Fast Connected Components Computation in Large Graphs by Vertex Pruning

Alessandro Lulli, Emanuele Carlini, Patrizio Dazzi, Claudio Lucchese, and Laura Ricci

Abstract—Finding connected components is a fundamental task in applications dealing with graph analytics, such as social network analysis, web graph mining and image processing. The exponentially growing size of today's graphs has required the definition of new computational models and algorithms for their efficient processing on highly distributed architectures. In this paper we present CRACKER, an *efficient* iterative MapReduce-like algorithm to detect connected components in large graphs. The strategy of CRACKER is to transform the input graph in a set of trees, one for each connected component in the graph. Nodes are iteratively removed from the graph and added to the trees, reducing the amount of computation at each iteration. We prove the correctness of the algorithm, evaluate its computational cost and provide an extensive experimental evaluation considering a wide variety of synthetic and real-world graphs. The experimental results show that CRACKER consistently outperforms state-of-the-art approaches both in terms of total computation time and volume of messages exchanged.

http://dx.doi.org/10.1109/TPDS.2016.2591038

Index Terms—Distributed computing; Distributed algorithms; Graph algorithms;

1 Introduction

The graph formalism is very effective when it comes to model the relationships existing among different entities. Now more than ever, graphs are a useful tool to abstract the representation of a world becoming more and more interconnected: humans, smart devices, phones, computers and the information they share need to be always active and reachable leading to the establishment of an authentic *web of connections*. Today's graphs can be huge. Consider, for instance, the Web graph (Common Crawl provides 3.5 billion pages with 128 billion hyperlinks[?]), the Linked Open Data datasets (the LOD2 project indexes for 5.7 billion triples/edges [?]) or the Facebook and Twitter social networks (respectively 1.35 billion and 284 million monthly active users).

The size of these graphs makes it infeasible to rely on classic solutions for their processing, i.e., solutions that assume to perform the computation in a sequential way or on a shared-memory parallel machine. In fact, the amount of storage and computation required to process graphs of such size is far beyond the capabilities provided by a single, even high-performance, machine. In addition, graph data is often distributed by nature, as it is collected and stored on distributed platforms. Alternatively, distributed computing platform can be programmed by exploiting efficient but low-level techniques (e.g., send/receive message passing, unstructured shared memory mechanisms). Such approaches, traditionally adopted to process large datasets on clusters, lead to several issues. Low-level solutions are complex, error-prone, hard to maintain and their tailored nature hinder their portability.

- *A. Lulli and L. Ricci are with the Department of Computer Science, University of Pisa, Italy, E-mail: {lulli, ricci}@di.unipi.it*
- *E. Carlini, P. Dazzi and C. Lucchese are with the Information Science and Technologies Institute "A. Faedo", National Research Council of Italy, E-mail: {emanuele.carlini, patrizio.dazzi, claudio.lucchese}@isti.cnr.it*

To address these issues, across the years have been proposed several models, programming paradigms, approaches and technologies [?], [?], [?], [?], [?] that hide most of the complexity of distributed programming by providing pre-determined patterns or skeletons that developers combine to implement their applications. What differentiates these solutions from other high-level parallel programming models is that orchestration and synchronization of the parallel activities is not just simpler to approach but implicitly defined by the skeletons (or by patterns). In fact, developers do not have to specify the synchronizations between the application's sequential parts. When the communication/-data access patterns are known in advance, cost models can be applied to schedule skeletons programs to achieve an increased efficiency in the exploitation of computational resources.

Among the existing structured parallel programming paradigms, the MapReduce, which, in its current form has been proposed by Dean and Gemawat [?], has been one of the most widely adopted skeletons. According to this model, programmers are requested to provide problem-specific code segments for two functions, Map and Reduce. The Map function is applied to the input and emits a list of intermediate key-value pairs, while the Reduce function aggregates the values according to the keys. This programming paradigm is currently adopted for approaching a large set of data-intensive problems (e.g., large scale indexing, large scale graph computations, processing of satellite imagery, large-scale machine learning, etc.) that are usually processed exploiting clusters of (inexpensive) commodity hardware.

In this paper we focus on the problem of finding *connected components* (CC) in large graphs using a MapReduce approach. This problem is of fundamental importance in graph theory and can be applied to a wide range of computer science fields. For example, finding *CC* is the building block in many research topics, such as to generate group of features in image clustering [?], study the analysis of structure and evolution of on-line social networks [?], derive community structure in social networks [?], group together similar spam messages to detect spam campaigns [?], estimate the population from mobile calls [?]. *CC*s are used in the process of validation of graph algorithms [?], [?].

We propose CRACKER, a highly-efficient distributed iterative algorithm for the identification of connected components in large graphs. CRACKER works by iteratively growing a tree for each connected component belonging to the graph. The nodes added to the trees are no longer involved in the computation in the subsequent iterations. By means of trimming the number of nodes involved during each iteration, CRACKER significantly reduces the total computation time as well as the volume of information transferred via network.

CRACKER has been implemented on the Apache Spark framework [?], which is currently gaining momentum as it enables fast memory-based computations. It supports the development of applications structured according to the MapReduce paradigm but also allows programmers to interact with its execution support at a lower level [?].

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

1.1 Contribution

We already proposed a base version of CRACKER in [?]. We have extended our contributions as the following:

- we provide a complete theoretical analysis of CRACKER for undirected graph in terms of (i) correctness, (ii) computational cost and (iii) number of messages.
- we extend the base algorithm with three optimisations. We provide experimental evidence that these optimisations greatly improve its performance;
- we give a detailed description on the implementation of the CRACKER algorithm on the Apache Spark framework;
- we expand the experimental results, by (i) testing the optimizations introduced, (ii) testing the CRACKER algorithm on larger graph, and (iii) adding two further competitors.

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

2 Related Work

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

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

To structure our comparison, we frame a selection of existing solutions belonging to the conceptual framework of vertex-centric approaches. According to such model, each vertex of the graph is seen as a computational unit able to communicate with its graph neighbours. The computation is defined for a generic vertex, and it is repeated by all the vertices in the graph. In *CC* discovery algorithms, a vertex usually propagates and maintains information about the connected component it belongs to and the computation is iterated until convergence. Yan et al. [?] introduce the notion of balanced practical Pregel algorithm to characterize some nice-to-have properties for *CC* algorithms making use of this model of computation. For instance, Feng et al. [?] presents a *CC* algorithm targeted for Pregel framework with performance similar to HASH-TO-MIN.

In Table 1 we provide a characterization of some of the most relevant state-of-the-art approaches, presenting them on the basis of their qualitative behaviour, i.e. *detection strategy*, *communication pattern* and *vertex pruning*, and of the theoretical bounds for the number of iterations and of messages per iteration. Regarding the detection strategy, we distinguish between *labelling* and *clustering* approaches. The former associates, to each vertex, the id of the *CC* it belongs to, which is usually given by the smallest id of the vertices belonging to the *CC*. The latter assumes that one vertex for each *CC* knows the identifiers of all the other vertices of the same *CC*. As a consequence, *labelling* requires to process less amount of information; however the *CC*s can be efficiently reconstructed by a post-processing step.

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

The last feature we consider is *vertex pruning*, namely the ability of excluding vertices from computation. State-of-the-art algorithms keep iterating the same vertex-centric computation on all vertices of the graph until convergence. In this way, a large number of vertices remains involved in the computation even if they do not provide useful information toward convergence. For instance, a small *CC*

1. https://github.com/hpclab/cracker
2. http://www.di.unipi.it/~lulli/project/cracker.htm

TABLE 1: Characterization of state-of-the-art algorithms. d is diameter, n is the number of nodes, m is the number of edges

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

could be excluded from the computation when it reaches convergence, without affecting the discovery of other connected components.

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

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

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

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

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

Another recent solution for *CC* discovery, which we refer to as SGC, has been presented by Qin et al. [?]. It exploits a set of join operators defined by the authors for the Hadoop framework to model an iterative MapReduce computation. Similarly to CRACKER, their algorithm outputs a forest of trees, each representing a connected component. Initially each node becomes part of a tree-like graph by setting as a parent the node in its neighbourhood with the lower identifier, thus creating a forest of (possibly interconnected) trees. Then, one-node (i.e. singleton) and non-isolated trees (i.e. connected by an edge to another tree) are iteratively merged until all trees become isolated (the *hooking* phase). Subsequently, each tree is transformed into a star-shaped graph with the root in the center, so that nodes get to know the *CC* they belong (the *pointer jumping* phase). This last phase has the same goal of CRACKER's *seed propagation* (see Section 3.2) but it requires to access to the 2-hop neighbourhood of nodes. SGC performs only a limited amount of vertex pruning by deactivating at each step the nodes that do not match some criteria. Further, it requires a large number of MapReduce step in each iteration to verify some properties on each node and to identify if a tree can be merged.

3 The CRACKER Algorithm

Let $G = (V, E)$ be an undirected graph where V is a set of n vertices uniquely identified by values in $\mathbb{Z}$, and $E \subseteq V \times V$ is

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

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

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

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

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

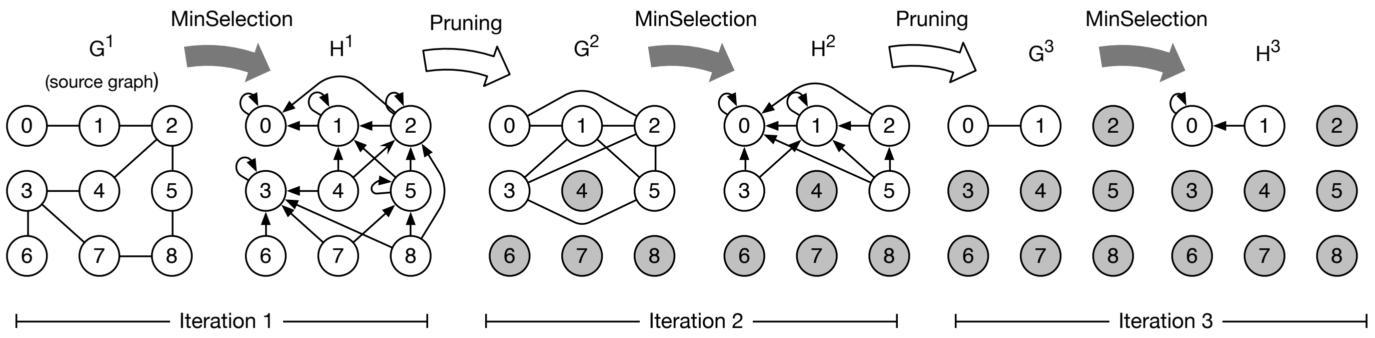


Fig. 1: CRACKER: example of seed identification. Gray vertices are excluded from the computation

the corresponding set of m edges. A connected component (*CC*) in G is a maximal subgraph $S = (V^S, E^S)$ such that for any two vertices $u, v \in V^S$ there is an undirected path in S connecting them. We conform to the convention to identify each *CC* of the graph with the smallest vertex identifier belonging to that component. The vertex having this identifier is the *seed* of the connected component.

The CRACKER algorithm (see Algorithm 1) achieves the identification of the *CC*s into two phases:

- *Seeds Identification*: for each *CC*, CRACKER identifies the *seed* vertices of the graph, and it iteratively builds a *seed propagation tree* rooted in the *seed*; whenever a vertex is added to the tree, it is excluded from computation in the subsequent iterations (see Alg. 1 lines 5–9). When all the vertices are excluded from the computation the *Seed Identification* terminates and the *Seed Propagation* begins.
- *Seeds Propagation*: propagates the *seed* to all the vertices belonging to the *CC* by exploiting the seed propagation tree built in the previous phase. (see Alg. 1 line 10).

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

3.1 Seed Identification

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

As shown in Algorithm 1, each vertex $u \in G$ is initially marked as *active*, meaning that at the beginning all vertices participate to the computation. The seed identification is, in turn, an iterative algorithm made of two steps: *MinSelection* and *Pruning*. The *Seed Identification* phase is exemplified in Figures 1 (Graph) and 2 (Tree) and detailed below.

3.1.1 MinSelection

This step serves to identify those vertices that are guaranteed to *not* be seed of any connected component (see Algorithm 2). From the point of view of the entire graph, it takes in input a undirected graph G^t at iteration t, and builds a new *directed* graph H^t. The edges of H^t are created as the following. For each vertex $u \in G^t$, the v_{min} is selected as the vertex with the minimum *id* from the set $NN_{G^t}(u) \cup \{u\}$ (see line 2 in Algorithm 2), where $NN_{G^t}(u)$ is the set of neighbors of u in G^t. The v_{min} is then notified to all the neighbours of u and to u itself. This communication is materialised as the addition of new directed edges $v \rightarrow v_{min}$ for every $v \in \{NN_{G^t}(u) \cup u\}$ (see line 4 in Algorithm 2). After all vertices in G^t completed the MinSelection, for each vertex $u \in H^t$ it holds the following: (i) if u is not a v_{min} for any $NN_{G^t}(u)$, it has no incoming links; (ii) u has an outgoing link to its v_{min} and with the v_{min} of every node in $NN_{G^t}(u)$. According to the algorithm, a vertex is considered a *potential seed* if it is a local minimum in the neighbourhood of some vertex, and in such case, it has at least one incoming edge in H^t. Therefore, after the MinSelection, the nodes that

Algorithm 1: The CRACKER algorithm

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

Algorithm 2: Min_Selection (u)

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

have no incoming edges are guaranteed to not be seed of any connected component.

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

3.1.2 Pruning

The *Pruning* step (see Algorithm 3) removes from H^t, and thereby excludes, all the vertices that cannot become *seeds*. The vertices excluded during the *Pruning* grow a forest of seed propagation trees T each covering a distinct CC of the graph. From the point of view of the entire graph, it takes in input a directed graph H^t and generates a new *directed* graph G^t.

In the Pruning, each node recomputes v_{min} considering $NN_{H^t}(u)$, which is composed by all the outgoing edges. Then, for every node v in $NN_{H^t}(u)$ (except v_{min}), a new undirected edge v with v_{min} is added to the graph G^{t+1} (see line 5). Note that $NN_{H^t}(u)$ is in general different from $NN_{G^t}(u)$, with the former normally having lower identifiers. For example, in Figure 1 $NN_{G^t}(5) = \{2, 8\}$ and $NN_{H^t}(5) = \{1, 2\}$. These undirected edges make sure that the nodes in G^{t+1} are not disconnected in case u is deactivated and therefore not included in the graph G^{t+1}. At the end of the Pruning, the nodes identified as non seed in the MinSelection have no edges and therefore are excluded by the computation. According to the algorithm of the MinSelection, this can be verified by checking whether a node has a self-link in H^t: if it does not it cannot be the minimum of the local neighbourhood (which includes itself) and can be safely excluded (see line 9). The nodes marked for exclusion are inserted in the seed propagation tree T (see line 10). Finally, a node is *finalized* as a seed when it is the only active node in its neighbourhood $NN_{G^{t+1}}(u)$. It is marked for exclusion and added to T as the root of a CC.

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

Algorithm 3: `Pruning`(u, T)

```
Input  : a node $u \in G$ and the seed propagation tree $T$
1  $NN_{H^t}(u) = \{v : (u \to v) \in H^t\}$
2  $v_{min} = \min(NN_{H^t}(u))$
3  if $|NN_{H^t}(u)| > 1$ then
4      forall $v \in NN_{H^t}(u) \setminus v_{min}$ do
5          AddEdge $((v \leftrightarrow v_{min}), G^{t+1})$
6      end
7  end
8  if $u \notin NN_{H^t}(u)$ then
9      $u$.Active = $False$
10     AddEdge $((v_{min} \to u), T))$
11 end
12 if IsSeed $(u)$ then
13     $u$.Active = $False$
14 end
```

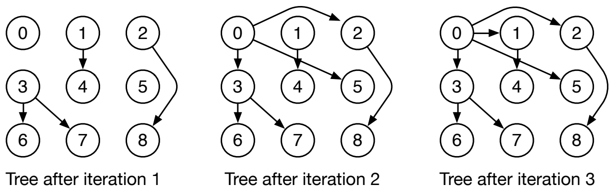


Fig. 2: CRACKER seed propagation tree

3.2 Seed propagation

Please recall that a seed propagation tree for each component of the graph is incrementally built during the Pruning. When a vertex v is excluded from the computation, a directed edge $v_{min} \to v$ is added to the tree structure T (see Line 10 in Algorithm 3).

The Seed Propagation phase starts when there are no more active nodes after the execution of the Pruning. At this there exists in T for each CC a seed propagation tree rooted in its *seed* node. Figure 2 shows the tree at each iteration resulting from the example presented in Figure 1. Such tree is then used to propagate the seed identifier to all the nodes in the tree. In details, the propagation starts from the root of each tree. The roots send their identifier to their children in one MapReduce iteration, this identifier will be the identifier of the CC. In every iteration, each node that receives the identifier propagates it to its children. The execution stops when the identifiers reach the leaves of the tree.

3.3 Cracker correctness

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

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

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

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

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

We can now prove the following.

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

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

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

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

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

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

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

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

3.4 Cracker computational cost

In this section we discuss the computational complexity of CRACKER both in terms of number of iterations and number of messages. We use Figure 3 to exemplify the notations and properties exploited, with reference to the same graph used in Figure 1.

Any given connected graph G can be organized into levels $L_0, \ldots, L_i, \ldots, L_{l-1}$, such that level L_0 contains nodes u having $u^1_{min} = u$, while level L_i contains nodes v such as their local minimum v^1_{min} is in level L_{i-1}. If we consider only edges of the kind $v \leftrightarrow v^1_{min}$, each node in L_0 is the root of a tree, where root-to-leaf paths traverse nodes with increasing ids. Figure 3 shows the two *increasing trees* present in the exemplifying graph.

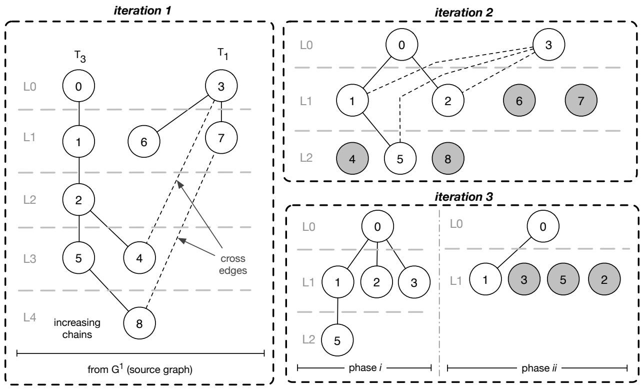


Fig. 3: Illustration of algorithm CRK on the same graph in Figure 1. Grey nodes are deactivated, and generated new cross-edges. Only the first three iterations are reported.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

□

Note, the actual number of messages is much smaller as by removing nodes at each iteration also edges are removed. To corroborate the above claim, in the experiments we evaluated the number of nodes and edges for a generic graph (Fig. 9a and 9b). Results show that the number of nodes and edges decreases exponentially, dramatically reducing the number of messages exchanged per iteration. Moreover, CRACKER always sends a number of message lower than HASH-TO-MIN, for all the tested graphs, as described in Table 3.

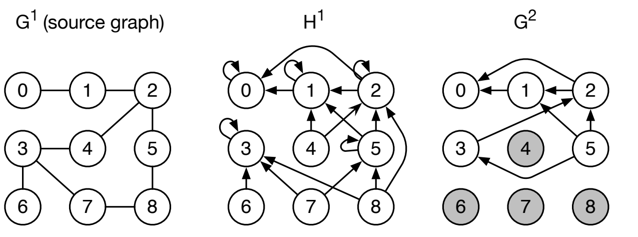


Fig. 4: Seed identification with the EP and OS optimizations.

3.5 Optimisations to CRACKER

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

3.5.1 Edge pruning

Edge Pruning (EP) operates during the MinSelection by reducing the number of redundant edges created, and therefore speeding up the computation. The idea is that when a node is already the minimum of its neighbourhood, it does not need to notify this information to its neighbours as this information is redundant. In EP, if a vertex $u \in G^t$ is a potential seed of its neighbourhood, then it does not add any edge in H^t, as instead would happen in the `ForAll` operation at line 3 in Algorithm 2.

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

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

The second case is shown in Figure 4 in H^1, in which EP avoids the creation of the directed edge $(4, 3)$. Instead, in the original algorithm, vertex 3 would have created the edge $(4, 3)$, which is useless since vertex 4 knows a better candidate, i.e. vertex 1.

3.5.2 Oblivious seed

The goal of the Oblivious Seed (OS) optimization is to reduce the number of edges created from potential seeds to other nodes of the CC. This optimization operates in the Pruning, specifically at the `AddEdge` (Line 5) in Algorithm 3. In the original version of CRACKER, a generic node u creates a set of undirected edges from $NN_{H^t}(u)$ to u_{min}. With OS, u would create only the directed edges from the $NN_{H^t}(u)$ to u_{min}, in fact creating a directed graph rather than a undirected one. The effect can be seen in Figure 4, in which the G^2 graph is a directed graph created by enabling the OS optimization. This optimization yields two benefits:

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

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

3.5.3 Finish computation sequentially

The third optimisation, Finish Computation Serially (FCS), has been inspired from the work of Salihouglu *et al.* [?] targeting Pregel-like systems. The assumption is that exist algorithms leading to a fast convergence of most of the nodes composing the graph (and the subsequent "deactivation" from computation) but with a small fraction of the graph that requires several additional steps of computation to converge. The idea surrounding their optimisation is to gather into a single machine all the nodes that still require some processing to converge. By means of this mechanism it is possible to avoid the execution of super-steps involving a large set of the computational resources when the actual processing involves only a very small fraction of the input graph. FCS monitors the size of the active subgraph, i.e., the fraction of the graph that still did not converge. When the size of the subgraph goes below a given threshold *K*, the subgraph is sent to a machine that performs the remaining of the processing serially. By construction, in CRACKER the set of active vertices is monitored in each iteration to check for the termination (see Algorithm 1 Line 9).

3.6 Implementation

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

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

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

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

4 Experimental Evaluation

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

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

4.1 Dataset Description

The following datasets have been chosen to build a comprehensive scenario to generalise as much as possible the empirical evaluation of CRACKER. We made all of them publicly available to foster a fair comparison. A summary of datasets' characteristics is presented in Table 2.

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

3. http://download.geofabrik.de/

4.2 Evaluation of Optimizations

This section discusses the impact of the three optimizations presented in Section 3: Edge Pruning (EP), Oblivious Seed (OS), and Finish Computation Sequentially (FCS). In order to test each optimization, both in isolation and in combination, we compare four different versions of the algorithm: (i) the plain CRACKER version, (ii) the CRACKER +EP version, (iii) the CRACKER +OS version, and the CRACKER +EP +OS version. We call SALTY-CRACKER the version of our algorithm with all the optimizations described in Section 3.5. Key findings: 1) OS allows to reduce both the maximum vertex degree and the number of edges at the cost of extra steps. However, each of these extra steps takes considerable less time and they can be cut with the FCS optimization; 2) EP and OS combined give a greater reduction on the completion time with respect to the simple sum of the reductions obtained by the two optimizations in isolation. 3) SALTY-CRACKER is faster than CRACKER thanks to the optimizations.

4.2.1 Edge Pruning and Oblivious Seed

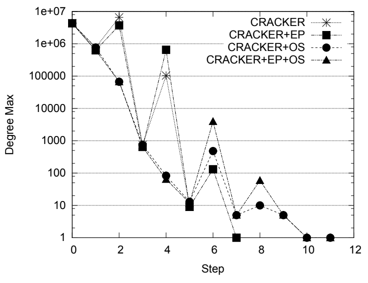
In these experiments we show the effectiveness of the *edge pruning* and *oblivious seed* optimizations. For this evaluation we used the PLD (see Table 2) dataset due to its large *CC* composed by the 99% of the entire graph, and for the large number of high degree vertices [**?**].

Figure 5a shows the maximum degree in the graph, and we used this metrics as an indicator of the balance of the computation, as higher values usually indicates unbalanced computations. In Figure 5b we report the cumulative completion time. Each of these metrics is sampled at each step of the MapReduce computation, specifically CRACKER executes two steps (MinSelection and Pruning) per algorithm iteration.

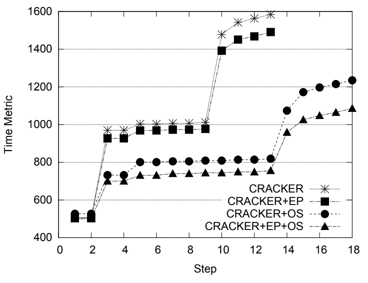
The main idea of the EP optimization acts when a node is already the candidate for itself in the MinSelection. In this scenario the node does not need to notify this information to its neighbours. i.e. it still be active in the next iteration and will be notified by neighbours if exist a better candidate. This optimization has few or no impact on balancing as we can see from Figure 5a. In some cases (for instance at the 4th

TABLE 2: Datasets description

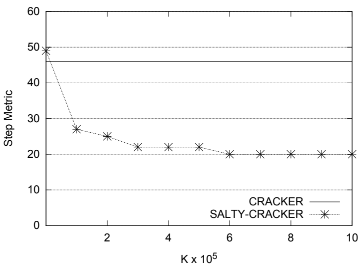
Name	|V|	|E|	β-index	ccNumber	ccMaxSize	diameter	AVG degree	MAX degree
Italy	19,006,129	19,939,100	0.95	153,876	14,694,405	10,534	2.09	16
Twitter [?]	24,159,954	532,138,866	0.05	14,038	24,129,131	N/A	44.05	1,848,376
LiveJournal [?]	5,204,176	77,402,652	0.07	4,533	5,189,809	17	29.75	15023
PLD [?]	39,497,204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



(a) Degree Max with EP and OS



(b) **TIME** metric with EP and OS



(c) **STEPS** metric with FCS optimization

Fig. 5: Evaluation of Optimizations

step) the highest degree is higher than the one measured with plain CRACKER. However, the number of edges not generated thanks to EP yields a beneficial, even if limited, impact on the completion time (-6%).

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

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

4.2.2 *Finish Computation Sequentially*

The main goal of the FCS optimization (Section 3.5.3) is the reduction of MapReduce iterations. From our theoretical demonstration the number of steps are primarily affected by the diameter of the graph. Therefore, to test the FCS optimization we synthetically generated a *path* graph with 5×10^6 vertices with randomly distributed identifiers.

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

4.3 Comparison with the State of the Art

We compared SALTY-CRACKER with the following competitors: (i) CCF [?], as we found it to be the best competitor in [?], (ii) ALT-OPT [?] (their fastest MapReduce implementation), (iii) SGC [?] as it is the most recent approach we know of, (iv) HASH-TO-MIN [?] because it is the de-facto standard for *CC* computation in MapReduce. We excluded from the comparison other approaches as they have already been proven slower in a previous work [?]. To conduct a fair com-

TABLE 3: Performances with real world datasets: **MESSAGE NUMBER** and **MESSAGE VOLUME** are values $\times 10^6$

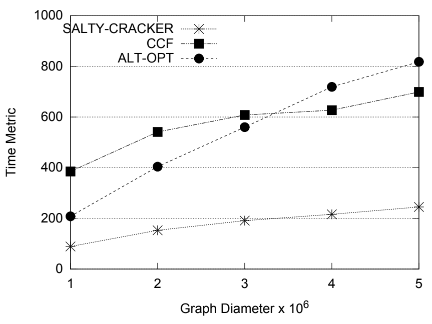
Twitter	**TIME**	**STEPS**	Msg	Vol
SALTY-CRACKER	**898**	9	1589	3520
CRACKER	1650 (1.84×)	12	1603	4001
CCF	3215 (3.58×)	7	819	5500
ALT-OPT	2230 (2.48×)	15	4158	8316
HASH-TO-MIN	9222 (10.27×)	7	2920	9807
SGC	15409 (17.16×)	72	1946	5743

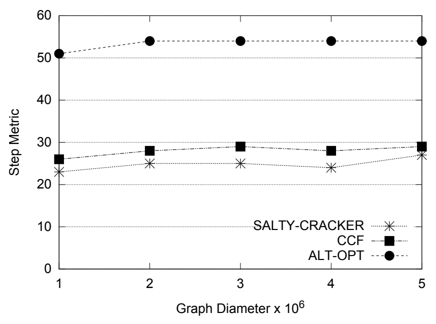
PLD	**TIME**	**STEPS**	Msg	Vol
SALTY-CRACKER	**1105**	10	2282	5218
CRACKER	1592 (1.44×)	13	2522	6302
CCF	20742 (18.77×)	7	1796	8690
ALT-OPT	8583 (16.82×)	15	4477	9378
HASH-TO-MIN	> 30×			
SGC	> 30×			

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

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

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


(a) **TIME** metric


(b) **STEPS** metric

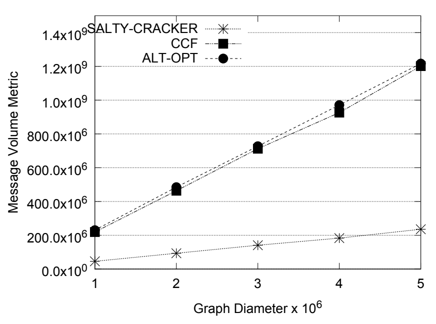
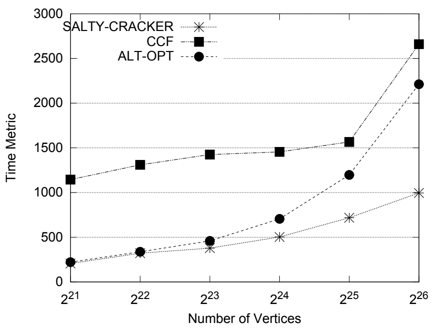
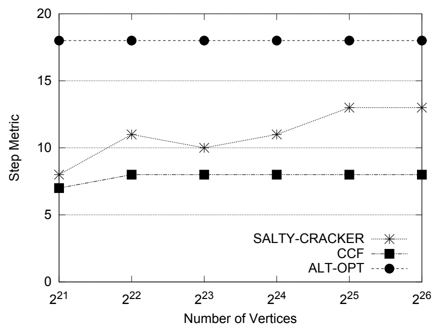

(c) **MESSAGE VOLUME** metric

Fig. 6: Sensitivity to Diameter


(a) **TIME** Metric


(b) **STEPS** Metric

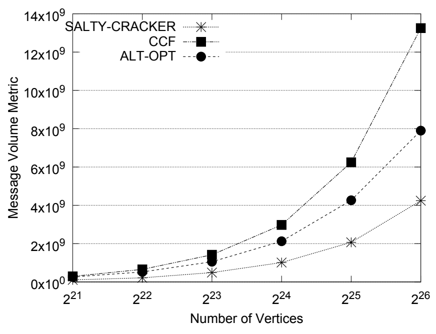
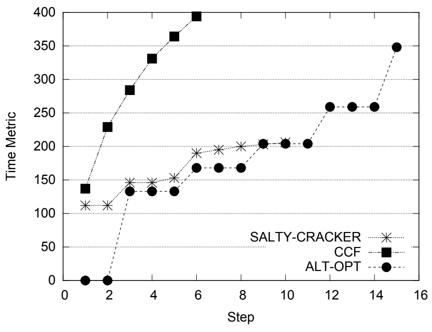
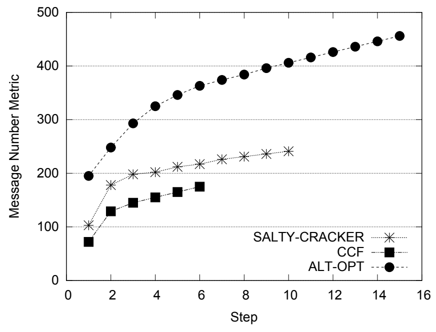

(c) **MESSAGE VOLUME** metric

Fig. 7: Sensitivity to Vertices Number


(a) LiveJournal: **TIME** metric


(b) LiveJournal: **MESSAGE NUMBER** metric

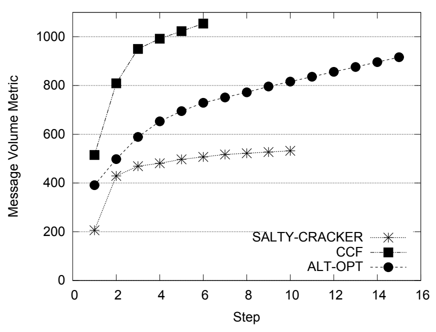
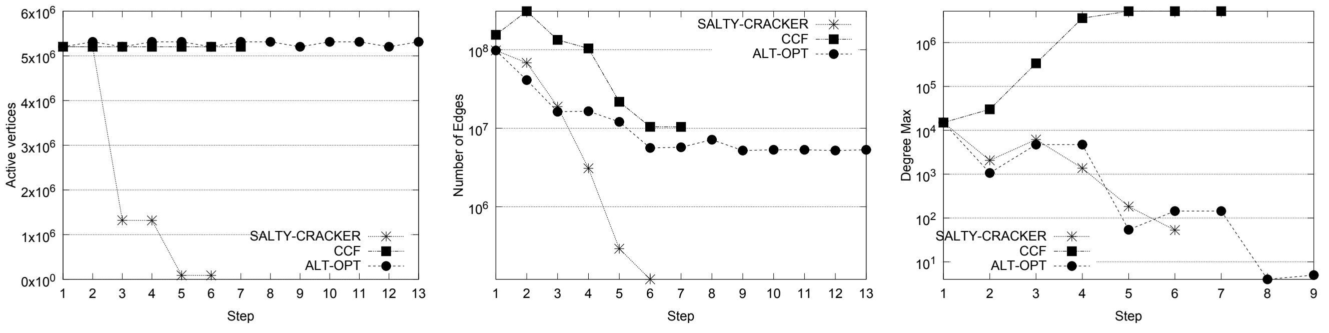

(c) LiveJournal: **MESSAGE VOLUME** metric

Fig. 8: Step By Step Comparison

parison, we implemented all the algorithms within the same Apache Spark [?] framework and with the same code-level optimisations. All source code used for the experimentation is publicly available. We show below that SALTY-CRACKER is the best performing algorithm, effectively reduces both the number of vertices and edges thanks to the proposed pruning strategy.

4.3.1 Performance on Real World Graphs

Table 3 presents a summary of the results obtained by the execution of SALTY-CRACKER and the competitors on all the real datasets. In terms of **TIME**, SALTY-CRACKER is the fastest approach with all the datasets. Apart from the plain version of CRACKER, best competitors are either ALT-OPT or CCF, except for PPI-All in which HASH-TO-MIN resulted to be the best competitor, suggesting that it works nice with dense graphs. In terms of **MESSAGE NUMBER** CCF is better than SALTY-CRACKER in all graph datasets except Italy, but when considering **MESSAGE VOLUME**, SALTY-CRACKER is the most efficient solution in all datasets. Finally, CCF is the best solution for **STEPS** with any dataset. Interestingly, we can observe how CCF adopts a very different strategy than SALTY-CRACKER. While the former sends large messages in a few number of iterations, the latter sends many small messages over a large amount of iterations. Conversely, ALT-OPT employs a lot of communication over a large number of iterations. However, in our experimental setup, it is clear that the approach of SALTY-CRACKER is the one guarantee-



(a) LiveJournal: Active Vertices (b) LiveJournal: Number of Edges (c) LiveJournal: Degree Max

Fig. 9: Graph Topology Evolution

ing the lowest execution times.

4.3.2 Sensitivity to Diameter

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

4.3.3 Sensitivity to Vertices Number

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

Figure 7a reports the results for the **TIME** metric. With a smaller number of vertices, until 2^{23}, SALTY-CRACKER performs similarly to ALT-OPT. However, as the graph size increases, the performance of SALTY-CRACKER gets better than ALT-OPT. The CCF algorithm is always at least 5 times slower than SALTY-CRACKER. In Figure 7b we show the results for the **STEPS** metric. CCF and ALT-OPT are very stable, while SALTY-CRACKER slightly increases in the number of steps. However, SALTY-CRACKER stays much below the theoretical bound of $O(\lceil \log_2 d \rceil)$ given in the Theorem 4. The results of the **MESSAGE VOLUME** are presented in Figure 7c. SALTY-CRACKER is always the algorithm requiring less communication cost with respect to the competitors thanks to its pruning mechanism. For instance, for the dataset having 2^{25} vertices, SALTY-CRACKER requires a **MESSAGE VOLUME** of approximately 2×10^9, instead ALT-OPT requires 4×10^9 and CCF 6×10^9.

4.3.4 Step by Step Evaluation

This evaluation was conducted by measuring the cumulative of **TIME**, **MESSAGE NUMBER**, and **MESSAGE VOLUME** metrics. Figure 8 reports the results of the most representative dataset, i.e. the LiveJournal. The other datasets exhibited similar results. The analysis of the results unravelled several interesting properties of the algorithms.

The cost per steps (in terms of all the metrics considered) is high in the starting steps for all the algorithms. However, thanks to the pruning mechanism and the optimizations, the performances of SALTY-CRACKER *increase* in the steps subsequent to the initial ones, while the competitors performances decrease or remain constant. For example, the time per step between the 3rd and 6th steps is almost zero resulting in a fast regain of computational time with respect to the closest competitor ALT-OPT.

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

Regarding SALTY-CRACKER, the transition from the *seed identification* to the *seed propagation* phase is clearly visible due to a peak in the computation time at the 6th step in Figure 8c. The reason of the peak is because the propagation tree is stored in a separate RDD, whose lineage is resolved by the Spark framework only at the start of the propagation phase, with the consequent increment in the computational time. To remove this peak, we experimented with further optimization (e.g., starting the seed propagation phase while the seed identification was still active) but all of them resulted in longer total running time.

4.3.5 Graph Topology Evolution

As we described in Section 3, one of the innovative features of SALTY-CRACKER is the pruning of vertices. A positive side effect of reducing vertices is the reduction in the number of edges. Figure 9 provides empirical results about the benefits of the pruning considering the LiveJournal computation.

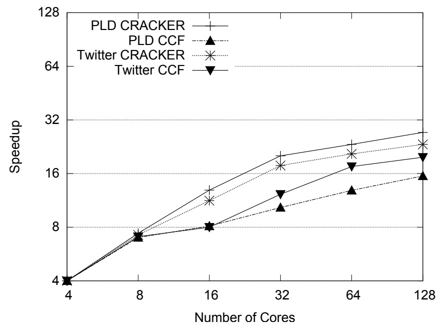


Fig. 10: Scalability evaluation

Figure 9a shows that the pruning mechanism of SALTY-CRACKER is very effective in reducing the number of active vertices. For example, after only one Pruning (at step 3), the number of active vertices is $1/3$ of the original input graph. Rather, in the competitors the number of active vertices is practically the same during all the computation. Figure 9b shows that the number of edges decreases in the initial steps of the computation for both SALTY-CRACKER and ALT-OPT. Also, thanks to the node pruning, SALTY-CRACKER overperforms ALT-OPT in the number of edges, although ALT-OPT has been designed to reduce edges.

Figure 9c shows that, for SALTY-CRACKER and ALT-OPT, the maximum degree decreases during the computation, and it is small in relation with the size of the graph. By comparison, the maximum degree of CCF increases during the computation.

4.4 Scalability

Finally, we tested the scalability of CRACKER, as well as comparing its performance with CCF, by varying the number of cores in the range [4-128]. Figure 10 depicts the results we achieved with the PLD and Twitter datasets. Similar patterns have been observed with the other datasets. In all the tests CRACKER always provides a better level of scalability than its competitors. We obtained an almost linear scalability using 8 cores, a still good level scalability with 16 cores, then the value tends to stabilise, providing only a small advantage going from 64 to 128 cores.

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

5 Conclusion

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

In this work we give a detailed description of these phases. We also proved the correctness of CRACKER by focusing on few invariants and properties that characterise the algorithm. We implemented two versions of CRACKER, a first basic version and an optimised one. In addition, we also implemented some of the most interesting state-of-the-art approaches that we compared against our solution. We evaluated the performance provided by CRACKER by means of a comprehensive set of experiments. All the algorithms have been implemented according to the MapReduce model using the Apache Spark framework.

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

As a future work we plan to conduct an in depth analysis focused on the factors (e.g., diameter, denseness, etc.) impacting both on the simplification mechanism and on the completion time. Moreover, as we mentioned in the experimental section, to conduct our evaluation we run our experiments on a cluster made of 5 machines. We plan to test the performance of CRACKER against the other approaches when using hundreds of machines.

Alessandro Lulli Alessandro Lulli received his Master Degree in Computer Science from the University of Pisa, Italy, in 2011. He is currently a Ph.D. student at the University of Pisa. His interests cover P2P Systems, Distributed Large Graph Processing and Graph Analytics.

Emanuele Carlini Emanuele Carlini received his MS.c in computer science from the University of Pisa in 2008, and Ph.D in Computer Science and Engineering from IMT Institute for Advanced Studies Lucca in 2012. He is currently a researcher with the Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo" of the National Research Council of Italy (ISTI-CNR). His research interests include Cloud computing, peer-to-peer systems, and large-scale graph processing. He is the co-author of more than 25 publications in peer reviewed international journals and conferences. He participated to and coordinated activities in national and European projects.

Patrizio Dazzi Dr. Patrizio Dazzi is a researcher with the Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo" of the National Research Council of Italy (ISTI-CNR). He received his MS.c in computer technologies from the University of Pisa in 2004, and Ph.D in Computer Science and Engineering from IMT Institute for Advanced Studies Lucca in 2008. His main research activities are in the areas of distributed computing, parallel programming models and cloud computing. He has published more than 50 papers on these topics in peer reviewed international journals, conferences and other venues. He participated to and coordinated activities in national and EC projects.

Claudio Lucchese Dr. Claudio Lucchese is a researcher with the Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo" of the National Research Council of Italy (ISTI-CNR). He received his MS.c. and Ph.D. from the Università Ca' Foscari di Venezia in 2003 and 2008, respectively. His main research activities are in the areas of data mining techniques for information retrieval, large-scale data processing and cloud computing. He has published more than 80 papers on these topics in peer reviewed international journals, conferences and other venues. He has taught courses on data mining and parallel computing at the Computer Science dept. of the Università of Firenze He participated to and coordinated activities in national and EC projects.

Laura Ricci Laura Ricci received the M. Computer Science from the University of Pisa in 1983 and the Ph.D. from the University of Pisa in 1990. Currently, she is an Assistant Professor at the Department of Computer Science, University of Pisa, Italy and a Research Associate with the Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo" of the National Research Council of Italy (ISTI-CNR). Her research interests include parallel and distributed systems, grid computing, peer-to-peer networks, and opportunistic networks. In this field, she has coauthored over 80 papers in refereed scientific journals and conference proceedings. She has served as a program committee member of several conferences and has been a reviewer for several journals.