Università degli Studi di Padova

Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Master's Thesis

Efficient MapReduce Algorithms for computing the diameter of very large graphs.

Advisors:
Prof. Andrea Alberto Pietracaprina

Prof. Geppino Pucci

Student:
Matteo Ceccarello

Academic year 2012-2013

October, 8 2013

Abstract

Graph data structures are ubiquitous in computer science: the Web, the Internet and social networks are only a few examples. Computer science is not the only field in which graphs are used. In biology, graphs are used to model interactions between proteins or metabolites, or for sequencing DNA. Sensors networks that are used, among other things, in earth science are described as graphs.

The graphs that are used in this wide variety of contexts share a common property: they tend to be huge objects, with millions or even billions of nodes. When dealing with objects of this magnitude, using a single machine to extract data is inefficient or simply not feasible, because of hardware limitations. To overcome these limitations and have a boost in performance, parallel algorithms on distributed architectures need to be developed.

MapReduce is a model of computation developed to process large amounts of data in parallel using clusters of commodity machines. Originally developed to process large text files, recently it has been used in the context of graph algorithms, scientific calculus and data mining.

One of the most interesting properties of graphs is the *diameter*, the longest shortest path in the graph. Questions about the evolution of the Web, the degrees of separation between any two people in a social network or the maximum number of reactions to obtain a metabolite from another one can be answered using the diameter.

In this thesis we study the adaptation of several existing algorithms for the diameter computation to the MapReduce model and we develop a novel approach based on the reduction of the graph size.

Ringraziamenti

Desidero ringraziare i miei genitori, Liliana e Ermanno, che in questi anni mi hanno sempre supportato e incoraggiato nel mio percorso di studio e non solo.

Ringrazio Martina, che condivide con me tante passioni, dalla musica al tango, e che pazientemente ha sopportato i miei deliri su grafi e algoritmi.

Ringrazio Marco, l'amico di una vita, che mi ha sempre capito e che suona e cucina divinamente.

Ringrazio poi tutte le persone che ho incontrato finora e con cui ho condiviso esperienze importanti: i compagni di corso, di musica e di ballo.

Infine ringrazio i miei relatori, i professori Pietracaprina e Pucci, per la guida nel percorso di ricerca.

CONTENTS

LIST OF ALGORITHMS

1 INTRODUCTION

Graph data structures are ubiquitous in computer science. The most famous examples are perhaps the Internet and the World Wide Web. Recently we have seen so-called social graphs become widespread: Facebook, Twitter, LinkedIn are just a few of them.

Graphs are not limited to computer science, however. A graph can be used to model a road network, for instance. Or it can represent a network of sensors spread over a geographical area [VKR+05, HM06]. Graph data structures often arise in biology, where they are used to represent interactions between proteins or metabolites [MRK04, PPP05].

All these diverse graphs share a common property: they tend to be really large objects. In 2012 the Facebook graph had approximately 721 million nodes and 69 billion edges [BBR+12]. The World Wide Web graph is even bigger: it contains 3 billion nodes and 50 billion arcs, but these are just lower bounds obtained by search engines [BV03].

Since graphs are so widespread, it is important to study their properties in order to better understand the domain they represent. An interesting property of graphs is the diameter, i.e. the maximum distance between any two nodes.

The diameter of a graph is an interesting property in network optimization problems. For instance, a flight company may be interested in minimizing the number of flights that are needed to reach any airport from any other one. If we represent the network of airports as a graph, where each airport is a node and flights are edges, the problem of minimizing the number of flights that are needed to reach any airport from any other one corresponds to minimizing the diameter of the graph.

Many graphs are subject to the *small world* phenomenon [WS98]. This occurs when a graph with n nodes has a diameter of the order of $\log n$. Historically this property has been observed in synthetic, random graphs [BR04, Bur73, Bol81]. Recently, the small world property has been spotted in real world graphs, such as the Facebook graph [BBR+12].

In the case of the Web graph, the diameter indicates how quickly we can reach any page from any other page. Studying the evolution of the diameter over time, using different snapshots of the Web, allows us to understand how the Web evolves.

In biological networks, for example in metabolic networks, the diameter indicates how many reactions are needed to produce any metabolite from any other metabolite [JS08].

Given the importance of the diameter as a metric of a graph, efficient algorithms for its computation are needed. The textbook algorithm involves the solution of the All Pairs Shortest Paths problem [CLRS09], whose complexity is prohibitive for large graphs.

In this thesis we analyze several approaches to the diameter computation problem and propose a novel approach based on the graph size reduction.

1.1 OVERVIEW

This thesis describes different approaches to the diameter problem. First we will give some graph definitions that will be used throughout the thesis. We will also describe the MapReduce paradigm and the associated computational model.

The first approach described (chapter 2) is based on the classical textbook algorithm: solving the All Pairs Shortest Paths problem. We analyze the problem and the algorithm in the MapReduce paradigm. This approach gives us the exact value of the diameter. However it requires multiplying big matrices, an operation that can be too costly in many situations.

As it often happens when dealing with problems whose solution takes a long time to compute, randomized algorithms come to help. Chapter 3 analyzes four such algorithms. They all follow the same algorithmic pattern and they all return an approximation of the result.

Getting the answer to our problem directly is surely nice, but graphs can become really big. What would happen if we reduced the size of the graph before computing? Chapters 4, 5 and 6 are devoted to answering to this question.

Chapter 4 describes the modular decomposition, a technique that can reduce a graph's size without changing the diameter. The drawback of this technique is that the actual graph size reductions we found experimentally are not significant.

Continuing along this path, in chapters 5 and 6 we introduce the star decomposition and ball decomposition. With these two techniques we give up the requirement of obtaining a graph with the exact same diameter and we get in turn great size reductions. Moreover we prove that the diameter reduction is bounded and that experimentally is often much better than the theoretical bound.

We conclude with appendices A and B by describing the datasets and the software used in the experiments.

1.2 COMMON GRAPH DEFINITIONS AND NOTATION

An undirected graph $G = (V, E)$ is made up by two sets: V, the set of vertices, and $E \subset V\text{x}V$, the set of edges. In this thesis, every time we write $G = (V, E)$ we refer to an undirected graph.

A directed graph $G = (V, A)$ is made up by the vertex set V and the set of arcs $A \subset V\text{x}V$. In this thesis the notation $G = (V, A)$ refers always to a directed graph.

Unless otherwise noted, the number of nodes of a graph is n. This is also called the *cardinality* of the graph. The number of edges is denoted as m.

We will now define the two graph attributes that are the focus of this thesis.

Definition 1 (Diameter). *The* diameter *of a graph is the length of the longest shortest path of a graph. It is denoted with d.*

Definition 2 (Effective diameter). *Given a graph G and a number $\alpha \in [0, 1]$, the* effective diameter *of G at α is the smallest value d_α such that for at least a fraction α of the pairs of nodes (u, v) that can reach each other, the distance between u and v is d_α.*

More formally, we can define N_α as the number of pairs of vertices of the graph that can reach each other in at most d_α hops. Let N be the number pairs of vertices in G that can reach each other, then d_α is the effective diameter if

$$\frac{N_\alpha}{N} \geq \alpha$$

1.3 THE MAP-REDUCE MODEL

MapReduce is model of computation developed to process large amounts of data in parallel using clusters of commodity machines [DG08].

Algorithms in this model are expresses in terms of *map* and *reduce* functions. This approach is reminiscent of map and reduce functions found in functional languages such as Lisp or Haskell. These functions operate on key-value pairs:

- A map function takes as input a pair of type $< K_{in}, V_{in} >$ and outputs a finite number (possibly zero) of key value pairs of type $< K_{out}, V_{out} >$.

- A reduce function takes as input a key K and the list of all the values associated with that key. It then reduces this list of values to a possibly smaller one.

These functions can be run in parallel on different parts of the input. For the map function, in fact, the set of key-value pairs produced depends only on the input pair. Similarly, for the reduce function, the result depends only on the input key and list of values. Hence the input can be divided in several *splits* that are processed in parallel, possibly on different machines.

A computation in the MapReduce model is structured in three phases:

1. Map phase. The input is fed to the map function that processes each key-value pair.

2. Shuffle phase. The results of map functions are sent across the network. All the values associated with the same key are sent to the same machine, ready to be processed in the next step.

3. Reduce phase. After all the values associated with a key are grouped together, the reduce function processes them to produce the output.

Most algorithms are too complex to be expressed by only one map and reduce pair of functions. The computation is then organized in *rounds*, where each round is made up by a map and reduce function. The output of the reduce function of a round is fed as input to the map function of the next round.

1.3.1 *Computational model*

The original paper [DG08] does not provide a computational model to analyze the time and space complexity of algorithms expressed in MapReduce. Several models of computation for MapReduce have been proposed, like in [GSZ11, KSV10]. However they are somewhat incomplete, in that they do not take into account the space limitations of single computational nodes.

A more complete model is given in [PPR+12]. The model is defined in terms of two parameters, m and M and is called $MR(m, M)$. Algorithms in this model are specified as a series of rounds. Round r takes as input a multiset W_r and outputs two multisets: W_{r+1}, that will be the input of the next round, and O_r, that is the output of the round.

The parameters m and M specify the constraints on the memory that are imposed on each round of a MapReduce algorithm. The parameter m stands for the local memory of each computational node, whereas M is the aggregated memory of all the computational nodes. The model imposes that in each round the memory used at each computational node is $O(m)$ and the memory used across all computational nodes is $O(M)$. The only requirement for map and reduce functions is that they run in time polynomial in the input size n.

The complexity of an algorithm in MapReduce is the number of rounds that it executes in the worst case, as a function of the input size n and the memory parameters m and M.

For a more in-depth discussion of the model, we refer the reader to [PPR+12].

1.3.2 *The Cogroup primitive*

The original MapReduce model embodies only two primitives: the map and the reduce functions. In the algorithms described in this thesis, another primitive is useful: the *cogroup* primitive. Given two datasets of key value pairs, $< K, V_1 >$ and $< K, V_2 >$, the cogroup operation produces a third dataset of type $< K, ([V_1], [V_2]) >$, where the $[\quad]$ symbol denotes a list. Given a key k, the lists associated to it are made up by all the values of the two set of values V_1 and V_2 associated to k in the original datasets.

This operation can be easily simulated in a constant number of rounds in MapReduce. First, two map functions, one for each dataset, will output the key-value pairs of each dataset, where the value is marked as coming from the first or second dataset. Then a reduce function will gather all the marked values associated to a key. It will then create the output lists, putting the values coming from the original datasets in the appropriate list. Each reducer will receive a number of values equal to $n_1^k + n_2^k$, where n_1^k is the number of values associated to a key k in the first dataset and n_2^k is the number of values associated to the same key k in the second dataset. Since the reducer simply outputs the aggregation of the values, the output size is also $n_1^k + n_2^k$.

In the rest of the thesis we will assume that we have this primitive at our disposal, during the description of the algorithms.

1.3.3 *Implementations*

There are several open source MapReduce implementation, in a wide variety of programming languages. The most widely used is Hadoop, implemented in Java [Whi12]. The distributed filesystem it provides, HDFS, is used by many other frameworks. Among these there is Spark [ZCF+10], implemented in Scala. This framework has been used to perform the experiments of this thesis. The reason we preferred it over the more used Hadoop is that it runs faster for algorithms with more than one round, like the ones presented in this thesis. This performance advantage is due to the fact that Spark caches the results between subsequent rounds in memory. On the contrary, Hadoop writes data to the distributed filesystem after each round. For multi-round algorithms this is a serious performance issue. Finally, Spark is written in Scala, meaning that applications are much faster to write and require less code.

2 COMPUTING THE DIAMETER BY SOLVING ALL PAIRS SHORTEST PATHS

In this chapter we present an approach to diameter computation based on the solution of the All Pairs Shortest Path problem. If we compute all the shortest paths in a graph, the longest one will give us the diameter, by definition.

2.1 DEFINITIONS

In this section we present some definitions that we will need in what follows.

Definition 3. *Given a graph $G = (V, A)$ with n vertices, the incidence matrix I_G is the n x n matrix that has a 1 in position (i, j) if there is an edge between i and j and $+\infty$ otherwise.*

Definition 4 (Distance matrix)**.** *Given a graph $G = (V, A)$ with n vertices, the distance matrix D_G is the n x n matrix that in position (i, j) holds the length of the shortest path between nodes i and j.*

For unweighted graphs the value of each matrix element is the number of hops of the shortest path. If there is no path between two nodes then in the matrix the corresponding element will be $+\infty$.

2.2 SERIAL ALGORITHM FOR APSP

One method to compute the distance matrix is to repeatedly square the incidence matrix until there is no element that changes. The procedure is depicted in algorithm 1. The algorithm and the subsequent analysis are based on [JaJ97]

The matrix multiplication used in the algorithm is defined over the semiring $(\mathbb{N}, \min, +)$. The multiplication $C = A \cdot B$ of two nxn matrices is then defined as follows

$$C_{ij} = \min_{1 \le k \le n} \{A_{ik} + B_{kj}\}$$

We are now going to prove the correctness and running time of algorithm 1.

Theorem 1. *Let I_G be the incidence matrix of a graph $G = (V, A)$. For each $s > 0$, let $D^{(s)}$ be the matrix such that $D^{(s)}(i, j)$ is equal to the weight of the shortest path between i and j that contains at most s arcs. Then*

$$D_{ij}^{(s+1)} = \min_{1 \le h \le n} \{D_{ih}^{(s)} + I_{G_{hj}}\}$$

Algorithm 1: Algorithm to compute the distance matrix

```
Input: $I_G$, the incidence matrix of graph $G$
Output: $D_G$, the distance matrix of graph $G$
begin
    $D_{prev} \leftarrow I_G$
    $D \leftarrow I_G \cdot I_G$
    while $D \neq D_{prev}$ do
        $D_{prev} \leftarrow D$
        $D \leftarrow D \cdot D$
    end
    return $D$
end
```

is the matrix containing the weights of the shortest paths between each i and j with at most $s+1$ arcs.

Proof. The theorem is proved by induction. Let $D^{(0)}$ be the matrix defined as follows:

$$D^{(0)}_{ij} = \begin{cases} 0 & i = j \\ +\infty & i \neq j \end{cases}$$

This matrix is the zero-step distance matrix.

The base case is for $s = 0$. Being $+\infty$ the identity element for the min operation, we have that $D^{(0)}$ is the identity matrix for the product defined over the semiring $(\mathbb{N}, \min, +)$. Hence we have

$$D^{(1)} = D^{(0)} I_G = I_G$$

so $D^{(1)}_{ij}$ is equal to the weight of the arc connecting i and j: 1 if the vertices are adjacent and $+\infty$ if they are not.

Assume now that the induction hypothesis holds for every value less than or equal to s, with $s > 0$. We have that $D^{(s+1)}_{ij} = \min_{1 \leq h \leq n}\{D^{(s)}_{ih} + I_{G_{hj}}\}$. By the inductive hypothesis we know that $D^{(s)}_{ih}$ is the weight of the shortest path between i and h with at most s arcs. Hence $D^{(s)}_{ih} + I_{G_{hj}}$ is the weight of a path from i to j passing through h as the last step. By taking the minimum of all these sums we get the weight of the shortest path from i to j in $s+1$ steps. □

Theorem 2. *Algorithm 1 computes the distance matrix of a graph of diameter d with $\lceil \log_2 d \rceil$ matrix multiplications.*

Proof. The distance matrix D_G of a graph G is clearly I^d_G. To compute this matrix we can square I_G $\lceil \log_2 d \rceil$ times. So the running time of the algorithm is $O(\log n)$ if we consider the matrix multiplication a base operation. □

We are now going to describe a slightly different version of the distance matrix algorithm that will be useful later, when we will deal with the computation of the effective diameter.

Lemma 1. *Given a connected graph G, its distance matrix D_G has no element equal to $+\infty$.*

Thanks to this lemma, we can now develop the algorithm for distance matrix multiplication depicted in pseudocode 2. The basic idea of the algorithm is to repeatedly square the incidence matrix of the graph until no elements are equal to $+\infty$.

Algorithm 2: A different distance matrix algorithm.

```
Input: $I_G$, the incidence matrix of graph $G$
Output: $D_G$, the distance matrix of graph $G$
begin
    $D \leftarrow I_G$
    while There are elements of $D$ equals to $\infty$ do
        $D \leftarrow D \cdot D$
    end
    return $D$
end
```

Theorem 3. *Algorithm 2 computes the diameter of a graph with* $\log_2 d$ *matrix multiplications.*

Proof. As stated in lemma 1 the distance matrix of a connected graph has no element equal to ∞. If we repeatedly square the incidence matrix of G until we have no more ∞ elements we are basically computing the distance matrix. As we have seen before this matrix is I_G^d. Since at each iteration we perform a squaring, to obtain I_G^d we need $\log_2 d$ iterations. □

2.3 COMPUTING THE DIAMETER

In this section we will see how to use the distance matrix to compute the diameter of a graph G.

The distance matrix can be computed using algorithm 1. Recall that the element (i, j) in the distance matrix is the lenght of the shortest path between nodes i and j in the graph. Hence the maximum value in the matrix is the longest shortest path in the graph, i.e. its diameter. In algorithm 3 we depict a MapReduce algorithm to compute the diameter, given the distance matrix.

The basic idea of the algorithm is to divide the matrix into submatrices and look for the maximum in each submatrix. The diameter of the graph is the maximum of these maxima.

We will analyze the given algorithm in the $MR(m, M)$ model described in [PPR$^+$12]. In what follows, n is the number of elements in the matrix D_G rather than the number of nodes of the graph. Moreover, m is the local memory of each computational node and M is the aggregated memory of all nodes.

Algorithm 3: Computing the diameter of a graph from its distance matrix.

```
begin
    Split the matrix into smaller ones
    Compute the maximum of each submatrix
    Compute the global maximum with a prefix computation
end
```

The $\sqrt{n}$ x $\sqrt{n}$ input matrix D_G is divided into sub-matrices of size $\sqrt{m}$ x $\sqrt{m}$.

The algorithm proceeds in rounds, dealing with $K = M/m$ sub-matrices at a time. At the end of round $n/(m \cdot K) = n/M$ all maxima have been computed. The global maximum (i.e. the diameter) can be computed using a prefix computation.

Theorem 4. *The above algorithm finds the maximum element in a $\sqrt{n}$ x $\sqrt{n}$ matrix in*

$$O\left(\frac{n}{M} + \log_m \frac{n}{m}\right)$$

rounds in the $MR(m, M)$ model.

Proof. The algorithm complies with the memory constraints of $MR(m, M)$ [PPR+12], since each reducer deals with matrices whose size is linear in the local memory of each computational node. Moreover the at each round the memory requirements of the algorithm never exceeds the aggregate memory M.

The number of rounds used to compute the maximum for each submatrix is n/M. The final prefix computation, since it has to compute the maximum among n/m values, takes $O(\log_m n/m)$ rounds [PPR+12]. Combining these two results yields the total complexity. □

We observe that the algorithm executes in a constant number of rounds whenever $m = \Omega(n^\epsilon)$, with $\epsilon > 0$, and $M = \Omega(n)$.

To compute the distance matrix we can use one of the matrix multiplication algorithms described in [PPR+12]. The running time of the computation of the diameter, taking into account the matrix exponentiation and the maximum computation, is then

$$O\left(\log d\left(\frac{n^{3/2}}{M\sqrt{m}} + \log_m n\right) + \left(\frac{n}{M} + \log_m \frac{n}{m}\right)\right)$$

For suitable values of the parameters m and M the algorithm takes a logarithmic number of rounds to complete. More precisely, if

$$\begin{aligned} m &= \Omega(n^\epsilon), \text{ with } \epsilon > 0 \\ M\sqrt{m} &= \Omega(n^{3/2}) \\ M &= \Omega(n) \end{aligned}$$

we have that the algorithm completes in $O(\log d)$ rounds.

Algorithm 4: Algorithm to compute the effective diameter of a graph G.

```
Input: $I_G$, the incidence matrix of graph $G$
Output: $d_\alpha$, the effective diameter of $G$ at $\alpha$
begin
    $D \leftarrow I_G$
    while There are more than $n(1-\alpha)$ elements of $D$ equals to $\infty$ do
        $D \leftarrow D \cdot D$
    end
    $d_\alpha \leftarrow \max\{D\}$
    return $d_\alpha$
end
```

2.4 COMPUTING THE EFFECTIVE DIAMETER

In this section we describe a modification of the algorithm presented in above that can compute the *effective diameter* in a smaller number of rounds.

To compute the effective diameter the idea is to use algorithm 2 with a different stopping condition, based on the desired value of α. Instead of stopping the iterations when there are no elements equals to ∞, we stop the computation when there are no more than $n(1-\alpha)$ elements equals to ∞.

After we have computed I_G^s, the stopping condition is equivalent to say that there must be at least αn vertices that can reach each other in at most s hops. Hence we seek to compute $I_G^{d_\alpha}$, i.e. the matrix that contains the length of the shortest paths between nodes that can reach each other in at most d_α hops. The algorithm to compute such a matrix is presented in pseudocode 4.

The algorithm to find the maximum element in the matrix is a slightly different version than the one presented in pseudocode 3: elements that are equal to ∞ are ignored. This does not affect the asymptotic running time of the algorithm.

We are now going to prove the correctness and running time of algorithm 4.

Theorem 5. *Given a graph G and a value $\alpha \in [0,1]$, algorithm 4 correctly computes the effective diameter of G at α.*

Proof. Suppose that we have computed the matrix I_G^s for which the condition that there are no more than $\alpha(n-1)$ elements equals to infinity holds. According to theorem 1 the elements of this matrix are the lengths of the shortest paths shorter than s between the vertices of the graph. Recalling the definition of effective diameter at α, we have that among these elements there is the value d_α of the effective diameter. Since this value is the maximum among the elements of the matrix, the max procedure finds correctly the effective diameter. □

Theorem 6. *Given a graph G and a value $\alpha \in [0,1]$, algorithm 4 finds the effective diameter d_α of G at α in*

$$O\left(\log d_\alpha \left(\frac{n^{3/2}}{M\sqrt{m}} + \log_m n\right) + \left(\frac{n}{M} + \log_m \frac{n}{m}\right)\right)$$

in the $MR(M,m)$ model.

Proof. The time complexity of the matrix multiplication and the maximum algorithm is the same as in algorithm 2. The only thing that changes is the number of matrix multiplications performed. Since we have to compute $I_G^{d_\alpha}$ we have to perform $\log_2 d_\alpha$ matrix multiplications. Hence we have the claimed number of rounds. □

We can observe that for suitable values of the memory parameters the algorithm runs in a logarithmic number of rounds. More precisely, if

$$\begin{aligned} m &= \Omega(n^\epsilon), \text{ with } \epsilon > 0 \\ M\sqrt{m} &= \Omega(n^{3/2}) \\ M &= \Omega(n) \end{aligned}$$

then algorithm 4 runs in $O(\log d_\alpha)$ rounds. This is better than the round complexity of the algorithm that computes the diameter, since the effective diameter may be significantly smaller.

3 ITERATIVE, PROBABILISTIC ALGORITHMS FOR DIAMETER COMPUTATION

When data sets become too big to handle or when algorithms have too high time complexities, the usual approach is to resort to randomized or probabilistic algorithms. The graph diameter calculation problem makes no exception.

Here we study three algorithms, namely ANF [PGF02], HyperANF [BRV11] and HADI [KTA+08]. Finally, we propose an adaptation of HyperANF to the MapReduce framework.

All these algoritms share the same structure: the computation of the diameter is performed through the iterative computation of the *neighbourhood function*. The difference between the algorithms reside in how they keep track of the number of nodes that can be reached from a given node in a given number of steps. Moreover ANF and HyperANF are algorithms that run on a single machine, whereas HADI is a MapReduce algorithm.

In what follows we give an overview of the concepts shared by all algorithms, of the probabilistic algorithms used to count the number of reachable nodes and finally a description of the algorithms themselves.

3.1 COMMON DEFINITIONS

Given a graph (directed or undirected) $G = (V, E)$ the *neighbourhood function* $N_G(h)$ is the number of pairs that can reach each other in at most h steps, for all h.

More formally, we define the *ball* of radius r.

Definition 5. *The ball of radius r centered in vertex x is the set*

$$B(x, 0) = \{x\}$$

$$B(x, r) = \bigcup_{(x,y)\in E} B(y, r-1)$$

that is the number of vertices reachable from vertex x in at most r hops.

The neighbourhood function is then defined as follows

Definition 6. *The neighbourhood function is*

$$N(t) = \sum_{v \in V} |B(v, t)|$$

Closely related to this function is the *cumulative distribution function of distances* (*distance cdf* in short):

Definition 7 (Distance CDF). *Given a graph G the cumulative distribution function of distances is the fraction of reachable pairs at distance at most t:*

$$H_G(t) = \frac{N_G(t)}{\max_t N_G(t)}$$

Note that this function is monotone in t.

The diameter of the graph G can easily be obtained as the smallest t such that $H_G(t) = 1$. Similarly, the effective diameter at α is the smallest t such that $H_G(t) \geq \alpha$. If α is omitted, it is assumed to be $\alpha = 0.9$.

So far we have talked about relations that hold when the exact value of the neighbourhood function $N_G(t)$ is known for all t. However the algorithms that will be described later are probabilistic, in the sense that they provide an estimation of the neighbourhood function. Suppose then that we have an estimation $\hat{N}_G(t)$ of the neighbourhood function we can state the following theorem:

Theorem 7 ([BRV11]). *Assume $N_G(t)$ is known for each t with error ϵ and confidence $1 - \delta$:*

$$\Pr\left[\frac{\hat{N}_G(t)}{N_G(t)} \in [1-\epsilon, 1+\epsilon]\right] \geq 1 - \delta$$

The function

$$\hat{H}_G(t) = \hat{N}_G(t) / \max_t \hat{N}_G(t)$$

is an (almost) unbiased estimator for $H_G(t)$. Moreover, for a fixed sequence $t_0, t_1, \ldots, t_{k-1}$, for every ϵ and all $0 \leq i < k$ we have that the error on $\hat{H}_G(t_k)$ is 2ϵ and the confidence is $1 - (k+1)\delta$:

$$\Pr\left[\bigwedge_{i \in [0,k]} \frac{\hat{H}_G(t)}{H_G(t)} \in (1-2\epsilon, 1+2\epsilon)\right] \geq 1 - (k+1)\delta$$

Proof. If the neighbourhood function is known with error ϵ then we have

$$1 - \epsilon \leq \frac{\hat{N}_G(t)}{N_G(t)} \leq 1 + \epsilon$$

and the inverse

$$\frac{1}{1+\epsilon} \leq \frac{N_G(t)}{\hat{N}_G(t)} \leq \frac{1}{1-\epsilon}$$

Note that even if the maxima of $\hat{N}_G(t)$ and $N_G(t)$ can be attained for different values, say t_1 and t_2, the ratio of the two maxima remains the same for every $t \geq \max\{t_1, t_2\}$. Hence, by taking such a t, we can say

$$1 - \epsilon \leq \frac{\max_t \hat{N}_G(t)}{\max_t N_G(t)} \leq 1 + \epsilon$$

Algorithm 5: Algorithm to compute the effective diameter at α of a graph G

```
foreach $t = 0, 1, 2, \dots$ do
    compute $\hat{N}_G(t)$ (with error $\epsilon$ and confidence $1 - \delta$)
    if Some termination condition holds then break
end
$M \leftarrow \max_t \hat{N}_G(t)$
find the largest $D^-$ such that $\frac{\hat{N}_G(D^-)}{M} \le \alpha(1 - 2\epsilon)$
find the largest $D^+$ such that $\frac{\hat{N}_G(D^+)}{M} \ge \alpha(1 + 2\epsilon)$
return $[D^-, D^+]$ with confidence $1 - 3\epsilon$
```

As a consequence, we have

$$1 - 2\epsilon \le \frac{1-\epsilon}{1+\epsilon} \le \frac{\hat{H}_G(t)}{H_G(t)} \le \frac{1+\epsilon}{1-\epsilon} \le 1 + 2\epsilon$$

The probability $1 - (k+1)\delta$ is derived from th union bound, since we are considering $k + 1$ event at the same time. □

Using this theorem we can obtain the following result:

Corollary 1 ([BRV11]). *Assume that $\hat{N}_G(t)$ is known for each t with error ϵ and confidence $1 - \delta$ and there are points s and t such that*

$$\frac{\hat{H}_G(s)}{1 - 2\epsilon} \le \alpha \le \frac{\hat{H}_G(t)}{1 + 2\epsilon}$$

than, with probability $1 - 3\delta$ the effective diameter at α lies in $[s, t]$.

Hence, to compute the effective diameter of a graph, we can use algorithm 5, depicted in [BRV11]

3.2 probabilistic counters

All the algorithms that compute the neighbourhood function described here share a common problem: at iteration i they have to keep track of the cardinalities of the balls (definition 5) of radius i associated with each vertex.

Keeping explicitly the elements of each ball is not practicable for big graphs. To solve this problem the algorithms described here make use of probabilistic counters, namely Flajolet-Martins counters [FMM85] and HyperLogLog counters [FFGea07]. By using these counters we accept to be able only to add new elements to the set and to query about its size. Moreover we accept to deal with estimations rather than exact values for the cardinalities. The gain is that the space used to represent the set drops from at least $O(n)$ to $O(\log n)$

Algorithm 6: Algorithm to add elements to a Flajolet-Martins counter.

```
Input: $x$, an element of the set and $BITMAP$, the counter.
begin
    $i \leftarrow \rho(hash(x))$
    $BITMAP[i] = 1$
end
```

(in the case of Flajolet-Martins counters) or $O(\log\log n)$ (for HyperLogLog counters).

In what follows we describe two different algorithms to represent sets in a concise way: Flayolet-Martins counters and HyperLogLog counters.

3.2.1 *Flajolet-Martins counters*

In [FMM85], the authors present an algorithm to keep track of the number of distinct elements in large collections of data. This algorithm is based on statistical observation made on bits of hashed values of records.

We assume to have at out disposal a hash function that maps the elements of the set V into the set of bit strings of length L

$$hash : V \rightarrow \{0,1\}^L$$

The function $bit(y,k)$ returns the value of the k-th bit in the bit string y. Thus we have that the integer value of y given its binary representation is

$$y = \sum_{k \geq 0} bit(y,k) 2^k$$

We also define the function $\rho(y)$ that returns the position of the least significant 1-bit in the bit string y.

$$\begin{aligned} \rho(y) &= \min_{k \geq 0} bit(y,k) \neq 0 && \text{if } y > 0 \\ &= 0 && \text{if } y = 0 \end{aligned}$$

The key observation is the following

Observation 1. *If the values of hash(x) are uniformly distributed, then the problability of the pattern 0^k1 is 2^{-k-1}.*

The algorithm keeps track of the occurrences of such patterns in a vector $BITMAP$ with L elements. The procedure to add an element of the set to the counter is depicted in algorithm 6.

So $BITMAP[i]$ is equals to 1 if and only if the pattern 0^i1 has appeared among hashed values. From the observations made on bit pattern probabilities, it follows that, at the end of the execution,

$BITMAP[i]$ will be almost certainly 0 if $i \gg \log_2 n$ and will be 1 if $i \ll \log_2 n$. For $i \approx \log_2 n$ there will be a fringe of zeros and ones.

In [FMM85] the authors propose the index R of the leftmost 0 (i.e. the 0 with the smallest index) as an indicator of $\log_2 n$.

More precisely we have that the expectation and the standard deviation of R are

$$E(R) \approx \log_2 \varphi n \quad \varphi = 0.77351\ldots \tag{1}$$

$$\sigma(R) \approx 1.12 \tag{2}$$

The non trivial proof is provided in [FMM85] and we refer the interested reader to that paper.

We can observe that using R as an indicator of the cardinality of the set gives us an error that is typically of one order of binary magnitude. To reduce this problem, a first approach would be tu use m bitmaps and m hash functions, one for each bitmap. We can then take the average

$$A = \frac{R^1 + R^2 + \cdots + R^m}{m} \tag{3}$$

where R^i is the index of the leftmost 0 in the i-th bitmap. This way A is what we take as an indicator for the cardinality of the set, with the following mean and standard deviation

$$E(A) = \log_2 \varphi n \tag{4}$$

$$\sigma(A) = \frac{\sigma_\infty}{\sqrt{m}} \tag{5}$$

where $\varphi \simeq 0.77351$ and $\sigma_\infty \simeq 1.12127$ [FMM85].

This approach is simple and enables to control the standard deviation using the parameter m. However it's not simple to obtain m distinct hash functions.

The solution is to use *stochastic averaging*. We assign each of the m bitmaps an index. Then we use a single hash function $h(x)$ to distribute each record of the set into one of the m lots by computing $\alpha = h(x) \mod m$. We then update only the bitmap with index α, using the value $h(x)$ div $m \equiv \lfloor h(x)/m \rfloor$ to set to one the bit at the appropriate index. The procedure to add an element to the counter in this way is depicted in algorithm 7.

At the end of execution the estimate of n is

$$\boxed{n = m \left(\frac{1}{\varphi}\right) 2^A} \tag{6}$$

We now define the bias and the standard error as quality metrics.

Definition 8 (Standard error). *The standard error of an estimate of n is the quotient of the standard deviation of the estimate by the value of n.*

The standard error is thus an indication of the *expected relative accuracy* of an algorithm estimating n.

Algorithm 7: Adding an element to a counter with stochastic averaging

```
Input: $x$, an element of the set and $m$ bitmaps, the counters
begin
    $hash \leftarrow h(x)$
    $\alpha \leftarrow hash \mod m$
    $i \leftarrow \rho(hash \text{ div } m)$
    $BITMAP_\alpha[i] \leftarrow 1$
end
```

Definition 9 (Bias). *The bias of an algorithm that estimates n is the ratio between the estimate of n and its exact value, for n large.*

The quality of Flajolet-Martins counters is then measured as follows:

$$\text{bias:} \quad 1 + \frac{0.31}{m} \tag{7}$$

$$\text{standard error:} \quad \frac{0.78}{\sqrt{m}} \tag{8}$$

The proof of these values can be found in [FMM85].

3.2.2 *HyperLogLog counters*

In [FFGea07] the authors propose an algorithm similar to the one presented in the previous section but based on a different observable. This is leads to a standard error that is actually worse but enables a substantial gain in the space required. In fact this algorithm only needs $O(\log \log n)$ memory (hence the name) to keep track of cardinalities up to n.

In the description that follows we assume again that we have an hash function $h(x)$ that maps the set elements into binary strings. The function $\rho(y)$ is used to get the position of the leftmost 1-bit in the binary string y.

This algorithm, like the previous one, uses stochastic averaging to control the standard error on the estimated value. The input multiset $\mathcal{M}$ is partitioned in $m = 2^b$ multisets $\mathcal{M}_1 \dots \mathcal{M}_m$ using the first b bits of the hashed values.

Then for each $\mathcal{M}_j$ the observable is

$$M^j = \max_{x \in \mathcal{M}_j} \rho(x) \tag{9}$$

The intuition underlying the algorithm is the following: each multiset $\mathcal{M}_j$ is expected to contain n/m distinct elements at the end of the execution. Hence the parameter M^j should be close to $\log_2(n/m)$.

Algorithm 8: HyperLogLog algorithm

```
Input: An element $x$ of the multiset and a collection of
       registers $M^1 \dots M^m$
begin
    $h \leftarrow hash(x)$            // $h = h_1 h_2, \cdots h_b h_{b+1}, \cdots$
    $j \leftarrow 1 + < h_1 \cdots h_b >_2$       // Index of the register to
    update
    $M^j \leftarrow \max(M^j, \rho(h_{b+1} h_{b+2} \cdots))$
end
```

The harmonic mean of the quantities 2^{M^j} should then be in the order of n/m.

The algorithm returns the following estimate of the cardinality

$$N = \frac{\alpha_m m^2}{\sum_{j=1}^{m} 2^{-M^j}} \quad \text{with} \quad \alpha_m = \left(m \int_0^\infty \left(\log_2 \frac{2+u}{1+u} \right)^m du \right)^{-1} \tag{10}$$

The parameter α_m is used to correct a multiplicative bias of the algorithm. The counting procedure is specified in algorithm 8.

The quality of the algorithm, measured with bias and standard error, is assured by the following theorem.

Theorem 8 ([FFGea07]). *Let algorithm 8 be applied to a multiset of cardinality n, using $m > 3$ registers. The value N is the estimated cardinality.*

1. *The estimate N is* almost unbiased, *in the sense that*

$$E(N) \underset{n \to \infty}{=} 1 + \delta_1(n) + o(1)$$

   where $|\delta_1(n)| < 5 \cdot 10^{-5}$ for $m \geq 16$.

2. *The standard error σ/n satisfies, for $n \to \infty$*

$$\frac{\sigma}{n} \underset{n \to \infty}{=} \frac{\beta_m}{\sqrt{m}} + \delta_2(n) + o(1)$$

   where $|\delta_2(n)| < 5 \cdot 10^{-4}$ for $m \geq 16$. The constants β_m are bounded, with $\beta_{16} = 1.106$, $\beta_{32} = 1.070$, $\beta_{64} = 1.054$, $\beta_{128} = 1.046$ and $\beta_\infty = 1.03896$.

The proof of this theorem, as well as the derivation of the constants α and β, is provided in [FFGea07].

3.3 ALGORITHMS

In this section we describe four different algorithms. The first two algorithms, namely ANF and HyperANF are single-machine, sequential algorithms. The last two, HADI and our own D-HANF are algorithms on MapReduce. The algorithms described in this section all share a common structure, they differ only in the type of counter used. Before introducing the algorithms, we describe the general pattern they follow.

The diameter is computed by means of the neighbourhood function computation (see definition 6). By theorem 1 we can compute the effective diameter at α given the neighbourhood function of a graph. The problem is hence to compute the neighbourhood function of the given graph. The algorithms proceed iteratively. At iteration t, each node v collects the cardinality of the ball of radius t centered in v. The sum of all these cardinalities is $N(t)$, the t-th value of the neighbourhood function. Since the neighbourhood function is strictly monotone, once we get $N(t) = N(t-1)$ we can stop the algorithm, since we arrived at the diameter.

The problem is how to count the cardinalities of the balls, whose radius increases at each iteration. Keeping an explicit set for every node is impractical, since it leads to a memory requirement that is $O(n)$ for each node, that is $O(n^2)$ for the entire graph. This is where probabilistic counters come into play. Each node maintains a counter that estimates the size of the ball centered in the node at the current iteration. The type of counter used is what differentiates the algorithms. Now instead of computing explicitly the balls, at each iteration each node sends its counter to all its neighbours. Then each node performs the union of all the counters it received. The result is the counter that estimates the cardinality of the ball at the current iteration.

3.3.1 *ANF*

ANF stands for Approximate Neighbourhood Function [PGF02]. This algorithm uses Flajolet-Martins counters, described in section 3.2.1. It is implemented in the `snap` tool[1].

Since the algorithm stops once it reaches the diameter and runs sequentially through all the nodes of the graph at each iteration, its running time is $O(nd)$.

Each counter takes $O(\log n)$ bits, hence the space requirement of the algorithm is $O(n \log n)$. This is a big improvement on the $O(n^2)$ solution, however for really big graphs it is impractical.

3.3.2 *HyperANF*

HyperANF [BRV11] is an improvement over ANF. The running time is the same, however the space requirements are much better. The counters used are HyperLogLog counters. This means that they take up only $O(\log \log n)$ bits each. For the typical situation of a graph with 32-bit integer IDs, this means that each HyperLogLog counter is 5 bits long, versus the 32 bits of Flajolet-Martins counters. The overall

1 `http://snap.stanford.edu`

space requirement of the algorithm is $O(n \log \log n)$. This allows the implementation of the algorithm to deal with extremely big graphs[2].

3.3.3 *HADI*

The data parallel nature of each iteration of the general algorithm lends itself to a parallelization on MapReduce. HADI [KTA+08] is a parallelization of ANF on MapReduce.

Each iteration is divided in two steps. The first one maps, for each counter, the current counter to all the neighbours of the node. The reduce function then performs the union of all the counters associated to a node. The second step performs the sum of all the estimates of the cardinalites, yielding at iteration t the t-th value of the neighbourhood function.

Each iteration runs in a constant number of rounds, hence the running time of the algorithm is $O(d)$.

The algorithm is implemented in Hadoop[3]. This has a rather severe impact on the actual performance, since Hadoop stores every intermediate result to disk. This means that at the beginning of each iteration, at its end and even between the two steps that compose the iteration, the distributed filesystem is written or read. Each access to the distributed filesystem bears a significant time penality. This is one of the reasons why HyperANF can outperform HADI, even if it uses a single machine [BRV11]. The others are smart programming techniques, like broadword programming or systolic computation. See the paper for further details.

3.3.4 *D-HANF: Distriduted HyperANF*

Much as HADI is a parallelization of ANF, D-HANF is a parallelization of HyperANF. The algorithm has the same iterative structure of the others.

Pseudocode 9 depicts the algorithm. The first step, initialization, consists in a single map that associates the node id to a newly created counter. The node id is pushed into the counter, using the algorithm described in pseudocode 8. Then the iterative process begins. The counters and graph datasets are cogrouped, yielding a dataset with the same number of elements. A map function sends the counter of each node to all the neighbours. The reduce function then combines all received counters using the union operation over them. This yields the new counters dataset. Finally a map function takes the counters and computes their estimated sized. The sum of these sizes, performed as a prefix computation, gives the value $N(t)$ of the neighbourhood function at iteration t.

2 `http://webgraph.di.unimi.it/`
3 http://www.cs.cmu.edu/ pegasus/

Algorithm 9: D-HANF algorithm

```
Input: A graph $G$ represented as a $< nodeId, neighbourhood >$
       dataset.
Output: The neighbourhood function of the graph

Function newCounter (nodeId) is
    counter ← a new empty HyperLogLog counter ;
    add nodeId to the counter ;
    return counter ;
end
counters ← Map (id, neighbourhood) to (id, newCounter(id)) ;
$t \leftarrow 0$;
while there are changed counters do
    Cogroup counters, graph on nodeId ;
    Map ( nodeId, (counter, neighbourhood) ) to
        foreach n in neighbourhood do Output (n, counter) ;
    end
    counters ← Reduce ( nodeId, counters ) to
        newCounter ← union of all counters;
        Output ( nodeId, newCounter ) ;
    end
    Map ( nodeId, counter ) to ( nodeId, size of counter ) ;
    $N(t) \leftarrow$ sum of all counter sizes ;
    $t \leftarrow t + 1$;
end
```

Since each iteration runs in a constant number of rounds, the entire algorithm runs in $O(d)$ rounds.

The implementation of the algorithm is available as free software[4]. To implement the algorithm, given its iterative nature, we leveraged the in memory caching capabilities of the Spark framework. This is an improvement over an Hadoop based implementation, in that it avoids costly repeated writes to the distributed filesystem.

3.3.5 *Scalability and performance of D-HANF*

We have run the implementation of the algorithm[5] against some datases, namely DBLP, Amazon and Youtube (see appendix A). As shown in figures 1, 2 and 3 the algorithm has a good scalability. The plots are in logarithmic scale on both axes. The figures show that incrementing the number of processors regularly decreases the running time of the algorithm.

4 `https://github.com/Cecca/spark-graph`

5 `https://github.com/Cecca/spark-graph`

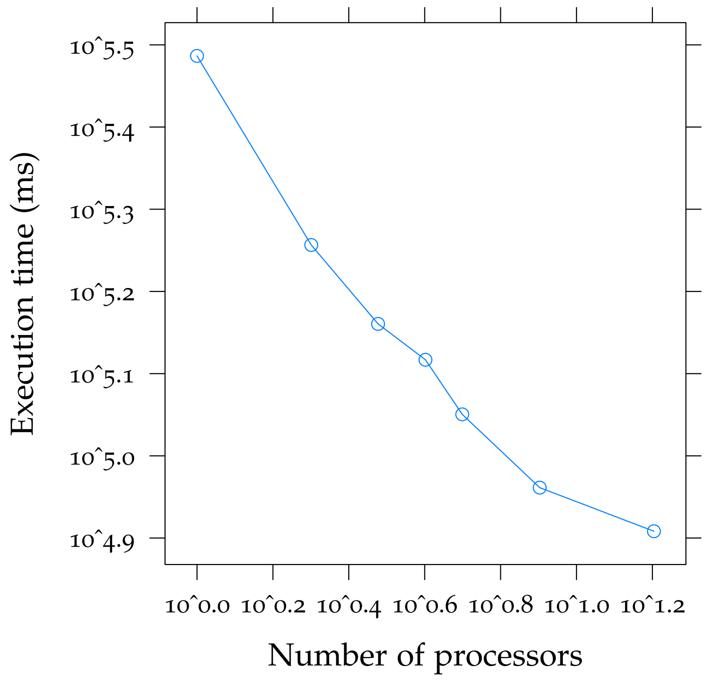


Figure 1.: Scalability of D-HANF on the DBLP dataset.

All tests have been performed with on a Power 7 machine with 128 GB of RAM and 48 cores, running SUSE Linux 11 SP2.

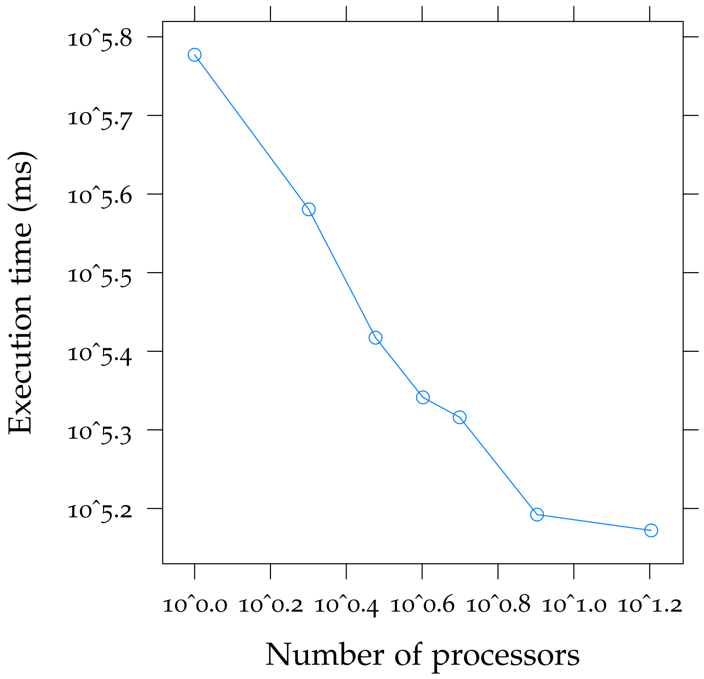


Figure 2.: Scalability of D-HANF on the Amazon dataset.

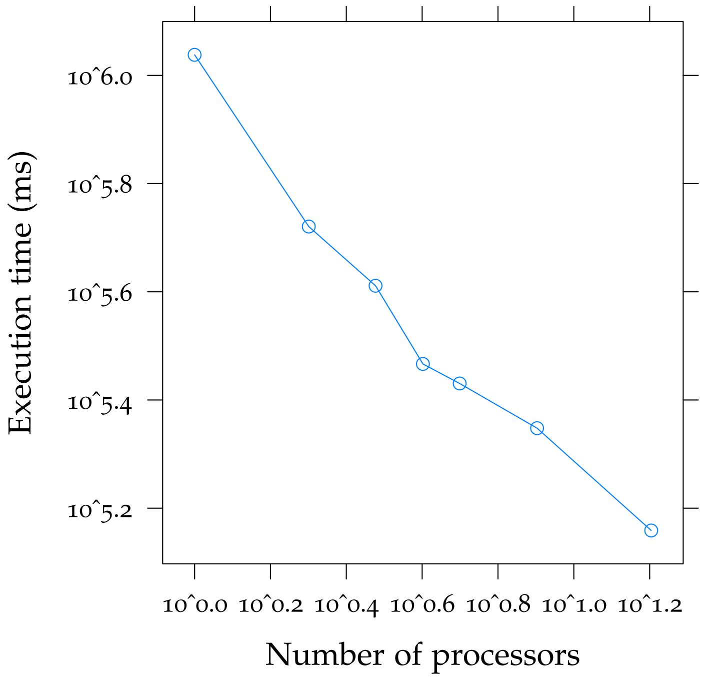


Figure 3.: Scalability of D-HANF on the youtube dataset.

4 MODULAR DECOMPOSITION

Modular decomposition is a technique that is often used as a preprocessing step in many graph-related algorithms. The main applications are in transitive orientation algorithms, optimizations problems on graphs and graph drawing [HP10]. A module of a graph is a subset M of vertices that share the same neighbourhood outside M.

This chapter is organized as follows. First we define the modular decomposition of a graph. Then we present a theorem that states that the diameter of the modular decomposition and the original graph are equals. Finally we present some experimental results showing that, unfortunately, the modular decomposition does not significantly reduce the cardinality of the graph.

4.1 DEFINITIONS

In this section are gathered some basic definitions related to the modular decomposition theory. We deal with undirected graphs, using the usual notation: $G(V, E)$ is a graph with V the set of vertices and E the set of edges. The neighbourhood of a vertex v of graph G is denoted $N_G(v)$ or $N(v)$ if there is no possible ambiguity. $\bar{G}$ denotes the complement graph of G. The description that follows is based on [HP10].

4.1.1 *Modules and modular partitions*

Let $M \subseteq V$ be a set of vertices of graph $G(V, E)$ and x a vertex in $V \setminus M$. x is said to be a *splitter* of M if there exist $y, z \in M$ such that $\{x, y\} \in E$ and $\{x, z\} \notin E$. M is *uniform* or *homogeneous* with respect to x if x is not a splitter of M. A set M uniform with respect to every $y \in V \setminus M$ is called a *module*. More formally

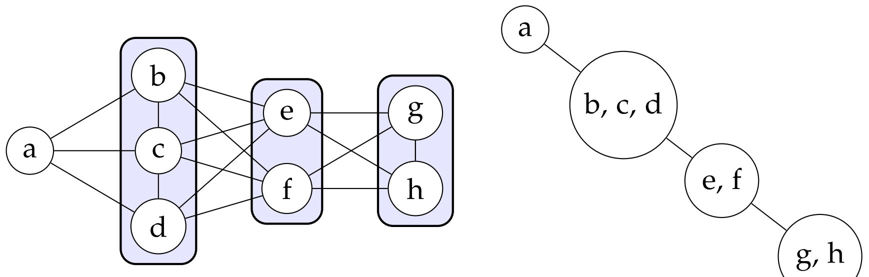
Definition 10 (Module)**.** *Given a graph $G(V, E)$ a set $M \subseteq V$ is a module if and only if*

$$\forall v \in V \setminus E : (\forall x \in M : \{x, v\} \in E) \vee (\forall x \in M : \{x, v\} \notin E)$$

The vertex set V and the singleton sets $\{v\}$ with $v \in V$ are obviously modules and are called *trivial modules*. A graph that only has trivial modules is called *prime*.

We say that two modules A and B are overlapping if $A \cap B \neq \emptyset$, $A \setminus B \neq \emptyset$ and $B \setminus A \neq \emptyset$ and we write $A \bot B$ [HP10].

Note that the definition of overlapping modules implies that if a module is a subset of another module then they are not considered



(a) The original graph. Modules are shown by boxes.

(b) The quotient graph.

Figure 4.: A graph, its modules and the quotient graph

overlapping. This will be a key concept in the definition of the modular decomposition tree.

Definition 11 (Strong module). *Given a set of modules $\mathcal{M}$, a module $M \in \mathcal{M}$ is said to be* strong *if it does not overlap any other module in $\mathcal{M}$. More formally, M is a module if and only if*

$$\begin{aligned}\forall X \in \mathcal{M}, X \neq M : M \cap X &= \emptyset \\ \wedge\, M \setminus X &= \emptyset \\ \wedge\, X \setminus M &= \emptyset\end{aligned}$$

Definition 12 (Maximal module). *A module M is* maximal *with respect to a set of vertices $\mathcal{S}$ if $M \subset \mathcal{S}$ and there is no module M' such that $M \subset M' \subset \mathcal{S}$. If the set $\mathcal{S}$ is not specified, the entire vertex set is intended.*

A particularly interesting case is when a set of modules defines a partition of the vertex set V. In this case we talk of a *modular partition*.

Definition 13 (Modular partition). *Let $\mathcal{P} = \{M_1, M_2, \cdots, M_k\}$ a partition of the vertex set of a graph $G(V, E)$. If $\forall i \in [1, k]$, M_i is a module of G, then $\mathcal{P}$ is said to be a* modular partition.

The following observation and the subsequent *quotient graph* definition will enable us to represent graphs in a more compact way.

Observation 2. *Two disjoint modules are either adjacent or non-adjacent.*

We say that two modules X and Y are adjacent if and only if any vertex in X is connected to all vertices in Y. Conversely, X and Y are not adjacent if no vertex of X is adjacent to any vertex of Y.

This observation leads to the definition of the *quotient graph*:

Definition 14 (Quotient graph). *Given a graph $G(V, E)$ and a modular partition $\mathcal{P} = \{M_1, M_2, \cdots, M_k\}$ of G, the quotient graph is $G_{\setminus P} = (V_{\setminus P}, E_{\setminus P})$ where each vertex in $V_{\setminus P}$ corresponds to a module of $\mathcal{P}$ and there is an edge between v_i and v_j if their corresponding modules are adjacent.*

Figure 4 shows an example of a graph, one of its modular partitions and the corresponding quotient graph.

4.1.2 *Modular decomposition*

The problem with modular partitions is that their number can be exponential in the cardinality of the vertex set. This is the case of complete graphs. However, given that modules can be nested one into another, a compact representation of all possible modules exists, namely the *modular decomposition tree*.

Before moving to the modular decomposition theorem that defines the tree, let's define a useful concept: maximal modular partitions.

Definition 15 (Maximal modular partition). *A* maximal modular partition *is a partition $\mathcal{P}$ that contains only maximal strong modules.*

There are a couple of caveats when dealing with maximal modular partitions. If a graph is not connected, every union of its connected components is a module. In this case the maximal modular partition is considered the one made up by the connected components, even if these modules aren't maximal. In the case of a connected graph whose complement graph is disconnected, the same argument applies: the modules of the maximal modular partition are the co-connected components, i.e. the connected components of the complement graph.

With this convention we can state that *each graph has a unique maximal modular partition.*

We can now move on to the statement of the modular decomposition theorem.

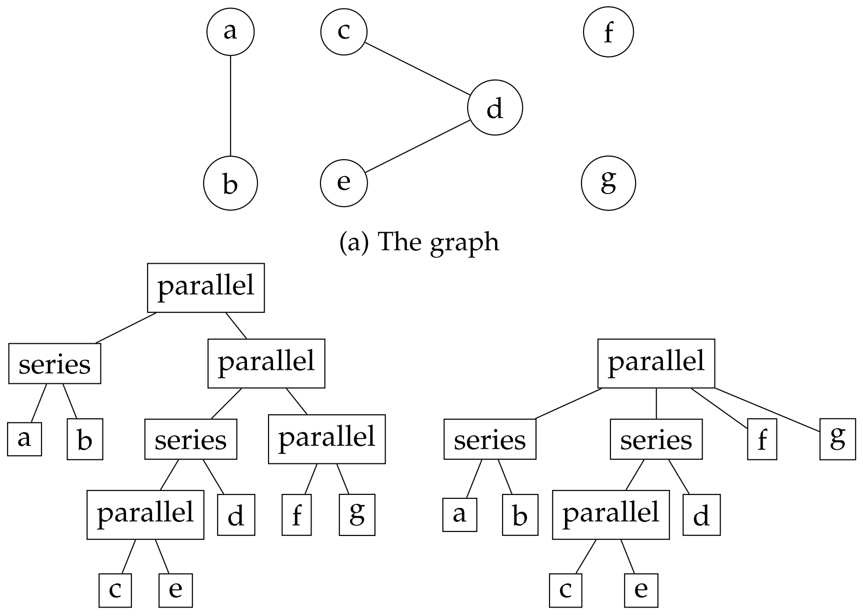
Theorem 9 (Modular decomposition theorem). *For any graph $G(V, E)$ one of the following conditions is valid:*

1. *G is not connected*
2. *$\bar{G}$ is not connected*
3. *both G and $\bar{G}$ are connected. The quotient graph $G_{\backslash\mathcal{P}}$, with $\mathcal{P}$ the maximal modular partition of G, is a prime graph.*

From this theorem follows the recursive definition of the modular decomposition tree.

Definition 16 (Modular decomposition tree). *Given a graph G, the root of the modular decomposition tree is the entire graph.*

1. *if G has a single node, then root of the modular decomposition tree has no children.*
2. *if G is not connected the children of the root are the roots of the modular decomposition trees of the connected components.*
3. *if $\bar{G}$ is not connected then the children of the root are the roots of the modular decomposition trees of the connected co-components.*



(a) The graph

(b) Unreduced form

(c) Reduced form

Figure 5.: A modular decomposition tree in unreduced and reduced form

4. *if both G and $\bar{G}$ are connected the children of the root are the roots of the modular decomposition trees of the modules of the maximal modular partition of G.*

This definition gives us a modular decomposition tree in *reduced* form [MS00]. A modular decomposition tree is not in reduced form (or is *unreduced*) if there is at least a parallel node that is child of another parallel node or a series node that is child of another series node. Figure 5b shows an example of such a tree.

Since the maximal modular partition of a graph is unique, we can conclude that the modular decomposition tree in reduced form is unique.

4.2 RELATIONS BETWEEN GRAPH DIAMETER AND MODULAR PARTITIONS

In this section we describe the relations between the diameter of undirected graphs and the diameter of their nontrivial modular partitions. In particular the diameter of an undirected graph is equal to the diameter of its nontrivial modular partitions if it is greater than 2.

The theorems proved in the remainder can be summarized as follows:

- The quotient graph of a nontrivial modular partition of a connected graph is itself connected.

- A path between two nodes in the same module is at most of length 2 if the module is connected to another one.
- The length of shortest path between two nodes in different modules of the quotient graph of a nontrivial modular partition is equal to the length of the shortest path between their containing modules.
- If the diameter of a connected graph is greater than 2, then it is equal to the diameter of the quotient graphs of its nontrivial modular partitions.

Lemma 2. *A nontrivial modular partition of a graph $G = (V, E)$ is connected if and only if G is connected.*

Proof. **Case 1**: The graph G is connected $\Rightarrow$ the quotient graph of any modular partition of G is connected.

Assume that the quotient graph $\hat{G} = (\hat{V}, \hat{E})$ of a nontrivial modular partition is not connected. Then $\exists \hat{V}' \in \hat{V}$ and $\hat{V}'' = \hat{V} \backslash \hat{V}'$ such that $\forall x \in \hat{V}'$, $\forall y \in \hat{V}''$ we have $\{x, y\} \notin \hat{E}$. Let $\hat{V}' = \{x \in V | x$ is in a module in $\hat{V}'\}$ and $\hat{V}'' = \{x \in V | x$ is in a module in $\hat{V}''\}$. From the definition of module follows that $\forall x \in V'$, $\forall y \in V''$, $\{x, y\} \notin E$, against the hypothesis that G is connected.

Case 2: The quotient graph $\hat{G}$ of a modular partition of G is connected $\Rightarrow$ The graph G is connected.

Assume that the graph $G = (V, E)$ is not connected. The simplest (i.e. with the smallest number of nodes) nontrivial modular partition is the one made up by a module that contains all the nodes of a connected component and another module with all the remaining nodes. Since the graph is not connected there is no edge between the two modules, hence any modular partition is not connected, against the hypothesis. □

Lemma 3. *Given a graph $G = (V, E)$ and the quotient graph $\hat{G} = (\hat{V}, \hat{E})$ of a nontrivial modular partition of G the shortest path between two nodes in the same module is at most 2 if the module is connected to at least another one.*

Proof. The intuition behind the proof is that if two nodes in the same module, connected to another module, are not neighbours, then there exists a path between them passing through a node in the other module. This gives a path of length 2.

Case 1: The two nodes are neighbours. Then the shortest path between them is of length 1.

Case 2: The nodes v_i and v_j in the same module M are either not connected by a path in M or such a path has length longer than 2. Let M' be the module of $\hat{G}$ connected to M. Then from the definition of module follows that all the vertices in M are connected to all vertices in M'. Let v_k be a node in M'. Then the path $< v_i, v_k, v_j >$ is a path of length 2 connecting v_i and v_j in G. □

Lemma 4. *Given a graph G and the quotient graph $\hat{G}$ of one of its nontrivial modular partitions, the shortest path between two vertices of G in different modules is equals to the shortest path between between the two modules in $\hat{G}$.*

Proof. The idea is that the shortest path between two vertices takes at most one node from each module.

Assume that the shortest path between vertices $v_i, v_j \in V$ contains two vertices u and w from the same module. Let this path be

$$\psi =< v_i, \cdots, t, u, \cdots, w, z, \cdots, v_j >$$

From the definition of module $\{t, u\} \in E \Rightarrow \{t, w\} \in E$ and $\{w, z\} \in E \Rightarrow \{u, z\} \in E$ since u and w are from the same module. Hence the path

$$\psi' =< v_i, \cdots, t, u, z, \cdots, v_j >$$

connects v_i and v_j and is shorter than ψ, so the assumption of ψ being the shortest path is absurd. From his follows that the shortest path between two nodes in different modules takes at most one node from each module.

Since a different module is associated to each node in a shortest path between v_i and v_j, then such a path has the same length of the shortest path between the modules containing v_i and v_j. □

Theorem 10. *Given a connected graph G with diameter greater than 2, its diameter is equal to the diameter of the quotient graph $\hat{G}$ of any of its nontrivial modular partitions.*

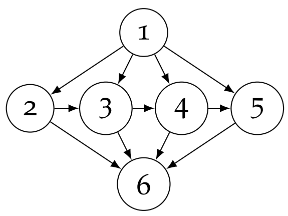
Proof. From lemma 4 we know that the shortest path between two modules is equal to the shortest path between all the nodes that they respectively contain. From the hypothesis of connection and from lemmas 2 and 3 we know that the shortest path between two vertices in the same module is at most 2. Since the diameter of the graph is greater than 2 for hypothesis, then the longest shortest can't be between two nodes in the same module. Being the longest shortest path in G between nodes in different modules, we have that the longest shortest path in $\hat{G}$ has the same length. Hence the diameter of G and $\hat{G}$ is the same.

□

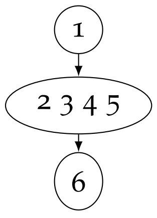
4.3 RELATION WITH DIRECTED GRAPHS

Unfortunately the theorems of the previous section do not hold anymore for directed graphs. This is due to the fact that lemma 3 does not hold anymore.

An example of this situation is shown in figure 6. Figure 6a shows a directed graph where the longest shortest path is between vertices 2 and 5. These two vertices are in the same module . The diameter



(a) A directed graph with diameter 3.



(b) A nontrivial modular partition of the graph in figure 6a.

Figure 6.: A case of a directed graph and a modular partition with different diameters.

of this graph is 3. However the quotient graph shown has diameter 2. This is due to the fact that since lemma 3 does not hold we are not guaranteed that the shortest path between two nodes in the same module has length at most 2.

4.4 EXPERIMENTAL RESULTS

We have conducted some experiments to verify the reduction in cardinality given by the modular decomposition. The experiments have been conducted on datasets described in Appendix A.

The datasets used are Facebook (A.1), Autonomous Systems (A.2), DBLP (A.3) and the Power Law graphs (A.7). The quotient graph is obtained by grouping together all the nodes that pertain to the same module in the first level of the modular decomposition tree. Using other levels of the modular decomposition tree does not make sense, since they yield quotient graphs with more nodes.

Figure 7 shows the changes in cardinality between the original graph and the graph obtained through modular decomposition. Table 1 compares the cardinalities of the original and quotient graph for each dataset.

Reductions in cardinalities are negligible. For power law graphs the quotient graph is identical to the original one. For the three real world dataset used (Facebook, Autonomous Systems and DBLP) the reduction is around 15%, in the best case (DBLP).

These experiments show that, although interesting from a theoretical point of view, the modular decomposition is of little practical interest in this case, since it does not bring substantial reductions in the graph size.

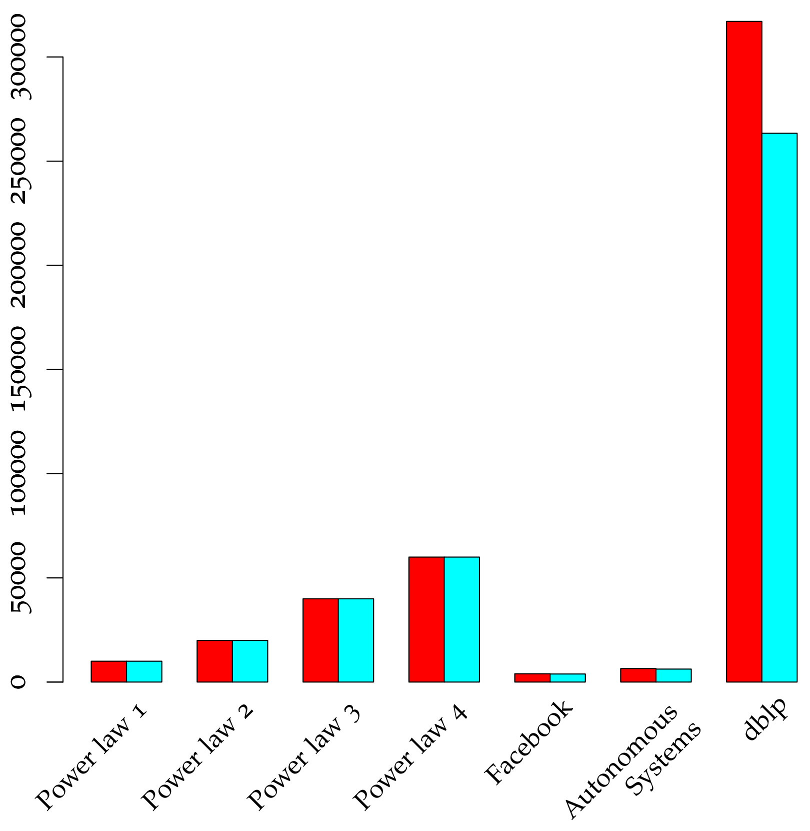


Figure 7.: Cardinality change of graphs using modular decomposition

Dataset	Original cardinality	Quotient graph cardinality
Power law 1	9996	9996
Power law 2	19985	19985
Power law 3	39938	39938
Power law 4	59974	59974
Facebook	3963	3858
Autonomous systems	6474	6225
DBLP	317080	263388

Table 1.: Cardinality values for original graph and its modular decomposition for various datasets

5 STAR DECOMPOSITION

We have seen that the modular decomposition does not yield significant reductions in the graph cardinality. This is mainly due to the very rigid nature of the definition of modular decomposition. Moreover, the algorithms to find the modular decomposition are somewhat convoluted. Finally, the parallelization of the algorithm is not trivial.

Here we are going to describe a decomposition technique that is simpler to compute and with better compression ratios. However, we are going to pay this gain in simplicity and cardinality reduction with an approximate result. This algorithm no longer finds a quotient graph with the exact same diameter as the original one. Instead it finds a graph with a diameter that is smaller, within constant bounds.

This decomposition is called *star decompositions* after the idea from which is builds up.

The basic idea is that, as for the other decompositions, we need to substitute a set of nodes of the graph with a single node that in some way represents a summary of the connectivity of the set. So we can take stars in the graph and replace them with a single node, that inherits all the edges that the star has towards the outside.

5.1 STAR DECOMPOSITION

In this section we describe *star decomposition*. The idea is that we want to decompose the graph in stars, where each star is then replaced by a single node.

Definition 17 (star)**.** *Given a graph $G = (V, E)$ a set of nodes $S \in V$ is said to be a star if it is composed of a vertex c, named the center of the star, and* all *the neighbours of c.*

Definition 18 (star decomposition)**.** *Given a graph $G = (V, E)$, a star decomposition is a graph $\hat{G}$ such that:*

1. *there is a bijective function $\varphi : \hat{V} \to \mathbb{S}$ between the set of nodes $\hat{V}$ of $\hat{G}$ and a partition $\mathbb{S}$ of V. Each element $S \in \mathbb{S}$ is a subset of a star in G of center c, with $c \in S$.*
2. *given any two vertices u and v in $\hat{G}$ there is an edge between u and v if there is an edge between any $x \in \varphi(u)$ and $y \in \varphi(v)$.*

We are interested in the decomposition with the smallest possible number of nodes. Note that finding such a decomposition is an instance of Set Cover and hence is NP-hard [CLRS09]. In fact we can take the vertex set V as the universe set and the stars of G as the

Algorithm 10: Serial algorithm to compute the star decomposition of a graph G

```
Input: A graph $G$
Output: The star decomposition of $G$
begin
    $nodes \leftarrow$ list of nodes of $G$ sorted by degree;
    foreach node $c \in nodes$ do
        if $c$ is not colored then
            foreach $v \in neighbourhood(c)$, $v$ not colored do
                color $v$ with the $c$'s color;
            end
        end
    end
    foreach edge $(u, v)$ of the graph do
        $c_u \leftarrow$ the color of $u$ ;
        $c_v \leftarrow$ the color of $v$ ;
        Create nodes $c_u$ and $c_v$ in the quotient graph ;
        Add edge $(c_u, c_v)$ to the quotient graph ;
    end
end
```

sets we are using to cover V. The algorithms that we will develop will then be targeted to finding a star decomposition with a small number of nodes that may not be optimal.

Now that we have formally defined the decomposition, we can move on to the description of the serial algorithm to find it.

5.1.1 *Algorithm for the star decomposition*

The algorithm should find a decomposition with a small number of nodes. The idea is then to first sort the nodes by decreasing degree. Then each vertex is processed following this order. If it is not already assigned to a star it is taken as a center for a new star. All of its unassigned neighbours will become part of the new star. Since nodes are processed in a decreasing degree order, we are sure that condition 2 is met, since if a node is assigned to a star with center c it means that no node with degree higher than c claimed that node for its own star. Pseudocode 10 describes the algorithm.

In this algorithm a node is colored with the ID of the center of the star it pertains to. Clearly each edge is visited at most one time, so, ignoring the initial sorting, the running time of the algorithm is $O(m)$, where m is the number of edges of the graph G.

5.1.2 *Bounds on the diameter of the star-decomposition*

In this section we shall see that there are precise and constant bounds to the diameter of the star decomposition's quotient graph. First, we state a lemma about the distance of the nodes in subsets of stars.

Lemma 5. *Let $G = (V, E)$ be a graph and $S \subseteq \bar{S}$, where $\bar{S}$ is a star in G with center c, with $c \in S$. Then, $\forall u, v \in S$ the distance between u and v is at most two.*

Proof. The nodes are either adjacent or connected by the node c. Hence the shortest path length between u and v is 1 or 2. □

With this lemma we can state the following theorem, that puts a bound on the diameter of the quotient graph of the star decomposition.

Theorem 11. *Let $G = (V, E)$ be a graph and $\hat{G} = (\hat{V}, \hat{E})$ a star decomposition of G. Then*

$$\frac{d-2}{3} \leq \hat{d} \leq d \tag{11}$$

where d is the diameter of G and $\hat{d}$ is the diameter of $\hat{G}$.

Proof. Recall from definition 18 that $\varphi(v)$ is the bijective function that maps the nodes $v \in \hat{V}$ to the sets $S \in \mathbb{S}$.

Consider two nodes $u, v \in \hat{V}$ at distance $\hat{d}$. Consider now $x \in \varphi(u)$ and $y \in \varphi(v)$ and assume that there is a path σ between x and y whose length is strictly less than $\hat{d}$. If such a path exists, then we can build a path between u and v in $\hat{G}$ made up by $\varphi^{-1}(w), \forall w \in \sigma$. This path will have a length $l \leq \text{len}(\sigma) < \hat{d}$ which is absurd, since u and v are at distance $\hat{d}$. Since $\hat{d} \leq \text{len}(\sigma) \leq d$, the upper bound $\hat{d} \leq d$ holds.

Conversely, for the lower bound, consider two nodes x and y in V at distance d. Consider then $u = \varphi^{-1}(x)$ and $v = \varphi^{-1}(y)$ and assume that there is a path $\hat{\sigma}$ between u and v (see figure 8) such that

$$\text{len}(\hat{\sigma}) < \frac{d-2}{3}$$

If such a path exists, then we can build a path σ from x to y in G by taking nodes from $\varphi(w)$, for each $w \in \hat{\sigma}$. Thanks to lemma 5, we can state that each node $w \in \hat{\sigma}$ corresponds to at most 2 arcs in σ. The length of σ is then

$$\text{len}(\sigma) \leq 2\,(\text{len}(\hat{\sigma}) + 1) + \text{len}(\hat{\sigma}) = 3\,\text{len}(\hat{\sigma}) + 2 < d$$

however, this is absurd, since the distance between x and y is d. Since $\frac{d-2}{3} \leq \text{len}(\hat{\sigma}) \leq \hat{d}$, we have that the lower bound holds. □

We might be interested in obtaining a greater reduction in the graph's size than the one that the star decomposition allows us to

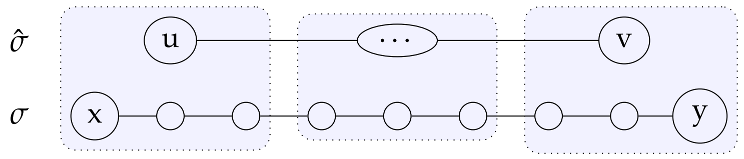


Figure 8.: The expansion of the path $\hat{\sigma}$ between u and v in $\hat{G}$ yields the path σ in G.

Table 2.: Statistics for the Facebook dataset

	Original graph	Star decomposition
nodes	4039	250
edges	88234	976
diameter	8	3
average degree	44.5	7.808
maximum degree	1034	240

achieve. We can apply the star decomposition repeatedly to the quotient graph of the previous application. In this way we can get a quotient graph that is arbitrarily smaller than the original one. However the degradation of the diameter becomes so big that the quotient graph is unlikely to be useful. In fact the bound on the diameter worsens of a factor of approximately 3 at each application. This means that after k applications, the bound on the diameter of the quotient graph is approximately 3^k.

In the next chapters we shall see alternative decompositions that address the problem of reducing the graph size at will while maintaining the bound on the diameter under control.

5.1.3 *Experimental results*

The star decomposition has been applied to a some real world datasets. They are all described in Appendix A.

Note that for the bigger graphs (namely DBLP, Amazon, Youtube and the California road network) the diameter has been computed using WebGraph [BV03], using the Hyper Approximate Neighbourhood Function, hence it as an estimate of the real diameter.

Figure 9 summarize the results of the application of the decomposition on the number of nodes of the graphs.

Figure 10 shows the relation between the diameters of the original and decomposed graph for various datasets.

FACEBOOK GRAPH STAR DECOMPOSITION The star decomposition has been applied on a Facebook graph (Section A.1). For statistics about the graph and its star decomposition refer to the first column of table 2.

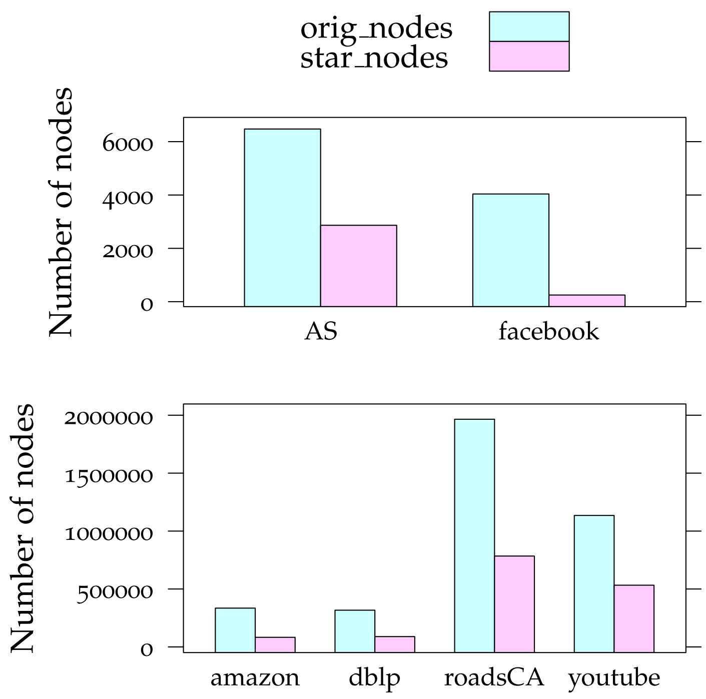


Figure 9.: Cardinality of the original and star decomposed graph for various datasets

AUTONOMOUS SYSTEMS GRAPH STAR DECOMPOSITION The Autonomous Systems graph (Section A.2) represents the graph of routers comprising the Internet. Statistics about this graph and its star decomposition can be found in the first column of table 3.

DBLP GRAPH STAR DECOMPOSITION The DBLP graph (Section A.3) represents relationships between scientific papers authors.

Statistics about the original graph are displayed in the first column of table 4, whereas its second column gives information about the star decomposition of the graph.

Table 3.: Statistics for the Autonomous Systems Graph

	Original graph	Star decomposition
nodes	6474	2865
edges	12572	3585
diameter	9	5
average degree	3.89	2.5
maximum degree	1458	2363

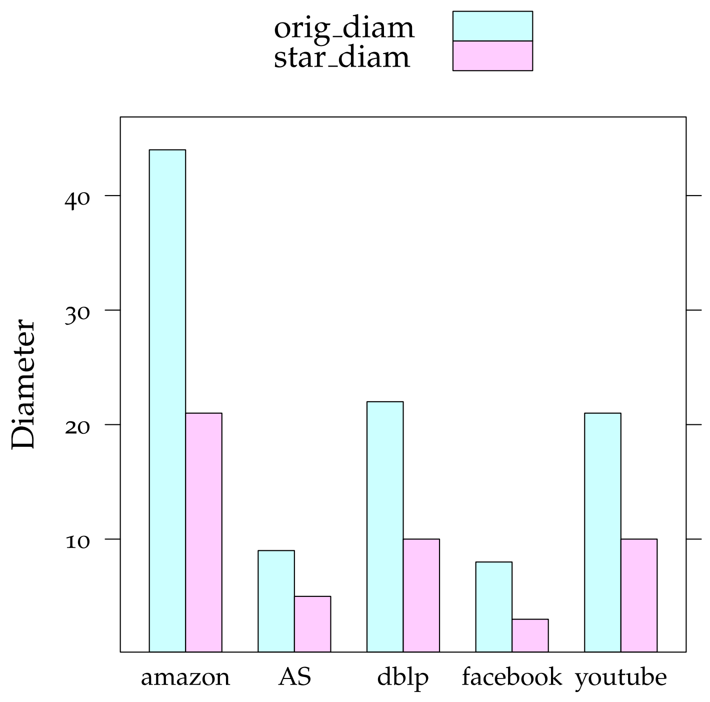


Figure 10.: Relation between the diameter of graphs and their star decomposition. The result relative to the California roads dataset (that can be found in table 7) isn't included for scale reasons.

Table 4.: Statistics for the DBLP graph and its star decomposition

	Original graph	Star decomposition
nodes	317080	89420
edges	1049866	239640
diameter	22	10
average degree	6.62	5.36
maximum degree	343	1741

Table 5.: Statistics for the Amazon graph and its star decomposition

	Original graph	Star decomposition
nodes	334863	82944
edges	925872	193526
diameter	44	21
average degree	5.53	4.67
maximum degree	549	651

Table 6.: Statistics for the Youtube graph and its star decomposition

	Original graph	Star decomposition
nodes	1134890	532723
edges	2987624	1004143
diameter	21	10
average degree	5.27	3.77
maximum degree	28754	198711

Table 7.: Statistics for the California roads graph and its star decomposition

	Original graph	Star decomposition
nodes	1965206	784305
edges	2766607	1284533
diameter	850	444
average degree	2.81	3.28
maximum degree	12	17

AMAZON GRAPH STAR DECOMPOSITION This graph was build by crawling the Amazon website (Section A.4. Statistics on the graph and its star decomposition are contained in table 5.

YOUTUBE GRAPH The Youtube graph (Section A.5) represents relations of friendship between Youtube users. Statistics abuot the graph and its star decomposition can be found in table 6.

CALIFORNIA ROADS GRAPH This graph represents a road network of California (Section A.6). Nodes represents intersections and endpoints, while the roads connecting these intersections or road endpoints are represented by undirected edges.

Statistics regarding this graph and is star decomposition can be found in table 7.

6 BALL DECOMPOSITION

In chapter 5 we have introduced the star decomposition. This decomposition leads to good results, however if one wants to achieve even better cardinality reductions, he is forced to repeatedly apply the decomposition. This leads to a degradation of the bound on the diameter that gets quickly useless. Applying k times the decomposition in fact leads to an approximation factor of 3^k.

We seek an approach to cardinality reduction that can lead to arbitrary reductions by controlling a parameter r. Moreover, we want this approach to have a bound on the diameter that is at most linear in the parameter r.

This approach is the *ball decomposition*. This is a generalization of the star decomposition: instead of grouping nodes by stars of radius 1, we group nodes by balls of radius r. If two nodes are connected by an edge in the original graph, then the balls to which they pertain are connected in the quotient graph.

By changing the parameter r we can get different reductions in the graph cardinality. The bigger the value of r, the greater the reduction. When r equals the graph's diameter, we have a quotient graph with only one node. In the case $r = 1$, the ball decomposition is equivalent to the star decomposition.

In what follows we will formalize the notion of ball decomposition.

Definition 19 (Ball)**.** *Given a graph $G = (V, E)$ and an integer r, a set made up by $c \in V$ and all the nodes that whose distance from c is at most r is said to be a* ball.

Definition 20 (Ball cardinality)**.** *The* ball cardinality *at r of a node v is the number of nodes that are comprised in the ball of center v and radius r.*

Definition 21 (Ball decomposition)**.** *Given a graph $G = (V, E)$ and an integer r, a ball decomposition of radius r of G is a graph $\hat{G}$ such that:*

1. *there is a bijective function $\varphi : \hat{V} \rightarrow \mathbb{B}$ between the set of nodes $\hat{V}$ of $\hat{G}$ and a partition $\mathbb{B}$ of V. Each element $B \in \mathbb{B}$ is a subset of a ball in G of radius r and center c, with $c \in B$.*

2. *given any two vertices u and v in $\hat{G}$ there is an edge between u and v if there is an edge between any $x \in \varphi(u)$ and $y \in \varphi(v)$.*

Finding the ball decomposition with the minimum number of nodes is an instance of Set Cover and hence is NP-hard [CLRS09]. The universe set is the vertex set V of the original graph and the sets with which we want to cover it are the balls of G with radius r.

6.1 BOUNDS ON THE DIAMETER

As for the star decomposition, also the ball decomposition has well defined bounds on the diameter. We shall see that the bound is linear in the parameter r.

Lemma 6. *Let $G = (V, E)$ be a graph and $B \subset \bar{B}$, where $\bar{B}$ is a ball in G of radius r and center c, with $c \in B$. Then, $\forall u, v \in B$, the distance between u and v is at most $2r$.*

Proof. The paths $< c, \cdots, u >$ and $< c, \cdots, v >$ are at most of length r. Hence the path $< u, \cdots, c, \cdots, v >$ is at most of length $2r$. □

We can now state the following theorem.

Theorem 12. *Let $G = (V, E)$ be a graph and $\hat{G} = (\hat{V}, \hat{E})$ a ball decomposition with radius r of G. Then*

$$\frac{d-2r}{2r+1} \leq \hat{d} \leq d \tag{12}$$

where d is the diameter of G and $\hat{d}$ is the diameter of $\hat{G}$

Proof. Recall from definition 21 that $\varphi(v)$ is the bijective function that maps the nodes $v \in \hat{V}$ to the sets $B \in \mathbb{B}$.

Consider two nodes $u, v \in \hat{V}$ at distance $\hat{d}$ and the nodes $x \in \varphi(u)$ and $y \in \varphi(v)$. Assume that there is a path σ between x and y whose length is strictly less than $\hat{d}$. If such a path exists, then we can build a path between u and v in $\hat{G}$ made up by $\varphi^{-1}(w), \forall w \in \sigma$. This path will have a length $l \leq \text{len}(\sigma) < \hat{d}$ which is absurd, since u and v are at distance $\hat{d}$. Being $\hat{d} \leq \text{len}(\sigma) \leq d$, the upper bound holds.

As for the upper bound, consider two nodes x and y in V at distance d. Then, consider $u = \varphi^{-1}(u)$ and $v = \varphi^{-1}(y)$ and assume that there is a path $\hat{\sigma}$ between u and v such that

$$\text{len}(\hat{\sigma}) < \frac{d-2r}{2r+1}$$

If such a path exists, then we can build a path σ from x to y in G by taking nodes from $\varphi(w)$, for each $w \in \hat{\sigma}$ (figure 11). Thanks to lemma 6, we can state that each node $w \in \hat{\sigma}$ corresponds to at most $2r$ arcs in σ. The length of σ is then

$$\text{len}(\sigma) \leq 2r\,(\text{len}(\hat{\sigma}) + 1) + \text{len}(\hat{\sigma}) = (2r+1)\,\text{len}(\hat{\sigma}) + 2r < d$$

This is absurd, since the distance between x and y is d. Since $\frac{d-2r}{2r+1} \leq \text{len}(\hat{\sigma}) \leq \hat{d}$, the lower bound holds. □

Note that theorem 11 is a special case of this one. In fact, since the star decomposition is a ball decomposition with $r = 1$, the bound of theorem 12 becomes

$$\frac{d-2}{3} \leq \hat{d} \leq d$$

which is exactly the bound stated in theorem 11.

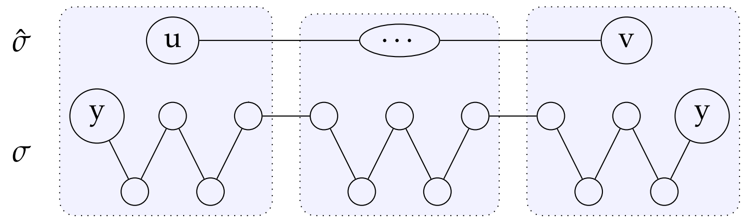


Figure 11.: A path $\hat{\sigma}$ in $\hat{G}$ is expanded to a path σ in G, with $r = 2$. To each node in $\hat{\sigma}$ correspond at most $2r$ arcs in σ.

6.2 SEQUENTIAL ALGORITHM

In this section we study a sequential greedy algorithm to find the ball decomposition of a graph. The algorithm is a generalization of the one described in pseudocode 10.

The idea is the following. First we compute the ball of radius r associated with each node. Then we sort all the nodes according to their ball cardinality. For each uncolored node, starting with the one with the highest ball cardinality, we color all the nodes within its ball, if they are not already colored. The color assigned to a node is the ID of the center of the ball to which it belongs. Once we have colored all nodes, we can merge all the nodes within the same ball in a single node. The whole algorithm is depicted in pseudocode 11. To compute the ball associated to each node one can use, for instance, a simple BFS.

6.3 MAPREDUCE ALGORITHM

The parallelization of the greedy algorithm described in the previous section is non trivial. The main problem is that the algorithm relies on the ordering of the nodes to assign them to the right ball, that is the one with the greatest ball cardinality.

To solve this problem, we can look at it from a different perspective. Instead of assign the node to the ball whose center has the biggest ball cardinality, we make balls from nodes that are not dominated by others. In this context a node v is dominated by another node c if v can be in the ball with center c and c has a ball cardinality bigger than that of v. Saying this is equivalent to impose a partial ordering of the nodes based on their ball cardinality.

The algorithm is structured as follows:

1. Ball computation: the ball associated with each node is computed.

2. Graph coloring: the nodes of the graph are colored with the ID of the center of the ball they belong to.

Algorithm 11: Serial algorithm to compute the ball decomposition of a graph G

```
Input: A graph $G$ and an integer $r$
Output: The ball decomposition of $G$
begin
    foreach node $v$ in $G$ do
        $balls(v) \leftarrow$ the ball of radius $r$ centered in $v$
    end
    $nodes \leftarrow$ list of nodes of $G$ sorted by ball cardinality
    foreach node $c \in nodes$ do
        if $c$ is not colored then
            foreach $v \in balls(c)$, $v$ not colored do
                color $v$ with the $c$'s color
            end
        end
    end
    foreach edge $(u, v)$ of the graph do
        $c_u \leftarrow$ the color of $u$
        $c_v \leftarrow$ the color of $v$
        Create nodes $c_u$ and $c_v$ in the quotient graph
        Add edge $(c_u, c_v)$ to the quotient graph
    end
end
```

3. Graph contraction: the nodes in the same ball are contracted into a single node that inherits all the edges.

In what follows we will describe the MapReduce algorithm for each phase.

Ball computation The ball computation consists in r iterations, where r is the radius of the balls. At iteration i, to each key v is associated the ball of radius i centered in v. In the first iteration, this ball is equal to the neighbourhood of the node.

At the beginning of each iteration the map function sends, for each vertex v, the ball centered in v to each neighbour. The reduce function will then perform the union of the balls. The resulting dataset is then fed as input to the next iteration.

At the end of the r-th iteration each node has received the identifiers of all the nodes it can reach in at most r hops, i.e. its ball of radius r. The algorithm is depicted in pseudocode 12.

Graph coloring In this phase, each node is assigned to a ball. Following point 2 of definition 21, if a node can be part of more than one ball, then we should consider it as belonging to only one of these

Algorithm 12: MapReduce algorithm to compute the balls of radius r associated with each vertex in a given graph.

```
Input: A graph $G$ in the form $(nodeId, neighbourhood)$ and an
       integer $r$.
Output: The balls dataset in the form $(nodeId, ball)$

Map Function sendBalls (nodeId, (neighbourhood, ball)) is
    foreach $v$ in neighbourhood do
        Output ($v$, ball) ;
    end
end
Reduce Function mergeBalls (nodeId, balls) is
    $ball \leftarrow \bigcup_{B \in balls} B$;
    Output (nodeId, ball);
end

begin
    The initial ball associated to each node is its
    neighbourhood;
    for $i \leftarrow 1$ to $r$ do
        Use map function sendBalls;
        Use reduce function mergeBalls;
    end
end
```

balls. As we have done in the serial algorithm, we want the nodes with the greatest ball cardinality to cover as many nodes as possible. Since in a parallel setting we can't rely on an explicit ordering of the nodes by their ball cardinality, we have to adopt a different strategy, an iterative one.

Before describing the algorithm, we define the concept of *domination* in the context of this algorithm. A node v is said to be *dominated* by a node u if both u and v are uncolored and u has a ball cardinality greater than v. In this case we say that u is a *dominator* of v.

The basic idea of the algorithm is to pick at each iteration the nodes that have no dominators and to color the nodes in their balls that are still uncolored.

Each node can be in one of three states: *Uncolored*, *Candidate* and *Colored*. In the *Uncolored* state, the node is not assigned to any ball. The *Candidate* state means that the node is uncolored and is meant to become a ball center in the current iteration. Finally, the *Colored* state means that the node has already been assigned to a ball, hence it can't be colored again.

Each iteration is divided in two steps:

1. Candidates identification: the nodes without dominators are identified through a voting procedure.

2. Covered nodes coloring: the nodes marked as candidates send their color to all the nodes within their balls. The ones among these that are still uncolored are colored.

Let's analyze more in detail the candidate identification phase. Before selecting a node to become a ball center, we have to make sure that it can't included in another ball. Since in our greedy algorithm a node v can be included only in balls whose center has a ball cardinality greater than v, we have to make sure that v has no dominators. This means that we have to check that either there are no nodes with a ball cardinality greater than v or that all the nodes with a ball cardinality greater than v are already colored. To do so, at the beginning of each iteration all the nodes send their ball cardinality to the nodes within their balls, along with a vote. This vote, sent by node v, is used by the nodes in the ball centered in v to tell if they are dominated by v. Thus, if a node is colored it sends a positive vote, otherwise it sends a negative vote. A positive vote, for the node receiving it, has the meaning of approval to become a center. The vote sending can be done with a map function (function `sendVotes` in pseudocode 14).

Once a node has received all the votes, it can check if all the ones coming from nodes with a higher ball cardinality are positive. In case they are all positive, the node is marked as a candidate. Otherwise, it means that the node has a dominator, hence it can't become a ball center in this iteration. This can be done with a reduce function (function `markCandidate` in pseudocode 14).

After the candidate identification phase, we can color the nodes covered by the new ball centers. With a map function (`colorDominated`, pseudocode 14), the nodes marked as candidate send their ID and ball cardinality to all the nodes within their balls. Then, with a reduce function, the nodes that are not yet colored and that received one or more colors are colored with the color associated with the biggest cardinality. Ties on the cardinality associated to a color are broken by taking the one with the highest ID.

After all the nodes are colored, each node is assigned to exactly one ball. Pseudocode 13 illustrates this algorithm.

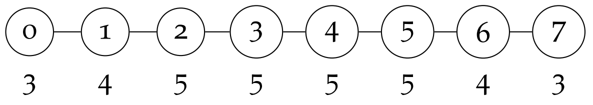
Graph contraction The last step of the algorithm is a simple constant-round algorithm. What we aim to do is to replace, for each edge, the source and destination IDs with the color they were assigned. To accomplish this task, we should first represent the graph as a collection of edges, that is a dataset made up by pairs of IDs: one for the source node of an edge and one for its corresponding destination. Then, to relabel the nodes, we proceed in two rounds.

In the first round the edges and the colors are paired on the source ID of each edge. The source ID will then be replaced with its color. The second round pairs the edges and the colors on the destination ID of each edge, that will then be replaced by its color.

Algorithm 13: MapReduce algorithm to color a graph given the balls. Functions described in pseudocode 14 are used

```
begin
    all nodes start with an Uncolored status;
    the dataset of tagged nodes is called taggedGraph;
    while there are uncolored nodes do
        votes ← Use the vote map function on taggedGraph;
        groupdVotes ← Cogroup(taggedGraph, votes);
        taggedGraph ← Use markCandidate on groupedVotes;
        newColors ← Map colorDominated on taggedGraph and
        then reduce with max;
        taggedGraph ← Cogroup(taggedGraph, newColors) and
        then map applyColors;
    end
end
```

Figure 12.: A linear array. The number inside each node is its ID, whereas the number below is the cardinality of the associated ball of radius 2.



At the end of this process we will have all the edges with sources and destinations relabeled with their respective colors. By removing all duplicates we obtain the quotient graph as a collection of edges. Pseudocode 15 describes the algorithm.

6.3.1 *Complexity*

We will now analyze the number of rounds required by the algorithm to complete, in the worst case. The ball computation in the first phase of the algorithm takes up a number of rounds proportional to the radius r. Hence, for a fixed value of r, it takes a constant number of rounds.

The second part of the algorithm is more tricky. Each iteration of the voting procedure can be performed in a constant number of rounds. We shall thus determine the number of iterations performed by the algorithm in the worst case. Consider a linear array, like the one shown in figure 12.The numbers inside the vertices are the node IDs, whereas the numbers below them are the ball cardinalities (including the center).

At the first iteration of the algorithm, the only node that gets selected is node 5: it has ball cardinality 5. The other nodes with cardinality 5 are not selected because ties are broken deterministically

Algorithm 14: Map and Reduce functions used to compute the ball decomposition

```
Map Function vote (nodeId, (status, ball)) is
    cardinality ← size of ball;
    if status = Colored then
        foreach n in ball do Output (n, (true, cardinality));
    else
        foreach n in ball do Output (n, (false, cardinality));
    end
end

Reduce Function markCandidates (nodeId, ((status, ball) ,votes))
is
    if status = Colored then
        Output (n, (color, ball))
    else
        valid votes ← votes from nodes with higher ball
        cardinality;
        if all valid votes are positive then
            Output (n, (Candidate, ball));
        else
            Output (n, (Uncolored, ball));
        end
    end
end

Map Function colorDominated (nodeId, (status, ball)) is
    if status = Candidate then
        cardinality ← ball size;
        foreach n, in ball do Output (n, (nodeId, cardinality));
    end
end

Reduce Function max (nodeId, colors) is
    color ← the color with the maximum associated ball
    cardinality, breaking ties by id;
    Output (nodeId, color);
end

Map Function applyColor (nodeId, ((status, ball), color)) is
    if status = Colored then  Output (nodeId, (old color, ball));
    else  Output (nodeId, (color, ball)) ;
end
```

Algorithm 15: Algorithm to compute the quotient graph of the ball decomposition, given the adjacency matrix representation of the graph and the colors assigned to each node

```
Map Function relabel (nodeID, (color, edgeEndPoints)) is
    foreach n in edgeEndPoints do
        Output (n, color);
    end
end

begin
    groupedSources ← Cogroup(colors, edges) ;
    relabeledSources ← apply relabel to groupedSources ;
    groupedDestinations ← Cogroup(colors, edges) ;
    relabeledDestinations ← apply relabel to
    groupedDestinations ;
end
```

by ID. In this way they are all potentially dominated by the higher ID nodes. Thus at the first iteration only node 5 is selected. It then colors all the nodes in its ball: 3, 4, 5, 6 and 7. In the second iteration only node 2 is selected and colors nodes 0, 1 and 2. At this point the algorithm has colored all the nodes and ends. Consider now the case of a linear array with n nodes and let r be the chosen ball radius. As for the linear array with 8 nodes, the only node selected in the first iteration is the one with ID $n - r$. In the second iteration the only node selected will be the one with ID $n - 2r$. At the i-th iteration the node $n - ir$ will be selected. The number of iterations to complete the algorithm in the case of the linear array is thus $O(n)$.

The linear array however is a pathological case. Real world graphs expose much more parallelism. We shall see that experiments shows that the algorithm has good scalability for real world graphs.

6.3.2 *Randomized algorithm*

The algorithm analyzed in the last section has a bad theoretical complexity. This is mainly due to its iterative nature and the fact that ties are broken deterministically by ID. To achieve a better running time we resort to a randomized algorithm. The idea is to select ball centers at random and then to make them color all the nodes within their balls.

First of all, balls are computed using algorithm 12. After balls have been computed, we proceed to the coloring phase.

The first step consists in ball centers selection. Each node becomes a ball center with probability p. This can be done with a single map function: for each node the output of the map function will be the node marked with *Candidate* with probability p or *Uncolored* with probability $1 - p$.

In the second step, with a map function, all the nodes that have been marked as candidates send their own color, along with their ball cardinality, to all the nodes within their balls. A reduce function will then, for each node, apply the color with the highest cardinality among the ones received. Ties are broken by ID.

In the reduce function it may happen that a node has received no colors, because it isn't included in any of the randomly chosen balls. In this case the node is colored with its own ID.

This algorithm is depicted in pseudocode 16. After the nodes have been colored, the graph is relabeled using algorithm 15.

Even if ties are still broken by ID, the randomization step at the beginning of the algorithm ensures that a situation like the one of the linear array for the deterministic algorithm is unlikely to happen.

This algorithm runs in a constant number of rounds, since no iteration is involved. However the cardinality reduction is worse than the one achieved with the deterministic algorithm.

6.4 EXPERIMENTAL RESULTS

In this section we present some experimental results. The data was obtained by running the algorithms described in this section against the datasets described in appendix A. First we will present some results that show the cardinality and diameter reduction of the datasets after applying the ball decomposition algorithm. Next, we compare the cardinality reduction of the deterministic and randomized versions of the ball decomposition. Finally, we show some performance and scalability measures, that shows that the algorithms achieve in practice a good scalability for big datasets.

6.4.1 *Cardinality and diameter reduction*

In this section we present some experimental results related to the application of the ball decomposition algorithm to various datasets presented in appendix A. In general we can state that the ball decomposition gives significant cardinality reductions even for small values of the ball radius r. Moreover the experiments show that the diameter reduction is often smaller than the theoretical bound. Thus the ball decomposition appears to be a viable technique to approach the diameter problem. In what follows we will analyze the experimental results for each dataset. Each figure shows in the left panel the quotient graph cardinality versus the value of r, whereas the right panel shows the diameter of the quotient graph (blue line) and the theoretical bound (red line) versus the value of r.

Algorithm 16: Randomized algorithm for the ball decomposition.

```
Input: A graph, with ball of radius $r$ already associated to
       each node.

Map Function colorDominated (nodeId, (status, ball)) is
    if status = Candidate then
        cardinality ← ball size;
        foreach n, in ball do Output (n, (nodeId, cardinality));
    end
end

Reduce Function max (nodeId, colors) is
    color ← the color with the maximum associated ball
    cardinality, breaking ties by id;
    Output (nodeId, color);
end

Map Function applyColor (nodeId, ((status, ball), color)) is
    if status = Candidate or no color received then
        Output (nodeId, (nodeId, ball));
    else
        Output (nodeId, (color, ball));
    end
end

begin
    taggedGraph ← Mark nodes as Candidate with probability $p$;
    newColors ← Map colorDominated on taggedGraph and
    then reduce with max;
    taggedGraph ← Cogroup(taggedGraph, newColors) and then
    map applyColors;
end
```

Amazon Applying the ball decomposition algorithm to the Amazon dataset yields the results shown in figure 13. The cardinality reduction is sensible: with $r = 2$ the quotient graph has 28629 nodes, when the original graph had 334863 nodes, a reduction of more than 11 times. The diameter of the reduced graph is 19, while the original graph had a diameter of 44. The theoretical lower bound for $r = 4$ given by theorem 12 is 8. Thus in this case the diameter of the reduced graph is much better than the lower bound.

We can try to reduce the size of the graph even more: $r = 4$ yields a quotient graph with 9084 nodes (37 times smaller than the original) and a diameter of 16 (the theoretical bound is 7,2).

Autonomous systems The effects of the application of the ball decomposition algorithm on the Autonomous Systems dataset are shown in figure 14. The original graph has 6474 nodes and a diameter of 9. The quotient graph with $r = 3$ has only 106 nodes and a diameter of 5. The number of nodes is reduced more than 60 times, while the diameter of the quotient graph is bigger than the half of the original diameter. This graph has an interesting behaviour, in that the diameter of the quotient graph for $r \in \{1, 2, 3\}$ does not change.

DBLP The quotient graph of the ball decomposition applied to the DBLP dataset (fig. 15), with $r = 2$, has a 26099 nodes and a diameter of 11. The cardinality of the original graph is 317080. Its diameter is 21. In this case the theoretical lower bound is 3.4. Thus, also in this case, the diameter for the ball decomposition is much better than the theoretical lower bound.

Like the Autonomous Systems case, the diameter of the graph for $r = 1$ and 2 is the same: 11. If we wish to reduce the graph's size even more, by setting $r = 3$ we get a graph with 14945 nodes (21 times smaller than the original) and with a diameter of 9 (the theoretical bound is 3.75).

Californa Roads The cardinality reduction of the California roads dataset (fig. 16) is surprisingly regular, for increasing values of r. Recall that the original graph had 1965206 nodes and a diameter of 849. The ball decomposition graph with $r = 2$ has only 365512 nodes. The diameter in this case is 323. The theoretical lower bound is 218.

Pyweb syntetic graphs Figures 17, 18, 19 and 20 show the cardinality and diameter reductions for the four synthetic power law graphs. In all cases the cardinality reduction is more than 6 times, even with a ball radius of 1. The diameter reduction, on the other hand, is less than a half with $r = 1$, in every case.

Let's analyze more in detail the results obtained in these experiments. As expected, increasing the value of r results in smaller quotient graphs. This can be used in practice to achieve arbitrary graph size reductions. The drawback of using high values of r is that, together with the cardinality of the graph, also the diameter becomes smaller. However the diameter of the quotient graph is always greater than a well defined lower bound, as predicted by theorem 12. In many cases, the diameter reduction is much less than the one predicted by the theorem. This lets us hope that a more strict lower bound can be found.

6.4.2 *Cardinality reduction given by randomized algorithm*

The randomized ball decomposition algorithm is expected to run faster than the deterministic one, however it gives worse results on

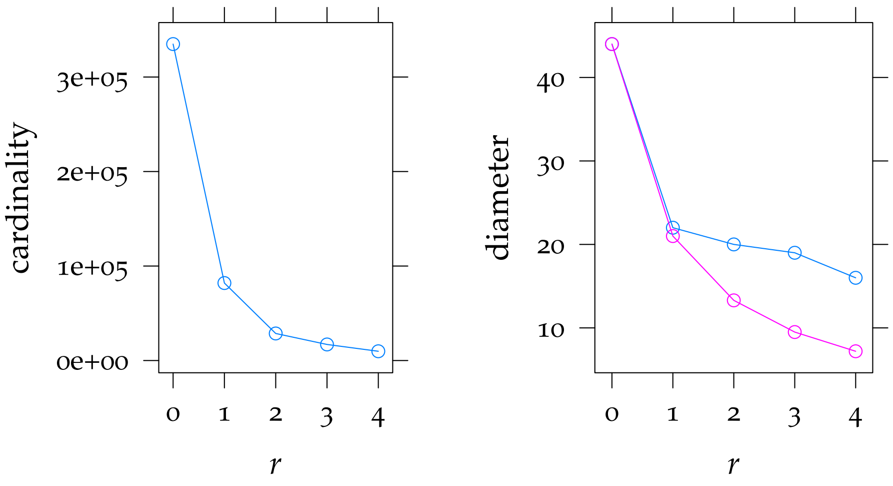


Figure 13.: Cardinality and diameter reduction for the Amazon dataset after applying the ball decomposition with radius r

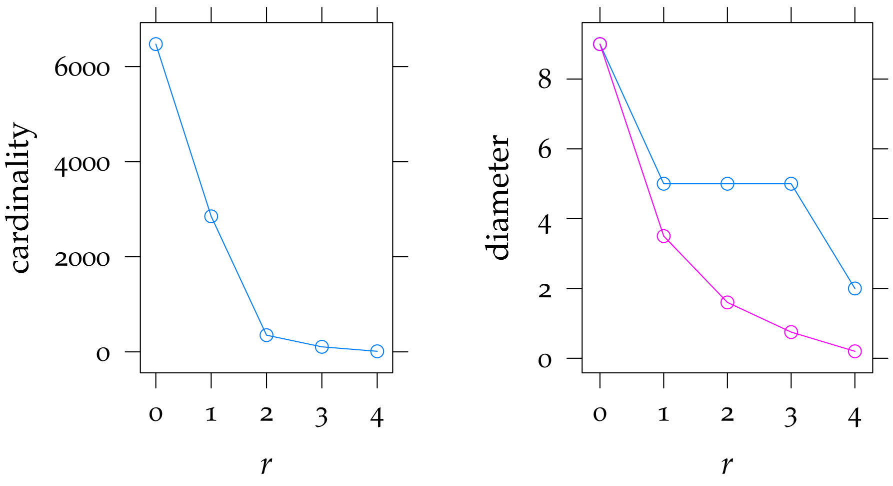


Figure 14.: Cardinality and diameter reduction for Autonomous Systems after applying the ball decomposition with radius r

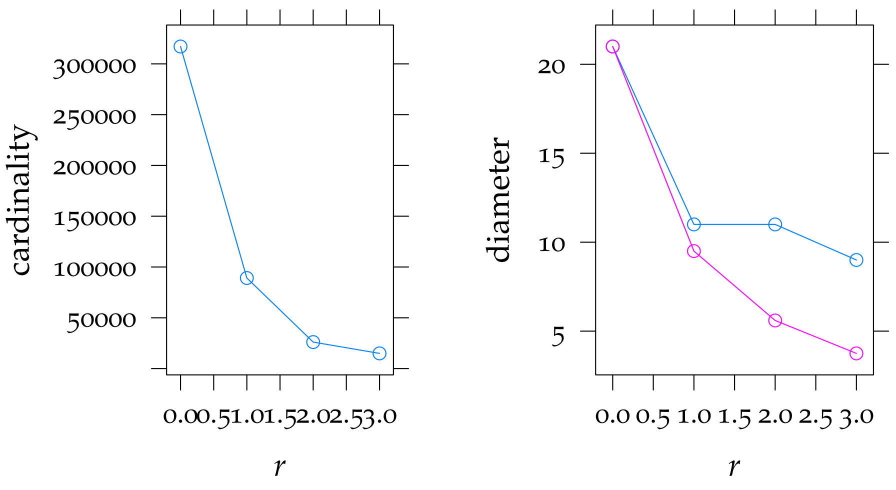


Figure 15.: Cardinality and diameter reduction for DBLP after applying the ball decomposition with radius r

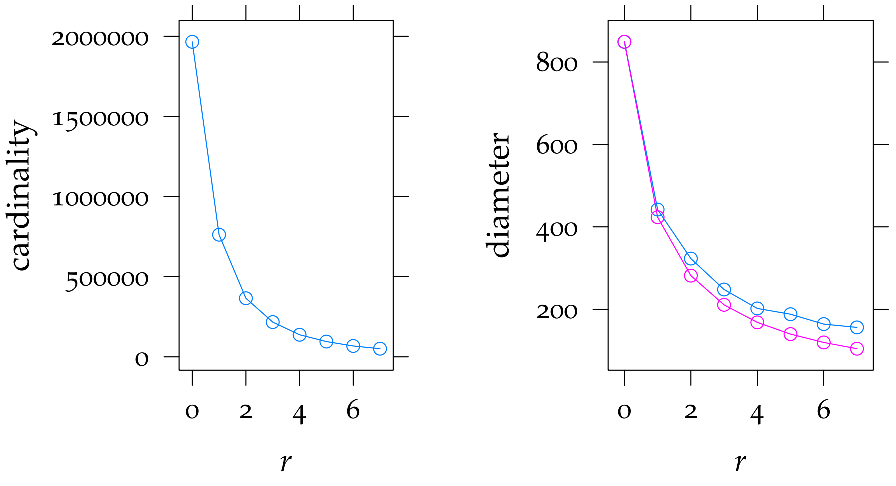


Figure 16.: Cardinality and diameter reduction for California Roads after applying the ball decomposition with radius r

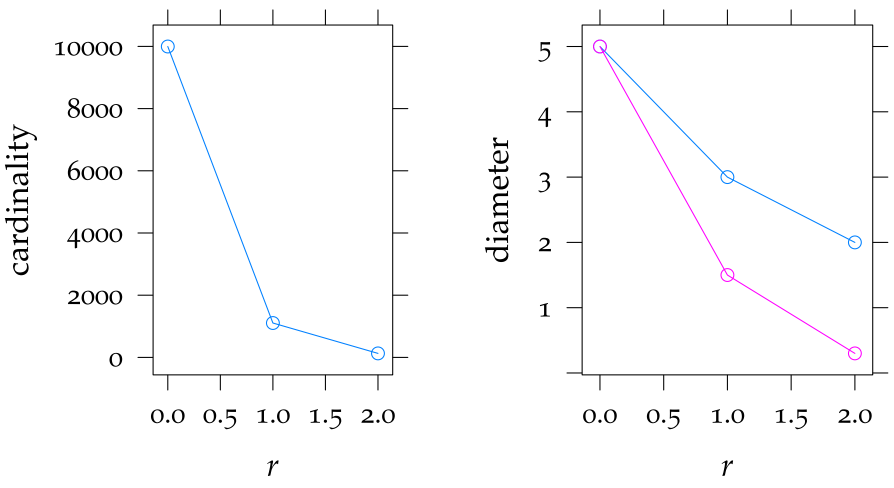


Figure 17.: Cardinality and diameter reduction for pyweb 1 after applying the ball decomposition with radius r

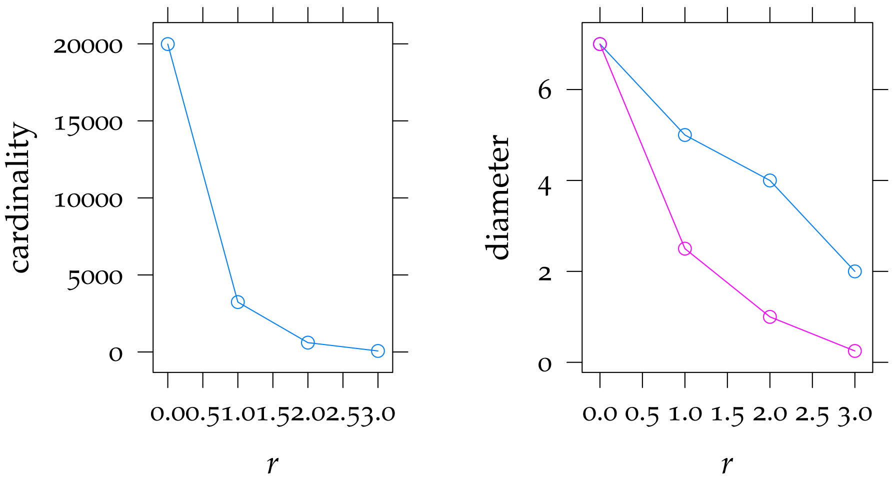


Figure 18.: Cardinality and diameter reduction for pyweb 2 after applying the ball decomposition with radius r

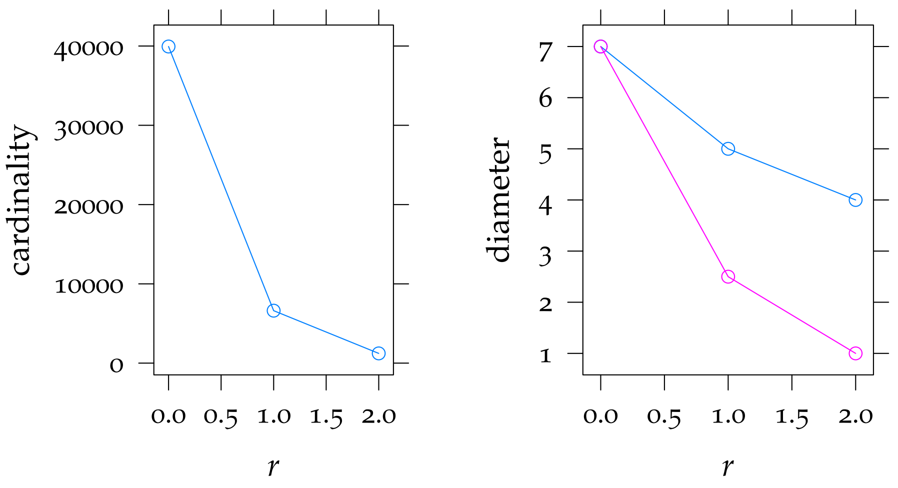


Figure 19.: Cardinality and diameter reduction for pyweb 3 after applying the ball decomposition with radius r

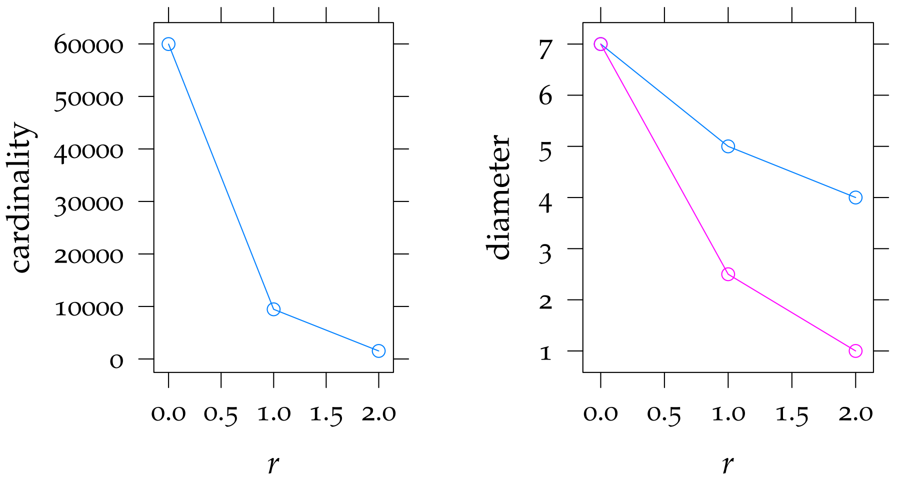


Figure 20.: Cardinality and diameter reduction for pyweb 4 after applying the ball decomposition with radius r

Dataset	Original	Ball dec.	$p = 0.1$	$p = 0.2$	$p = 0.3$
Aut. Systems	6474	2851	5535	4794	4039
DBLP	317080	89177	213182	184182	178453
Amazon	334863	82007	226069	190268	184385
Youtube	1134890	527871	909414	822292	794522
Roads CA	1965206	761897	1558428	1371384	1312539

Table 8.: Comparison between the cardinality reductions given by the deterministic ball decomposition algorithm (third column) and by the randomized one (last three columns), for $p \in \{0.1, 0.2, 0.3\}$

the side of cardinality reductions. Experimental results are shown in table 8. The cardinality of the quotient graph given by the randomized ball decomposition algorithm is always greater than the cardinality of the one given by the deterministic algorithm, for each chosen value of p. For each dataset the quotient graph with the smallest cardinality given by the randomized algorithm is between 1.4 and 2.2 times bigger than the quotient graph given by the deterministic algorithm.

6.4.3 *Performance and scalability*

As we saw in section 6.3.1, the theoretical performance of the algorithm is quite bad. However the worst case for the algorithm (the linear array), is a pathological case. Real world graphs expose more parallel work for the algorithm to perform. Figures 21, 22, 23, 24 and 25 show performance measurements for these real world graphs, in logarithmic scale. The blue line shows the performance of the deterministic ball decomposition algorithm. Magenta, green and red lines show the performance of the randomized algorithm for $p = 0.1$, $p = 0.2$ and $p = 0.3$ respectively, where p is the probability for a node to become a center.

All tests have been performed with the software described in appendix B.2 on a Power 7 machine with 128 GB of RAM and 48 cores, running SUSE Linux 11 SP2.

As the figures show, increasing the number of cores improves performance up to a point where the overhead of managing many cores and moving data between them overcomes the benefits of the parallel processing. For small graphs this point is early on the number of processors axis: as soon as we use more than 5 cores, the program is actually slower (fig 21), due to the framework overhead. When dealing with larger graphs, using up to 16 cores brings performance gains before the overhead becomes dominant (fir 22, 23 and 24). Figure 25

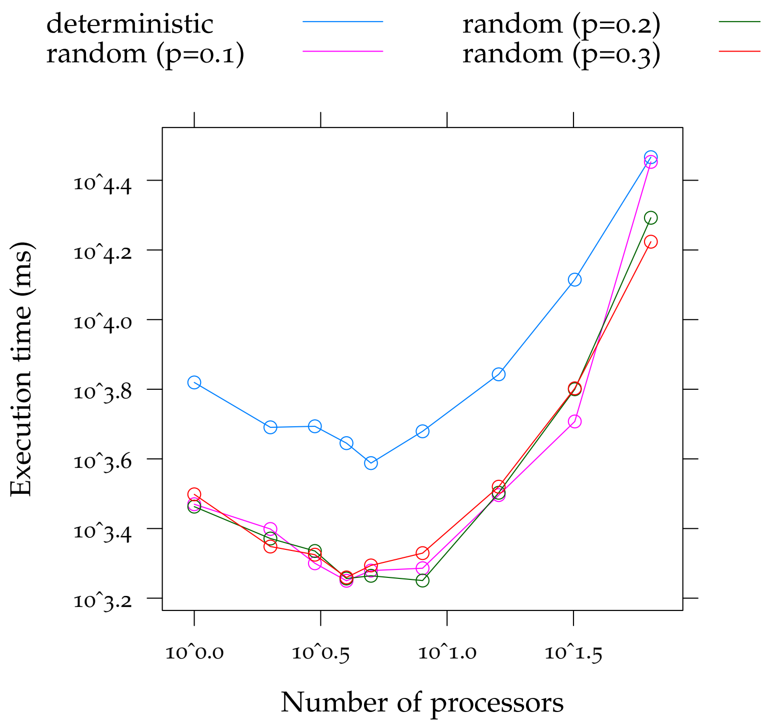


Figure 21.: Execution time versus number of processors for the Autonomous Systems dataset.

shows how on the Youtube graph (1134890 nodes, 2987624 edges), the algorithm scales up to 64 processors, at least for the deterministic algorithm.

This trend in the relation between the dataset size and the scalability of the algorithm suggests that when dealing with even larger graphs, the algorithm can take full advantage of the processing power of the computing infrastructure at our disposal.

We can compare the performance of the ball decomposition shown in this section with the performance of our own implementation of D-HANF, analyzed in section 3.3.5. For instance, the time taken by D-HANF to compute the diameter of the DBLP dataset with 16 processor is approximately 81 seconds. The time to compute the ball decomposition of the same graph with the same number of processors is approximately 52 seconds. This last time does not take into account the time taken to compute the diameter of the quotient graph. However, being the quotient graph significantly smaller, the performance advantage of the ball decomposition is clear. For the Amazon dataset, D-HANF takes 148 seconds to complete whereas the ball decomposition takes 92 seconds.

Both implementations are still not optimized and there's room for a lot of improvement, however these first results motivate further work on the path of the ball decomposition.

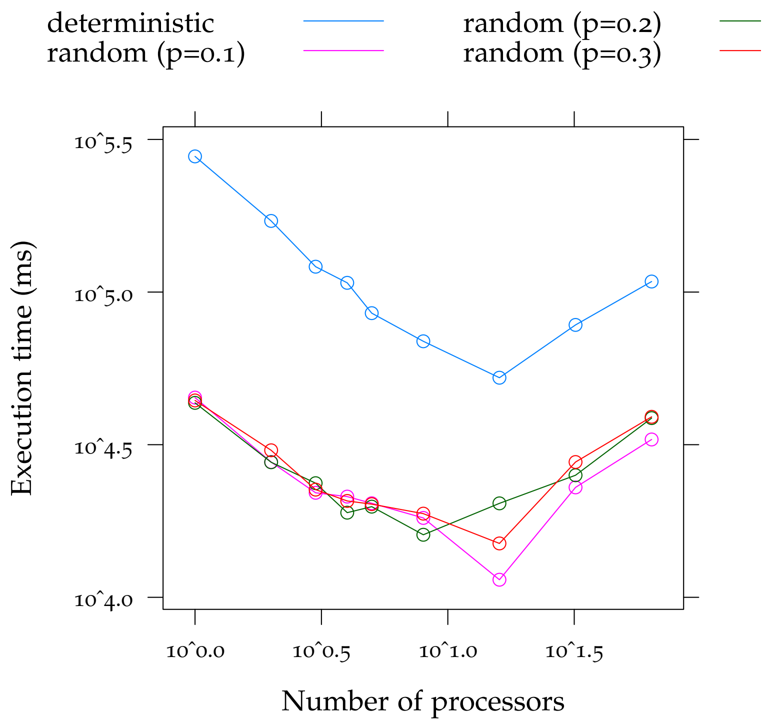


Figure 22.: Execution time versus number of processors for the DBLP dataset.

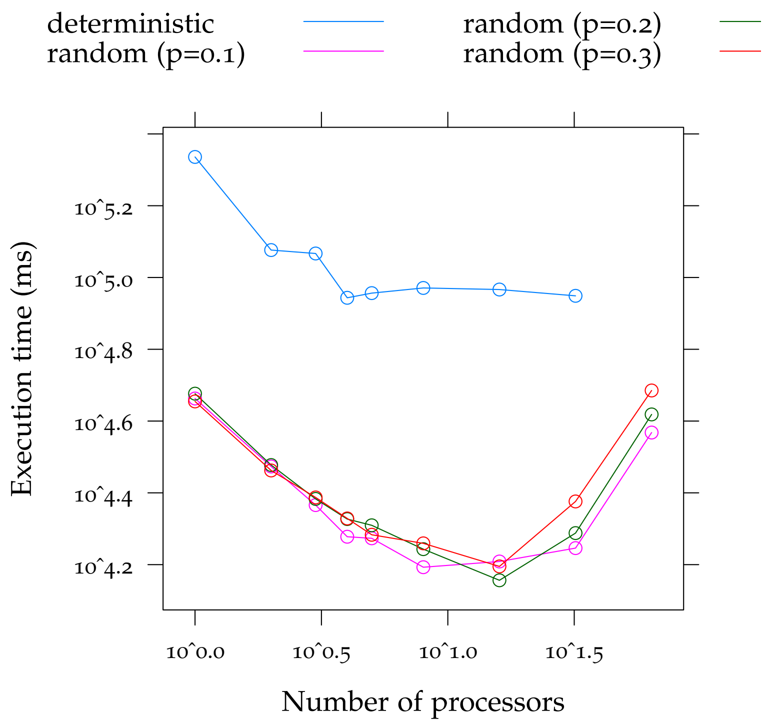


Figure 23.: Execution time versus number of processors for the Amazon dataset.

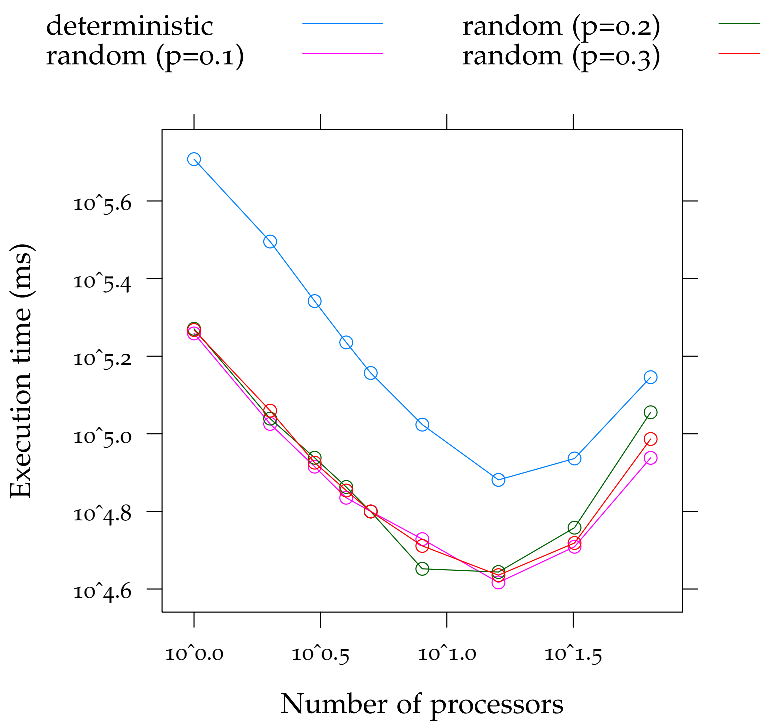


Figure 24.: Execution time versus number of processors for the Roads of California dataset.

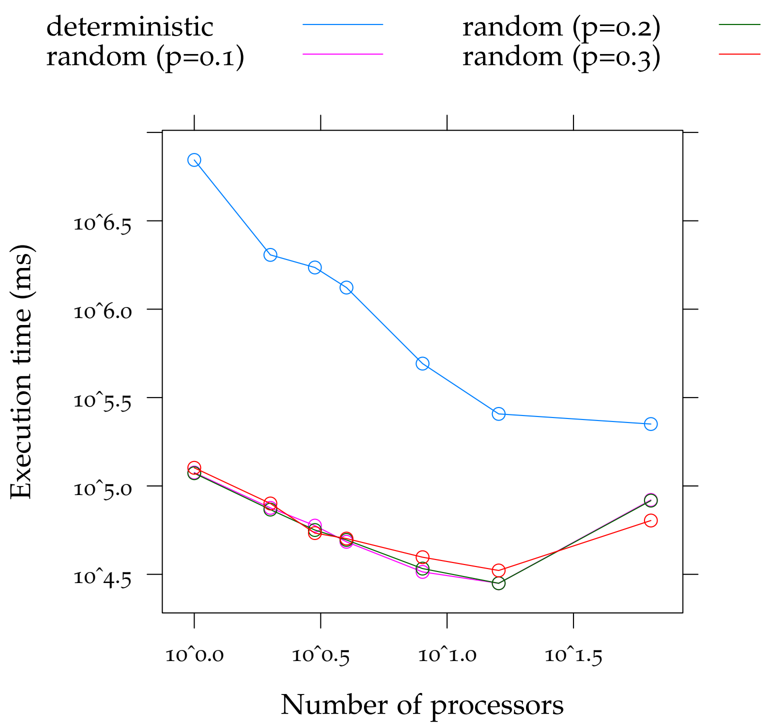


Figure 25.: Execution time versus number of processors for the Youtube dataset.

7 CONCLUSIONS

The diameter is an important metric when studying graphs. Due to the large size of graphs that are studied in a wide variety of contexts (Web, social networks, citation networks, biological networks, etc.) parallel algorithms that can leverage the power of distributed systems are needed.

A computational paradigm that is gaining more and more popularity is MapReduce [DG08]. In this paradigm algorithms are expressed as a series of *map* and *reduce* functions, much like in functional languages such as Lisp. Implementations of this paradigm [Whi12, ZCF+10] can then distribute the computation across clusters of commodity machines, with the purpose of scalability and fault tolerance.

In this thesis we studied solutions to the diameter problem in a MapReduce setting. A few approaches were already available, from the classic textbook algorithm to probabilistic iterative algorithms. Beside studying these approaches, we proposed a novel approach, based on the size reduction of the graph. To test the scalability and accuracy of our algorithms, we implemented a software, with the Spark framework, to deal with large graphs. Using this software, we tested our algorithms against several real world datasets.

In chapter 2 we adapted the classic textbook algorithm to the MapReduce paradigm. This approach involves the solution of the All Pairs Shortest Path problem: once we have the shortest paths between any two nodes, it is trivial to get the longest one, i.e. the diameter. To solve the APSP problem we resorted to the repeated squaring of the incidence matrix of the graph in the $(\mathbb{N}, \min, +)$ semiring. Since the matrix multiplication can be performed in a constant number of rounds [PPR+12], the algorithm has a $O(\log d)$ complexity, where d is the diameter of the graph. If one is interested in the effective diameter d_α, the complexity is $O(log d_\alpha)$.

In chapter 3 we studied several iterative probabilistic approaches to the diameter problem. All these algorithms [PGF02, BRV11, KTA+08] follow a common pattern. They all use probabilistic counters [FMM85, FFGea07] to approximate the neighbourhood function of a graph. Given the neighbourhood function of a graph, an approximation of the diameter or the effective diameter is easily derived. We proposed and implemented a parallelization of [BRV11], namely D-HANF. All these algorithms run in $O(d)$ time, where d is the diameter of the graph.

The rest of the thesis (chapters 4, 5 and 6) was devoted to the study of the graph size reduction approach. The idea is to reduce the size of the graph in parallel, while retaining guarantees on the diameter of the quotient graph, to solve the diameter problem on the reduced graph, possibly on a single machine.

The modular decomposition (chapter 4) is a promising technique from a theoretical point of view, in that it guarantees that the diameter in the quotient graph is the same as in the original graph. However we experimentally found that the cardinality reductions achieved with this approach are not significant. Thus, despite the appealing property of keeping the diameter unchanged, this approach is not interesting from a practical point of view.

By accepting a degradation of the diameter we were able to achieve much greater reductions in the graph's size. Chapters 5 and 6 described the approaches that enabled us to achieve this result. The idea is to decompose the graph in stars (the *star decomposition*) or balls of radius r (the *ball decomposition*). Each star or ball is replaced, in the quotient graph, by a single node. We have proved, in the more general case of the ball decomposition, that the diameter of the quotient graph is subject to the following bound

$$\frac{\mathrm{diam}(G) - 2r}{2r + 1} \leq \mathrm{diam}(\hat{G}) \leq \mathrm{diam}(G)$$

where r is the radius of the balls (corollary 12). In the ball decomposition, the parameter r allows to trade accuracy on the diameter with greater graph's size reductions. The worst case running time of the MapReduce algorithms is $O(n)$, however we have experimentally shown that in practice these algorithms achieve a good scalability, since real world graphs expose much more parallelism than the pathological case of the linear array.

The graph decomposition approach leaves many open paths for future research. There are mainly two directions to improve the approach: the bound on the diameter and the algorithms.

As for the bound on the diameter, the current one ignores the structure of single balls. To make it tighter we could relate it to the diameter of single balls or some other structural properties of the balls.

The algorithms that compute the ball decomposition suffer from a bad worst case complexity. An algorithm with stronger theoretical guarantees on the running time is demanded. An algorithm with a constant round running time would be ideal.

From the iterative probabilistic algorithms point of view, there is room to experiment with different counters other than the ones described in [FFGea07, FMM85]. For instance there are counters based on order statistics [Gir09] or on a balls and bins approach [Nel11].

Finally, some argue [MAB$^+$10] that MapReduce is not ideal for large scale graph processing. In particular, the functional heritage of the paradigm imposes to explicitly pass the structure of the graph from one round to another. In [MAB$^+$10] a different paradigm, reminiscent of the Bulk Synchronous Parallel model, is proposed. This paradigm, called Pregel, structures the computation as a series of *supersteps*. In each superstep a user-defined function is run on each vertex, yielding a set of messages to be delivered to other vertices for the next superstep. An interesting research path is to express the algorithms for the diameter computation in such a paradigm.

A DATASETS

To conduct the experiments described in later chapters, several datasets have been used. In this appendix we describe each dataset.

A.1 FACEBOOK DATASET

The Facebook dataset is made up by 'friends lists' from Facebook. The dataset is taken from the Stanford Large Network Datasets collection [1] All data has been collected on a voluntary basis using a Facebook app [2]. Table 9 shows the statistics of this dataset

A.2 AUTONOMOUS SYSTEMS DATASET

The Autonomous Systems dataset represents the connections between the autonomous systems that make up the Internet. Each Autonomous System exchanges traffic flows with some neighbors. We can construct a communication from the BGP (Border Gateway Protocol) logs. The dataset is available at the Stanford Large Neworks Dataset collection[3]. Table 9 shows statistics about this graph.

A.3 DBLP DATASET

The DBLP database provides bibliographic information on major computer science journals and proceedings[4]. This dataset is a co-authorship network. Every node represents an author. Two authors are connected together if they publish at least one paper together. Table 9

1 http://snap.stanford.edu/data/egonets-Facebook.html
2 https://www.facebook.com/apps/application.php?id=201704403232744
3 http://snap.stanford.edu/data/as.html
4 http://dblp.uni-trier.de/

	Nodes	Edges	Diameter	Diameter at 0.9
Facebook	4039	88234	8	4.7
Autonomous Systems	6474	13895	9	4.6
DBLP	317080	1049866	21	8
Amazon	334863	925872	44	15
Youtube	1134890	2987624	20	6.5
Roads of California	1965206	2766607	849	500

Table 9.: Statistics for real world datasets.

	Nodes	Edges	Diameter	Diameter at 0.9
Power Law 1	9996	378035	5	2.85
Power Law 2	19985	391393	7	3.54
Power Law 3	39938	788736	7	3.72
Power Law 4	59974	1186032	7	3.75

Table 10.: Statistics for randomly generated power law datasets

shows statistics about this dataset. The source of the dataset is the Stanford Large Network Dataset collection[5].

A.4 AMAZON DATASET

Amazon is an on-line store selling various goods all over the world. This graph is taken from the Stanford Large Network Dataset collection[6]. Each node in this graph represents a product. Two products are connected by an edge if they are frequently purchased together. Statistics for this graph are shown in table 9.

A.5 YOUTUBE DATASET

Youtube is a video sharing platform with social network features. In the Youtube social network, people form friendship with each other. In this graph each node represents a user. Two users are connected if they are friends. The dataset is taken from the Stanford Large Network Dataset collection[7]. Table 9 shows statistics for this graph.

A.6 CALIFORNIA ROAD NETWORK DATASET

This graph represent the road network of California. Each node represents an intersection or endpoint. Each road is represented as an undirected edge. The source of this dataset is the Stanford Large Network Dataset collection[8]. Statistics for this graph are shown in table 9.

A.7 RANDOM POWER LAW DATASETS

Since many real world graphs seem to follow a power law distribution of degrees these datasets have been generated randomly using pywebgraph[9]. This software allows to generate arbitrary sized power

5 http://snap.stanford.edu/data/com-DBLP.html
6 http://snap.stanford.edu/data/com-Amazon.html
7 http://snap.stanford.edu/data/com-Youtube.html
8 http://snap.stanford.edu/data/roadNet-CA.html
9 http://pywebgraph.sourceforge.net/

law graphs. We have generated four datasets, with different sizes: approximately 10000, 20000, 40000 and 60000 nodes each. These datasets are available for download[10]. Table 10 shows statistics about all the four datasets.

10 http://www.dei.unipd.it/∼ceccarel/datasets

B SOFTWARE

In this appendix we will briefly describe the implementation of the algorithms described in this thesis. We will first give details about the implementation of the sequential version of the star and ball decomposition and then we will describe the parallel implementation of the ball decomposition.

B.1 IMPLEMENTATION OF SEQUENTIAL ALGORITHMS

The sequential algorithms have been implemented within the Sage framework[1]. The sofware is available under the GPL-3 license at Github[2].

Sage [S+13] is a free and open source mathematics software. It provides, among its other features, a library to deal with graphs. Sage is written in C and Python, with a Python API. We choose this software both because it included many useful graph algorithms and because of the ease of use of Python. These characteristics were ideal during the prototyping phase, where performance is not an issue.

In Sage a graph is represented as a `Graph` object. This object allows to access the the neighbourhood of a vertex with using a syntax similar to array and list access: `graph[id]`.

A vertex of a sage graph can be labeled with an arbitrary object, using the function `graph.set_vertex(id, label)`. The label can be later retrieved with `graph.get_vertex(id)`. In our implementation, the label attached to each node is the color of the node. This color, for a node v belonging to a ball or star of center c, corresponds to the ID of the node c.

Once we have all nodes colored, the quotient graph is produced by shrinking the original graph. Listing 1 shows the python code performing this shrinking. The `quotient` graph is created by adding edges between nodes with the color of each node of the input graph. After this shrinking is performed, a relabeling is necessary, in order to have all the node IDs into the range $[0, n]$.

The star decomposition of a graph is computed by the function described in Listing 2. First the vertices of the graph are sorted in decreasing degree order. Then, for each vertex in this sorted list, if the vertex is not already colored it colors all the nodes within its star. Once the graph is completely colored, we call the `shrink_graph` function to get the quotient graph.

1 `http://sagemath.org`
2 `https://github.com/Cecca/sage-graph-decompositions`

Listing 1: Function to shrink a colored graph.

```python
def shrink_graph(graph):
    quotient = Graph()
    for v in graph.vertices():
        # The star containing ‘v‘
        star_v = graph.get_vertex(v)
        for neigh in graph[v]:
            # The star containing the neighbour of ‘v‘
            star_neigh = graph.get_vertex(neigh)
            if star_v != star_neigh:
                quotient.add_edge(star_v, star_neigh)

    # now a relabeling is necessary
    print 'Relabeling quotient graph'
    quotient.relabel()

    return quotient
```

Listing 2: Sequential python implementation of the star decomposition algorithm.

```python
@print_timing
def star_decomposition(graph):
    print 'Sorting vertices'
    verts = sorted(graph.vertices(), key=graph.degree, reverse=True)
    print 'Coloring graph'
    for v in verts:
        if graph.get_vertex(v) == None:
            graph.set_vertex(v, v)
            for neigh in graph[v]:
                if graph.get_vertex(neigh) == None:
                    graph.set_vertex(neigh, v)
    print 'Building stars'
    quotient = shrink_graph(graph)
    return quotient
```

Listing 3: Sequential python implementation of the ball decomposition algorithm.

```python
def get_ball(graph, vertex, radius):
    return graph.breadth_first_search(vertex, distance=radius)

@print_timing
def ball_decomposition(graph, radius):
    print 'Building balls'
    ball_cardinalities = dict()
    for v in graph.vertices():
        ball_cardinalities[v] = len(list(get_ball(graph,v,radius)))
    print 'Sorting vertices'
    # this sorting is to break ties in favor
    # of the node with the highest ID
    vertices = sorted(graph.vertices(), reverse=True)
    vertices = sorted( vertices,
                       key=lambda x: ball_cardinalities[x],
                       reverse=True)
    print 'Coloring graph'
    for v in vertices:
        if graph.get_vertex(v) == None:
            graph.set_vertex(v, v)
            for u in get_ball(graph, v, radius):
                if graph.get_vertex(u) == None:
                    graph.set_vertex(u, v)
    print 'Shrinking graph'
    return shrink_graph(graph), ball_cardinalities
```

Finally the ball decomposition implementation is given in Listing 3. First, using the subroutine `get_ball`, we populate a dictionary, where the key is the vertex ID and the value is the cardinality of its associated ball. Then the vertices are sorted in decreasing ball cardinality order, using the values stored in the dictionary. Then the nodes are colored using the same procedure of the star decomposition, with the only difference that a node that become center colors all the nodes within its ball. At the end of the coloring phase the quotient graph is built using the `shrink_graph` function.

All these functions are compiled to C code by Sage at load time. In this way they are much faster than standard, interpreted python. This enabled us to leverage the expressiveness and ease of use of Python without sacrificing too much performance.

For the bigger datasets it was impossible to compute the diameter exactly using Sage, due to software limitations. In these cases we resorted to the use of the WebGraph software [BV03][3]. This software implements the HyperANF algorithm described in [BRV11], enabling us to deal with large graphs on a single machine.

3 `http://webgraph.di.unimi.it/`

B.2 IMPLEMENTATION OF PARALLEL ALGORITHMS

The parallel algorithms described in this thesis have been implemented using the Spark framework[4] [ZCF$^+$10]. We choose this framework over the more popular Hadoop for the following reasons:

- It is explicitly tailored for iterative applications, like many algorithms in this thesis. In fact, to support algorithms that work repeatedly on the same dataset, it caches it in the distributed memory, instead of dumping it to disk every time.

- It's written in Scala. Being MapReduce a paradigm with a functional flavor, a functional language is better suited to develop MapReduce applications than an Object Oriented language like Java. In particular Scala's support for anonymous functions (also called *closures* or *lambda functions*) and higher order functions makes writing MapReduce programs considerably more concise and rapid.

- Spark programs can be easily set up to run on a single multicore machine, like the Power7 on which the experiments were performed.

The core abstraction for a dataset in Spark is the `RDD`, that stands for Resilent Distributed Dataset. This object represents a dataset partitioned across all the nodes in the cluster. If multiple operations have to be performed on the same dataset, it is cached in memory, avoiding expensive writes and reads from the distributed filesystem. To provide fault tolerance in absence of a disk backup of the data the `RDD` stores some additional information: each partition stores the sequence of operations that were performed to compute it. If the machine that holds a partition fails, the framework is able to recompute only that partition on another machine, starting from the initial data. In some situations it may be preferable to have a copy of the dataset on disk anyway. In this case Spark provides a way to dump the dataset on the distributed filesystem. For further details on the implementation of `RDD`s and fault tolerance, we refer the reader to [ZCF$^+$10].

Our implementation of the algorithms described in this thesis is available as free software under the GPL-3 license on Github[5].

In our software, a graph is represented as a `RDD[(NodeId, Neighbourhood)]`, i.e. a distributed dataset of `NodeId` and `Neighbourhood` pairs. `NodeId` is a type alias for `Int`, Scala's 32 bit integer data type. `Neighbourhood` is a type alias for `Array[NodeId]`. The use of type aliases makes the software more readable and maintainable. In fact if we want to switch

4 `http://spark.incubator.apache.org/`
5 `https://github.com/Cecca/spark-graph`

to 64 bit node IDs, to handle larger graphs, we only need to change the `NodeId`'s definition.

Before describing the algorithm implementation, we describe a couple of helper traits. Traits in Scala can be thought as an hybrid between an interface (a class can implement multiple traits) and an abstract class (a trait can have implementation of its methods).

B.2.1 *The* `BallComputer` *trait*

First, we will look at the `BallComputer` trait, in Listing 4. This trait provides a `computeBalls` function that takes as input a graph and a radius and returns an `RDD` of `(NodeId, Ball)` pairs. `Ball` is a type synonym for `Array[NodeId]`. This function is the implementation of algorithm 12. The map function used is `sendBalls` that takes a `NodeId` and a pair `(Neighbourhood, Ball)` and outputs a set of pairs `(NodeId, Ball)`, where the node IDs are taken from the neighbourhood and the ball is the one in input to the function. The reduce function is `merge`, that takes two balls and performs a set union on them.

B.2.2 *The* `ArcRelabeler` *trait*

The second trait we look a is `ArcRelabeler`. This trait provides the `relabelArcs` function, that is used to shrink the graph. The idea is to replace each node ID with the color assigned to that node. In this way we automatically have the shrank graph. The first step is to transform the dataset from adjacency list format to edge list format: each record of the dataset is a single edge, encoded as a pair `(sourceId, destinationId)`. This transformation is performed at line 18. Then we cogroup, for two times, this edges dataset with the color dataset, in order to replace the source and destination IDs with their respective colors. This is accomplished by lines 24 through 33. Finally the graph is reverted back to its original adjacency list representation on line 36.

B.2.3 *The* `BallDecomposition` *implementation*

Now we are ready to describe the implementation of the entire ball decomposition algorithm. First of all, let's take a look at the algorithm skeleton:

```scala
def ballDecomposition( graph: RDD[(NodeId, Neighbourhood)],
                       radius: Int) = {

  val balls = computeBalls(graph, radius)
  val colors = colorGraph(balls)
  val relabeled = relabelArcs(graph, colors)
  val numNodes = relabeled.count()
  logger info ("Quotient cardinality: {}", numNodes)
```

Listing 4: The `BallComputer` trait. Classes inheriting from this trait can compute balls of radius r, given a graph represented as an RDD of `(NodeId, Neighbourhood)` pairs.

```
package it.unipd.dei.graph.decompositions

import it.unipd.dei.graph._
import spark.RDD
import spark.SparkContext._

/**
 * Trait for classes that can compute balls
 */
trait BallComputer {

  // ------------------------------------------------------------
  // Map and reduce functions

  def sendBalls(data: (NodeId, (Neighbourhood, Ball)))
  : TraversableOnce[(NodeId, Ball)] = data match {
    case (nodeId, (neigh, ball)) =>
      neigh.map((_,ball)) :+ (nodeId, ball)
  }

  def merge(ballA: Ball, ballB: Ball) =
    (ballA.distinct ++ ballB.distinct).distinct

  // ------------------------------------------------------------
  // Function on RDDs

  def computeBalls(graph: RDD[(NodeId,Neighbourhood)], radius: Int)
  : RDD[(NodeId, Ball)] = {

    var balls = graph.map(data => data) // simply copy the graph

      if ( radius == 1 ) {
        balls = balls.map { case (nodeId, neigh) =>
          (nodeId, neigh :+ nodeId)
        }
      } else {
        for(i <- 1 until radius) {
          val augmentedGraph = graph.join(balls)
            balls = augmentedGraph.flatMap(sendBalls)
                                  .reduceByKey(merge)
        }
      }

    return balls
  }


}
```

Listing 5: The `ArcRelabeler` trait. Classes inheriting from this trait can relabel the nodes of a graph given the `RDD` of the colors associated to each node.

```
package it.unipd.dei.graph.decompositions

import spark.RDD
import spark.SparkContext._
import it.unipd.dei.graph._

/**
 * This trait adds the capability to relabel arcs.
 *
 * An arc is relabeled by changing the IDs of its
 * endpoints with their respective colors.
 */
trait ArcRelabeler {
  def relabelArcs( graph: RDD[(NodeId,Neighbourhood)],
                   colors: RDD[(NodeId, Color)])
    : RDD[(NodeId,Neighbourhood)] = {

      var edges: RDD[(NodeId,NodeId)] =
        graph.flatMap { case (src, neighs) =>
          neighs map { (src,_) }
        }

      // replace sources with their color
      edges = edges.join(colors)
        .map{ case (src, (dst, srcColor)) =>
          (dst, srcColor)
        }

      // replace destinations with their colors
      edges = edges.join(colors)
        .map{ case (dst, (srcColor, dstColor)) =>
          (srcColor, dstColor)
        }

      // now revert to an adjacency list representation
      edges.groupByKey().map { case (node, neighs) =>
        (node, neighs.distinct.toArray)
      }
    }
}
```

```
  relabeled
}
```

First of all the balls are computed. Then the dataset of the balls is used to compute colors. Finally the graph is relabeled using the computed colors. The functions `computeBalls` and `relabelArcs` come from the traits we described in previous sections.

The `colorGraph` function looks like the following

```
150 def colorGraph( balls: RDD[(NodeId, Ball)] )
151 : RDD[(NodeId, Color)] = {
152
153   var taggedGraph: TaggedGraph =
154     balls.map { case (node,ball) =>
155       (node, (Left(Uncolored), ball))
156     }
157
158   var uncolored = countUncolored(taggedGraph)
159     var iter = 0
160
161     while (uncolored > 0) {
162       logger debug ("Uncolored " + uncolored)
163
164         val rawVotes = taggedGraph.flatMap(vote)
165         val votes = rawVotes.groupByKey()
166
167         taggedGraph = taggedGraph.leftOuterJoin(votes)
168                                  .map(markCandidate)
169
170         val newColors = taggedGraph.flatMap(colorDominated)
171                                    .reduceByKey(max)
172         taggedGraph = taggedGraph.leftOuterJoin(newColors)
173                                  .map(applyColors)
174
175         uncolored = countUncolored(taggedGraph)
176         iter += 1
177     }
178   logger info ("Number of iterations: {}", iter)
179
180   taggedGraph.map(extractColor)
181 }
```

The `TaggedGraph` data type is a graph representation that , along with node ID and ball information, contains information about the color of a node. Colors are represented using Scala data type `Either`. This data type represents entities that can take values from two sets of values. The type constructors are:

- `Right`: we use this constructor to indicate that the node is already colored and to hold the color
- `Left`: we use this constructor to indicate that the node is not yet colored. The information carried by a `Left` value is the status of the node: it can be simply `Uncolored` or it can be a `Candidate`, meaning that it will become the center of a ball.

This function is an implementation of algorithm 13. Iteratively, while there are uncolored nodes, the function first computes votes (lines 164 and 165). Then candidates are marked (line 167) and the new colors are computed (line 170) and applied to nodes (line 172).

colorGraph uses several helper functions. The first one is the vote function:

```scala
def vote(data: (NodeId, NodeTag))
: TraversableOnce[(NodeId, Vote)]= data match {
  case (node, (Right(_), ball)) => {
    val card = ball.size
      ball map { (_, (true, card)) }
  }
  case (node, (Left(Uncolored), ball)) => {
    val card = ball.size
      ball map { (_, (false, card)) }
  }
  case (_, (Left(Candidate), _)) =>
    throw new IllegalArgumentException(
      "Candidates can't express a vote")
}
```

This function takes a tagged node and, if it is Uncolored it sends a negative vote to all the nodes in the ball. Conversely, if the node is already colored, it sends a positive vote to all its neighbours.

One a node has received all the votes, the markCandidates function marks the nodes that can become node centers:

```scala
def markCandidate( data: ( NodeId,
                           ( NodeTag,
                             Option[Seq[(Boolean, Cardinality)]])))
: (NodeId, NodeTag) = data match {
  case (node, ((color@Right(_), ball), _)) => (node, (color, ball))
    case (node, ((Left(status), ball), Some(votes))) => {
      val card = ball.size
        val validVotes =
          votes. filter { case (v,c) =>
            c > card
          }.map { case (v,c) => v }
      val vote = (true +: validVotes) reduce { _ && _ }
      if (vote)
        (node, (Left(Candidate), ball))
      else
        (node, (Left(status), ball))
    }
  case (node, ((status@Left(_), ball), None)) =>
    (node, (status, ball))
}
```

This function marks as a Candidate a node that is not already colored and that has received all positive votes from nodes whose ball cardinality is higher than his.

Once the candidates have been marked, the colorDominated function propagates their color to all the nodes within their balls.

```scala
def colorDominated(data: (NodeId, NodeTag))
: TraversableOnce[(NodeId,(Color, Cardinality))] = data match {
  case (node, (Left(Candidate), ball)) => {
    val card = ball.size
      ball map { (_,(node, card)) }
  }
  case _ => Seq()
}
```

Then a reduce operation is performed for each key, assigning to each node the color with the maximum associated cardinality and by breaking ties by ID.

Finally the newly created colors are assigned to their nodes, using the function `applyColors`

```
def applyColors(data: (NodeId, (NodeTag, Option[(Color,Cardinality)])))
: (NodeId, NodeTag) = data match {
  case (node, ((status, ball), maybeNewColor)) =>
    status match {
      case Right(color) => (node, (Right(color), ball))
        case _ =>
        maybeNewColor map { case (color,_) =>
          (node, (Right(color), ball))
        } getOrElse {
          (node, (status, ball))
        }
    }
}
```

This function assigns the color to a node if this node is not already colored and if the node actually received at least a color during the current iteration.

B.2.4 *Compiling the application*

The software makes use of the `sbt` build tool to compile code and manage dependencies on third party libraries. In order to build the software, only the following steps are required

```
git clone https://github.com/Cecca/spark-graph.git
cd spark-graph
sbt/sbt assembly
```

The last step fetches all the dependencies, compiles the software, runs all tests and packages all the classes in a standalone, runnable `jar` file.

B.2.5 *Running the application*

Once the application has been packaged with `sbt/sbt assembly`, the program help can be obtained using the following command

```
bin/spark-graph --help
```

The tool is organized in subcommands, each corresponding to an algorithm

- `ball-dec` runs the ball decomposition
- `rnd-ball-dec-simple` runs the randomized version of the ball decomposition, as described in section 6.3.2.
- `hyper-anf` runs the parallel version of the HyperANF, as described in section 3.3.4.

All these subcommands accept a set of common options:

`--master` to set the Spark master node. For instance, to run the application locally using 64 cores, use `--master local[64]`. Further details on Spark clusters can be found in the Spark documentation[6]

`--input` the input graph dataset

`--output` the output graph dataset

Finally, the ball decomposition algorithm accepts the `--radius` option to specify the radius of the balls in the graph. In addition, the randomized ball decomposition algorithm accepts a `--probabiliy` option to set the probability that a node becomes a ball center.

6 `http://spark.incubator.apache.org/docs/latest/`

BIBLIOGRAPHY

[BBR$^+$12] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. Four degrees of separation. In *Proceedings of the 3rd Annual ACM Web Science Conference*, page 33–42, 2012.

[Bol81] Belá Bollobás. The diameter of random graphs. *Transactions of the American Mathematical Society*, 267(1):41–52, 1981.

[BR04] Béla Bollobás and Oliver Riordan. The diameter of a scale-free random graph. *Combinatorica*, 24(1):5–34, 2004.

[BRV11] Paolo Boldi, Marco Rosa, and Sebastiano Vigna. Hyperanf: Approximating the neighbourhood function of very large graphs on a budget, 2011.

[Bur73] Ju D. Burtin. Asymptotic estimates of the diameter and the independence and domination numbers of a random graph. In *Dokl. Akad. Nauk SSSR*, volume 209, page 765–768, 1973.

[BV03] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: Compression techniques. In *In Proc. of the Thirteenth International World Wide Web Conference*, pages 595–601. ACM Press, 2003.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. The MIT press, 2009.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[FFGea07] Philippe Flajolet, Eric Fusy, Olivier Gandouet, and et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA '07: proceedings of the 2007 International Conference on Analysis of Algorithms*, 2007.

[FMM85] Philippe Flajolet, G. N. Martin, and G. Nigel Martin. Probabilistic counting algorithms for data base applications, 1985.

[Gir09] Frédéric Giroire. Order statistics and estimating cardinalities of massive data sets. *Discrete Applied Mathematics*, 157(2):406–427, 2009.

[GSZ11] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the MapReduce framework. In *Algorithms and Computation*, page 374–383. Springer, 2011.

[HM06] Jane K. Hart and Kirk Martinez. Environmental sensor networks: A revolution in the earth system science? *Earth-Science Reviews*, 78(3-4):177–191, October 2006.

[HP10] Michel Habib and Christophe Paul. Survey: A survey of the algorithmic aspects of modular decomposition. *Comput. Sci. Rev.*, 4(1):41–59, February 2010.

[JaJ97] J. JaJa. *An Introduction to Parallel Algorithms*. Addison Wesley, 1997.

[JS08] Björn H. Junker and Falk Schreiber. *Analysis of biological networks*, volume 2. John Wiley & Sons, 2008.

[KSV10] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, page 938–948, 2010.

[KTA$^+$08] U Kang, Charalampos Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. Hadi: Fast diameter estimation and mining in massive graphs with hadoop, 2008.

[MAB$^+$10] Grzegorz Malewicz, Matthew H. Austern, Aart JC Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, page 135–146, 2010.

[MRK04] A. R. Mashaghi, A. Ramezanpour, and V. Karimipour. Investigation of a protein complex network. *The European Physical Journal B-Condensed Matter and Complex Systems*, 41(1):113–121, 2004.

[MS00] Ross M. McConnell and Jeremy P. Spinrad. Ordered vertex partitioning, 2000.

[Nel11] Jelani Nelson. *Sketching and streaming high-dimensional vectors*. PhD thesis, Citeseer, 2011.

[PGF02] Christopher R. Palmer, Phillip B. Gibbons, and Christos Faloutsos. Anf: A fast and scalable tool for data mining in massive graphs. In *International conference on knowledge discovery and data mining*, pages 81–90. ACM, 2002.

[PPP05] Stephen R. Proulx, Daniel E.L. Promislow, and Patrick C. Phillips. Network thinking in ecology and evolution. *Trends in Ecology & Evolution*, 20(6):345–353, June 2005.

[PPR$^+$12] Andrea Pietracaprina, Geppino Pucci, Matteo Riondato, Francesco Silvestri, and Eli Upfal. Space-round tradeoffs for mapreduce computations. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 235–244, New York, NY, USA, 2012. ACM.

[S$^+$13] W. A. Stein et al. *Sage Mathematics Software (Version 5.9)*. The Sage Development Team, 2013. `http://www.sagemath.org`.

[VKR$^+$05] Iuliu Vasilescu, Keith Kotay, Daniela Rus, Matthew Dunbabin, and Peter I. Corke. Data collection, storage, and retrieval with an underwater sensor network. In *Conference On Embedded Networked Sensor Systems*, page 154–165, 2005.

[Whi12] Tom White. *Hadoop: the definitive guide*. O'Reilly, 2012.

[WS98] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 1998.

[ZCF$^+$10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, page 10–10, 2010.