SubgraphExplorer

Big Data Class — 2015 Contest

Adriano Di Luzio Danilo Francati

May 24, 2015

Contents

1 The Contest **3**

2 The input data **3**
2.1 Input pre-processing . . . 3

3 The Map-Reduce Algorithm **4**
3.1 Overview . . . 4
3.1.1 Neighbourhood builder . . . 4
3.1.2 Subgraph explorer . . . 4
3.2 Final overview . . . 5

4 Implementation details **6**
4.1 Heuristic factor . . . 6
4.2 Neighbourhood filtering . . . 7
4.3 K-Promising Nodes . . . 7
4.4 Best result selection . . . 8
4.5 Input pruning . . . 8

5 Experimental results **8**

6 Executing the algorithm **9**
6.1 Our environment . . . 9
6.2 Execution . . . 9
6.3 Reading the results . . . 10
6.4 Command line parameters . . . 10
6.5 Running times . . . 10

1 The Contest

The goal of this contest is to find the *smallest dense-enough subgraph* in a given graph.

Specifically, let $G = (V, E)$ be an undirected graph. For any set of vertices $V' \subseteq V$ we define the density of the subgraph induced by V' as follows:

$$\rho(V') = \frac{|E[V']|}{|V'|}$$

where $E[V'] \subseteq E$ is the set of edges whose endpoints lie in V'.

Our goal is to design and implement a *Map-Reduce* algorithm to find the smallest possible subgraph that still satisfies the given density requirements $\hat{\rho}$. The dimension of a graph $G = (V, E)$ is specified as $|V|$, the number of nodes it contains.

Mathematically, the contest's goal can be expressed as:

$$\min_{|V'|}\{V' : V' \subseteq V \wedge \rho(V') \geq \hat{\rho}\}$$

2 The input data

The contest test-bed is composed of three graphs from the SNAP website (*Stanford Network Analysis Project*):

Graph name	Number of nodes	Number of edges
`loc-gowalla`	196591	950327
`web-BerkStan`	685230	7600595
`as-skitter`	1696415	11095298

Table 1: The contest graphs.

We present experimental results produced by our algorithm on those graphs in Sec. 5.

2.1 Input pre-processing

Each graph is stored in a slightly different format and is either directed or undirected. As a consequence, we pre-process the input data, requiring each graph in the following form:

a b
c d
e f
...

Table 2: Our input format.

Where $\langle a \quad b \rangle$ means an undirected edge between nodes a and b. This edge must appear only once in the input file.

3 The Map-Reduce Algorithm

3.1 Overview

Our algorithm is composed of *two different Map-Reduce jobs*. The first job parses the input data and builds the list of neighbour of each node in the graph. The second job is instead iterative: At each iteration round it builds a cluster of nodes (*i.e.* a subgraph), evaluates the result and passes it to the next round. We stop when we are satisfied with the result we obtained.

3.1.1 Neighbourhood builder

As the name suggests this job parses the input format and outputs the adjacency lists representation of the graph.

Neighbourhood builder Map-Reduce algorithm

function MAP(Node u, Node v)
  emit(u, v)
  emit(v, u)
function REDUCE(Node u, Node[] $\Gamma(u)$)
  emit$(\{u : \Gamma(u)\})$

Please note that when we output a value of the $\{u : \Gamma(u)\}$ we're outputting a specific data structure that we'll explain in Sec. 3.1.2. This structure is *syntactic sugar* for the reader and under the hood it is implemented using pairs of keys and values that conforms to the Map-Reduce data paradigm.

We store the results on disk. Each result will be used several times by the following jobs.

3.1.2 Subgraph explorer

This is an iterative job, at each round it handles the output of the previous job and prepares the input for the following one.

It works on what we call a *Neighbourhood*: A dictionary (*i.e.* a map) composed as follows:

$$\begin{cases} v_1 : & \Gamma(v_1) \\ v_2 : & \Gamma(v_2) \\ v_3 : & \Gamma(v_3) \\ v_4 : & \Gamma(v_4) \\ & \dots \end{cases}$$

The keys of this dictionary represent the subgraph we have built until now. The values represent the neighbourhood of the subgraph. We use them to choose the next most promising node — we add this node to the subgraph and we continue iterating. We stop once we have found a subgraph whose density meets the input requirements.

Please note that the output of the *Neighbourhood builder* consists of a *Neighbourhood* containing only a single node: *i.e.* a *singleton* subgraph.

As usual, this job is composed of a *Map* and *Reduce* phase. The *Map* phase must choose, as a key, the most promising node and emit it. The emitted value is the very same map received as input.

Consequently, the *Reduce* phase receives as input a single node (the one that was chosen as most promising by the mappers) and a set of *Neighbourhoods*. The elements of this set are different subgraphs that picked the same key node as the most promising one.

We also deliver to the *Reducer* the neighbourhood of the key node. We merge this neighbourhood and the others to build a new, expanded, subgraph.

Subgraph explorer Map-Reduce algorithm

function MAP$_{explorer}$(Neighbourhood map)
  $v \leftarrow$ **mostPromisingNode(map)**
  emit(v, map)
function MAP$_{reader}$($\{u : \Gamma(u)\}$)
  emit$(u, \Gamma(u))$
function REDUCE(Node u, Neighbourhood[] neighbourhoods)
  $newMap \leftarrow merge(neighbourhoods)$
  if $\rho(newMap) \geq \hat{\rho}$ **then**
    output$(newMap)$
  else
    emit$(newMap)$

As you can notice, our architecture is unusual. We employ two different mappers:

Map$_{explorer}$ chooses a node among the neighbourhood and emits it. This node is the most promising one: *i.e.* is the one that is more linked with the current subgraph.

Map$_{reader}$ reads, at each iteration, the output produced by the *Neighbourhood Job*. We use it to deliver to the reducer the neighbourhood of the newly-chosen node. As a consequence, $newMap$ will also contain the entry $\{u : \Gamma(u)\}$.

3.2 Final overview

Fig. 1 shows a graphical overview of the chaining of our jobs.

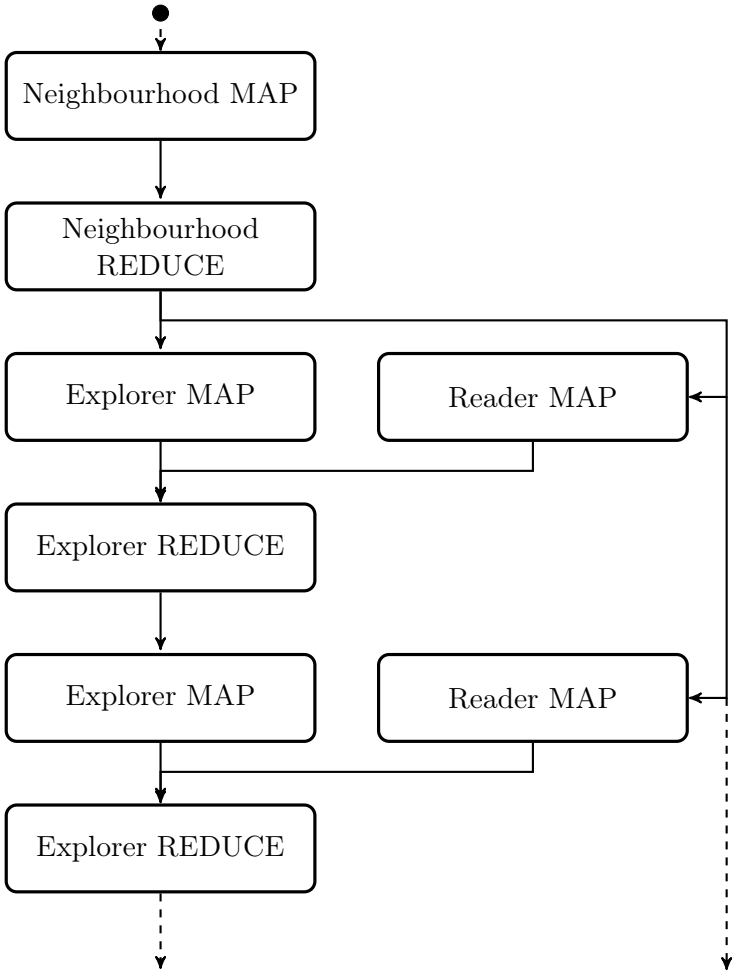

Figure 1: Graphical overview of our jobs.

4 Implementation details

We describe in this section all the implementation details that we have skipped in the previous section (Sec. 3).

4.1 Heuristic factor

Choosing the most promising node means finding the node which appears to be the most linked to the subgraph. Obviously this node has to be external to the subgraph and should satisfy a specific criterion.

As an example, if we require this node to be connected to every node already in the subgraph, we would only find *cliques* (*i.e.* complete graphs). In order to specify different criteria we use an *heuristic factor*.

This value specifies the percentage of subgraph nodes we require a candidate node to be connected with, in order to be considered by our choosing function. *E.g.* an *heuristic factor* of 0.75 means that we consider only nodes that are connected with at least the 75% of the subgraph and ignore all the others. If there are multiple possible candidates, we choose the one with more links to the subgraph.

Fig. 2 shows how we change the heuristic factor according to the round: After a few iterations, we loosen our criterion in order to find larger subgraphs. Furthermore, we use the same *heuristic factor* in the filtering step (Sec. 4.2).

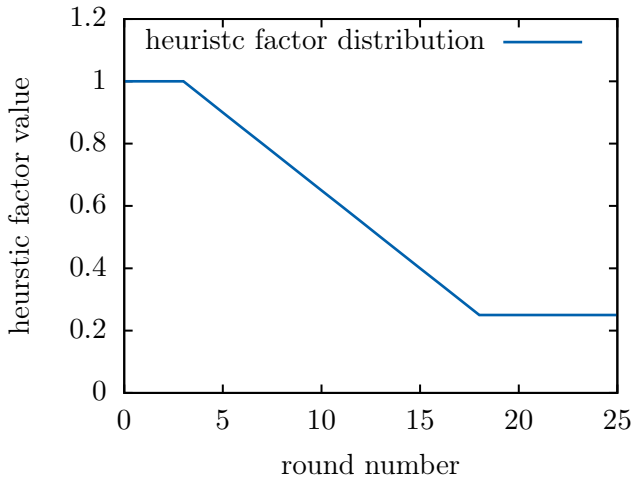

Figure 2: The distribution per round of the heuristic factor we use.

4.2 Neighbourhood filtering

After merging different neighbourhoods in the *Subgraph explorer* job, *newMap* could contain nodes that do not respect our *heuristic factor criterion*. As a consequence, before calculating the density of the subgraph we filter those nodes according to the same *heuristic factor*.

This step is particularly useful in order to maintain consistency between iterations and to achieve optimal results.

4.3 K-Promising Nodes

Until now we've used the notion of *most promising node*. In our implementation we have decided to emit the *K-Promising Nodes*. This means that we emit the same *Neighbourhood* with different key nodes, *branching* more widely our research space.

Specifically, we use a K value of 5, that has proven to be the best among performance and results.

4.4 Best result selection

We have noticed that for each value of $\hat{\rho}$ exists a lower bound for the size of any subgraph V satisfying the required density, as follows:

$$|V| \geq (\hat{\rho} \cdot 2) + 1$$

Sometimes we end up with a subgraph V' that meets the $\hat{\rho}$ requirement, is a complete graph (a *clique*) and has a size $|V'| > |V|$. We can thus remove $|V'| - |V|$ nodes from V (that remains a clique) and still have a subgraph that satisfies the density requirement.

4.5 Input pruning

We have noticed that a lot of *real-world* graphs follow the *power-law degree distribution*. To achieve a better efficiency we have thus decided to prune the input from all the nodes whose degree is lower than 1.

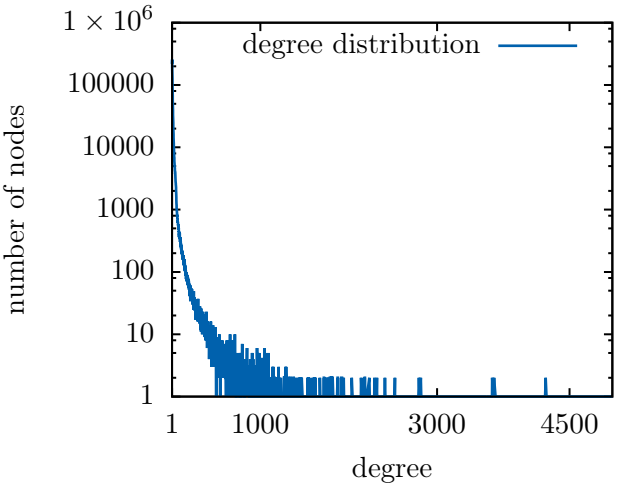

Figure 3: `as-skitter` degree distribution follows a power-law.

5 Experimental results

As we have shown in Sec. 4.4, given a specific $\hat{\rho}$ we know a lower bound on the size of a subgraph satisfying the requirement. Such subgraph is a clique by definition of density.

As a first experiment we have thus launched trying to find the biggest *clique* we could. Table 3 shows our result.

Please notice again that we can *select* from those cliques a number k of nodes and obtain a subgraph whose density is:

$$\rho - (k \cdot 0.5)$$

We have thus found results for each $\hat{\rho}$ such that:

$$1 \leq \hat{\rho} \leq \rho$$

Graph	Maximum clique	ρ	Running time
`loc-gowalla`	29	14	$27''$
`web-BerkStan`	201	100	$8'15''$
`as-skitter`	67	33	$5'$

Table 3: The maximum clique we have found for each graph.

After this, we began exploring greater $\hat{\rho}$ values:

Graph	Minimum subgraph	ρ	$\hat{\rho}$	Running time
`loc-gowalla`	32	15.31	15	$1'30''$
`loc-gowalla`	51	20.50	20	$2'$
`loc-gowalla`	73	25.15	25	$2'30''$
`web-BerkStan`	392	103.40	100.5	$8'40''$
`as-skitter`	70	34.34	34	$6'$

Table 4: The minimum subgraph we have found for each graph and $\hat{\rho}$ value.

6 Executing the algorithm

6.1 Our environment

We assume you have the last Apache Hadoop version available installed. We tested our implementation against:

- Hadoop version 2.7.0 configured in local or pseudo cluster mode.
- Java Virtual Machine version 1.8.0_20.

6.2 Execution

We assume that you have downloaded our project JAR file in the current directory as

`SubgraphExplorer-jar-with-dependencies.jar`

Before running Hadoop you have to configure its `HADOOP_CLASSPATH` in order to find our project JAR and let the system be aware of our custom data types.

```
export HADOOP_CLASSPATH=‘pwd‘/SubgraphExplorer-jar-with-dependencies.jar
```

You can then launch Hadoop as follows:

```
hadoop jar SubgraphExplorer-jar-with-dependencies.jar \
    -input graph-file.txt -rho RHO_VALUE
```

You can further configure the algorithm by providing the appropriate command line parameters (see Sec. 6.4).

6.3 Reading the results

We write our results in the `result` folder within the HDFS. Each file of this folder contains a subgraph, found by the last reduce step, that satisfies the $\hat{\rho}$ density. It is up to you choosing the minimum one. In our case, we use an ad-hoc Python script.

6.4 Command line parameters

We defined a few command line parameters that you can use to configure our algorithm.

-input expects the path to a graph file respecting our format.

-rho the $\hat{\rho}$ value you want.

-heuristic_factor specifies a custom heuristic factor that won't be changed after each round.

-knodes specifies the K most promising nodes picked at each step. Varying this parameter influences the running time or the space used by each node.

6.5 Running times

Every run of our algorithms takes, on our system, at most 10 minutes: our approach is bottom-up, so we expect running times to be directly proportional to the size of the graph and to the value of $\hat{\rho}$. Nonetheless, we have never used an Amazon EMR cluster to find our results.