Aggregate Graph Statistics

Giorgio Audrito
University of Torino, Italy
giorgio.audrito@unito.it

Ferruccio Damiani
University of Torino, Italy
ferruccio.damiani@unito.it

Mirko Viroli
University of Bologna, Italy
mirko.viroli@unibo.it

Collecting statistic from graph-based data is an increasingly studied topic in the data mining community. We argue that these statistics have great value as well in dynamic IoT contexts: they can support complex computational activities involving distributed coordination and provision of situation recognition. We show that the HyperANF algorithm for calculating the neighbourhood function of vertices of a graph naturally allows for a fully distributed and asynchronous implementation, thanks to a mapping to the field calculus, a distribution model proposed for collective adaptive systems. This mapping gives evidence that the field calculus framework is well-suited to accommodate massively parallel computations over graphs. Furthermore, it provides a new "self-stabilising" building block which can be used in aggregate computing in several contexts, there including improved leader election or network vulnerabilities detection.

1 Introduction

Statistical informations are having an increasingly important role in handling massive graph-based data, such as the Web Graph or social network graphs. In this context, statistical summaries are able to detect local- or global-level features which can give approximate solutions to problems which would not be solvable exactly for graphs of that size. A particularly versatile statistic is the *neighbourhood function N_G*, which calculates the number of vertices within a certain distance from a given vertex, and can be computed efficiently for massive graphs through the state-of-the-art HyperANF algorithm [4]. Through N_G several specific problems can be addressed [11]: among many others, ranking vertices by their *harmonic centrality* (which is easily derivable from N_G).

Vertex ranking is particularly relevant for distributed IoT scenarios, and has already been considered in this context for fixpoint-based measures such as *PageRank* [10]. These measures can help selecting "first-class" vertices (which could fruitfully be promoted to local communication hubs), or weaknesses in the network infrastructure (vertices with few close-range neighbours). However, algorithms suitable for fully-distributed IoT scenarios need to satisfy several non-trivial requirements, such as: resilience to network changes, continuous adaptation to input evolution (formalizable by *self-stabilisation* [12]), near-constant time complexity for each computational unit, locality of interactions (obtainable with the *aggregate programming* paradigm [2] for designing individual behaviour from a collective viewpoint). Porting algorithms to the IoT is thus a non-trivial task which may require extensive work where natural translations are not possible or sufficiently performing.

In this paper, we provide a natural translation of HyperANF in Field Calculus [5, 6], a tiny functional language for aggregate programs. On the one hand, this translation gives an example suggesting that field calculus is well-suited for expressing massively parallel computations over graphs. On the other hand, we obtain in this way a self-stabilising and local program which can be applied to solve problems such as improved leader election or network vulnerabilities detection. Section 2 introduces the concepts needed to express the translation, with an overview of the related works. Section 3 provides the translation, together with a motivation outlining its possible use cases.

2 Background and Related Work

2.1 Graph Statistics

Many different techniques for collecting statistics from graph-based data are being studied in the data mining community, each able to address different questions. In this paper, we focus on statistics derivable from (variations of) the *neighbourhood function* of a graph.

Definition 1 (Neighbourhood Function). *Let $G = \langle V, E \rangle$ be a graph with n vertices and m edges. The* generalized individual neighbourhood function $N_G(v, h, C)$, *given $v \in V$, $h \geq 0$ and $C \subseteq V$, counts the number of vertices $u \in C$ which lie within distance h from v. In formulas, $N_G(v, h, C) = |\{u \in C : \text{dist}(v, u) \leq h\}|$.*

Elaborations of the N_G values have been used to answer many different questions [4, 11], including: graph similarity, vertex ranking, robustness monitoring, network classification. Since exact computation of N_G is impractical, requiring $O(nm)$ time in linear memory and $O(n^{2.38})$ time in quadratic memory, effort has been spent in developing fast algorithms approximating N_G up to a desired precision.

In Palmer et al. [11], the problem of computing N_G is reduced to that of succinctly maintaining size estimates for sets upon set unions. Define $\mathcal{M}_G(v, h, C)$ as the set of vertices in C within distance h from v, so that $N_G(v, h, C) = |\mathcal{M}_G(v, h, C)|$. Notice that $\mathcal{M}_G$ can be computed recursively as:

$$\mathcal{M}_G(v, h, C) = \bigcup_{(vu) \in E} \mathcal{M}_G(u, h-1, C)$$

where $\mathcal{M}_G(v, 0, C) = \{v\} \cap C$. If we represent the sets $\mathcal{M}_G(v, h, C)$ through succinct counters, the previous formula translates into an algorithm computing estimations of N_G. Vigna et al. [4] later improved the original algorithm by using a more effective class of estimators, the *HyperLogLog counters* [8], by expressing the "counter unions" through a minimal number of broadword operations, and by engineering refined parallelisation strategies. HyperLogLog counters maintain size estimates with asymptotic relative standard deviation $\sigma/\mu \leq 1.06/\sqrt{k}$, where k is a parameter, in $(1 + o(1)) \cdot k \cdot \log\log(n/k)$ bits of space. Updates are carried out through k independent "max" operations on $\log\log(n/k)$-sized words. It follows that N_G, given a fixed precision, can be computed in $O(nh)$ time and $O(n\log\log n)$ memory, allowing it to be applied on very large graphs such as the Facebook graph [1].

2.2 Field Calculus

The Field Calculus is a tiny functional language for formally and practically expressing aggregate programs: a detailed account of it is given in [5, 6]—we hereby recollect its most basic characteristics in order to be able to use it as common language to express the algorithms used in this paper with actual executable code.

In field calculus a program P consists of a sequence of function declarations $\overline{\text{F}}$ and of a main expression e—following [9], the overbar notation denotes metavariables over sequences: e.g., $\overline{\text{e}}$ ranges over sequences of expressions $\text{e}_1, \ldots, \text{e}_n$ with $n \geq 0$. On each device δ the expression e evaluates to a value v that may depend on the state of δ (values sensed by δ, the result of previous evaluation, and information coming from neighbours). Therefore the expression e induces a *computational field* Φ, which can be represented as a time-varying map $\delta_1 \mapsto \text{v}_1, \ldots, \delta_n \mapsto \text{v}_n$, assigning a value v_i to each device δ_i in a network. Each device δ updates its value (by evaluating e) in asynchronous computational rounds.

The syntax of an expression e comprises several constructs, which we briefly present. Firstly, e can be a variable x, used as function formal parameter, or a value v which in turn could be either a *neighbouring field value* ϕ (associating to each device a map from neighbours to local values), or a *local*

value ℓ (built-in functions b, user-defined functions d, anonymous functions $(\overline{x})$=>e, or built-in values). Secondly, e can be a function call $e(\overline{e})$ where an expression of functional type e is applied by-value to arguments $e_1, \ldots e_n$; or an if-expression $if(e_0)\{e_1\}\{e_2\}$ modelling domain restriction, which computes e_1 in the devices where e_0 is true, and e_2 on the others.

Finally, e can be a rep-expression or an nbr-expression, which are the two most characteristic constructs of field calculus: a rep-expression $e = rep(e_1)\{(x)\text{=>}e_2\}$ models time evolution and state preservation, by repeatedly applying the anonymous function (x)=>e_2 to the value computed for e in the previous round, starting from e_1 if no previous value is available; an an nbr-expression $nbr\{e\}$ models neighbourhood observation, by producing a neighbouring field value that represents an "observation map" of neighbour's values for expression e, namely, associating to each device δ (that has evaluated $nbr\{e\}$ during its last update) a map from neighbours to their latest evaluation of e.

In order to conveniently present the code of the algorithms, we extend the core calculus with few additional constructs, which can be defined as syntactic sugar in terms of the presented ones: let-binding expressions $let\ x = e_1\ in\ e_2$ (operationally equivalent to $((x)\text{=>}e_2)e_1$), binary operators in infix notation, and the notation $[e_1, \ldots, e_n]$ for tuples.

3   Aggregate Graph Statistics

In the next subsection we shall show how the HyperANF algorithm presented in Section 2.1 can be naturally translated into field calculus. This transformation suggests that the field calculus is a natural framework for parallel computations, so that a future cloud-based implementation of it [13] could be used to address a relevant class of "traditional" massively parallel computations. However, this translation has further relevance in the aggregate computing context, since the resultant algorithm is able self-adjust to dynamic changes in inputs (i.e., has the *self-stabilisation* property and belongs to the *self-stabilising fragment* of the calculus [12]). In fact, HyperANF provides a new *building block* which can be used to produce various dynamic vertex rankings, able to classify vertices by features of their neighbourhoods. This rankings can in turn be used either to recognise network vulnerabilities (vertices with few short-range neighbours and many long-range neighbours), or to elect leaders for other coordination mechanisms (vertices with high degrees of "centrality"). For example, N_G can be used to compute the *harmonic centrality* of each vertex through the formula $H(v) = \sum_{h>0} N_G(v, h, V)/h$. Vertices with high harmonic centrality are best-suited to be elected as leaders for coordination mechanisms, since they are connected to many other vertices through a small number of hops.

3.1   HyperANF in Field Calculus

Assume that we are given a HyperLogLog type, which can be constructed out of numerical elements, composed through a union operator, and inspected for size through function estimate. We can express HyperANF through the following simple field calculus code, computing a list of floating-point estimations $\langle N_G(\delta, i, C) : i = 0 \ldots h \rangle$ in each device δ, provided that source is true if and only if $\delta \in C$.

```
def HyperANF(h, source) { // has type: (num, bool) → (HyperLogLog, list(float))
  if (h == 0) {
    let c = if (source) {HyperLogLog(uid)} {HyperLogLog()} in [c, list(estimate(c))]
  } {
    let r = HyperANF(h-1, source) in
    let c = unionhood( nbr{1st(r)} ) in [c, cons(2nd(r), estimate(c))]
} }
```

In Vigna et al. [4], non-trivial effort was made to allow for efficient parallel computation of Hyper-ANF, since the algorithm did not fit inside existing parallel computation frameworks (e.g., MapReduce [7]). Thus, it is worth noting that the field calculus is instead able to easily capture the algorithm, allowing for an "automatic" parallelisation and naturally extending the original algorithm to an "on-line" context (i.e., self-adapting to changes of either network structure or source vertices).

Function `HyperANF` can be used to implement real-time vertex ranking calculations for efficient leader election or network weaknesses detection. As a motivating example, we can integrate the leader election routine `S` [3] with HyperANF, by passing harmonic centrality instead of a random UID to the same inner function `BreakUsingUIDs`, thus favouring central devices to be elected as leaders.

```
def HarmonicCentrality(n) {
  if (length(n) <= 1) { 0 } { head(n)/(length(n)-1) + HarmonicCentrality(tail(n)) }        }
def LeaderElection(grain, metric, hmax, source) {
  BreakUsingUIDs(HarmonicCentrality( 2nd(HyperANF(hmax, source)) ), grain, metric)        }
```

References

[1] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander & Sebastiano Vigna (2012): *Four degrees of separation*. In: *Web Science 2012, WebSci '12, Evanston, IL, USA - June 22 - 24, 2012*, pp. 33–42, doi:10.1145/2380718.2380723.

[2] Jacob Beal, Danilo Pianini & Mirko Viroli (2015): *Aggregate Programming for the Internet of Things*. *IEEE Computer* 48(9), doi:10.1109/MC.2015.261.

[3] Jacob Beal & Mirko Viroli (2014): *Building blocks for aggregate programming of self-organising applications*. In: *2nd FoCAS Workshop on Fundamentals of Collective Systems*, IEEE CS, pp. 8–13, doi:10.1109/SASOW.2014.6.

[4] Paolo Boldi, Marco Rosa & Sebastiano Vigna (2011): *HyperANF: approximating the neighbourhood function of very large graphs on a budget*. In: *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pp. 625–634, doi:10.1145/1963405.1963493.

[5] Ferruccio Damiani, Mirko Viroli & Jacob Beal (2016): *A type-sound calculus of computational fields*. *Science of Computer Programming* 117, pp. 17 – 44, doi:10.1016/j.scico.2015.11.005.

[6] Ferruccio Damiani, Mirko Viroli, Danilo Pianini & Jacob Beal (2015): *Code Mobility Meets Self-organisation: A Higher-Order Calculus of Computational Fields*. In: *Formal Techniques for Distributed Objects, Components, and Systems*, Lecture Notes in Computer Science 9039, Springer, pp. 113–128, doi:10.1007/978-3-319-19195-9.

[7] Jeffrey Dean & Sanjay Ghemawat (2008): *MapReduce: simplified data processing on large clusters*. *Commun. ACM* 51(1), pp. 107–113, doi:10.1145/1327452.1327492.

[8] Philippe Flajolet, Éric Fusy, Olivier Gandouet & Frédéric Meunier (2007): *HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm*. In: *Analysis of Algorithms 2007 (AofA07)*, pp. 127–146.

[9] Atsushi Igarashi, Benjamin C. Pierce & Philip Wadler (2001): *Featherweight Java: A Minimal Core Calculus for Java and GJ*. *ACM Transactions on Programming Languages and Systems* 23(3), doi:10.1145/503502.503505.

[10] Alberto Lluch-Lafuente, Michele Loreti & Ugo Montanari (2017): *Asynchronous Distributed Execution Of Fixpoint-Based Computational Fields*. *Logical Methods in Computer Science* 13(1), doi:10.23638/LMCS-13(1:13)2017.

[11] Christopher R. Palmer, Phillip B. Gibbons & Christos Faloutsos (2002): *ANF: a fast and scalable tool for data mining in massive graphs*. In: *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23-26, 2002, Edmonton, Alberta, Canada*, pp. 81–90, doi:10.1145/775047.775059.

[12] Mirko Viroli, Jacob Beal, Ferruccio Damiani & Danilo Pianini (2015): *Efficient Engineering of Complex Self-Organising Systems by Self-Stabilising Fields*. In: *IEEE SASO 2015*, IEEE, pp. 81–90, doi:10.1109/SASO.2015.16.

[13] Mirko Viroli, Roberto Casadei & Danilo Pianini (2016): *On Execution Platforms for Large-scale Aggregate Computing*. In: *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*, UbiComp '16, ACM, New York, NY, USA, pp. 1321–1326, doi:10.1145/2968219.2979129.