Distributed Clustering of Uncertain Graphs

Dimitrios I. Pesios

Supervisor: Prof. Apostolos Papadopoulos

School of Informatics
Aristotle University of Thessaloniki

This dissertation is submitted in partial fulfilment of the requirements for the Master of Science degree M.Sc. in Data and Web Science

Faculty of Sciences  July 2019
Abstract

In this dissertation, we were initiated with the purpose of implementing the pKwik clustering algorithm in a distributed fashion, which is an algorithm designed for uncertain graphs, that is graphs for which there is a probability of existence incurred with every edge. Unlike deterministic or conventional graphs, uncertain ones require special treatment since they present computational challenges in even the simplest and ordinary problems of data science. To overcome those challenges many strategies are employed such as specialized approximation algorithms, however, mainly in a centralized fashion. The novelty of this dissertation lies in the fact that distributed processing can facilitate the formation of solutions when it is too costly or impossible for the solution to be designed in a centralized fashion. In order to leverage parallel processing, we use the well-known Apache Spark framework and its GraphX module on top of it, in conjunction with the Scala programming language in a 80-core mainframe machine dedicated for heavy computations. Our results demonstrated that the pKwik algorithm scales very well when the size of input graph increases, while it was observed that the baseline Louvain algorithm is irrespective of the graph size scaling. Furthermore, when the available execution cores in the mainframe machine increase both algorithms scale well and it seems that the number of 16 cores is adequate for both since there is no substantial variation in the execution time. We used two indexes, the Rand index and a Dunn-like index, to measure the robustness of the clustering algorithms produced, the first one based on ground-truth data and the second one on more intuitive criteria. We calculated that the Rand Index for the full-fledged graph is 0.6 for the pKwik algorithm while is even smaller and equal with 0.1 for the baseline Louvain algorithm, which means that the specialized pKwik algorithm is more close to the ground-truth. Additionally, the Dunn-like Index is even higher and equal with 0.7 for the pKwik algorithm and 0.4 for the baseline Louvain algorithm which essentially means that the clusters created are more compact and well-separated.
Σε αυτήν την διπλωματική εργασία, είχαμε ως σκοπό την υλοποίηση του αλγορίθμου συσταδοποίησης pKwik με κατανεμημένο τρόπο, ο οποίος είναι ένας αλγόριθμος σχεδιασμένος για αβέβαια γραφήματα, δηλαδή γραφήματα για τα οποία υπάρχει μια πιθανότητα ύπαρξης επισυναπτόμενης με κάθε ακμή του. Σε αντίθεση με τα αυτοτά ή συμβατικά γραφήματα, τα αβέβαια γραφήματα απαιτούν ειδική αντιμετώπιση γιατί παρουσιάζουν υπολογιστικές προκλήσεις προκλημένες σε ακόμη και τα πιο απλά και συνηθισμένα προβλήματα της Επιστήμης Δεδομένων. Για να ξεπεραστούν αυτές οι προκλήσεις έχουν επιστρατευθεί πολλές στρατηγικές όπως εξειδικευμένοι προσεγγιστικοί αλγόριθμοι, ωστόσο, με κεντρικοποιημένο τρόπο. Η καινοτομία της διπλωματικής έγκειται στο γεγονός ότι η κατανεμημένη επεξεργασία μπορεί να διευκολύνει την δημιουργία λύσεων όταν είναι ακριβή η αδύνατο να σχεδιαστεί μια λύση με κεντρικοποιημένο τρόπο. Για να χρησιμοποιήσουμε παράλληλη επεξεργασία, επιλέξαμε το ευρέως γνωστό Apache Spark framework και την υποβιβλιοθήκη αυτού GraphX, σε συνδυασμό με γλώσσα προγραμματισμού Scala σε έναν υπολογιστή 80 πυρήνων προορισμένο για βαρείς υπολογισμούς. Τα αποτελέσματα μας έδειξαν ότι ο αλγόριθμος pKwik κλιμακώνεται πολύ καλά όταν αυξάνεται το μέγεθος του γραφήματος, ενώ παρατηρήθηκε ότι baseline αλγόριθμος Louvain δεν επηρεάζεται από την κλιμάκωση του μεγέθους του γραφήματος. Επιπλέον, όταν οι διαθέσιμοι πυρήνες του υπολογιστή αυξάνονται και οι δύο αλγόριθμοι κλιμακώνονται πολύ καλά και φαίνεται ότι ο αριθμός των 16 πυρήνων είναι αρκετός και για τους δυο διότι δεν υπάρχει σημαντική απόκλιση στον χρόνο εκτέλεσης για παραπάνω πυρήνες. Χρησιμοποιήσαμε δύο δείκτες, τον Rand index και έναν δείκτη που μοιάζει με τον Dunn Index, για να μετρήσουμε την ευρωστία των συσταδοποιήσεων που παρήγαγαν οι αλγόριθμοι, με τον πρώτο να βασίζεται σε δεδομένα βασικής αλήθειας και τον δεύτερο σε πιο διαισθητικά κριτήρια. Υπολογίσαμε ότι ο Rand Index για το πλήρες γράφημα είναι 0.6 για τον pKwik αλγόριθμο, ενώ είναι για τον baseline αλγόριθμο είναι μικρότερος και ίσος με 0.1, το οποίο σημαίνει ότι ο pKwik αλγόριθμος είναι πιο κοντά στην βασική αλήθεια. Επιπρόσθετα, o Dunn Index είναι ακόμη μεγαλύτερος και ίσος με 0.7 για τον pKwik αλγόριθμο και 0.4 για τον baseline Louvain αλγόριθμο, το οποίο στην
ουσία σημαίνει ότι οι συστάδες που δημιουργήθηκαν από τον πρώτο είναι πιο συμπαγείς και διαχωρισμένες.
## Contents

### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ix</td>
<td>ix</td>
</tr>
</tbody>
</table>

### 1 Introduction

1.1 Preliminaries ........................................... 1
   1.1.1 Graph Data Structures .............................. 1
   1.1.2 The Clustering Problem in Graphs ................. 2
   1.1.3 Distributed Data Processing Algorithms .......... 3

1.2 Uncertainty and Graphs .................................. 4
   1.2.1 Random vs. Uncertain Graphs ....................... 5
   1.2.2 Models ............................................ 5
   1.2.3 Processing Challenges ............................. 7

1.3 Applications of Uncertain Graphs ....................... 8
   1.3.1 Protein-Protein Interaction Networks ............. 8
   1.3.2 Social Networks ................................ 9

1.4 The Apache Spark Framework ............................. 9

### 2 Algorithms for Uncertain Graphs Clustering

2.1 Prerequisites ........................................... 13
   2.1.1 Notions of Distance in Uncertain Graphs .......... 13
   2.1.2 Edit Distance ..................................... 15
   2.1.3 Correlation Clustering Problem .................... 16

2.2 The Louvain Algorithm (as Baseline) .................. 16

2.3 The pKwik Algorithm ................................... 18

2.4 The Furthest Algorithm ................................ 19

2.5 The Agglomerative Algorithm .......................... 22

2.6 Other State-Of-The-Art Algorithms ..................... 23
3 Experimental Study 25
   3.1 Background ................................................. 25
   3.2 Dataset Description ........................................... 25
   3.3 Time Performance ............................................. 26
      3.3.1 Graph Size vs. Running Time .............................. 26
      3.3.2 Running Time vs. Number of Cores .......................... 27
      3.3.3 Number of Clusters vs. Alpha Parameter (pKwik) .......... 29
   3.4 Validity Measures ............................................. 29
      3.4.1 Rand Index of Clusterings - Accuracy ...................... 30
      3.4.2 Dunn-like Index of Clusterings ........................... 31

4 Conclusions and Further Research 33

Bibliography 35
List of Figures

1.1 The graphs illustrated above are random graphs on 10 vertices with edge probabilities distributed uniformly in [0, 1] (Source: mathworld.wolfram.com) 6
1.2 The logo of Apache Spark Framework (Source: https://spark.apache.org/) 10

2.1 pKwik Algorithm Pseudocode .............................. 18
2.2 Furthest Algorithm Pseudocode .............................. 21
2.3 Agglomerative Algorithm Pseudocode ......................... 22

3.1 Running time of pKwik and Louvain algorithms against the graph size as described previously. (Produced using SciDAVis) ......................... 26
3.2 Response time of the pKwik algorithm as the number of cores increases by a power of two (Produced using SciDAVis) ......................... 28
3.3 Response time of the louvain algorithm as the number of cores increases by a power of two (Produced using SciDAVis) ......................... 28
3.4 Number of clusters produced by pKwik algorithm by varying the alpha input parameter. (Produced using SciDAVis) ......................... 29
Chapter 1

Introduction

1.1 Preliminaries

In the first part of this dissertation, it is imperative that we introduce the reader to the concepts, and other necessary general prerequisites, which are going to be heavily used in the forthcoming chapters. Our purpose is not to be completely thorough and detailed, but concise enough in order to present a consistent flow to follow.

Accordingly, we are going to introduce the structures our dataset is characterized as, that is Graphs in the first, while later on we are going to talk about the problem to which implemented algorithms provide a solution, the so-called clustering problem in graphs. It is also useful to mention that the algorithms implementation will be formulated in a distributed fashion, using the Apache Spark Framework, and it would be necessary for various reasons to define what distributed processing means.

1.1.1 Graph Data Structures

Graphs [7], as the most general abstract data type, comprise the most indispensable part of Computer Science because they have numerous applications in many scenarios where the situation can be modeled as graph.

Another concept which is abundant in Computer Science is the one of object which can seamlessly be combined with the concept of graph to form more complex objects that can be considered as a whole. In order to clarify things further, someone has to talk about the nodes (or vertices) in the graph, which denote those combination of object as a finite set, and edges which denote the relation among them. This inter-relation can be immediate, between a pair of nodes for example or indirect when another node intervenes. Typically, in many
text-books, graphs are depicted as a set of dots for the nodes, joined by lines or curved arcs as the set of edges.

The graphs can be categorized depending on the nature of relation among their edges. So, we have undirected graphs when this relation between a pair of nodes is mutual or directed graphs when there is a direction on the arc meaning that only part of the pair is influenced. For example, in the social network of humans, friendship is mutual and can be modeled with an undirected graph, while in computer networks data flow is not, and it can be better modeled as a directed graph since it is natural to consider that the second part of the pair receives the data. Furthermore, graphs can additionally be weighted. A weighted graph is a graph in which a number (the weight) is assigned to each edge. Such weights might represent for example costs, lengths or capacities, depending on the problem at hand. Some authors call such a graph a network.

Formally, we symbolize graphs using the letter \( G \), and in many times we write it as the ordered pair \( G = (V, E) \) where \( V \) is the set of vertices and \( E \) is the set of edges.

Different data structures for the representation of graphs are used in practice \[7\]:

- **ADJACENCY LIST**: Vertices are stored as records or objects, and every vertex stores a list of adjacent vertices. This data structure allows the storage of additional data on the vertices. Additional data can be stored if edges are also stored as objects, in which case each vertex stores its incident edges and each edge stores its incident vertices.

- **ADJACENCY MATRIX**: A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices. Data on edges and vertices must be stored externally. Only the cost for one edge can be stored between each pair of vertices.

- **INCIINDEENCE MATRIX**: A two-dimensional Boolean matrix, in which the rows represent the vertices and columns represent the edges. The entries indicate whether the vertex at a row is incident to the edge at a column.

### 1.1.2 The Clustering Problem in Graphs

These are scenarios where we are interested in finding groups of objects in our dataset in such a way that objects in the same group (called a *cluster*) are more similar (in some sense) to each other than to those in other groups (clusters) \[4\]. This problem occurs naturally in many field of Data Science and it is a main task in exploratory data mining. With respect to other data mining techniques the solutions require the so-called unsupervised learning, meaning
that a training procedure is not necessary to produce the groups (lack of classification model) and moreover the best number of groups is not always known.

In order to produce those clusters, many algorithms can be employed and if we run into literature we will discover a plethora of them. However, the most widely known and used algorithm is \textit{k-means} \cite{9} and in many introductory texts it is taught in conjunction with the clustering problem.

The very problem itself can be viewed as an optimization problem \cite{16}, where given a set of points (objects or nodes when a graph is concerned), one tries to minimized an multi-objective function (configuration costs, for example) having as a constraint the number of produced clusters (among others w.r.t. the problem at hand). However, so as to formulate the problem solution, it is very critical to define the distance among those objects, a metric in mathematical terms, which will enable us to discern similar objects. When distance between two objects is small, we can say that those two objects are similar in a way and dissimilar vice-versa.

\subsection*{1.1.3 Distributed Data Processing Algorithms}

Moreover, it is also necessary to introduce the concept of \textit{distributed algorithms} which have many applications in computing since they can solve many practical problems in the fraction of time needed to do the computation in a centralized fashion (which in turn is the opposite concept of what the title of this dissertation suggests).

A \textit{distributed algorithm} is an algorithm designed to run on computer hardware constructed from interconnected processors. In more formal aspect, and as it is stated in Wikipedia \cite{5}, distributed algorithms are a sub-type of parallel algorithms, typically executed concurrently, with separate parts of the algorithm being run simultaneously on independent processors, and having limited information about what the other parts of the algorithm are doing. One of the major challenges in developing and implementing distributed algorithms is successfully coordinating the behavior of the independent parts of the algorithm in the face of processor failures and unreliable communications links. The choice of an appropriate distributed algorithm to solve a given problem depends on both the characteristics of the problem, and characteristics of the system the algorithm will run on such as the type and probability of processor or link failures, the kind of inter-process communication that can be performed, and the level of timing synchronization between separate processes.

However broad this concept may seem, it is very handful in the data processing field of computer science. The very essence of it derives from the term "distributed" which is another sub-field of computer science, the so-called \textit{distributed systems}. A distributed system is a system whose components are located on different networked computers, which
communicate and coordinate their actions by passing messages to one another, having to achieve a common goal which in the former case might be the computational solution to a problem. A computer program that runs within a distributed system is called a distributed program.

### 1.2 Uncertainty and Graphs

At this point, it is also imperative we introduce the structures we are going to get engaged with in this dissertation as well as the - widespread in many fields - concept of uncertainty. We are referring to uncertain graphs.

The term uncertain (or sometimes probabilistic) graphs obviously consists of two sub-terms; **uncertainty**, as the main originate noun, and **graphs** that we have already introduced in a previous subsection.

Uncertainty, generally, corresponds to many things depending on the scientific field we deal with. However, usually uncertainty is connected with a level of existence which is convenient mathematically to be modeled as probability. So, a probability value usually denotes whether something will exist or not. In computer science and data management specifically, there is a whole field called **uncertain data** [20] which has to do with designing computationally efficient solutions in situations where data will be uncertain regarding its existence.

As regards graphs, this uncertainty is usually connected with potential edge existence, but there can be occasions where it is the nodes set that potentially exists according to a specific probability. The second case is out of our scope since we will only consider edge probability.

It is, moreover, necessary we state that the quantification of uncertainty in this type of structures and especially how it came up is also out of the scope of this dissertation. There is also a whole inter-disciplinary field called Uncertainty Quantification [19] that deals with this matter, and despite being interesting we want touch it. We will refer to some general characteristics of specified applications in the respective section of this chapter.

In a way, uncertain graphs where introduced by Erdos and Renyi as the so-called random graphs [18]. Uncertain graphs can be thought as generalized random graphs since every edge may carry a different value of probability, whereas in random graphs every edge has the same value of probability which is usually 0.5.
1.2 Uncertainty and Graphs

1.2.1 Random vs. Uncertain Graphs

By random graphs, as it is stated in [18], we generally refer to "probability distributions over graphs". So strictly speaking the term "random graphs" is literally a misnomer [2]. It is many times enough to characterize these kind of graphs by means of the process that generates them. Mathematically speaking, random graph refers almost exclusively to the Erdős–Rényi random graph generative model. However, for historical reasons we have to mention that random graphs were first defined by Paul Erdős and Alfréd Rényi in their 1959 paper "On Random Graphs" and independently by Gilbert in his paper "Random graphs".

So, depending on the mathematical perspective we follow, there are two closely related models for generating random graphs; the Erdős–Rényi model we already mentioned and the Gilbert model. In the first model, all graphs on a fixed node set with a fixed number of edges are equally likely to come into existence, while in the second one each edge has a fixed edge of being present or absent, independently of the other edges. "These models can be used in the probabilistic method to prove the existence of graphs satisfying various properties, or to provide a rigorous definition of what it means for a property to hold for almost all graphs", as it is stated in [6]. We will stick to the Gilbert model for the rest of this dissertation, by not taking into consideration the other one.

Uncertain graphs best fit with the second case for their mathematical representation (Gilbert model) and can be considered as more generalized random graphs, in that each edge has its own, maybe unique, probability value of existence and not the same with every other. Hence, the generative process of producing a random graph and mentioned above, can be modified and be used to produce an uncertain with no much additional pain. Even though conceptually close with each other, random and uncertain graphs do not share much in respect with the process of mathematically proving some of the properties that uncertain graphs may have, for example phase transition. We will dedicate a special subsection on this later on.

Furthermore, uncertain graphs, in contrast with random graphs, seem to model much more real-world situations than the latter. Random graphs are considered more abstract as mathematical structures, but they are useful because they share and borrow many concepts to the former.

1.2.2 Models

We are going to briefly define both models mentioned above so that a distinction is made, one for random graphs and one for the most generalized uncertain graphs respectively. However, we will once again emphasize that Gilbert modeling is of concern to us (the probability
model), which is going to be presented in the case of uncertain graph modeling. In other words, even random graphs could have been presented in terms of it.

**Random Graph Model**

Let $n$ be a positive integer and $0 \leq p \leq 1$, the probability [2]. The random graph $G(n, p)$ is a probability space over the set of graphs on the vertex set $\{1, \ldots, n\}$ determined by

$$Pr[(i, j) \in G] = p$$

with these events mutually independent. This model is often used as a probabilistic method for proving the existence of certain graphs.

So, having said so, we can argue that there is a compelling dynamic model for random graphs which goes as follows. For all pairs $i, j$ let $x_{ij}$ be selected uniformly from $[0, 1]$, while the choices are mutually independent. We can imagine $p$ going from 0 to 1. Originally, all potential edges are non-existent. The edge from $i$ to $j$ is present when $p$ reaches $x_{ij}$ and remains present (there is no concept of time inserted). On the other hand, when $p$ equals 1, all edges are present. We can see that a distribution $G(n, p)$ is defined and, as $p$ increases, $G(n, p)$ evolves from empty to full.

A random graph $G(n, e)$, that is one with certain number of edges, can be similarly defined using this dynamic model. Both graphs, it is mathematically proven, will have similar properties.

**Uncertain Graph Model**

In order to introduce the uncertain graph model we are going to use the description as it is provided in one of research papers studied for the purpose of fulfillment of this dissertation. Needless to say, the description would differ if we had used a different research paper since the very term is not well-established but still in the process of continuous research for the time being.
1.2 Uncertainty and Graphs

So, as it is stated in [11], uncertain (or as the authors exceptionally suggest "probabilistic") graphs can be undirected or directed and carry additional labels on the edges (such as weights). We focus on undirected probabilistic graphs for the sake of simplicity and independence among edges. Graphs with no incurred edge probabilities, that preserve the classic case, are called deterministic graphs.

Having said so, we represent an uncertain graph \( G \) using tuple \( G = (V, P, W) \); \( V \) which corresponds to the set of nodes in \( G \), and we assume that \( \|V\| = n \). \( P \) maps every pair of nodes to a real number in \([0, 1]\); \( P_{uv} \) represents the probability that edge \( \{u, v\} \) exists. For weighted graphs we also use \( W : V \times V \rightarrow R \) to denote the weight associated with every edge \( \{u, v\} \in V \times V \). In this paper we focus on unweighted uncertain graphs. In this case, we represent the uncertain graph as \( G = (V, P) \). For an uncertain graph \( G = (V, P) \), we define its complement to be the uncertain graph \( G' = (V, 1 - P) \).

We can think of an uncertain graph as a generative model for deterministic graphs. A deterministic graph \( G = (V, E_G) \) is generated by \( G \) by connecting two nodes \( u, v \) via an edge with probability \( P_{uv} \). Deterministic graphs are an instance of uncertain graphs for which \( P_{uv} \in \{0, 1\} \). Also, \( G_n, p \) Erdos-Renyi random graphs are an instance of probabilistic graphs where all edge probabilities are the same and equal to \( p \).

This way possible world semantics is presented, a concept well-known from probabilistic databases research field. So, in other words, every uncertain graph is treated as generative model for deterministic graphs and each such a graph is considered to be a possible world, or simply an instantiation of the uncertain generative one, associated with a certain probability to be generated.

We write \( G \sqsubseteq G \) to denote that \( G \) was generated from \( G \), where the second is an uncertain graph. The probability that \( G = (V, E_G) \) is sampled from \( G = (V, P) \) is:

\[
Pr[G] = \prod_{\{u, v\} \in E_G} P_{uv} \prod_{\{u, v\} \in (V \times V) \setminus E_G} (1 - P_{uv})
\]

If \( M = \binom{|V|}{2} \), then there are \( 2^M \) distinct graphs that can be generated by \( G \). We use the term possible world to refer to each such graph, as we already pointed out.

### 1.2.3 Processing Challenges

In the context of uncertain graphs there can be both semantic and computational challenges, according to [10].

Semantically speaking, we can argue that there is no uniform model of uncertain graphs, that is the interpretation of probability of existence is application specific and a different
model is needed to be formulated in each case. One extraordinary example is the definition of shortest path, which is no trivial compared to deterministic graphs and heavily application-dependent.

As regards the computational perspective, while many graph algorithms are inherently hard problems, in the context of uncertain even the simplest algorithms such as reachability and shortest path finding become \#P-Complete and hence more costly over uncertain graphs. Thus, exact computation is impossible to achieve with nowadays massive datasets and the research focus is on approximative and greedy solutions with efficient sampling, indexing and filtering techniques.

To further enhance our intuition on the challenges emerging, let us consider the possible world concept. We will show later on, that on the clustering solution offered by [11], the computation of the cost function (edit distance) to be computed exhaustively requires the enumeration of $2^{|V|}$ possible worlds, not to mention that some nodes might become disconnected in some of the instantiations.

### 1.3 Applications of Uncertain Graphs

Graphs themselves can model a plethora of applications. There are both situations that can occur in nature which can be viewed this way such as biological networks, cellular networks and others, and human artifacts that can be viewed under this perspective such as computer networks, social networks, transportation systems and many many more, under some circumstances.

It may come strange to the reader that the edge in these networks can modeled with a probability of existence, but as we will see this comes more or less naturally depending on the problem we want to analyze and finally solve. Accordingly, when there is a potential for a network to be instantiated out of a uncertain or probabilistic one, we can seamlessly consider it an uncertain graph.

Our purpose is not to be exhausted regarding the applications of uncertain graphs, but rather point out the most cutting-edge applications of this perception.

#### 1.3.1 Protein-Protein Interaction Networks

As it is stated in [https://ppi-net.org/], "The development of new medicines relies upon the ability of scientists to understand the biological details of a disease and also the way in which new molecular medicines can be designed to either cure the illness or alleviate its symptoms. One very important biological mechanism is the way in which one protein recognizes and
binds to another protein in order to regulate its function. This functional regulation by protein-protein interactions underpins most of the biological activity in living cells, and yet we do not understand what properties of a protein allow it to bind to another, nor do we understand how to design molecules to prevent or enhance such interactions. Gaining such an understanding would be a huge advance. It is estimated that there are approximately 650,000 types of specific protein-protein interactions in each human cell. This means that there are potentially 650,000 targets for modifying biological function by the use of drugs. Essentially every part of the biological process, and therefore every disease, could in principle be addressed by such drugs. Although the knowledge of how to do this remains beyond our grasp, progress has been made in some parts of the problem, and new ways of looking at the problem have also been developed with others already in development." So, in a few words, PPI networks can be regarded as mathematical representations of the physical contacts between proteins in the cell.

In the context of clustering, and as a purpose of this dissertation, we focus on the problem of partitioning this network into clusters. That is finding complexes of proteins in the network that have similar behavior and potentially interact with some way with each other.

1.3.2 Social Networks

Social interactions additionally can naturally be modeled or be associated with a probability. One problem social network theorists are trying to solve is the so-called link prediction problem [13] which has to do with predicting within a certain threshold of probability that two directly unconnected nodes (persons) get connected in a later on point of time, that is becoming friends. The input to the solution algorithm would take into consideration both the weights of tightness among existing nodes and similar phenomena of connection in the past.

Another more straightforward problem occurs in the context of information diffusion. For information to be diffused in the network some kind of influence has to be modeled among nodes by using a probability value. The bigger the probability the more likely for the information to be passed to consequent nodes. Even the potential of the information diffusing can viewed by means of an uncertainty level, that is it going to happen in a certain probability.

1.4 The Apache Spark Framework

It would be a huge omission not to mention some words about framework we are going to use for the fulfillment of the technical part of this dissertation which is the implementation
of algorithms some researchers proposed for the solution of the clustering problem over uncertain graphs. We are referring to Apache Spark framework [https://spark.apache.org/], which as the years pass by, gains popularity in the data science community since it can solve and be applied to various scenarios and applications.

Apache Spark is an open-source distributed general-purpose cluster-computing framework. Originally developed at the University of California, Berkeley’s AMPLab, the Spark codebase was later donated to the Apache Software Foundation, which has maintained it since. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. Historically speaking, there was another framework which preceded it and borrowed many concepts to it, it was Hadoop.

But how do Hadoop and Apache Spark relate to each other ? In order to answer this question we have to mention that Spark is a fast and general processing engine compatible with Hadoop data. It can run in Hadoop clusters through YARN (the Resource Manager) or Spark’s standalone mode, and it can process data in HDFS, HBase, Cassandra, Hive, and any Hadoop InputFormat. It is designed to perform both batch processing (similar to MapReduce) and new workloads like streaming, interactive queries, and machine learning.

We can consider it as a superset functionality compare to Hadoop, since the logic of the latter relies only on implementing two functions, the Map and Reduce function according to MapReduce programming paradigm. Spark can add more programming logic and abstraction over the MapReduce paradigm by allowing developers to implement complex programming logic. So, Spark facilitates the implementation of both iterative algorithms, which visit their data set multiple times in a loop, and interactive/exploratory data analysis, i.e., the repeated database-style querying of data. It is also stated that , the latency of such applications may be reduced by several orders of magnitude compared to Apache Hadoop MapReduce implementation. Among the class of iterative algorithms are the training algorithms for machine learning systems, which formed the initial impetus for developing Apache Spark.

For implementing the algorithms in this dissertation we use GraphX [https://spark.apache.org/graphx/]. GraphX is a distributed graph-processing framework on top of Apache Spark.
It provides two separate APIs for implementation of massively parallel algorithms (such as PageRank): a Pregel abstraction, and a more general MapReduce-style API. Unlike its predecessor Bagel, which was formally deprecated since older Spark versions, GraphX has full support for property graphs (graphs where properties can be attached to edges and vertices). One of those properties might be the probability in an uncertain graph. Like Apache Spark, GraphX initially started as a research project at UC Berkeley’s AMPLab and Databricks, and was later donated to the Apache Software Foundation and the Spark project.
Chapter 2

Algorithms for Uncertain Graphs
Clustering

2.1 Prerequisites

In the second part of this dissertation, we are going to describe and present some of the algorithms implemented so as to be ready for the experimental procedure we are going to follow in the next part. These algorithms are considered to be state-of-the-art regarding the applications problems they are promising to solve, and are mainly proposed in [11].

Before getting on their descriptions, it is important that we introduce some concepts they are based on such as the various notions of distances in uncertain graphs and the edit distance cost function. First and foremost, we are going to describe the baseline algorithm which is the well-known Louvain algorithm, an algorithm also used in deterministic graphs clustering. Secondly, we are going to present the pKwik algorithm along with the Furthest algorithm, where both of them presented in [11]. Agglomerative algorithm is going also to be presented but finally it is not implemented in this dissertation, as well as the Farthest algorithm, mainly due to the various of implementations other researchers achieved in the past. Last but not least, we are going to present some other algorithms found in literature.

2.1.1 Notions of Distance in Uncertain Graphs

The problem of finding shortest paths in uncertain graphs (or more simply distances among nodes) is one of the first problems that drew interest in computer science due both to its importance and its elegant solution, the Dijkstra algorithm [17]. In this section, we will consider definitions of distances in uncertain graphs that extend the common shortest path distance, according to aforementioned reference.
The basic problem is as follows:

*Most Probable Path Problem*: Given an uncertain graph, as described earlier, and any pair of nodes \((s, t) \in V \times V\), find the most probable path between those two nodes.

The approaches presented here are based on what was proposed in [17]. We have the following four approaches.

**Naive Approach**

The problem can be easily solved by considering a deterministic weighted graph with the same set of edges and nodes, where its weights would equal \(w(e) = -\log(p(e))\), and consequently running the Dijkstra algorithm on the weighted graph. We should note that the path found may have very small probability and its distance might be different compared to what it is found using other typical distances in sampled graphs of the uncertain graph. For this reason, this approach is judged as "naive" since it does not take into account the statistical distribution of the given path among possible worlds.

**Majority Distance Approach**

Given again the uncertain graph and any pair of nodes we define majority distance \(d_j(s, t)\) to be the most probable shortest-path distance:

\[
d_j(s, t) = \text{argmax}_{d} p_{s,t}(d)
\]

In other words, we are looking for the shortest path distance that is more likely to occur or observed when sampling a random graph from the uncertain graph. This problem is more interesting in unweighted graphs, or graphs with integer weights.

**Expected Distance Approach**

We define expected distance \(d_E(s, t)\) to be the expected shortest path distance among all possible worlds:

\[
d_E(s, t) = \sum_d d \cdot p_{s,t}(d)
\]

The above definition is problematic because the expected distance is trivially infinite for interesting settings. This is due to the fact that most likely there exists a possible world where \(s\) and \(t\) are disconnected. We modify this definition to a more meaningful one. We consider only graphs for which there exists a path between \(s\) and \(t\). Along with the distance, we also
2.1 Prerequisites

calculate the probability of this event (reliability) to quantify how meaningful this expected value is.

Expected Reliable Distance Approach

Given again an uncertain graph and a pair of nodes from it, we define Expected Reliable Distance to be the expected shortest path between those two nodes in all possible worlds in which there exists a path, and the probability \( p(s,t) \) that there exists a path between the two nodes, \( s \) and \( t \). So the answer to a related query is a tuple of the form \( \langle d_{ER}(s,t), p(s,t) \rangle \), where

\[
d_{ER}(s,t) = \sum_{d|d<\infty} d \cdot \frac{p_{s,t}(d)}{1 - p_{s,t}(\infty)}
\]

Last but not least, we will also define Median Distance.

Median Distance Approach

This approach is using the definition of median shortest path distance among all possible worlds and is given as:

\[
d_M(s,t) = \arg\max_d \left\{ \sum_{d=0}^{D} p_{s,t}(d) \leq \frac{1}{2} \right\}
\]

Notice that the median distance may be infinite for some pairs \( s \) and \( t \) that are far away, however, it is not trivially infinite as it is in the case of the expected distance.

2.1.2 Edit Distance

Given two deterministic graphs \( G = (V, E_G) \) and \( Q = (V, E_Q) \) we define the edit distance between \( G \) and \( Q \) to be the number of edges that need to be added or deleted from \( G \) in order to be transformed into \( Q \). We formalize as:

\[
D(G, Q) = |E_G\setminus E_Q| + |E_Q\setminus E_G|
\]

We must extend this definition to the edit distance between an uncertain graph and a deterministic graph. We define this new edit distance to be the expected edit distance deterministic graphs derived from the uncertain one, with the deterministic graph (that is the expected value in the distribution).
2.1.3 Correlation Clustering Problem

The correlation clustering problem goes as follows [11]: given $n$ objects in $V$ and between any pair of objects $u$ and $v$ there is a (+) or a (-) relation. We use $E^+$ to denote the set of pairs that are +related (and equivalently -related). The goal is to find a partition of the nodes covering $V$ so as to minimizing the set of disagreement pairs, that is (+) that are in different cluster and (-) that are in the same cluster.

In its weighted version, each $(u,v)$ pair is assigned two weights $W^+$ and $W^-$. The goal here is to minimize the sum of pairs’ $W^+$ that are in different cluster over $W^-$ that are in the same cluster.

When $(W^+) + (W^-) = 1$ we can argue that the weights satisfy a probability constraint.

2.2 The Louvain Algorithm (as Baseline)

The Louvain Method [15] for community detection (another view of clustering) is a method to extract communities (groups) from large networks created by Blondel et al. from the University of Louvain. The method is a greedy optimization method that appears to run in sub-linear time.

The inspiration for this method of community detection is the optimization of Modularity as the algorithm progresses. Modularity is a scale value between -1 and 1 that measures the density of edges inside communities to edges outside communities. Optimizing this value theoretically results in the best possible grouping of the nodes of a given network, however going through all possible iterations of the nodes into groups is impractical so heuristic algorithms are used. In the Louvain Method of community detection, first small communities are found by optimizing modularity locally on all nodes, then each small community is grouped into one node and the first step is repeated.

The value to be optimized is modularity, defined as a value between -1 and 1 that measures the density of links inside communities compared to links between communities. For a weighted graph, modularity is defined as:

$$Q = \frac{1}{2m} \sum_{ij} [A_{ij} - \frac{k_i k_j}{2m}] \delta(c_i, c_j)$$

where

- $A_{ij}$ represents the edge weight between nodes $i$ and $j$,
- $k_i$ and $k_j$ are the sum of the weights of the edges attached to nodes $i$ and $j$ respectively,
2.2 The Louvain Algorithm (as Baseline)

- $2m$ is the sum of all of the edge weights in the graph,
- $c_i$ and $c_j$ are the communities of the nodes and
- $\delta$ is a simple delta function

In order to maximize this value efficiently, the Louvain Method has two phases that are repeated iteratively.

**Phase 1**

First, each node in the network is assigned to its own community. Then for each node $i$, the change in modularity is calculated for removing $i$ from its own community and moving it into the community of each neighbor $j$ of $i$. This value is easily calculated by two steps: (1) removing $i$ from its original community, and (2) inserting $i$ to the community of $j$. The two equations are quite similar, and the equation for step (2) is:

$$\Delta Q = \left[ \frac{\Sigma_{in} + 2k_{i,in}}{2m} - \left( \frac{\Sigma_{tot} + k_i}{2m} \right)^2 \right] - \left[ \frac{\Sigma_{in}}{2m} - \left( \frac{\Sigma_{tot}}{2m} \right)^2 - \left( \frac{k_i}{2m} \right)^2 \right]$$

Where $\Sigma_{in}$ is sum of all the weights of the links inside the community $i$ is moving into, $\Sigma_{tot}$ is the sum of all the weights of the links to nodes in the community $i$ is moving into, $k_i$ is the weighted degree of $i$, $k_{in}$ is the sum of the weights of the links between $i$ and other nodes in the community that $i$ is moving into, and $m$ is the sum of the weights of all links in the network. Then, once this value is calculated for all communities $i$ is connected to, $i$ is placed into the community that resulted in the greatest modularity increase. If no increase is possible, $i$ remains in its original community. This process is applied repeatedly and sequentially to all nodes until no modularity increase can occur. Once this local maximum of modularity is hit, the first phase has ended.

**Phase 2**

In the second phase of the algorithm, it groups all of the nodes in the same community and builds a new network where nodes are the communities from the previous phase. Any links between nodes of the same community are now represented by self-loops on the new community node and links from multiple nodes in the same community to a node in a different community are represented by weighted edges between communities. Once the new network is created, the second phase has ended and the first phase can be re-applied to the new network.
2.3 The pKwik Algorithm

The pKwik algorithm is a simple algorithm proposed by the authors of [11] and is used exclusively for uncertain graphs, unless it is modified properly so as to be used in deterministic weighted graphs (where initially inspired from). The latter, however, is not our case.

The pKwikCluster algorithm was originally proposed by Ailon et al. [1] as KwikCluster algorithm for the weighted correlation clustering problem, as described previously. The researchers of [11] adopted their concepts and subsequently proposed the pKwikCluster algorithms or as we will simply call it pKwik algorithm exploiting some notions from the prerequisites of this chapter and specifically the notion of edit distance, to be used in the field of uncertain graphs.

In general terms, the algorithm starts with a random node $u$ and creates a cluster with all neighbors of $u$ that are connected with $u$ with probability greater that $1/2$. If no such node exists, $u$ defines a singleton cluster. Having $u$ and all of its neighbors the algorithm iteratively proceeds with the rest of the graph until there is no node left.

In order to be more explanatory regarding the design of the algorithm, we have extracted its implementation in pseudo-code, so as the reader would be more comfortable of judging among author’s implementation of their various proposed algorithms.

```
Algorithm pKwikCluster(G)
Input: Uncertain Graph G
Output: List of Clusters L_c = { C_1, C_2, ..., C_k }

1. i = 0
2. Initialize auxiliary List of marked (considered removed) nodes L_m
3. While there are non-marked nodes in G (e.g. G - L_m is not empty) do:
4. Pick randomly a node u from G - L_m
5. i = i + 1
6. Create a List L_u with all the neighbors of u that have not been included in L_m
7. If L_u is empty then
8. Create a singleton Cluster C_i with u
9. Add u into L_m
10. Else
11. Create an empty Cluster C_i
12. Foreach n in L_u do:
13. If Pr(u,n) of edge (u,n) > 1/2
14. Add n into C_i
15. End if
16. Add n into L_m
17. End foreach
18. L_m = L_m U {u}
19. End if
20. Add C_i into L_c
21. End while
```

Figure 2.1 Pseudocode of the pKwikCluster algorithm as proposed by the researchers and is going to be implemented in this dissertation using Apache Spark.
**pKwik Pseudocode Description**

The input to the pKwik algorithm is the whole uncertain graph $G$ and its output is a list of node lists $L_C$. There are no auxiliary variables as we will in next algorithms implementations.

As an initial step, the list of node lists is initialized. Then, a while-loop is used to describe the iterative procedure we talked about earlier. However, we could have used a do-while loop since if we get as input a single-node uncertain graph (not a realistic scenario) the iteration would happen at least once. We generalize more by utilizing a while-loop since we can consider as input an empty set ($G = \emptyset$). This discussion is quite trivial and we will not further analyze due to obvious reasons.

When the block of the while-loop is executed, a random node is picked and soon after its neighbors are collected based on the condition that are already present in the current iteration, that is they have not been removed in a previous one. So, if this list of neighbors is empty a singleton cluster must be created, and otherwise a non-singleton be created by incorporating all nodes of random nodes’ neighborhood having an edge with it, with a probability greater than the parameter we have beforehand defined (the alpha parameter which in the presented case equals with $1/2$).

In the end of the loop, and when the cluster (node list) has been populated with nodes, the initial random node picked is also incorporated into this cluster. Those nodes are then removed (or marked as visited) in order not to be considered in next iteration. Finally, the cluster is appended into the list of clusters to produce the output.

As the authors point out, this algorithm is a expected 5-approximation of the solution of the clustering problem for uncertain graphs. Furthermore, as we can intuitively judge, the worst running time scenario is linear, that is $O(m)$ where $m$ is the size of the input ($m = \binom{|V|}{2}$ and directly proportional to the number of edges). Even for deterministic graphs, this algorithm can theoretically achieve the very same approximation.

### 2.4 The Furthest Algorithm

The Furthest algorithm is the second algorithm authors of [11] proposed in their respective paper and was not implemented as part of this dissertation. This algorithm is a bit more advanced than the previous one and computes a better clustering result since it makes more passes to the input data, which is the whole uncertain graph.

The Furthest algorithm uses the notion of centers to partition the probabilistic graph $G$ in a top-down fashion. The algorithm starts by placing all nodes into a single cluster. Then, it finds the pair of nodes that have the smallest probability of having an edge between them.
These two nodes become the centers of the two new clusters. The remaining nodes are assigned to the center with which it is more probable to share an edge.

This procedure is repeated iteratively. At each step we select as center the node that is not a center and has the maximum distance (i.e., minimum probability) to the current centers. The distance of a node from the current centers is defined as the maximum among the probabilities of having an edge with the current centers. The nodes are then assigned to the center to which they are connected with maximum probability. At the end of iteration \( i \), the cluster graph \( C_i \) is formed which has cost \( D(G, C_i) \). If \( D(G, C_i) < D(G, C_{i-1}) \) then the algorithm proceeds to iteration \((i + 1)\), otherwise it stops and outputs \( C_{i-1} \).

**Furthest Pseudocode Description**

The input to the Furthest algorithm is considered again to be the whole uncertain graph \( G \) and its output a list of clusters (node lists). In this case, there are a few auxiliary variables used in the computation logic. Namely, we are referring to the \( old, ost \) and \( new, ost \) variables which designate whether the while (we will see next) is going to be repeated in terms of edit distance cost, as described earlier in this chapter. There is also the auxiliary list of centers which keeps in each iteration the updated set of centers considered.

The implementation consist of two steps; a preliminary one and an iterative one. In the first step, we consider \( G \) to be a single big cluster, we want to decompose, and we initialize two empty clusters. Later, we look up the pair of nodes that have the smallest edge probability between those two, and we consider them as the two created centers of this step. We add them to the auxiliary variable \( centerList \). Finally, in this step, we scan all the nodes in \( G \) and computer the Dijkstra distance with these two centers and assign each node to the cluster of which this distance is smaller, as the if-condition suggests. We output the intermediate cluster graph \( C' \).

In the second step, things get a bit more complicated. We initialize the auxiliary \( old, ost \) variable to find out the so far edit distance cost. We enter the while-loop where two sub-procedures are placed. The first one B1 is for finding a new center and the second one B2 for re-assigning the nodes to the new center if necessary.

The logic for finding maximums and minimums is based on an exhaustive plan of scanning. For each node and each center we compute distances and that is why the Furthest algorithm is costly. When B2 exits, we re-compute the edit distance cost to check if necessary to repeat the loop by swapping the auxiliary variables related with cost.

As the authors suggest, the Furthest algorithm needs to compute the distance between every node with the selected cluster centers. If \( k \) clusters are formed in the end, the worst-case running time of the Furthest algorithm is \( O(nk^2) \).
Algorithm Furthest(G, C)
Input: Global Ucinman Graph G, C subgraph (cluster) from G
Output: Global List of Clusters C = \{ C_1, C_2, ..., C_J \}
Auxiliary Parameters: Long old_cost = 0, Long new_cost = 0
Auxiliary List of Centers: Global center_list (initially empty)

1. A. Perform initial step
2. Consider entire G as a single cluster
3. Initialize empty clusters C_1 and C_2
4. Find pair of (v, u) nodes with the smallest probability in G
5. Consider k_u = u and k_v = v as the centers of C_1 and C_2
6. Add k_u into C_1
7. Add k_v into C_2
8. Add k_u and k_v into center_list
9. For each u in C
10. d_1 = Compute distance u’ from k_u (e.g. from weighted Dijkstra)
11. d_2 = Compute distance u’ from k_v (e.g. from weighted Dijkstra)
12. If d_1 < d_2
13. Assign u’ into C_1
14. Else
15. Assign u’ into C_2
16. End if
17. End for each
18. Output subgraph C*
19. End of sub-procedure A.

1. B. Perform iterative operation
2. old_cost = Compute edk distance cost of initial step procedure output C*
3. While new_cost < old_cost do
4. B1. Find new centers
5. Foreach u in G do
6. Foreach center c in center_list do
7. max_distance = Compute maximum distance between u and c (Dijkstra algorithm)
8. Keep maximum distance from c and equivalent node u
9. End foreach
10. Promote u as new center in the center_list
11. B2. Adjust nodes according to updated center list
12. Foreach u in G do
13. Foreach center c in center_list do
14. min_distance = Compute minimum distance between u and c (inverse Dijkstra algorithm)
15. Keep minimum distance and assign node u to current center c
16. End foreach
17. End foreach
18. new_cost = Compute edk distance cost of so far output C*
19. Assign old_cost = new_cost
20. End While

Figure 2.2 Pseudocode of the Furthest algorithm as proposed by the researchers and is going to be implemented in this dissertation using Apache Spark.
2.5 The Agglomerative Algorithm

The third algorithm, which is proposed by the authors of [11], is not going to be implemented in the technical part of this dissertation. However, it is presented in this section for complementary grounds.

The Agglomerative algorithm is a bottom-up procedure for the clustering problem. It starts by placing every node of the probabilistic graph \( G = (V, P) \) into a singleton cluster. At every iteration \( i \), it constructs the cluster graph \( C_i \) that merges the pair of clusters that appeared in \( C_{i-1} \) that have the largest average edge probability. If the average probability of the closest pair of clusters is more than \( 1/2 \), then the two clusters are merged into a single cluster. Otherwise the algorithm stops and outputs the current clustering.

We will only present the pseudo-code, since it is not implemented in this dissertation.

---

**Algorithm Agglomerative\( (G) \)**

Input: Uncertain Graph \( G \)

Output: List of Clusters \( L_c = \{ C_0, C_1, ..., C_k \} \)

1. Construct pair clusters
   1. Consider each node \( u \) of \( G \) as singleton cluster
   2. Traverse \( G \) in a BFS fashion (Breadth First Search)
   3. a. Retrieve all neighbors of node \( u \) as List \( L_u \)
      b. Foreach \( v \) in \( L_u \) do:
          c. If \( v \) not included in any cluster yet (not in \( L_c \)) AND \( Pr[u, v] = \max \)
          d. Create a node pair cluster \( C_i \) with \( u \) and \( v \)
          e. Add \( C_i \) into \( L_c \)
          f. End if
          g. End foreach

2. Merge pairs into bigger clusters
   4. Foreach \( C_i \) in \( L_c \) do:
   5.   Foreach \( C_j \) in \( L_c \) do:
   6.       Calculate \( \text{avg} = \text{AverageEdgeProbability}(C_i, C_j) \)
   7.       If \( \text{avg} \) is max AND \( \text{avg} > \frac{1}{2} \)
   8.           \( C_i = \text{Merge}(C_i, C_j) \)
   9.       Add \( C_i \) into \( L_c \)
   10.      Remove \( C_i \) and \( C_j \) from \( L_c \)
   11.     End if
   12.    End foreach
   13.  End foreach

---

Figure 2.3 Pseudocode of the Agglomerative algorithm as proposed by the researchers and is NOT going to be implemented in this dissertation using Apache Spark.

A naive implementation of the Agglomerative algorithm requires \( O(kn^2) \) time, where \( k \) is the number of clusters in the output. However, using a heap for retrieving the closest pair of clusters reduces the time complexity of the algorithm to \( O(kn \log n) \).
2.6 Other State-Of-The-Art Algorithms

We are going to mention two remarkable papers in the field of uncertain graphs clustering that have drawn our attention.

**Reliability Clustering Algorithm**

The first one [14], exploits the concept of reliability and is inspired by the general field of Information Theory. Researchers of [14] used this concept to produce reliable clusters which are not likely to be disconnected in the context of different instantiations of the uncertain. Connectivity is a fundamental graph property and plays an important role in graph clustering. From an information-theoretic perspective reliability is an extension of deterministic graphs connectivity graphs to uncertain ones, as they point out. The general implementation of their algorithms and their experimental procedure can be found in the references.

**Connection Probability Algorithm**

In the second one [3], the authors contend that given an uncertain graph $G$ and an integer $k$, can partition its nodes into $k$ clusters, each featuring a distinguished center node, so to maximize the minimum/average connection probability of any node to its cluster’s center, in a random possible world. To achieve the afore-mentioned, they adopt the connection probability between two nodes (a.k.a. two-terminal reliability), that is, the probability that the two nodes belong to the same connected component in a random possible world, as the distance measure upon which they will base their clustering. As a first technical contribution, which may be of independent interest for the broader area of network reliability, they show that this measure satisfies a form of triangle inequality, unlike other distance measures used in previous works. This property allows them to cast the problem of clustering uncertain graphs into the same framework as traditional clustering approaches on metric spaces, while still enabling an effective integration with the possible world semantics that must be taken into account in the estimation of the connection probability. Their implementation and experimental procedure can also be found in the references.
Chapter 3

Experimental Study

3.1 Background

The experiments were conducted in a 80-core mainframe computer with 1TB of RAM. The software environment featured an Ubuntu Linux 14.04 LTS operating system, Apache Spark version 2.4.0, Sbt version 1.1.2 and Scala version 2.11.8.

It is crucial to mention at this point, that the experiments were conducted in a shared fashion which means that many users were competing for the same resources and due lack of user privileges it was not possible to give the highest priority to the Java processes spawned during the execution (and furthermore it was not polite to do so). Having said so, the performance results can be a little biased and rather indicative of the actual algorithms performance but surely worth taking into consideration to gain some perspective.

3.2 Dataset Description

For the purpose of conducting experiments with the implemented algorithms we use the so-called CORE dataset. This dataset is also called the core interaction network and is provided by Krogan et al. in their respective reference [12]. The network consists of 2708 nodes that correspond to proteins and there are 7123 protein interactions which are actually the edges of the network. The edge probabilities encode the confidence of the interaction between nodes. We will refer to this network as CORE and we denote the underlying probabilistic graph as $G_{core}$. There is no edge in the network with probability less than 0.27. Also, about 20% of the edges have probability greater than 0.98. The edge probabilities are uniformly spread in the remaining range $[0.27,0.98]$. The dataset exhibits power-law degree distribution, short paths, and high clustering coefficient.
### 3.3 Time Performance

#### 3.3.1 Graph Size vs. Running Time

Before going into details about the diagram depicted below, it is necessary that we clarify some things about the settings we configured in the running environment to conduct the experiment.

First of all, we chose 4 as the number of cores in the mainframe and split the graph by bearing in mind the probability of existence each edge of the graph held. Our strategy goes like this: when it is depicted that the graph size is 0.3, it means that 30% of the bigger probability edges where kept. So, we iteratively run the algorithm (using a wrapper shell script) by initially keeping the strong core of it, that is the edges with bigger probability edges and later started taking into account the smaller ones as graph size percentage increased. For the pKwik algorithm the alpha parameter was set to 0.5 (despite this might not be the best chosen value). The results are quite intuitive and expected expect a minor case demonstrated for the Louvain algorithm.

![Running Time vs. Graph Size](image)

**Figure 3.1** Running time of pKwik and Louvain algorithms against the graph size as described previously. (Produced using SciDAVis)

As we observe in the figure 3.1, and regarding the pKwik algorithm, as the graph size increases the running time also increases in the fashion depicted. The monotonicity of the relation between graph size and running time is always positive, or in other words we do not upside-downs in the diagram, which was more or less expected. The smallest running time
when the graph size is minimum (10%) is two orders of magnitude smaller that the biggest one when the graph size is maximum (100%).

As regards the Louvain algorithm, it seems that its running time performance is irrespective of its size input. We can clearly see in the diagram that when graph size is scaled up, running time does not significantly variates its value. Perhaps, this algorithm, which uses the concept of modularity, is mainly designed for deterministic graphs instead of uncertain ones in contrast with the pKwik algorithm which was originally designed for the second case and that is why outperforms the latter in scaling issues.

3.3.2 Running Time vs. Number of Cores

Now, at this point, a very interesting investigation to make is see how the running time (or response time) variates when the number of cores changes. To obtain parallelism the number of partitions in the edges RDD of GraphX was three times bigger than the number of cores and the number of partitions of the nodes RDD was stable 2. We increased at each step the number of cores by a power of two (1,2,4, ..., 64) and recorded the running time. The following two diagrams occurred; one for the pKwik algorithm and one for the baseline louvain algorithm.

In the figure 3.2, and for pKwik algorithm, we can observe a descending curve of running time as the number of cores increases. The biggest running time (around 3700 seconds) occurs there is a single core executing the algorithm, that is when there is no parallelism. Another interesting point to mention would be that 16 cores to execute the algorithm is enough since there is no significant differentiation for 32 and 64 cores, running times are essentially the same (around 900 seconds).
In the second figure 3.3, the one for the louvain algorithm, we can also observe the descending pattern of the curve. The partitioning of the scheme did not change as for the pKwik algorithm. The biggest running time (52000 seconds) occurred for no parallelism at all and as pKwik there is no significant differentiation for more than 16 cores. However, the running times for this baseline algorithms are one order of magnitude bigger than those of pKwik. This is expected since the time complexity of the louvain algorithms is bigger.
3.4 Validity Measures

3.3.3 Number of Clusters vs. Alpha Parameter (pKwik)

In this subsection, we are going to make a very interesting investigation on the pKwik algorithm which has to do with the number of clusters it produces in respect with its input parameter alpha according to the given CORE dataset.

We can clearly observe in the figure 3.4, that the number of cluster this algorithm produces is not stable and variates significantly when alpha parameter changes its value. We can also see that the maximum number of clusters is achieved around the value alpha = 0.2, where we slightly more than 150 cluster in the output. We have one more time to mention than probabilities below 0.27 are missing from the CORE dataset.

![Figure 3.4 Number of clusters produced by pKwik algorithm by variating the alpha input parameter. (Produced using SciDAVis)](image_url)

By further increasing the value of alpha parameter over the value of 0.2, we can conclude that the number of clusters is constantly falling in a smooth way. When alpha parameter get the value of 1, we do not get any cluster since according to the algorithm description we introduced in the previous chapter, all the edges are removed along with their corresponding nodes.

3.4 Validity Measures

In order to validate and verify the clusterings obtained by the pKwik and louvain algorithms it is necessary to quantify their significance in statistical terms, that is the cluster graphs produced do not resemble other clusterings produced with random procedure, and obviously
compare clusterings among different procedures. For this purpose, we leverage two indexes; the well-known Rand index and the suitable for our case (graph data) Dunn-like index.

Rand index is a very intuitive approach to compare clusterings by counting pairs of objects (in our case nodes) that are "classified" in a similar way in two configurations; the cluster configuration our algorithm produced and the ground-truth or nearly best configuration which must be known in advance. It ranges between 0 and 1, and the bigger its value the better [21].

Dunn-like index, on the other hand, does not impose for the ground-truth to be known beforehand since its value is independent of it. It is based on relative criteria and does not involve statistical tests. The basic idea lies in the fact that the best clustering is chosen among a set of clustering schemes according to pre-specified criteria. Furthermore, dunn-like index tell us how well and compact clusters occur. We can identify good clusterings by a large number of this index that will indicate the presence of compactness and well-separation of individual clusters [8].

### 3.4.1 Rand Index of Clusterings - Accuracy

Having said so, we wrote special scripts in Scala/Spark to implement these indexes as described in literature [21]. We found that the Rand index of the clustering pKwik produced equals 0.6 approximately and 0.16 for clustering the baseline algorithm produced, that is the louvain algorithm.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Rand Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>pKwik</td>
<td>0.602</td>
</tr>
<tr>
<td>Louvain</td>
<td>0.159</td>
</tr>
</tbody>
</table>

We can clearly conclude that the clusters produced by the first algorithm are quite meaningful and worth taking into account when deploying application related with uncertain graph data. However, the clusters produced by the second algorithm are not quite important compare to the ground-truth clustering scheme since the value of the Rand index is far below 0.5.

The afore-mentioned are quite well-understood and expected because louvain algorithm is not an algorithm destined for uncertain graphs, but for deterministic graphs instead. Needless to say, furthers or agglomerative algorithms would give even more significant clusterings since they make more passes over the data (higher time complexity).
3.4.2 Dunn-like Index of Clusterings

Another meaningful index that is not based on ground-truth and worth taking into is, as we already said, the Dunn index. A special script in Scala/Spark was created to calculate it and the results for the two algorithms are as follows:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Dunn-like Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>pKwik</td>
<td>0.732</td>
</tr>
<tr>
<td>Louvain</td>
<td>0.401</td>
</tr>
</tbody>
</table>

We can see that the clusters produced by the pKwik algorithm and for the specific CORE dataset are quite compact and well-separated while this does not happen to the extent it happens for the baseline algorithm. The distances among nodes in the same cluster and among clusters where calculated using the naive approach described in the second chapter, that is we transformed probabilities into weights by means of the logarithmic formula described.
Chapter 4

Conclusions and Further Research

The conclusions drawn as a result of the experiments conducted in this dissertation can be interpreted and described, in a nutshell, in the following:

• **Graph size scaling**: The pKwik algorithm scales well when the size of the input graph increases and produced a result in a reasonable time, while the baseline Louvain algorithm seems to be irrespective of the input size since the running time does not variate.

• **Number of cores scaling**: The pKwik algorithm also scales up very well when the number of cores increases. It seems that 16 number of cores are adequate for parallelism to take place and reduce the running time of the algorithm. This is also the case for the baseline algorithm but the running time is completely scaled up orders of magnitude.

• **Validity of clusterings**: We inferred the validness of clustering by means of two indexes that had high values for the pKwik algorithms and low ones for the baseline algorithm. This is expected since pKwik is designed for uncertain graph while Louvain is not.

• **Other conclusions**: The number of clusters produced by pKwik is heavily dependent on the value of alpha input parameter and it seems a tuning is required and for a specified dataset when deploying applications of uncertain graphs. Moreover, although the Furthest algorithms was implemented its execution were not feasible to make a comparison.

As further research we would propose the taking into account different notions of distances in calculations in the different aspects of the algorithms and especially for the not-feasible-to-implement Furthest algorithms. We would also propose the usage of different
datasets other than protein-protein interaction datasets, social networks related datasets for example, and re-conduct the experiments.
Bibliography


