University of Pisa

Department of Computer Science
Ph.D. in Computer Science

Ph.D. Thesis

Distributed Graph Processing: Algorithms And Applications

Alessandro Lulli

Supervisor
Laura Ricci

Supervisor
Patrizio Dazzi

Abstract

Thinking Like A Vertex (TLAV) is a popular computational paradigm suitable to express many distributed and iterative graph algorithms. It has been adopted as base computational paradigm for many of the currently available distributed frameworks and endorsed by numerous industries and academias. Also, it has been exploited to define algorithms to extract useful information from the nowadays increasing production of data which can be modeled as graphs. These facts strengthen the idea that exploiting distributed frameworks for graph analysis is an hot topic of research. As a matter of fact, we found that a solution for several algorithms is not always available or state-of-art algorithms are unsatisfactory, under many points of view.

This thesis aims at providing guidelines for defining distributed graph algorithms structured according to TLAV and showing their applicability to real applications. We show how approximation, simplification and versatility can be combined to define novel distributed algorithms to improve the currently available solutions with the goal to enhance the functionalities of the algorithms. We show also how algorithms may be combined to define complex solutions and how they can be employed to solve relevant applications. In particular, we present novel algorithms for computing betweenness centrality, connected components and clustering. Such algorithms are exploited for Spam campaign detection, population estimation and hashtag centrality. To this end, we make use of real large dataset provided from our collaborations, Symantec for Spam emails, a large Italian Mobile Phone provider, mobile calls, and ISTI, CNR for a two years collection of real tweets from the Twitter social network.

Contents

List of Publications **xiii**

1 Introduction **1**
1.1 Issues affecting existing solutions 3
1.2 Thesis Contributions 5
1.2.1 Guidelines to define a TLAV algorithm 5
1.2.2 Proposed solutions 6
1.3 Outline of the Thesis 9

I Related Works **11**

2 The TLAV approach: frameworks and algorithms **13**
2.1 The Think Like A Vertex approach 13
2.1.1 Timing 14
2.1.2 Computation 15
2.1.3 Communication 16
2.1.4 Graph partitioning 16
2.2 Framework for graph processing 16
2.2.1 MapReduce 17
2.2.2 Pregel 19
2.2.3 GraphLab 22
2.2.4 PowerGraph 23
2.2.5 Apache Spark 25
2.3 Case studies 28
2.3.1 Graph Problems 28
2.3.2 PageRank 29
2.3.3 Connected Components: HASH-TO-MIN 32
2.4 Conclusion 33

II TLAV Algorithms **35**

3 Duckweed: Distributed Current Flow Betweenness Centrality **37**
3.1 Related work 38
3.2 Preliminaries 39
3.2.1 Random Walk Betweenness 39
3.2.2 Current Flow Betweenness 40
3.3 Duckweed 41
3.3.1 Computational Model 41
3.3.2 The DUCKWEED approach 42
3.3.3 Flow computation 42
3.3.4 Centrality computation 45

3.4 Experimental evaluation . . . 46
3.4.1 Evaluation of Correlation . . . 46
3.4.2 Convergence Time . . . 48
3.4.3 Centrality on Large Graphs . . . 48
3.4.4 Message Volume . . . 50
3.4.5 Flow Creation Strategy . . . 51
3.5 Duckweed on Apache Spark: the algorithm . . . 52
3.5.1 Reducing the number of messages . . . 52
3.5.2 Reducing the size of the messages . . . 53
3.5.3 Flow Generation . . . 53
3.5.4 Termination . . . 53
3.5.5 Implementation Details . . . 54
3.6 Experimental evaluation on Apache Spark . . . 54
3.7 Conclusion . . . 56

4 Connected Components: Cracker 57
4.1 Related Work . . . 58
4.2 The CRACKER Algorithm . . . 61
4.2.1 Seed Identification . . . 62
4.2.2 Seed propagation . . . 64
4.2.3 Cracker correctness . . . 64
4.2.4 Cracker computational cost . . . 66
4.2.5 Optimisations to CRACKER . . . 69
4.2.6 Implementation . . . 71
4.3 Experimental Evaluation . . . 71
4.3.1 Dataset Description . . . 72
4.3.2 Evaluation of Optimizations . . . 72
4.3.3 Comparison with the State of the Art . . . 75
4.3.4 Scalability . . . 78
4.4 Conclusion . . . 78

5 Clustering 81
5.1 Related Work . . . 82
5.2 k-NN based clustering . . . 83
5.2.1 Phase 1: k-NN Graph Construction . . . 83
5.2.2 k-NN Graph Pruning . . . 84
5.2.3 Phase 2: Connected Components . . . 85
5.3 Experimental Setup . . . 85
5.4 Results . . . 87
5.5 Conclusion . . . 90

6 Density Based Clustering 91
6.1 Background and Related Work . . . 92
6.1.1 The DBSCAN Algorithm . . . 92
6.1.2 Distributed Density-Based Clustering . . . 93
6.1.3 Graph-Based Clustering . . . 94
6.1.4 Density-Based Clustering for High-Dimensional Data . 94
6.2 NG-DBSCAN: Approximate and Flexible DBSCAN . . . 94
6.2.1 Overview . . . 95
6.2.2 Phase 1: Building the ε-Graph . . . 96
6.2.3 Phase 2: Discovering Dense Regions . . . 100

6.3 Experimental Setup . . . 103
6.3.1 Experimental Platform . . . 103
6.3.2 Evaluation Metrics . . . 103
6.3.3 The Datasets . . . 104
6.3.4 Alternative Approaches . . . 104
6.4 Results . . . 105
6.4.1 Analysis of the Parameter Space . . . 105
6.4.2 Performance in a 2D Space . . . 108
6.4.3 Performance in $\boldsymbol{d}$-Dimensional Spaces . . . 110
6.4.4 Performance with Text Data . . . 111
6.4.5 Scalability . . . 113
6.4.6 Discussion . . . 114
6.5 Conclusion . . . 115

III Applications 117

7 Spam campaign analysis 119
7.1 Our approach . . . 119
7.2 Data Description . . . 120
7.3 Results . . . 121
7.4 Conclusion . . . 123

8 Population estimation 125
8.1 Related Work . . . 125
8.1.1 Mobile phones data analysis . . . 125
8.1.2 Scalable clustering algorithms . . . 127
8.2 Data description . . . 127
8.3 Preliminaries: Muchness analytical process . . . 129
8.4 From Muchness to Muchness+: a framework for census . . . 130
8.4.1 Improving the clustering algorithm . . . 131
8.4.2 Adapt the metric to the data instead of the algorithm 133
8.5 Experimental evaluation . . . 136
8.5.1 How to configure Muchness+ . . . 136
8.5.2 Road to Muchness+: evaluating optimizations . . . 137
8.5.3 Studying individuals mobility . . . 138
8.5.4 Comparing with competitors and census data . . . 140
8.5.5 Evaluating individual profiles . . . 141
8.5.6 Scalability . . . 143
8.6 Conclusions . . . 143

9 Hashtag centrality 145
9.1 Related works . . . 145
9.2 Data Description . . . 146
9.3 Our Approach . . . 146
9.3.1 Graph Construction . . . 147
9.3.2 Centrality Computation . . . 147
9.4 Validation . . . 148
9.4.1 Identification of cyclic events . . . 148
9.4.2 Identification of seasons . . . 148

9.5 Conclusion 150

10 Conclusion 151

10.1 Discussion 152

10.2 Future Works 152

10.2.1 Simplifications Techniques 152

10.2.2 From TLAV to Thinking Like a Sub-Graph 153

10.2.3 Algorithms on dynamic graphs 153

10.2.4 Improve Frameworks: Parallelism and Compression . . 153

Bibliography 154

List of Figures

1.1 Thesis workflow 6

2.1 Synchronous and Asynchronous computation. 15
2.2 An overview of an execution in MapReduce. 18
2.3 Pregel's superstep as BSP implementation 19
2.4 Superstep: an example 20
2.5 Differences between vertex cut and edges cuts. 24
2.6 Spark Architecture 24
2.7 An RDD of Strings partitioned on 4 machines. 26
2.8 Spark Architecture 27
2.9 Spark Example 27

3.1 Evolution of a flow from s to t over time 42
3.2 Kendall Tau Correlation with NetworkX 47
3.3 Kendall Tau Correlation: complete vs approximated 48
3.4 Convergence Time 49
3.5 CC Number (large number is better) 49
3.6 CC Number: Graph Cut While Increasing Duckweed approximation 50
3.7 Message Volume Evaluation 51
3.8 Evaluation of the approximation. Kendall tau correlation with NetworkX. 55
3.9 Comparison with competitors and DUCKWEED_X, DUCKWEED_MP. 55

4.1 CRACKER: example of seed identification 61
4.2 CRACKER seed propagation tree 64
4.3 Illustration of algorithm CRK on the same graph in Figure 4.1 66
4.4 Seed identification with the EP and OS optimizations. 69
4.5 Evaluation of Optimizations 73
4.6 Sensitivity to Diameter 74
4.7 Sensitivity to Vertices Number 75
4.8 Step By Step Comparison 75
4.9 Graph Topology Evolution 76
4.10 Scalability evaluation 78

5.1 Impact of the pruning phase threshold θ. Impact of number of *iterations* for the k-NN graph construction phase of our algorithm. Clustering quality in terms of clustering Silhouette for the *full* Symantec dataset. 85
5.2 Twitter Dataset: Clustering quality in terms of inter and intra cluster similarity, and clustering Silhouette. As for the Symantec dataset, the algorithm proposed in this work outperforms the baseline method in all metrics. 88
5.3 Scalability. Our approach scales roughly linearly. 89

6.1 Phase 1: ε-graph construction. 97
6.2 Phase 2 – dense region discovery. 101
6.3 Analysis of the termination mechanism. 106
6.4 Analysis of the Parameter Space. 107
6.5 Synthetic datasets plot. 110
6.6 Performance in a 2D space: Scalability. 110
6.7 Performance in a d-dimensional space. 111
6.8 Evaluation of the scalability. 113

7.1 Illustration of our approach 120
7.2 Symantec dataset: feature distribution and average similarity 121
7.3 Symantec Dataset: Clustering quality in terms of inter and intra cluster similarity, and clustering Silhouette. The algorithm proposed in this work outperforms the baseline method, and is only marginally affected by approximation quality. . . 122
7.4 Clustering quality in terms of the unique number of features in each cluster output by clustering algorithms. For manual inspection to be useful, a small number of unique feature, such as bot-nets or time-frame of each cluster item, is preferred. Our approach outperforms the baseline algorithm. 123

8.1 Individuals perform calls under a given cell (relative to a municipality). Each call is collected from the Telco operators. We receive such data and we aggregate the calls of each individual creating a user calling profile (ICP). 128
8.2 Amount of calls in the dataset. Darker colors represent an higher amount of calls collected in the dataset. 128
8.3 In-coming and out-coming workers for the Pisa municipality . 129
8.4 Muchness analytical process. A : for each individual we assign an ICP. B : each ICP becomes a node in a graph. C : we search for similar nodes and at the end we prune low similarity edges (dashed). D : we search for connected components and we identify outliers (node 2). E : for each cluster we define an exemplar (icons) classified as Resident, Commuter or Visitor. 130
8.5 How to configure Muchness+: analysing the sampling parameter (ρ) 137
8.6 Studying individuals mobility: Individuals travelling from home to work 139
8.7 Studying individuals mobility: Individuals travelling from home to visit places 140
8.8 Comparing with competitors and census data 141
8.9 Evaluating individual profiles 142
8.10 Scalability evaluation 143

9.1 How to construct a co-occurrence graph from raw tweets . . . 147
9.2 Identification of cyclic events 148
9.3 Identification of periods 149
9.4 Identification of seasons 149

List of Tables

1.1 TLAV algorithms developed in this thesis 5

2.1 Implementing the TLAV model using RDD. In the example one RDD composed of two partitions. Each element has a key, the identifier, and a value, with the adjacency list of the vertex. 25

3.1 Comparison of the flow creation strategies for the Random graph (in terms of CCs created) 52
3.2 Execution time on real world graphs. 56

4.1 State-of-the-art algorithms 58
4.2 Datasets description 71
4.3 Performances with real world datasets: **message number** and **message volume** are values $\times 10^6$ 74

5.1 Twitter Dataset: Manual Investigation 88
5.2 Symantec dataset: breakdown of the algorithm runtime (in seconds) 89
5.3 Symantec dataset: K-means, baseline algorithm runtime (in seconds) 89

6.1 Overview of parallel density-based clustering algorithms. . . . 93
6.2 How to set ρ. 108
6.3 Performance in a 2D space: Clustering Quality. 109
6.4 Spam and Tweets dataset: manual investigation. 111
6.5 Evaluation using text data: Twitter and Spam datasets comparison with k-MEANS. "C" stands for compactness and "S" for separation. 112
6.6 Distance function comparison for Twitter. 112

7.1 Symantec Dataset: characteristics 120
7.2 Symantec Dataset: Manual Investigation 122

8.1 Overview of frameworks to estimate population 127
8.2 Similar ICPs extracted by expertises. A comparison of similarity values using: EUC, JAC and EUC+JAC 134
8.3 Road to Muchness+: adaptive k optimization 137
8.4 Road to Muchness+: early termination 138
8.5 Muchness+ vs Muchness 138
8.6 Comparing with competitors and census data: Median estimation errors 140
8.7 Evaluating individual profiles: exemplars 143

List of Publications

International Journals

J003 A. Lulli, M. Dell'Amico, P. Michiardi, and L. Ricci. **NG-DBSCAN: a Scalable, Approximate Density-Based Clustering Algorithm for Arbitrary Similarity Metrics**. *Proceedings of the VLDB Endowment*, 10(3), 2016 (to appear)

J002 A. Lulli, L. Ricci, E. Carlini, P. Dazzi, and C. Lucchese. **Fast Connected Components Computation in Large Graphs by Vertex Pruning**. *IEEE Transactions on Parallel and Distributed systems*, 22(6):931–945, 2016 (to appear)

J001 E. Carlini, A. Lulli, and L. Ricci. **Dragon: Multidimensional range queries on distributed aggregation trees**. *Future Generation Comp. Syst.*, 55:101–115, 2016

International Conferences

I have been the presenter author for [C003,C004,C006].

C008 A. Kavalionak, E. Carlini, A. Lulli, G. Amato, C. Gennaro, C. Meghini, and L. Ricci. **A prediction-based distributed tracking protocol for video surveillance**. In *Networking, Sensing and Control (ICNSC), 2017 IEEE International Conference on*. IEEE, 2017. submitted

C007 M. Bertolucci, A. Lulli, and L. Ricci. **Current flow betweenness centrality with Apache Spark**. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2016 (to appear)

C006 A. Lulli, L. Gabrielli, P. Dazzi, M. Dell'Amico, P. Michiardi, M. Nanni, and L. Ricci. **Improving Population Estimation From Mobile Calls: a Clustering Approach**. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 1097–1102. IEEE, 2016

C005 E. Carlini, P. Dazzi, A. Lulli, and L. Ricci. **Distributed graph processing: an approach based on overlay composition**. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, pages 1912–1917, 2016

C004 A. Lulli, T. Debatty, M. Dell'Amico, P. Michiardi, and L. Ricci. **Scalable k-NN based text clustering**. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 958–963, 2015

C003 A. Lulli, L. Ricci, E. Carlini, P. Dazzi, and C. Lucchese. **Cracker: Crumbling large graphs into connected components**. In *2015 IEEE Symposium on Computers and Communication, ISCC 2015, Larnaca, Cyprus, July 6-9, 2015*, pages 574–581, 2015

C002 M. Bertolucci, E. Carlini, P. Dazzi, A. Lulli, and L. Ricci. **Static and Dynamic Big Data Partitioning on Apache Spark**. In *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, 1-4 September 2015, Edinburgh, Scotland, UK*, pages 489–498, 2015

C001 A. Lulli, L. Ricci, E. Carlini, and P. Dazzi. **Distributed Current Flow Betweenness Centrality**. In *2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems, Cambridge, MA, USA, September 21-25, 2015*, pages 71–80, 2015

International Workshops

I have been the presenter author for [W001,W002,W003].

W004 E. Carlini, A. Lulli, and L. Ricci. **TRACE: generating traces from mobility models for Distributed Virtual Environments**. In *Euro-Par 2016: Parallel Processing Workshops - Euro-Par 2016 International Workshops, Grenoble, France, August 22-25, 2016, Revised Selected Papers, Part I*, pages 129–140, 2016 (to appear)

W003 E. Carlini, P. Dazzi, M. Mordacchini, A. Lulli, and L. Ricci. **Community Discovery for Interest Management in DVEs: A Case Study**. In *Euro-Par 2015: Parallel Processing Workshops - Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers*, pages 273–285, 2015

W002 A. Lulli, P. Dazzi, L. Ricci, and E. Carlini. **A Multi-layer Framework for Graph Processing via Overlay Composition**. In *Euro-Par 2015: Parallel Processing Workshops - Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers*, pages 515–527, 2015

W001 E. Carlini, P. Dazzi, A. Esposito, A. Lulli, and L. Ricci. **Balanced Graph Partitioning with Apache Spark**. In *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part I*, pages 129–140, 2014

Technical Reports

T001 E. Carlini, P. Dazzi, A. Lulli, and L. Ricci. **Layered Thinking in Vertex Centric Computations**. *ERCIM News*, 2015(102), 2015

Submitted papers

S003 E. Carlini, A. Lulli, and L. Ricci. **TRACE: Generation and Analysis of Mobility Traces for Distributed Virtual Environments**. *Concurrency and Computation: Practice and Experience*, 2016. submitted

S002 A. Lulli, E. Carlini, P. Dazzi, and L. Ricci. **TELOS: An Approach for Distributed Graph Processing Based on Overlay Composition**. *Scalable Computing: Practice and Experience*, 2016. submitted

S001 A. Lulli, L. Gabrielli, P. Dazzi, M. Dell'Amico, P. Michiardi, M. Nanni, and L. Ricci. **Scalable and flexible clustering solutions for mobile phone based population indicators**. *International Journal of Data Science and Analytics*, 2016. submitted

Chapter 1

Introduction

The graph abstraction has been largely used since the 18th century, both in mathematics and science in general. The pervasive usage of graphs has been motivated by the fact that their structure lets to formally identify the relationships between objects. In the past century, as soon as computer science was born, many efforts were focused on identifying and defining efficient solutions to detect properties and find algorithms for graphs. For instance, in 1959 Dijkstra [58] published an algorithm to find the Shortest Paths on a graph. This algorithm is still taught in universities and used in many application domains, such as the optimization of telecommunication networks [69].

In recent years, the interest in graphs is getting a renovated momentum and graphs started to be used also in a different manner. In fact, if in the early ages graphs have been exploited as a common model to visualize and see relationships, nowadays, although graph visualization is still an important branch of research, graphs are used to model complex relationships between different kind of conceptual entities with the objective of extracting valuable information from the continuously increasing amount of data. This is motivated by the fact that, from the birth of Internet and even more with Web 2.0 and Social Networks, the amount of information started to increase with an unprecedented pace. In particular, in the last years, with the proliferation of different kinds of computational resources, ranging from computers and IoT devices to mobile phones in everyday life, data has been produced in many manners and different extents. For instance, users of Internet, generate contents for web pages, post comments and likes on social networks, perform calls and send messages with mobile phones and countless more.

A significant portion of such data can be modeled using graphs. For instance, the knowledge of social relationships in a social network dramatically increases the effectiveness of proposing new products and services tailored for users in the form of ad-hoc advertisements and search results [191]. Graphs representing links between web pages have been successfully exploited to develop ranking methods (e.g. PageRank [142]) to improve decision making. Searching for vulnerability in road networks is used to minimize risks [55], and many more.

One common trait of these examples is the large size of such graph data. For instance, a popular dataset containing the follower relationships in Twitter [101] has around 40 million nodes and 1.5 billion edges. Due to this, a popular way to perform computation on graphs makes use of distributed environments to make possible or facilitate the computation of large graphs. One of the most popular forerunner of this kind of computation is MapReduce framework proposed by Google [51]. This work has lead the way to many more contributions in the field of distributed computation. Although the first distributed frameworks were general purpose and not specialized for graph computation, more recently many frameworks for distributed graph computation have been proposed. Also in this case, Google has been one of the first to propose a specialized framework called Pregel [124]. The scientific contributions on this field range from novel frameworks [112, 78, 193, 186, 157, 33] to optimizations that can be introduced to both facilitate their adoption and to reduce the computational overhead [184, 189, 46, 158]. The increasing amount of proposals of more and more specialized and optimized frameworks for graph computation reveals the increasing interest on distributed graph computation. In these frameworks the computation is partitioned on multiple machines and this requires the definition of suitable models of computation. The majority of these solutions started to adopt what is called the vertex-centric approach or Thinking Like A Vertex (TLAV) [131]. The idea is that a program must be executable from the point of view of a vertex of the graph. To be more precise, the program must rely only on the local knowledge of a vertex, i.e. its internal state and the state of its neighbors. Such program will be executed iteratively on each vertex of the graph until a stopping criteria is met.

The research in this context is manifold, ranging from frameworks improvements, to algorithms and applications and many more. For instance, most graph-based applications may be composed by a different set of modules, often interplaying in a complex way. Such tasks may be very simple, in the sense that just one task (algorithm) may be resulting in complex outcomes for an application. For instance, community discovery has been successfully used to perform viral marketing [104]. With a similar perspective, we can observe how triangle counting has been used to determine the presence of spamming activities in large-scale web graph [16].

On the other end, other applications require more complex chains of tasks to be solved. One common approach to deal with this kind of applications is to introduce decomposition, whose main idea is to start from a big task and to decompose it in smaller ones. Than, each small task is solved in isolation and finally they are composed to obtain the global solution.

This thesis aims at providing useful guidelines for supporting the design and implementation of distributed graph-based algorithms and their applicability on real problems. To this end, from the definition of algorithms following our guidelines, we finally concentrate on applications where our algorithms can be employed for providing the solution to real problems. Such algorithms can be *simple*, they solve just one predetermined task, or *composed*, they use a composition of simple ones collaborating for the achievement of a solution. We define and develop multiple chains of algorithms and we show how they can be applied to different applications. Coherently with the concepts and assumptions underpinning TLAV, our focus is on algorithms working on large graphs in a distributed environment.

1.1 Issues affecting existing solutions

In order to identify the existing issues on state-of-art solutions and algorithms, structured according to the TLAV approach, and to give our contributions to this field, we performed some preliminary studies on several distinct existing algorithms and applications exploiting the TLAV approach. These studies allowed us to identify some of the most critical issues characterizing these solutions.

We started analyzing how an algorithm defined according to the TLAV approach can be ported to a real framework implementing this paradigm. In our works, we make use of Apache Spark [193] as the reference architecture implementing TLAV. To this end, we study an algorithm called Ja-Be-Ja [148], performing *balanced k-way graph partitioning*. This algorithm is commonly undertaken to partition a graph on k machines. We identified a few notable issues that we addressed, allowing us to achieve good results when porting it on Apache Spark [35]. Also, we studied many popular algorithms such as *PageRank*, *Triangle Counting*, *Connected Components* and we gave a proof of the impact of a good partitioning strategy to reduce the execution time with respect to the most common strategy used in Apache Spark [20]. We also studied how to query a distributed environment to answer *multi attribute range queries* [39] and we evaluated and port different algorithms for *community discovery* [38]. Finally, to have a deeper knowledge on the mechanics of a framework supporting TLAV computation, we defined a layered architecture on a popular distributed frameworks which allows the definition of multiple graph views in the same algorithm [37, 115, 36].

These problems helped us to understand the importance of graph analysis and the issues to face when defining it. Due to this, we identified some guidelines (Section 1.2.1) that we think are important when defining or porting an algorithm to a distributed environment. Before introducing the contributions of this thesis in the following section, we summarize the main issues we identify in the current state-of-the-art solutions.

- **does already exist a solution?** Although graphs have been around for many years, some problems still miss a distributed definition and solution. For instance, when we studied centrality measures on graphs we found that the current techniques are mainly centralized and not easily portable to distributed environments. To overcome this lack of solutions, we propose an approach that exploits the analogy with electrical circuits to provide an elegant solution perfectly fitting the TLAV model.

- **is it always feasible to re-adapt existing algorithms in TLAV?** Porting sequential algorithms to a distributed context may be either a straightforward task or a really hard challenge. In fact, single machine and multiple machines environments have different characteristics and an algorithm must rely on different assumptions. For instance, it is not difficult to implement a graph traversal algorithm on a single machine, where it is possible to access in a uniform way each vertex, but it can be cumbersome when traversing means jumping the pointer from one machine to another one.

- **is an exact solution required?** For certain algorithms and applications, achieving the exact solution is the only viable possibility. However, in many situations an exact solution is not required. For instance, when ranking vertices of a graph, it is more important to obtain a good ranking in an acceptable time, than obtaining all the exact values of the ranking function. In addition, when working with large graphs, an approximated solution may be the only viable option. As a consequence, it is fundamental to identify where approximation can be introduced without affecting too much the final result.

- **is a versatile solution always possible?** For versatile solution we mean a solution able to adapt to more kind of datasets. This is an important quality of algorithms because many sources of data exists and may exhibit different characteristics. Easily switching between different datasets, while preserving the same algorithmic solution, may facilitate in providing valuable information from the data in less time.

Further, additional issues emerge when dealing with TLAV processing:

- **skewed computation load.** Many graphs exhibit a power-law distribution of the number of neighbors of a node. Due to this, some algorithms accessing the neighbours of each vertex may lead to skewed computation, caused by the different size of the neighborhood. Due to this, it is important to adapt the algorithm in order to avoid unbalanced computation. This can be achieved in many ways, for instance, approximation techniques can be introduced to re-size the neighbourhood of each vertex or the computation of a vertex can be splitted in multiple machines.

- **data partitioning.** One of the key aspect characterizing the distributed breakdown typical of TLAV approaches, is a proper data partitioning. That is a fundamental element for achieving an effective and efficient distributed computation. The data distribution strategies drive the placement of vertices and edges in a distributed environment on the available machines. Addressing the problem of a proper distribution leads to two different perspectives, firstly related to the proper orchestration of the computation and to the data access pattern, secondly to the efficient decomposition of data to enhance the performances by exploiting data locality. Usually, existing frameworks adopt simple and fast data partitioning strategies relying on hashing the vertex identifier. However, some algorithms may get performance boosts introducing a more sophisticated partitioning strategy.

- **unnecessary computation.** The TLAV model of computation requires to iteratively execute a given program on all the vertices of a graph. However, it may happen that some vertices reach a state in which further updates on that vertex are not necessary. In such scenarios, it is a waste of resources to continue executing the computation on all the vertices. Instead, simplification techniques can be adopted to shrink the number of vertices which continue the computation and continue the execution only where required. This has multiple benefits such as reducing both the time to complete an iteration and the

	Name	Approximation	Simplification	Versatility
Current Flow Betweenness (Ch. 3)	DUCKWEED	✓	(✓)	n/a
Connected Components (Ch. 4)	CRACKER	-	✓	n/a
Clustering k-NN + CC (Ch. 5)	k-NN+CC	✓	-	✓
Density Based Clustering (Ch. 6)	NG-DBSCAN	✓	✓	✓

TABLE 1.1: TLAV algorithms developed in this thesis.

memory required to save the intermediate data, because less vertices are involved.

1.2 Thesis Contributions

The goal of the thesis is to show how complex applications may be implemented by composing TLAV algorithms defined following useful guidelines. Also, as we showed in the previous section, although many works push on delivering novel solutions, many issues still exists. Due to this, existing algorithms often lack of specialized solutions or can be improved.

The algorithms we propose have been developed following some common guidelines that we think are important when defining a novel TLAV algorithm. In the following of this section, we provide a description of the guidelines and the algorithms defined.

1.2.1 Guidelines to define a TLAV algorithm

Table 1.1 shows the algorithms developed in this thesis and classify them according to a set of guidelines which are presented in the following.

- **thinking in TLAV (local knowledge)**. Even if the first guideline may sound redundant, many state-of-art solutions still rely on assumptions that are not always achievable in a distributed environment. For instance, most existing algorithms exploit a global view of the graph. Exploiting TLAV implies that each node during its computation may rely only on its local view of the graph. Due to this, it is infrequent that sequential solutions for a centralized environment may be easily adapted to such model. It is required to think about local solutions for the global problems and this requires to change the point of view of the algorithm definition.

- **approximation**. It is one of the most important guidelines when defining an algorithm. This is of particular importance in large graphs because some problems are intractable due to their size. Searching for approximated solutions is, in some scenarios, the only viable solution. In addition, as we stated in the previous section, sometimes an exact solution is not required because an approximated one may be enough for the problem's objective. Approximation can be achieved in numerous ways. In TLAV, where each vertex relies on its local view, it is possible to limit its view, for instance, in scenarios where the number of neighbors is skewed and, as a consequence, the computation would be unbalanced. This solution has been adopted for the clustering and the density based clustering presented in Chapter 5 and 6. Another source of approximation can be the early termination of the algorithm.

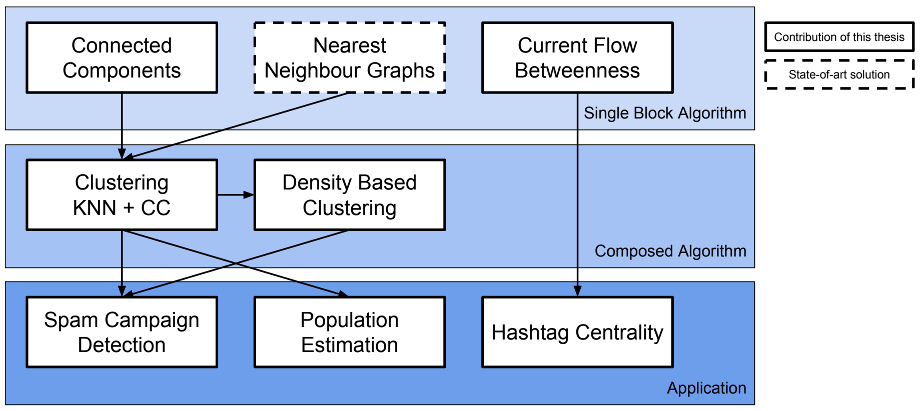


FIGURE 1.1: Thesis workflow. From single block algorithms to composed algorithms.

For instance, before reaching convergence, in some cases, a good solution is achieved in a few iterations and most of the remaining time to reach convergence is spent on minor improvements. Our aim is to detect when such scenarios occur and stop the computation early.

- **simplification**. In iterative algorithms implemented according to the TLAV paradigm, the computation is executed on each node, in each step of the computation. Simplification techniques can be introduced to reduce the number of computing nodes, while preserving the quality of the solution. We noticed that in some algorithms the execution continues also if the node has reached a local convergence or has already found an acceptable result. As a consequence, it is of relevant importance to keep active only the nodes which may improve the global solution. This has great benefits because less nodes means less time to execute an iteration and less memory to save the intermediate data. We introduce simplification techniques in the connected components and density based clustering algorithms shown in Chapter 4 and 6. In both the algorithms we remove nodes from the computation as soon as we discover the solution for the node or a solution good enough for the requisites of the problem.

- **versatility**. To provide an algorithm capable of handling arbitrary data is of paramount importance in order to permit the evaluation on different datasets. However, when studying state-of-art clustering algorithms the majority stems in providing optimizations for particular kind of data. We introduce the versatility in both the clustering algorithms we propose, respectively, Chapter 5 and 6. We show that such solutions have comparable computing time with respect to specialized solutions, although being able to accommodate any kind of data.

1.2.2 Proposed solutions

The contributions and the workflow of the thesis are summarized in Figure 1.1. We started providing our contributions by exploiting the TLAV model to solve classical graph problems:

- **connected components** (CRACKER) [122, 123]. Although being one of the most popular analysis on graphs and several distributed algorithms exists, we discovered that state-of-art solutions for the detection of connected components leave room for improvements. In particular, in such algorithms it is useless to continue processing nodes that already discovered the connected component to which they belong. Our solution is an algorithm, CRACKER, that employs a simplification technique by iteratively removing inactive nodes from the computation. CRACKER has been proven to return the same exact results as the other algorithms in literature but using only a fraction of their time.

- **betweenness centrality** (DUCKWEED) [121, 21]. We studied the current flow betweenness centrality where the importance of nodes in a graph is not evaluated only in term of the number of shortest paths traversing a node but on all the paths connecting each couple of nodes in the graph. An interesting property of this kind of betweenness is its similarity with electrical circuits, because each node can be seen as a conjunction between two or more resistors and an edge corresponds to a resistor. Current state of the art solutions for current flow betweenness centrality are all centralized, and a distributed solution for the computation of this measure is missing. In DUCKWEED we exploit the laws of current conservation to calculate the current flow betweenness using only the local knowledge of each node. Since the computation requires to consider all the possible couples of nodes in the graph, we use an *approximation* technique in order to reduce the costs to determine the solution. We show that, despite the approximation, a good correlation of the ranking achieved by our algorithm with respect to exact results is obtained. Also, we are able to provide the ranking also for large graphs where classical algorithms for betweenness centrality fail to deliver a result in an acceptable time or with an acceptable amount of resources. In DUCKWEED, we only select a subset of all the possible flows to determine the betweenness and each flow is iteratively computed by each node until an approximated local convergence is reached, i.e. the difference in the values between two consecutive iterations is under a threshold.

Next, we show how the previously defined algorithms may interact to construct more complex TLAV algorithms:

- **clustering** (**k-NN**+CC) [117]. The first task defined by a chain of TLAV algorithms presented in the thesis is a two phase clustering. The first phase exploits a parallel adaptation of a popular nearest neighbour (k-NN) construction graph algorithm. We choose such algorithm because it exploits a *local knowledge* to reach the solution and it is *versatile*, it is possible to accommodate any kind of data. In the second phase we make use of our CRACKER algorithm to define each cluster equal to one connected component of the k-NN graph. The k-NN phase is *approximate* since we do not require an exact k-NN graph to provide a good clustering quality. To this end, we conduct a deep study on the trade-off between clustering quality and early stop of the iterative process.

- **density based clustering (NG-DBSCAN)** [118]. Starting from the previously defined clustering algorithm, we noticed that it has some similarities with density based clustering. Due to this, we define an additional clustering algorithm which enhances the previous two phases clustering to deliver a result which approximates to the one provided by the DBSCAN algorithm [63]. We call this algorithm NG-DBSCAN (Neighbour Graph Density Based Clustering) and we introduce an *approximation* technique to exploit both a limited view on each node to avoid skewed computation and an early termination able to provide a good approximation of the DBSCAN algorithm avoiding longer running time. Also, we introduce *simplification*, by stopping the computation on such nodes that, at a given iteration, have already collected a number of similar nodes above a certain threshold. Results show that nodes are incredibly fast in discovering similar nodes and the majority of nodes stop the computation in the first iterations. Finally, we borrowed the *versatility*. Versatility is a plus in this scenario and both k-NN+CC and NG-DBSCAN exploit solutions able to accommodate arbitrary data. We show also that such solutions have comparable computing time with respect to specialized solutions although being able to accommodate any kind of data.

Finally, we applied all the above algorithms to real world applications:

- **spam campaign detection** [117]. The input of this application is a set of emails already tagged as SPAM. SPAM campaign detection is the task of identifying group of emails to perform root-cause analysis of large scale SPAM email campaigns originated from bot networks. This is of relevant importance for security providers in order to recognize the origin of the campaigns. We solved this task by clustering emails sharing similar subjects. This is an adversarial context because spammers manipulate text to avoid SPAM emails being clustered in the same campaign. Hence, the similarity metrics used for clustering must cope with text mangling, which require non-metric distances that disregard typos, character swapping, and other techniques to avoid detection. Due to this, the *versatility* of our proposed clustering algorithm has been relevant to solve this problem[1].

- **population estimation** [119, 120]. The goal of this application is to provide to statistical and political authorities a novel method to estimate the population in an area. This task is usually undertaken in the form of census activity. However, this is a costly tool and cannot be done frequently. Due to this, the idea is to exploit mobile calls to monitor the population living in an area. Thanks to a collaboration with an Italian mobile phone provider, we are able to obtain the calls performed in Tuscany. We characterize each user through an aggregated information in order to recognize if a user is living, working or just visiting a particular municipality. Thanks to the *versatility* of our clustering algorithm, we are able to aggregate and count the amount of people flowing in the municipalities. Results show that our approach is, with respect to the state-of-art, the one providing the smaller error

[1]This application has been possible thanks to a collaboration with Symantec Research Labs that actively collaborates to provide insights and the datasets to perform our analysis.

in estimating residents and it is able to detect also the commuters. Finally, such application permits to improve our clustering algorithm. In particular, we enhance the algorithm to deliver good results without requiring parameters' configuration from the users[2].

- **hashtag centrality**. The goal of this application is to identify on a daily basis which are the most important hashtags in Twitter, i.e. the hashtags where the majority of the information flows in a specific day. These may have different meaning, for instance, they may reveal the main topics discussed in a given day or may be used to inject novel informations reaching the largest amount of people. We analysed 606 days of tweets previously collected at ISTI-CNR[3]. Each tweet having at least two hashtags contributes to construct a graph where a node corresponds to an hashtag and an edge between two hashtags means that they co-occur in the same tweet. Results reveal that our algorithm DUCKWEED is able to identify the most important topic of each day and to correctly recognize the importance of the hashtags.

1.3 Outline of the Thesis

The remainder of this thesis is structured as follows:

Chapter 2 describes the Think Like a Vertex programming model and presents the state-of-art frameworks devoted to graph processing. Particular attention will be spent on Apache Spark because it has been the framework we choose to implement all the algorithms proposed in this thesis. Finally, we consider two popular algorithms as case study and we show how it is possible to implement them by exploiting three different frameworks supporting TLAV model.

Chapter 3 is the first chapter presenting the core contributions of this thesis. It presents our algorithm DUCKWEED for the approximation of the current flow betweenness centrality and how it is possible to port such algorithm on Apache Spark.

The idea, definition and evaluation of DUCKWEED *have been published in IEEE Conference on Self-Adaptive and Self-Organizing Systems with the title "Distributed Current Flow Betweenness Centrality" [121]. The porting of the algorithm in Apache Spark has been published in the International Conference on Algorithms and Architectures for Parallel Processing with the title "Current flow betweenness centrality with Apache Spark" [21].*

Chapter 4 presents the definition of the algorithm called CRACKER for the computation of connected components. The chapter presents also an extensive experimental campaign where we compared CRACKER with respect many competitors.

The seminal idea of CRACKER *has been published in the paper "Cracker: Crumbling large graphs into connected components" [122] presented at IEEE Symposium on Computers and Communication. An extension of the algorithm with optimizations and a theoretical study on the costs of the algorithm has been published in the*

[2]This application has been possible thanks to a collaboration with KDD Lab at ISTI-CNR.

[3]`http://rojo.isti.cnr.it/`

journal IEEE Transactions on Parallel and Distributed systems with the title "Fast Connected Components Computation in Large Graphs by Vertex Pruning" [123].

Chapter 5 presents our first application, text clustering, defined by composing two TLAV algorithms. The chapter presents also an evaluation of the trade-off between the approximation when calculating a k-NN graph and the clustering quality.

The proposal of this clustering algorithm has been published in IEEE Conference on Big Data with the title "Scalable k-NN based text clustering" [117].

Chapter 6 presents the algorithm NG-DBSCAN to perform density based clustering, which exploits all the guidelines introduced in Section 1.2.1.

The NG-DBSCAN *algorithm has been accepted for publication in the Proceedings of the VLDB Endowment, Vol. 10, No. 3 with the title "NG-DBSCAN: Approximate Scalable Density-Based Clustering for Arbitrary Data".*

The **Chapters 7, 8, 9** present the three applications we have implemented by exploiting the novel algorithms proposed in the thesis. These chapters present, respectively, SPAM campaign detection, population estimation and hashtags centrality.

The outcome of the SPAM campaign application has been published in IEEE Conference on Big Data with the title "Scalable k-NN based text clustering" [117]. The population estimation has been originally presented in the IEEE Symposium on Computers and Communication with the title ""Improving Population Estimation From Mobile Calls: a Clustering Approach" [119] and an extension is currently submitted to a journal [120].

Finally, **Chapter 10** presents the conclusion of the thesis, some undergoing works and future works.

Part I

Related Works

Chapter 2

The TLAV approach: frameworks and algorithms

The interest for large scale graph processing has noticeably increased. This is partially motivated by the fact that most of the problems faced in the Big Data analysis can be easily modeled by graphs. For instance, new graphs are generated daily to represent relationships in social networks, such as Facebook and Twitter, road networks and biological systems, such as protein structures and human brain connectome. All these examples are characterized by big datasets represented by graphs with a huge number of vertices and edges. In most cases the amount of data managed is so huge that a large graph representing the data either cannot fit in the memory of single computer or the fitting requires huge costs to manage ad hoc solutions.

In this chapter we introduce the TLAV approach and we present a set of frameworks implementing it. Furthermore, we review some popular graph algorithms that have been implemented following the TLAV approach. The literature related to the algorithms we have implemented is reviewed in the chapter where the corresponding algorithm is presented.

2.1 The Think Like A Vertex approach

The idea of Think Like a Vertex (TLAV) is to execute iteratively a user-defined program over the vertices of the graph. Such program must use local data as input (which include data from adjacent vertices or incoming edges) and runs a function, called "vertex-update", on each node of the graph. The output is communicated to adjacent vertices through outgoing edges. The goal is to find a solution (sometimes approximated), using a termination condition such as reaching a fixed number of "iterations" or exploiting halt voting procedures.

This approach is implemented by high-level frameworks whose focus is to allow a different kind of programming, focused on the viewpoint of each single vertex. Indeed, the vertex-centric vision helps in cases where having a whole comprehension of the entire graph can be difficult (and "costly", programmatically speaking).

The TLAV approach can be analyzed according to four fundamental pillars [131] which are Timing, Computation, Communication and Partitioning, each one exploring state-of-art solutions and opening a debate on which combination of features could be better than another when working on a specific algorithm. Please notice that pillars are independent to each other; indeed there are sets of feature choices that are more reasonable and others that do not work well intuitively.

2.1.1 Timing

When we speak about running the vertex-update function, we also need to decide when and which nodes must be chosen for updates. The timing of a TLAV framework describes the way in which “active” vertices are processed and handled by the underlying framework’s scheduler. Various models adopt the synchronous timing, which was first developed by Bulk Synchronous Parallel (BSP) [175] processing model, thus updating a subset of the active vertices in parallel and using a global barrier to achieve synchronization. In BSP, the computation is organized in “super-steps” which roughly correspond to the iterations of an iterative algorithm, and each super-step uses the output produced in the previous one as input for the vertices that are scheduled for the current super-step. The barrier prevents the computation of the next super-step from starting until previous super-step is completed. Additional details about the BSP model can be found in Section 2.2.2. Vertices are divided among the processing unit (PU), and the order vertices are scheduled within a PU does not affect the overall output of the computation.

Conversely, asynchronous timing allows an update function to be scheduled and run in parallel as soon as a PU is available for it. This way every vertex update function accesses the most recent state of its neighborhood when the execution starts. Moreover, computation can evolve without an explicit scheme, allowing vertices to be scheduled by the algorithm itself. Figure 2.1 shows the difference between the two models. Data updates are represented by the dotted arrows. In the synchronous timing, a single PU can be seen as a worker which executes a pre-determined number of update functions, and then waits for other workers to synchronize all data. Conversely, in the asynchronous setting workers execute the update functions independently from each other. There is no correlation between the time a node executes the update and the time its neighborhood changes. The absence of explicit synchronization barriers poses the typical problems of concurrency and race conditions on shared data; moreover the scheduler can reorganize dynamically the vertex execution order. This results in larger degree of flexibility but, on the other hand, the complexity of the framework (scheduling and data consistency) increases.

There are also frameworks that use both synchronous and asynchronous timing, and others that have chosen an hybrid approach, trying to pull out the best features of each pattern. For example, PowerSwitch [184] uses heuristics and statistics to predict which timing will perform better, and evaluations prove that on many benchmarking algorithms this approach performs well. Choosing the best model depends on what one wants to do, and efficient predictions are not so easy to be made; moreover, there are properties of the graph that can vary during the computation, thus reducing the initial advantages of the chosen pattern. We can observe that synchronous timing can be easier to design and it is almost always deterministic, but introduces trade-off performances; for example the throughput of each superstep is given by the slowest PU, and the communication between PUs can generate a significant overhead, so the framework is useful for lightweight computation with small load variability.

A study whose results are released in [131] has found that running an instance of “find the shortest path” problem on an highly-partitioned graph, synchronization accounted for 80% of the total time, thus warning us that we

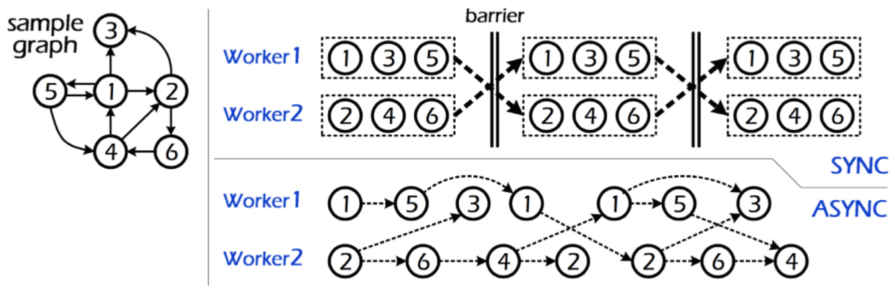


FIGURE 2.1: Synchronous and Asynchronous computation.

have to consider the bigger picture, including all the other pillars of a TLAV framework. Further, if the workload is not balanced, there is the risk to have many inactive workers waiting for the overloaded ones, acting as bottlenecks reducing the effective parallelism. In the asynchronous model, unbalanced workload is handled much better and generally these frameworks have shown to outperform the synchronous ones (however, exceptions are common); in case of vertices that are not always active, the workload can be very variable and asynchronous solutions could be preferred. Again, all depends from the specific algorithm: experimental analysis shows that synchronous execution is generally better for IO-bound algorithms, while asynchronous works well on CPU-bound algorithms [131].

2.1.2 Computation

The choices regarding the computational model of TLAV regard: i) how the vertex update is computed and ii) in which ways vertices/edges information can be accessed. We start answering the first question by introducing the One Phase model. The simplest way of TLAV is to acquire input data from any adjacent edge/vertex, update a vertex value by running the vertex-update function and finally send the update to the vertex's neighbors (modifying edges' or vertices' values), all in the same "atomic action". This is simple and direct, but there exist more flexible solutions that increase complexity a bit. This is the case of the Two Phase computational model, which divides the computation, leaving update-distribution alone in the second phase; we indeed speak about a Gather-Scatter model, where Gathering means getting the input and applying the vertex-function and Scattering is the last action of "sending" values towards the neighborhood.

Another consideration has to be done exploring in which way information flows. Basically there are two possible solutions: the pull mode of computation and the push one [45]. In the pull mode an active node of the graph can access neighboring vertices values and produce its own value, as well as ask the system to schedule one or more neighboring vertices, so information flows from the neighbors to the active vertex. On the contrary, the push mode makes information flow from active vertices out to near vertices under the form of messages; it makes possible the aggregation of all parallel updates in a single message from a machine to a remote vertex (this technique is known as sender-side aggregation and can reduce messages size by 90%).

2.1.3 Communication

This section discusses how communication and data sharing between machines is realized. The communication can be performed in two different manners, namely the message passing and shared memory.

First of all, we could think to send messages from a vertex to another (behaving differently if it is local or remote) and this can be implemented by a Message Passing framework. A message can hold vertex data, and edges are seen only as the connection between vertices. Synchronous execution often implements this communication pattern, in the sense that since value updates will be visible only in the next super-step, it is reasonable to send a block of messages during the synchronization barrier; it is shown to ensure data consistency without low-level implementations and is optimal for IO-bound algorithms.

The use of a synchronous model enables an optimization, i.e. message aggregation. The messages that need to reach the same destination can be combined in one, reducing network traffic at the end of a super-step.

On the other hand, the Shared-Memory pattern, allows to read and modify the state of neighbors directly without the need to send messages. This technique requires local data synchronization between multiple machines.

2.1.4 Graph partitioning

The last and deeper pillar is partitioning. This is a complex issue that basically tries to solve an inherent NP-Hard problem [81], so it is not surprising that it is the topic of a research field and plenty of heuristics have been developed. The graph partitioning focuses on obtaining a distribution of a graph on available machines such that the workload is balanced, while the number of edge cuts is minimized, in order to avoid networking overhead. We must notice that this is partially independent from the specific kind of framework, as all the previous abstractions rely on a given graph partition. Graph partitioning strongly impacts on the performances, because a well partitioned graph triggers a chain of runtime gains such as: reduced communication latency, decreased stuck waits on the synchronization barrier (if present), and so on.

2.2 Framework for graph processing

In the last years many novel solutions addressing parallel processing on graphs have been proposed. The aim of these techniques is to execute graph algorithms in a fast and efficient way. Some interesting innovations focus on smart techniques for graph partitioning and distribution.

In this section we give an overview of how the TLAV paradigm is implemented in real frameworks [51][146][124][113][193]. Crafting a custom distributed infrastructure, typically requires a substantial implementation effort. In addition, this process is error prone and requires specialists in distributed computing. The aim of these frameworks is to help the developers to focus on the algorithms to be implemented instead of dealing with low-levels details of the underlying architecture. All the details such as the creation of threads and processes, the data access, the data distribution and the managements of machines failures are usually handled by the framework. In the following subsections, we present some of the frameworks that are currently getting momentum.

In addition, since this thesis focuses on algorithms on large graphs, along with the presentation of frameworks, we introduce some well known problems and algorithms and how they can be implemented in some presented frameworks.

2.2.1 MapReduce

MapReduce [51] aims to process large quantity of data using massive parallel computation. It is inspired by the *map* and *reduce* primitives present in Lisp [129] and many other functional languages [86]. It has been shown to be suitable for many real tasks e.g. PageRank [29], K-Means Clustering [108], Multimedia Data Mining [180], Genetic Algorithms [178] and sometimes for Graph Analysis [47]. The programs written according to this model can be easily parallelized to be executed on distributed systems, clusters and even on commodity machines. The runtime system will take care about all the issues concerning the partition of the input data, load balancing and fault tolerance capabilities. Also, it will manage the inter-communication between machines and all the details that are architecturally dependent. All these features provided by the runtime enable the programmer to write code without knowing actually the architecture where the code will be executed on, and also without deep knowledge of parallel programming. From an high level perspective, it takes a set of input key/value pairs, and produces a set of output key/value pairs. The computation is expressed by two functions:

- The **Map** takes an input key/value pair and produces a set of intermediate key/value pairs;
- The **Reduce** takes an intermediate key and a set of values for that key and merges together these values to form a possibly smaller set. Typically just zero or one output value is produced per Reduce invocation.

The runtime groups together the pairs with the same key that are out-coming from the map invocation, and gives them to the reduce function. The final set of key/values pairs out-coming from the reduce execution is the result of the computation.

At the beginning of a MapReduce computation, the input data is partitioned into small blocks. Each idle worker (machine) takes a block and executes on it the map function and finally writes on disk the intermediate pairs. After, all the intermediate pairs are stored on disks, all the workers are idle. At this point, each idle worker takes a block of intermediate data and executes the reduce function on it and again the result is stored on disk. The consequence of this writing/reading mechanism of data is to enable the processing of large quantity of data that does not fit into the physical memory. Figure 2.2, which is taken from [51], shows the overall architecture in a possible master-workers fashion. In case of failures of one or more machines, the runtime system reschedules the map/reduce tasks relying on the fact they are re-computable, i.e. the model relies on re-execution of tasks as the primarily mechanism of fault tolerance. This usually forces the tasks to be independent.

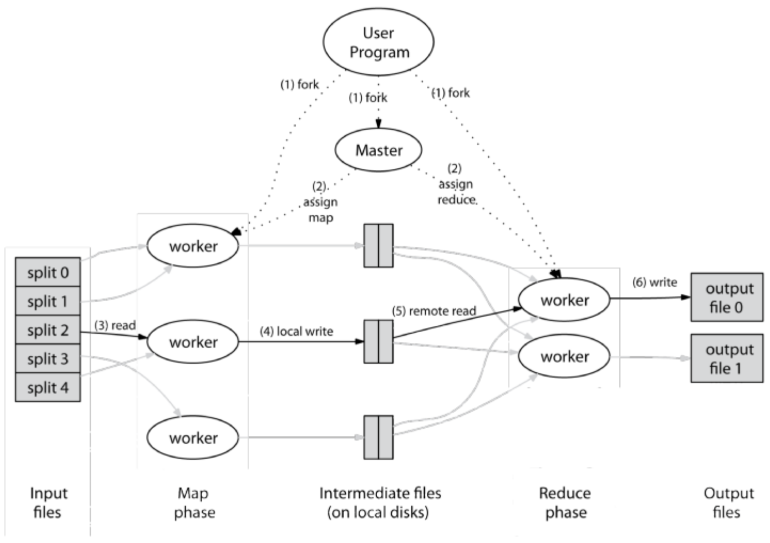


FIGURE 2.2: An overview of an execution in MapReduce.

In the following we present a concrete example implementing the PageRank algorithm to show the characteristics of the model.

Example (PagerRank). The PageRank of a vertex v is defined by the following equation:

$$Rank(v) = \alpha + (1 - \alpha) \sum_{u \in Neigh(v)} \frac{Rank(u)}{OutDegree(u)} \tag{2.1}$$

where *Rank*(v) identifies the value of relevance of vertex v, *OutDegree* identifies the number of edges outgoing the vertex u, and α is set to 0.15. The PageRank algorithm iterates on the function 2.1 until convergence. The convergence is obtained when the value of *rank* between two consecutive iterations is less than a fixed value ϵ.

An high level description of the PageRank algorithm according to the MapReduce model is presented in Algorithm 2.1. In the pseudo-code are presented the two mandatory functions *Map* and *Reduce*.

The Map function (see Algorithm 2.1 line 1) takes key/value input. The value part represents an adjacency list, i.e. the identifiers of all its neighbours. In the initialization, each vertex has a rank value equal to 1. The function call *SendMsg* (see line 4) is not a communication primitive but it generates a sequence of (key,value) pairs where the value is the contribution of the vertex and key is the identifier to which the value must be dispatched. All the data produced in the above manner are then grouped by key, so that all the data relative to a vertex are grouped in a same set. After the *map* phase, each worker creates an intermediate file organized by key and such data is the input of the next phase called *Reduce*. An instance of the Reduce function is executed for each vertex, it iterates on the contribution relative to a vertex to generate the rank value (see line 11). At the end a key/value structure will be the input of an other MapReduce iteration. It is important to notice that the MapReduce model is stateless. Due to this,

Algorithm 2.1: MapReduce Pagerank

```
input  : Graph G(V,E)
output: Vertices' rank
/* Map Function                                                    */
Map (Vertex N, Rank K, Iterator<Vertex> out_neighbors) begin
    Nn ← Size (out_neighbors) ;
    foreach nbr ∈ out_neighbors do
        SendMsg (nbr, Message(N, K / Nn) );
    end
    SendMsg (N, out_neighbors );
end
/* Reduce Function                                                 */
Reduce ( Iterator<Message(Vertex K, Rank N)> message) begin
    RankK ← 0 ;
    foreach nbr ∈ message do
        RankK ← RankK + nbr.Rank * 0.85 + 0.15 ;
    end
    SendMsg ( (K, RankK) → Iterator<Vertex> out_neighbors ) ;
end
```

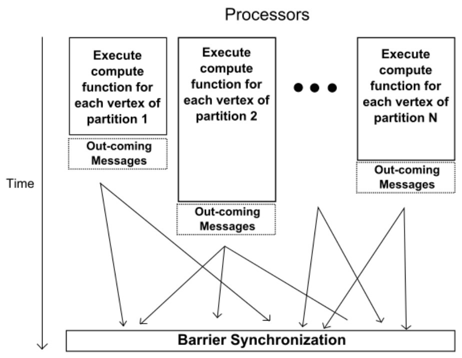

Processors
Execute compute function for each vertex of partition 1
Execute compute function for each vertex of partition 2
Execute compute function for each vertex of partition N
Out-coming Messages
Out-coming Messages
Out-coming Messages
Time
Barrier Synchronization

Figure 2.3: Pregel's superstep as BSP implementation

in each phase, the state of a vertex, its adjacency list, must be propagated again to be available in the following iterations.

2.2.2 Pregel

Pregel [124] is a framework, created by Google, to compute algorithms on large scale graphs which takes inspiration from the BSP model. The motivation behind its introduction, as mentioned above, derives from the fact that the MapReduce is suitable for a large number of graph analysis algorithms, but not for all of them. Some algorithms use global knowledge of the graph to compute their result, in that cases MapReduce could suit well. Instead, the algorithms which are developed according to the TLAV approach may get benefits in the definition of another framework. In these cases, Pregel could suit better than MapReduce, because it supplies the "lack of expressiveness" of MapReduce by changing the abstraction of the data. Pregel's programs are vertex-centric, which means the operations are defined as "looking from the point of view of a vertex". Being a BSP-inspired model, a Pregel's computation is divided into supersteps and during each superstep the function associated to each vertex of the graph is executed. We provide additional details about the BSP in a subsequent sub-section.

Figure 2.3 shows a superstep from the point of view of the actual computation of Pregel. The result of the total execution is changing the state of the vertices, for example inner attributes but also the incident edges, i.e. the topology could be different at the end of the computation. In addition, along the supersteps, vertices could sum up some values into accumulators which are global structures managed by the framework safely along the computations. An accumulator is parametrized by an accumulation function which has to be associative and commutative, sums up the current accumulator's value and the one given as parameter.

Fault tolerance is achieved through checkpoints and re-computation. A checkpoint saves the data on disks in such a way that, in case of failure, it limits the number of the vertices that have to be recomputed and also they have not to be re-computed from the beginning, i.e. superstep 0, but from the last check-pointed superstep.

Each processor exchanges a lot of messages and it may happen that a subset of those is redundant. An optimization is the use of combiners which are executed after the computation and the generation of the out-coming messages, but before that the communication is actually performed. Combiners could merge the out-coming messages into a smaller set in such a way the data communication is lower (e.g. algorithms that perform a summation of values from incoming messages could create a combiner that merges out-coming messages that are going to the same vertex into one message whose value is the sum of the others).

Bulk Synchronous Parallel

The Bulk Synchronous Parallel (BSP) [175] is a bridging model to define parallel algorithms. It was originally conceived as a theoretical model, however recently has gained momentum and implemented on popular graph processing frameworks. The BSP requires the following architectural features:

- components able to compute using local memory;
- a structure able to dispatch and receive messages between two components;
- a component demanded to the synchronization of the other components.

A computation in BSP is described by a sequence of supersteps (take as reference Figure 2.4) where each superstep is divided in three phases:

- **parallel computation**. Each component executes independently using only its local data.
- **communication**. At the end of the computation phase each component may send messages to other components.
- **synchronization barrier**. When a component reaches this phase it waits all the other components before restarting the computation phase.

The BSP model is implemented on many distributed frameworks because it is a perfect match to define iterative algorithms on graphs and it permits to exploit the TLAV model making use of a parallel architecture. There is a strict correspondence between the BSP model and the TLAV approach

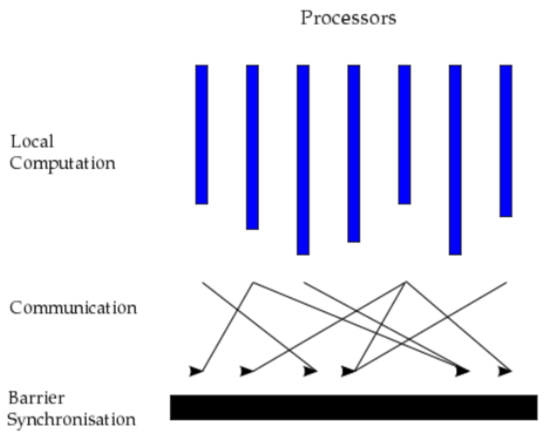


FIGURE 2.4: Superstep: an example

implemented on a synchronous framework where each vertex is a component able to perform the computation due to its local memory and its adjacency list. It provides primitives for communication and synchronization between the computations executed on the vertices. At the same time the BSP model may be mapped on a real architecture composed of processors with local memory and interconnected by a network, where each processor executes the computation of a set of vertices sequentially.

The BSP model defines a cost model. The cost of a superstep is the sum of the following terms: T_i is the execution time of the component i, H_i is the number of messages sent and received by the component i, g is the time required to send a message, and L the time required to synchronize all the components. The cost of a generic superstep results in:

$$C_{superstep} = \max_{i=0..p}(T_i) + \max_{i=0..p}(H_i g) + L \tag{2.2}$$

Whereas the cost has three major contributors, the first one represents the longest computation of a component, the second one is the communication having the largest cost, the last one is the synchronization cost. If the execution requires S supersteps, the total cost becomes the sum of each superstep. Calling $W_s = \max_{i=0..p}(T_i)$, the maximum completion time in superstep s, and $H_s = \max_{i=0..p}(H_i)$ the communication having max cost in superstep s we get:

$$C_{tot} = \sum_{s=1}^{S} W_s + g \sum_{s=1}^{S} H_s + SL \tag{2.3}$$

The total cost is C_{tot} the sum of all the maximum computational times of each superstep $\sum_{s=1}^{S} W_s$ plus the sum of each maximum communication times $g \sum_{s=1}^{S} H_s$ plus each of the synchronization times SL.

Other implementations

Pregel is a closed and proprietary solution developed and used at Google. A lot of alternative implementations have been developed, each one differing for its actual implementation, the set of native operations provided to work on graph, and the fault tolerance mechanism that is employed.

First, it is important to notice that also other solutions exist which provide similar API to work on large scale graph but are not inspired directly to the BSP model. The most noticeable is GraphLab [112] which works on shared memory and it is pretty fast on single machine and clusters. Another noticeable alternative is Combinatorial BLAS [31], while widely used open-source Pregel implementation is Hadoop Hama [162]. Hadoop is the open-source implementation of MapReduce and Hama is the implementation of Pregel that runs over Hadoop but does not have any fault tolerance mechanisms. Yahoo! has also developed its own open-source framework called Apache Giraph. It runs over Hadoop and performs fault tolerance through check-pointing. Another framework is "GPS: A Graph Processing System" [157]. It is also BSP inspired and provides a specific domain language Green-Marl [84] that enables intuitive and simple expression of complicated algorithms.

2.2.3 GraphLab

GraphLab [112] was developed with the aim to implement parallel machine learning algorithms efficiently, and to provide scheduler's customizations and data consistency to address the needs of a wide range of Machine Learning problems, whose computational patterns may vary significantly. The developers of GraphLab also developed GraphChi [102]. The fundamental building blocks of GraphLab are the data graph (e.g the web graph for PageRank algorithm, where each vertex corresponds to a web page), a range of schedulers to express the way computation must evolve, and a vertex-update function. The update function takes as arguments the vertex and its scope.

Definition 1 (Scope)**.** *The scope S_v of a vertex v is defined as the set including v, its adjacent edges and its neighboring vertices. Thus, we can define the vertex function as $f(v, S_v) \rightarrow (S'_v, \mathcal{T}')$ where $\mathcal{T}'$ is a set of vertices and S'_v the new scope. The scope S_v is the set needed from the point of view of a vertex v to perform its computation, and is the extent of the graph which can be accessed by v.*

Algorithm 2.2: GraphLab execution model.

```
  Input: Data graph $G = (V, E, D)$
  Input: Initial vertex set $\mathcal{T} = \{v_1, v_2, ...\}$
1 while $\mathcal{T} \neq \emptyset$ do
2     $v \leftarrow$RemoveNext($\mathcal{T}$);
3     $(\mathcal{T}', S_v) \leftarrow f(v, S_v)$;
4     $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$;
5 end
```

The computation may update the scope S_v and return a set $\mathcal{T}$ of vertices which will be added to the scheduler for further execution. The basic

Algorithm 2.3: The GAS interface.

```
interface GASVertexProgram(u)
    gather $(D_u, D_{(u,v)}, D_v) \rightarrow Accum$
    sum (Accum left, Accum right) $\rightarrow$ Accum
    apply $(D_u, Accum) \rightarrow D_u^{new}$
    scatter $(D_u^{new}, D_{(u,v)}, D_v) \rightarrow (F_{u,v}^{new}, Accum)$
```

execution model of GraphLab is extremely simple and is shown in Algorithm 2.2. Even if this model has to be modified when we consider a distributed execution, it is presented here to show the basic characteristics of the framework. The model is thought as a loop that executes sequentially the update-function of a vertex extracted from $\mathcal{T}$, until $\mathcal{T}$ becomes empty. Other two main features of GraphLab are the definition of data consistency models and ghost vertices. The programmer can choose among three difference consistency levels in order to increase or reduce parallelism. The consistency level can be chosen before the computation starts, according to the characteristics of the algorithm that has to be run. The higher level of consistency corresponds to a lower level of parallelism. On the other hand, ghosts vertices are used to implement the shared-memory pattern, since the graph can be split across machines cutting a subset of edges. Indeed, every ghost represents a specific vertex which resides on a machine and other ghosts of the same vertex communicate to maintain their values consistent.

2.2.4 PowerGraph

Algorithm 2.4: PowerGraph execution semantics.

```
Input: Center vertex $u$
if cached accumulator $a_u$ is empty then
    forall the $v \in neighbor(u)$ do
        $a_u \leftarrow$ sum($a_u$,gather($D_u, D_{(u,v)}, D_v$));
    end
end
$D_u \leftarrow$ apply($D_u, a_u$);
forall the $v \in neighbor(u)$ do
    $(D_{(u,v)}, \Delta a) \leftarrow$ scatter($D_u, D_{(u,v)}, D_v$);
    if $a_v \wedge \Delta a \neq \emptyset$ then
        $a_v \leftarrow$ sum($a_v, \Delta a$);
    else
        $a_v \leftarrow$ Empty;
    end
end
```

PowerGraph[78] is considered to be the successor of GraphLab, though it takes some features from Pregel. PowerGraph's focus is to handle natural graphs and their power-law distribution. PowerGraph model is also called the Three Phase Model [131], which is composed by: the **G**ather phase, the **A**pply phase, the **S**catter phase. This is why it is also called the GAS model, introduced for the first time by PowerGraph itself. Like in GraphLab, the state of a PowerGraph program is stored in the data graph, in vertices and edges. Every program written in this framework has to implement the interface shown in Algorithm 2.3 including the signature of three functions, each one corresponding to a GAS phase, plus a "sum" function. The gather

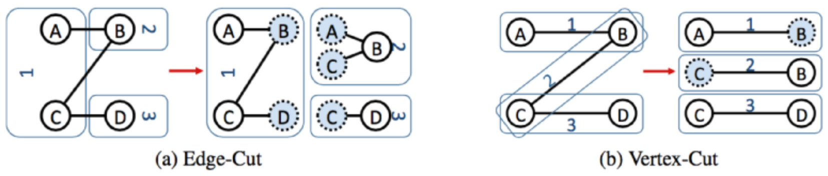


Figure 2.5: Differences between vertex cut and edges cuts.

phase is used to get the necessary information to perform the "apply" function of the GAS interface. The "gather" function is run in parallel on the vertex's neighbors, instead, the "sum" function aggregates values obtained by the execution of different gather functions. The scatter phase is used to activate adjacent vertices and update edges values. Although it seems very similar to GraphLab, the execution model is different, given the fact that, for example, the engine can be synchronous or asynchronous.

Computation starts activating one or all vertices of the graph, and proceeds until there are no more vertices to be activated. Every function can only activate vertices that are accessible to itself, and they will be scheduled in an order decided by the engine. The power of this abstraction is the fusion of this kind of model, which allows to execute concurrently the GAS phases, with the vertex-cut partitioning, which is described afterwards in the section. The engine splits the computation of a single vertex among different machines and this grants PowerGraph an high amount of parallelism, since these operations can be run concurrently on neighbors. The general execution semantics is described in Algorithm 2.4, which is independent from the engine used. D_u represents the current value in vertex u, while $D_{(u,v)}$ the value on the edge (u, v). The execution semantics encapsulates the functions of the GAS interface defined by the programmer, and an original point to notice is the use of an accumulator a_u in the scatter and gather phases. This particular feature of the framework is called Delta Caching: more in details, during the Gather phase the vertex takes inputs from adjacent edges and combines them for next steps. When there is little or no change in neighboring vertices, a cached accumulator a_u can be used to maintain the value computed in the last Gather phase just adding a Δa value. Rather than doing another Gather computation, which can be expensive and useless since many edges have not changed their value, a vertex can reuse this accumulator, atomically modified by neighbors over time.

PowerGraph is well suited for natural graphs. To achieve its effectiveness in this context, the framework makes use of vertex-cuts. This means that a vertex may be replicated on more machines and its edges distributed on all these machines. This greatly helps to distribute the edges of high degree nodes, which characterize power-law graphs, on more machines. Figure 2.5 show a comparison between edge-cuts and vertex-cuts. It can be noticed that with edge-cuts the number of replicas can be very high, while vertex-cuts need a very smaller number of "mirrors". This can greatly reduce the amount of communication workload and it is one of the most important features of PowerGraph.

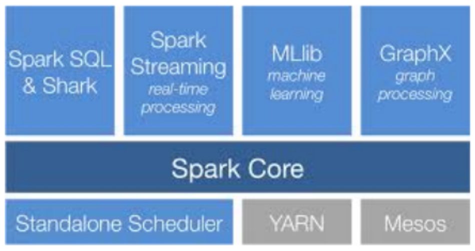


FIGURE 2.6: Spark Architecture

2.2.5 Apache Spark

All the algorithms presented in the following of this thesis have been implemented by exploiting the Apache Spark framework [193] which is a popular framework for distributed processing. It is BSP-inspired and, as its website reports, "Spark is a general engine for large-scale data processing". Its main target are distributed systems, especially commodity hardware clusters. It provides a rich set of API and algorithms ranging from MapReduce to a large built-in library for machine learning, a concrete integration with SQL and streaming functionalities. In the following of this section the concept of Resilient Distributed Datasets (RDD) which are the beating heart of Apache Spark will be introduced. RDDs define how data are handled and processed. Then, we describe how an execution works and we provide a description of the graph processing layer called GraphX, developed over Apache Spark.

Resilient Distributed Datasets (RDD)

Apache Spark exploits data parallelism through Resilient Distributed Datasets (RDDs) [192]. RDD is a distributed memory abstraction consisting of immutable collection of objects spread across a cluster that are read-only and can be generated only through deterministic and finite operations from either a dataset in a stable storage or other existing RDDs. These data structures enable Spark to store intermediate results in memory between two iterations. This overcomes the limitation of the MapReduce frameworks, where data have to be stored on disks. A RDD is organised into a number of partitions, which are atomic pieces of information and can be stored on different nodes of a cluster. RDDs contains all the informations required to derive the current state of the data from the stable storage.

The parallel distributed computation is expressed by means of operations over RDDs. These operations, called transformations are computed lazily. The derivations are computed in a lazy way by exploiting a logging structure, called lineage, which is also the basic mechanism for fault tolerance, when fault occurs it is possible to recompute the RDD directly from the stable storage using the chain of derivation. The usage of checkpoints is also possible, but it is expensive because of materializing of the intermediate results. On the other hand, it is useful when the derivation chain is becoming longer and longer to prevent very long re-computation in case of faults.

The RDD abstraction can be used both with MapReduce, as well as with the vertex-centric models of computation. Each element of an RDD can be used to represent a vertex with its own state and its own adjacency list (see Table 2.1). Next, it is possible to use the rich set of APIs to define the computation. For instance, the *map* function can be used to execute

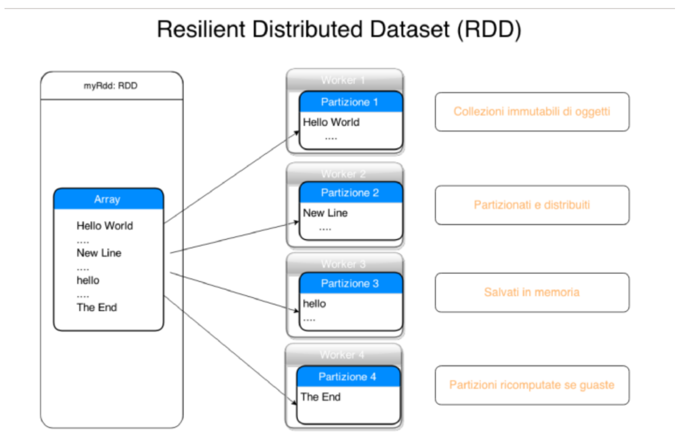


FIGURE 2.7: An RDD of Strings partitioned on 4 machines.

Partition 1	
Id	*Adjacency list*
1	[2]
2	[1, 3, 4]
Partition 2	
Id	*Adjacency list*
3	[2, 4]
4	[2, 3]

TABLE 2.1: Implementing the TLAV model using RDD. In the example one RDD composed of two partitions. Each element has a key, the identifier, and a value, with the adjacency list of the vertex.

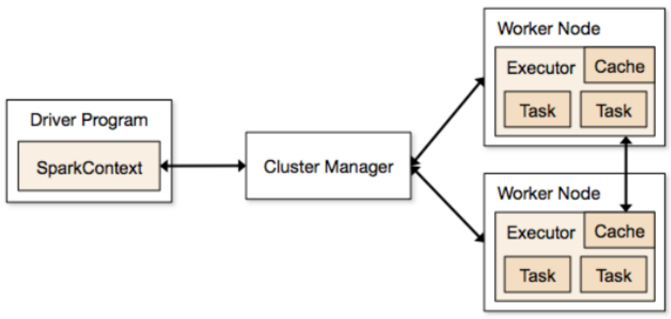


FIGURE 2.8: Spark Architecture

an user provided function on each vertex. An example is provided by the function ***Degree***, in Algorithm 2.5, which permits to compute the degree on each vertex.

Algorithm 2.5: Apache Spark, an example of a function.

```
  /* Degree function.  Each vertex computes its number of edges equals to its
     degree.                                                                  */
1 Degree (element $< id, adjacencyList >$) begin
2  |  return $element.adjacencyList.size()$
3 end
```

To emulate the communication between vertices it is necessary that each element (vertex) is uniquely identified. A common approach is to assign to each vertex a unique name (see Table 2.1). Such identifier has a double scope: (i) it permits an efficient partitioning of the vertices on multiple partitions (only a few partitioning strategies are built-in in Apache Spark), (ii) it permits an easy and efficient mechanism of routing messages on each vertex.

How it works

Spark is written in the Scala language and provides API for Scala, Java and Python. A collection of Java Virtual Machines is responsible for the execution of all the programs in Spark.

To make use of the Spark framework, developers write a program called *driver* which is connected to the cluster via the master node. Figure 2.8 shows a view of how the machines are organized in Spark. The architecture includes a cluster manager and multiple worker nodes. The computation is partitioned by the manager in multiple tasks and each task is assigned to a worker node. Then, the developer defines in the driver one or more RDDs through transformations on the data. The methods that can be called on an RDD can be *transformations* and *actions*. Transformations are operations that take as input an RDD and provide as output a modified RDD, for instance: map, filter and join. Actions are operations that return values, for instance, count, first and reduce. It is important to understand that transformations are lazily evaluated, they are not computed until an action is called on the specific RDD.

Each action requires the triggering of the job scheduler which needs to retrieve all the data dependencies of the RDD and creates an execution

```
 1  import org.apache.spark.SparkContext
 2  object Exemple {
 3     /*
 4     * Metodo eseguito sul driver
 5     */
 6     def main(){
 7        val spark = SparkContext()
 8        val lines = spark.textFile(edgelist)
 9        val vertex = lines.flatMap { s =>
10           val arco = s.split(" ")
11           (arco(0).toInt, arco(1).toInt)
12        }.distinct().groupByKey()
13        vertex.count()
14        ....
15     }
16  }
```

Figure 2.9: Spark Example

plan. The operations identified in the execution plan are then assigned to the workers which receive the operation to be executed on the partition of data handled by each of them by exploiting both locality of data and load balancing.

Figure 2.9 presents a fragment of Spark code. Such code shows how to create, modify and execute transformations and operations on RDDs. The application takes as input a file containing a graph where each line is a vertex with its adjacency list. The code in the *main* is executed on the machine where the driver program is launched. The reference *spark* in line 7 is the point of access to the cluster. In line 8 is created an RDD making use of the Spark's API that returns an RDD from a text file where each line is an element of the RDD. The *flatMap* at line 9 builds an RDD from another RDD. Instead, at line 13, is presented an action (count) that counts the number of elements of the RDD.

GraphX

GraphX [186][185] is a distributed graph computation framework that unifies graph-parallel and data-parallel computation. In addition to graph computation it provides also operations for constructing the graph from an external source and for modifying the graph structure. GraphX presents a unified abstraction, which allows the same data to be viewed both as a graph and as a table without requiring any data movement or duplication. In addition to the standard data-parallel operators such as map, reduce, filter and join, GraphX introduces a set of graph parallel operators such as subgraph and mapReduceTriplets which updates graph data through a highly parallel edge-centric API.

The GraphX system adopts the BSP model and it ensures deterministic execution, simplifies debugging and enables fault tolerance. GraphX is implemented on top of Spark[193] that provides facilities and indexing to speed up the computation on graph. The data structures of GraphX are immutable, as a consequence its support to evolving graphs is not efficient because adding or removing edges or vertices requires to rebuild the indices on every graph modification. This issue limits its usage on those graph algorithms where the topology of the graph does not change during the execution.

In GraphX, graph data is represented as a property graph, which associates user-defined properties with each vertex and edge. Similarly to PowerGraph the computation is organized making use of the Gather-Apply-Scatter (GAS) decomposition. The motivation is that most vertex programs interact with neighboring vertices by collecting messages in the form of a generalized commutative associative sum and then broadcast new messages in an inherently parallel loop. In addition, such model of computation enables the vertex-cut partitioning. This, in some cases, for instance when the graph exhibits some vertices with a very large number of neighbours, permits lower movement of data and to introduce optimizations. Instead of delivering high degree vertices to each neighbor is enough to mirror such vertices on multiple machines and synchronize the different mirrors.

2.3 Case studies

This Section presents some popular algorithms that will be analyzed with respect to their high level programming with TLAV. Also, will be presented considerations on how the same algorithm can be implemented in different frameworks supporting TLAV computation and which are the benefits of the different frameworks. Before we introduce which are the typologies of problems that is possible to tackle thanks to graph processing.

2.3.1 Graph Problems

In the following we present some classes of problems requiring graphs processing, as presented in [59]:

- *Traversal operations*: are operations that start from a single node and explore recursively the neighbourhood of a node until a final condition is reached. An example is the k-core decomposition to compute the k-coreness measure of each node in the graph, a problem recently studied in [136].

- *Topology analysis*: graph analysis includes the study of the topology of graphs to analyse their complexity. It is conducted to verify some specific data distributions, to evaluate a potential match against a specific pattern, or to get detailed information about the role of nodes and edges. For instance, this analysis is used in Bio-Informatics to find significant mutations and pathways in cancer genomics [177].

- *Connected components*: a connected component is a subset of the nodes composing a graph where there exists a path between any pair of such nodes. Thus, a node only belongs to a single connected component of the graph. Finding connected components is usually crucial for many operations and it is typically conducted in a pre-processing phase. Computing connected components is also useful to study the vulnerability of a graph, or the probability to separate a connected component into two other components. For instance finding connected components is crucial to search for critical locations in a spatial network [55].

- *Community detection*: a community is generally considered to be a set of nodes where each node is closer to the other ones within the community than to nodes outside it. Communities can be found in many real world graphs, for instance social networks. Community detection is exploited, for example, to find communities devoted to the discussion of controversial topics, where one can expect to find strong interactions [30].

- *Centrality measures*: a centrality measure aims at giving an indication of the importance of a node based on how well this node connects the network. The most well-known centrality measures are degree, closeness and betweenness centrality. As an example, recently the web graph has been analysed and the ranks returned by different centrality measures have been compared [1].
- *Graph anonymization*: is the process of generating a new graph with properties similar to the original one, avoiding potential intruders to re-identify nodes or edges. This problem gets more complex when the nodes and edges contain attributes and the problem goes beyond the anonymization of the pure graph structure [22]. The anonymization of graphs becomes important when several actors exchange datasets that include personal information.
- *Clustering*: consists in finding groups of related data items, according to a definition of similarity that is application specific. Clustering algorithms are fundamental in data analysis, providing an unsupervised way to aid understanding and interpreting data by grouping similar objects together.

In the following of this section we select several case studies and we show how is possible to perform their implementation on two different frameworks supporting the TLAV approach.

2.3.2 PageRank

Known as one of the most popular algorithms, PageRank [142] allows Google to compute web pages' ranking. PageRank runs periodically in order to obtain the updated rank for old and new web pages. The computation of such large graphs, the graph of web pages has billions vertices and edges, has conducted Google to develop the MapReduce framework. This, exploiting commodity hardware and distributing the computation on a large number of machines, permits to compute the result wherein a single machine would not be enough due to the size of such data.

Here, we are not going to give an in-depth analysis of PageRank details of the entire PageRank pseudocode, rather we analyze the "design" power of its TLAV version. This is a very high level pseudocode, but we are talking about "designing efficiency", not creating efficient algorithms. We can briefly notice that a vertex needs to acquire the rank of the pages that "point" it (through the in-edges), compute its new rank and finally send the new value to the pages it "points" (through the out-edges).

GraphLab. Running PageRank on distributed GraphLab allow to achieve a parallel exploitation of resources by means of its asynchronous timing. This is particularly evident when dealing with graphs leading to an unbalanced workload. GraphLab requires that programmers ensure data consistency. The most reasonable choice seems to be the edges consistency

[1]http://wwwranking.webdatacommons.org/

Algorithm 2.6: GraphLab pseudocode for PageRank.

```
**Input**: $R(v)$: vertex data from $S_v$;
// $S_v$being the scope of vertex$v$. **Input**: Edge data $\{w_{u,v} : u \in Neighbor(v)\}$ from $S_v$;
$R_{old}(v) \leftarrow R(v)$;
$R(v) \leftarrow \frac{\alpha}{n}$;
forall the $u \in N[v]$ do
    $R(v) \leftarrow R(v) + (1 - \alpha) * w_{u,v} * R(u)$;
end
if $|R(v) - R_{old}(v)| > \varepsilon$ then
    return $\{u : u \in Neighbor(v)\}$;
end
```

model, since adjacent vertices must not execute concurrently to avoid inconsistency. PageRank pseudocode shown in Algorithm 2.6 schedules updates only if vertex rank changes sufficiently, so this version does not expect keeping vertices active for all computation runtime. A complete implementation in a GraphLab engine may allow vertices voting to halt the computation, and the periodical Sync function could check this votes to stop and return a result. As for communication, the presence of a natural graph such as the Web, where a few web pages have a huge number of connections, while many other ones are not linked at all, tends to increase the network overhead due to an high number of ghosts. Ghost updates are sent before releasing the lock on a vertex scope, and this permits neighbor vertices that will be scheduled to read the most updated values.

Algorithm 2.7: PageRank GAS implementation.

```
// in the following $D_u$ represents the current state of vertex $u$ procedure
gather($D_u, D_{u,v}, D_v$)
    return $\frac{D_v.rank}{|neighbor(v)|}$;
    //an accumulator
procedure sum(Accum $a$, Accum $b$)
    return $a + b$
procedure apply($D_u$, Accum $a$)
    $rnew = 0.15 + 0.85 * a$;
    $D_u.delta = \frac{r_{new} - D_u.rank}{neighbors(u)}$;
    $D_u.rank = rnew$;
procedure scatter($D_u, D_{u,v}, D_v$)
    if $D_v.delta > \varepsilon$ then
        Activate($v$);
        return $D_v.delta$;
    end
```

PowerGraph. Since this framework relies on the GAS model of computation, we design PageRank algorithm providing an implementation of the Gather-Apply-Scatter interface. As we can observe in Algorithm 2.7, the *Gather* phase is executed on each vertex v. It returns the contribution that must be sent to each neighbor. It is calculated dividing the current rank of v by the number of neighbors. The *Apply* phase takes care of computing u's new rank and also updates delta value, which is responsible for both delta caching technique and halt voting. Indeed in the scatter phase delta value is checked to decide if a neighboring vertex v needs to be activated, i.e the value changes significantly, and also this value is returned to allow PowerGraph applying delta caching. Notice that vertex-cut and parallel communication are invisible to the programmer and handled by the engine.

Gather function runs in parallel on u adjacent vertices (or a subset of them), as well as the scatter one. Intuitively, a PageRank algorithm could keep vertices always active, and stop the computation when a convergence criterion is met; this could work also in a synchronous framework where there is no need to re-activate vertices, given the fact that a barrier semantically acts as a vertices activation. However, PowerGraph follows GraphLab's idea and schedules vertices execution only when "necessary", and this is due to the fact that delta caching helps to reduce the amount of runtime [78].

Apache Spark. Apache Spark provides a rich set of API that can be used for processing data. As aforementioned, such APIs include but are not limited to MapReduce. However, when implementing this algorithm on Spark only the two functions *Map* and *ReduceByKey* need to be exploited similarly as making use of MapReduce. ReduceByKey is the equivalent of the Reduce function in MapReduce when working with key/value data. In particular, it permits to use the provided function on each machine and combine the values directed to one single vertex in only one value before performing the communication between different machines. Data can be organized according to Table 2.1 adding to each vertex a double value representing the rank. The computation is synchronous, i.e. before executing the ReduceByKey all the Map function are executed.

Since Apache Spark provides also GraphX, the PageRank algorithm may be implemented also with such API. The computation is organized similarly to PowerGraph thanks to the GAS abstraction. However, it is not possible to use delta caching or vertex activation/deactivation and the function is executed on all the vertices in each iteration, until a stop criterion is met. Spark with GraphX comes with the PageRank algorithm built-in. Due to this, it is required to perform such specific API call equivalently as calling any other function provided by the API.

Algorithm 2.8: HASH-TO-MIN vertex-centric pseudocode for vertex v.

```
   // Initialization
1  $C_v = v \cup neighbors(v)$;
   // Computation
2  if $\neg halt$ then
3      $v_{min} = min(C_v)$;
4      for $u \in C_v$ do
5          emit($u, v_{min}$);
6      end
7      emit($v_{min}, C_v$);
8  end
9  for $u \in C_v$ do
10     $C_{new} = C_{new} \cup C_u$;
11 end
12 if $C_{new} = C_v$ then
13     $halt = true$;
14 else
15     $C_v = C_{new}$;
16     $halt = false$;
17 end
```

2.3.3 Connected Components: HASH-TO-MIN

HASH-TO-MIN is part of a more general algorithm for computing connected components in a graph in $\mathcal{O}(log(n))$ rounds (in MapReduce), where n is the number of nodes in the largest connected component [151]. It was developed to overcome limitations of its previous version, called Hash-to-All, which is characterized by an high communication overhead.

The high level description of this algorithm can be given by two functions: an hashing function h in the mapper phase and a merging function m in the reducer phase. Take as reference Algorithm 2.8 for an high level description. Given a connected component C_v, and the minimum node identifier $v_{min} \in C_v$, the value v_{min} is sent to all the members of the connected components and the identifiers of the members of the connected components are sent to v_{min}. If we wanted to build this algorithm in a vertex-centric way, we would need to:

- acquire the most recent connected components from the vertex neighbors;
- merge overlapping sets with the vertex's one, finding a bigger connected component;
- send the new connected component of the vertex.

The *emit*() function and the first loop are the core of the problem, so it will be described how and if it could be implemented on GraphLab, PowerGraph and Apache Spark. Please notice that the amount of communication can become prohibitive, so data compression techniques could be necessary, but are not covered here.

GraphLab. This framework does not support neither evolving graphs nor non-neighbor accesses, thus, is difficult to design a proper solution. As for GraphLab properties, vertices can be active or re-activated until a global convergence goal is not reached, so we could use the Sync function to perform this check. Also, the asynchronous computation would be beneficial in this context because connected components may largely vary in dimension so that the vertex handling the biggest component requires more execution with respect to vertices handling small components. Similarly to the PageRank algorithm, it is preferable to adopt the consistency level that guarantees, when executing the program on a vertex v, the consistent access to the contexts of all the neighbours of v.

PowerGraph. In [78] evolving graphs are not handled. Due to this, we have the same difficulties as in GraphLab. However, the advantages of GAS model for HASH-TO-MIN are considerable, especially for nodes having an high degree or connected with such kind of vertices. Indeed merging the neighborhood connected components (CC) can require a non trivial amount of time, and parallelizing the Gather function, adopting the divide-et-impera technique, is reasonable and possible by means of vertex-cuts. As for timing, asynchronous engines are preferable compared to the synchronous one, due to computation unbalancing and dynamic workload.

Communication overhead is a big issue, so we could try "smart" heuristics that minimize the number of edges/vertices (vertex-cut) shared between machines, but it would be the same as solving the problem. Every vertex needs to send C_v only one time for each update execution, so it can be considered the asymptotically superior limit for a vertex communication. Furthermore, authors of HASH-TO-MIN [151] agree that the amount of information shared between machines will be approximately the same (linear in the dimension of the biggest connected component). This is a problem which could require algorithmic optimization in order to perform well under communication, rather than working on what kind of framework use.

Apache Spark. It is possible to implement HASH-TO-MIN within GraphX however a complete re-indexing of all the RDDs is required in each iteration.

Conversely, the MapReduce paradigm offered by Apache Spark is a perfect fit for the HASH-TO-MIN algorithm. An implementation of the algorithm can be found in the following sections because this algorithm is one of the competitors we have considered when evaluating one of the algorithm of this thesis. However, due to the HASH-TO-MIN nature, this algorithm encounters unbalance workload, because the components are deployed entirely to the vertex elected to be the identifier of the component. This can be problematic when a component is of large size. In the original paper [151] this is mitigated by exploiting a common technique exploited in MapReduce called secondary indices. This, permits to load the data relative to a vertex in an ordered way and just one time to avoid memory errors. However, this very same technique cannot be implemented in Apache Spark. Using the RDD abstraction, it is not possible to exploit the same feature because data are kept in memory and the data relative to a partition must fit in the memory of a single machine.

2.4 Conclusion

In this chapter we set the concepts and the elements underpinning the contributions of this thesis. We covered several frameworks supporting TLAV and discussed how it is possible to implement selected algorithms on three of these frameworks.

In this thesis, we choose Apache Spark as the reference architecture. It is one of the framework having the highest usage increase, in the last years it has been exploited by many industries and academias. Also, it provides a rich set of APIs that share a common layer to handle the data. In conclusion, we believe that this framework is becoming a reference for the implementation of distributed graph algorithms. Its exploitation enable us to share our code in a vast community of developers and to compare our algorithms with different competitors' implementations.

Part II

TLAV Algorithms

Chapter 3

Duckweed: Distributed Current Flow Betweenness Centrality

Centrality measures are important tools in graph analysis by providing information on the structural prominence of nodes and edges. They support the identification of the key elements of the graph. As an example, consider a social network: it could be interesting to find the most important actors in large social interaction graphs; or in a data network could be worth to find the nodes that are subject to the highest traffic to prevent network congestion and disruption.

In the scientific literature, there are not too many works focusing on the computation of betweenness centrality in distributed scenarios. There exist different definitions of betweenness centrality. The mostly used Betweenness Centrality [70] measures the importance of a node by taking into account the number of times it lies on the shortest path between two other nodes. A limit of shortest-paths based measures is that they do not take into account the information spread occurring along non-shortest paths, hence, they are not the best choice when information conveying is governed by other rules. To overcome this limitation, in literature have been proposed different measures based on the information flow along the graph.

One of these measures proposed both by Newman [139] and by Brandes et al. [27]. They proposed to model the network as an electric circuit where a current flow is injected to a source node and exits to the target node. The resulting index, the *current flow betweenness* named also *random-walk betweenness* is able to compute contributions from all paths existing between the source and the target node. The current state-of-the-art for the computation of the current flow betweenness includes both exact [140, 27] and approximated [27, 24, 12] approaches. However, these solutions are all centralized, and their applicability is limited only to small graphs.

In this chapter we propose DUCKWEED a solution aimed at addressing such limitations. It consists in a novel algorithm for the distributed computation of the approximated current flow betweenness centrality. The main contributions of this chapter are the following:

- to the best of our knowledge DUCKWEED is the first proposal to compute current flow betweenness centrality in distributed environment;

- DUCKWEED makes use of an approximation technique in order to provide a valuable result of the current flow betweenness centrality in large graphs;

- DUCKWEED can be implemented in both distributed frameworks [21] for the analysis of large graphs (such as Apache Spark [193] or Hadoop [164]) and in peer-to-peer networks;
- we show how is possible to port DUCKWEED on a real distributed environment exploiting the GAS decomposition.

The idea, definition and evaluation of DUCKWEED have been published in IEEE Conference on Self-Adaptive and Self-Organizing Systems with the title *"Distributed Current Flow Betweenness Centrality"* [121]. The porting of the algorithm in Apache Spark has been published in the International Conference on Algorithms and Architectures for Parallel Processing with the title *"Current flow betweenness centrality with Apache Spark"* [21].

3.1 Related work

The computation of the betweenness centrality index is intrinsically expensive. In fact, to be determined it requires the computation of all the shortest paths. The naive centralized algorithm requires $\theta(n^3)$ time and $\theta(n^2)$ space, where n is the number of vertices. An improvement to the basic solution has been proposed by Brandes [26] in 2001. In his proposal, he introduces the notion of vertex dependency, recursively defined on the whole structure of the graph. Following this approach it is no longer required the combinatorial counting of all paths and $O(n + m)$ space and $O(nm)$ time bounds, respectively, are obtained for undirected graphs (with m the number of edges). Another approach is based on the definition of algorithms for computing approximated values of the index. Riondato *et al.* [153] proposed two efficient randomized algorithms for the estimation of betweenness based on random sampling. These algorithms offer probabilistic guarantees on the quality of the approximation. To bound the size of the sample exploited to achieve their approximated result, they rely on the results from the Vapnik-Chervonenkis theory, which allows to use small sample sizes.

The current flow betweenness centrality has gained momentum in the last years as an alternative index to measure centrality of nodes in a graph. Similarly to the classical betweenness centrality, the straightforward algorithm to determine current flow centrality is to compute information flows for all the possible pairs of node in the graphs. Newman [139] and Brandes *et al.* [27] provided a formulation derived from Kirchoff's law of current conservation, in which edges are resistors with a given conductance, and the nodes are junctions between resistors. They propose an algorithm for estimating the current flow betweenness centrality having a computational complexity of $O(I(n-1) + mn^2)$, where $O(I(n-1))$ is the complexity to invert a $n \times n$ matrix. Moreover, they propose an approximated version of the algorithm that selects uniformly at random a small fraction of all pairs $s \neq t \in V$. The bound on the number of pairs is computed by exploiting the Hoeffding bound [83].

Bozzo and Franceschet [24] [25] proposed an algorithm for current flow betweenness centrality that reduces the complexity deriving from the inversion the Laplacian matrix describing the graph by choosing a subset of its eigenvalues and eigenvectors. Avrachenkov *et al.* [12] introduce the α-current flow betweenness centrality by adding a "ground node" to the original graph, and connecting each node to it. The introduction of the ground node simplifies the computation and leads to a reduction of the complexity of the matrix calculations. In addition, they approximate the betweenness by randomly selecting a subset of all possible pairs, similarly to Brandes *et al.* [27]. They also introduce the truncated α-current flow betweenness, an approach in which the scores on the edges starting from the source of the flow are not considered in the computation of the betweenness. They proved empirically that the resulting estimation increases the correlation with the exact current flow betweenness.

Despite the optimizations introduced by the solutions mentioned above, the computational cost to determine path-based centrality indices requires the adoption of distributed solutions when dealing with large graphs. Lehmann and Kaufmann [103] propose a general framework for defining decentralized algorithms that compute centrality indices. To this end they take into account four different centralities, closeness, stress, graph and betweenness, with emphasis on the betweenness centrality.

As far as we know, there is no distributed algorithm for the current flow betweenness centrality. Rather, novel definitions of the centrality indices have been proposed, which are suitable to be computed in a distributed environment. Wehmuth and Ziviani [181] redefined the closeness centrality index of a vertex by considering its h-neighbourhood, i.e., the vertices within a radius h around it. The centrality value of each vertex in the network is defined as the sum of the degrees of the vertices in its h-neighbourhood, i.e., the volume of its h-neighbourhood. This notion of centrality is equivalent to the degree centrality when $h = 0$. The distributed algorithm consists in a TTL-restricted flooding of the degree of each node within its h-neighbourhood. The experimental results show a high degree of correlation between this notion of centrality and the closeness centrality. Kermarrec *et al.* [97] introduces a novel notion of centrality based on random walks, called *second order centrality*. According to it, each node computes its centrality index by starting an unbiased random walker and counting how much time the walker requires to return to the node. Their approach collects a certain number of this measure in order to calculate the standard deviation. This calculation is performed incrementally each time the walker returns to the node, improving the estimation of the centrality of the nodes over time.

3.2 Preliminaries

In this section we introduce the notions required for the definition of the main theoretical framework at the basis of our approach. We first introduce the Random Walk Betweenness and then we present the relation between elementary electric network theory and random walks, which is at the basis of the definition of the Current Flow Betweenness.

3.2.1 Random Walk Betweenness

Let us consider a network and suppose that a node s generates an information (conveyed as a message) whose target is node t. Each node receiving the message propagates it to one of its neighbours, chosen uniformly at random. This strategy is usually referred as random walk. The notion of Random Walk Betweenness Centrality introduced by Newman [140] measures, for a given a node n, the expected net number of times a random walker passes through n on its way from s to t, averaged on all s and t. As Newman states, when computing the net number of times, two visits of the walker to the same vertex coming from opposite directions must be cancelled out. This avoid scenarios where the walker passes forth and back a vertex many times, without actually going anywhere.

3.2.2 Current Flow Betweenness

Doyle and Snell [61] give an exhaustive presentation of the relations existing between random walks end electric networks. We briefly summarize the main results of their work in this section.

Let $G = (V, E)$ be an undirected graph, where $E \subseteq V \times V$, with $n = |V|$ vertices and $m = |E|$ edges. We assume that there are no self-loops from one vertex to itself and no pairs of vertices connected by multiple edges. Each edge $(i, j) \in E$ connecting vertices i and j has a weight $w_{i,j}$. The matrix $A_{i,j}$ is the adjacency matrix of the graph, i.e. $A_{i,j} = 1$ if and only if there exist an edge connecting vertices i and j.

An electrical network may be represented by a graph by assigning to each edge a positive weight indicating the conductance of the corresponding electric wire (or the resistance of the wire, which is the inverse of the conductance). According to this representation, the vertices of the graph are junctions between resistors. For the sake of simplicity, we consider unitary conductance (or resistance), i.e., $w_{i,j} = 1$, $\forall i, j$.

In particular, we are interested in how current flows through the network, when injected to a source node s and picked up at a target node t, for all possible choices of s and t.

Definition 2. *We define a current flow $F^{(s,t)}$ over the graph G as follows:*

- *s is the source of the flow, the current enters the network through it;*
- *t is the target of the flow, the current leaves the network through it;*
- *$u_i^{(s,t)}$, called supply vector, is a vector such that $\sum_i u_i = 0$ and $u_s = -u_t = 1$:*

Let $v_i^{(s,t)}$ be the potential at node i for $F^{(s,t)}$. Kirchhoff's law of current conservation states that the current that enters into a node is equal to the current that flows out of it. This implies that the potentials of a node satisfy the following equation for every node i:

$$\sum_j A_{ij}(v_i^{(s,t)} - v_j^{(s,t)}) = u_i^{(s,t)} \tag{3.1}$$

$$v_i^{(s,t)} = \frac{\sum_j A_{ij} v_j^{(s,t)} + u_i^{(s,t)}}{\sum_j A_{ij}} \tag{3.2}$$

By considering only unitary resistances as stated above, we have $A_{ij} = 1$ if exists the edge (i, j), otherwise 0 and that $\sum_j A_{ij} = \deg(i)$.

Given the above, let us consider the current flow that passes through the vertex when a unit of current is injected in a source vertex and removed from a target vertex, averaged over all source-target pairs [27, 140]. Doyle *et al.* [61] show that this current flow is equal to the net expected number of times that a walker, starting at s and walking until it reaches t, will pass through that vertex. The random walker betweenness can be therefore computed by considering the electrical circuit associated to the graph and by applying the laws of electrical circuits. The resulting betweenness index is also referred as Current Flow Betweenness.

To compute the current flow betweenness of a vertex i, $\neq s, t$, it is therefore required to compute the current flowing through i, which, given a flow $F^{(s,t)}$, is defined as half of the sum of the absolute values of the currents flowing along the edges incident on that vertex:

$$I_i^{(s,t)} = \frac{1}{2} \sum_j A_{i,j} \mid v_i^{(s,t)} - v_j^{(s,t)} \mid \tag{3.3}$$

The current-flow betweenness centrality b_i [27, 140] is the average of the current flows over all the source-target pairs:

$$b_i = \frac{\sum_{s<t} I_i^{(s,t)}}{(1/2)n(n-1)} \tag{3.4}$$

Bozzo *et al.* [25] show how to solve the system of equations 3.2 by exploiting classical matrix calculus when considering all the possible source and target pairs. However, when the graph is very large, this calculus may exceed the computational capability of a single machine. On the other hand, we note that a distributed computation of the system of equations 3.2 is feasible, because each node requires only local information, i.e. information about its neighbours, to iteratively compute the value of its potential for a given s and t pair, and calculates its betweenness by means of its currents flowing according to Equation 3.3. This observation is at the base of the distributed approach we present in the next section.

3.3 Duckweed

As far as we know, current state-of-the-art approaches for current flow betweenness centrality (CFBTW) are centralized, and their applicability is limited to small graphs [27, 12]. In this work we present our proposal to fill this gap. We propose DUCKWEED, a novel distributed approach that computes an estimation of the current flow betweenness centrality, which is suitable for large graphs. Approximate CFBTW has a practical applicability, essentially for two reasons: (i) computing a precise CFBTW for a large graph is very time consuming, and (ii) many applications of centrality indexes require to find a ranking of the top-most central nodes, rather than the exact values of centrality for each vertex.

3.3.1 Computational Model

The computation performed by DUCKWEED follows a vertex-centric approach in which the nodes of the graph are considered as the unity of computation. We assume that nodes of the graph are processed periodically and asynchronously. Nodes do not have access to the entire graph, but only to their immediate neighbours in the graph, and to a small set of other nodes, provided by means of information diffusion protocols. DUCKWEED does not

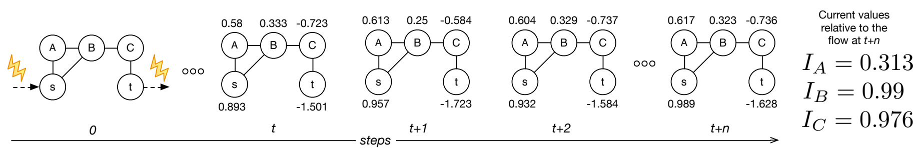


FIGURE 3.1: Evolution of a flow from s to t over time

require any kind of shared memory, as nodes communicate only through explicit messages. This makes DUCKWEED suitable for graph computing frameworks, and for peer-to-peer networks, in case each node is assigned to a peer of the network.

3.3.2 The DUCKWEED approach

The main idea behind DUCKWEED is to exploit Kirchhoff's law to calculate the electric potentials of all the vertices, as building blocks for the computation of CFBTW for the entire graph. The computation is organized in such a way that each node, locally and autonomously, can compute its own electric potential and its own value of CFBTW centrality. DUCKWEED is based on the following two main modules: the **Flow computation** and the **Centrality computation**.

The Flow computation drives the creation of flows and the computation of potentials. In particular, for a given $F^{(s,t)}$, the corresponding potential can be computed using Equation 3.2 and exploiting only the knowledge about the neighbours potentials (due to the fact that the conductance between two nodes is zero if they are not neighbours, see considerations in Section 3.2). The detailed description of the Flow computation is given in Section 3.3.3.

For Centrality computation, each node collects the potentials of its neighbours to compute independently the actual values of the CFBTW. In particular, to compute the incremental CFBTW for a vertex i, Centrality computation exploits Equations 3.3 and 3.4 in the following way:

$$b_i^k = \frac{(k-1)b_i^{k-1} + I_i^{(s,t)}}{k} \tag{3.5}$$

where k is the amount of computed flows known by the vertex. The fact that DUCKWEED computes the CFBTW incrementally yields two relevant impacts. First, it is possible to have an idea of what are the most central nodes without waiting for the computation to be completed. Second, it is possible to define an automatic mechanism of termination, so that DUCKWEED stops the computation when it reaches a given level of approximation. A detailed description of the Centrality computation is provided in Section 3.3.4.

3.3.3 Flow computation

This section describes in details how the computation of a single generic flow $F^{(s,t)}$ is realized in DUCKWEED. The computation of a single flow is then extended to the concurrent computation of n flows, considering that each flow $F^{(s,t)}$ can be identified uniquely using its source and target vertex, respectively s and t, by convention $s < t$.

Algorithm 3.1 shows the pseudo code of the Flow computation in a node. The node first receives the information about the flows computed at the

Algorithm 3.1: Flow Computation

```
Data: $F$: the set of flows known by the node
1  $R \leftarrow$ receive flows from neighbours
2  $N \leftarrow$ flowCreation()
3  $F \leftarrow F \cup R \cup N$
4  forall the $f \in F$ do
5      update potential for $f$
6      $\Delta f \leftarrow$ difference with potential of the previous iteration;
7      if $\Delta f < \mathscr{D}_\epsilon$ then
8          mark $f$ as completed
9      end
10 end
11 send $F$ to neighbours
12 CentralityComputation()
13 $F \leftarrow F \setminus \{f$ is completed$\}$
```

previous iteration by its neighbours, and then, if necessary, creates new flows. Subsequently, it updates the potential relative to each flow and marks a flow as terminated if the changing in potential is under the threshold $\mathscr{D}_\epsilon$. Further, the node sends the set of updated flows to its neighbours, it updates its current flow betweenness centrality value, and removes the completed flows from the local state. For the sake of the explanation, we divide the Flow computation into three main steps: *creation*, *update*, and *termination.* In the following we describe these three steps.

Creation

The creation of a flow $F^{(s,t)}$ is done locally by each node, by considering itself as the source s and choosing a target t among the other nodes. Since the algorithm is fully distributed, deciding *when* to start a new flow, and *which* node to choose as target are relevant aspects.

To define when a node shall create a flow, we designed DUCKWEED such that the number of concurrent flows in the network are probabilistically limited in any given point in time, so to avoid to oversaturate the nodes. To this end, we define the system-wide parameter φ as the number of maximum concurrent flows active on each node; the number of flows active on a generic node i is defined as φ_i. In addition, to avoid the re-creation of an already computed flow, each vertex u maintains a list of all the flows processed for which u is the source or the target of the flow.

The flow creation relies on a combination of different gossip-like protocols, organized in layers (as usual), to realize the distributed computation of the flows. These protocols are popular in peer-to-peer networks, as they proved to be efficient solutions to tackle very different problems, ranging from distributed data clustering [14, 15, 49, 138], resource and service discovery [34, 13], online games [42]. Moreover, they can also be implemented and exploited in graph computing frameworks. These protocols are based on a *random node sampling* layer that provides a selection of random nodes from the graph, similarly to the random peer sampling gossip protocols used in distributed applications [89, 179]. In addition, the Flow computation exploits the following protocols: (i) the *size estimator* implements a well-know protocol to count the number of peers in a gossip fashion scenario [126, 88]; note that, at any given time, a node has its own estimation of the graph size, which in general is different from the one of the other nodes. In the following, we refer to the size estimation of node i as n_i. (ii) The *average betweenness* provides an estimation of the average CFBTW of the whole graph. This protocol is similar to the *size estimator* protocols, and we refer to the CFBTW average estimation of node i as BTW_i. (iii) The *k-partitioning* performs a distributed k-way partitioning on input graph, similar to the one proposed in [149]. The result of the partitioning is to colour the nodes of the graph with different colours according to their partition. The *k-partitioning* and *average betweenness* are optional, as they depends on the particular strategy used for the generation of the flow.

Using these concepts, we defined three different strategies to generate a new flow:

- *random*: a node generates a new flow $F^{(s,t)}$ with a certain probability p_{random}, computed as the following:

$$p_{random} = max\left(0, \frac{\varphi - \varphi_i}{n}\right) \tag{3.6}$$

  where n is the number of (estimated) nodes in the graph. If the node is supposed to create a flow, selects t from its random sampling view.

- *adaptive*: the aim of this strategy is to favour the creation of flows between nodes that are at the border of the graph, and to disfavour flow betweens nodes in the center. Recall that to be source (or target) of a flow, do not improve its own betweenness value. In other words, this strategy tries to accelerate the computation of the nodes with already an higher betweenness. To implement this strategy, a generic node i considers the local estimation of the average CFBTW for the whole graph (BTW_i). If i has its current centrality value larger than BTW_i, then it generates no flow. Otherwise, it creates a flow with probability $p_{adaptive}$, defined as the following:

$$p_{adaptive} = max\left(0, \frac{\varphi - \varphi_i}{n/2}\right) \tag{3.7}$$

  Note that $p_{adaptive}$ is twice p_{random}, so to compensate the node with no chance to create a flow. A node u elected to generate a new flow sets $s = u$ and t equals to the node having the smallest CFBTW value in its random sampling view.

- *partitioner*: the aim of this strategy is to favour the generation of flows with s and t being in different partitions of the graph. Intuitively, nodes on the path between two nodes of different partitions are more likely to have higher betweenness centrality. A node u generates a new flow with a probability equals to p_{random} and $s = u$, and exploits the colouring from the *k-partitioning* protocol to choose t from the random sampling view, such as t's colour is different from s's colour.

Update

During every step, each vertex collects all the flows received by its neighbourhood in the previous step and applies Equation 3.2 to update the potential of each flow. The updated potentials, along with the flow identifiers, are then sent to all the neighbourhood to continue the computation in the next step. This computation can be easily extended to handle weighted graphs introducing the weights in Equation 3.2 as described by Bozzo et al [25] using as A the weighted adjacency matrix.

Figure 3.1 depicts a flow computation where a unit of current is injected in node s and removed from node t. In step $t+1$ the node B receives from its neighbours $\{A, C, s\}$ the potentials calculated on step t equals to $\{0.58, -0.723, 0.893\}$. It applies Equation 3.2 and it obtains a potential equals to 0.28. This potential will be available to its neighbours in the following step.

Termination

The termination of a flow is regulated by the system-wide parameter $\mathscr{D}_\epsilon$, which represent the minimum difference in the potential of a flow between two consecutive steps. Therefore, on a generic vertex i, the flow $F^{(s,t)}$ is completed when both the following conditions are verified:

- $v_i^{(s,t)}$ has converged to at least $\mathscr{D}_\epsilon$;
- all the neighbours of i have converged to at least $\mathscr{D}_\epsilon$.

When a flow is marked as completed, the vertex stops the propagation of the potential relative to such flow. This eventually terminates the update of the flow $F^{(s,t)}$ in all the vertices of the graph.

3.3.4 Centrality computation

In the previous section, we described how the computation of a single flow is performed in DUCKWEED. This section describes how DUCKWEED combines the results coming from the computation of multiple flows to incrementally compute the CFBTW for the nodes of the graph.

When the computation of a generic flow $F^{(s,t)}$ is completed (as described in Section 3.3.3), vertex i uses Equation 3.3 to compute $I_i^{(s,t)}$ for the $F^{(s,t)}$. Vertex i then updates its current-flow betweenness b_i using $I_i^{(s,t)}$ and Equation 3.5. Note that the source s and target t do not consider $I_s^{(s,t)}$ and $I_t^{(s,t)}$ for their betweenness calculation. Since we provide an estimation of the CF-BTW by computing only a subset of the all possible flows, this assumption is required to avoid biased computation of centrality, which would occur if considering any value for $I_s^{(s,t)}$ and $I_t^{(s,t)}$, for example equals to 0 or 1 as suggested in [24]. Figure 3.1 (on the right) show the current flowing on each node of the graph calculated with Equation 3.3 when the flow is completed at step $t+n$. Considering node B we obtained:

$$\begin{aligned} I_B &= \frac{1}{2}(|0.323 - 0.617| + |0.323 - 0.949| + |0.323 + 0.736|) \\ &= 0.99 \end{aligned} \tag{3.8}$$

This value will be used by node B in Equation 3.5 to incrementally calculate its current flow betweenness value.

The precision of the CFBTW increases as more and more flows are updated and completed. To detect when the results of the CFBTW are precise enough, and hence to terminate the generation of new flows, it is possible to define a parameter k to stop the generation of new flows when all the nodes have already processed k flows. Otherwise it is possible to introduce more sophisticated techniques to let each node decide autonomously when stopping flow generation and hence terminate the computation. Note that the termination of the computation of the CFBTW is *different* from the termination of a single flow. In the former case we refer to the termination of the whole system (i.e. no more flows are created), in the latter (described in Section 3.3.3) we refer to the termination of a *single* flow.

In order to equip each node with the possibility of locally terminating the computation of the CFBTW, we adapt the centralized mechanism introduced by Brandes et al. [27] to a distribute context. This method exploits the Hoeffdings bound [82] to identify a value of k such that the error ϵ_{BTW} on the betweenness values b_v on each node is sufficiently small. Hoeffding's bound gives:

$$\mathbb{P}\left(\left|\frac{c^*}{k}\sum_{i=1}^{k} X_v^{(i)} - b_v\right| \geq \epsilon_{BTW}\right) \leq \frac{2}{n^{2l}} \tag{3.9}$$

when choosing:

$$k = l \cdot \lceil (c^*/\epsilon_{BTW})^2 \log_n \rceil \tag{3.10}$$

for arbitrary l, where $c^* = n/(n-2)$ and $X_v^{(1)}, ..., X_v^{(k)}$ are independent random variables that return $F^{(s,t)}$, for a pair $s \neq t$, picked uniformly at random. We adapted the above calculation to be suited for the distributed environment of DUCKWEED. To pick a flow uniformly at random we exploited a random peer sampling layer and each node, for instance using the random flow creation heuristic, has the same probability to generate a flow with a node taken from the random peer sampling. Also, each node has its estimation k_i about the number of flows already computed in the system equals to the number of flows that i has computed. The size of the network n can be estimated in the same way explained in Section 3.3.3. Given the above, each node can autonomously calculate k and stop the flow generation when $k_i > k$.

3.4 Experimental evaluation

The aim of the evaluation is to verify the effectiveness of DUCKWEED both in terms of the quality of results, and on its applicability on large graphs. The evaluation of DUCKWEED was conducted by means of simulations. We implemented DUCKWEED, and all the associated protocols, on the discrete-event PeerSim [137] simulator.

3.4.1 Evaluation of Correlation

To conduct an evaluation of DUCKWEED we measured the correlation of the results provided by our approach with the ones given by the other algorithms. To measure the correlation, we use the KENDALL TAU metrics, which is a measure of rank correlation. It measures the degree of similarity between two distinct rankings by assigning a value in the range $[-1, 1]$. If two rankings have the same values, the coefficient equals to 1, whereas if the disagreement between the two rankings is perfect (i.e., one ranking is the reverse of the other) the coefficient has value -1. For this evaluation we generated 3 graphs of 1000 nodes each, using the Snap library [107] with

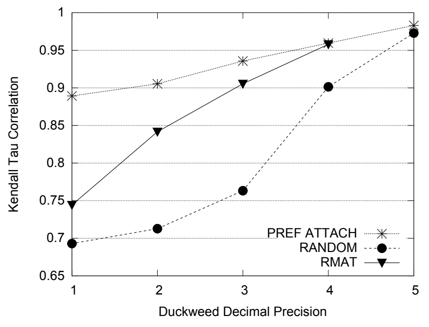


FIGURE 3.2: Kendall Tau Correlation with NetworkX

the following strategies: preferential attachment (Barabasi-Albert), random (Erdos-Renyi) and RMAT[1] [44]. These graphs are purposely small to ease the computation of exact values for all the centrality measures.

Validation against NetworkX

In this first set of experiments we validate the precision of the current flow betweenness centrality provided by DUCKWEED against the one provided by NetworkX [160]. NetworkX is a popular tool to analyse network structure and it provides an implementation of the algorithm presented by Brandes *et al.* [27] to calculate the current flow betweenness centrality. We run NetworkX on the three aforementioned graphs to compute the exact CFBTW. Then, we run DUCKWEED on the same graphs, computing all flows and varying the decimal precision of $\mathscr{D}_\epsilon$ up to five decimals. We compared the *top* 100 nodes of NetworkX and DUCKWEED with the KENDALL TAU metrics. It is worth to point out that in the context of all our experiments we refer to the *top X* nodes to indicate the X nodes that received the highest value of centrality according to a given measure.

Results are presented in Figure 3.2. It is evident that in all datasets the correlation increases when increasing the decimal precision. This is an expected result because increasing the precision on each flow leads to a more precise calculation of the current flow betweenness. The value of centrality, computed by DUCKWEED on the preferential attachment graph, exhibits the best correlation in all the configurations. In fact, even with a single-decimal precision the value of its correlation with NetworkX is around 0.9. Conversely, the random graph exhibits a correlation of 0.7 when adopting single-decimal precision, however the correlation value rapidly increases when using additional decimal precision. It is worth to notice that with a precision of 4 decimals, all the datasets exhibits a correlation value greater than 0.9, suggesting that DUCKWEED correctly approximates the current flow betweenness centrality also with a reasonable value of $\mathscr{D}_\epsilon$.

DUCKWEED complete vs approximated

This test compares the complete version of DUCKWEED (i.e. the one in which all flows are computed), against the approximated version, in which only a portion of all the flows is computed. The results presented in Figure 3.3 show the KENDALL TAU correlation of the top 100 nodes between the complete and

[1]RMAT graphs were generated with parameters (.6, .1, .15, .15)

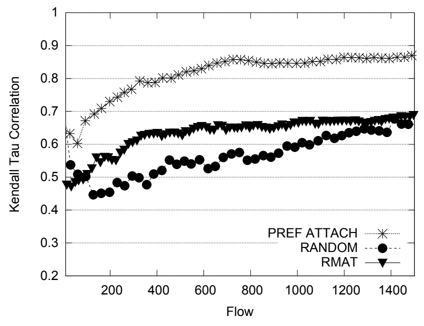


FIGURE 3.3: Kendall Tau Correlation: complete vs approximated

the approximated version of DUCKWEED, when the amount of flows executed varies.

The experimental results show that the correlation increases quite fast becoming closer to the complete one, even using a number of flows that represents less than the 1% of all the possible flows. We obtained the best results when using the preferential attachment graph. As can be observed, it starts from 0.6 and reach 0.9 in step 200, whereas both the RMAT and random graphs achieve a correlation value not greater than 0.7.

3.4.2 Convergence Time

In Section 3.4.1 we evaluated the results of DUCKWEED when varying the $\mathscr{D}_\epsilon$ parameter. We found that the results achieved using an increased value of $\mathscr{D}_\epsilon$ have an increased correlation with the exact values of current flow betweenness centrality. Here, we evaluate the impact of $\mathscr{D}_\epsilon$ on the time spent to *terminate* a flow. We measure this time in terms of STEP, i.e., the number of times DUCKWEED is executed on each node.

Figure 3.4 presents the number of STEP required by adopting different values of $\mathscr{D}_\epsilon$ for the analysis of the very same graph. The results presented are computed as an average of 1000 flow computations. As expected, an higher decimal precision corresponds to an higher amount of STEP. In particular, a decimal precision greater than 3 leads to a sensible increase of STEP, that reach its maximum of 22 STEP with a decimal precision of 5 on the RMAT graph. Anyhow, it is worth to notice that with a decimal precision below 3 the amount of STEP is always less than 5 for any given datasets. This is quite interesting, in fact with a decimal precision of 3, DUCKWEED already achieves good correlation values with the exact values.

3.4.3 Centrality on Large Graphs

The evaluation of certain centrality measures in large graphs is an issue by itself. In fact the exact computation of some centrality measures can be very costly, from a computational viewpoint, when the size of the graph is large. A common strategy to evaluate the centrality of a given set of nodes, without having to compute exact results, is to remove the whole set from the graph and measure the amount of connected components (CCs) characterising the graph after this process [12, 55].

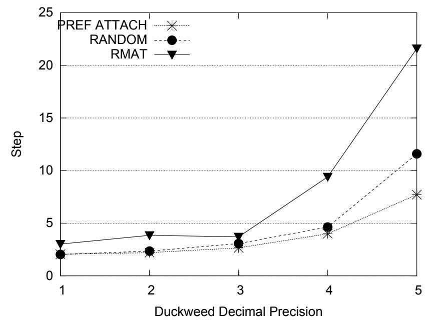
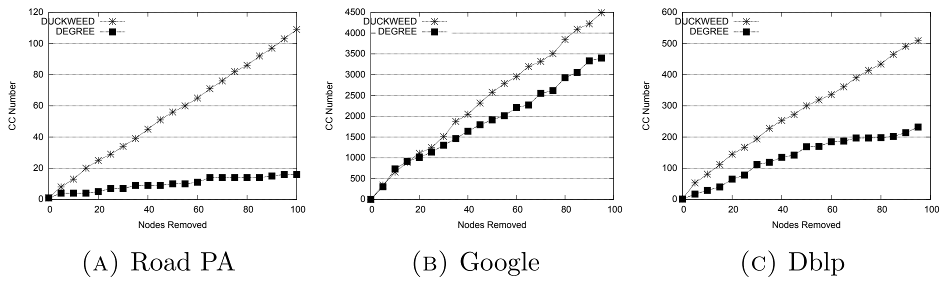


FIGURE 3.4: Convergence Time



(A) Road PA (B) Google (C) Dblp

FIGURE 3.5: CC Number (large number is better)

Clearly, an higher number of CCs corresponds to a better identification of nodes responsible to maintain the network connectivity. In the following we refer to this metrics as CC NUMBER.

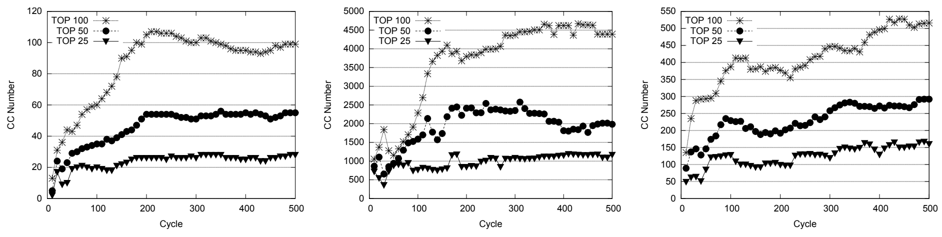
Using this metrics we conducted two different sets of experiments. The first testbed is aimed at evaluating, in terms of CC NUMBER, the quality of the results produced by DUCKWEED with respect to the result obtained by the `Degree` centrality. Previous results for CFBTW approximation provided result for graphs in the order of 10^3 nodes in Brandes *et al.* [28] and 3×10^3 in Avrachenkov *et al.* [12]. We chose the `Degree` centrality as the baseline since it is the least expensive in terms of computational complexity among popular centrality measures, and therefore suitable to be computed on large graphs.

For our evaluation we considered three graphs taken from the SNAP [105] website: (i) the *RoadPA* graph represents the roads network of Pennsylvania; (ii) the *DBLP* graph provides a co-authorship network of paper indexed by the DBLP service; (iii) the *Google* graph represents web pages and hyperlinks released in 2002 by Google.

From these graphs we extracted each largest connected component, and we use these as the input for our evaluation. The connected components were extracted using our algorithm described in Chapter 4, and measure respectively $1,087,562$, $317,080$ and $855,802$ nodes.

The results we achieved show that DUCKWEED is able to provide good results, i.e., identifies nodes with a centrality index sensibly higher than the one provided by `Degree` centrality.

The second experiment evaluates the approximation provided by DUCKWEED during the simulation. More in details, Figure 3.6 shows the amount of CC NUMBER that would be introduced if the top 100, 50 and 25 rankings



(A) Road PA (B) Google (C) Dblp

FIGURE 3.6: CC Number: Graph Cut While Increasing Duckweed approximation

were removed by the graph. The timespan considered focuses on the initial 500 STEP of the simulation. As expected, the top 25 nodes require less STEP to be identified with respect to the top 100. In fact, let us consider, for instance, Figure 3.6a. It can be noticed how after 50 STEP the top 25 curve identifies 20 CC NUMBER, a value that increases only marginally in the remaining of the simulation. Conversely, the top 100 curve rapidly increases till STEP 200 and then remains stable right to the end. In conclusion, the results presented in Figure 3.6a show that DUCKWEED is able to achieve a good level of approximation in a reasonable number of STEP (around 200). The results reported in figures 3.6b and 3.6c, are slightly different but still confirm the ability of DUCKWEED in quickly finding nodes that if removed lead to the creation of a consistent amount of connected component.

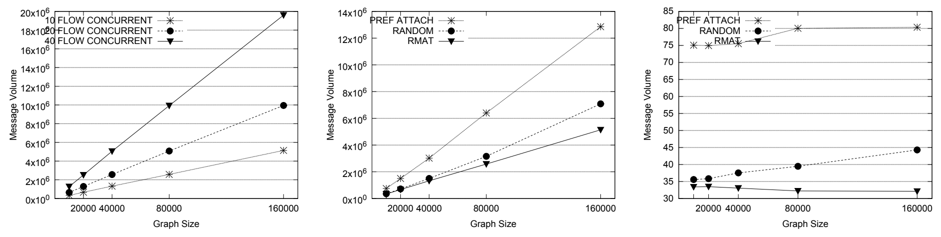
3.4.4 Message Volume

In order to evaluate the cost of DUCKWEED, in terms of messages required for its execution, we measured the amount of messages as the total number of flows sent during a single STEP, and we called this metrics MSG VOLUME. To conduct our evaluation we considered three types of graph, each one generated according the following models: preferential attachment, random and RMAT. For each type we generated five graphs each having a different size: $\{10000, 20000, 40000, 80000, 160000\}$. We run the experiments by varying the number of concurrent flows, per step, in the following set: $\{10, 20, 40\}$.

As can be noticed in Figure 3.7a the value of MSG VOLUME for the RMAT graph increases linearly with the graph size (the results obtained using preferential attachment and random graphs are not included because are equivalent to the RMAT ones). This is easy to see in Figure 3.7a, in fact with 10 concurrent flows the MSG VOLUME value equals to 2.5×10^6 for a graph size of 80000 nodes and about 5×10^6 for a graph of 160000 nodes. Similarly, an increment on the number of concurrent flows leads to a linear increment of MSG VOLUME.

Figure 3.7b presents the results achieved using different types of graphs and fixing the number of concurrent flows to 10. We observed that all the graphs behave a similar way when increasing the network size. However, the preferential attachment graph requires the largest MSG VOLUME, whereas RMAT the smallest. This is essentially due to the total amount of edges in each graph: the greatest in preferential attachment and the lowest in RMAT.

Figure 3.7c shows the MSG VOLUME *per node*, with the number of concurrent flows fixed to 10. The results we achieved show that the network



(A) Increasing graph size (B) Different Graph Model (C) Volume Per Node

FIGURE 3.7: Message Volume Evaluation

size affects only marginally the MSG VOLUME per node. In particular, with the preferential attachment and RMAT the MSG VOLUME remains constant, whereas it slightly increases with random graph. These results suggest that DUCKWEED scales in terms on MSG VOLUME with the size of the graph, which makes it suitable for computation on large graphs.

3.4.5 Flow Creation Strategy

This last set of experiments is aimed at evaluating the impact of the flow creation strategies (random, adaptive, partitioner, described in Section 3.3.3) on the identification of central nodes. To this end we executed a simulation of 5000 flows with all the graph types, but fixing the graph size to 160000 nodes. The CC NUMBER achieved by DUCKWEED were sampled every 10 completed flows by removing the top-{25,50,100} most central nodes. From the sample, we computed the minimum, maximum, average and standard deviation. An overview of the results for the Random graph are reported in Table 3.1 (the results with the other graph types are omitted as the results were almost the same).

Even if Brandes *et al.* [28] observed that from a theoretical viewpoint a random selection of the flow performs better than more complex heuristics, from the results we achieved it can be observed that the adaptive strategy provides the best result on average. However, some aspects are worth to notice: (i) the random strategy considered by Brandes *et al.* exploits a "perfect" uniform random, rather, the random nodes in DUCKWEED are taken from the random peer sampling service, that only from a theoretical point of view converges to an uniform random; (ii) from a temporal analysis of the results, we observed that the differences among the strategies is more evident during the initial steps of the simulation when the number of completed flows is still low. Instead, at the end of the simulation all the strategies achieved similar results.

Overall, these considerations suggest that in a distributed context where an uniform random distribution is not easily available, a more elaborate strategies can have an impact on the computation of the CFBTW, especially if does not require to compute complex global measures of the graph. Instead, the considerations of Brandes et al. still hold in a controlled environment that provides an uniform random.

A last consideration can be made about the partitioner strategy, which appears to be, on average, the least performer. In this case the introduction of a gossip layer to compute the distributed partitioning of a graph appears to be not justified, according to the poor performances of the strategy.

	TOP 25				TOP 50				TOP 100			
	MIN	MAX	AVG	SDEV	MIN	MAX	AVG	SDEV	MIN	MAX	AVG	SDEV
RANDOM	2	11	8.49	2.25	4	12	10.28	1.79	8	18	13.46	2.3
ADAPTIVE	3	14	**9.45**	2.74	4	17	**11.63**	2.51	11	21	**15.54**	2.68
PARTITIONER	3	11	7.48	1.37	5	14	9.24	1.79	7	19	12.28	3

TABLE 3.1: Comparison of the flow creation strategies for the Random graph (in terms of CCs created)

Algorithm 3.2: The GAS Duckweed Algorithm

```
def Gather (M1,M2 ← receive messages from neighbours)
    Map ← old collected message;
    for $\forall F^{(s,t)} \in M1 \cup M2$ do
        $F_{old}$ ← Map.get ($F^{(s,t)}$);
        if $F^{(s,t)}$ isConverged then
            $F_{old}$.PutList ($F^{(s,t)}$)
        else
            $F_{old}$.PutSum($F^{(s,t)}$)
        end
    end
def Apply (F ← aggregated message)
    forall the $f \in F$ do
        $p_f$ ← Update () ;
        $\Delta f$ ← difference between $p_f$ and $p_f$ of the previous iteration ;
        if Terminate( $\Delta f$) then
            mark f as terminate
        end
    end
    UpdateCentrality (f) ;
def Scatter ()
    F ← new message with all flows $p_f$ updated;
    Send($F$) to neigthbours ;
```

3.5 Duckweed on Apache Spark: the algorithm

In this section we describe how the computation of the current flow betweenness centrality has been defined by exploiting the *GAS (Gather-Apply-Scatter)* model, an iterative graph-parallel paradigm [76] in order to be implemented on real distributed frameworks like Spark and GraphX. In the following, we present the algorithm, GAS-Duckweed and the optimizations which exploit the characteristics of the model.

Take as reference Algorithm 3.2, that shows the high level structure of GAS-Duckweed. The algorithm is splitted into three data-parallel phases:

- *Gather* combines incoming messages, each message is the potential value of a neighbour, relative to a specific flow identified by the *(s,t)* pair. It optimizes communications by aggregating the potentials relative to the same flow.
- *Apply* consumes the aggregated message, in particular it updates all potentials of the flows, and eventually updates the CFBTW.
- *Scatter* defines the messages that are dispatched to the neighbours.

3.5.1 Reducing the number of messages

The algorithm is characterized by a neighbour-to-neighbour communication pattern. In each iteration all vertices receive and send messages through their edges. The value of a message exchanged between neighbours is a map consisting of key-value pairs, where the keys uniquely identify the flows by s and t, respectively the source and the target of the flow, and the value is the current potential for that flow. Thanks to pull-based model of message computation we can reduce the number of the sent messages by applying the gather function on messages sent to the same vertex.

3.5.2 Reducing the size of the messages

All the messages include information of a set of flows. If all the flows are sent in all the iterations many duplicate potentials may be sent in consecutive iterations. We can reduce the size of the messages by *avoid re-sending* or *combine message* tecniques.

Avoid re-sending. Each vertex has a local view of the flows that have been injected and that are flowing through the graph. Each flow is characterized, by its state, which can be *Active*, *Converged*, or *Terminated.* The *Active* state identifies flows currently processed by the vertex. A flow is considered *Converged* when the difference of the values of its potential, in two consecutive iterations, is less than a system-wide parameter D_ϵ. A flow is considered *Terminated* when it has reached convergence in i and in all its neighbours. When a flow $F^{(s,t)}$ is *Converged* on i but still *Active* on some neighbours, such neighbours still requires $F_{P_i}^{(s,t)}$ to continue the computation in the next iterations. If this is the case, i sends one last time its converged flow $F_{P_i}^{(s,t)}$ with a flag $\overline{F}_{P_i}^{(s,t)}$. Thanks to this, all the neighbours knows that this is the last time that they will receive the potential for such flow from i.

Combine message. In each iteration, each vertex receives the messages generated in the previous iteration. In GAS-Duckweed during the gather phase we sum the values of each flow separately to reduce the size of all messages even more. Note, not all the values of the flow can be aggregated. The converged flows have to be saved in local memory and thus they must not be considered in the gather phase. Algorithm 3.2 shows the pseudo code of the Gather function. This is applied to all messages and reduces both the number of messages and the size of each message.

3.5.3 Flow Generation

We assume that the vertices are labelled from 0 to $n-1$, where n is the number of vertices in the graph. In order to create the flow $F^{(s,t)}$, the source and the target are selected by generating a random value $\in [0, n-1]$.We define a system-wide parameter φ equals to the maximum number of flows in status *Active* on each node. In each iteration t, DUCKWEED monitors the average number of flow in status *Active* on each node equals to A_t. In each iteration, k flows are generated by DUCKWEED. k is equal to $\varphi - A_t$.

3.5.4 Termination

In each iteration DUCKWEED removes completed flows from the computation and generates new ones. Due to this, in each iteration, DUCKWEED can provide an approximation of the CFBTW. Clearly, when more flows are completed the approximation will be closer to the exact value. We define the parameter θ equals to the percentage of all possible flows that need to be calculated before stopping. In each iteration t, DUCKWEED sets the value to a variable F_t equals to the average number of completed flows on each node.

Given F_{all} equal to the amount of all the possible flows that can be generated for the graph G, DUCKWEED stops the computation when $F_t = \theta F_{all}$. The algorithm exploits also a parameter *iter* to avoid too long executions. In any case, DUCKWEED stops the execution when *iter* iterations have been completed.

3.5.5 Implementation Details

Algorithm 3.3 shows how the GAS functions described in the previous sections are used to provide two different implementations DUCKWEED_X and DUCKWEED_MP. The DUCKWEED_X implementation uses the graph model abstraction provided by GraphX. We exploit the *Pregel* method, provided by GraphX, that requires the aforementioned three functions to compute a graph parallel computation on a graph. Also, we have implemented a different version based on Apache Spark using operator like Map, Reduce, Join without the graph abstraction support. The graph can be logically represented as a pair of vertex and edge property collections. The main difference between the two implementation is the data partitioning. GraphX enables vertex partitioning, whereas DUCKWEED_MP exploits an edge partitioning.

Algorithm 3.3: Different Implementation DUCKWEED_X, DUCKWEED_MP

```
def DUCKWEED_X (Gather, Apply, Scatter)
    G ← Graph(V,E);
    pregel_like(G, Gahter, Apply, Scatter);
def DUCKWEED_MP Gather, Apply, Scatter)
    (Link, Vertices) ← Graph(V,E);
    MSGs ← Vertices.join.(Link).flatMap(Scatter);
    MSGs.reduceByKy(Gather);
    G.vertices.join(MSGs).map(Apply);
```

3.6 Experimental evaluation on Apache Spark

In this section, we evaluate the performance of the DUCKWEED_X and DUCKWEED_MP algorithms which are the implementations of the Algorithm 3.2 which make use, respectively, of GraphX and Spark. Evaluations were performed on a cluster of 4 nodes, each having 4 cores and 16 GB of memory. We compare our approach with two competitors, Betweenness Centrality[70] [2] and K-betweenness[91], both implemented by their proposers in Spark: [3].

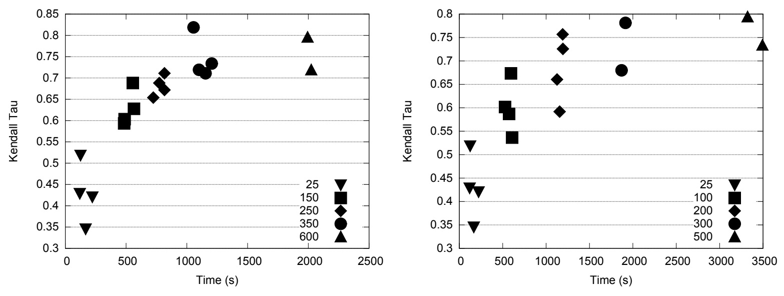
Evaluation of the Approximation. We measure the correlation between the results obtained by our approach and those obtained by NetworkX [160] which provides an implementation of the algorithm proposed by Brandes et. al.[27] to compute the CFBTW and returns an exact result. We generate a preferential attachment graph by means of the Snap library [4]. The size of the graph is restricted to 10 000 nodes, because in NetworkX it is not possible to obtain the exact value of CFBTW for larger graphs.

Figure 3.8a shows the result returned when using a different number of concurrent flows with DUCKWEED_MP (we obtain analogous result with DUCKWEED_X). The Y axis reports the Kendall Tau correlation of the top

[2] `http://neo4j.com/developer/apache-spark/`

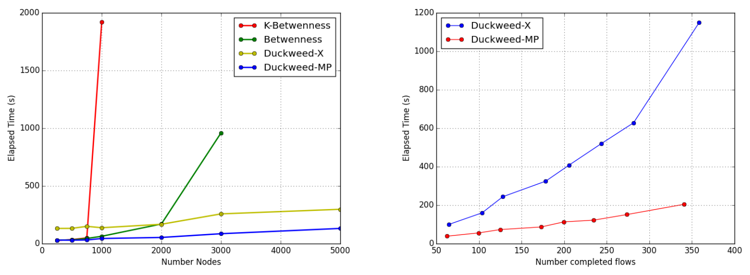
[3] `https://github.com/kbastani/neo4j-mazerunner`

[4] `http://snap.stanford.edu/`



(A) Concurrent Flows. (B) Iterations.

FIGURE 3.8: Evaluation of the approximation. Kendall tau correlation with NetworkX.



(A) Preferential Attachment graphs. (B) Comparison between DUCKWEED_X and DUCKWEED_MP.

FIGURE 3.9: Comparison with competitors and DUCKWEED_X, DUCKWEED_MP.

50 nodes, while the X axis the computational time. We fix the iteration number to 25. We run the algorithm 4 times for each configuration. We can observe that the correlation increases very quickly until it reaches the 0,9%. We obtain the best result with $\varphi = 350$. Greater values requires more time to complete and do not bring benefits in terms of quality. Figure 3.8b shows the results when varying the number of iterations. We fix the number of concurrent flows to 25 and we increase the number of iterations. Similarly as before, a high number of iterations does not bring to a large increase of the correlation value.

Performance Evaluation. We compare the computational time of our algorithm against the two competitors. Figure 3.9a shows the execution time of the different algorithms when the size of the graph varies in the set $\{250, 500, 750, 1000,$
$2000, 3000, 5000\}$ for a preferential attachment graph. We omit results for random graphs that show similar outcome. DUCKWEED_MP yields better completion times in all cases. In addition, the two competitors are unable to provide a result for the larger graphs due to memory errors. DUCKWEED_MP is able to provide a result 10 times faster with respect to the best competitor. Next, we perform an evaluation using real graphs, results are presented in Table 3.2. We make use of publicly available graphs by Konect[5]. We are unable to run the two competitors on the larger dataset due to memory errors. Figure 3.9b compares the execution time of DUCKWEED_X and

[5]`http://konect.uni-koblenz.de/networks/`

Table 3.2: Execution time on real world graphs.

Algorithm	Netscience(379)	Email(1133)	Brightike(56739)	WordNet(145145)
Betweenness	62	160	-	-
K-Betweenness	30	49	-	-
DUCKWEED_X	410	110	-	-
DUCKWEED_MP	33	51	4232	4636

DUCKWEED_MP with the same configuration parameters. DUCKWEED_MP is able to calculate the same number of completed flows in less time with respect to DUCKWEED_X. For instance, with DUCKWEED_MP we calculate 200 flows in 150 seconds whereas DUCKWEED_X requires around 400 seconds.

3.7 Conclusion

In this chapter we presented DUCKWEED, a distributed approach to calculate the current flow betweenness centrality. To the best of our knowledge, it is the first solution able to compute a centrality index, based on global properties of the graph, in a fully distribute way.

Each node relies only on local information, i.e. on information stored by itself or by its neighbours, to adaptively estimate the value of its current flow betweenness. We empirically prove that DUCKWEED is able to provide a good approximation of the CFBTW and correctly identifies central nodes which guarantee network connectivity. DUCKWEED delivers good results also for large graphs composed by millions of nodes, whereas existing approaches considered only thousands of nodes. DUCKWEED provides good scalability figures achieved through its ability to approximate the result computing only a subset of all the possible flows of the network.

As a future work, we plan to conduct further studies to analyse the behaviour of our approach for dynamic graphs. Furthermore, we plan to investigate whether Kirchhoff's circuit simplification rules can be exploited to detect block of nodes which can be reduced to a single node, with the aim of optimizing the overall computation. The aim of introducing such simplification technique is to continue to provide good result increasing the level of approximation and simplification.

Chapter 4

Connected Components: Cracker

In this chapter we focus on the problem of finding *connected components* (*CC*) in large graphs by leveraging the TLAV approach. This problem is of fundamental importance in graph theory and can be applied to a wide range of different research fields. For example, finding *CC* is the building block in many research topics, such as to generate group of features in image clustering [7], study the analysis of structure and evolution of on-line social networks [100], derive community structure in social networks [68], group together similar spam messages to detect spam campaigns (Chapter 7), estimate the population from mobile calls (Chapter 8).

In this chapter we propose CRACKER, a highly-efficient distributed iterative algorithm for the identification of connected components in large graphs. In the context of this thesis, this work is of paramount importance as it has been exploited as building block to perform data clustering. CRACKER works by iteratively growing a tree for each connected component belonging to the graph. The nodes added to the trees are no longer involved in the computation in the subsequent iterations. This corresponds to the *simplification* guideline shown in the introduction. By means of trimming the number of nodes involved during each iteration, CRACKER significantly reduces the total computation time as well as the volume of information transferred via the network.

In order to perform a fair evaluation of CRACKER, the most relevant state-of-the-art algorithms have been implemented on the same framework (i.e. Apache Spark). The evaluation has been conducted exploiting several synthetic and real-word datasets. The results we achieved show that CRACKER out-performs competitor solutions in a wide range of setups.

The key points characterizing our proposal are the following:

- our algorithm exploits a novel node pruning strategy that allows to dramatically reduce its computational cost;
- we present an extensive experimental evaluation, conducted both on synthetic and real-world datasets, where CRACKER is compared against state-of-the-art algorithms; the analysis includes both computational and communication costs and also scalability with respect to graphs size and complexity; experimental evidence shows that CRACKER significantly improves over the state-of-the-art;
- we provide a complete theoretical analysis of CRACKER for undirected graph in terms of (i) correctness, (ii) computational cost and (iii) number of messages.

Table 4.1: State-of-the-art algorithms. d is diameter, n is the number of nodes, m is the number of edges

	communication pattern	detection strategy	vertex pruning	number of iterations	number of messages per iteration
PEGASUS [54]	static	labelling	no	$O(d)^3$	$O(m+n)^3$
HASH-TO-MIN [151]	dynamic	clustering	no	$O(\log(d))^4$	$2(m+n)$
CCMR [161]	dynamic	labelling	no	N/A	N/A
ZONES [47]	dynamic	labelling	no	$O(d)^3$	$O(m+n)^3$
CCF [93]	dynamic	labelling	limited	N/A	N/A
ALT-OPT [99]	dynamic	labelling	no	$O(\log n)^5$	$O(m)$
SGC [147]	dynamic	labelling	limited	$O(\log n)^6$	$O(m+n)$
CRACKER	dynamic	labelling	yes	$O(\log n)$	$O(\frac{nm}{\log n})$

- we extend the base algorithm with three optimisations. We provide experimental evidence that these optimisations greatly improve its performance;
- we give a detailed description of the implementation of the CRACKER algorithm on the Apache Spark framework.

In order to make our results reproducible we made publicly available the source code of CRACKER[1] (as well as the code of all the competitors used in the comparison) and the graph datasets used in the experimental evaluation[2].

The CRACKER algorithm has been initially published in the paper *"Cracker: Crumbling large graphs into connected components"* [122] presented at IEEE Symposium on Computers and Communication. An extension of the algorithm with optimizations and a theoretical study on the costs of the algorithm has been published in the journal IEEE Transactions on Parallel and Distributed systems with the title *"Fast Connected Components Computation in Large Graphs by Vertex Pruning"* [123].

4.1 Related Work

Finding connected components is a well-known and deeply studied problem in graph analytics. So far, many different solutions have been proposed. When the graph can be kept in the main memory of a single machine, a visit of the graph can find connected components in linear time [85]. Many distributed approaches have been proposed to tackle the very same problem in large graphs. Earlier solutions considered the PRAM model [94, 92]. However, often the implementation of these solutions is complex, error-prone and not efficiently matching the programming models provided by the current distributed frameworks [151].

Many proposals dedicated to the problem of finding connected components have been thought for today's distributed frameworks, in particular for MapReduce platforms. In this section we analyse and compare the proposals that are most related with CRACKER.

To structure our comparison, we frame a selection of existing solutions belonging to the conceptual framework of vertex-centric approaches. According to such model, each vertex of the graph is seen as a computational unit able to communicate with its graph neighbours. The computation is defined

[1]`https://github.com/hpclab/cracker`

[2]`http://www.di.unipi.it/ lulli/project/cracker.htm`

for a generic vertex, and it is repeated by all the vertices in the graph. In *CC* discovery algorithms, a vertex usually propagates and maintains information about the connected component it belongs to and the computation is iterated until convergence. Yan et al. [190] introduce the notion of balanced practical Pregel algorithm to characterize some nice-to-have properties for *CC* algorithms making use of this model of computation. For instance, Feng et al. [66] presents a *CC* algorithm targeted for Pregel framework with performance similar to HASH-TO-MIN.

In Table 4.1 we provide a characterization of some of the most relevant state-of-the-art approaches, presenting them on the basis of their qualitative behaviour, i.e. *detection strategy*, *communication pattern* and *vertex pruning*, and of the theoretical bounds for the number of iterations and of messages per iteration. Regarding the detection strategy, we distinguish between *labelling* and *clustering* approaches. The former associates, to each vertex, the id of the *CC* it belongs to, which is usually given by the smallest id of the vertices belonging to the *CC*. The latter assumes that one vertex for each *CC* knows the identifiers of all the other vertices of the same *CC*. As a consequence, *labelling* requires to process less amount of information; however the *CC*s can be efficiently reconstructed by a post-processing step.

Different communication patterns specify how vertices exchange information one to each others. A *static* pattern happens when each vertex considers the same set of edges at every iteration, usually its neighbors in the input graph. This pattern is straightforward to implement, but is characterized by slow convergence. To address this issue, other approaches employ a *dynamic* pattern, in which the set of edges evolves over time. This approach is usually more efficient as it can add new connections and remove stale ones, with the aim of reducing the diameter of the *CC* and, in turn, speeding up convergence.

The last feature we consider is *vertex pruning*, namely the ability of excluding vertices from computation. State-of-the-art algorithms keep iterating the same vertex-centric computation on all vertices of the graph until convergence. In this way, a large number of vertices remains involved in the computation even if they do not provide useful information toward convergence. For instance, a small *CC* could be excluded from the computation when it reaches convergence, without affecting the discovery of other connected components.

In 2009, Cohen [47] proposed an iterative MapReduce solution (which we refer to as ZONES) that groups connected vertices around the vertex with the smallest identifier. Initially, the algorithm constructs one zone for each vertex. During each iteration, each edge is tested to understand if it connects vertices from different zones. If this is the case, the lower order zone absorbs the higher order one. When there are no zones to be merged, each zone is known to be a connected component. The main drawback of this approach is that all edges are checked during every iteration (no vertex pruning), resulting in long convergence time.

Seidl et al. [161] proposed an improved version of ZONES called CCMR. The idea surrounding CCMR is to add *shortcut* edges, such that fewer iterations are needed to spread information across the graph. The CCMR algorithm modifies the input graph during each iteration, until each connected component is transformed in a star-shaped sub-graph where all vertices are connected with the one having the smallest identifier. Thanks to these improvements, CCMR yields lower running times with respects to ZONES.

Deelman et al. proposed an algorithm for the detection of connected components within the graph mining system PEGASUS [54]. They employ a static communication pattern. During each iteration, each node sends the smallest node identifier it knows to all its neighbours. In turn, each node updates its knowledge with the received identifiers. The algorithm labels all nodes with the seed identifier in $O(d)$ MapReduce steps, with d the diameter of the largest connected component. Similarly, Rastogi et al. [151] proposed HASH-TO-MIN, a vertex-centric algorithm parametrized by an hashing and a merging function determining the information travelling across the graph. The HASH-TO-MIN algorithm iterates as PEGASUS by propagating the smallest node identifier seen so far, but in addition it also communicates the whole set of known nodes so as to create new connections among nodes being at more than one hop distance.

Kardes et al. [93] proposed a MapReduce algorithm in two phases named CCF. The first phase is similar to the HASH-TO-MIN approach but they introduce some improvements that reduce the computation cost in spite of more MapReduce steps. The second phase of CCF is an optimization that reduces the amount of duplicated messages. CCF employs vertex pruning limited to the seed nodes, whereas CRACKER processes only the relevant vertices, while discarding vertices that have no useful information to share.

Recently, Kiveris et al. [99] proposed ALT-OPT. The algorithm selectively removes edges from the graph, until each connected component is identified by a star-shaped graph centred on the seed. To avoid unbalanced computations, it splits vertices with high degree in multiple copies, in fact speeding up the computation at the expense of more MapReduce steps. ALT-OPT shares many traits with CRACKER, being based on a dynamic communication pattern and using labelling as the detection strategy. However, CRACKER excludes nodes over time, which reduces the overall computational cost and allows to collapse the computation in a single machine when the number of active nodes is sufficiently small.

Another recent solution for *CC* discovery, which we refer to as SGC, has been presented by Qin et al. [147]. It exploits a set of join operators defined by the authors for the Hadoop framework to model an iterative MapReduce computation. Similarly to CRACKER, their algorithm outputs a forest of trees, each representing a connected component. Initially each node becomes part of a tree-like graph by setting as a parent the node in its neighbourhood with the lower identifier, thus creating a forest of (possibly interconnected) trees. Then, one-node (i.e. singleton) and non-isolated trees (i.e. connected by an edge to another tree) are iteratively merged until all trees become isolated (the *hooking* phase). Subsequently, each tree is transformed into a star-shaped graph with the root in the center, so that nodes get to know the *CC* they belong (the *pointer jumping* phase). This last phase has the same goal of CRACKER's *seed propagation* (see Section 4.2.2) but it requires to access to the 2-hop neighbourhood of nodes. SGC performs only a limited amount of vertex pruning by deactivating at each step the nodes that do not match some criteria. Further, it requires a large number of MapReduce step in each iteration to verify some properties on each node and to identify if a

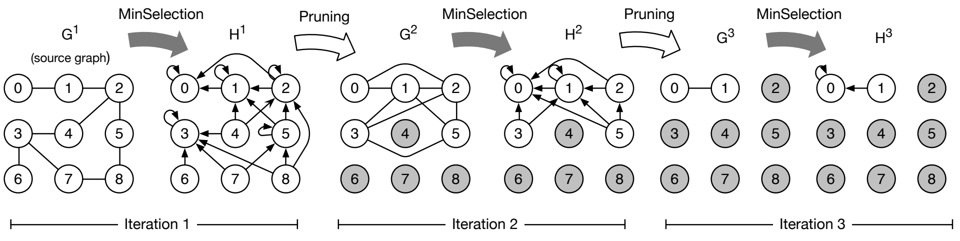


FIGURE 4.1: CRACKER: example of seed identification. Gray vertices are excluded from the computation

tree can be merged.

4.2 The CRACKER Algorithm

Let $G = (V, E)$ be an undirected graph where V is a set of n vertices uniquely identified by values in $\mathbb{Z}$, and $E \subseteq V \times V$ is the corresponding set of m edges. A connected component (CC) in G is a maximal subgraph $S = (V^S, E^S)$ such that for any two vertices $u, v \in V^S$ there is an undirected path in S connecting them. We conform to the convention to identify each CC of the graph with the smallest vertex identifier belonging to that component. The vertex having this identifier is the *seed* of the connected component.

The CRACKER algorithm (see Algorithm 4.1) achieves the identification of the CCs into two phases:

- *Seeds Identification*: for each CC, CRACKER identifies the *seed* vertices of the graph, and it iteratively builds a *seed propagation tree* rooted

[3]Not available in the original paper and taken from [151].

[4]Rastogi et al. [151] conjecture that HASH-TO-MIN finishes in $2(\log d)$ iterations on all inputs. They prove also that HASH-TO-MIN terminates in $4(\log n)$ iterations on any path graph.

[5]Kiveris et al. show a complexity of $O(\log^2 n)$ for the algorithm called Two-Phase. Here we refer to an optimization of it called ALT-OPT with only a claimed complexity without any theoretical proof.

[6]In Qin et al. [147] the proof is omitted due to lack of space.

[7]In Feng et al. [66] is presented the total communication cost, however in the first iteration if a node is connected to all the other nodes the BFS cost $O(m)$ message.

Algorithm 4.1: The CRACKER algorithm

```
Input  : an undirected graph $G = (V, E)$
Output: a graph where every vertex is labeled with the seed of its $CC$
1  $u$.Active = $True$  $\forall u \in G$
2  $T \leftarrow (V, \emptyset)$
3  $t \leftarrow 1$
4  $G^t \leftarrow G$
5  repeat
6      $H^t \leftarrow$ Min_Selection($u$) $\forall u \in G^t$
7      $G^{t+1} \leftarrow$ Pruning($u, T$) $\forall u \in H^t$
8      $t \leftarrow t + 1$
9  until $G^t = \emptyset$
10 $G^* \leftarrow$ Seed_Propagation($T$)
11 return $G^*$
```

in the *seed*; whenever a vertex is added to the tree, it is excluded from computation in the subsequent iterations (see Alg. 4.1 lines 6–6). When all the vertices are excluded from the computation the *Seed Identification* terminates and the *Seed Propagation* begins.

- *Seeds Propagation*: propagates the *seed* to all the vertices belonging to the *CC* by exploiting the seed propagation tree built in the previous phase. (see Alg. 4.1 line 42).

In the following we describe in detail the two phases. The presentation is given adopting a vertex-centric computing metaphor: at each iteration the vertices of the input graph are processed independently and in parallel.

4.2.1 Seed Identification

The basic idea of the *Seed Identification* phase is to iteratively reduce the graph size by progressively pruning vertices until only one vertex for each connected component is left, i.e., its *seed*. When a vertex discovers its own *CC*, it is excluded from computation since does not impact on the other *CC*s in the graph. In short, a vertex v discovers its *CC* by interacting only with its (evolving) neighbourhood. At any iteration, a vertex may discover in its neighbourhood another vertex q with a lower identifier value. If this happens, v connects to q. If v is not chosen by any neighbour, v is excluded from the computation and it becomes the child of q in the seed propagation tree that, at the end of the algorithm, will include all the vertices of the connected component.

As shown in Algorithm 4.1, each vertex $u \in G$ is initially marked as *active*, meaning that at the beginning all vertices participate to the computation. The seed identification is, in turn, an iterative algorithm made of two steps: *MinSelection* and *Pruning*. The *Seed Identification* phase is exemplified in Figures 4.1 (Graph) and 4.2 (Tree) and detailed below.

MinSelection

This step serves to identify those vertices that are guaranteed to *not* be seed of any connected component (see Algorithm 4.2). From the point of view of the entire graph, it takes in input a undirected graph G^t at iteration t, and builds a new *directed* graph H^t. The edges of H^t are created as the following. For each vertex $u \in G^t$, the v_{min} is selected as the vertex with the minimum *id* from the set $NN_{G^t}(u) \cup \{u\}$ (see line 9 in Algorithm 4.2), where $NN_{G^t}(u)$ is the set of neighbors of u in G^t. The v_{min} is then notified to all the neighbours of u and to u itself. This communication is materialised as the addition of new directed edges $v \to v_{min}$ for every $v \in \{NN_{G^t}(u) \cup u\}$ (see

Algorithm 4.2: `Min_Selection` (u)

```
Input : a vertex $u \in G$
1 $NN_{G^t}(u) = \{v : (u \leftrightarrow v) \in G^t\}$
2 $v_{min} = \min(NN_{G^t}(u) \cup \{u\})$
3 forall the $v \in NN_{G^t}(u) \cup \{u\}$ do
4     AddEdge $((v \to v_{min}), H^t)$
5 end
```

line 4 in Algorithm 4.2). After all vertices in G^t completed the MinSelection, for each vertex $u \in H^t$ it holds the following: (i) if u is not a v_{min} for any $NN_{G^t}(u)$, it has no incoming links; (ii) u has an outgoing link to its v_{min} and with the v_{min} of every node in $NN_{G^t}(u)$. According to the algorithm, a vertex is considered a *potential seed* if it is a local minimum in the neighbourhood of some vertex, and in such case, it has at least one incoming edge in H^t. Therefore, after the MinSelection, the nodes that have no incoming edges are guaranteed to not be seed of any connected component.

For instance, let us consider vertex 8 in the graph H^1 in Fig. 4.1 produced by the first iteration of the MinSelection. Vertex 8 has three outgoing edges: (i) $8 \to 5$, which has been created by 8 itself as 5 was its v_{min} in the input graph G^1; (ii) $8 \to 2$, created by 5 connecting its v_{min} with 8; (iii) $8 \to 3$, created by 7 connecting its v_{min} with 8. Therefore, the knowledge of node 8 about G is improved only by information exchanged with its neighbours. In the same way all the other nodes improve their knowledge about the graph.

Pruning

The *Pruning* step (see Algorithm 4.3) removes from H^t, and thereby excludes, all the vertices that cannot become *seeds*. The vertices excluded during the *Pruning* grow a forest of seed propagation trees T each covering a distinct CC of the graph. From the point of view of the entire graph, it takes in input a directed graph H^t and generates a new *directed* graph G^t.

In the Pruning, each node recomputes v_{min} considering $NN_{H^t}(u)$, which is composed by all the outgoing edges. Then, for every node v in $NN_{H^t}(u)$ (except v_{min}), a new undirected edge v with v_{min} is added to the graph G^{t+1} (see line 5). Note that $NN_{H^t}(u)$ is in general different from $NN_{G^t}(u)$, with the former normally having lower identifiers. For example, in Figure 4.1 $NN_{G^t}(5) = \{2, 8\}$ and $NN_{H^t}(5) = \{1, 2\}$. These undirected edges make sure that the nodes in G^{t+1} are not disconnected in case u is deactivated and therefore not included in the graph G^{t+1}. At the end of the Pruning, the nodes identified as non seed in the MinSelection have no edges and therefore are excluded by the computation. According to the algorithm of

Algorithm 4.3: `Pruning`(u, T)

```
Input : a node $u \in G$ and the seed propagation tree $T$
$NN_{H^t}(u) = \{v : (u \to v) \in H^t\}$
$v_{min} = \min(NN_{H^t}(u))$
if $|NN_{H^t}(u)| > 1$ then
    forall the $v \in NN_{H^t}(u) \setminus v_{min}$ do
        AddEdge $((v \leftrightarrow v_{min}), G^{t+1})$
    end
end
if $u \notin NN_{H^t}(u)$ then
    $u$.Active = $False$
    AddEdge $((v_{min} \to u), T))$
end
if IsSeed $(u)$ then
    $u$.Active = $False$
end
```

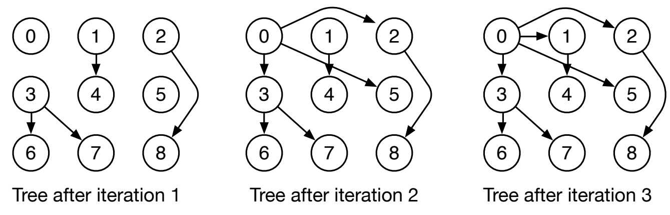


FIGURE 4.2: CRACKER seed propagation tree

the MinSelection, this can be verified by checking whether a node has a self-link in H^t: if it does not it cannot be the minimum of the local neighbourhood (which includes itself) and can be safely excluded (see line 9). The nodes marked for exclusion are inserted in the seed propagation tree T (see line 34). Finally, a node is *finalized* as a seed when it is the only active node in its neighbourhood $NN_{G^{t+1}}(u)$. It is marked for exclusion and added to T as the root of a CC.

Now, let us consider again the example in Fig. 4.1. Nodes 4, 6, 7 and 8 are excluded from G^2 because they have not been chosen as v_{min} of any node at the previous iteration. Graphically, excluded vertices can be easily spotted as they do not have any ingoing edge. Being excluded, these nodes are connected to their v_{min} in the seed propagation tree T as shown in Fig. 4.2. Specifically, in the propagation tree, the vertex 3 has 6 and 7 as children (being their v_{min} in H^1). Similarly, vertex 2 has vertex 8 as child, and vertex 1 has vertex 4. Note, G^2 preserves the connectivity of the remaining vertices, and this holds in general for every G^t.

4.2.2 Seed propagation

Please recall that a seed propagation tree for each component of the graph is incrementally built during the Pruning. When a vertex v is excluded from the computation, a directed edge $v_{min} \rightarrow v$ is added to the tree structure T (see Line 10 in Algorithm 4.3).

The Seed Propagation phase starts when there are no more active nodes after the execution of the Pruning. At this there exists in T for each CC a seed propagation tree rooted in its *seed* node. Figure 4.2 shows the tree at each iteration resulting from the example presented in Figure 4.1. Such tree is then used to propagate the seed identifier to all the nodes in the tree. In details, the propagation starts from the root of each tree. The roots send their identifier to their children in one MapReduce iteration, this identifier will be the identifier of the CC. In every iteration, each node that receives the identifier propagates it to its children. The execution stops when the identifiers reach the leaves of the tree.

4.2.3 Cracker correctness

We denote with $NN^d_{G^t}(u)$ the set of vertices at distance at most d from u in G^t. In the following, we first highlight a few properties which can be derived from the CRACKER algorithm.

Property 1 (Active Vertices). *An active vertex $u \in G^t$ will stay active in G^{t+1}* iff *it is a* local minimum *for any of its neighbors or for itself.*

Property 2 (New edges)**.** *Given a node $u \in G^t$, let u^1_{min} and u^2_{min} be the smallest nodes in $NN^1_{G^t}(u)$ and $NN^2_{G^t}(u)$, respectively. The graph G^{t+1} will have an edge $u^1_{min} \leftrightarrow u^2_{min}$, and, if u is still active, an edge $u \leftrightarrow u^2_{min}$.*

Property 3 (Edges of Neighbors)**.** *Given two neighboring nodes $u, v \in G^t$, an edge $v^1_{min} \leftrightarrow u^2_{min}$ is created in G^{t+1}.*

The first property holds because a local minimum node is never deactivated. The second property holds because the node u creates links between its neighbors in H^t (possibly including itself) to the new minimum in $NN^2_{G^t}(u)$. The third property holds because u's neighbors in H^t include the local minimum of v.

We can now prove the following.

Theorem 1 (Path Preservation after Pruning)**.** *If a vertex $u \in G^t$ becomes inactive, other vertices in the same connected component will still be connected in G^{t+1}, if active.*

Proof. We equivalently prove that if a node $u \in G^t$ is removed, its neighbors that remain active are still connected in G^{t+1}. According to Property 2, every such neighbor v of u becomes connected to v^2_{min} and indirectly to v^1_{min}. Moreover, according to Property 3, v^1_{min} has an edge to u^2_{min}. Therefore, if active, every neighbor of u is connected to u^2_{min} through a path in G^{t+1}. □

Note that in case of multiple node removals, their local minima are never removed by CRACKER, which guarantees that at least one neighbor for each removed node is kept active at the next iteration. Indeed, the nodes u^1_{min} and u^2_{min}, for every $u \in G^t$, form the new *connectivity backbone* of graph G^{t+1}.

Theorem 2 (Seed Propagation Tree)**.** *Given a connected component, the seed propagation tree T built by* CRACKER *is a spanning tree of the connected component.*

Proof. New edges from inactive to active vertices are generated in the propagation tree T after each iteration. This process has three important properties. First, according to Theorem 1, the remaining active vertices do not alter the connectivity of the original CC. Second, at least one vertex is deactivated and added to T after each iteration, i.e., the vertex with the largest id, until the seed vertex is left. Third, newly added edges of T always link an inactive vertex to an active one, thus avoiding loops.

The first condition implies that only one tree T is generated as the connected component is never partitioned. The second condition implies that T is actually a tree, while the third condition implies that every vertex in the CC is eventually added to T which is rooted at the seed vertex. □

Theorem 3 (Correctness)**.** *The* CRACKER *algorithm correctly detects all the connected components in the given input graph.*

Proof. According to Theorem 2, a propagation tree is built for each CC in the input graph. Clearly CRACKER does not add edges in the propagation trees between two vertices not being in the same connected component, as they cannot be neighbours at any iteration. Therefore, the propagation trees built by CRACKER uniquely identify the CCs in the input graph. □

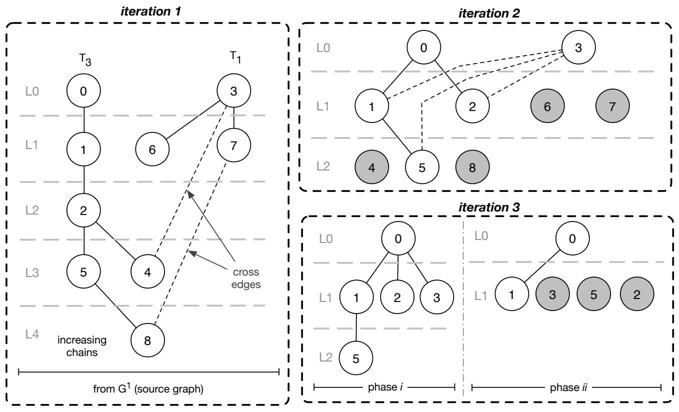


FIGURE 4.3: Illustration of algorithm CRK on the same graph in Figure 4.1. Grey nodes are deactivated, and generated new cross-edges. Only the first three iterations are reported.

4.2.4 Cracker computational cost

In this section we discuss the computational complexity of CRACKER both in terms of number of iterations and number of messages. We use Figure 4.3 to exemplify the notations and properties exploited, with reference to the same graph used in Figure 4.1.

Any given connected graph G can be organized into levels $L_0, \ldots, L_i, \ldots, L_{l-1}$, such that level L_0 contains nodes u having $u^1_{min} = u$, while level L_i contains nodes v such as their local minimum v^1_{min} is in level L_{i-1}. If we consider only edges of the kind $v \leftrightarrow v^1_{min}$, each node in L_0 is the root of a tree, where root-to-leaf paths traverse nodes with increasing ids. Figure 4.3 shows the two *increasing trees* present in the exemplifying graph.

Property 4 (Increasing tree cost). *The* CRACKER *algorithm takes $O(\log l)$ iterations to process an* increasing tree *of height l.*

It can be trivially seen that, according to Prop. 2, after iteration $t = 1$ each node in level L_i is linked to a node L_{i-2}, i.e., its smallest neighbor at 2 hops distance, to a node in L_{i-4} after iteration $t = 2$, and to a node in L_{i-2^t} after iteration t, so that after $\log(l)$ iterations every node becomes aware of the root in L_0, being the node with the smallest identifier in the tree. Moreover, when a node becomes a leaf of the tree it is deactivated according to Property 1. Therefore, after $\log(l)$ iterations, all the nodes but u_0 are deactivated and the algorithm completes.

The above property can be easily generalized if we also consider the edges of the graph not covered by the *increasing trees*. Indeed, such edges potentially links two nodes at different levels L_i and L_j, and they will generate edges between levels $L_{i/2}$ and $L_{j/2}$ (or lower) thus speeding up the convergence to the root node.

In general, any given connected graph can be organized in a set of interlinked *increasing trees*, with additional edges, named *cross-edges*, across those trees. Figure 4.3 illustrates the two *cross-edges*: $(4, 3)$ and $(8, 7)$. Based on the notion of *increasing trees*, we show that at every step of the CRACKER algorithm, these trees are *reduced* in height and *merged* until only the seed node is left.

Theorem 4 (Number of Seeds identification iterations)**.** *Given a connected graph G having n nodes, the number of iterations taken by the seed identification phase is $O(\log n)$.*

Proof. Let T_h be the set of *increasing trees* in G having depth $\leq 2^h$ and not included in T_{h-1}. We show that CRACKER reduces the height of such trees after each iteration and merges them until only the seed node is left. Specifically, we base this proof on a simplified variant of CRACKER, named CRK, which alternates two phases: *(i)* one CRACKER iteration processes only the *increasing trees* in T_0; *(ii)* one CRACKER iteration processes each *increasing tree* in the graph.

CRK is thus similar to CRACKER with some limitations. During *phase i*, only nodes in T_0 trees may find their best local minumum through the cross edges, and during *phase ii*, *cross-edges* are not exploited to find the new best local minimum, but only to guarantee connectivity when nodes are deactivated. Such limitations make CRK computationally more expensive than CRACKER. However, we prove, that CRK satisfies the Theorem 4.

Phase i. Let's consider the *increasing trees* in T_0, i.e., composed of a single node u. By construction, u has a neighbor v reachable through *cross-edges*, with $v^1_{min} = z$, $z < u$ and $z < v$. After one iteration node u is linked to z thanks to Prop. 2, possibly increasing by one the height of the *increasing tree* containing z. In Fig. 4.3, in iteration 2, the node 3 has node 1 as neighbor which has local minimum node 0, and therefore node 3 is linked to node 0. All the *increasing tree* in T_0 are merged with other trees analogously. Note that if T_0 is empty then *phase i* does not take place.

Phase ii. We consider the *increasing trees* of G in isolation, i.e., without exploiting *cross-edges* to find a new best local minimum. In this setting, the height of each tree is halved at each iteration according to Prop. 4. In addition, leaf nodes are deactivated as they are not the local minimum of any other node (see Prop. 1). In case of deactivation, CRK allows to consider *cross-edges* for the purpose of preserving connectivity according to Th. 1: a leaf node u with a *cross-edge* to v creates a new *cross-edge* between u^2_{min} (in u's tree) and v^1_{min} (in v's tree). In the example in Fig. 4.3, during the deactivation of node 4, the cross-edge $(4,3)$ generates a new cross edge $(3,1)$, as $4^2_{min} = 1$ and $3^1_{min} = 3$. Similarly, during the deactivation of nodes 8 and 7, the cross-edge $(7,8)$ generates the two cross edges $(5,3)$ and $(2,3)$. After *phase ii*, each tree has halved its height, and therefore trees in T_h become trees in T_{h-1}, and in particular trees in T_1 become trees in T_0 as their leaf nodes are deactivated.

Note, the tallest *increasing tree* has height at most n, and therefore it requires at most $\lceil \log_2 n \rceil$ iterations of the two CRK phases to be shrunk into a single node (also according to Prop. 4). Moreover, nodes in T_0 may increase the height of the tallest tree at each iteration. The total number of added nodes is at most n. Therefore, the algorithm requires less than $\lceil \log_2 n \rceil$ additional iterations of the two phases to process all of such nodes.

We conclude that the number of phases required by CRK, and therefore of CRACKER iterations, is $2 \cdot \lceil \log_2 n \rceil + 2 \cdot \lceil \log_2 n \rceil$, i.e., $O(\log n)$. □

Theorem 5 (Height of seed propagation tree)**.** *The height of the seed propagation tree is at most h with $h = O(\log n)$.*

Proof. Recall that each directed edge (u, v) added to the propagation tree links a node v being deactivated to a node u which is staying active in the next iteration of the CRACKER algorithm (see Th.2). This implies that the height of the propagation tree is at most equal to the number of iterations taken by the seeds identification phase. Thus, from Theorem 4, it holds that $h = O(\log n)$. □

Theorem 6 (Number of CRACKER iterations)**.** *Given a connected graph G having n nodes, the number of iterations taken by* CRACKER *algorithm is $O(\log n)$.*

Proof. The proof comes directly from the proofs of Theorem 4 and 5. Since the two phases of CRACKER are executed one after the other and both of them have a cost of, in terms of iterations of $O(\log n)$, the total cost of the CRACKER algorithm is $O(\log n)$. □

Theorem 7 (Number of deactivated vertices)**.** *Given a connected graph G, at least $2^t - 1$ vertices have been deactivated after iteration t.*

Proof. Similarly as for Theorem 4, we provide a proof based on the notion of *increasing trees*. Note, the smallest number of deactivations is achieved when only one increasing tree is present in G, otherwise multiple leaf nodes are deactivated on multiple trees. Recall that after t iterations every node initially at level L_i is linked to a node initially in level L_{i-2^t}. This implies that in an *increasing tree* of height h, nodes initially in levels from $h - 2^t$ (excluded) to h cannot be a local minimum after t iterations, i.e., they are not linked to nodes in higher levels. As each level in the initial graph contains at least one node, we conclude that at least $2^t - 1$ vertices have been deactivated after t iterations. □

Theorem 8 (Number of messages per iteration)**.** *Let n be the number of nodes and m the number of edges in the given graph. The number of* CRACKER *messages is $O(\frac{nm}{\log n})$.*

Proof. As in typical *CC* discovery algorithms, the creation of edges in each graph G^t and H^t is implemented with node-to-node messages. Let's consider the first iteration. During the MinSelection step, each node first sends a message to each of its neighbors in G^0 and to itself to select the minimum among them and this requires 2 messages for each edge plus n messages, thus $2m + n$. Then, in the Pruning step each node generates undirected edges for G^1 starting from H^0. Each node follows the pattern of generating an undirected edge between its minimum in H^0 and each of its neighbors. This generates $2m$ undirected edges in G^1 and requires $2 \cdot 2m$ messages. The first iteration has thus a total cost of $6m + n$ messages and generates a new graph G^1 with $2m$ edges. By iterating the same argument, we obtain that the number of edges at iteration t is bounded by $2^t m$.

Given that the number of iterations is $\log n$, we have that the average number of messages per iteration is $\frac{1}{\log n} \sum_{t=1}^{\log n} \left(2^t m\right) \leq \frac{2 \cdot 2^{\log n} m}{\log n} = O(\frac{nm}{\log n})$.

Finally, the Seed Propagation phase requires $n-1$ messages to propagate the seed identifier, as the seed propagation tree contains $n - 1$ edges, i.e. $O(n)$. Thus, the seed propagation has no impact on the message complexity. □

Note, the actual number of messages is much smaller as by removing nodes at each iteration also edges are removed. To corroborate the above

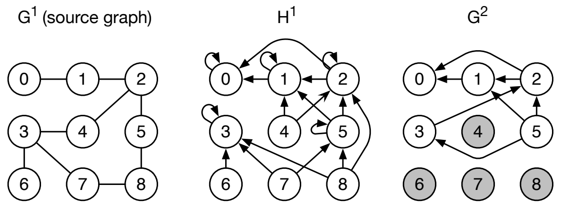


FIGURE 4.4: Seed identification with the EP and OS optimizations.

claim, in the experiments we evaluated the number of nodes and edges for a generic graph (Fig. 4.9a and 4.9b). Results show that the number of nodes and edges decreases exponentially, dramatically reducing the number of messages exchanged per iteration. Moreover, CRACKER always sends a number of message lower than HASH-TO-MIN, for all the tested graphs, as described in Table 4.3.

4.2.5 Optimisations to CRACKER

Most of the algorithms we considered in our study, CRACKER included, exhibit a running time that is highly dependent on the degree of nodes belonging to the graph. In addition, most of them, during their computation, enrich the graph with artificial edges, usually linking the *seed* with the other nodes belonging to the *CC*. As a consequence, the degree of some nodes is considerably increased, sensibly affecting the computational cost of the algorithms. To address this issue, we introduce in CRACKER three optimisations, described in the following.

Edge pruning

Edge Pruning (EP) operates during the MinSelection by reducing the number of redundant edges created, and therefore speeding up the computation. The idea is that when a node is already the minimum of its neighbourhood, it does not need to notify this information to its neighbours as this information is redundant. In EP, if a vertex $u \in G^t$ is a potential seed of its neighbourhood, then it does not add any edge in H^t, as instead would happen in the `ForAll` operation at line 3 in Algorithm 4.2.

More in detail, when a node u is the local minimum in $NN(u)$, i.e., $u = u_{min}$ there are two exclusive cases:

- $z \in NN(u)$ considers u as the z_{min}. In such case z creates the directed edge (z, u). Note that in the original algorithm this edge would be created twice, one time by z and the other by u.
- $z \in NN(u)$ considers another node, say w, as the z_{min}. In this case z creates the directed edges (u, w) and (z, w). In the original algorithm u would have created the edge (z, u), which in this case is useless as w is a better potential seed than z. Note that the correctness of the algorithm holds, as u an w are connected both with and without the optimization.

The second case is shown in Figure 4.4 in H^1, in which EP avoids the creation of the directed edge $(4, 3)$. Instead, in the original algorithm, vertex

3 would have created the edge $(4, 3)$, which is useless since vertex 4 knows a better candidate, i.e. vertex 1.

Oblivious seed

The goal of the Oblivious Seed (OS) optimization is to reduce the number of edges created from potential seeds to other nodes of the CC. This optimization operates in the Pruning, specifically at the `AddEdge` (Line 5) in Algorithm 4.3. In the original version of CRACKER, a generic node u creates a set of undirected edges from $NN_{H^t}(u)$ to u_{min}. With OS, u would create only the directed edges from the $NN_{H^t}(u)$ to u_{min}, in fact creating a directed graph rather than a undirected one. The effect can be seen in Figure 4.4, in which the G^2 graph is a directed graph created by enabling the OS optimization. This optimization yields two benefits:

- reduces the amount of edges created on the potential seed of half. When CCs are large this amount is significant and speeds up the computation. Note that this does not impact on the correctness of the algorithm, since nodes still have direct edges connecting them to better candidates;
- avoids the creation of stars centered on the potential seeds, which makes the computation faster by removing the computational bottlenecks given by potential seeds.

However, the last benefit comes with a cost since the potential seeds cannot connect directly to other potential seeds. This increases the number of iterations needed to the algorithm to reaching a convergence state. In other word, OS realizes the tradeoff between the running time of single iterations (due to the large running time of large stars on potential seed) and the number of iterations. Therefore, OS is enabled at the earlier iterations of the algorithm, when the number of node active is still high and the stars created on the potential seed can be huge. After few iterations, the amount of active node decreases according to Theorem 7 and OS is disabled to favour convergence in a minor number of iterations, rather than decreasing their completion time.

Finish computation sequentially

The third optimisation, Finish Computation Serially (FCS), has been inspired from the work of Salihouglu *et al.* [159] targeting Pregel-like systems. The assumption is that exist algorithms leading to a fast convergence of most of the nodes composing the graph (and the subsequent "deactivation" from computation) but with a small fraction of the graph that requires several additional steps of computation to converge. The idea surrounding their optimisation is to gather into a single machine all the nodes that still require some processing to converge. By means of this mechanism it is possible to avoid the execution of super-steps involving a large set of the computational resources when the actual processing involves only a very small fraction of the input graph. FCS monitors the size of the active subgraph, i.e., the fraction of the graph that still did not converge. When the size of the subgraph goes below a given threshold K, the subgraph is sent to a machine that performs the remaining of the processing serially. By construction, in

Table 4.2: Datasets description

Name	|V|	|E|	β-index	ccNumber	ccMaxSize	diameter	AVG degree	MAX degree
Italy	19,006,129	19,939,100	0.95	153,876	14,694,405	10,534	2.09	16
Twitter [101]	24,159,954	532,138,866	0.05	14,038	24,129,131	N/A	44.05	1,848,376
LiveJournal [135]	5,204,176	77,402,652	0.07	4,533	5,189,809	17	29.75	15023
PLD [133]	39,497,204	623,056,313	0.06	56,304	39,374,588	N/A	31.55	4,933,011
PPI-All [6]	4,670,194	664,471,350	<0.01	16,018	36,255	4	142.28	8,561

CRACKER the set of active vertices is monitored in each iteration to check for the termination (see Algorithm 4.1 Line 6).

4.2.6 Implementation

To validate and fairly evaluate CRACKER with respect to the existing alternative approaches we implemented both our proposed algorithm and all the other solutions using the same methodology, technologies and running environment. All the algorithms have been developed using the Scala language, a Java-like programming language aimed at unifying object–oriented and functional programming. All the implementations are organised according to the MapReduce model exploiting the Apache Spark framework [193]. All the implementations have been realised without any specific code-level optimisation and using the same data structures (i.e. the `Set` structure provided by the Scala base class library).

All the implementations of the tested algorithms have been run using the same installation of Spark, that was already up and running in the computational resources used during the experimental evaluation. Additional details on the running environment are presented in the next section.

The graphs used as input were represented using text files organised as edge-list. Such files have been loaded by Spark framework from an HDFS-based drive. The graphs have been partitioned in a different number of slices, depending on the graph size. The amount of slices is independent from the algorithm, i.e., the amount of partitions in which a graph has been decomposed is the same for any algorithm used for its processing. We make use of the default partitioning strategy available in Spark.

Finally, the logging system has been disabled to avoid a potential overhead, both from the computational and network bandwidth viewpoint.

4.3 Experimental Evaluation

This section evaluates our approach in a wide range of setups. The evaluation has been conducted using both synthetic and real-world datasets. All the experiments have been conducted on a cluster running Ubuntu Linux 12.04 consisting of 5 nodes (1 master and 4 slaves), each equipped with 128 GBytes of RAM and with two 16-core CPUs, inter-connected via a 1 Gbit Ethernet network.

In evaluating the performance of the different competitors, we considered several metrics, including: (i) **time**, as the total time in seconds from the loading of the input graph until the algorithm terminates; (ii) **steps**, as the number of MapReduce steps required; (iii) **message number**, as the total number of messages sent between the map and reduce jobs; **message volume**: as the amount of vertex identifiers sent. All the values considered in the evaluation are the average of 10 independent runs.

4.3.1 Dataset Description

The following datasets have been chosen to build a comprehensive scenario to generalise as much as possible the empirical evaluation of CRACKER. We made all of them publicly available to foster a fair comparison. A summary of datasets' characteristics is presented in Table 4.2.

- **Streets of Italy.** This graph has been generated starting from the data harvested from Geofabrik[3], which collects data from the Open Street Map project [79]. From the whole collection, we extracted the data about Italy. The dataset is characterized by a very large connected component covering the 75% of the entire graph and a large number of smaller CC.
- **Twitter.** A Twitter dataset containing follower relationships between Twitter users has been collected by Kwak *et al.* [101].
- **LiveJournal.** This datasets is one of the most used when comparing different algorithms of this kind on social relationship graphs. It contains social relationships between users of the LiveJournal social network.
- **Pay-level domain (PLD).** The graph has been extracted from the 2012 version of the Common Crawl web corpora and it is publicly available [133]. From the authors' description, in the dataset each vertex represents a pay-level-domain (like uni-mannheim.de). An edge exists if at least one hyperlink was found between pages contained in a pair pay-level-domains. We use this dataset as an undirected graph.
- **PPI-All dataset.** The PPI-All dataset is a protein network describing all the species contained in the STRING database [6]. Vertices correspond to protein and edges correspond to interactions between them thus forming a protein network. Among those the considered datasets, PPI-All is the one with the largest number edges (∼665 millions), but with a diameter as small as 4.

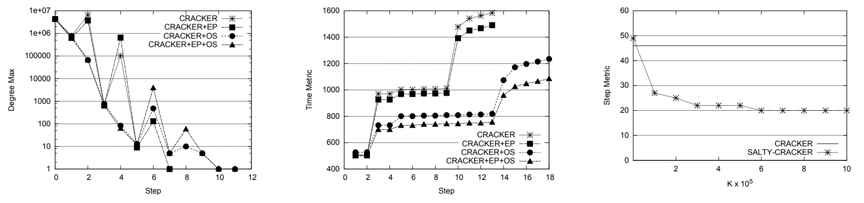
4.3.2 Evaluation of Optimizations

This section discusses the impact of the three optimizations presented in Section 4.2: Edge Pruning (EP), Oblivious Seed (OS), and Finish Computation Sequentially (FCS). In order to test each optimization, both in isolation and in combination, we compare four different versions of the algorithm: (i) the plain CRACKER version, (ii) the CRACKER +EP version, (iii) the CRACKER +OS version, and the CRACKER +EP +OS version. We call SALTY-CRACKER the version of our algorithm with all the optimizations described in Section 4.2.5. Key findings:

- OS allows to reduce both the maximum vertex degree and the number of edges at the cost of extra steps. However, each of these extra steps takes considerable less time and they can be cut with the FCS optimization;

[3] `http://download.geofabrik.de/`

- EP and OS combined give a greater reduction on the completion time with respect to the simple sum of the reductions obtained by the two optimizations in isolation.
- SALTY-CRACKER is faster than CRACKER thanks to the optimizations.



(A) Degree Max with EP and OS (B) **time** metric with EP and OS (C) **steps** metric with FCS optimization

FIGURE 4.5: Evaluation of Optimizations

Edge Pruning and Oblivious Seed

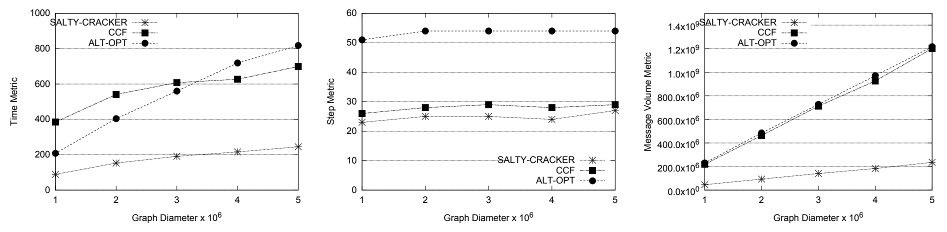
In these experiments we show the effectiveness of the *edge pruning* and *oblivious seed* optimizations. For this evaluation we used the PLD (see Table 4.2) dataset due to its large *CC* composed by the 99% of the entire graph, and for the large number of high degree vertices [133].

Figure 4.5a shows the maximum degree in the graph, and we used this metrics as an indicator of the balance of the computation, as higher values usually indicates unbalanced computations. In Figure 4.5b we report the cumulative completion time. Each of these metrics is sampled at each step of the MapReduce computation, specifically CRACKER executes two steps (MinSelection and Pruning) per algorithm iteration.

The main idea of the EP optimization acts when a node is already the candidate for itself in the MinSelection. In this scenario the node does not need to notify this information to its neighbours. i.e. it still be active in the next iteration and will be notified by neighbours if exist a better candidate. This optimization has few or no impact on balancing as we can see from Figure 4.5a. In some cases (for instance at the 4th step) the highest degree is higher than the one measured with plain CRACKER. However, the number of edges not generated thanks to EP yields a beneficial, even if limited, impact on the completion time (-6%).

The aim of the OS optimization is to avoid potential seed to collect information that are redundant for the identification of the *CC*. With respect to CRACKER +EP, the CRACKER +OS version has a greater impact on both the metrics considered. Regarding the balancing, OS minimizes the creation of high degree vertices at the expenses of few additional steps in the computation. Indeed, while EP converges at the 13th step, OS converges at the 18th. However, these extra steps are much faster and this has great beneficial impact on the completion time in the order of -22% with respect to the plain version and -17% with respect to CRACKER +EP.

It is interesting to notice that the combination of the EP and OS gives a greater reduction on the completion time with respect to the simple sum of the reductions obtained by the two optimizations in isolation. It brings a total improvement of 12% instead of the expected 6%. This confirms that the two optimisations complement each other, in fact allowing for an even larger reduction in the number of edges created.



(A) **time** metric (B) **steps** metric (C) **message volume** metric

FIGURE 4.6: Sensitivity to Diameter

Finish Computation Sequentially

The main goal of the FCS optimization (Section 4.2.5) is the reduction of MapReduce iterations. From our theoretical demonstration the number of steps are primarily affected by the diameter of the graph. Therefore, to test the FCS optimization we synthetically generated a *path* graph with 5×10^6 vertices with randomly distributed identifiers.

TABLE 4.3: Performances with real world datasets: **message number** and **message volume** are values $\times 10^6$

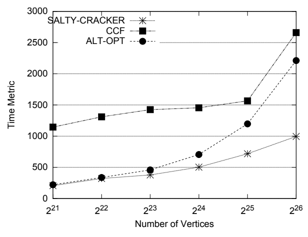
Twitter	**time**	**steps**	Msg	Vol
SALTY-CRACKER	**898**	9	1589	3520
CRACKER	1650 (1.84×)	12	1603	4001
CCF	3215 (3.58×)	7	819	5500
ALT-OPT	2230 (2.48×)	15	4158	8316
HASH-TO-MIN	9222 (10.27×)	7	2920	9807
SGC	15409 (17.16×)	72	1946	5743

PLD	**time**	**steps**	Msg	Vol
SALTY-CRACKER	**1105**	10	2282	5218
CRACKER	1592 (1.44×)	13	2522	6302
CCF	20742 (18.77×)	7	1796	8690
ALT-OPT	8583 (16.82×)	15	4477	9378
HASH-TO-MIN	> 30×			
SGC	> 30×			

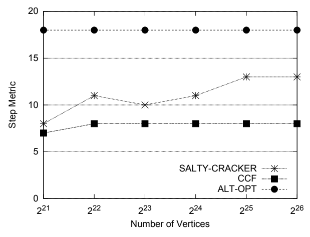
PPI-All	**time**	**steps**	Msg	Vol
SALTY-CRACKER	**330**	6	893	1952
CRACKER	359 (1.09×)	12	896	2136
CCF	1247 (3.78×)	6	239	3733
ALT-OPT	797 (2.42×)	15	1887	3774
HASH-TO-MIN	415 (1.26×)	6	1104	4360
SGC	1957 (5.93×)	72	359	3799

Italy	**time**	**steps**	Msg	Vol
SALTY-CRACKER	**1338**	30	745	1479
CRACKER	1381 (1.03×)	33	780	1734
CCF	1889 (1.41×)	18	1214	4744
ALT-OPT	2052 (1.53×)	39	1864	3727
HASH-TO-MIN	2071 (1.55×)	18	1774	6457
SGC	> 30×	> 114		

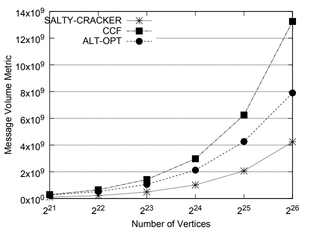
LiveJournal	**time**	**steps**	Msg	Vol
SALTY-CRACKER	**201**	10	246	536
CRACKER	297 (1.48×)	12	258	639
CCF	313 (1.56×)	6	176	1056
ALT-OPT	345 (1.72×)	15	462	925
HASH-TO-MIN	620 (3.08×)	7	408	1562
SGC	1087 (5.41×)	72	436	1136



(A) **time** Metric

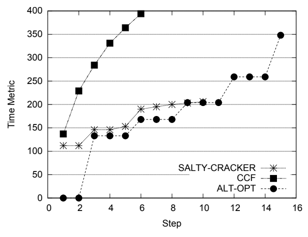


(B) **steps** Metric

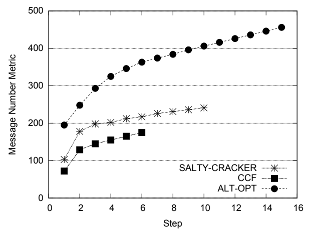


(C) **message volume** metric

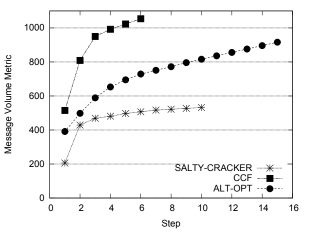
FIGURE 4.7: Sensitivity to Vertices Number



(A) LiveJournal: **time** metric



(B) LiveJournal: **message number** metric



(C) LiveJournal: **message volume** metric

FIGURE 4.8: Step By Step Comparison

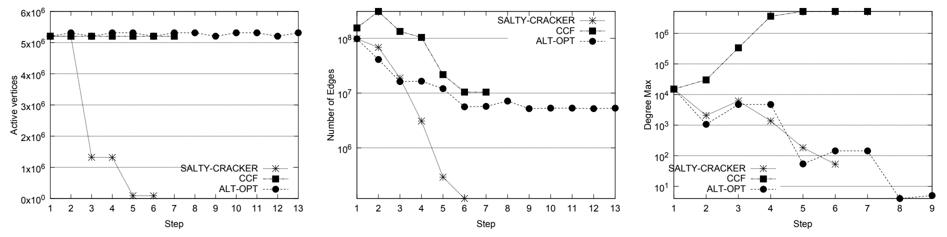
The number of steps are reported in Figure 4.5c, as a function of the parameter K of the FCS optimization, which we varied in the range 0-1,000,000 (when the number of active vertices is below K we switch to serial computation). From the figure it is evident that the FCS optimization help reducing the number of steps. For instance, with $K = 2 \times 10^5$ the number of steps halves with respect to the CRACKER implementation.

4.3.3 Comparison with the State of the Art

We compared SALTY-CRACKER with the following competitors: (i) CCF [93], as we found it to be the best competitor in [122], (ii) ALT-OPT [99] (their fastest MapReduce implementation), (iii) SGC [147] as it is the most recent approach we know of, (iv) HASH-TO-MIN [151] because it is the de-facto standard for CC computation in MapReduce. To conduct a fair comparison, we implemented all the algorithms within the same Apache Spark [193] framework and with the same code-level optimisations. All source code used for the experimentation is publicly available. We show below that SALTY-CRACKER is the best performing algorithm, effectively reduces both the number of vertices and edges thanks to the proposed pruning strategy.

Performance on Real World Graphs

Table 4.3 presents a summary of the results obtained by the execution of SALTY-CRACKER and the competitors on all the real datasets. In terms of **time**, SALTY-CRACKER is the fastest approach with all the datasets. Apart from the plain version of CRACKER, best competitors are either ALT-OPT or CCF, except for PPI-All in which HASH-TO-MIN resulted to be the best competitor, suggesting that it works nice with dense graphs. In terms of **message number** CCF is better than SALTY-CRACKER in all graph datasets



(A) LiveJournal: Active Vertices (B) LiveJournal: Number of Edges (C) LiveJournal: Degree Max

FIGURE 4.9: Graph Topology Evolution

except Italy, but when considering **message volume**, SALTY-CRACKER is the most efficient solution in all datasets. Finally, CCF is the best solution for **steps** with any dataset. Interestingly, we can observe how CCF adopts a very different strategy than SALTY-CRACKER. While the former sends large messages in a few number of iterations, the latter sends many small messages over a large amount of iterations. Conversely, ALT-OPT employs a lot of communication over a large number of iterations. However, in our experimental setup, it is clear that the approach of SALTY-CRACKER is the one guaranteeing the lowest execution times.

Sensitivity to Diameter

To measure the sensitivity to diameter of the algorithms, we generated 5 *path* graphs with diameter in the order of 10^6, in which identifiers are randomly distributed. In this experiments we considered CCF and ALT-OPT, resulting the best competitors from the performances on real world graphs. Figure 4.6a depicts the results with the **time** metric in function of the diameter. SALTY-CRACKER outperforms the others, being 3.5 times faster than the best competitor (CCF) with the largest diameter. In addition, SALTY-CRACKER shows good scalability, as the running time grows slower than competitor when increasing diameter. Figure 4.6b presents the results for the **steps** metric. The results show that the number of steps is stable in all the approaches, with the ALT-OPT requiring more steps than CCF and SALTY-CRACKER, which obtains the best results thanks to the FCS optimization. In terms of **message volume** (Figure 4.6c), SALTY-CRACKER requires 3 times less messages than the competitors.

Sensitivity to Vertices Number

In order to investigate the scalability of SALTY-CRACKER with respect to our competitors we synthetically generated 6 datasets with an increasing number of vertices (from 2^{21} to 2^{26}), using the Erdos-Renyi random graphs model bundled with the Snap library [106]. Each graph consists of 100 connected components approximately of the same size.

Figure 4.7a reports the results for the **time** metric. With a smaller number of vertices, until 2^{23}, SALTY-CRACKER performs similarly to ALT-OPT. However, as the graph size increases, the performance of SALTY-CRACKER gets better than ALT-OPT. The CCF algorithm is always at least 5 times slower than SALTY-CRACKER. In Figure 4.7b we show the results for the **steps** metric. CCF and ALT-OPT are very stable, while SALTY-CRACKER slightly increases in the number of steps. However, SALTY-CRACKER stays much below the theoretical bound of $O(\lceil \log_2 d \rceil)$ given in the Theorem 4. The results of the **message volume** are presented in Figure 4.7c. SALTY-CRACKER is always the algorithm requiring less communication cost with respect to the competitors thanks to its pruning mechanism. For instance, for the dataset having 2^{25} vertices, SALTY-CRACKER requires a **message volume** of approximately 2×10^9, instead ALT-OPT requires 4×10^9 and CCF 6×10^9.

Step by Step Evaluation

This evaluation was conducted by measuring the cumulative of **time**, **message number**, and **message volume** metrics. Figure 4.8 reports the results of the most representative dataset, i.e. the LiveJournal. The other datasets exhibited similar results. The analysis of the results unravelled several interesting properties of the algorithms.

The cost per steps (in terms of all the metrics considered) is high in the starting steps for all the algorithms. However, thanks to the pruning mechanism and the optimizations, the performances of SALTY-CRACKER *increase* in the steps subsequent to the initial ones, while the competitors performances decrease or remain constant. For example, the time per step between the 3rd and 5th steps is almost zero resulting in a fast regain of computational time with respect to the closest competitor ALT-OPT.

An interesting thing to notice is how SALTY-CRACKER and ALT-OPT groups the **time** into bunches of three iterations. This is due to the lazy computation mechanism of Spark, which triggers the computations only when an explicit output is required (i.e. to check the termination). By comparison, the CCF algorithm requires an explicit output at every iteration, so its running time is spread over all the steps.

Regarding SALTY-CRACKER, the transition from the *seed identification* to the *seed propagation* phase is clearly visible due to a peak in the computation time at the 6th step in Figure 4.8a. The reason of the peak is because the propagation tree is stored in a separate RDD, whose lineage is resolved by the Spark framework only at the start of the propagation phase, with the consequent increment in the computational time. To remove this peak, we experimented with further optimization (e.g., starting the seed propagation phase while the seed identification was still active) but all of them resulted in longer total running time.

Graph Topology Evolution

As we described in Section 4.2, one of the innovative features of SALTY-CRACKER is the pruning of vertices. A positive side effect of reducing vertices is the reduction in the number of edges. Figure 4.9 provides empirical results about the benefits of the pruning considering the LiveJournal computation.

Figure 4.9a shows that the pruning mechanism of SALTY-CRACKER is very effective in reducing the number of active vertices. For example, after only one Pruning (at step 3), the number of active vertices is 1/3 of the original input graph. Rather, in the competitors the number of active vertices is practically the same during all the computation. Figure 4.9b shows that the number of edges decreases in the initial steps of the computation for both

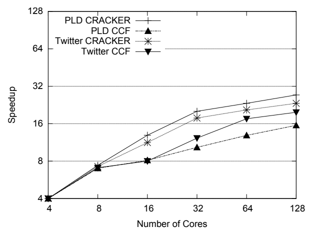


FIGURE 4.10: Scalability evaluation

SALTY-CRACKER and ALT-OPT. Also, thanks to the node pruning, SALTY-CRACKER overperforms ALT-OPT in the number of edges, although ALT-OPT has been designed to reduce edges.

Figure 4.9c shows that, for SALTY-CRACKER and ALT-OPT, the maximum degree decreases during the computation, and it is small in relation with the size of the graph. By comparison, the maximum degree of CCF increases during the computation.

4.3.4 Scalability

Finally, we tested the scalability of SALTY-CRACKER, as well as comparing its performance with CCF, by varying the number of cores in the range [4-128]. Figure 4.10 depicts the results we achieved with the PLD and Twitter datasets. Similar patterns have been observed with the other datasets. In all the tests SALTY-CRACKER always provides a better level of scalability than its competitors. We obtained an almost linear scalability using 8 cores, a still good level scalability with 16 cores, then the value tends to stabilise, providing only a small advantage going from 64 to 128 cores.

These results can be motivated with several considerations about the testing environment. Spark allocates the cores according to a round robin policy: when using 4 cores Spark exploits one core from each of the 4 machines. As a consequence, by using only 4 cores (of the 128 available) we exploit the total amount of memory available in the cluster. Considering that each machines has two CPUs, we reach the maximum available CPU-memory bandwidth, and thus linear scalability, when using 8 cores (one core per CPU). Since finding connected component with Spark is essentially a memory-bound problem, adding more cores and keeping fixed the amount of memory scales only marginally.

4.4 Conclusion

In this chapter we described CRACKER, an algorithm for finding connected components. The CRACKER algorithm is organised in two distinct phases. The first one consists in an iterative process that is, in turn, structured in two alternating steps. The first step is devoted to the identification of the vertex having the smallest identifier that will be used as the *CC*s identifier, whereas the other step performs the graph simplification through vertices pruning. The second phase of CRACKER is aimed at labelling each node with the *id* of the *CC* whom it belongs to.

The experiments have been conducted on a wide spectrum of synthetic and real-world data. In all the experiments CRACKER proved to be a very effective and fast solution for finding CCs in large graphs. In terms of time, CRACKER outperforms its competitors in every dataset used. In addition, CRACKER generated the least volume of messages among all its competitors.

This algorithm has been important for the outline of this thesis for many reasons. It exploits a well defined simplification technique that is able to provide the same result of state of art solution in less time. Also, it arises as the first building block to construct more complex algorithms like the clustering algorithms proposed in the following chapters. In such algorithms CRACKER implements a fundamental phase in order to achieve good clustering results.

Chapter 5

Clustering

Data clustering is a fundamental methodological tool supporting data analysis that consists in finding groups of related items, according to a given definition of similarity. Data clustering can be challenging to perform when dealing with big amount of data: this calls for the design of scalable algorithms, capable of ingesting millions of data points and cluster them in meaningful ways. Scalability, is not the only feature required when developing a clustering algorithm, also the *versatility* of the approach is of paramount importance. Most solutions are specialized in clustering items having specific shapes. For instance, when clustering text data, the widespread K-means clustering algorithm requires text data to be transformed into d-dimensional vectors to operate correctly. However, such transformations generally imply high-dimensional vectors, making distance functions problematic, an issue known as the "curse of dimensionality" [167]. Similarly, other quite diffused approaches for text clustering, based on frequency analysis of groups of letters suffers from high-dimensionality problems. Moreover, the common techniques discussed above are not suitable for non-metric spaces (*i.e.,* those in which the triangle inequality does not hold), which is a feature of the application domain we are considering. As a consequence, this may result in poor clustering performances.

To overcome this limitations, we conceived and realized a novel approach to clustering. This work has been defined and developed during my period abroad spent at Eurecom in Sophia-Antipolis (France), during my second PhD year. The key point of this work is exploiting already defined algorithm in TLAV to construct larger solutions. In particular, this chapter will show how to perform clustering making use of the previously defined algorithm for connected components in Chapter 4. The gist of our approach consists in building an *approximate* k-NN graph of the input text data, and compute its connected components, which identify data clusters. Also, we aim at understanding the trade-off that exists between accuracy, scalability and, ultimately, clustering quality. To this end, we perform an extensive experimental evaluation of our method with real, large-scale datasets, using our implementation for the Apache Spark framework. Finally, one of our goals is to borrow the versatility from a popular k-NN algorithm and verify if it is possible to achieve the same versatility in a clustering algorithm.

In summary, the contributions of this work are the following ones:

- we design and implement a *scalable* algorithm for text clustering, which works in an "adversarial" setting, and that produces high quality clusters;

- we perform a detailed experimental analysis, where we show the impact of the parameters that govern the degree of approximation of our method. Our results indicate that even rough approximations are sufficient to obtain high quality clusters;

- we use real-life datasets and evaluate the overall clustering quality of our approach both using traditional metrics and with the help of domain experts through manual investigation, highlighting the interpretability of clustering results.

The proposal of this clustering algorithm has been published in IEEE Conference on Big Data with the title *"Scalable k-NN based text clustering"* [117].

5.1 Related Work

Data clustering has been widely studied in the literature, with nuances ranging from graph theoretic and data mining principles [68, 8] to experimental approaches [165, 195].

One of the most popular algorithm for clustering is K-means [111], which is a simple approach that can be used to perform clustering, where K indicates the number of clusters the algorithm produces. Since K-means operates on d-dimensional vectors, text clustering requires a *transformation phase* to encode sentences and words into vectors [95, 171]. For example, Mikolov et al. [134] present an efficient implementation of the continuous bag-of-words and skip-gram architectures for computing vector representations of words. In our work, we use such approach as a **baseline** to which we compare our method, and show that it suffers from the underlying inability to accept *non-metric* distance measures, which are essential to detect similarity between mangled sentences, and from its poor scalability.

Alternative approaches search for frequent terms in the dataset to identify clusters [18, 17, 127]: the idea is to find subsets of frequent term sets, which are a proxy for clusters, and map data items containing elements of such subsets to the same cluster. Such approaches scale poorly, and do not take into account similarity metrics resilient to mangling.

Other approaches aim to optimize the computation of pairwise similarity between text items using matrix computations [141]. In this category, recently, Lin et al. [109] present an optimized algorithm to retrieve clustering of text data from a similarity matrix, using cosine similarity. In general, such approaches do not accommodate non-metric similarity measures and are difficult to scale, although recent work [23] has shown the benefits of approximate matrix operations, which scale better than exact, all-pair similarity computations.

An approach that targets goals that are similar to ours is Triage [169], which addresses the same application domain we target in this chapter. However, the focus of Triage is on multi-feature data items, and not on scalability: the authors mainly address problems related to information fusion, by defining a method to merge several different distance metrics operating on text, categorical and numerical values. Recently, a parallel version of Triage has been proposed [163], which partially addresses scalability issues. However, the approach still computes all-pair similarity among representative, prototype items, with an $O\left(n^2\right)$ complexity that still makes handling very large data sets difficult.

5.2 k-NN based clustering

We present our approach for text clustering, which is based on a scalable, randomized algorithm, and recognizes the role played by *approximation* which constitutes an important contribution to our analysis of the trade-off that exists between clustering quality and the scalability of our method.

The problem of text clustering we consider is particularly challenging due to the application scenario we study. We face an adversarial setting in which text data is generated such that finding similar items is cumbersome: SPAM campaigns introduce text mangling, spelling errors and generally variations on some baseline text which makes SPAM items belonging to the same campaign appear different one from each other. As a consequence, we need to use a similarity metric between items that can overcome, or at least mitigate, the problem.

Similarity Metric. There exist numerous similarity metrics in the vast literature on the subject of this work. In particular, for text data, the Hamming distance and Levenshtein distance have been extensively used to determine the similarity among text items. In this work, for the reasons illustrated above, we choose the Jaro-Winkler [87, 183] similarity metric which, simply stated, counts the common characters between two strings even if they are misplaced, misspelled, and mangled by a "short" distance. Note that, the Jaro-Winkler metric has a codomain in $[-1, 1]$.

Given the choice of the similarity metric we use in this work, the wide spectrum of techniques to find clusters of similar text items reduces to few methods. This is the main driving factor that steers the algorithmic design choices we make in this work.

5.2.1 Phase 1: k-NN Graph Construction

Algorithm 5.1: k-NN construction

```
procedure Map(Node n, NeighborList(n))
    forall the $u \in NeighborList(n) \cup n$ do
        forall the $v \in NeighborList(n) \cup n \setminus u$ do
            EMIT($u$, ($v$, SIMILARITY($u$, $v$)))
        end
    end
procedure Reduce(Node n,List[(Node u, Similarity s)] $l$)
    orderedList = ORDERDESC($l$).LIMIT (k)
    EMIT($n$, orderedList)
```

In the first phase of our method, we build a k-NN graph. Essentially, the construction of a k-NN graph is the process of building a directed graph from a set of items V, with vertex set equals to V and an edge from each $v \in V$ to its k most similar items in V under a given similarity measure. In this work, we limit our attention to text features: for example, we extract the subject of a SPAM email as the only representative feature of the item. Considering additional, heterogeneous features is outside the scope of this work, and we defer it to an extension of our approach.

The naïve approach to build a k-NN graph consists in finding all-pairs similarity among all items of a dataset, then select the k most similar items to each item. Clearly, this "brute-force" approach is not scalable, as it requires

$O(n^2)$ similarity computations, where n is the number of items in the dataset. Note that the "brute-force" algorithm produces *exact* k-NN graphs, which we use in this work as a baseline to determine the approximation quality of our method.

In this work, we design a parallel version of the NNDescent algorithm [60], which is an elegant, and widely used method to build approximate k-NN graphs through an iterative procedure.[1] From NNDescent, our approach inherits the capability of using *arbitrary* similarity metrics, including Jaro-Winkler. Although alternative approaches to build k-NN graphs exist, for example using locality sensitive hashing (LSH) [194, 150], such methods do not extend to arbitrary similarity metrics.[2]

The main idea behind the k-NN graph algorithm we use in the first phase of our method is to iteratively improve an initial, random k-NN graph, by "swapping" the neighborhood of each node, searching for similar candidates among its two-hop neighborhood. Increasing the number of iterations allows the algorithm to converge to better approximations of the k-NN graph, at the cost of higher convergence time.

Algorithm 5.1 illustrates the pseudo-code of a generic iteration of our parallel k-NN graph algorithm, which we cast through the MapReduce programming model. Note that the output of the algorithm is a *weighted* k-NN graph, where each edge is labelled with the similarity measure between its end vertexes. First, we initialize the algorithm by building a random, undirected k-NN graph:[3] each node of the graph (*i.e.* a data item) is assigned k random neighbors. In the "map phase", the algorithm accepts as input the random k-NN graph, explores the two-hop neighborhood of each node and computes the similarity among each pair, as shown the `Map` procedure. Note the *vertex-centric* nature of the algorithm: each vertex n considers a couple of its neighbours (u,v) and "unselfishly" computes the similarity between them. The result of this computation is then sent to the interested nodes through the `EMIT` operation. In the reduce phase, `Reduce`, each node selects its top-k similar nodes, and produces a new approximated k-NN graph, that is used as an input for the next iteration of the algorithm. Note that `NeighborList` is composed of `(Node, Similarity)` pairs.

In this work we are particularly interested in the role of the number of iterations of the algorithm, which determines its approximation quality. We claim that even rough approximations of the k-NN graph are sufficient for the ultimate goal of our clustering method. Intuitively, the existence of a path between similar items on the k-NN graph is sufficient for the last phase of the clustering algorithm we propose.

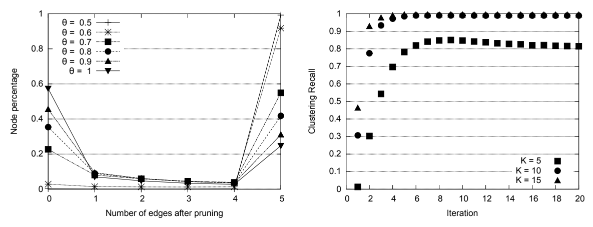
5.2.2 k-NN Graph Pruning

The iterative procedure to build an approximate k-NN graph may induce neighboring relations among text items that have a low pairwise similarity. Indeed, the algorithm necessarily outputs the k most similar neighbors for each item: any skew in the distribution of the pairwise similarities may

[1]Although an Hadoop MapReduce version of NNDescent is discussed in [60], we are not aware of any experimental validation of it. Moreover, in our work we use Spark, a more efficient MapReduce framework geared toward iterative algorithms.

[2]We are currently working on an extension of this work to include modern LSH-based algorithms that can support arbitrary similarity metrics [90, 53, 52] as well.

[3]We omit the pseudo-code of this phase, as it is trivial.



(A) Edges Removed with k=5. (B) Iterations vs. Clustering recall.

FIGURE 5.1: Impact of the pruning phase threshold θ. Impact of number of *iterations* for the k-NN graph construction phase of our algorithm. Clustering quality in terms of clustering Silhouette for the *full* Symantec dataset.

produce a k-NN graph in which some nodes are only “loosely” similar. We thus introduce a *pruning phase*, that uses a parameter $\theta \in [0, 1]$ to determine a cut-off similarity value, below which edges between any pair of nodes are eliminated. The pruning phase inspects each node of the k-NN graph, and prunes such edges.

Choosing an appropriate threshold θ determines the final output of our clustering method: as $\theta \to 1$ clustering is *strict*, which leads to a large number of small clusters of essentially identical items; as $\theta \to 0$ clustering is *loose*, leading toward the degenerate case of a single, giant cluster.

5.2.3 Phase 2: Connected Components

The second and last phase of our approach uses the pruned k-NN graph, and outputs its connected components, that we use as a proxy for identifying clusters of similar items. Recall that the problem of finding the connected components of a graph amounts to searching for sub-graphs in which any two vertexes are connected to each other by paths.

Finding connected components in large-scale graphs using scalable algorithms is a well studied and understood problem, as shown in the rich literature on the subject [151, 99, 122, 123].

Here, we use the connected component algorithm described in Chapter 4 as the building block for the second phase.

5.3 Experimental Setup

This section provides details about our experimental setup, including datasets used, evaluation metrics, parameters and system environment.

Experimental platform. All the experiments have been conducted on a cluster running Ubuntu Linux consisting of 17 nodes (1 master and 16 slaves), each equipped with 12 GB of RAM, a 4-core CPU and a 1 Gbit interconnect.

To implement our approach and the baseline method we use for our comparative analysis using Apache Spark [2]: our source code is publicly available[4].

Evaluation Metrics. We now discuss the metrics we use to analyse the parameter space of our approach, and for its global validation in terms of

[4]`https://github.com/alessandrolulli/knnMeetsConnectedComponents`

clustering quality. Also, we manually investigate the clusters we obtain, using domain knowledge to evaluate the goodness of clustering.

We study the role of the parameters of our approach using the following metrics:

- **N. of clusters**: measures the number of clusters identified by the clustering algorithm. If not otherwise stated, we only consider clusters to be "useful" if they have more than 1,000 elements because small clusters are not valuable for a manual investigation by expertises;
- **Largest cluster size**: measures the size of the largest cluster identified by the algorithm.

We compute clustering quality using well-known metrics [56, 110], that we report below:

- **Compactness**: measures how closely related the items in a cluster are. We obtain the compactness by computing the average pairwise similarity among items in each cluster. Higher values are preferred.
- **Separation**: measures how well clusters are separate from each other. Separation is obtained by computing the average similarity between items in different clusters. Lower values are preferred.
- **Silhouette** [155]: constitutes an aggregate metric, that takes into account the inter- and intra-cluster pairwise similarity between items. Higher values are preferred.
- **Recall**: this metric relates two data clustering obtained by different methods. Using clustering C as a reference, we compute the recall of clustering D by computing the fraction of items that belong to the same cluster in both C and D. In particular, we use as a reference the exact clustering we obtain with the "brute force" approach to compute the k-NN graph. Higher values of recall are preferred.

It is important to notice that computing the above metrics is computationally as hard as computing the clustering we intend to evaluate. For this reason, we resort to uniform sampling: instead of computing the all-to-all pairwise similarity between items, we pick items uniformly at random, with a sampling rate of 1%, increasing it up to 10% for small clusters.

The datasets. The main dataset we use in our evaluation consists of a subset of SPAM emails collected by Symantec Research Labs, between 2010-10-01 and 2012-01-02, which is composed by $3,886,371$ email samples. Each item of the dataset is formatted according to JSON and contains the common features of an email, such as: subject, sending date, geographical information, the bot-net used for the SPAM campaign as labeled by Symantec systems, and many more. For instance, a subject of an email in the dataset is "19.12.2011 Rolex For You -85%" and the sending day is "2011-12-19". A deeper description of such dataset can be found in Chapter 7 where we describe the application for Spam Campaign Detection.

To cross-validate our approach on a different dataset, we also use a data obtained using the Twitter API, and consisting of $1,530,623$ tweets in JSON format.

5.4 Results

In this section we present our result, and we organize it as follows. First, we analyze the parameter space of our algorithm, and discuss the impact of such parameters on the metrics we defined above. Then, we focus on clustering quality, and compare the performance of our approach to that of the *baseline* algorithm we discuss in section 5.1. Finally, we study the clustering scalability.

Analysis of the parameter space. First, we summarize the parameters underlying our algorithm and discuss about their role. Our approach has 3 main parameters: k, the number of neighbors to construct the k-NN graph; the number of *iterations* of the first phase of the algorithm; and θ, the pruning threshold.

The experimental results we show in this section are obtained with a sampled version of the Symantec dataset, and account for 800,000 data items. A sampled dataset allows us to execute the “brute force” method to compute the k-NN graph.

In what follows, we let the number of *iterations* and θ to be free parameters, and instead select a few representative values for k. We chose k to be small, *i.e.*, we allow a few neighbors per node in the k-NN graph.

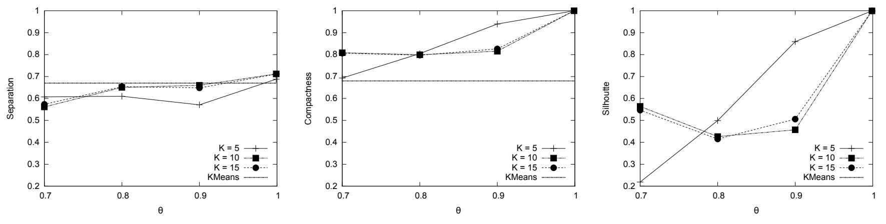
Impact of the pruning threshold θ. We now discuss how the pruning mechanism modifies the k-NN graph, and what is the impact on clustering. As discussed in section 5.2, as θ tends to 0, pruning is less effective, and the k-NN graph tends to have a single giant component. Instead, when θ tends to one, only very similar neighbors survive pruning, and the k-NN graph is fractioned in a large number of small clusters.

Figure 5.1a shows the fraction of nodes for which a given number of edges are removed after pruning, as a function of θ. For values of $\theta < 0.8$, pruning is less effective, as the number of pruned edges is small. Instead, for $\theta > 0.9$, a large fraction of nodes remain with one or fewer edges after the pruning phase. This translates in sizes of the largest clusters to approach the entire dataset, for $\theta = 0.5$ already, or to be extremely small, for $\theta = 1$.

Overall impact of approximation. We now study the impact of the k-NN graph approximation on clustering quality, by analysing the deviation of our approach from the results obtained from an *exact* k-NN graph computed using the “brute force” approach. Our results indicate that approximate k-NN graphs obtained with a low k and few *iterations* are sufficient to obtain data clustering that is practically indistinguishable from that obtained by an onerous $O(n^2)$ k-NN graph construction phase.

Figure 5.1b shows how clustering recall varies as a function of the k-NN *iterations*. As shown in the Figure, $k = 10$ and 5 *iterations* are sufficient to obtain a clustering which is *essentially identical* to that obtained with the *exact* k-NN graph. Even a very low value of $k = 5$ settles to a 0.8 recall, after roughly 10 *iterations*. Even rough approximations of the k-NN graph, obtained with a small number of *iterations*, are sufficient for the algorithm to stabilize.

Analysis of the clustering quality. We now move to a global evaluation of the algorithm we present in this work, and compare clustering quality to the **baseline** algorithm described in section 5.1. In particular, we use the efficient K-means implementation available in Spark's MLLib package [3], and the `word2vec` package [5], as illustrated in [4]. If not otherwise



(A) Separation (lower is better). (B) Compactness (higher is better). (C) Silhouette (higher is better).

FIGURE 5.2: Twitter Dataset: Clustering quality in terms of inter and intra cluster similarity, and clustering Silhouette. As for the Symantec dataset, the algorithm proposed in this work outperforms the baseline method in all metrics.

TABLE 5.1: Twitter Dataset: Manual Investigation

cluster size	sample
1542	Wind 9.1 km/h WSW. Barometer 1002.0 hPa, Falling. Temperature 0.5. Rain today 0.3 mm. Humidity 34% Wind 0.0 mph —. Barometer 1022.4 mb, Steady. Temperature 21.4. Rain today 0.00 in. Humidity 79%
194	"#IfIHadItMyWay I would have a fast metabolism so I could eat MORE!" "#IfIHadItMyWay Chicken strips would be served everyday at ranger" "#IfIHadItMyWay school would be just gym. And back home"
564	"Miami Valley Hospital: CLINICAL NURSE - CNN (#TROY , OH) http://t.co/adVBYzCK #Nursing #Job #Jobs #TweetMyJobs" "Miami Valley Hospital: OB SURGICAL TECH (#DAYTON , OH) http://t.co/Bh80vn0k #Healthcare #Job #Jobs #TweetMyJobs"
228	"#PrettyLittleLiars was intense!!" "#prettylittleliars marathon!" "Pretty Little Liars with breakfast :)"
773	"I just became the mayor of Roszkowski Haus on @foursquare! http://t.co/4RzAisZg" "I just became the mayor of Bertha's Place on @foursquare! http://t.co/t07P13EL" "I just became the mayor of Mini Mini Mart on @foursquare! http://t.co/d6xFIIQM"

specified, we set $K = 1000$ such that the baseline algorithm output 1,000 clusters similarly to our approach. Also, this configuration yields the best result in term of Silhouette metric.

In this section, we cross-validate our results with the Twitter dataset, which includes $1,530,623$ tweets sent in USA and collected on the day 2012/02/21. This dataset is fundamentally more diverse than the Symantec dataset: the average weight on the generated k-NN graph (*i.e.,* the similarity of the neighbours) is 0.75, as compared to 0.96 for the Symantec dataset.

Figure 5.2 shows separation, compactness and Silhouette, as a function of θ, and for various values of k, for both our approach, and for the baseline algorithm based on K-means (with $K = 1000$). A glance at the Figure indicates that our approach achieves similar or better performance than the baseline method for clustering. It is interesting to notice that for the Twitter dataset, $k = 5$ – which produces very rough approximations of the k-NN graph – achieves better performance than for larger values of k.

We conclude the analysis of the Twitter dataset with a manual investigation of the clusters, as shown in Table 5.1. We focus on clusters with at least 100 tweets: in this case, a smaller cluster size is justified by the lower similarity between tweets as compared to the Symantec dataset. Our results confirm that clusters are meaningful, for example clustering weather forecasts in one case and job positions at the Miami hospital in another. We also have identified clusters related to hashtags such as *#IfIHadItMyWay* and a popular TV-series, *#PrettyLittleLiars.*

Table 5.2: Symantec dataset: breakdown of the algorithm runtime (in seconds)

k-NN graph phase	**Iteration**				
k	5	10	15	20	**CC**
5	675	2293	3185	4897	66
10	2281	4610	5320	7061	81
15	4061	8475	13594	18203	107

Table 5.3: Symantec dataset: K-means, baseline algorithm runtime (in seconds)

	K	**time**
K-means	1000	3498 (1.53×)
	2000	10004 (4.39×)
	3000	28411 (12.46×)
	4000	56008 (24.55×)
Our approach, $k = 10$ and 5 iterations		**2281**

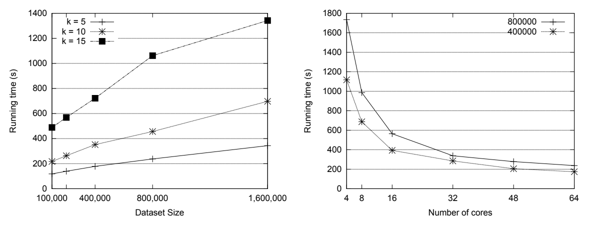
Analysis of algorithm scalability. We study the scalability of our approach and compare it to the baseline algorithm discussed earlier: first, we vary the dataset size maintaining the same number of compute machines that execute the parallel algorithms, then we keep the dataset size constant, and increase the level of parallelism by adding compute machines.

Figure 5.3a, shows the algorithm runtime with varying dataset sizes, using 5 different samples of the Symantec dataset of size 100,000, 200,000, 400,000, 800,000 and 1,600,000 emails respectively. All values plotted are the average of 5 independent executions. Our results indicate roughly a linear scalability with respect to the size of the dataset, an observation that holds irrespectively of the value of k.

Figure 5.3b shows the algorithm runtime as the number of cores we devote to the computation varies between 4 and 64, considering datasets 400,000 and 800,000 items; in both cases, our results indicate a quasi-linear speedup, especially for the biggest dataset. For example, increasing doubling the number of cores from 8 to 16, for the large dataset, cuts almost in half the algorithm runtime.

Finally, Table 5.2, reports the runtime breakdown of the k-NN graph construction phase, and of the connected component phase of our algorithm, for several values of k and for $\theta = 0.9$. Again, all values are the average of 5 independent executions.

As expected, the k-NN graph construction runtime increases both with k



(a) Dataset sizes. (b) Number of cores.

Figure 5.3: Scalability. Our approach scales roughly linearly.

and with the *iterations* number, although more slowly than the worst scale asymptotic analysis and experimental results presented in [60].[5] Note that the first phase of our approach dominates the overall algorithm runtime, as computing the connected components is fast. Once the k-NN graph is built, it is possible to quickly proceed with various versions of the pruning phase (tuning θ for the application at hand) and obtain different clusters.

Table 5.3 illustrates the runtime of the baseline algorithm that uses K-means: the table reports the "slow-down" of the baseline algorithm with respect to our approach, when $k = 10$ and with 5 *iterations*, and for different values of K, the number of clusters K-means constructs. Our approach outperforms the baseline algorithm in terms of end-to-end clustering times, even for small values of K.

5.5 Conclusion

In this chapter, we presented a scalable approach for distributed text data clustering, that accommodates arbitrary similarity measures and that produces high quality clusters.

To overcome the complexity of typical approaches to text clustering, this work studied the role of approximation in establishing a trade-off between high clustering quality and fast algorithmic runtime. We showed, through a detailed experimental campaign, that our method does not require accurate representations of pairwise similarity across data items to produce high quality, interpretable clusters.

In addition, the connected component algorithm CRACKER presented in Chapter 4 has been successfully integrated in this algorithm to show how it is possible to give an implementation of a composed task by the composition of two basic algorithms. Finally, due to the use of k-NN graphs, this solution can be extended to provide density based clustering. We show how this can be achieved in the subsequent chapter.

[5]Recall that we use a different parallel execution framework in our work, Spark, which is geared towards efficient execution of iterative algorithms.

Chapter 6

Density Based Clustering

The clustering algorithm presented in Chapter 5 has been our first approach to clustering. In this Chapter we define and develop an additional clustering algorithm making use of the similar building blocks used in the previous one. In particular, we noticed that, with some improvements, such clustering algorithm can be extended to perform density based clustering. Also, it is possible to maintain, and provide, many features missing in most of the algorithms performing density based clustering. We borrowed from the previously defined algorithm the versatility. This permits to deal with any kind of data and perform density based clustering. In addition, we introduce *approximation* and *simplification* techniques which actively help to reduce the computational cost of the algorithm and to provide a good result in less time.

The main idea is to adapt the previously defined algorithm to approximate DBSCAN. With DBSCAN, Ester et al. [63] introduced the idea of *density-based* clustering: grouping data packed in high-density regions of the feature space. dbscan is very well known and appreciated (it received the KDD test of time award in 2014) because of two very desirable features: first, it separates "core points" appearing in dense regions of the feature spaces from outliers ("noise points") which are classified as not belonging to any cluster; second, it recognizes clusters on complex manifolds, having arbitrary shapes rather than being limited to "ball-shaped" ones, which are all similar to a given centroid.

Unfortunately, two limitations restrict DBSCAN's applicability to increasingly common cases: first, it is difficult to run it on very large databases as its scalability is limited; second, existing implementations do not lend themselves well to heterogeneous data sets where item similarity is best represented via arbitrarily complex functions. We target both problems, proposing an *approximated, scalable, distributed, versatile* dbscan implementation which is able to handle *any symmetric distance function*, and can handle *arbitrary data* items, rather than being limited to points in Euclidean space.

NG-DBSCAN is implemented in Spark, and it is suitable to be ported to frameworks that enable distributed TLAV computation; in our experimental evaluation we evaluate both the scalability of the algorithm and the quality of the results, i.e., how close these results are to those of an exact computation of dbscan. We compare NG-DBSCAN with competing dbscan implementations, on real and synthetic datasets.

Our results show that NG-DBSCAN often outperforms competing dbscan implementations, while the approximation imposes small or negligible impact on the results. Furthermore, we investigate the case of clustering text based on a `word2vec` embedding: we show that – if one is indeed interested in clustering text based on edit-distance similarity – in the existing approaches, the penalty in terms of clustering quality is substantial, unlike what happens with the approach enabled by NG-DBSCAN.

NG-DBSCAN is an approximated and distributed implementation of dbscan. Its main merits are:

- **Efficiency.** It often outperforms other dbscan distributed implementations, while the approximation has a small to negligible impact on results.
- **Versatility.** The vertex-centric approach enables distribution without needing Euclidean spaces to partition. NG-DBSCAN allows experts to represent item dissimilarity through any symmetric distance function, allowing them to tailor their definition to domain-specific knowledge.

Our experimental evaluation supports these claims through an extensive comparison between NG-DBSCAN and alternative implementations, on a variety of real and synthetic datasets.

The NG-DBSCAN algorithm has been accepted for publication in the Proceedings of the VLDB Endowment, Vol. 10, No. 3 with the title "*NG-DBSCAN: Approximate Scalable Density-Based Clustering for Arbitrary Data*".

6.1 Background and Related Work

In this Section we first revisit the dbscan algorithm, then we discuss existing distributed implementations of density-based clustering. We conclude with an overview of graph-based clustering and *ad-hoc* techniques to cluster text and/or high-dimensional data.

6.1.1 The DBSCAN Algorithm

Ester et al. defined dbscan as a sequential algorithm [63]. Data points are clustered by density, which is defined via two parameters: ε and MinPts. The *ε-neighborhood* of a point p is the set of points within distance ε from p.

Core points are those with at least MinPts points in their ε-neighborhood. Other points are either *border* or *noise* points: border points have at least one core point in their ε-neighborhood, whereas noise points do not. Noise points are assigned to no cluster.

A cluster is formed by the set of *density-reachable* points from a given core point c: those in c's ε-neighborhood and, recursively, those that are density-reachable from core points in c's ε-neighborhood. Dbscan identifies clusters by iteratively picking unlabeled core points and identifying their clusters by exploring density-reachable points, until all core points are labeled. Note that dbscan clustering results can vary slightly if the order in which clusters are explored changes, since border points with several core points in their ε-neighborhood may be assigned to different clusters.

For 17 years, the time complexity of dbscan has been believed to be $O(n \log n)$. Recently, Gan and Tao [73] discovered that the complexity is in

Table 6.1: Overview of parallel density-based clustering algorithms.

Name	Parallel model	Implements dbscan	Approximated	Partitioner	Data object type	Distance function supported
ρ-dbscan [73]	single machine	yes	yes	grid	point in n-D	Euclidean
MR-DBSCAN [80]	MapReduce	yes	no	yes	point in n-D	Euclidean
SPARK-DBSCAN	Apache Spark	yes	no	yes	point in n-D	Euclidean
IRVINGC-DBSCAN	Apache Spark	yes	no	yes	point in 2-D	Euclidean
DBSCAN-MR [48]	MapReduce	yes	no	yes	point in n-D	Euclidean
MR. SCAN [182]	MRNet + GPGPU	yes	no	yes	point in 2-D	Euclidean
PARDICLE [143]	MPI	yes	yes	yes	point in n-D	Euclidean
DBCURE-MR [98]	MapReduce	no	no	yes	point in n-D	Euclidean
NG-DBSCAN	**MapReduce**	yes	yes	**no**	**arbitrary type**	**arbitrary symmetric**

fact higher – which explains why existing implementations only evaluated dbscan for rather limited numbers of points – and proposed an approximate algorithm, ρ-dbscan, running in $O(n)$ time. Unfortunately, the data structure at the core of ρ-dbscan does not allow handling arbitrary data or similarity measures, and only Euclidean distance is used in both the description and experimental evaluation.

We remark that the definition of dbscan revolves on the ability of finding the ε-neighborhood of each data point: as long as a distance measure is given, the ε-neighborhood of a point p is well-defined no matter what the type of p is. NG-DBSCAN does not impose any limitation on the type of data points nor on the properties of the distance function, except symmetry.

6.1.2 Distributed Density-Based Clustering

MR-DBSCAN [80] is the first proposal of a distributed dbscan implementation realized as a 4-stage MapReduce algorithm: partitioning, clustering, and two stages devoted to merging. This approach concentrates on defining a clever partitioning of data in a d-dimensional Euclidean space, where each partition is assigned to a worker node. A modified version of PDBSCAN [187], a popular dbscan implementation, is executed on the sub-space of each partition. Nodes within distance ε from a partition's border are replicated, and two stages are in charge of merging clusters between different partitions. Unfortunately, MR-DBSCAN's evaluation does not compare it to other dbscan implementations, and only considers points in a 2D space.

In the Evaluation section, we compare our results to SPARK-DBSCAN and IRVINGC-DBSCAN, two implementations inspired by MR-DBSCAN and implemented in Apache Spark.

DBSCAN-MR [48] is a similar approach which again implements dbscan as a 4-stage MapReduce algorithm, but uses a k-d tree for the single-machine implementation, and a partitioning algorithm that recursively divides data in slices to minimize the number of boundary points and to balance the computation.

MR. SCAN [182] is another similar 4-stage implementation, this time exploiting GPGPU acceleration for the local clustering stage. Authors only implemented a 2D version, but claim it is feasible to extend the approach to any d-dimensional Euclidean space.

PARDICLE [143] is an approximated algorithm for Euclidean spaces, focused on density estimation rather than exact ε-neighborhood queries. It uses MPI, and adjusts the estimation precision according to how close the density of a given area is with respect to the ε threshold separating core and non-core points.

DBCURE-MR [98] is a density-based MapReduce algorithm which is not equivalent to dbscan: rather than circular ε-neighborhoods, it is based on ellipsoidal τ-neighborhoods. DBCURE-MR is again implemented as a 4-stage MapReduce algorithm.

Table 6.1 summarizes current parallel implementations of density-based clustering algorithms, together with their execution environment, and their features. In all these approaches, the algorithm is distributed by partitioning a d-dimensional space, and only Euclidean distance is supported. Our approach to parallelization does not involve data partitioning, and is instead based on a vertex-centric design, which ultimately is the key to support arbitrary data and similarity measures between points, and to avoid scalability problems due to high-dimensional data.

6.1.3 Graph-Based Clustering

Graph-based clustering algorithms [65, 154] build a clustering based on input graphs whose edges represent item similarity. These approaches can be seen as related to NG-DBSCAN, since its second phase takes a graph as input to build a clustering. The difference with these approaches, which consider the input graph as given, is that our approach builds the graph in its first phase; doing this efficiently is not trivial, since some of the most common choices (such as ε-neighbor or k-nearest neighbor graphs) require $O(n^2)$ computational cost for generic distance functions; our approximated approach obtains a substantial cut on these costs.

6.1.4 Density-Based Clustering for High-Dimensional Data

We conclude our discussion of related work with density-based approaches suitable for text and high-dimensional data in general.

Tran et al. [172] propose a method to identify clusters with different densities. Instead of defining a threshold for a local density function, low-density regions separating two clusters can be detected by calculating the number of shared neighbors. If the number of shared neighbors is below a threshold, then the two objects belong to two different clusters. Tran et al. report that their approach has high computational complexity, and the algorithm was evaluated using only a small dataset (below 1 000 objects). In addition, as the authors point out, this approach is unsuited for finding clusters that are very elongated or have particular shapes.

Zhou et al. [196] define a different way to identify dense regions. For each object p, their algorithm computes the ratio between the size of p's ε-neighborhood and those of its neighbors, to distinguish nodes that are at the center of clusters. This approach is once again only evaluated and compared with dbscan in a 2D space.

6.2 NG-DBSCAN: Approximate and Flexible DBSCAN

NG-DBSCAN is an approximate, distributed, scalable algorithm for density-based clustering, supporting any symmetric distance function. We adopt the *vertex-centric*, or "think like a vertex" programming paradigm, in which computation is partitioned by and logically performed at the vertexes of a graph, and vertexes exchange messages. The vertex-centric approach is widely used due to its scalability properties and expressivity [130].

Several vertex-centric computing frameworks exist [125, 77, 1]: these are distributed systems that iteratively execute a user-defined program over vertices of a graph, accepting input data from adjacent vertices and *emitting* output data that is communicated along outgoing edges. In particular, our work relies on frameworks supporting Valiant's Bulk Synchronous Parallel (BSP) model [176], which employs a shared nothing architecture geared toward synchronous execution. Next, for clarity and generality of exposition, we gloss over the technicalities of the framework, focusing instead on the principles underlying our algorithm. Our implementation uses the Apache Spark framework; its source code is available online.[1]

6.2.1 Overview

Together with efficiency, the main design goal of NG-DBSCAN is flexibility: indeed, we can handle data of any type and any distance function to represent dissimilarity between items. The only additional requirement is that the distance function d should be *symmetric*: that is, $d(x, y)$ should be equal to $d(y, x)$ for all x and y. It is technically possible to modify NG-DBSCAN to allow for asymmetric distance, but for clustering – where the goal is grouping similar items – asymmetry is conceptually problematic, since it is difficult to choose whether x should be grouped with y if, for example, $d(x, y)$ is large and $d(y, x)$ is small. If needed, we advise using standard symmetrization techniques: for example, defining a $d'(x, y)$ equal to the minimum, maximum or average between $d(x, y)$ and $d(y, x)$ [67].

The main reason why dbscan is expensive when applied to arbitrary distance measures is that it requires retrieving each point's ε-neighborhood, for which the distance between *all* node pairs needs to be computed, resulting in $O\left(n^2\right)$ calls to the distance function. NG-DBSCAN avoids this cost by dropping the requirement of computing ε-neighborhoods exactly, and proceeds in two phases.

The first phase creates the ε-graph, a data structure which will be used to avoid ε-neighborhood queries: ε-graph nodes are data points, and each node's neighbors are a subset of its ε-neighborhood. This phase is implemented through an auxiliary graph called *neighbor graph* which gradually converges from a random starting configuration towards an approximation of a k-nearest neighbor (k-NN) graph by computing the distance of nodes at a 2-hop distance in the neighbor graph; as soon as pairs of nodes at distance ε or less are found, they are inserted in the ε-graph.

The second phase takes the ε-graph as an input and computes the clusters which are the final output of NG-DBSCAN; cheap neighbor lookups on the ε-graph replace expensive ε-neighborhood queries. In its original description, dbscan is a sequential algorithm. We base our parallel implementation on the realization that a set of density-reachable core nodes corresponds to a connected component in the ε-graph– the graph where each core node is connected to all core nodes in its ε-neighborhood. As such, our Phase 2 implementation builds on a distributed algorithm to compute connected

[1] `https://github.com/alessandrolulli/gdbscan`

components, amending it to distinguish between core nodes (which generate clusters), noise points (which do not participate to this phase) and border nodes (which are treated as a special case, as they do not generate clusters).

NG-DBSCAN's parameters determine a trade-off between speed and accuracy, in terms of fidelity of the results to the exact dbscan implementation: in the following, we describe in detail our algorithm and its parameters; in Section 6.4.1, we quantify this trade-off and provide recommended settings.

6.2.2 Phase 1: Building the ε-Graph

As introduced above, Phase 1 builds the ε-graph, that will be looked up to avoid expensive ε-neighborhood queries in Phase 2. We use an auxiliary structure called neighbor graph, which is a directed graph having data items as nodes and distances between them as edge weights.

The neighbor graph is initialised by connecting each node to k random other nodes, where k is an NG-DBSCAN parameter. At each iteration, all pairs of nodes (x, y) separated by 2 hops in the neighbor graph are considered: if the distance between them is smaller than the largest weight on an outgoing edge e from either node, then e is discarded and replaced with (x, y). Through this step, as soon as a pair of nodes at distance ε or less is discovered, the corresponding edge is added to the ε-graph.

The neighbor graph and its evolution are inspired by the approach that Dong at al. [60] used to compute approximated k-NN graphs. By letting our algorithm run indefinitely, the neighbor graph would indeed converge to an approximated k-NN graph: in our case, rather than being interested in finding the k nearest neighbors of an item, we want to be able to distinguish whether that item is a core point. Hence, as soon as a node has M_{max} neighbors in the ε-graph, where M_{max} is an NG-DBSCAN parameter, we consider that we have enough information about that node and we remove it from the neighbor graph to speed up the computation. M_{max} and k handle the speed-accuracy trade-off: optimal values may vary depending on the dataset, but our experimental study in Sections 6.4.1 and 6.4.1, shows that choosing $k = 10$ and $M_{max} = \max(MinPts, 2k)$ provides consistently good results. We consider automatic approaches to set both variables as an open issue for further work.

Phase 1 is repeated iteratively: details on the termination condition are described in Section 6.2.2.

Example. Figure 6.1 illustrates Phase 1 with a running example; in this case, for simplicity, $k = M_{max} = 2$. The algorithm is initialised by creating an ε-graph with no edges and a neighbor graph with $k = 2$ outgoing edges per nodes chosen at random.

Each iteration proceeds through three steps, indicated in Figure 6.1 with hexagons labeled 1, 2, and 3. In step 1, the directed neighbor graph is transformed in an undirected one. Then, through the transition labeled 2, edges are added to the ε-graph if their distance is $\leq \varepsilon$. For instance, edge $(2, 6)$ is added to the ε-graph in the first iteration. Finally, in step 3 each node explores its two-hop neighborhood and builds a new neighbor graph while keeping connections to the k closest nodes. Nodes with at least M_{max} neighbors in the ε-graph are deactivated (marked in grey) and will disappear from the neighbor graph in the following iteration.

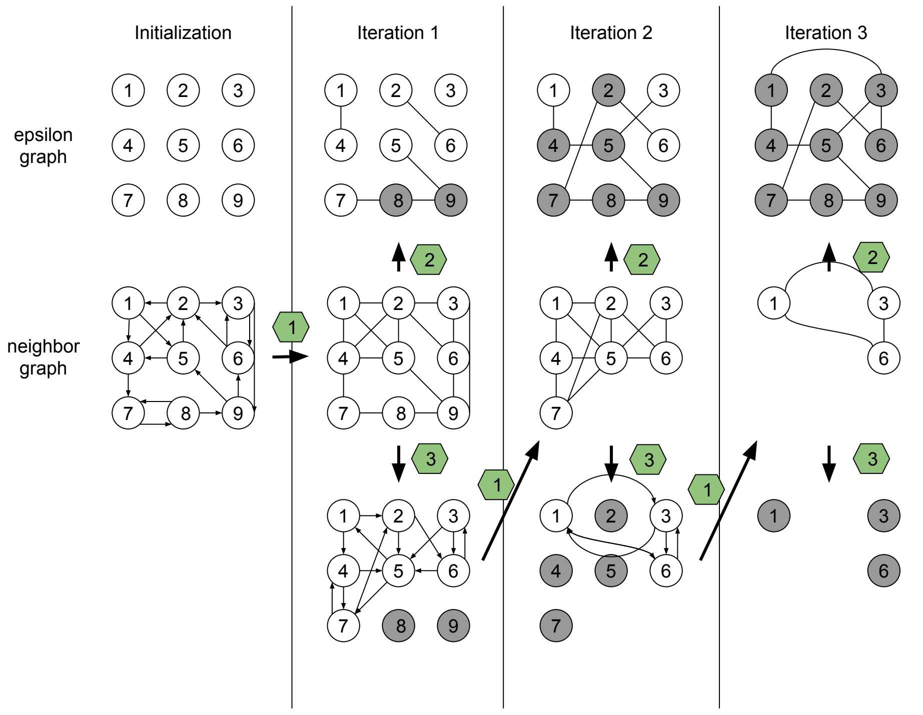


FIGURE 6.1: Phase 1: ε-graph construction.

Termination Condition

In addition to having a maximum number *iter* of iterations to ensure termination in degenerate cases with a majority of noise points, phase 1 terminates according to two parameters: $\mathcal{T}_n$ and $\mathcal{T}_r$.

Informally, the idea is as follows. Our algorithm proceeds by examining, in each iteration, only active nodes in the neighbor graph: the number of active nodes $a(t)$ decreases as the algorithm runs. Hence, it would be tempting to wait for T^* iterations, such that $a(t^*) = 0$. However, the careful reader will recall that noise points cannot be deactivated: as such, a sensible alternative is to set the stop condition to t^* such that $a(t^*) < \mathcal{T}_n$.

The above inequality alone is difficult to tune: small values of $\mathcal{T}_n$ might stop the algorithm too late, performing long computations with limited value in terms of clustering quality. To overcome this problem, we introduce an additional threshold that operates on the number of nodes that has been deactivated in the last iteration $\Delta_a(t) = a(t-1) - a(t)$: we complement our stop condition by finding T^* such that $\Delta_a(t^*) < \mathcal{T}_r$.

In Figure 6.3 we see, from an example run of NG-DBSCAN, the number of active nodes $a(t)$ and of nodes removed from the neighbor graph in the last iteration, $\Delta_a(t)$. Neither of the above conditions alone would be a good termination criterion: both would stop the algorithm too early. Indeed, $\Delta_a(t^*) < \mathcal{T}_r$ can be satisfied both at the early stage of the algorithm or toward its convergence, while $a(t^*) < \mathcal{T}_n$ makes the algorithm progress past the few first iterations. Then, towards convergence, the $\mathcal{T}_r$ inequality allows the algorithm to continue past the $\mathcal{T}_n$ threshold, but avoids running for too long.

We have found empirically (see Section 6.4.1) that setting $\mathcal{T}_n = 0.7n$ and $\mathcal{T}_r = 0.01n$ yields fast convergence while keeping the results similar to those of an exact dbscan computation. In general, the former parameter suggests the minimum number of core nodes available in the dataset whereas the latter drives the stopping of the algorithm when too few nodes are removed in an iteration.

Implementation Details

Since NG-DBSCAN accepts arbitrary distance functions, computing some of them can be very expensive: a solution for this is memoization (i.e., caching results to avoid computing the distance function between the same elements several times). Writing a solution to perform memoization is almost trivial in a high-level language such as Scala,[2] but various design choices – such as choice of data structure for the cache and/or eviction policy – are available, and choosing appropriate ones depends on the particular function to be evaluated. We therefore consider memoization as an orthogonal problem, and rely on users to provide a distance function which performs memoization if it is useful or necessary.

To limit the number of message exchanges, we adopt two techniques. The first is an "altruistic" mechanism to compute neighborhoods: each node computes distances between all its neighbors in the neighbor graph, and sends them the k nodes with the smallest distance. In this way it is not necessary to collect, at each node, information about each of their neighbors-of-neighbors. The second technique avoids that a node with many neighbors sends too many messages. We introduce a parameter ρ to avoid bad performance in degenerate cases, limiting the number of nodes considered in each neighborhood to ρk.

Finally, to avoid memory issues that typically arise in vertex-centric computing frameworks relying on RAM to store messages, we have implemented an option to divide a single logical iteration (that is, a *super-step* in the BSP terminology[130]) into multiple ones. Specifically, an optional parameter $\mathcal{S}$ allows splitting each iteration in $t = \lceil \hat{n}/\mathcal{S} \rceil$ sub-iterations, where $\hat{n}$ is the number of nodes currently in the neighbor graph. When this option is set, at most $\mathcal{S}$ nodes use the altruistic approach to explore distances between neighbors in each sub-iteration. Each node is activated exactly once within the t sub-iterations, therefore this option has no impact of the final results which are equivalent to the ones with logical iterations only.

Phase 1 in Detail

Algorithm 6.1 shows the pseudocode of the ε-graph construction phase. For clarity and brevity, we consider graphs as distributed data structures, and describe the algorithm in terms of high-level graph operations such as "add_edge" or "remove_node". The algorithm is a series of steps, each introduced by the "for ... do in parallel" loop and separated by synchronization barriers; since each parallel for loop is logically executed syncronously, modifications to the graphs are visible only at the end of the loop. For details on how distributed graph-parallel computation is implemented through a BSP paradigm, we refer to work such as Pregel [124] or PowerGraph [75].

[2]See, e.g., `http://stackoverflow.com/a/16257628`.

Algorithm 6.1: Phase 1 – ε-graph construction.

```
1  $\varepsilon G \leftarrow$ new undirected, unweighted graph ;                // $\varepsilon$-graph
2  $NG \leftarrow$ random neighbor graph initialization;
3  for $i \leftarrow 1 \ldots iter$ do
       // Add reverse edges
4      for $n \in$ active nodes in $NG$ do in parallel
5          for $(n, u, w) \leftarrow NG.edges\_from(n)$ do
6              $NG$.add_edge$(u, n, w)$
7          end
8      end
       // Compute distances and update $\varepsilon G$
9      for $n \leftarrow$ active nodes in $NG$ do in parallel
10         $N \leftarrow$ at most $\rho k$ nodes from $NG$.neighbors$(n)$;
11         for $u \leftarrow N$ do
12             for $v \leftarrow N \setminus \{u\}$ do
13                 $w \leftarrow$ DISTANCE$(u, v)$;
14                 $NG$.add_edge$(u, v, w)$;
15                 if $w \leqslant \varepsilon$ then $\varepsilon G$.add_edge$(u, v)$;
16             end
17         end
18     end
       // Shrink $NG$
19     $\Delta \leftarrow 0$ ;                          // number of removed nodes
20     for $n \leftarrow$ active nodes in $NG$ do in parallel
21         if $|\varepsilon G.neighbors(n)| \geqslant M_{max}$ then
22             $NG$.remove_node$(n)$;
23             $\Delta \leftarrow \Delta + 1$
24         end
25     end
       // Termination condition
26     if $|NG.nodes| < \mathcal{T}_n \wedge \Delta < \mathcal{T}_r$ then break ;
       // Keep the $k$ closest neighbors in $NG$
27     for $n \in$ active nodes in $NG$ do in parallel
28         $l \leftarrow NG$.edges_from$(n)$;
29         remove from $l$ the $k$ edges with smallest weights;
30         for $(n, u, w) \leftarrow l$ do
31             $NG$.delete_edge$(n, u, w)$
32         end
33     end
34 end
35 return $\varepsilon G$
```

The algorithm uses the neighbor graph NG to drive the computation, and stores the final result in the ε-graph εG. After initializing NG by connecting each node to k random neighbors the main iteration starts. In lines 4–8, which correspond to step 1 of Figure 6.1 on page 97, we convert NG to an undirected graph by adding for each edge another one in the opposite direction. Lines 9–18 are the most expensive part of the algorithm, where each node computes the distances between each pair of neighbors; pairs of nodes at distance at most ε get added to εG as in step 2 of Figure 6.1. In lines 20–25, NG is shrunk by removing nodes having at least M_{max} neighbors in εG. The termination condition is checked at line 26, and if the computation continues the edges that do not correspond to the k closest neighbors found are removed from NG in lines 27–33, corresponding to step 3 of Figure 6.1.

Complexity Analysis

Unless the early termination condition is met, Phase 1 runs for a user-specified number of iterations. Since the number of nodes in the neighbor graph decreases with time, the first iterations are the most expensive (i.e., when a node is removed from the neighbor graph, it is never added again). Hence, we study the complexity of the first iteration, which has the highest cost since all nodes are present in the neighbor graph. Note that here we consider the cost of a logical iteration, corresponding to the sum of its sub-iterations (see Section 6.2.2) if parameter $\mathcal{S}$ is defined.

The loop of lines lines 4–8 requires m steps, where $m = kn$ is the number of edges in NG. Hence, it has complexity $\mathcal{O}(kn)$.

The loop of lines 9–18 computes distances between at most ρk neighbors of each node, where NG has at most $2kn$ edges, and each node has at least k neighbors. The worst case is when neighbor lists are distributed as unevenly as possible, that is when $n/(\rho - 1)$ nodes have ρk neighbors, and all the others only have k. In that case, $\mathcal{O}(n/\rho)$ nodes would compute $\mathcal{O}\left(\rho^2 k^2\right)$ comparisons, and $\mathcal{O}(n)$ nodes computing $\mathcal{O}(k^2)$ comparisons. The result is

$$\mathcal{O}\left(\frac{n}{\rho}\rho^2 k^2 + nk^2\right) = \mathcal{O}(\rho n k^2).$$

Since each distance computation can add one new edge to NG, the graph now has at most $\mathcal{O}(\rho n k^2)$ edges. The loops of lines 20–25, and lines 27–33, each in the worst case act on $\mathcal{O}(\rho n k^2)$ edges. The operations of line 29 can be implemented efficiently with a total cost of $\mathcal{O}(\rho n k^2 + nk \log k) = \mathcal{O}(\rho n k^2)$ with priority queue data structures such as binary heaps.

In conclusion, the total computational complexity for an iteration of Phase 1 is $\mathcal{O}(\rho n k^2)$. Note that, in general, ρ and k should take small values (the default values we suggest in Section 6.4.1 are $\rho = 3$ and $k = 10$), therefore the computation cost is dominated by n.

6.2.3 Phase 2: Discovering Dense Regions

As introduced in Section 6.2.1, Phase 2 outputs the clustering by taking as input the ε-graph, performing neighbor lookups on it instead of expensive ε-neighborhood queries. Realizing the analogies between density-reachability and connected components, we inspire our implementation on Cracker (Chapter 4).

We attribute node roles based on their properties in the ε-graph: nodes with at least $MinPts - 1$ neighbors are considered core;[3] between non-core nodes, those with core nodes as neighbors are considered border nodes, while others will be treated as noise. Noise nodes are immediately deactivated, and they will not contribute to the computation anymore.

[3]The $MinPts - 1$ value stems from the fact that, in the original dbscan implementation, a node itself counts when evaluating the cardinality of its ε-neighborhood.

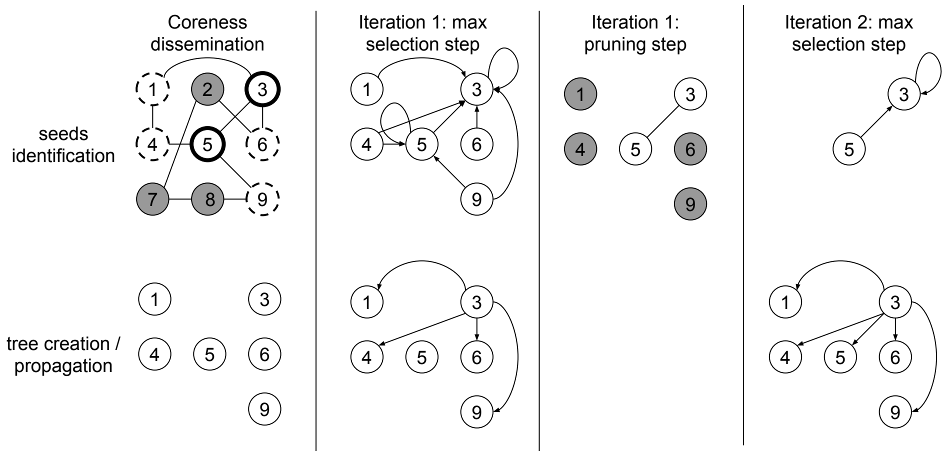

Coreness dissemination
Iteration 1: max selection step
Iteration 1: pruning step
Iteration 2: max selection step
seeds identification
tree creation / propagation
1
2
3
4
5
6
7
8
9

FIGURE 6.2: Phase 2 – dense region discovery.

Like several other algorithms for graph connectivity, our algorithm requires a total ordering between nodes, such that each cluster will be labeled with the smallest or largest node according to this ordering. A typical choice is an arbitrary node identifier; for performance reasons that we discuss in the following, we use the node with the largest degree instead and resort to the node identifier to break ties in favor of the smaller ID. In the following, we will refer to the (degree, nodeID) pair as *coreness*; as a result of the algorithm, each cluster will be tagged with the ID of the highest coreness node in its cluster. We will call *seed* of a cluster the node with the highest coreness.

Phase 2 is illustrated in Algorithm 6.2; the algorithm proceeds in three steps: after an initalization step called *coreness dissemination*, an iterative step called *seed identification* is performed until convergence. Clusters are finally built in the *seed propagation* step. We describe them in the following, with the help of the running example in Figure 6.2.

Coreness dissemination. In this step, each node sends a message with its coreness value to its neighbors in the ε-graph. For example. in Figure 6.2, nodes 3 and 5 have the highest coreness; 1, 4, 6 and 9 are border nodes, and the others are noise. We omit the pseudocode for brevity. Note that, although the following step modify the graph structure, coreness values are *immutable.*

Seed Identification. This step finds the seeds of all clusters, and builds a set of trees that we call *propagation forest* that ultimately link each core and border node to their seed. This step proceeds by alternating two sub-steps until convergence: *MinSelection* and *Pruning.* The ε-graph is iteratively simplified, until only seed nodes remain in it; at the end of this step, information to reconstruct clusters is encoded in the propagation forest.

With reference to Algorithm 6.2, in the MinSelection each node identifies the current neighbor with maximum coreness as its proposed seed (Line 9); each node will create a link between each of its neighbors – plus themselves – and the seed it proposes. Border nodes have a special behavior (Line 10)): they only propose a seed for themselves and their own proposed seed rather than for their whole neighborhood (Line 15)). In the first iteration of Figure 6.2, for example, node 4 – which is a border node – is responsible for creating edges $(4, 5)$ and $(5, 5)$. On the other hand, node 5 – which is a

Algorithm 6.2: Phase 2 – Discovering dense regions.

```
$G$ = Coreness_Dissemination($\varepsilon G$)
for $n \leftarrow$ nodes in $G$ do in parallel
    $n$.Active $\leftarrow$ True
end
$T \leftarrow$ empty graph                         // Propagation forest
// Seed Identification
while $|G.nodes| > 0$ do
    // Max Selection Step
    $H \leftarrow$ empty graph
    for $n \leftarrow G.nodes$ do in parallel
        $n_{max} \leftarrow$ maxCoreNode($G$.neighbors($n$) $\cup \{n\}$)
        if $n$ is not-core then
            $H$.add_edge($n, n_{max}$)
            $H$.add_edge($n_{max}, n_{max}$)
        else
            for $v \leftarrow G.neighbors(n) \cup \{n\}$ do
                $H$.add_edge($v, n_{max}$)
            end
        end
    end
    // Pruning Step
    $G \leftarrow$ empty graph
    for $n \leftarrow H.nodes$ do in parallel
        $n_{max} \leftarrow$ maxCoreNode($H.neighbors(n)$)
        if $n$ is not-core then
            $n$.Active $\leftarrow$ False
            $T$.add_edge($n_{max}, n$)
        else
            if $|H.neighbors(n)| > 1$ then
                for $v \leftarrow H.neighbors(n) \setminus \{n_{max}\}$ do
                    $G$.add_edge($v, n_{max}$)
                    $G$.add_edge($n_{max}, v$)
                end
            end
            if $n \notin H.neighbors(n)$ then
                $n$.Active $\leftarrow$ False
                $T$.add_edge($n_{max}, n$)
            end
            if IsSeed ($n$) then
                $n$.Active $\leftarrow$ False
            end
        end
    end
end
return Seed_Propagation($PropagationTree$)
```

core node – identifies 3 as a proposed seed, and creates edges $(4, 3)$, $(5, 3)$, and $(3, 3)$.

In the Pruning, starting in Line 19, nodes not proposed as seeds (i.e., those with no incoming edges) are deactivated (Line 33). An edge between deactivated nodes and their outgoing edge with highest coreness is created (Line 34). For example, in the first iteration of the algorithm, node 4 is deactivated and the $(4, 3)$ edge is created in the propagation forest.

Eventually, the seeds remain the only active nodes in the computation. Upon their deactivation, seed identification terminates and seed propagation is triggered.

Seed Propagation. The output of the seed identification step is the propagation forest: a directed acyclic graph where each node with zero out-degree is the seed of a cluster, and the root of a tree covering all nodes in the cluster. Clusters are generated by exploring these trees; the pseudocode of this phase is omitted for brevity.

6.3 Experimental Setup

We evaluate NG-DBSCAN through a comprehensive set of experiments, evaluating well-known measures of clustering quality on real and synthetic datasets, and comparing it to alternative approaches. In the following, we provide details about our experimental setup.

6.3.1 Experimental Platform

All the experiments have been conducted on a cluster running Ubuntu Linux consisting of 17 nodes (1 master and 16 slaves), each equipped with 12 GB of RAM, a 4-core CPU and a 1 Gbit interconnect. Both the implementation of our approach and the alternative algorithms we use for our comparative analysis use the Apache Spark [2] API.[4]

6.3.2 Evaluation Metrics

We now discuss the metrics we use to analyse the performance of our approach and the most important parameters of NG-DBSCAN. We also proceed with manual investigation of the clusters we obtain on some dataset, using domain knowledge to evaluate their quality.

We study the role of the parameters of our approach and the clustering quality using well-known measures of quality [56, 110]:

- **Compactness**: measures how closely related the items in a cluster are. We obtain the compactness by computing the average pairwise similarity among items in each cluster. Higher values are preferred.
- **Separation**: measures how well clusters are separate from each other. Separation is obtained by computing the average similarity between items in different clusters. Lower values are preferred.
- **Recall**: this metric relates two different data clusterings. Using clustering C as a reference, all node pairs that belong to the same cluster in C are generated. The recall of clustering D is the fraction of those pairs that are in the same cluster in D as well. In particular, we use

[4]Precisely, we use the Scala API and rely on advanced features such as RDD caching for efficiency reasons.

as a reference the exact clustering we obtain with the standard dbscan implementation of the SciKit library [144]. Higher values are preferred.

Note that computing the above metrics is computationally as hard as computing the clustering we intend to evaluate. For this reason, we resort to uniform sampling: instead of computing the all-to-all pairwise similarity between items, we pick items uniformly at random, with a sampling rate of 1%.[5]

Additionally, we also consider algorithm **Speed-Up**: this metric measures the algorithm runtime improvement when increasing the number of cores dedicated to the computation, using 4 cores (a single machine) as the baseline.

Our results are obtained by averaging 5 independent runs for each data point. In all plots we also show the standard deviation of the metrics used through error bars; we remark that in some cases, they are too small to be visible.

6.3.3 The Datasets

Next, we describe the datasets used in our experiments. We consider the following datasets:

- *Twitter Dataset.* We collected[6] 5 602 349 geotagged tweets sent in USA the week between 2012/02/15 and 2012/02/21. Each tweet is in JSON format. This dataset is used to evaluate NG-DBSCAN in two distinct cases: (i) using the latitude and longitude values to cluster tweets using the Euclidean distance metric, (ii) using the text field to cluster tweets according to the Jaro-Winkler metric [87].

- *Spam Dataset.* A subset of SPAM emails collected by Symantec Research Labs, between 2010-10-01 and 2012-01-02, which is composed by 3 886 371 email samples. Each item of the dataset is formatted in JSON and contains the common features of an email, such as: subject, sending date, geographical information, the bot-net used for the SPAM campaign as labeled by Symantec systems, and many more. For instance, a subject of an email in the dataset is “19.12.2011 Rolex For You -85%” and the sending day is “2011-12-19”.

In addition we also use synthetically generated input data using the SciKit library [144]. We generated three different types of input data called, respectively, circle, moon and blobs. These graphs are usually considered as a baseline for testing clustering algorithms in a d-dimensional space.

6.3.4 Alternative Approaches

We compare NG-DBSCAN to existing algorithms that produce data clustering. We use the following alternatives:

- dbscan: this approach uses the SciKit library dbscan implementation [144]. Clustering results obtained with this method can be thought

[5] We increase the sampling rate up to 10% for clusters with less than 10 000 elements.

[6] We implemented a simple crawler following the methodology described in [101]. Although Twitter ToS does not allow such data to be shared, it is rather simple to write such a crawler and obtain similar data.

of as our baseline, to which we compare NG-DBSCAN, in terms of clustering recall.

- SPARK-DBSCAN: this approach uses a parallel dbscan implementation for Apache Spark.[7] This work is an implementation of MR-DBSCAN. We treat this method as our direct competitor, and compare the runtime performance and clustering quality.
- IRVINGC-DBSCAN: This is another Spark implementation inspired by MR-DBSCAN.[8] With respect to SPARK-DBSCAN, this implementation is often faster but limited to 2D data.
- k-MEANS: we convert text to vectors using `word2vec` [5], and cluster those vectors using the k-MEANS implementation in Spark's MLLib library [3].

Because it is not a parallel algorithm, we do not include here comparisons to ρ-dbscan by Gan and Tao [73]. As it can be expected from an efficient single-machine algorithm [132], this algorithm is very efficient as long as its memory requirement fit into a single machine, since communication costs are lower by orders of magnitude. We remark that we obtained errors not allowing us to run ρ-dbscan on data points with more than 8 dimensions; Gan and Tao's own evaluation [73] considers only data points having maximum dimensionality 7.

6.4 Results

Through our experiments, we first study the role of NG-DBSCAN's parameters. Then, we evaluate clustering quality and the scalability with respect to SPARK-DBSCAN with 2D and n dimensional datasets. Finally, we study the ability of NG-DBSCAN to use arbitrary similarity metrics, by performing text clustering. Where not otherwise mentioned, we use Euclidean distance between items.

6.4.1 Analysis of the Parameter Space

NG-DBSCAN has the following parameters: *i)* $\mathcal{T}_n$ and $\mathcal{T}_r$, which regulate the termination mechanism; *ii)* k, the number of neighbors per node in the neighbor graph; *iii)* M_{max}, the threshold of neighbors in the ε-graph to remove nodes from the neighbor graph; *iv)* ρ, which limits the number of comparisons in extreme cases during Phase 1; *v)* $\mathcal{S}$, which limits the memory requirements by dividing logical iterations in several physical sub-iterations, with less nodes involved in the computation.

Termination Mechanism

We start our evaluation by analyzing the termination mechanism; we use here the Twitter dataset (latitude and longitude values). Figure 6.3 shows the number of active (Active) and removed (Removed_Tot) nodes, and the removal rate (Removed) in subsequent iterations of the NG-DBSCAN algorithm. To help understanding the analysis, we include in the Figure also

[7]`https://github.com/alitouka/spark_dbscan`
[8]`https://github.com/irvingc/dbscan-on-spark`

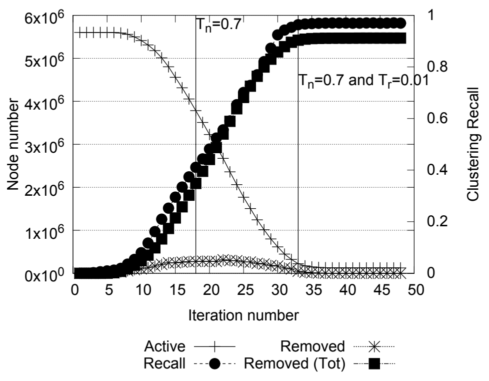


Figure 6.3: Analysis of the termination mechanism.

the clustering recall that we compute in every iteration of the algorithm. The results we present are obtained using $k = 10$ and $M_{max} = 20$; analogous results can be obtained with different configurations.

In the first 10 iterations, the number nodes in the neighbor graph remains roughly constant; this is the time required to start finding useful edges. Then, the number of active nodes rapidly decreases, indicating that a large fraction of the nodes reach convergence. Towards the last iterations, the number of active nodes reaches a plateau due to noise points.

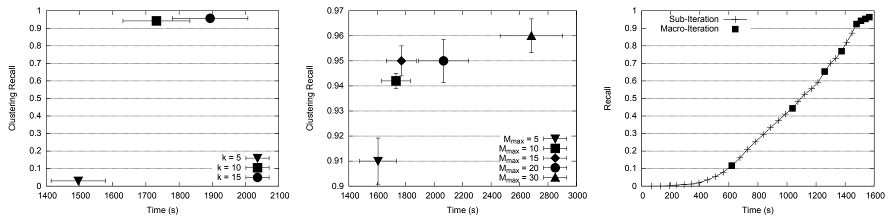
As discussed in Section 6.2.2, the $\mathcal{T}_n$ threshold, which indicates the number of active nodes required terminating the algorithm, avoids premature terminations that might occur if we only used the $\mathcal{T}_r$ threshold and the corresponding inequality. Instead, the $\mathcal{T}_r$ parameter, which measures the rate at which nodes are de-activated in subsequent iterations, avoids both premature terminations and lengthy and marginally beneficial convergence processes.

In particular, without the $\mathcal{T}_r$ threshold, the algorithm would stop at the $\mathcal{T}_n$ threshold, that is – in our experiment – at iteration 18. As the recall metric of roughly 0.5 indicates, stopping the algorithm too early results in poor performance. Instead, with both thresholds, the algorithm stops at iteration 33, where the recall is greater than 0.9. Subsequent iterations only marginally improve the recall.

How to Set k

We now consider the k parameter, which affects the number of neighbours in the neighbor graph, and perform clustering of the Twitter dataset (latitude and longitude values).

Figure 6.4a depicts the clustering recall we obtained with $k \in \{5, 10, 15\}$, as a function of the algorithm running time. Clearly $k = 5$ is not enough to obtain a good result, which confirms the findings of previous works on k-NN graphs [116, 60]. However, already with $k = 10$, the recall is considerably high, indicating that we retrieve approximately the same clusters as the exact dbscan algorithm. Increasing this parameter improves the quality of the result only marginally at the cost of a larger amount of algorithm runtimes. With a standard deviation lower than 1% on recall between different algorithm runs, the quality of results remains stable; running time has



(A) How to set k. (B) How to set M_{max}. (C) Analysis of sub-iterations and parameter $\mathcal{S}$.

FIGURE 6.4: Analysis of the Parameter Space.

a standard deviation of the order of 6%. Due to the above considerations we think that $k = 10$ is an acceptable configuration value.

How to Set M_{max}

We analyse the impact of the M_{max} parameter using the Twitter dataset, and set $k = 10$. Results with different values of k lead to analogous observations. Figure 6.4b shows the clustering recall achieved for values of $M_{max} \in \{5, 10, 15, 20, 30\}$, as a function of the algorithm running time.

The recall achieved by NG-DBSCAN is always larger than 0.9 when the algorithm terminates. Increasing M_{max} has positive effects on the recall: this confirms that a larger M_{max} improves the connectivity of the ε-graph in dense regions. However, there are "diminishing returns" when increasing M_{max}: a larger M_{max} value requires more time to meet the termination conditions because more edges must be collected at each node.

Overall, our empirical remarks indicate that M_{max} can be kept similar to k. In particular, values of $M_{max} \in [10, 20] = [k, 2k]$ give the better trade-offs between recall and completion time. Also in this case, the standard deviations in terms of recall and time are respectively smaller than 1% and 6%.

How to set ρ

The ρ parameter sets a limit to the number of nodes examined in the neighborhood of each node: if this is not done, in degenerate cases where nodes have a massive degree in the neighbor graph, the worst-case complexity of the first step of NG-DBSCAN could grow up to $\mathcal{O}\left(n^2\right)$. We have found, however, that our mechanism to remove nodes from the neighbor graph in practice already avoids the case in our experiments.

In Table 6.2 we show results for values of $\rho \in \{1, 2, 3, 6\}$. With a value of $\rho = 1$, the bound on the size of neighborhoods explored is too stringent, and NG-DBSCAN cannot explore new nodes quickly enough; as ρ grows, the algorithm performs better in terms of both recall and runtime.

We set a default value of $\rho = 3$ to ensure that algorithm terminates fast with good quality, while avoiding an increase in computational complexity for degenerate cases.

TABLE 6.2: How to set ρ.

ρ	1	2	3	6
Time (s)	3 985	2 303	2 233	2 241
Recall	0.089	0.95	0.944	0.951
Sub-Iterations	80	37	33	33

Sub-Iterations and $\mathcal{S}$

Figure 6.4c shows the amount of time to complete a sub-iteration and a macro-iteration as described in Section 6.2.2. As a reminder, the parameter $\mathcal{S}$ imposes a limit on the number of computing nodes in a given iteration. In this example run, we set $\mathcal{S} = 500\,000$ for the Twitter dataset of 5 602 349 tweets; this means that each sub-iteration involves approximately 500 000 nodes. The number of needed sub-iterations to complete the first macro-iteration should be $\lceil 5\,602\,349/500\,000 \rceil = 12$, but only 11 sub-iterations are actually necessary because some nodes already get deactivated in the first sub-iterations. As nodes get deactivated, macro-iterations become less and less expensive, requiring less sub-iterations and less time to complete. Since sub-iterations operate on approximately the same number of nodes, they keep a roughly constant size.

Parameters Discussion

We end this section discussing a set of parameters that we use as default, and give us a good trade-off between recall and completion time. $\mathcal{T}_n = 0.7$ and $\mathcal{T}_r = 0.01$ stop the algorithm when only very marginal benefit can be obtained by continuing processing; $k = 10$, $M_{max} = 2k$ and $\rho = 3$ yield a good trade-off between recall and run-time. In subsequent experiments we use the above configuration to evaluate NG-DBSCAN.

6.4.2 Performance in a 2D Space

We now move to a global evaluation of NG-DBSCAN, and compare the clustering quality we obtain to single-machine dbscan, and to the SPARK-DBSCAN and IRVINGC-DBSCAN alternatives. We use both synthetically generated datasets and the latitude and longitude values of the Twitter dataset.

Clustering Quality

We begin with the synthetically generated datasets (described in Section 6.3.3) because they are commonly used to compare clustering algorithms. Figure 6.5 presents the shape of the three datasets called respectively *Circle*, *Moon* and *Blobs*. Each dataset has 100 000 items to cluster: such a small input size allows computing data clustering using the *exact* SciKit dbscan implementation and to make a preliminary validation of our approach. Results are presented in Table 6.3. NG-DBSCAN obtains nearly perfect clustering recall for all the datasets, when compared to the exact dbscan implementation. The completion time of NG-DBSCAN, SPARK-DBSCAN and IRVINGC-DBSCAN are comparable in such small datasets. It is interesting to note that SPARK-DBSCAN and IRVINGC-DBSCAN perform comparably better in the Blob dataset, where partitioning can cover each

Table 6.3: Performance in a 2D space: Clustering Quality.

	NG-DBSCAN		SPARK-DBSCAN		IRVINGC-DBSCAN	
	Time (s)	Recall	Time (s)	Recall	Time (s)	Recall
Twitter	**1 822**	**0.951**	N/A	N/A	N/A	N/A
Circle	**96**	**1**	192	**1**	135	**1**
Moon	103	**1**	132	**1**	**72**	**1**
Blob	123	0.92	83	1	**61**	**1**

cluster in a different partition. Instead, in the circle and moon datasets, each cluster covers multiple partition and this slows down the algorithm.

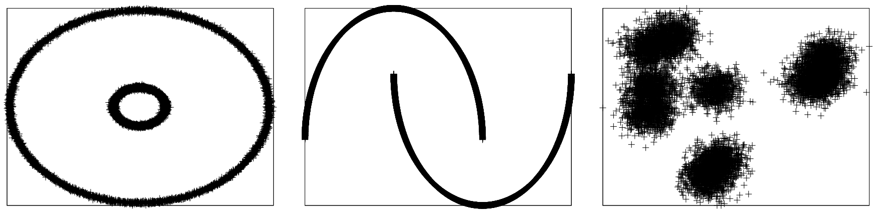
In the Twitter dataset, NG-DBSCAN is able to achieve a good clustering recall, as described also in previous Sections. Instead, SPARK-DBSCAN and IRVINGC-DBSCAN are not able to complete the computation due to memory errors. In the following we dive deeper in this respect, analyzing the impact of dataset size.

Scalability

We now compare the scalability of NG-DBSCAN to that of SPARK-DBSCAN and IRVINGC-DBSCAN. Figure 6.6a shows the algorithm runtime as a function of the dataset size, while using our entire compute cluster. We use 6 different samples of the Twitter dataset of size approximately 175 000, 350 000, 700 000, 1 400 000, 2 800 000 and 5 600 000 (i.e., the entire dataset) tweets respectively. For smaller datasets, up to roughly 1 400 000 samples, all three algorithms appear to scale roughly linearly, and IRVINGC-DBSCAN performs best. For larger datasets, instead, the algorithm runtime increases considerably. In general, we note that SPARK-DBSCAN is always slower than NG-DBSCAN, by a factor of at least of 1.74; SPARK-DBSCAN cannot complete the computation for the largest dataset, and with a size of 2 800 000 it is already 4.43 times slower than NG-DBSCAN. IRVINGC-DBSCAN cannot complete the computation due to memory errors on datasets larger than 1 400 000 elements.

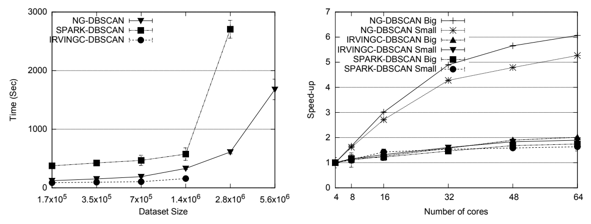
Figure 6.6b shows the algorithm speed-up of the three algorithms as the number of cores we devote to the computation varies between 4 and 64, considering a small dataset of 350 000 tweets and a larger dataset of 1 400 000 tweets. Our results indicate that NG-DBSCAN always outperforms SPARK-DBSCAN ans IRVINGC-DBSCAN, which cannot fully reap the benefits of more compute nodes: we explain this with the fact that adding new cores results in smaller partitions, which increase the communication cost. Past the cap of 32 cores, NG-DBSCAN's speedup grows more slowly, and doubling the compute cores does not double the speedup; we attribute this to the fact that communication costs start to dominate computation costs.

These results indicate that our approach is scalable – both as the dataset and cluster size grows. The time needed to compute our results with the configurations of Section 6.4.1 – which proved to be a desirable choice – is always in the order of minutes, demonstrating that our approach is viable in several concrete scenarios.



(A) Circle. (B) Moon. (C) Blobs.

FIGURE 6.5: Synthetic datasets plot.



(A) Scalability: Dataset Size. (B) Scalability: Number of Cores.

FIGURE 6.6: Performance in a 2D space: Scalability.

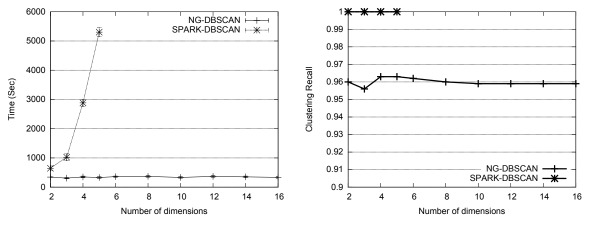
6.4.3 Performance in d-Dimensional Spaces

Next, we evaluate the impact of d-dimensional datasets in terms of clustering quality and algorithm running time. For our experiments, we synthetically generate 10 different datasets, respectively of dimensionality $d \in \{2, 3, 4, 5, 6, 8, 10, 12, 14, 16\}$ of approximately 1 500 000 elements each. The values in each dimension are a sample of the latitude and longitude values of the Twitter dataset. Unlike other approaches, NG-DBSCAN can scale to datasets having even higher dimensionality: we discuss in the following a case of dimensionality 1 000.

Figure 6.7a presents the running time of both NG-DBSCAN and SPARK-DBSCAN as a function of the dimensionality d of the dataset: we recall that IRVINGC-DBSCAN only allows 2-dimensional points as data. Results indicate that our approach is unaffected by the dimensionality of the dataset: algorithm runtime is roughly similar, independently of d. Instead, the running time of SPARK-DBSCAN significantly increases as the dimensionality grows: in particular, SPARK-DBSCAN does not complete for datasets in which $d \geqslant 6$. Even for small d, however, NG-DBSCAN significantly outperforms SPARK-DBSCAN.

Figure 6.7b shows the clustering recall as a function of d. Clustering quality is not affected by high dimensionality, albeit SPARK-DBSCAN does not complete for $d \geqslant 6$. The clustering recall of NG-DBSCAN settles at 0.96, due its approximate nature.

To evaluate NG-DBSCAN on even larger dimensionalities, we generate a dataset of 100 000 strings taken from the Twitter dataset, and use `word2vec` to embed them in a space having 1 000 dimensions. Even in this case, NG-DBSCAN achieves a recall of 0.96 with a running time of 640 seconds, which is comparable to what is obtained on datasets having lower dimensionality.



(A) Time. (B) Clustering Recall.

Figure 6.7: Performance in a d-dimensional space.

Table 6.4: Spam and Tweets dataset: manual investigation.

Dataset	Cluster size	Sample
Spam	101 547	"[. . .]@[. . .].com Rolex For You -36%" "[. . .]@[. . .].com Rolex.com For You -53%" "[. . .]@[. . .].com Rolex.com For You -13%"
Spam	42 315	"Refill Your Xanax No PreScript Needed!" "We have MaleSex Medications No PreScript Needed!" "Refill Your MaleSex Medications No PreScript Needed!"
Spam	83 841	"[. . .]@[. . .].com VIAGRA Official -26%" "[. . .]@[. . .].com VIAGRA Official -83%" "[. . .]@[. . .].com VIAGRA Official Site 57% 0FF."
Twitter	7 017	"I just ousted @hugoquinones as the mayor of Preparatoria #2 on @foursquare! http://t.co/y5a24YMn" "I just ousted Lisa T. as the mayor of FedEx Office Print & Ship Center on @foursquare! http://t.co/cNUjL2L5" "I just ousted @sombrerogood as the mayor of Bus Stop #61013 on @foursquare! http://t.co/SwC3p33w"
Twitter	1 033	"#IGoToASchool where your smarter than the teachers !" "#IGoToASchool where guys don't shower . They just drown themselves in axe ." "#IGoToASchool where if u seen wit a female every other female think yall go together"
Twitter	23 884	"I'm at Walmart Supercenter (2501 Walton Blvd, Warsaw) http://t.co/4Mju6hCd" "I'm at The Spa At Griffin Gate (Lexington) http://t.co/Jb5JU8bT" "I'm at My Bed (Chicago, Illinois) http://t.co/n9UHV2UK"

In conclusion, NG-DBSCAN performs well irrespectively of the dimensionality of the datasets both in terms of runtime and clustering quality. This is a distinguishing feature of our approach, and is in stark contrast with respect to algorithms constructed to partition the data space, such as SPARK-DBSCAN and the majority of the state of the art approaches (see Table 6.1), for which the runtime worsens exponentially with the dataset dimensionality.

6.4.4 Performance with Text Data

We conclude our analysis of NG-DBSCAN by evaluating its effectiveness when using arbitrary similarity measures. In particular, we perform the evaluation using text data by means of two datasets: the textual values of the Twitter dataset, and a collection of spam email subjects collected by Symantec. As distance metric, we use the Jaro-Winkler edit distance.

Comparison with k-means

Since alternative dbscan implementations do not support Jaro-Winkler distance (or any other kind of edit distance), we compare our results with those obtained using k-means on text data converted into vectors using `word2vec` using the default dimensionality of 100, as described in Section 6.3.4. To proceed with a fair comparison, we first run NG-DBSCAN and use the number of clusters output by our approach to set the parameter K of k-means. We recall that other dbscan implementations are not viable in this case, since neither a string data type nor the large dimensionality of `word2vec` vectors can be handled by them.

TABLE 6.5: Evaluation using text data: Twitter and Spam datasets comparison with *k*-MEANS. "C" stands for compactness and "S" for separation.

Algorithm	Twitter			Spam			Spam 25%		
	C	S	Time	C	S	Time	C	S	Time
NG-DBSCAN	**0.65**	**0.2**	**2 980**	**0.88**	**0.63**	**4 178**	**0.88**	**0.66**	**654**
k-MEANS	0.64	0.42	4 477	N/A	N/A	N/A	0.84	0.67	27 557

TABLE 6.6: Distance function comparison for Twitter.

distance	#clusters	max size	C	S	Time
Jaro-Winkler	**1 605**	**58 973**	**0.65**	**0.2**	2 980
`word2vec` + cosine	3 238	24 117	0.64	0.29	**2 908**

We begin with a manual inspection of the clusters returned by NG-DBSCAN: results are shown in Table 6.4. We report 3 clusters for each dataset, along with a sample of the clustered data. Note that subjects or tweets are all related, albeit not identical. Clusters, in particular in case of the Spam dataset, are quite big. This is of paramount importance because specialists usually prefer to analyse large clusters with respect to small clusters. For instance, we obtain a cluster of 42 315 emails related to selling medicines without prescription, and a cluster of 23 884 tweets aggregating text data of people communicating where they are through Foursquare.

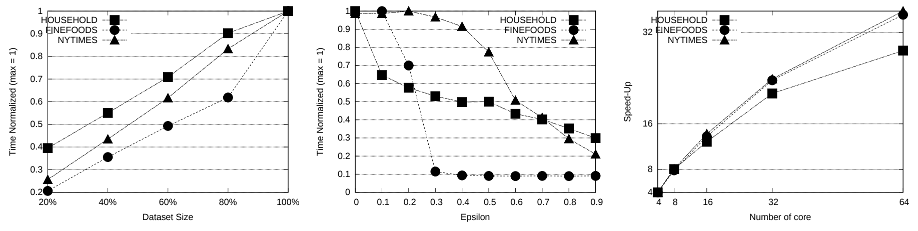
Next, we compare NG-DBSCAN with *k*-MEANS using the well-known internal clustering validation metrics we introduced in Section 6.3.2, basing them on Jaro-Winkler edit distance. Recall that compactness (C) measures how closely related the items in a cluster are, whereas separation (S) measures how well clusters are separated from each other. We perform several experiments with both Twitter and Spam datasets: Table 6.5 summarizes our results.

For what concerns compactness, higher values are better and both NG-DBSCAN and *k*-MEANS behave similarly. However, in the full Spam dataset, we are unable to complete the computation of *k*-MEANS: indeed, the *k*-MEANS running time is highly affected by its parameter K. In this scenario we have $K = 17\,704$ and the *k*-MEANS computation does not terminate after more than 10 hours. Hence, we down-sample the Spam dataset to 25% of its original size (we have the very same issues with a sample size of the 50%). With such a reduced dataset, we obtain $K = 3\,375$ and *k*-MEANS manages to complete, although its running time is considerably longer than that of NG-DBSCAN. The quality of the clusters produced by the two algorithms are very similar.

For the separation metric, where lower values are better, NG-DBSCAN clearly outperforms *k*-MEANS. In particular in the Twitter dataset we achieve 0.2 instead of 0.42 suggesting that the clusters are more separated in NG-DBSCAN with respect to *k*-MEANS.

Impact of Text Embedding

NG-DBSCAN offers the peculiar feature of allowing arbitrary data and distance functions: we used it in the previous experiment to show that our algorithm, running directly on the original data, can perform better than



(A) Dataset sizes. (B) ε values. (C) Number of core.

FIGURE 6.8: Evaluation of the scalability.

existing algorithms which embed strings in vectors on which the clustering algorithm is run. Here, we perform an experiment aimed at evaluating this feature, comparing NG-DBSCAN running on raw text, using Jaro-Winkler distance, against the same algorithm running on the vectors obtained through the `word2vec` embedding.

Table 6.6 presents the results on the Twitter dataset. They indicate that, indeed, transforming text to a vector representation induces a clustering quality loss, when quality is defined using compactness and separation according to the Jaro-Winkler distance measure: the cluster separation is worse, and clusters are more fragmented (i.e., more clusters of smaller size) when NG-DBSCAN uses the traditional `word2vec` embedding. This result emphasizes a key feature of NG-DBSCAN: it allows working with arbitrary data; the opportunity of tailoring distance metrics to the data allow obtaining, as a result, clusters with better quality.

6.4.5 Scalability

We now perform an evaluation of the scalability of NG-DBSCAN. We show how different dataset sizes, ε values and number of cores affect the execution time of the algorithm. To this end, we choose three datasets from the UCI Machine Learning web site:

- *NYTIMES* [11]. The dataset has been constructed removing stopwords and the vocabulary of unique words was truncated by only keeping words that occurred more than ten times. The cosine similarity has been chosen to compare two documents.
- *HOUSEHOLD* [11]. It is a 7-dimensional dataset with cardinality 2 049 280 which includes all the attributes of the Household database from the UCI archive [11] except the temporal columns date and time. This scenario is the very same of the one tested by Gan et al.[73]. We also make use of the Euclidean distance to compare data.
- *FINEFOODS* [128]. This dataset consists of reviews of fine foods from Amazon. The data span a period of more than 10 years, including all 500 000 reviews up to October 2012. Reviews include product and user information, ratings, and a plaintext review. We make use of the plaintext to aggregate the reviews sharing a similar content. We compare such data with the JaroWinkler [87, 183] similarity metric.

The three datasets span different typologies of data and distance metrics in order to evaluate how different kind of datasets behaves in such scenarios.

Dataset sizes The first scalability evaluation regards the impact of the dataset size on the execution time. Figure 6.8a shows the execution time on the Y axis and the dataset size on the X axis. We sampled the following different sizes of the original dataset $\{20\%, 40\%, 60\%, 80\%, 100\%\}$. As expected, the full dataset size is the one requiring the larger execution time. We normalized the other values on the Y axis setting such max value equals to 1. The HOUSEHOLD dataset seems the one less affected by the dataset size. On the other hand, in NYTIMES dataset, the execution time linearly increases with the size of the dataset. Instead, FINEFOODS is working better when the dataset size is small and undergoes more with respect to the other datasets when increasing the dataset size. All the results are the average of 5 independent runs and we obtain always a standard deviation less than 1%.

ε **values** Here, we provide an evaluation of the running time with respect to different ε values. Figure 6.8b presents the results for the three considered datasets. We normalized the values setting the value 1 to the longest execution time. When ε is high NG-DBSCAN needs only few iterations to complete the first phase because all the nodes find neighbors valid according to the threshold. This results in shorter running time, for instance, FINEFOODS completes 10 times faster with $\varepsilon \geq 0.3$ with respect to $\varepsilon = 0.1$. As expected decreasing ε requires longer running time.

Number of cores Finally, we evaluate the speed-up of NG-DBSCAN using the three datasets. Figure 6.8c shows the algorithm speed-up as the number of cores varies between 4 and 64. Past the cap of 32 cores, NG-DBSCAN's speedup grows more slowly, and doubling the compute cores does not double the speedup; we attribute this to the fact that communication costs start to dominate computation costs. The datasets behave similarly, however HOUSEHOLD, that considers real points in a ND-space, is less scalable. We believe that the motivation is the different metric used in such dataset. The Euclidean distance is the metric requiring less time to compute with respect to the JaroWinkler and Cosine. In fact, HOUSEHOLD requires a lower execution time with respect to the other datasets though is the largest dataset. This suggests that the speed-up of NG-DBSCAN could be more evident when the size of the dataset is larger.

Considerations These results indicate that our approach is scalable – both as the dataset and cluster size grows. The time needed to compute our results is always in the order of minutes, demonstrating that our approach is viable in several concrete scenarios.

6.4.6 Discussion

We have provided a set of NG-DBSCAN parameters that consistently result in a desireable trade-off between speed and quality of the results (Section 6.4.1); we have found that, using these parameters, NG-DBSCAN scales better than other dbscan distributed implementations (Section 6.4.2); its qualities shine in datasets having large and very large dimensionalities (Section 6.4.3). In Section 6.4.4, we have seen that the ability of working with arbitrary data and using custom distance functions can enable higher-quality clustering than in existing approaches.

We summarize our experimental findings by concluding that NG-DBSCAN allows performing density-based clustering, approximating with high fidelity the well-known dbscan algorithm, even in the case of big and high-dimensional or arbitrary data, which was not handled satisfactorily by existing dbscan implementations.

6.5 Conclusion

Data clustering and analysis is a fundamental task in data mining and exploration. However, the need to analyse unprecedented large amounts of data require novel approaches to algorithm design, often calling for parallel frameworks that support flexible programming models, while operating on large scale clusters.

We presented NG-DBSCAN, a novel distributed algorithm for density-based clustering that produces quality clusters with arbitrary distance measures. This is of paramount importance because it allows versatility and *separation of concerns*: domain experts can chose the similarity function that is most appropriate for their data, given their knowledge of the context; instead, the burden of parallelism can be addressed by designers who are more familiar with framework APIs than with the peculiar data at hand.

We showed, through a detailed experimental campaign, that our approximate algorithm is on-par with the original dbscan algorithm, in terms of clustering results, for d-dimensional data. However, NG-DBSCAN scales to very large datasets, outperforming alternative designs.

Finally, the simplification technique of NG-DBSCAN permits to improve the performance of the algorithm without affecting the quality of the results.

Part III

Applications

Chapter 7

Spam campaign analysis

In this chapter we show how our algorithm presented in Chapter 5 can be exploited to solve a specific application. In this application, we focus on a particular data clustering task, which involves grouping text data items. The application domain of our work stems from the objective of identifying SPAM campaigns: for instance, we focus on data collected by Symantec Research Labs, that perform root-cause analysis of large scale SPAM email campaigns originated from bot networks. In this *adversarial* context, data clustering is even more challenging, because spammers manipulate text to avoid SPAM emails being identified as originating from the same campaign. As a consequence, the similarity metrics used for clustering must cope with text mangling, which require *non-metric* distances that disregard typos, character swapping, and other techniques to avoid detection.

In summary, the contributions of this application are as follows:

- we show how two algorithms (k-NN and connected components) can be combined to effectively solve an important application, in particular we make use of our proposed solution presented in Chapter 5;
- we use a real-life dataset and evaluate the overall clustering quality of our approach using both traditional metrics, and with the help of domain experts through manual investigation, highlighting the interpretability of clustering results.

The outcome of this application has been published in IEEE Conference on Big Data with the title *"Scalable k-NN based text clustering"* [117].

7.1 Our approach

We now present our approach for Spam Campaign detection. The idea is to group emails sharing similar email's subject. To this end, we perform text clustering, which is based on the scalable algorithm presented in Chapter 5. The problem of such text clustering is particularly challenging due to the application scenario we study. We face an adversarial setting in which text data is generated such that finding similar items is cumbersome: SPAM campaigns introduce text mangling, spelling errors and generally variations on some baseline text which makes SPAM items belonging to the same campaign appear different one from each other. As a consequence, we need to use a similarity metric between items that can overcome, or at least mitigate, the problem.

Similarity Metric. There exist numerous similarity metrics in the vast literature on the subject of this work. In particular, for text data, the Hamming distance and Levenshtein distance have been extensively used to determine

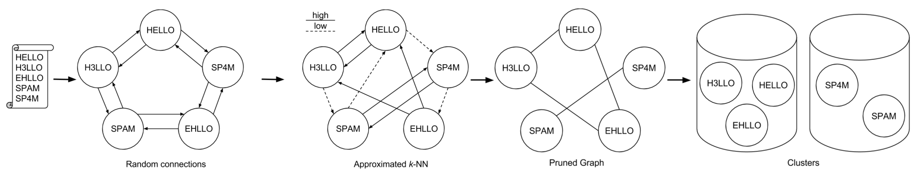


FIGURE 7.1: Illustration of our approach

TABLE 7.1: Symantec Dataset: characteristics

Feature	**Unique**	**Value 1**		**Value 2**		**Value 3**		**Value 4**		**Value 5**	
bot	11669	Lethic	20.41%	Unclassified	18.63%	Bagle	11.24%	Cutwail	9.36%	Grum	9.20%
city	33905		16.22%	Seoul	3.61%	Kiev	2.16%	Hanoi	1.76%	Moscow	1.70%
country	224	Russian Federation	10.46%	India	8.53%	Brazil	5.83%	Korea, Republic of	5.73%	Ukraine	4.62%
day	393	10/31/2010	0.31%	10/30/2010	0.30%	10/23/2010	0.30%	2/19/2011	0.30%	11/3/2010	0.29%
fromDomain	241117	domain555065.com	11.10%	domain359761.com	2.22%	domain572911.com	0.90%	domain425436.com	0.86%	domain384117.net	0.56%
host	32915		42.09%	airtelbroadband.in	1.71%	ukrtel.net	1.67%	localhost	1.62%	hinet.net	1.40%
ip	2227174	anonymous_IP_1	0.04%	anonymous_IP_2	0.03%	anonymous_IP_3	0.03%	anonymous_IP_4	0.02%	anonymous_IP_5	0.02%
rcptDomain	8641	domain555065.com	81.66%	domain806676.fr	3.27%	domain946987.org	2.18%	domain240360.br	1.93%	domain801669.com	1.64%

the similarity among text items. In this application, for the reasons illustrated above, we choose the Jaro-Winkler [87, 183] similarity metric which, simply stated, counts the common characters between two strings even if they are misplaced, misspelled, and mangled by a "short" distance. Note that, the Jaro-Winkler metric has a codomain in $[-1, 1]$.

Illustrative example. We provide an overview of our analytical process to detect spam campaign, and proceed with an illustrative example. Our text clustering approach works in two phases. In the first phase, it builds an approximate k-NN graph of text items. The first phase concludes with a pruning stage, which strives at eliminating spurious links between items with low similarity. In the second phase, we use a parallel approach to identify connected components in the k-NN graph, which are a proxy for clusters of similar items.

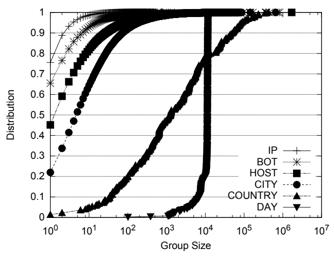
Figure 7.1 illustrates the process, where we consider 5 SPAM email subjects: note the swap of letters and typos typical of SPAM emails. Each email subject corresponds to a node in the intermediate graph structure our approach builds.

Initially, each node connects to $k = 2$ randomly chosen nodes. First, our algorithm *iteratively* builds an approximate 2-NN graph where edges having low similarity are dashed: the number of iterations constitutes one parameter of our approach. At this point, the pruning phase eliminates edges between nodes that have a low similarity measure, based on a threshold that constitutes the second parameter of our algorithm. Finally, using the pruned 2-NN graph, our method finds its connected components, which we use as a proxy of the clusters our method identifies.

7.2 Data Description

In the following of this section we make a deeper analysis and description about the SPAM dataset used to validate the application.

The dataset. The main dataset we use in our evaluation consists of a subset of SPAM emails collected by Symantec Research Labs, between 2010-10-01 and 2012-01-02, which is composed by $3,886,371$ email samples. Each item of the dataset is formatted according to JSON and contains the common features of an email, such as: subject, sending date, geographical information, the bot-net used for the SPAM campaign as labeled by Symantec



Feature	Similarity
bot	0.6
city	0.49
country	0.49
day	0.5
fromDomain	0.54
host	0.5
ip	0.51
rcptDomain	0.5

FIGURE 7.2: Symantec dataset: feature distribution and average similarity

systems, and many more. For instance, a subject of an email in the dataset is “19.12.2011 Rolex For You -85%” and the sending day is “2011-12-19”.

In this work, we are interested in identifying clusters of SPAM emails using subjects alone, as they constitute a compact description of the email. Next, we provide an overview of the dataset we use, to gain a better understanding of its characteristics; we also proceed with a naïve approach to clustering emails, by grouping them according to some of their fields.

Table 7.1 illustrates such a preliminary analysis: the “Unique” column identifies the number of distinct values for each feature we use, whereas additional columns in the table indicate the top individual values for each feature. For instance, “grouping by” the feature “bot” indicates that there are 11,669 unique bot-nets in the dataset, with “Lethic” taking roughly 20% of the emails, followed by 18% of “Unclassified” bot-nets and 11% from the “Bagle” bot-net. The dataset contains emails sent from 224 different countries, as shown when “grouping by” the feature “country”.

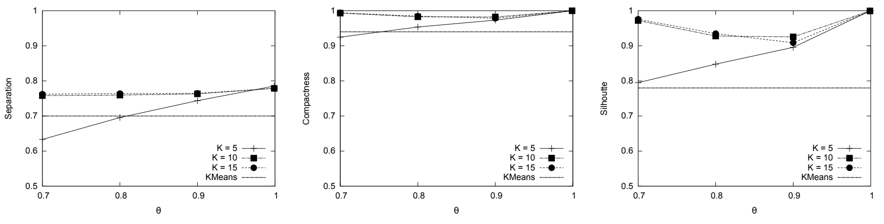
Figure 7.2 left side describes how the size of each group is distributed in the dataset. For example, 90% of the groups having the same bot-net value have a size lower than 10 emails, indicating skewness when considering the “bot” feature. Instead, groups having the same “day” feature are more uniformly distributed: only 10% of the days have less than 10^4 emails, while the remaining 90% of the days have similar group size around the value 10^4.

Finally, we verify if grouping emails according to the features of Table 7.1 results in email having similar subjects: in other words, we are interested in understanding if using similar subjects to cluster emails would boil down to simply grouping them by some other features. The right side of Figure 7.2 pinpoints at a negative answer: essentially, grouping by any of such features results in email subjects being very loosely similar, which is not sufficient to consider such groups a useful proxy for email clusters.

7.3 Results

We already presented in Section 5.4 results showing the characteristics of our clustering algorithm, the impact of the different parameters and how to configure it. In this section we present our result analysing the Symantec dataset and the outcomes of the spam campaign detection.

We compare clustering quality to a **baseline** algorithm. In particular, we use the efficient K-means implementation available in Spark’s MLLib package [3], and the `word2vec` package [5]. It is important to note that the parameter K, in K-means is substantially different from the parameter k of



(A) Separation (lower is better). (B) Compactness (higher is better). (C) Silhouette (higher is better).

FIGURE 7.3: Symantec Dataset: Clustering quality in terms of inter and intra cluster similarity, and clustering Silhouette. The algorithm proposed in this work outperforms the baseline method, and is only marginally affected by approximation quality.

TABLE 7.2: Symantec Dataset: Manual Investigation

cluster size	bot	days	subjects sample
7255	Grum, Unclassified	2011/12/14-2011/12/20	”17.12.2011 Rolex For You -73%” ”15.12.2011 Rolex For You -89%” ”19.12.2011 Rolex For You -85%”
4512	Rustock, Unclassified	2010/12/04-2010/12/06	”jadevnn, Alena (status-online) invites you for chat.” ”Hi zmes40, Alena (status-online) invites you for chat.” ”keumd,,Alena (status-online) invites you for chat.”
4412	Rustock, Unclassified	2011/01/28-2011/02/01	”Re: User kilmernn” ”Re: User anguinet” ”Re: User hudnalli”
4116	Rustock, Unclassified	2011/03/12-2011/03/14	”tdwilkey, you have a new PRIVATE MESSAGE”, ”dbeltondd, you have a new PRIVATE MESSAGE” ”bn, you have a new PRIVATE MESSAGE”
2992	Grum, Unclassified	2011/08/19-2011/08/23	”cseeberd@Amega.com VIAGRA ? 84% consensus!” ”Maia@Amega.com VIAGRA ? 50% consensus!” ”zelmo38dd@Amega.com VIAGRA ? 16% consensus!”

the k-NN graph algorithm: it indicates the number of clusters the K-means algorithm is set to produce. If not otherwise specified, we set $K = 1000$ such that the baseline algorithm output 1,000 clusters. This configuration yields the best result in term of Silhouette metric.

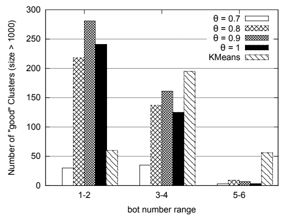
In what follows and if not otherwise specified, we set the operating parameters of our algorithm as follows: $k = 10$, and 10 *iterations*, which are the parameters that offer a good trade-off between clustering quality, approximation quality, and algorithm runtime.

Figure 7.3 shows the three main metrics we use to judge clustering quality, namely *separation*, *compactness* and *Silhouette*, as a function of θ, and for various values of k. Such metrics are computed both for our approach, and for the *baseline* algorithm based on K-means.

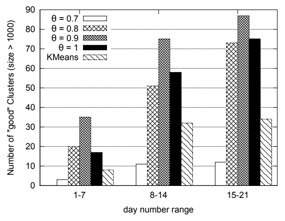
The separation metric evaluates how “far apart” the clusters output by the algorithms are: lower values of separation indicate that the inter-cluster distance is large, which is a desirable property to distinguish clusters well. As shown in Figure 7.3a, both our method and the baseline algorithm achieve good separation, with a slight advantage for the baseline method, that produces clusters that are more pairwise dissimilar.[1]

On the other hand, the compactness metric indicates how similar are the items within a cluster: larger values of compactness are desirable, because they are indicative of the absence of outliers that could “pollute” the quality of individual clusters with unrelated items. Figure 7.3b indicates that our approach is superior to the baseline method with respect to this metric, the

[1]An “artifact” due to the distance metric used in K-means, which separates text items even if they differ because of mangling.



(A) Bot network identifiers.



(B) Days.

FIGURE 7.4: Clustering quality in terms of the unique number of features in each cluster output by clustering algorithms. For manual inspection to be useful, a small number of unique feature, such as bot-nets or time-frame of each cluster item, is preferred. Our approach outperforms the baseline algorithm.

latter producing clusters with emails that are unrelated to the majority of other items in a cluster.

Figure 7.3c illustrates that the clustering Silhouette obtained by our approach is superior to the baseline algorithm, and this holds for all parameter choices. This is confirmed also in Figures 7.4a and 7.4b, which show the number of "good" clusters (with at least 1,000 items) as determined by domain knowledge metrics. Essentially, these figures report the number of clusters amenable to manual inspection of the results, as a function of features such as the number of bot-nets and the time-frame of a SPAM campaign. For example, in Figure 7.4a, domain experts can extract valuable information when the number of SPAM bots in a cluster is small, in the 1-2 range: in this case, our approach is superior to the baseline algorithm, which performs slightly better for the less interesting cases of 3-4 and 5-6 bots. Similarly, Figure 7.4b shows that the number of "good" clusters identified by our approach is always better than that of the baseline algorithm, and this is especially true for the 1-7 range, indicating cluster with emails spanning a 1 week time-frame. In summary, we find that the algorithm we present in this work performs well, especially when selecting an appropriate operating point, with $\theta \in [0.8, 0.9]$.

Finally, we proceed with a manual inspection of the clusters we obtain with our approach, to further illustrate the "goodness" of the clustering we achieve, with $k = 10$, 5 *iterations* and $\theta = 0.9$. Table 7.2 illustrates a few email samples in clusters where both the number of bot-nets is less or equal to 2, and all emails are all sent within one week time-frame. For instance, we obtain a cluster of 7255 emails sent from the Grum bot-net, between 2011/12/14 and 2011/12/20: the subjects of the email are related to a SPAM campaign involving a Rolex discount. Note that subjects are all related, albeit not identical.

7.4 Conclusion

In this chapter we presented how is possible to identify spam campaigns starting from emails' subjects. We showed how our clustering algorithm presented in Chapter 5 can be successfully exploited for this domain. In particular we validate the versatility of the approach and the quality of result although approximation. We supported our claims using real traces covering adversarial applications aimed at identifying SPAM campaigns, and through manual inspection by domain experts of the clusters output by our algorithm.

Chapter 8

Population estimation

This application stems for estimating population making use of clustering algorithms and using mobile phone data. This application is not just the mere execution of the clustering algorithm presented in Chapter 5 on a particular dataset, in fact it introduces many challenges:

- validate the claim that the algorithm is able to work with arbitrary data;
- construct a begin-to-end application where data needs to be initially processed and elaborated to be suitable for a clustering algorithm, until the final analysis of the clusters;
- improve the algorithm in particular to permit its usage in a completely unsupervised manner;
- study indicators about individuals movements such as the flows between home and work locations and from home to visit places.

This application has been originally presented in the IEEE Symposium on Computers and Communication with the title *"Improving Population Estimation From Mobile Calls: a Clustering Approach"* [119] and an extension is currently submitted to the journal “Data Mining and Knowledge Discovery”.

8.1 Related Work

Several studies use mobile phone data driven by their large market penetration in recent years. In fact, many works study all the possible social and economic indicators that can be extracted by such data. In this Section we discuss works related to ours about mobile calls data analysis (Section 8.1.1) and about scalable clustering algorithms (Section 8.1.2).

8.1.1 Mobile phones data analysis

Mobile phone traces have been utilized to monitor the traffic in cities and analyse tourist movements. In particular two popular works focus on this issue for the cities of Rome [32] and Graz [152]. From these works, many others, for instance Ahas *et al.* [9], analyse that is possible to individuate which are the places visited by the individuals analysing the calls performed. In addition, a plethora of works, for instance the winner of the Nokia Mobile Data Challenge [64], build predictors able to determine the next position of an individual given the current context.

De Jonge *et al.* [50] study different approaches making use of two weeks calls in Netherlands. They give insights on the indicators obtainable analysing the phone calls. For instance, such data can be used to estimate the level of the economic activity because the number of phone calls can be an indicator of the economic activity of a certain region. They make use of the KMeans clustering algorithm to determine day pattern clusters of the call activity. However, they suggest that a deeper study on the calling behaviour should be performed on a larger dataset covering multiple weeks to correctly estimate population density.

One of the first works using mobile data to estimate the population has been presented by Terada *et al.* [168]. In this work they monitor the presence of mobile terminals present in each base station area in different time intervals. Such data is refined with census information and at the end the per-cell populations are aggregated in grid sections or municipalities. This result may be affected by errors. Checking only the presence in a cell can not detect if an individual is a resident, who should be counted as living in the area, or just a visitor. Due to this, sub sequent works try to exploit mobile data in a different manner.

Deville *et al.* [57] improves the ideas of De Jonge and exploit mobile phone data for estimating population density. They propose a framework called MP. According to such methodology, population density is estimated as a function of the night-time phone calls occurring in a given area. However, a simple rule-based approach to identify user presence may hinder to derive some more useful information about the calling behaviour of the users. For instance, it would be cumbersome to define rules able to characterize individuals that are Commuters or Visitors.To overcome the aforementioned limitations, in a seminal work Furletti *et al.* [71] defined how to build individual profiles based on mobile phone calls. Such profiles characterize the calling behaviour of a user, in different time slots. By analysing these profiles, it is possible to identify three categories of users: Residents, Commuters or Visitors. Sociometer [72] focuses on this characterization to aggregate users having a similar calling behaviour with the k-MEANS clustering algorithm. The centroid of each cluster is compared with pre-defined archetypes representing the categories of interest, then, each cluster is classified by means of the associated archetype. Hereafter we use the term *exemplar* to refer to the cluster's centroid.

Our work Muchness+ advances the achievements of Sociometer in the following areas: *(i)* it provides a scalable distributed approach which can process a sensibly larger collection of data requiring no input from the user; *(ii)* it defines a personalized similarity metric that leads to better clustering results and is able to cluster different kind of data relative to different mobile calls aggregation strategy; *(iii)* it automatically removes outliers to improve the overall quality and to provide a better estimation of the population; *(iv)* it does not require to provide in advance the number of clusters as in k-MEANS; *(v)* it studies the indicators that can be extracted from mobile calls such as how individuals move to reach their working location. Table 8.1 shows the main differences between our work and the related works described in this section.

8.1.2 Scalable clustering algorithms

Mobile data are usually of large size. In this work, we analyse the phone calls performed daily in the Italian region of Tuscany. Due to this, when we need to cluster such amount of data a scalable clustering algorithm is of paramount importance. In this Section we cover several scalable clustering algorithms related to ours.

One of the most popular clustering algorithm is k-MEANS which aggregates data around K centroids. It has three main limitations: the K parameter has to be user-provided, it is limited to euclidean spaces, it has a bias on the initial selection of centroids. Moreover, despite parallel and distributed implementations of k-MEANS exist, they suffer of longer running time when K is large due to the large number of comparisons.

Another interesting class of clustering algorithm falls in the dbscan family, defined by Ester *et al.* [63]. The underpinning idea is to cluster items that have at least MINPTS neighbours at maximum distance ε. The main advantages against k-MEANS are the following: *(i)* it is not required to know the number of clusters in advantage; *(ii)* the ability to cluster items with complex shapes instead of aggregating items that are simply close (according to the euclidean distance) to a centroid. MR-dbscan [80] has been the first proposal targeting a distributed implementation of dbscan, realized as a 4-stage MapReduce algorithm. This approach focuses on the definition of an efficient data partitioning in a d-dimensional Euclidean space, where each partition is assigned to a worker node. This solution is limited, similarly to k-MEANS, to work on euclidean spaces.

In Chapter 5 we defined a distributed clustering algorithm based on nearest neighbour graphs able to deal with arbitrary similarity metrics has been proposed. This is at the basis of the approach of Muchness. Such approach have several drawbacks that we claim to overcome in this application. Albeit it is not necessary to define the number of clusters in advantage, it requires three parameters to tune the quality and the execution time of the algorithm. Due to this, in this chapter we extend such algorithm in order to remove the requirement of input from the user. This will facilitate its usage and target a good trade-off between clustering quality and execution time.

8.2 Data description

Telco operators collect customer data for billing purposes. Refer to Figure 8.1 to have an overview of how data are created, collected and aggregated. From one Telco operator in Italy, we received anonymized data of calls performed in Tuscany (Italy) recorded during the period between February and March 2014. Each call record is a tuple having the anonymous identifier of the user, the call timestamps and the cell id. We manage approximately 60×10^6 calls (column Telco data in Fig. 8.1). Each cell id can be assigned

TABLE 8.1: Overview of frameworks to estimate population

Name	Method	Clustering configurable	Residents	Commuters
MP [57]	rules on each data	N/A	yes	no
Sociometer [72]	clustering k-MEANS	number of clusters	yes	yes
Muchness	clustering k-NN based	3 main parameters	yes	yes
Muchness+	**clustering k-NN based**	**completely unsupervised**	**yes**	**yes**

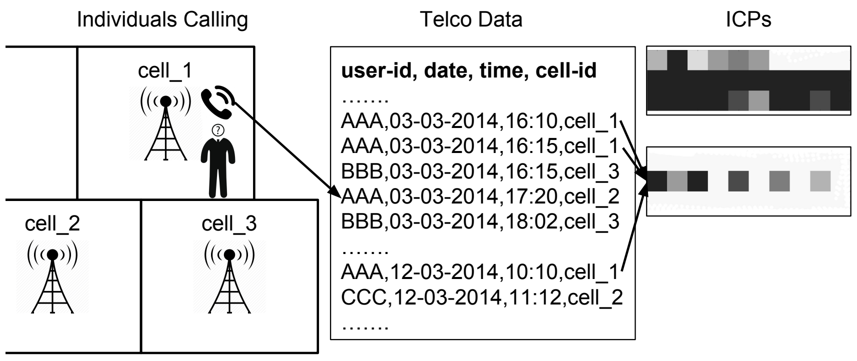

Individuals Calling
Telco Data
ICPs
cell_1
cell_2
cell_3
user-id, date, time, cell-id
....
AAA,03-03-2014,16:10,cell_1
AAA,03-03-2014,16:15,cell_1
BBB,03-03-2014,16:15,cell_3
AAA,03-03-2014,17:20,cell_2
BBB,03-03-2014,18:02,cell_3
....
AAA,12-03-2014,10:10,cell_1
CCC,12-03-2014,11:12,cell_2
....

FIGURE 8.1: Individuals perform calls under a given cell (relative to a municipality). Each call is collected from the Telco operators. We receive such data and we aggregate the calls of each individual creating a user calling profile (ICP).

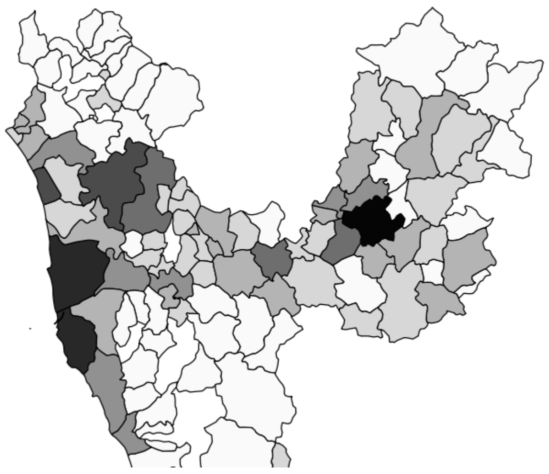
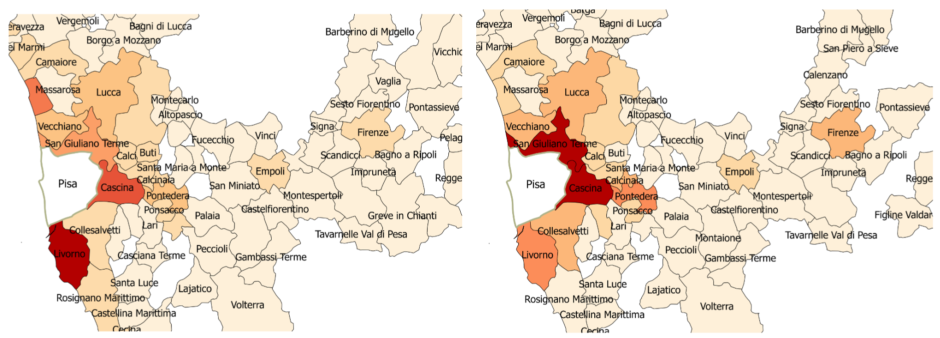


FIGURE 8.2: Amount of calls in the dataset. Darker colors represent an higher amount of calls collected in the dataset.

to a municipality. A municipality is an administrative tessellation of the territory. Our data span between municipalities having a density of population in the range 6 to 261 individuals per square kilometre. Figure 8.2 describes the amount of calls collected in the dataset for each municipality. As expected, the cities are characterized by the largest amount of calls with respect to small municipalities. For each individual, we compute an Individual Call Profile (ICP), following the approach defined in a paper from Furletti *et al.* [71]. An ICP represents the calling behaviour of an individual in a municipality (column ICPs in Fig. 8.1). Due to this, each individual may have multiple ICPs if the user performed calls in different municipalities in the time period. These are used to identify if an individual is a resident, commuter or visitor in the municipality. Each ICP is a 30-dimensional array in which each position represents a specific time slot of the day (morning, afternoon, evening) discriminating between weekdays and weekends for a total of the 5 weeks under analysis. A value greater than 0 indicates that the represented user performed at least one call in a specific time slot. At the end of the aggregation process we obtain around 2.6×10^6 ICPs representing calls generated by about 800k individuals from 115 different municipalities.

The clustering algorithm takes in input the ICPs to provide clusters of



(A) In-coming workers. (B) Out-coming workers.

Figure 8.3: In-coming and out-coming workers for the Pisa municipality

individuals and tag such clusters as Resident, Commuter or Visitor. Such information is eventually processed, to estimate the number of residents, commuters and visitors for each municipality. It is possible to extract many useful informations from such data. For instance, check Figure 8.3 where are described the amount of workers travelling in-coming and out-coming the municipality of Pisa from the other municipalities. In particular, the in-coming workers represent the amount of residents that are commuters in Pisa. It is nice to observe that the majority of the work travellers are from the surrounding municipalities, principally from Livorno. However, for the out-coming worker, despite the majority of the Pisa's resident work in the surrounding of Pisa, some travel everyday to Florence, the biggest city in Tuscany.

8.3 Preliminaries: Muchness analytical process

In this Section we provide some details about the Muchness technique targeting population estimation.

Muchness estimates the population in 3 phases:

- *individual characterization*: we start from raw data about mobile calls where for each call we have the timestamps, an individual identifier and the position of the caller. These data are aggregated resulting for each individual in an individual calling behaviour (ICP) for a given municipality similarly as in Furletti et al. [71]. An ICP provides information about the time of the day in which the individual perform calls.
- *clustering*: we cluster the ICPs with a specialized similarity metric;
- *classification*: each cluster is classified as composed of Residents, Commuters or Visitors.

Figure 8.4 gives an overview of the whole analytical process of Muchness. For each mobile user we build an ICP (see column A). Then, we start our clustering algorithm that has the peculiarity of accepting an arbitrary similarity metric. It is a two phase algorithm. First, we build iteratively a nearest neighbour graph (k-NN), according to the given similarity metric.

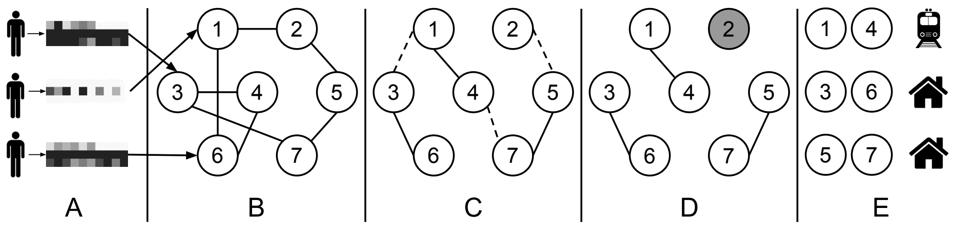


FIGURE 8.4: Muchness analytical process. A : for each individual we assign an ICP. B : each ICP becomes a node in a graph. C : we search for similar nodes and at the end we prune low similarity edges (dashed). D : we search for connected components and we identify outliers (node 2). E : for each cluster we define an exemplar (icons) classified as Resident, Commuter or Visitor.

Second, we search for connected components in the k-NN graph. We make use of the ICPs to generate the graph. At the bootstrap, we randomly link each node to few other nodes (see column B). Then, the algorithm iterates, starting from the initial graph, adjusting the neighbourhood of each node with the most similar nodes. In the following stage, the edges connecting nodes which similarity is below a given threshold parameter are pruned (see column C). The resulting clusters are the connected components (Chapter 4) derived from the pruned graph (column D). It is worth to notice how in this phase the nodes without neighbours are identified as outliers (situation represented in Figure 8.4 by node #2). Finally, for each cluster an exemplar is generated (column E), used by the automatic classifier to label the clusters as Resident, Commuter or Visitor.

This solution requires to specify three parameters: k, *numIter* and ε.

- k represents the number of neighbours for each node in the graph, it affects both the quality and the execution time of the clustering. In general is acceptable to set a value $\in [5, 10]$ to have a good trade-off between quality and time;
- *numIter* fixes the number of iterations performed by the algorithm. Larger value provides a better k-NN graph at the cost of a longer running time;
- ε is a threshold parameter that drive the edge pruning process to avoid that very different nodes would fall in the same cluster.

8.4 From Muchness to Muchness+: a framework for census

In this section we target to improve the analytical process of Muchness described in Section 8.3. In particular, we think it is of paramount importance to provide a completely unsupervised approach (i.e. where no configuration is required from the user) to avoid trial-and-error approach when changing the data to achieve a good result. To this end, we introduce some techniques to avoid the input of all the parameters required by the previous Muchness

Algorithm 8.1: Muchness+ clustering algorithm.

```
k-NN = RandomInitialization()
meanSimilarity = mean(k-NN)
tmp = -1
while $i < numIter \wedge |tmp - meanSimilarity| < 0.01$ do
    meanSimilarity = tmp
    H = $\{ReverseMap(n) \forall n \in$ k$-NN\}$
    T = {CheckNeighborhood(n) ∀n ∈H}
    k-NN = {ReduceNeighbor(n, l)∀(n, l) ∈T}
    tmp = meanSimilarity
    meanSimilarity = mean(k-NN)
    $i = i + 1$
end
$\mathcal{S}$={n ∈k-NN $|s_u < 1$}
$\varepsilon$=mean($\mathcal{S}$)
```

Algorithm 8.2: Muchness procedures.

```
procedure ReverseMap(Node n)
    forall the u ∈Neighborhood(n) do
        EMIT(n, u)
        EMIT(u, n)
    end
procedure CheckNeighborhood(Node n)
    forall the u ∈ Neighborhood(n).LIMIT (ρk)∪{n} do
        $l = \emptyset$
        forall the $v \in$ Neighborhood(n) $\cup \{n\} \setminus \{u\}$ do
            $l = l \cup ((v,$ DISTANCE(u, v)))
        end
        EMIT(u, l)
    end
procedure ReduceNeighbor(Node n, List⟨(Node, Distance)⟩ l)
    localMeanSimilarity=mean(l)
    if localMeanSimilarity ∼ 1 then
        orderedList = ORDERDESC(l).LIMIT (j/2)
        EMIT(n, orderedList)
    else
        orderedList = ORDERDESC(l).LIMIT (j)
        EMIT(n, orderedList)
    end
```

approach. We call this new version of the algorithm Muchness+. In addition, we define how we can find similarity metrics that adapt to mobile data without choosing them according to the algorithm implementation.

8.4.1 Improving the clustering algorithm

In this section we provide insights in how we improve the algorithm with respect to the one used in Muchness. In particular we concentrate on the following aspects:

- avoid bad performance in degenerate cases, bounding the number of messages to $O(\rho k)$ (Section 8.4.1);

- provide a completely unsupervised algorithm, which does not require parameters from the users. In particular, the previous algorithm requires three parameters that are not required any more in Muchness+: k the size of the neighbourhood on each node (Section 8.4.1), *numIter* the number of iterations (Section 8.4.1), ε to prune low similarity edges (Section 8.4.1).

We aim to improve the performance of the algorithm reducing the number of messages produced in the algorithm and reducing the number of iterations. We aim to achieve a better trade-off between execution time and quality of the results.

Refer to Algorithm 8.1 for an high level description of the clustering algorithm. Also, refer to Algorithm 8.2 for the details of the different phases of the algorithm and the optimizations introduced.

Introducing a sampling mechanism

Before giving the details of the sampling technique introduced in Muchness+, it is interesting to analyse the number of messages required to build the k-NN graph. Recall, the directed k-NN graph in each iteration is initially reversed to construct an undirected graph (see Alg. 8.2 Line 1). Due to this, it may happen that a node u, if by construction initially has k directed neighbours like all the other nodes, after the reverse operation in the worst case may have $n-1$ neighbours. If this is the case, $O(kn)$ messages are required in node u to communicate the 2-hop neighbourhoods to all its $n-1$ neighbours (see the Forall at Alg. 8.2 Line 7).

To avoid such degenerate scenarios we introduce a sampling parameter ρ. After the reverse operation each node keeps uniformly at random a maximum of ρk neighbours (see Line 7). This operation permits to bound the number of messages on each node, instead to $O(kn)$, in $O(\rho k)$.

Automatic neighbourhood selection

The k parameter in Muchness represents the number of neighbours for each node in the graph. It affects both the quality and the execution time of the clustering. In general, it is acceptable to set a value $\in [5, 10]$ to have a good trade-off between quality and time as suggested in Chapter 5. However, this may require an analysis and an input from the user. Due to this, we provide an heuristic to avoid such input and we let each node autonomously re-size its neighbourhood depending on its state.

On each node u, in each iteration t we define the average similarity between u and all its k neighbours equals to s_u^t. Our algorithm is iterative and improve the neighbours of each node in each iteration. Due to this given $t' > t$ we have $s_u^{t'} \geq s_u^t$. Because, if a node u discovers a neighbour v in iteration t, the neighbour v can be substituted in $t' > t$ only by a node z whose similarity with u is greater. Due to this, if u has already collected enough good neighbours we can limit its view, whereas if u have no good neighbours we need to keep k large to improve the possibility to discover better neighbours. Good means a similarity close to 1.

From the above hint, we defined a methodology able to automatically set on each node a correct value for the parameter k. We set $k = j$ when s_u^t is low (see Alg. 8.2 Line 20), and we set $k = j/2$ when s_u^t is high (see Line 17).

We let each node performs computation at two speed, respectively when it searches neighbours or when it has already found good neighbours. In this case j can be set to a value large enough to permit to discover many nodes. From previous evaluation in Chapter 5 is safe to set $j = 10$. This permits to perform more computation on nodes that needs to improve the neighbours and just preserve the connectivity on the other nodes.

Early termination

The *numIter* parameter defines the number of iterations that the algorithm performs. Previous results suggest that it is not required to perform a large number of iterations to obtain a good result. In addition, often a large part of the running time is spent for marginal improvements. Knowing the improvement of the solution through time enables the algorithm to decide on an early termination that may save longer running time. In addition, it is cumbersome to have a fixed amount of iterations to be performed without knowing the state of the algorithm.

We remark that our algorithm is improving the approximation in each iteration, and such approximation can be monitored to decide for an early termination. As before, we define on each node u, in each iteration t the average similarity between u and all its k neighbours equals to s_u^t. Also, we define S^t equals to the average of all the s_u^t for each node $u \in G$. In Algorithm 8.1 we make use of the mean function to compute S^t on each iteration (see Line 10). When the improvement of S^t in two subsequent iterations is less than 0.01 we stop early the computation of the algorithm (see Line 4). This means that the majority of the nodes do not improve the neighbours and we can safely stop the computation.

Automatic ε pruning

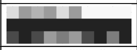
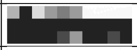
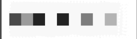
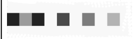
One of the most important parameters is ε. It is a threshold parameter that drives the edge pruning process to avoid that very different nodes would fall in the same cluster. It is affecting the second part of the algorithm (i.e. the same k-NN graph can be used with different ε values to cut a different number of edges). However, due to its nature, it is affecting the result considerably and it requires a trial-and-error approach to be refined.

In Muchness+ we define an heuristic capable of providing a good approximation to the expected value to be assigned to ε. As before, we define on each node u, in each iteration t the average similarity between u and all its k neighbours equals to s_u^t. At the end of the k-NN creation phase we collect the s^t values of the last iteration and we remove all the $s^t = 1$, we call such set $\mathcal{S}$ (see Alg. 8.1 Line 13). Each node u having all the neighbours identical to u must not affect the result to avoid biases. We set $\varepsilon = \overline{\mathcal{S}}$ the average of the values $\in \mathcal{S}$.

8.4.2 Adapt the metric to the data instead of the algorithm

One of the major characteristic of the clustering algorithm described before is its ability to handle arbitrary similarity metric. This contribution permits to adapt the similarity metric, used for the clustering algorithm, to the data instead of the algorithm. Thanks to this, we define and use similarity metric that are able to extract the most of the information from the data. In the

TABLE 8.2: Similar ICPs extracted by expertises. A comparison of similarity values using: EUC, JAC and EUC+JAC

			EUC	JAC	EUC+JAC
Residents			0.5	1	0.8
Commuters			0.78	1	0.91

following of this section we describe the similarity metrics used on such data (Section 8.4.2), how we can analyse each individual (Section 8.4.2) and the metrics used for that (Section 8.4.2)

Similarity metrics for ICPs

In this section we discuss on the metrics to use for our data. As introduced before, each ICP is a 30 dimensional array representing the calling behaviour of an individual. We define the *shape* of an ICP equal to the positions of its array where the values are greater than 0. The shape gives an idea about the presence of an individual in the territory without considering the amount of calls performed. The Euclidean similarity (EUC) is unable to grasp similarities between ICPs having similar shapes. Due to this, our main idea is to introduce a metrics able to capture the similarities between individual sharing a common shape.

A metric able to capture the shape of the array is the Jaccard similarity (JAC). In order to use JAC we modify each array in a boolean array where we set the value 1 in position i if in position i the data has a value greater than 0. However, the JAC takes into account exclusively the shape of the profiles but it loses all the informations about the weights in the array. Therefore we combine the two similarities, the EUC and the JAC. We define the EUC+JAC similarity as follow:

$$\text{EUC+JAC}(a,b) = \alpha\text{EUC}(a,b) + (1-\alpha)\text{JAC}(a.b) \tag{8.1}$$

Our goal is to identify the shape of the ICPs, due to this is acceptable to put more weight on the JAC. After a careful analysis we identified in $\alpha = 0.4$ an acceptable configuration.

We provide an example supporting our idea in Table 8.2. Table 8.2 shows examples of the values of the presented similarity metrics for two residents and two commuters having similar shapes. Table 8.2 represents in the first two columns the ICPs selected and in the last three columns the similarity values using different metrics. The ICPs have a very similar behaviour resulting in similar shapes. For instance, take in consideration the two residents in the first row of Table 8.2. Although some positions have different values, note the color darkness representing the value on a single position of the array, they have an equal shape representing the same calling behaviour. With the EUC we cannot assess that the two ICPs are similar (only 0.5 similarity) however the JAC (giving value 1) suggests that the two ICPs have identical shapes. With our EUC+JAC we can take the benefits of both the metrics and we obtain an high similarity of 0.8. Similar considerations can be applied also to the commuters example.

Beyond ICPs: individual profiles for individuals analysis

Once we have clustered and classified ICPs as Resident, Commuter or Visitor it is possible to estimate the population in the region and in each municipality. Another interesting analysis is understanding the different typologies of individuals. Since one ICP is relative to an individual in a municipality, an individual may have multiple ICPs, one for each municipality where he travelled in the period under analysis. To this end, we think is of paramount importance to identify how each individual moves in the region.

We define an individual profile (IP) for each individual i. It is constructed from the outcome of the clustering of the ICPs. An IP is a 3 dimensional array where each position represent the number of times i is respectively considered a Resident, a Commuter and a Visitor. The aim of this characterization is to identify groups of individuals sharing a common behaviour. In particular, we would answer the following questions:

- how many individuals are just visitors of the region?
- how do residents of a municipality move in the region?
- do individuals exist visiting many places and performing many calls? (i.e. maybe some individuals are classified residents in multiple municipalities)?

To answer these questions we cluster the IPs in order to aggregate similar individuals. Since we have specific questions to answer we need to carefully choose also in this case the correct similarity metric to be used for the IPs. Again, thanks to our algorithm that support arbitrary similarity metric we can define a metric suitable for our data without warring about its suitability in the algorithm. In the following section we define how we choose such metric.

Similarity metrics for individuals profiles

In this section we define several metric that can be used for clustering IPs. We think that the most important value in the IP is the number of times an individual may be considered a Resident. Note, a value of 0 or 1 represent an individual that is respectively not a resident in the region under exam and a resident in one of the municipality under exam. However, it may happens that an individual has a value greater than 1. This means that such individual is moving in many municipalities and it is performing many calls in each of the municipalities. For instance, consider salespeople. Individuals not having a fixed working place and their work is mainly characterized on meeting people in different places, organize such meetings by phone and keep in touch with all the customers. Due to this, they are individuals that in our data will emerge having multiple ICPs and in some of them, where they are more present, having an high number of calls resulting in a Resident profile.

For all the above motivations we need a metric capable of correctly identify clusters keeping well separated individuals having a different value in the Resident slot. The euclidean distance is not enough to grasp such differences. For instance, it gives the same importance to the values in the resident and visitor slot. However, it is more important to differentiate between an individual being a resident in 2 municipality from an individual resident in 3 with respect to two individuals being visitors respectively in 0 and 5 municipalities. Due to this, we defined a personalized metric. Such metric assign a similarity equals to 0 to individuals having a different values for Resident. Instead, it assigns a value equals to the euclidean distance between the values of commuters and visitors for those individuals having the same value in resident.

8.5 Experimental evaluation

All the experiments have been conducted on a cluster running Ubuntu Linux consisting of 5 nodes (1 master and 4 slaves), each equipped with 128 Gbytes of RAM and with two 16-cores CPU, inter-connected via a Gigabit Ethernet network. We implemented our approach using Apache Spark [2], the source code we used for conducting our experiments is publicly available on GitHub[1].

To study the performances of Muchness with respect to alternative existing approaches, we compared against the following competitors:

- Sociometer [72] is the primary competitor, it is the most similar to Muchness; both the approaches are based on clustering and designed for the same case study;
- MP [57] targets the same problem, however is not based on clustering but relies on rules, such as the calling hours to identify if an individual is a resident. Such approach requires the knowledge of additional data as for instance the total amount of individuals in a region. Since such data may be affected by fluctuation or can be missing we make use of a version of MP not requiring additional parameters;
- dbscan, we tried to conduct our experiments with an implementationof MR-dbscan [80] on Apache Spark, unfortunately we have not been unable to cluster more than the 10% of the dataset due to memory errors due to the high dimensionality of the ICPs.

8.5.1 How to configure Muchness+

With the optimizations introduced for Muchness+ we target to remove all the input from the users to facilitate the usage. Although, in Section 8.4 we described how to remove all the parameters, that was previously required by Muchness, we introduced the optional parameter ρ to avoid degenerate scenario and to perform a trade off between running time and quality. Figure 8.5 depicts the results when using a value of $\rho \in \{1, 2, 3, 6\}$ compared with Muchness. We found that $\rho = 1$ is a too strict configuration and does not permit to achieve a good result (i.e. the algorithm is not able to estimate correctly the number of residents). However, already with $\rho = 2$ we obtained a result comparable with the ones obtained with larger values of ρ. Also, keeping low ρ permit to have a shorter running time because each node sends ρk messages. Due to this, we suggest to use $\rho = 2$ because shows the better running time and a quality similar to different configurations.

[1]`https://github.com/alessandrolulli/knnMeetsConnectedComponents`

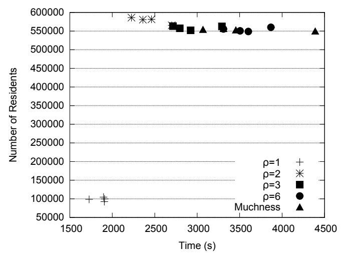


FIGURE 8.5: How to configure Muchness+: analysing the sampling parameter (ρ)

TABLE 8.3: Road to Muchness+: adaptive k optimization

	Time (s)	#cluster	Residents	Compactness	Separation
Muchness	3164 (±183)	569	552829	0.89	0.76
Adaptive k	1873 (±42)	265	632393	0.86	0.72

8.5.2 Road to Muchness+: evaluating optimizations

In this Section we evaluate the impacts of the optimizations introduced in Section 8.4. We start evaluating each optimization separately. At the end of the Section we build Muchness+ and we compare it with Muchness. According to the result of the previous Section we set the sampling mechanism in all the experiments equal to $\rho = 2$. All the results are the average of 5 independent runs. For the time metric we reported in parentheses also the 95% confidence interval.

Adaptive k

In the first set of experiment we evaluate the adaptive k optimization (Section 8.4.1). Table 8.3 depicts some comparison metric with respect to Muchness. In particular, with the adaptive k optimization we gain two major results. First, the time to reach the solution is around the 40% less with respect to the time of Muchness. This is motivated by two things. The sampling mechanism as seen in the previous Section has a remarkable impact on the execution time. However, this is not the only cause of the great gain, note that in Figure 8.5 all the values having $\rho = 2$ have execution time > 2000 seconds. Here we get an average execution time of 1873 suggesting that with respect to the execution time of $\rho = 2$ we obtain another gain thanks to the possibility to reduce the neighbour size of a large part of the nodes in the graph. This has no impact on the quality of the clusters obtained, in fact compactness and separation have similar values with respect to Muchness. The second gain regards the number of clusters. Keeping the number of cluster low has some benefits because it permits to analyse a lower number of clusters if a manual investigation is required.

Early termination

Next, we move to the analysis of the early termination mechanism. This optimization, described in Section 8.4.1 permits to avoid the use of the $NumIter$ parameter of Muchness. Despite this advantage, it shows also execution time advantages. Table 8.4 describes the results and shows a comparison with Muchness. As in the previous Section, we obtain a remarkable

Table 8.4: Road to Muchness+: early termination

	Time (s)	#cluster	Residents	Compactness	Separation
Muchness	3164 (±183)	569	552829	0.89	0.76
Termination	1697 (±80)	293	586398	0.87	0.73

Table 8.5: Muchness+ vs Muchness

	Time (s)	#cluster	Residents	Compactness	Separation
Muchness	3164 (±183)	569	552829	0.89	0.76
Muchness+	1309 (±15)	161	634402	0.87	0.71

advantage in terms of execution time. Thanks to the early termination we finish the computation in half of the time with respect to Muchness. To be more precise, the early termination ends the computation after 6 iterations. This suggest that the subsequent iterations performed by Muchness improves only marginally the result obtainable by this approach. Also, this is another confirmation that monitoring the results on iterative algorithms permits to have a deeper control about the quality of the results.

All

Finally, we build the Muchness+ algorithm inserting all the previously analysed enhancements. Here, we add also the optimization to automatically select the threshold parameter for the pruning mechanism before running the connected components. Table 8.5 shows the final results. With all the optimizations the execution time of Muchness+ is around 60% lower than the execution time of Muchness. In particular, it seems that the contributions of adaptive k and early termination optimizations are additive and both contribute to reduce the execution time. We get also a more stable running time in the 5 executions and we obtain only ±15 for what concern the confidence interval of the time metric. The number of clusters is sensibly smaller in Muchness+ but this is not affecting the number of residents estimated by such approach.

8.5.3 Studying individuals mobility

In this set of experiments we aim to answer some questions about the indicators that can be extracted from mobile calls. For instance, it is reasonable to assume that call activity in commercial or business areas is an indicator for economic activity. Also, in previous works [50] it has been suggested that phone calls can be an indicator of economic activity. First of all our method allows to use the calling behaviour to understand if an individual resides, works or is a casual visitor in a certain place. Due to this we answer to the following questions: *(i)* is it possible to classify regions as residential, commercial or business? *(ii)* is it possible to show how individuals move to reach their working position? *(iii)* is it possible to check which are the most visited places?

Using the outcome of Muchness+, we can see how individuals move into the territory under study. In particular, we analyse how individuals move from home to the working place or to visit a city. The thickness of the flows is proportional to the number of individuals travelling the path connecting the two municipality.

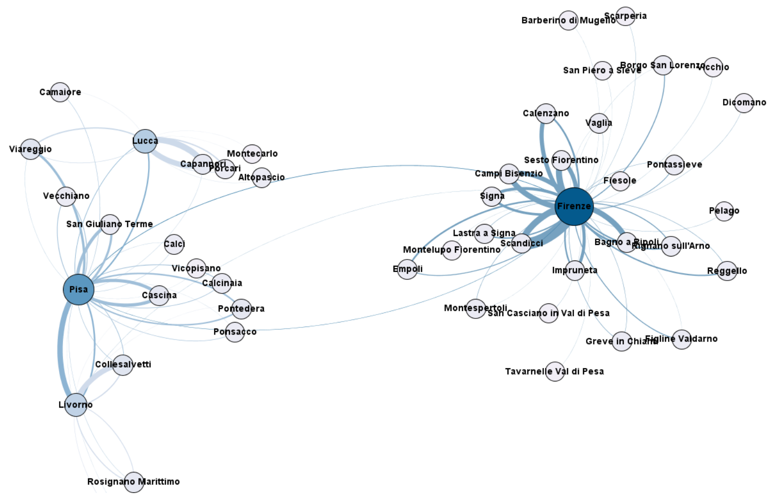


FIGURE 8.6: Studying individuals mobility: Individuals travelling from home to work

Individuals travelling from home to work

One possible use of the outcome of Muchness+ might be to systematically analyse the movement from home to work. This may have multiple advantages such as observe what is the potential market for public transport. Also, this service can be very useful for statistical institutes. Figure 8.6 shows the main home to work flows for each municipality in Tuscany. Clearly, Florence, being the biggest city in Tuscany, is the center of the working activity of the region. A large part of the individuals of the surrounding municipalities everyday move to Florence for the working hours. From the Figure also the other cities of Tuscany are highlighted, in order of importance: Pisa, Lucca and Livorno. Pisa, despite being smaller than other cities such as Livorno, is a center of numerous activities and of prestigious universities. Due to this it seems to be the second municipality, after Florence, to attract workers. From the figure it is possible to obtain two other major insight. First, the larger centres of working activities seem to attract workers from their surrounding municipalities, note that the flows directed to the cities are from the surrounding municipality (the edges have a clockwise direction). Second, flows exist also between the major cities. For instance, between Livorno and Pisa or Pisa and Florence. This may show the impact of rail transportation because a path connecting Livorno, Pisa and Florence by train exists.

Individuals travelling from home to visit places

While some statistics about systematic movements may be extracted also from census, this is not true for occasional visits. Due to this, it would be very helpful to know, for instance, who has attended an event and where they come from or how visitors are attracted in certain municipalities. This would enable to know the spread and importance of an event by measuring the attractiveness over the surrounding territory. Figure 8.7 depicts the flows between the municipality of residence and the visiting places. We can see that the amount of mobility that is created for occasional reasons is impressive, and certainly greater than that happening systematically and/or

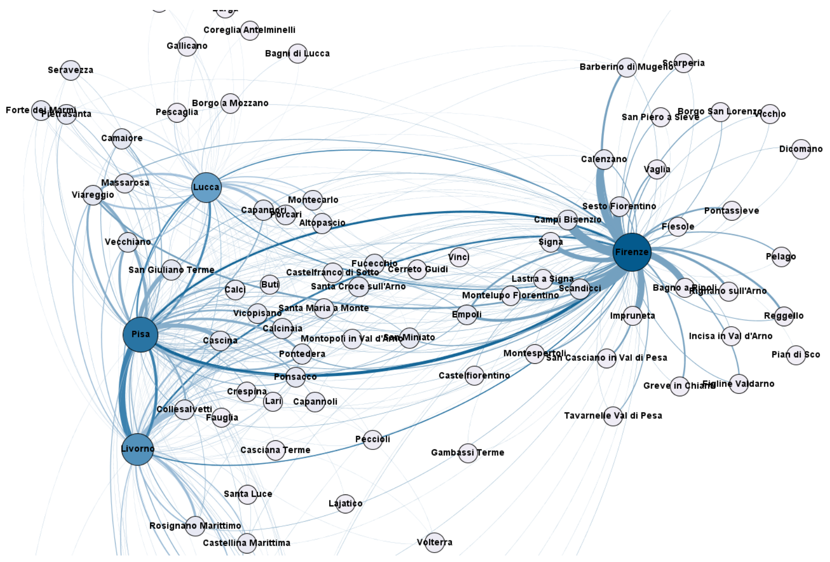


FIGURE 8.7: Studying individuals mobility: Individuals travelling from home to visit places

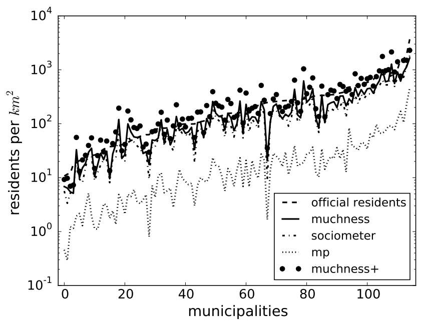
TABLE 8.6: Comparing with competitors and census data: Median estimation errors

	Residents $\times km^2$			
	<50	50 - 100	100 - 150	>150
MP	93%	91%	92%	94%
Sociometer	39%	39%	49%	52%
Muchness	24%	29%	42%	47%
Muchness+	**16%**	**14%**	**15%**	**23%**
	Commuters $\times km^2$			
Sociometer	**83%**	84%	86%	89%
Muchness	84%	**83%**	**81%**	**87%**
Muchness+	86%	84%	85%	89%

due to working activities. Again, the figure shows that in particular the movements involving occasional travels are to the four largest Tuscan cities considered. These are important destinations for tourism by Italian and foreign citizens. A difference with the movements for working activity is that not only the surrounding municipalities but the individuals of quite all the municipalities travel occasionally to the major cities. For instance, from Camaiore, a small municipality in the top-left corner of the figure there are flows directed to all the cities. Also, in the figure many more municipalities are present with respect to Figure 8.6. We can see that individuals travel occasionally to many more places than those they visit for working reasons.

8.5.4 Comparing with competitors and census data

In this Section we evaluate how Muchness+ is capable to be an indicator for measuring the amount of residents in a municipal area by comparing its results against MP, Sociometer and Muchness. In addition, we evaluate also the amount of estimated commuters against Sociometer and Muchness. Note, the MP method is limited and specialized in providing only the number of residents and does not provide any functionality to estimate commuters. It is worth to notice that all the estimations have been rescaled using the market share of our telco provider. The results are compared against official



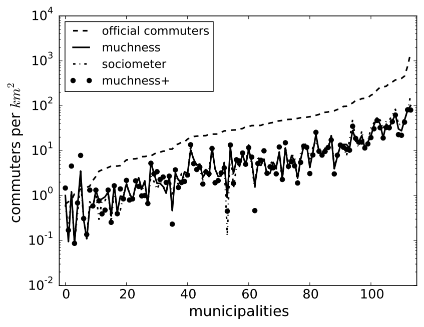


FIGURE 8.8: Comparing with competitors and census data

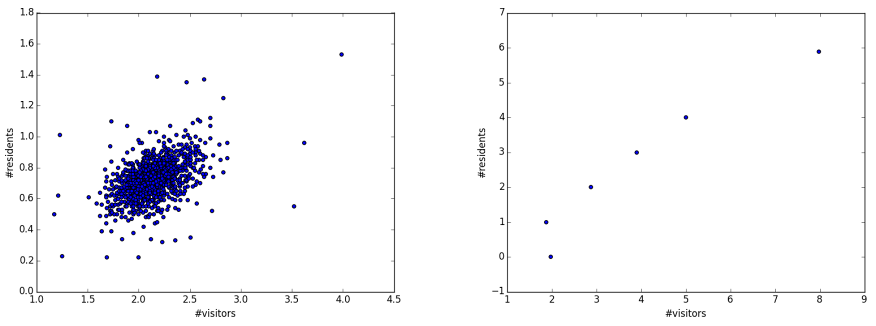
census statistics provided by Italian national institute of statistics (ISTAT). This data includes the amount of residents and commuters belonging to the 115 municipalities we studied.

Figure 8.8 depicts the number of residents identified for each municipality. On the Y axis we show the estimated population whereas on the X axis the municipalities ordered from the lowest to the highest population density. The results are compared with the real census provided by ISTAT. As it can be noticed, all the methods have spikes in the same municipalities. This suggests that although the methods are based on different approaches (MP defines rules, Sociometer and ours on clustering) all identify similar behaviours on the data and may suggest that census data itself could under or over estimate population. It is evident that MP is always under estimating the density with an error that is greater than Sociometer, Muchness and Muchness+. Muchness+ seems the one closer to the real census data. In particular for higher dense municipalities.

To have a better insight on the errors performed on the estimations, Table 8.6 presents the median error on the estimations. We divided the error on the estimations in 4 areas having different population density. Again, MP is providing the estimation affected by the larger error. Muchness and Sociometer provide similar results for the municipalities with higher density where the volume of available data is large and the clustering can rely on a rich set of information. Instead, Muchness+, as we noted before, provide a better estimation in all the municipalities and in particular it improves the result of Muchness on higher dense municipalities. Finally, we compare the commuters estimations. Also in this case the results are compared against real census data. All the approaches give approximately the same results in terms of estimation errors, for every density range.

8.5.5 Evaluating individual profiles

In Section 8.4.2 we defined the individual profile (IP) to identify different typologies of individuals and in Section 8.4.2 a metric capable of extracting useful information from such data. In this section we compare two metric for clustering individual profiles: the Euclidean distance and the one defined ad hoc for such data. Initially we take the output of Muchness+ and we constructed for each individual its IP. We obtained around 800k IP.



(A) Euclidean Distance (B) Personalized Distance

Figure 8.9: Evaluating individual profiles

First of all we compared the clusters obtained with the two metrics. Figure 8.9 depicts for each cluster a point in a 2D space where on the X axis is represented the number of times the exemplar of the cluster is a Visitor and on the Y axis the number of time is a Resident. With the Euclidean distance we obtained many more clusters with respect to the personalized metric specific for the data. With the personalized metric we obtain a small number of clusters and each cluster is well defined and separated from the others. Thanks to this, the result is easier to be analysed and two clusters having similar characteristics do not exist.

We then proceed with a manual investigation of the clusters obtained with the personalized metric. Table 8.7 presents the 6 clusters obtained with their sizes and the values of the 3 dimensional array. We observe that more than the half of the individuals (51%) are not residents in the region under exam. This means that the majority of the people travelling in the region are living outside the region and visit Tuscany for tourism or for short periods of time. As expected, the second biggest cluster is the one where individuals are residents in only one municipality. However, some individuals exist that are considered Residents in more that one municipality. This is of paramount interest because this highlight that a part of the population exist, albeit not being very large, that is used to travel a lot. Such individuals exhibit a routine in their movements because to be considered a resident in a municipality an individual must perform many calls in different moments of the day. Another interesting fact is that the individuals resulting resident in more than one place are the individuals resulting, on average, visitor of the highest amount of places. Note that the values of Visitor are increasing when also the values of Resident are increasing. Also, as expected, the size of the clusters are decreasing when the values of resident are increasing. Finally, we noted that the values in the Commuter column is always close to 0. The motivation is that the number of commuters, as we found when comparing with real data, is low. Due to this, albeit a part of the population being a Commuter, when clustered with other individuals the exemplar result in a value close to 0.

TABLE 8.7: Evaluating individual profiles: exemplars

Cluster size	Exemplar		
	# Resident	# Commuter	# Visitor
426767 (51%)	0	0.01	1.98
265629 (32%)	1	0.01	1.87
94412 (11%)	2	0.02	2.88
32460 (4%)	3	0.03	3.9
11001 (1%)	4	0.03	5
6744 (<1%)	5.88	0.06	7.98

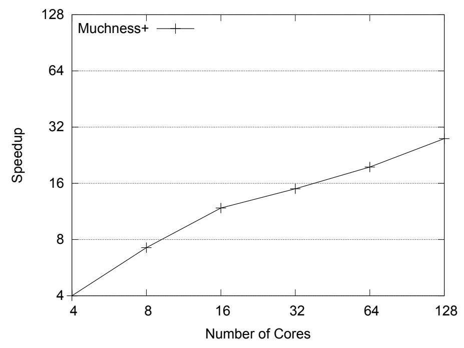


FIGURE 8.10: Scalability evaluation

8.5.6 Scalability

Finally, we tested the scalability of Muchness+, by varying the number of cores in the range [4,128]. Figure 8.10 depicts the results we achieved. Recall, Muchness+ has been built taking inspiration of two previous works for computing connected components (Chapter 4) and text clustering (Chapter 5). In such works similar patterns have been observed when evaluating the scalability using several different typologies of datasets. We obtained an almost linear scalability using 8 cores, a still good level scalability with 16 cores, then the value tends to stabilise albeit it is always improving while adding more cores. These results can be motivated with several considerations about the testing environment. Spark allocates the cores according to a round robin policy: when using 4 cores Spark exploits one core from each of the 4 machines. As a consequence, by using only 4 cores (of the 128 available) we exploit the total amount of memory available in the cluster. Considering that each machines has two CPUs, we reach the maximum available CPU-memory bandwidth, and thus linear scalability, when using 8 cores (one core per CPU).

8.6 Conclusions

This chapter presents an application for estimating the population making use of mobile calls. With respect to the existing solutions, we presented an unsupervised clustering algorithm that does not require any input from the user. It is versatile and able to accommodate arbitrary similarity metric and it is able to process any typologies of data.

Furthermore, we give an experimental evidence that our approach provides a very good estimation of the population density within the Italian region of Tuscany.

Chapter 9

Hashtag centrality

This chapter describes an application whose main goal is to identify, on a daily basis, which are the most important hashtags in Twitter. These are the hashtags where the majority of the information flows in a specific day [19]. This may have multiple applications such as identifying the topics of a given day, identifying the key hashtags for a community or maximizing the number of users reached by a tweet. In addition, central hashtags may be considered to suggest hashtags that users can track on an ongoing basis or to perform query expansion.

Our approach is to construct a graph of hashtags and search for the most central hashtags thanks to the DUCKWEED algorithm presented in Chapter 3. To this end, we make use of a dataset composed of 606 days of tweets previously collected at ISTI, CNR. Each tweet including at least two hashtags contributes to construct a graph where a node is an hashtag and an edge represents the co-occurrence, in a same tweet, of two hashtags. The main contributions of our approach are the following:

- we show how our algorithm DUCKWEED to compute the current flow betweenness centrality can be applied to this application;
- due to the size of the dataset, it is mandatory to use a scalable algorithm able to provide a solution even for large graphs. To this end we exploit DUCKWEED that by means of the approximation introduced in its definition, is providing a valuable solution. Also, it confirms that is able to work on large graphs as demonstrated by the experimental results of Chapter 3;
- our results reveal that our algorithm DUCKWEED is able to identify the topic of each day and to correctly recognize the importance of the hashtags.

9.1 Related works

A Twitter hashtag is a string of characters preceded by the hash (#) character. The first usage of an hashtag has been in August 2007 by Chris Messina, who posted on Twitter the tweet "how do you feel about using # (pound) for groups? As in #barcamp [msg]?" [145]. Nowadays hashtags have a pervasive usage and are used as topical markers, an indication of the context of the tweet or as the core idea expressed in the tweet, therefore hashtags are adopted by other users that contribute similar content or express a related idea. Such usages motivated a rich research about hashtags' analysis.

Efron et al. [62] focuses on suggesting a list of hashtags that are relevant to the information need of a query. They use the ranked list of suggested hashtags for the query expansion task. Similarly, Godin et al. [74] develop a language classifier in order to recommend hashtags for a given tweet.

Instead, Tsur et al. [173] focus on predicting the spread of ideas in online communities. They use an hybrid approach analysing both the social graph of Twitter, the graph representing users follower relationships, and the tweets' content. An interesting aspect is that they consider Twitter hashtags as ideas and they evaluate the spread of such kind of idea in a time frame.

Hashtagify [19] is an application more related to ours. It is a popular application to monitor the importance of hashtags in time, showing many charts with different characteristics. For instance, it provides the possibility to analyse hashtag popularity with weekly and monthly variation. It shows hashtags related to a given one and the trending hashtags. Such application has been exploited to show that nowadays social media plays a vital role in socialization [156]. Also, to analyse the trend of health-related posts [174] to aid in timing interventions.

9.2 Data Description

The dataset is composed of a random sample of the tweets in Twitter which have been downloaded thanks to the Garden Hose Streaming API provided by Twitter. Tweets are collected in one file per day, each file is encoded in JSON. Every day are collected the 1% of the tweets provided by the streaming API in an unique file, the compressed file size is around 20G and it contains approx 40M tweets. Such raw data is splitted in three different dumps, namely:

- *English Dump*: contains only the tweets written in English, 6/7GB (compressed) per day, 15M tweets per day;
- *Italian Dump*: contains only the tweets written in Italian, 250Mb (compressed) per day, 600K tweets per day;
- *Georef Dump*: contains only the tweets containing geotags latitude and longitude or the place, 600/700Mb (compressed) per day, 1M tweets per day.

The data has been collected by HPC Lab at ISTI, CNR and it is available at `http://rojo.isti.cnr.it/`.

9.3 Our Approach

In this Section we present our analytical process to identify central hashtags. The idea is to construct a co-occurrence graph. In details, we build a graph where each node is an hashtag and an edge exists between two nodes if the corresponding hashtags co-occurre in a same tweet. In order to evaluate the fluctuation in the popularity of the hashtags we use a time frame of one day and we re-build the graph for each day. Central hashtags have important characteristics:

- *trending topics*, in similar works [173] hashtags are defined as a way to represent ideas. Due to this, central hashtags may reveal the most important topics discussed in a specific day because they are the most used words to spread the ideas;

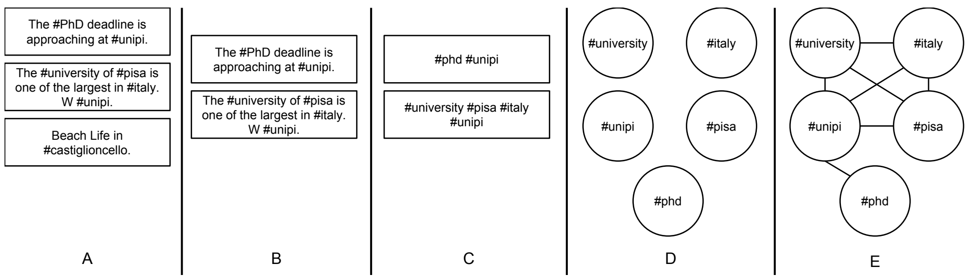

The #PhD deadline is approaching at #unipi.
The #university of #pisa is one of the largest in #italy. W #unipi.
Beach Life in #castiglioncello.
#phd #unipi
#university #pisa #italy #unipi
#university
#italy
#unipi
#pisa
#phd
A
B
C
D
E

FIGURE 9.1: How to construct a co-occurrence graph from raw tweets

- *key hashtags for a community*, if we take all the tweets relative to a community (e.g. relative to a specific hashtag) the most central hashtags are the ones attracting more discussion within the community;
- *query expansion*, central hashtags may be used to expand a given query with additional contents. This technique is usually employed to expand the search query in order to match additional contents. In this scenario additional hashtags matching the text topic may be introduced to improve the results;
- *hashtag suggestion*, given a tweet the central hashtags can be used to enrich the tweet text to reach more people. In this scenario, some hashtags, related to the text, may be suggested to the writer.

In the following of this section we describe how we construct the hashtag graph and our approach to identify central hashtags.

9.3.1 Graph Construction

In order to perform a graph analysis, the first step of our approach is to build the graph starting from raw data which is the Italian dump of daily tweets collected as described in Section 9.2. For each day we construct a different graph. This is motivated by the fact that we aim at identifying the most important hashtags in a given day to show the fluctuation of hashtags' popularity.

Take as reference Figure 9.1 where Column A presents a sample of 3 tweets. Initially, we preserve only the tweets having at least 2 hashtags (Column B). For each tweet we maintain only its hashtags list (Column C). Then, each hashtag becomes a node in a graph and we remove duplicates (Column D). Finally, an edge is inserted between two hashtags that have a co-occurrence in at least a same tweet (Column E). The weight of each edge is equal to the number of tweets in which the two hastags co-occurre. For instance, node #phd is connected to node #unipi because they initially appear in the same tweet "The #PhD deadline is approaching at #unipi."

9.3.2 Centrality Computation

In order to identify which are the most central hashtags in each day we employ our algorithm to compute the current flow betweenness centrality defined in Chapter 3. The main contribution of this chapter is to show how a single algorithm is able to provide a valuable result for an interesting

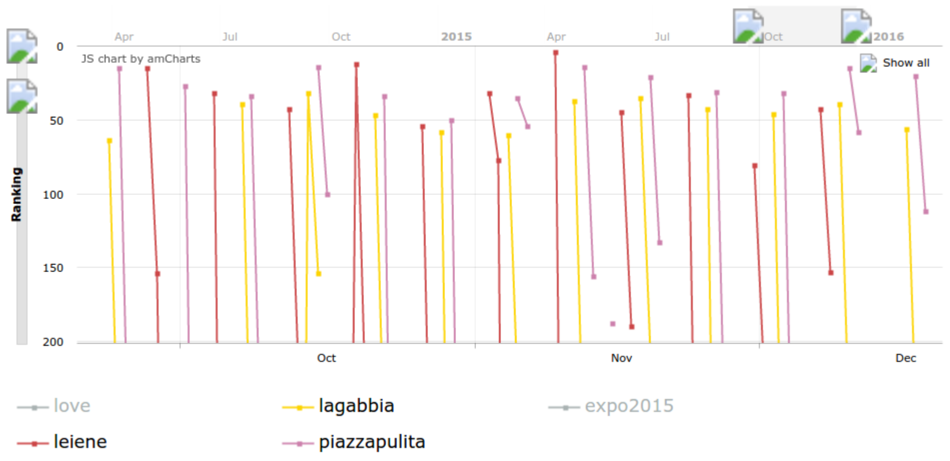


FIGURE 9.2: Identification of cyclic events

application domain. Also, thanks to the approximation of DUCKWEED, we are able to provide results in a reasonable time.

9.4 Validation

In this section we present some case study and information that can be extracted by our application. In particular, we consider how make use of the outcome of our analysis and how use our tools in order to get information about popular hashtags.

9.4.1 Identification of cyclic events

Hashtags, according to many experts and published works [145], can generate immediate, live, and interactive reactions and responses to specific topics. People use hashtags while watching their favourite TV program, listening to a debate on the radio and in other similar situations. Due to this, we evaluated if DUCKWEED is able to identify such trends while they occur. To this end, we selected three popular TV shows in Italy, namely "La Gabbia", "Le Iene" and "Piazza Pulita". Each of these programs has a specific hashtag advertised during the show, respectively #lagabbia, #leiene and #piazzapulita. We searched for these hashtags in the result provided by DUCKWEED and the results are presented in Figure 9.2. These hashtags have picks (i.e. higher ranking) in those days when the TV programs have been displayed, i.e. one specific day each week. Interestingly, it is evident how the peaks are repeated every 7 days. All these TV programs, usually, exhibit their hashtags in the top 100 hashtags used in their specific day. Finally, it seems that between the two politics related programs considered (#lagabbia and #piazzapulita), "Piazza Pulita" usually obtains an higher ranking resulting in an higher volume of tweets relative to it. We repeated this validation also for other TV programs and we encounter analogous results.

9.4.2 Identification of seasons

In this set of experiments we validate how DUCKWEED is able to recognize periods and seasons of the year. In the previous section we analysed some

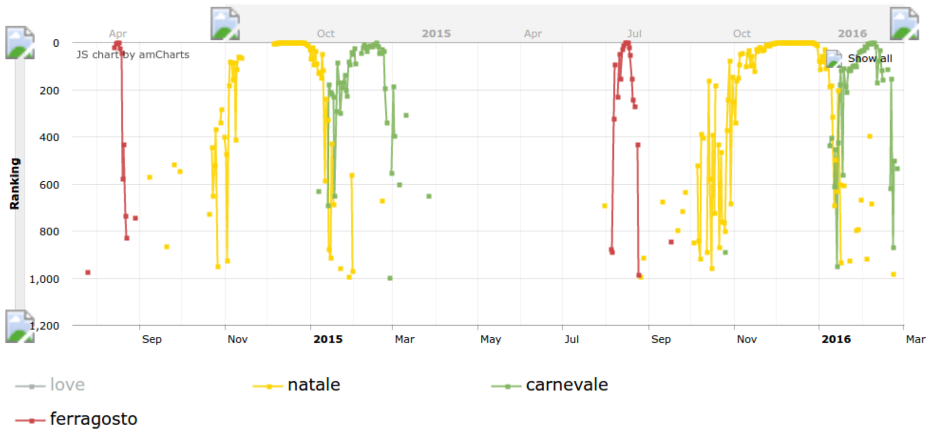


FIGURE 9.3: Identification of periods

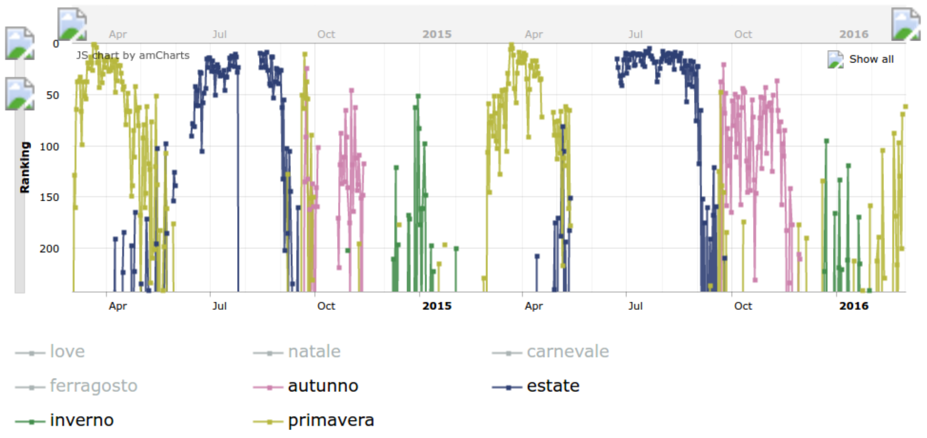


FIGURE 9.4: Identification of seasons

hashtags that have a popularity limited in time, they are popular only in specific situations. Instead, many hashtags have a popularity with an higher duration in time and may reveal important events that are happening around us.

For instance, during the year, hashtag can be paired to holiday periods such as Christmas (#natale), carnival (#carnevale) and August Bank Holiday (#ferragosto). Figure 9.3 presents the ranking of these 3 hashtags in a time spanning of two years. It is evident how #natale is highly discussed also more than one month before the 25th of December. In particular, #natale has a solid top 10 position for more than one month over the Christmas day.

Instead, Figure 9.4 shows how the 4 hashtags representing the seasons have impact in the tweets' discussions. We consider 4 tweets regarding the name of the seasons: summer (#estate), autumn (#autunno), winter (#inverno) and spring (#primavera). As expected, each of the hashtag has the higher ranking in its corresponding season. What is interesting is that people seems more likely to speak about summer with respect to winter. The #estate hashtag, in summer, results to have, on average, an higher ranking with respect to #inverno in winter. This may reflect that people enjoy discussing about summer more than winter, because it is recognized as an holiday period, for instance a typical tweet may be "Finally! Summer has begun! #summer".

9.5 Conclusion

In this chapter we presented how our algorithm DUCKWEED for the current flow betweenness centrality can be exploited to provide informations about the most discussed topics on Twitter. To this end, we constructed a co-occurrence graph of the hashtags and we ranked the node according to DUCKWEED. Results show that our approach is able to deliver useful insight about many applications regarding hashtags, for instance, to find the trending topics and analyse the flow of information in Twitter.

Chapter 10

Conclusion

How should we design applications able to extract valuable information from large graphs? This thesis shows how to exploit the “Thinking Like A Vertex” approach (TLAV) to design efficient algorithms supporting the implementation of applications dealing with graphs. TLAV is a popular approach and algorithms defined according to such methodology can be implemented on the majority of the currently available distributed environments. In order to improve the performances and the usability of the algorithms realized according to this approach, we identified a set of useful guidelines focused on *approximation*, *simplification* and *versatility*. Such guidelines have been exploited for the definition, conception and development of four algorithms, presented in this thesis for solving problems focused, respectively, on the detection of connected components, on the computation of the betweenness centrality, on clustering and on density based clustering. These algorithms have been leveraged to realize solutions targeting three different applications.

Our algorithm for the detection of connected components outperforms state-of-art competitors. Such achievement has been obtained by means of a smart simplification of the graph. It has been also exploited to realize our clustering algorithms. A remarkable feature of these algorithms is their versatility to accommodate any kind of data. This permits to achieve good performances regardless the different datasets that can be provided in input. Furthermore, we extended the original algorithm to support density based clustering. Along this, we also introduced a simplification technique aimed at improving the overall performances. Then, we defined a novel distributed algorithm for the computation of the current flow betweenness centrality which targets highly decentralized environments where, to the best of our knowledge, a distributed solution was missing in the reference literature.

These algorithms have been successfully exploited for developing three different applications: spam campaign detection, population estimation and hashtag centrality. All these applications show useful and valuable results. Even more, the study on the population estimation helped in improving the clustering algorithm.

Finally, it is worth to point out that one of the major contribution of the TLAV approach is to push programmers and data scientists to follow a distributed programming perspective when developing their applications. Recently, other programming models have been proposed, such as the ones modifying the granularity of the computation. However, the key features identified in this thesis remain as useful guidelines to leverage also in slightly different contexts.

10.1 Discussion

We conclude presenting a final discussion concerning the workflow of the thesis starting from the considerations described in the introduction. We can take as reference the very same Figure 1.1. The goal of this thesis has been showing how simple algorithms can be composed to build complex ones and how such algorithms are useful in nowadays applications to solve particular problems.

In the thesis we started presenting such simple algorithms, namely CRACKER and DUCKWEED. Such algorithms provide solutions to extract two important characteristics of graphs, the connected components and the current flow betweenness centrality. In many situations, these informations already provide valuable insights about the graphs analyzed. However, these algorithms may be used to compose and generate additional algorithms. In particular, CRACKER has been used in conjunction with another algorithm for the computation of a k-NN graph to perform clustering. The initial motivation on the study of this clustering algorithm has been also to show how to compose already available algorithms. The idea is to create a pipeline of algorithms where the output of the first one is the input of the second one. Due to this, we initially compute the k-NN graph with a specified algorithm where each item is connected to the most similar items and then, we search for the connected components, whereas each component becomes a cluster. Such work has been also refined to perform density based clustering in NG-DBSCAN. Finally, the clustering algorithms have been used successfully to solve two real applications for detecting spam campaigns and to estimate the population moving in a region, finding how much residents, commuters and visitors travel in that area.

Instead, DUCKWEED has been another proof of how a simple algorithm may be used in real world applications. To this end, our algorithm, has been successfully exploited to detect the most important hashtags. Such hashtags reveal, for instance, the topics discussed on a given day in Twitter and may become the first starting point to perform automatic event detection.

10.2 Future Works

Starting from the results achieved in this thesis, there exist many researches that can be conducted. In the remaining of this section, we identify a few potential research subjects and present some still open questions.

10.2.1 Simplifications Techniques

Simplification has an high impact on the performances of our algorithms. In fact, we think that introducing simplification techniques on the algorithms greatly improves their suitability for large graphs. For instance, the data driven simplification technique introduced in our connected components algorithm can be re-adapted to different solutions. It may provide valuable results also for different kind of graph analysis, such as the diameter estimation [43] or single source shortest path computation. Also, additional simplification techniques can be defined for different problems. An idea we are currently studying is to borrow the simplifications commonly used in electric circuits to find a way to simplify the graph partitions assigned to each distributed node.

10.2.2 From TLAV to Thinking Like a Sub-Graph

As we pointed out in our thesis, the majority of the most popular frameworks currently support the TLAV programming model. However, recently, frameworks adopting a different granularity of computation have been defined, for instance, the so called subgraph centric computation. A graph, can be partitioned into sub-graphs that can be stored into the memories of the distributed nodes. An interesting consequence is the reduction of edges, i.e. the connections between sub-graphs would be reduced with respect to the edges of the original graph [131]. Frameworks supporting such kind of computation are Giraph++ [170], Blogel [188] and Bladyg [10]. Changing the granularity of the computation requires the definition of novel algorithms able to exploit the optimizations introduced by the decomposition. Other approaches based on increasing the granularity of the computation recently proposed, are based on paths and sets.

10.2.3 Algorithms on dynamic graphs

In this thesis we defined algorithms working on static graphs that are processed using batch jobs. However, in some scenarios, graphs can be dynamic and change over time. More precisely, a graph may evolve in many ways, for instance, adding or removing nodes or changing the topology at fixed times and many more. In all these scenarios, the algorithms adopted should be aware of this and ready for the changes. Due to this, an interesting area of research is to define and implement algorithms that dynamically adapt to the underlying graphs and perform graph analysis in a continuous manner.

10.2.4 Improve Frameworks: Parallelism and Compression

Another line of research could regard the improvements of the frameworks devoted to the distributed computation. In this area exists a plethora of different frameworks and possible optimizations. However to wipe the slate clean and build a completely novel framework requires a huge effort. We think that exist two major branches of research from where it is possible to borrow important functionalities.

The first one is parallelism. All the distributed frameworks employ some sort of parallelism in each of the machine used. For instance, in MapReduce, after the workload is split on the machines, then all the cores of each machine execute a predefined task. Nevertheless, if we look at the single machine parallel framework literature, a lot of such frameworks are devoted to optimize the work on the single machine. One idea could be to integrate such kind of highly parallel single machine frameworks, for instance the ones exploiting GPUs and FPGAs to be exploited in the distributed frameworks [166].

The second one is compression. One of the largest cost when distributing the computation on multiple machines is the communication cost. Also in this case, usually, the distributed frameworks adopt some kind of compression to reduce the communication cost. However, some specialized compression strategy may be employed adapted for the TLAV pattern of communication.

Bibliography

[1] Apache Giraph. http://giraph.apache.org/.

[2] Apache spark. https://spark.apache.org.

[3] Apache spark machine learning library. https://spark.apache.org/mllib/.

[4] Clustering the News with Spark and MLLib. http://bigdatasciencebootcamp.com/posts/Part_3/clustering_news.html.

[5] Word2vector package. https://code.google.com/p/word2vec/.

[6] STRING database. http://string-db.org/, Feb. 2015.

[7] S. Agarwal, Y. Furukawa, N. Snavely, I. Simon, B. Curless, S. M. Seitz, and R. Szeliski. Building rome in a day. *Communications of the ACM*, 54(10):105–112, 2011.

[8] C. C. Aggarwal and C. Zhai. *Mining text data.* Springer Science & Business Media, 2012.

[9] R. Ahas et al. Using mobile positioning data to model locations meaningful to users of mobile phones. *Journal of Urban Technology*, 17(1):3–27, 2010.

[10] S. Aridhi, A. Montresor, and Y. Velegrakis. Bladyg: A novel block-centric framework for the analysis of large dynamic graphs. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, pages 39–42. ACM, 2016.

[11] A. Asuncion and D. Newman. Uci machine learning repository, 2007.

[12] K. Avrachenkov, N. Litvak, V. Medyanikov, and M. Sokol. Alpha current flow betweenness centrality. In *Algorithms and Models for the Web Graph*, pages 106–117. Springer, 2013.

[13] R. Baraglia, P. Dazzi, B. Guidi, and L. Ricci. Godel: Delaunay overlays in p2p networks via gossip. In *IEEE 12th International Conference on Peer-to-Peer Computing (P2P)*, pages 1–12. IEEE, 2012.

[14] R. Baraglia, P. Dazzi, M. Mordacchini, and L. Ricci. A peer-to-peer recommender system for self-emerging user communities based on gossip overlays. *Journal of Computer and System Sciences*, 79(2):291–308, 2013.

[15] R. Baraglia, P. Dazzi, M. Mordacchini, L. Ricci, and L. Alessi. Group: A gossip based building community protocol. In *Smart Spaces and Next Generation Wired/Wireless Networking*, pages 496–507. Springer Berlin Heidelberg, 2011.

[16] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 16–24. ACM, 2008.

[17] H. Becker et al. Beyond trending topics: Real-world event identification on twitter. In *Proc. of ICWSM*, 2011.

[18] F. Beil et al. Frequent term-based text clustering. In *Proc. of ACM SIGKDD*, 2002.

[19] S. Bennett. Visually explore twitter hashtags and their relationships with hashtagify, 2012.

[20] M. Bertolucci, E. Carlini, P. Dazzi, A. Lulli, and L. Ricci. **Static and Dynamic Big Data Partitioning on Apache Spark**. In *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, 1-4 September 2015, Edinburgh, Scotland, UK*, pages 489–498, 2015.

[21] M. Bertolucci, A. Lulli, and L. Ricci. **Current flow betweenness centrality with Apache Spark**. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2016.

[22] S. Bhagat, G. Cormode, B. Krishnamurthy, and D. Srivastava. Class-based graph anonymization for social network data. *Proceedings of the VLDB Endowment*, 2(1):766–777, 2009.

[23] R. Bosagh-Zadeh and A. Goel. Dimension independent similarity computation. In *Journal of Machine Learning Research*, 2012.

[24] E. Bozzo and M. Franceschet. Approximations of the generalized inverse of the graph laplacian matrix. *Internet Mathematics*, 8(4):456–481, 2012.

[25] E. Bozzo and M. Franceschet. Resistance distance, closeness, and betweenness. *Social Networks*, 35(3):460–469, 2013.

[26] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[27] U. Brandes and D. Fleischer. Centrality measures based on current flow. In *Lecture Notes in Computer Science*, volume 3404, pages 533–544, 2005.

[28] U. Brandes and C. Pich. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, 17(07):2303–2318, 2007.

[29] S. Brin and L. Page. Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer networks*, 56(18):3825–3833, 2012.

[30] M. J. Brzozowski, T. Hogg, and G. Szabo. Friends and foes: ideological social networking. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 817–820. ACM, 2008.

[31] A. Buluç and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications. *International Journal of High Performance Computing Applications*, page 1094342011403516, 2011.

[32] F. Calabrese et al. Real-time urban monitoring using cell phones: A case study in rome. *Intelligent Transportation Systems, IEEE Transactions on*, 12(1):141–151, 2011.

[33] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink^{TM}: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 2015.

[34] E. Carlini, M. Coppola, P. Dazzi, D. Laforenza, S. Martinelli, and L. Ricci. Service and resource discovery supports over p2p overlays. In *Ultra Modern Telecommunications & Workshops, 2009. ICUMT'09. International Conference on*, pages 1–8. IEEE, 2009.

[35] E. Carlini, P. Dazzi, A. Esposito, A. Lulli, and L. Ricci. **Balanced Graph Partitioning with Apache Spark**. In *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part I*, pages 129–140, 2014.

[36] E. Carlini, P. Dazzi, A. Lulli, and L. Ricci. **Layered Thinking in Vertex Centric Computations**. *ERCIM News*, 2015(102), 2015.

[37] E. Carlini, P. Dazzi, A. Lulli, and L. Ricci. **Distributed graph processing: an approach based on overlay composition**. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, pages 1912–1917, 2016.

[38] E. Carlini, P. Dazzi, M. Mordacchini, A. Lulli, and L. Ricci. **Community Discovery for Interest Management in DVEs: A Case Study**. In *Euro-Par 2015: Parallel Processing Workshops - Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers*, pages 273–285, 2015.

[39] E. Carlini, A. Lulli, and L. Ricci. **Dragon: Multidimensional range queries on distributed aggregation trees**. *Future Generation Comp. Syst.*, 55:101–115, 2016.

[40] E. Carlini, A. Lulli, and L. Ricci. **TRACE: generating traces from mobility models for Distributed Virtual Environments**. In *Euro-Par 2016: Parallel Processing Workshops - Euro-Par 2016 International Workshops, Grenoble, France, August 22-25, 2016, Revised Selected Papers, Part I*, pages 129–140, 2016.

[41] E. Carlini, A. Lulli, and L. Ricci. **TRACE: Generation and Analysis of Mobility Traces for Distributed Virtual Environments**. *Concurrency and Computation: Practice and Experience*, 2016. submitted.

[42] E. Carlini, L. Ricci, and M. Coppola. Reducing server load in mmog via p2p gossip. In *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games*, page 11. IEEE Press, 2012.

[43] M. Ceccarello, A. Pietracaprina, G. Pucci, and E. Upfal. Space and time efficient parallel graph decomposition, clustering, and diameter approximation. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 182–191. ACM, 2015.

[44] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

[45] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 85–98. ACM, 2012.

[46] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[47] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science & Engineering*, 11(4):29–41, 2009.

[48] B.-R. Dai et al. Efficient map/reduce-based dbscan algorithm with optimized data partition. In *Cloud Computing, IEEE 5th International Conference on*, pages 59–66. IEEE, 2012.

[49] P. Dazzi, P. Felber, L. Leonini, M. Mordacchini, R. Perego, M. Rajman, and É. Rivière. Peer-to-peer clustering of web-browsing users. *Proc. LSDS-IR*, pages 71–78, 2009.

[50] E. De Jonge, M. van Pelt, and M. Roos. Time patterns, geospatial clustering and mobility statistics based on mobile phone network data. In *Paper for the Federal Committee on Statistical Methodology research conference, Washington, USA*, 2012.

[51] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), 2008.

[52] T. Debatty et al. Building k-nn graphs from large text data. In *Proc. of IEEE BigData*, 2014.

[53] T. Debatty et al. Scalable graph building from text data. In *Proc. ACM BigMine*, 2014.

[54] E. e. a. Deelman. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[55] U. Demšar, O. Špatenková, and K. Virrantaus. Identifying critical locations in a spatial network with graph theory. *Transactions in GIS*, 12(1):61–82, 2008.

[56] B. Desgraupes. Clustering indices. *University of Paris Ouest*, 2013.

[57] P. Deville et al. Dynamic population mapping using mobile phone data. *Proceedings of the National Academy of Sciences*, 111(45):15888–15893, 2014.

[58] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[59] D. Dominguez-Sal, N. Martinez-Bazan, V. Muntes-Mulero, P. Baleta, and J. L. Larriba-Pey. A discussion on the design of graph database benchmarks. In *Performance Evaluation, Measurement and Characterization of Complex Systems*, pages 25–40. Springer, 2011.

[60] W. Dong et al. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proc. of ACM WWW*, 2011.

[61] P. G. Doyle and J. L. Snell. Random walks and electric networks, 2006.

[62] M. Efron. Hashtag retrieval in a microblogging environment. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 787–788. ACM, 2010.

[63] M. Ester et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, pages 226–231, 1996.

[64] V. Etter et al. Where to go from here? mobility prediction from instantaneous information. *Pervasive and Mobile Computing*, 9(6):784–797, 2013.

[65] T. Falkowski et al. Dengraph: A density-based community detection algorithm. In *Web Intelligence, IEEE/WIC/ACM International Conference on*, pages 112–115. IEEE, 2007.

[66] X. Feng, L. Chang, X. Lin, L. Qin, and W. Zhang. Computing connected components with linear communication cost in pregel-like systems. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 85–96, 2016.

[67] M. Filippone. Dealing with non-metric dissimilarities in fuzzy central clustering algorithms. *International Journal of Approximate Reasoning*, 50(2):363–384, 2009.

[68] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.

[69] B. Fortz and M. Thorup. Internet traffic engineering by optimizing ospf weights. In *INFOCOM 2000. Nineteenth annual joint conference of the IEEE computer and communications societies. Proceedings. IEEE*, volume 2, pages 519–528. IEEE, 2000.

[70] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

[71] B. Furletti et al. Use of mobile phone data to estimate mobility flows. measuring urban population and inter-city mobility using big data in an integrated approach. In *Proceedings of the 47th Meeting of the Italian Statistical Society*, 2014.

[72] L. Gabrielli et al. City users' classification with mobile phone data. In *Big Data, 2015 IEEE International Conference on*, pages 1007–1012. IEEE, 2015.

[73] J. Gan and Y. Tao. Dbscan revisited: mis-claim, un-fixability, and approximation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 519–530. ACM, 2015.

[74] F. Godin, V. Slavkovikj, W. De Neve, B. Schrauwen, and R. Van de Walle. Using topic models for twitter hashtag recommendation. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 593–596. ACM, 2013.

[75] J. E. Gonzalez et al. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[76] J. E. Gonzalez et al. Graphx: Graph processing in a distributed dataflow framework. In *(OSDI 14)*, pages 599–613, 2014.

[77] J. E. Gonzalez et al. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, 2014.

[78] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[79] M. Haklay and P. Weber. Openstreetmap: User-generated street maps. *Pervasive Computing, IEEE*, 7(4):12–18, 2008.

[80] Y. He et al. Mr-dbscan: An efficient parallel density-based clustering algorithm using mapreduce. In *Parallel and Distributed Systems, 2011 IEEE International Conference on*, pages 473–480. IEEE, 2011.

[81] B. Hendrickson and R. W. Leland. A multi-level algorithm for partitioning graphs. *SC*, 95:28, 1995.

[82] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58(301):13–30, 1963.

[83] W. Hoeffding. Probabilty inequalities for sums of bounded random variales. *Journal of american Statistical Association*, 58:13–30, 1963.

[84] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 349–362. ACM, 2012.

[85] J. Hopcroft and R. Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6), June 1973.

[86] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.

[87] M. A. Jaro. Probabilistic linkage of large public health data files. *Statistics in medicine*, 14(5-7), 1995.

[88] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems (TOCS)*, 23(3):219–252, 2005.

[89] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8, 2007.

[90] J. Ji et al. Super-bit locality-sensitive hashing. In *Proc. of NIPS*, 2012.

[91] K. a. Jiang. Generalizing k-betweenness centrality using short paths and a parallel multithreaded implementation. In *ICPP'09*, pages 542–549. IEEE, 2009.

[92] D. B. Johnson and P. Metaxas. Connected components in o(log(3/2 n)) parallel time for the crew pram. *journal of computer and system sciences*, 54(2):227–242, 1997.

[93] H. Kardes, S. Agrawal, X. Wang, and A. Sun. Ccf: Fast and scalable connected component computation in mapreduce. In *Computing, Networking and Communications (ICNC), 2014 International Conference on*, pages 994–998. IEEE, 2014.

[94] D. R. Karger, N. Nisan, and M. Parnas. Fast connected components algorithms for the erew pram. In *Proc. of the 4th Symposium on Parallel algorithms and architectures*, pages 373–381. ACM, 1992.

[95] G. Karypis and E.-H. S. Han. Fast supervised dimensionality reduction algorithm with applications to document categorization & retrieval. In *Proc. of ACM CIKM*, 2000.

[96] A. Kavalionak, E. Carlini, A. Lulli, G. Amato, C. Gennaro, C. Meghini, and L. Ricci. **A prediction-based distributed tracking protocol for video surveillance**. In *Networking, Sensing and Control (ICNSC), 2017 IEEE International Conference on.* IEEE, 2017. submitted.

[97] A. Kermarrec, E. L. Merrer, B. Sericola, and G. Trédan. Second order centrality: Distributed assessment of nodes criticity in complex networks. *Computer Communications*, 34(5):619–628, 2011.

[98] Y. Kim, K. Shim, M.-S. Kim, and J. S. Lee. Dbcure-mr: an efficient density-based clustering algorithm for large data using mapreduce. *Information Systems*, 42:15–35, 2014.

[99] R. Kiveris, S. Lattanzi, V. Mirrokni, V. Rastogi, and S. Vassilvitskii. Connected components in mapreduce and beyond. In *Proc. of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[100] R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In *Link mining: models, algorithms, and applications*, pages 337–357. Springer, 2010.

[101] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10: Proc. of the 19th intl. conference on World wide web*, pages 591–600, New York, 2010. ACM.

[102] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[103] K. A. Lehmann and M. Kaufmann. Decentralized algorithms for evaluating centrality in complex networks. Technical Report WSI-2003-10, Department of Computer Science, Michigan State University, Wilhelm-Schickard-Institut, October 2003.

[104] J. Leskovec, L. A. Adamic, and B. A. Huberman. The dynamics of viral marketing. *ACM Transactions on the Web (TWEB)*, 1(1):5, 2007.

[105] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[106] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[107] J. Leskovec and R. Sosič. Snap.py: SNAP for Python, a general purpose network analysis and graph mining tool in Python. `http://snap.stanford.edu/snappy`, June 2014.

[108] H.-G. Li, G.-Q. Wu, X.-G. Hu, J. Zhang, L. Li, and X. Wu. K-means clustering with bagging and mapreduce. In *System Sciences (HICSS), 2011 44th Hawaii International Conference on*, pages 1–8. IEEE, 2011.

[109] F. Lin and W. W. Cohen. A very fast method for clustering big text datasets. In *ECAI*, pages 303–308, 2010.

[110] Y. Liu et al. Understanding of internal clustering validation measures. In *Proc. of IEEE ICDM*, 2010.

[111] S. P. Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2), 1982.

[112] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[113] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

[114] A. Lulli, E. Carlini, P. Dazzi, and L. Ricci. **TELOS: An Approach for Distributed Graph Processing Based on Overlay Composition**. *Scalable Computing: Practice and Experience*, 2016. submitted.

[115] A. Lulli, P. Dazzi, L. Ricci, and E. Carlini. **A Multi-layer Framework for Graph Processing via Overlay Composition**. In *Euro-Par 2015: Parallel Processing Workshops - Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers*, pages 515–527, 2015.

[116] A. Lulli, T. Debatty, M. Dell'Amico, P. Michiardi, and L. Ricci. Scalable k-nn based text clustering. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 958–963. IEEE, 2015.

[117] A. Lulli, T. Debatty, M. Dell'Amico, P. Michiardi, and L. Ricci. **Scalable k-NN based text clustering**. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 958–963, 2015.

[118] A. Lulli, M. Dell'Amico, P. Michiardi, and L. Ricci. **NG-DBSCAN: a Scalable, Approximate Density-Based Clustering Algorithm for Arbitrary Similarity Metrics**. *Proceedings of the VLDB Endowment*, 10(3), 2016.

[119] A. Lulli, L. Gabrielli, P. Dazzi, M. Dell'Amico, P. Michiardi, M. Nanni, and L. Ricci. **Improving Population Estimation From Mobile Calls: a Clustering Approach**. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 1097–1102. IEEE, 2016.

[120] A. Lulli, L. Gabrielli, P. Dazzi, M. Dell'Amico, P. Michiardi, M. Nanni, and L. Ricci. **Scalable and flexible clustering solutions for mobile phone based population indicators**. *International Journal of Data Science and Analytics*, 2016. submitted.

[121] A. Lulli, L. Ricci, E. Carlini, and P. Dazzi. **Distributed Current Flow Betweenness Centrality**. In *2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems, Cambridge, MA, USA, September 21-25, 2015*, pages 71–80, 2015.

[122] A. Lulli, L. Ricci, E. Carlini, P. Dazzi, and C. Lucchese. **Cracker: Crumbling large graphs into connected components**. In *2015 IEEE Symposium on Computers and Communication, ISCC 2015, Larnaca, Cyprus, July 6-9, 2015*, pages 574–581, 2015.

[123] A. Lulli, L. Ricci, E. Carlini, P. Dazzi, and C. Lucchese. **Fast Connected Components Computation in Large Graphs by Vertex Pruning**. *IEEE Transactions on Parallel and Distributed systems*, 22(6):931–945, 2016.

[124] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[125] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[126] L. Massoulié, E. Le Merrer, A.-M. Kermarrec, and A. Ganesh. Peer counting and sampling in overlay networks: random walk methods. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 123–132. ACM, 2006.

[127] Y. Matsuo and M. Ishizuka. Keyword extraction from a single document using word co-occurrence statistical information. *International Journal on Artificial Intelligence Tools*, 13(1), 2004.

[128] J. J. McAuley and J. Leskovec. From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews. In *Proceedings of the 22nd international conference on World Wide Web*, pages 897–908. ACM, 2013.

[129] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[130] R. R. McCune et al. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015.

[131] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015.

[132] F. McSherry et al. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems*, 2015.

[133] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer. Graph structure in the web—revisited: a trick of the heavy tail. In *Proceedings of the 23rd conference on World wide web*, pages 427–432. International World Wide Web Conferences Steering Committee, 2014.

[134] T. Mikolov et al. Distributed representations of words and phrases and their compositionality. In *Proc. of NIPS*, 2013.

[135] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. In *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)*, San Diego, CA, October 2007.

[136] A. Montresor, F. De Pellegrini, and D. Miorandi. Distributed k-core decomposition. *IEEE Transactions on parallel and distributed systems*, 24(2):288–300, 2013.

[137] A. Montresor and M. Jelasity. Peersim: A scalable p2p simulator. In *Peer-to-Peer Computing, 2009, Ninth International Conference on*, pages 99–100. IEEE, 2009.

[138] M. Mordacchini, P. Dazzi, G. Tolomei, R. Baraglia, F. Silvestri, and S. Orlando. Challenges in designing an interest-based distributed aggregation of users in p2p systems. In *Ultra Modern Telecommunications & Workshops, 2009. ICUMT'09. International Conference on*, pages 1–8. IEEE, 2009.

[139] M. Newman. A measure of betweeness centrality based on random walks. *Social Networks*, 27, 2005.

[140] M. E. Newman. A measure of betweenness centrality based on random walks. *Social networks*, 27(1):39–54, 2005.

[141] M. E. Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical review E*, 74(3), 2006.

[142] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[143] M. M. A. Patwary et al. Pardicle: parallel approximate density-based clustering. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 560–571. IEEE Press, 2014.

[144] F. Pedregosa et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[145] N. Pervin, T. Q. Phan, A. Datta, H. Takeda, and F. Toriumi. Hashtag popularity on twitter: Analyzing co-occurrence of multiple hashtags. In *International Conference on Social Computing and Social Media*, pages 169–182. Springer, 2015.

[146] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, volume 10, pages 1–14, 2010.

[147] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin. Scalable big graph processing in mapreduce. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 827–838. ACM, 2014.

[148] F. Rahimian, A. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. In *Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on*, pages 51–60, Sept 2013.

[149] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. In *Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on*, pages 51–60. IEEE, 2013.

[150] A. Rajaraman et al. *Mining of massive datasets*, volume 77. Cambridge University Press Cambridge, 2012.

[151] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. Das Sarma. Finding connected components in map-reduce in logarithmic rounds. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 50–61. IEEE, 2013.

[152] C. Ratti et al. *Mobile landscapes: Graz in real time.* Springer, 2007.

[153] M. Riondato and E. M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. In *Proceedings of the 7th ACM Conference on Web Search and Data Mining, WSDM*, volume 14, 2013.

[154] L. M. Rocha, F. A. Cappabianco, and A. X. Falcão. Data clustering as an optimum-path forest problem with applications in image analysis. *International Journal of Imaging Systems and Technology*, 19(2):50–68, 2009.

[155] P. J. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 1987.

[156] M. N. Sadat, S. Ahmed, and M. T. Mohiuddin. Mining the social web to analyze the impact of social media on socialization. In *Informatics, Electronics & Vision (ICIEV), 2014 International Conference on*, pages 1–6. IEEE, 2014.

[157] S. Salihoglu and J. Widom. Gps: a graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.

[158] S. Salihoglu and J. Widom. Help: High-level primitives for large-scale graph processing. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6. ACM, 2014.

[159] S. Salihoglu and J. Widom. Optimizing graph algorithms on pregel-like systems. *Proc. of the VLDB Endowment*, 7(7), 2014.

[160] D. A. Schult and P. Swart. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conferences (SciPy 2008)*, volume 2008, pages 11–16, 2008.

[161] T. Seidl, B. Boden, and S. Fries. Cc-mr–finding connected components in huge graphs with mapreduce. In *Machine Learning and Knowledge Discovery in Databases*, pages 458–473. Springer, 2012.

[162] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 721–726. IEEE, 2010.

[163] Y. Shen et al. Mr-triage: Scalable multi-criteria clustering for big data security intelligence applications. In *Proc. of IEEE BigData*, 2014.

[164] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[165] M. Steinbach et al. A comparison of document clustering techniques. In *Proc. of KDD workshop on text mining*, 2000.

[166] I. Stoica. Trends and challenges in big data processing. *Proceedings of the VLDB Endowment*, 9(13), 2016.

[167] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614. ACM, 2011.

[168] M. Terada, T. Nagata, and M. Kobayashi. Population estimation technology for mobile spatial statistics. *NTT DOCOMO Techn. J*, 14:10–15, 2013.

[169] O. Thonnard and M. Dacier. A strategic analysis of spam botnets operations. In *Proc. of ACM CEAS*, 2011.

[170] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[171] Z. Toh and W. Wang. Dlirec: Aspect term extraction and term polarity classification system. In *Proc. of SemEval*, 2014.

[172] T. N. Tran et al. Knn-kernel density-based clustering for high-dimensional multivariate data. *Computational Statistics & Data Analysis*, 51(2):513–525, 2006.

[173] O. Tsur and A. Rappoport. What's in a hashtag?: content based prediction of the spread of ideas in microblogging communities. In *Proceedings of the fifth ACM international conference on Web search and data mining*, pages 643–652. ACM, 2012.

[174] G. M. Turner-McGrievy and M. W. Beets. Tweet for health: using an online social network to examine temporal trends in weight loss-related posts. *Translational behavioral medicine*, 5(2):160–166, 2015.

[175] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[176] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[177] F. Vandin, A. Papoutsaki, B. J. Raphael, and E. Upfal. Genome-wide survival analysis of somatic mutations in cancer. In *Research in Computational Molecular Biology*, pages 285–286. Springer, 2013.

[178] A. Verma, X. Llorà, D. E. Goldberg, and R. H. Campbell. Scaling genetic algorithms using mapreduce. In *2009 Ninth International Conference on Intelligent Systems Design and Applications*, pages 13–18. IEEE, 2009.

[179] S. Voulgaris, D. Gavidia, and M. Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.

[180] H. Wang, Y. Shen, L. Wang, K. Zhufeng, W. Wang, and C. Cheng. Large-scale multimedia data mining using mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 287–292. IEEE, 2012.

[181] K. Wehmuth and A. Ziviani. DACCER: distributed assessment of the closeness centrality ranking in complex networks. *Computer Networks*, 57(13):2536–2548, 2013.

[182] B. Welton et al. Mr. scan: Extreme scale density-based clustering using a tree-based network of gpgpu nodes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 84. ACM, 2013.

[183] W. E. Winkler. The state of record linkage and current research problems. In *Statistical Research Division, US Census Bureau*, 1999.

[184] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. Sync or async: Time to fuse for distributed graph-parallel computation. *ACM SIGPLAN Notices*, 50(8):194–204, 2015.

[185] R. S. Xin, D. Crankshaw, A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: Unifying data-parallel and graph-parallel analytics. *arXiv preprint arXiv:1402.2394*, 2014.

[186] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[187] X. Xu, J. Jäger, and H.-P. Kriegel. A fast parallel clustering algorithm for large spatial databases. In *High Performance Data Mining*, pages 263–290. Springer, 1999.

[188] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992, 2014.

[189] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1307–1317. ACM, 2015.

[190] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment*, 7(14):1821–1832, 2014.

[191] W.-S. Yang, J.-B. Dia, H.-C. Cheng, and H.-T. Lin. Mining social networks for targeted advertising. In *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, volume 6, pages 137a–137a. IEEE, 2006.

[192] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[193] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proc. of the 2nd USENIX conf. on Hot topics in cloud computing*, pages 10–10, 2010.

[194] Y.-m. Zhang et al. Fast knn graph construction with locality sensitive hashing. In *Proc. of ECML PKDD*, 2013.

[195] Y. Zhao and G. Karypis. Empirical and theoretical comparisons of selected criterion functions for document clustering. *Journal of Machine Learning*, 55(3), 2004.

[196] S. Zhou et al. A neighborhood-based clustering algorithm. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 361–371. 2005.