. Though Titan Graph Database scales well, it performs badly when whole graph is to be processed. It is not ideal for graph processing system.

We switched to Apache Spark Graph to implement the parallel algorithm to mine
statistically significant subgraph. GraphX and many other graph analytics engines deal with immutable data structures. New copy of graph is created whenever changes occur in existing graph. GraphX is also not vertex centric. Vertex cannot be reached with the just the vertex id. As apache spark runs over java virtual machine (JVM), we rely on apache spark for garbage collection. GraphX is suitable for the processing where there are not many structural changes in the graph.

In our parallel algorithm, we needed a engine where vertices can be referred with given vertex id and also need to process graph in memory for fast computation. Our algorithm consists of steps where we constantly make changes to vertex and edge properties. Also we need multiple copies of graph, but we have to rely on apache sparks garbage collection mechanism to remove the old copy of graph. We implemented algorithm on GraphX which performs far better than Titan Graph Database.
Chapter 3

Algorithm And Implementation

In this chapter we first describe the parallel algorithm for the construction of a super-graph. We will also explain the parallel algorithm to find the connected components in any graph. Later we present our implementation of the parallel algorithm on Apache Spark [15] using GraphX [16] library.

3.1 Construction of a Super-graph - Parallel

The Algorithm 1 to construct a super-graph can be divided into three major steps 1. Removing non-contracting edges 2. Finding connected components on modified graph and 3. Adding the super-edges in the super-graph. Each of these three steps can be parallelized. We will look at each step and will discuss why each step is parallelizable.

Step 1 - Removing Non-contracting Edges : In this step, each edge can be considered as a different graph of a single edge and two vertices. There is no dependency between any two edges to execute this step. As there lies no dependency between an edge which is to be processed and rest of the edges, this step can be run in parallel, producing same effect when run sequentially.

Step 2 - Finding Connected Components : The Algorithm 1 runs BFS algorithm to find the connected components. The BFS algorithm simply assigns a membership to each vertex to the component they belong to. The same effect
can be achieved by propagation algorithm. A vertex and all its neighbours will be member of the same component sharing same membership. So if the process to share membership is started from the vertex having lowest id, then eventually all vertices connected to the vertex in consideration will be member of same component. Then repeat the process from the vertex with second lowest id which has not been assigned a membership. We can clearly see that this step is easily parallelizable. A vertex and its neighbours can be considered as a separate graph. The operation of assigning membership to these graphs can be run in parallel.

**Step 2 - Add Super-edges**: This step is similar to step 1. In this step instead of removing a non contracting edge, a new edge is added between super-vertices. Adding an edge between two super-vertices is not dependent on any other edge in the graph. So all the edges can be process parallely to add the super-edge. The graph in consideration is a simple graph. So even if multiple edges are added between any two super-vertices, there will exist only single edge between those vertices.

In this section we described the parallel algorithm for a super-graph construction. Algorithm 4 first makes the copy $G_s = (V_s, E_s)$ of the original graph $G = (V, E)$ (line 1). It then processes each edge $e = (u, v) \in E_s$ in parallel and deletes it, if an edge is a non-contracting edge (lines 2-6). Algorithm 5 is then applied on $G_s$, which assigns each vertex $v \in V_s$ a component_id, which is the lowest vertex id in the connected component containing that vertex (line 7). In the next step, Algorithm 4 deletes all those $v \in V_s$, where component_id of $v$ does not match with its vertex id (lines 8-12). It then processes each edge $e = (u, v) \in E$ of original graph $G$ in parallel. If the end vertices $u$ and $v$ of an edge $e$ belongs to a different component, then the super-vertices $s_u$ and $s_v$ are found out. $s_u$ and $s_v$ are the vertices in $V$, such that $s_u.id$ and $s_v.id$ is same as $u.component_id$ and $v.component_id$ respectively (lines 13-19). The graph constructed at the end of Algorithm 4 is a super-graph $G_s$. 
Algorithm 4 Construction of a Super-graph - Parallel

Require: Input Graph $G = (V, E)$
Ensure: Super-graph $G_s = (V_s, E_s)$

1: Make a copy of original graph $G_s(V_s, E_s) \leftarrow G(V, E)$
2: for all edges $e = (u, v) \in E_s$ do in parallel
3:   if $e$ is non-contracting then
4:     Delete $e$ from Graph $G_s$
5:   end if
6: end for
7: Run Algorithm 5 on Graph $G_s$, which assigns component.id to every $v \in V_s$
8: for all vertices $v \in V_s$ do in parallel
9:   if $v.id \neq v.component.id$ then
10:      Delete $v$ from Graph $G_s$
11: end if
12: end for
13: for all edges $e = (u, v) \in E$ do in parallel
14:   if $u.component.id \neq v.component.id$ then
15:      $s_u \leftarrow w \in V_s$ such that $w.id == u.component.id$
16:      $s_v \leftarrow w \in V_s$ such that $w.id == v.component.id$
17:      Add super-edge $e_s = (s_u, s_v)$ to $G_s$
18:   end if
19: end for
20: return $G_s(V_s, E_s)$

3.2 Connected Component - Parallel

In this section we will explain the parallel algorithm to find connected component in any graph $G$ [20]. Algorithm 5 first makes the copy $G_{cc} = (V_{cc}, E_{cc})$ of a original graph $G = (V, E)$ (line 1). It then initializes the component.id for each vertex $v \in V_{cc}$ in parallel (lines 2-4). After initialization, all vertices $v \in V_{cc}$ shares their vertex id with their neighbour vertices. Each vertex $v$ compares it component.id with its neighbours vertex id. Vertex $v$ updates its component.id, if its current component.id is greater than any of its neighbours vertex id. This process repeats in parallel till no vertex $v$ updates its component.id (lines 5-12). At the end of the Algorithm 5, each vertex is marked with component.id, which is the lowest vertex id in the component containing that vertex.
Algorithm 5 Connected Component - Parallel

Require: Input Graph $G = (V, E)$
Ensure: Graph $G_{cc} = (V_{cc}, E_{cc})$

1: Make a copy of original graph $G_{cc}(V_{cc}, E_{cc}) \leftarrow G(V, E)$
2: for all vertex $v \in V_{cc}$ do in parallel
3: Initialize $v$.component.id = $v$.id
4: end for
5: repeat
6: for all vertex $v \in V_{cc}$ do in parallel
7: Share $v$.id with $w \in \text{neighbour}(v)$
8: if $v$.component.id > $w$.id then
9: $v$.component.id = $w$.id
10: end if
11: end for
12: until No $v \in V_{cc}$ change its component id
13: return $G_{cc}(V_{cc}, E_{cc})$

Algorithm 6 Reading Input Graph in Spark

Require: File $\text{Vertex}_\text{file}$, $\text{Edge}_\text{file}$
Ensure: Graph $G(V, E)$

1: $\text{RDD}_{\text{vertex}} \leftarrow \text{Map} \ \text{Vertex}_\text{file} \ \text{to create RDD}$
2: $\text{RDD}_{\text{edge}} \leftarrow \text{Map} \ \text{Edge}_\text{file} \ \text{to create RDD}$
3: Construct Graph $G(V, E) \leftarrow \text{Graph}(\text{RDD}_{\text{vertex}}, \text{RDD}_{\text{edge}})$
4: return $G(V, E)$

3.3 Super Graph Construction - GraphX

In this section we will give the pseudo code for implementation of the Algorithm 4 on Apache Spark using GraphX library. The $\text{Vertex}_\text{file}$ and the $\text{Edge}_\text{file}$ is stored on Hadoop Filesystem (HDFS). $\text{SparkContext}$ object is created and used to load the file into the spark. $\text{Vertex}_\text{file}$ and $\text{Edge}_\text{file}$ are processed separately and in parallel to create $\text{RDD}_{\text{vertex}}$ and $\text{RDD}_{\text{edge}}$. Graph $G$ is created from these $\text{RDDs}$. The pseudo code for loading the graph into the spark is given in Algorithm 6.

After loading the graph, we remove the non-contracting edges in graph $G(V, E)$ by taking subgraph. $\text{subgraph}$ method of GraphX takes an edge predicate and a vertex predicate as a input. Vertices and edges satisfying their respective predicates are preserved and the rest are discarded. We take $\text{subgraph} G_{\text{sub}}(V_{\text{sub}}, E_{\text{sub}})$ of a graph $G$ such that only those edges having identical labels to their end vertices
Algorithm 7 Super Graph Construction - GraphX

Require: Graph $G(V, E)$

Ensure: Super Graph $G_s(V_s, E_s)$

1: $G_{sub}(V_{sub}, E_{sub}) \leftarrow G_.subgraph(E_{pred} = (e = (u, v)) \rightarrow e.u.label == e.v.label)$
2: $G_{cc}(V_{cc}, E_{cc}) \leftarrow G_.connectedComponent()$
3: $V_{new} \leftarrow join (V_{cc}, V)$
4: $G_{temp}(V_{temp}, E_{temp}) \leftarrow Graph(V_{new}, E)$
5: for all vertex $v \in V_{temp}$ do in parallel
   6: if $v.id \neq v.componentId$ then
      7: Mark v as ”delete”
   8: end if
9: end for
10: for all edge $e = (u, v) \in E_{temp}$ do in parallel
11: if $u.label == v.label$ then
   12: Mark e as ”delete”
13: end if
14: end for
15: $G_s(V_s, E_s) \leftarrow G_{temp}.subgraph(E_{pred} = (e) \rightarrow e.attr \neq ”delete” \& V_{pred} = (v) \rightarrow v.attr \neq ”delete”)$
16: return $G_s(V_s, E_s)$

are preserved. We then run the connected component algorithm of GraphX on $G_{sub}$ to get $G_{cc}(V_{cc}, E_{cc})$. This connected component algorithm assigns each vertex $v \in V_{cc}$ its component id. New RDD vertex $V_{new}$ is constructed by joining $V_{cc}$ and $V$. Using $V_{new}$ and $E$, another graph $G_{temp}(V_{temp}, E_{temp})$ is created. All vertices of $G_{temp}$ are processed in parallel and a vertex $v \in V_{temp}$ is marked as a ”delete”, if its component id does not match with its vertex id. Similarly, all edges of $G_{temp}$ are processed in parallel and an edge $e \in E_{temp}$ is marked as a ”delete”, if the label of the end vertices are same. The subgraph $G_s(V_s, E_s)$ of a graph $G_{temp}$ is taken by applying subgraph method on a graph $G_{temp}$ to remove all vertices and edges marked as ”delete”.

After getting the super-graph $G_s(V_s, E_s)$, size of the super-graph is checked. If the size of the super-graph $G_s$ is less than a threshold $N_\theta$, then naïve Algorithm 3 is run on a super-graph $G_s$ to get the statistically significant subgraphs. If size of a super-graph is more than $N_\theta$, then by applying the reduction Algorithm 2 on a super-graph $G_s$ size of a super-graph $G_s$ is reduced and then naïve Algorithm 3 is run on the reduced graph $G_r = (V_r, E_r)$ to get the statistically significant subgraphs.
3.4 Time Complexity of the Algorithm

Our parallel Algorithm 4 for the super-graph construction is linear in time with respect to the size of the graph. There are three parallel steps in Algorithm 4. Step 1 is to remove all non-contracting edges from the graph. If there are \( n \) vertices and \( m \) edges in graph and \( k \) processors, where \( k \ll m \), then each processor takes \( O(m/k) \) time. Step 2 is to find connected components in the graph. Let \( d \) be the diameter of the largest component in the graph. As this step propagates its vertex \( id \) to its neighbours, this step will be executed \( d \) times. Thus this step takes \( O(d) \) time. Step 3 is similar to step 1 where all edges are processed. We can say that step 3 takes \( O(m/k) \) time. Thus we have shown that the time complexity of our parallel Algorithm 4 is linear when \( k \ll m \).

The time complexity of the reduction algorithm is given by Arora et al. [13] which is \( O(n_s \log m_s) \), where \( n_s \) is a number of super-vertices and \( m_s \) is a number of super-edges in the super-graph. The time complexity of the naïve algorithm is exponential on small constant.

If the original graph is dense, then the number of super-vertices is constant. In this case time complexity of the reduction algorithm and naïve algorithm can be ignored. Thus, the total time complexity of Algorithm 4 is linear.