



NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

**SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS**

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

Distributed and Streaming Graph Processing Techniques

Panagiotis N. Liakos

ATHENS

JUNE 2018



ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ**

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

**Τεχνικές Επεξεργασίας Κατανεμημένων Γράφων και
Ρευμάτων Γράφων**

Παναγιώτης Ν. Λιάκος

ΑΘΗΝΑ

ΙΟΥΝΙΟΣ 2018

PhD THESIS

Distributed and Streaming Graph Processing Techniques

Panagiotis N. Liakos

SUPERVISOR: Alex Delis, Professor NKUA

THREE-MEMBER ADVISORY COMMITTEE:

Alex Delis, Professor NKUA

Mema Roussopoulos, Associate Professor NKUA

Alexandros Ntoulas, Assistant Professor NKUA

SEVEN-MEMBER EXAMINATION COMMITTEE

Alex Delis,
Professor NKUA

Mema Roussopoulos,
Associate Professor NKUA

Alexandros Ntoulas,
Assistant Professor NKUA

Dimitris Gunopoulos,
Professor NKUA

Yiannis Ioannidis,
Professor NKUA

Antonios Deligiannakis,
Associate Professor TUC

Yiannis Kotidis,
Associate Professor AUEB

Examination Date: June 15, 2018

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Τεχνικές Επεξεργασίας Κατανεμημένων Γράφων και Ρευμάτων Γράφων

Παναγιώτης Ν. Λιάκος

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Αλέξιος Δελής, Καθηγητής ΕΚΠΑ

ΤΡΙΜΕΛΗΣ ΕΠΙΤΡΟΠΗ ΠΑΡΑΚΟΛΟΥΘΗΣΗΣ:

Αλέξιος Δελής, Καθηγητής ΕΚΠΑ

Μέμα Ρουσσopoύλου, Αναπληρώτρια Καθηγήτρια ΕΚΠΑ

Αλέξανδρος Ντούλας, Επίκουρος Καθηγητής ΕΚΠΑ

ΕΠΤΑΜΕΛΗΣ ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

**Αλέξιος Δελής,
Καθηγητής ΕΚΠΑ**

**Μέμα Ρουσσopoύλου,
Αναπληρώτρια Καθηγήτρια
ΕΚΠΑ**

**Αλέξανδρος Ντούλας,
Επίκουρος Καθηγητής ΕΚΠΑ**

**Δημήτρης Γουνόπουλος,
Καθηγητής ΕΚΠΑ**

**Γιάννης Ιωαννίδης,
Καθηγητής ΕΚΠΑ**

**Αντώνιος Δεληγιαννάκης,
Αναπληρωτής Καθηγητής
Πολυτεχνείου Κρήτης**

**Γιάννης Κοτίδης,
Αναπληρωτής Καθηγητής ΟΠΑ**

Ημερομηνία Εξέτασης: 15 Ιουνίου 2018

ABSTRACT

Beneath most complex systems playing a vital role in our daily lives lie intricate networks. Such real-world networks are routinely represented using graphs. The volume of graph data produced in today's interlinked world allows for realizing numerous fascinating applications but also poses important challenges. Consider for example the friendship graph of a social networking site and the findings we can come up with when executing network algorithms, such as community detection, on this graph. However, the volume that real-world networks reach oftentimes makes even the execution of fundamental graph algorithms infeasible when following traditional techniques.

In this thesis we focus on two directions that allow for handling large scale networks, namely *distributed graph processing*, and *streaming graph algorithms*. In this context, we first provide contributions with regard to memory usage of distributed graph processing systems by extending the available structures of a contemporary such system with memory-optimized representations. Then, we focus on the task of community detection and propose i) a local algorithm that reveals the community structure of a vertex and easily facilitates distributed execution and ii) a streaming algorithm that greatly outperforms non-streaming state-of-the-art approaches with respect to both execution time and memory usage. In addition, we propose a streaming sampling technique that allows for capturing the interesting part of an unmanageable volume of data produced by social activity. Finally, we exploit the available data of a popular social networking site to empirically investigate a well-studied opinion formation model, using a distributed algorithm.

SUBJECT AREA: Graph mining

KEYWORDS: Distributed graph processing, streaming graphs, graph compression, community detection, opinion formation.

ΠΕΡΙΛΗΨΗ

Κάτω από τα περισσότερα σύνθετα συστήματα που διαδραματίζουν έναν κομβικό ρόλο στην καθημερινή μας ζωή κρύβονται περίπλοκα δίκτυα. Τέτοια δίκτυα πραγματικού κόσμου μοντελοποιούνται συχνά με τη χρήση γράφων. Ο μεγάλος όγκος των γράφων που παράγονται στο σημερινό διασυνδεδεμένο κόσμο επιτρέπει την πραγματοποίηση πολυάριθμων συναρπαστικών εφαρμογών αλλά και εγείρει σημαντικές προκλήσεις. Αναλογιστείτε για παράδειγμα το γράφο φιλίας ενός ιστοχώρου κοινωνικής δικτύωσης και τα ευρήματα στα οποία μπορούμε να φτάσουμε αν εκτελέσουμε αλγορίθμους δικτύων, όπως η ανίχνευση κοινοτήτων, στο γράφο αυτό. Εντούτοις, ο όγκος τον οποίο αγγίζουν τα δίκτυα πραγματικού κόσμου συχνά καθιστούν την εκτέλεση ακόμη και θεμελιωδών αλγορίθμων γράφων αδύνατη όταν ακολουθούνται παραδοσιακές προσεγγίσεις.

Στην παρούσα διατριβή εστιάζουμε σε δύο κατευθύνσεις που επιτρέπουν τον χειρισμό δικτύων μεγάλης κλίμακας και συγκεκριμένα την *κατανεμημένη επεξεργασία γράφων* και τους *αλγορίθμους ρευμάτων γράφων*. Σε αυτό το πλαίσιο αρχικά συνεισφέρουμε όσον αφορά στη χρήση μνήμης των κατανεμημένων συστημάτων επεξεργασίας γράφων, επεκτείνοντας τις υπάρχουσες δομές ενός τέτοιου σύγχρονου συστήματος με συμπαγείς αναπαραστάσεις. Έπειτα, επικεντρωνόμαστε στο πρόβλημα της ανίχνευσης κοινοτήτων και προτείνουμε α) έναν τοπικό αλγόριθμο που αποκαλύπτει τη δομή κοινοτήτων γύρω από έναν κόμβο κι ο οποίος δύναται εύκολα να εκτελεστεί σε κατανεμημένο περιβάλλον και β) έναν αλγόριθμο ρευμάτων γράφων ο οποίος υπερκερά σημαντικά προσεγγίσεις τεχνολογίας αιχμής που χρησιμοποιούν ολόκληρο το γράφο, τόσο σε χρόνο εκτέλεσης όσο και σε χρήση μνήμης. Επιπρόσθετα, προτείνουμε μια τεχνική δειγματοληψίας που επιτρέπει την ανάκτηση του ενδιαφέροντος κομματιού ενός ρεύματος κοινωνικής δραστηριότητας που η εξ ολοκλήρου διαχείρισή του είναι αδύνατη εξαιτίας του όγκου του. Τέλος, εκμεταλλευόμαστε τα διαθέσιμα δεδομένα ενός δημοφιλούς ιστοχώρου κοινωνικής δικτύωσης ώστε να αξιολογήσουμε εμπειρικά ένα ευρέως διαδεδομένο μοντέλο διαμόρφωσης γνώμης, χρησιμοποιώντας έναν κατανεμημένο αλγόριθμο.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Εξόρυξη γράφων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Κατανεμημένη επεξεργασία γράφων, ρεύματα γράφων, συμπίεση γράφων, ανίχνευση κοινοτήτων, διαμόρφωση γνώμης.

ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΗΣ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

Η παρούσα διδακτορική διατριβή έχει τον τίτλο “Τεχνικές Επεξεργασίας Κατανεμημένων Γράφων και Ρευμάτων Γράφων (Distributed and Streaming Graph Processing Techniques)” και επικεντρώνεται στη θεματική περιοχή της μελέτης τεχνικών για αποτελεσματική επεξεργασία γράφων σε κατανεμημένα περιβάλλοντα και συστήματα ρευμάτων. Σκοπός της συγκεκριμένης διατριβής είναι να διευκολύνει το χειρισμό γράφων που αναπαριστούν δίκτυα μεγάλης κλίμακας και να προτείνει και μελετήσει τεχνικές ανάλυσής τους.

Αρχικά, επικεντρωνόμαστε στις αναπαραστάσεις που χρησιμοποιούν σύγχρονα κατανεμημένα συστήματα επεξεργασίας γράφων και προτείνουμε καινοτόμες συμπαγείς δομές με χρήση τεχνικών συμπίεσης. Οι δομές αυτές διευκολύνουν την εκτέλεση οποιουδήποτε αλγορίθμου, μειώνοντας τις απαιτήσεις του σε μνήμη. Έπειτα, προτείνουμε λύσεις σε καίρια σχετικά προβλήματα, όπως η ανίχνευση κοινοτήτων, η δειγματοληψία κοινωνικής δραστηριότητας και η διαμόρφωση γνώμης σε κοινωνικά πλαίσια. Οι λύσεις αυτές βρίσκουν εφαρμογή σε κατανεμημένα περιβάλλοντα ή σε συστήματα ρευμάτων κι έτσι προσφέρονται για το χειρισμό δικτύων μεγάλης κλίμακας.

Πιο συγκεκριμένα, η παρούσα διδακτορική διατριβή εξετάζει τα εξής πέντε θέματα:

Βελτιστοποίηση της χρήσης μνήμης των κατανεμημένων συστημάτων επεξεργασίας γράφων: Μια πληθώρα εφαρμογών χρησιμοποιεί ευρέως δεδομένα γράφων, το μέγεθος των οποίων φαίνεται να αυξάνει αδιάκοπα. Η τάση αυτή προκάλεσε την ανάπτυξη πλήθους κατανεμημένων συστημάτων επεξεργασίας γράφων μεταξύ των οποίων τα Pregel, Apache Giraph και GraphX.

Το μοντέλο Pregel καθιστά ευκολότερη την υλοποίηση κατανεμημένων αλγορίθμων και τα παραπάνω συστήματα που το υιοθετούν επιτρέπουν την εκτέλεση των αλγορίθμων αυτών πάνω σε δίκτυα μεγάλης κλίμακας, χρησιμοποιώντας συστάδες φθηνών υπολογιστικών κόμβων. Παρόλα αυτά, η άνευ προηγουμένου κλίμακα στην οποία φθάνουν πλέον οι γράφοι πραγματικού κόσμου, δυσχεραίνει την επεξεργασία τους ακόμη και σε κατανεμημένο περιβάλλον εξαιτίας των υπερβολικών απαιτήσεων μνήμης. Συνεπώς, η αναπαράσταση στη μνήμη αναδύεται ως πρωταρχικό μέλημα και στην κατανεμημένη επεξεργασία γράφων. Πιο συγκεκριμένα, τα σύγχρονα κατανεμημένα συστήματα επεξεργασίας γράφων χρησιμοποιούν δαπανηρές αναπαραστάσεις λιστών γειτνίασης, δίχως να εκμεταλλεύονται τεχνικές συμπίεσης που έχουν μελετηθεί στο ευρύτερο πεδίο της επεξεργασίας γράφων. Επιπλέον, ορισμένες αβλεψίες στις λεπτομέρειες υλοποίησης των αναπαραστάσεων που χρησιμοποιούν τα συστήματα αυτά, αυξάνουν περαιτέρω τις απαιτήσεις τους σε μνήμη.

Στην παρούσα εργασία αντιμετωπίζουμε τις παραπάνω προκλήσεις εκμεταλλευόμενοι εμπειρικές ιδιότητες γράφων πραγματικού κόσμου. Συγκεκριμένα, προτείνουμε:

- τρεις συμπίεσμένες αναπαραστάσεις λιστών γειτνίασης που μπορούν να εφαρμοστούν σε οποιοδήποτε κατανεμημένο σύστημα επεξεργασίας γράφων,

- μία συμπιεσμένη αναπαράσταση η οποία υποστηρίζει αποτελεσματικά γράφους με βάρη στις ακμές τους και
- μια συμπαγή δενδρική αναπαράσταση γειτόνων που εκτελεί αποτελεσματικά προσ-
θαφαιρέσεις ακμών.

Οι συμπιεσμένες αναπαραστάσεις αφορούν πρωτότυπες τεχνικές συμπίεσης που μειώνουν το κόστος αναπαράστασης χωρίς να επιβαρύνουν τα συστήματα επεξεργασίας με επιπρόσθετα κόστη ανάκτησης των στοιχείων του γράφου, αλλά και γνωστές τεχνικές στο πεδίο της συμπίεσης γράφων που δεν έχουν όμως δοκιμαστεί σε κατανεμημένο περιβάλλον.

Στην κατεύθυνση της υποστήριξης επεξεργασίας γράφων με βάρη στις ακμές τους, προτείνουμε μια ακόμη καινοτόμα αναπαράσταση η οποία χρησιμοποιεί *variable byte* κώδικες. Μάλιστα, η αναπαράσταση αυτή μπορεί να χρησιμοποιηθεί σε συνδυασμό με κάποιες από τις συμπιεσμένες αναπαραστάσεις λιστών γειτνίασης προσφέροντας επιπλέον όφελος.

Τέλος, η δενδρική αναπαράσταση γειτόνων που προτείνουμε χρησιμοποιεί ένα *Κόκκινο-μαύρο δένδρο* (Red-black tree) με το οποίο επιτυγχάνουμε σημαντικά οφέλη στην εξοικονόμηση μνήμης σε σύγκριση με την προκαθορισμένη αναπαράσταση ενός ευρέως διαδεδομένου κατανεμημένου συστήματος επεξεργασίας γράφων που βασίζεται σε έναν *πίνακα κατακερματισμού* (Hash table).

Η υλοποίηση όλων των αναπαραστάσεων μας έγινε επεκτείνοντας το Apache Giraph, το οποίο χρησιμοποιούμε και στην πειραματική μας αξιολόγηση ώστε να εκτιμήσουμε τα οφέλη που μπορούν να προσφέρουν σε ένα υπάρχον σύστημα οι τεχνικές μας. Εξετάζουμε συνολικά την εκτέλεση τριών κατανεμημένων αλγορίθμων χρησιμοποιώντας δημόσια διαθέσιμους γράφους, το μέγεθος των οποίων φτάνει τις 2 δισ. ακμές. Τα ευρήματά μας αφορούν τόσο τα οφέλη όσον αφορά στη χρήση μνήμης όσο και στο χρόνο εκτέλεσης που επιτυγχάνουμε με χρήση των τεχνικών μας. Συγκεκριμένα, αρχικά δείχνουμε πως οι προτεινόμενες αναπαραστάσεις μειώνουν τις απαιτήσεις σε μνήμη μέχρι και 5 φορές σε σχέση με την τεχνολογία αιχμής. Παράλληλα, οι τεχνικές μας διατηρούν την αποτελεσματικότητα ασυμπίεστων δομών επιτυγχάνοντας ταχύτητες εκτέλεσης αντίστοιχες με αυτές των αναπαραστάσεων που δεν χρησιμοποιούν συμπίεση. Τέλος, δείχνουμε πως οι προτεινόμενες τεχνικές επιτρέπουν την εκτέλεση αλγορίθμων σε γράφους μεγάλης κλίμακας σε ρυθμίσεις με τις οποίες σύγχρονες εναλλακτικές δομές αποτυγχάνουν λόγω περιορισμένης διαθέσιμης μνήμης.

Τοπική ανίχνευση κοινοτήτων σε δίκτυα μεγάλης κλίμακας: Η αποκάλυψη της δομής των κοινοτήτων ενός δικτύου είναι καίριας σημασίας για την καλύτερη κατανόηση της σύνθετης φύσης του. Εντούτοις, το συνεχώς αυξανόμενο μέγεθος των γράφων πραγματικού κόσμου και η εξέλιξη της αντίληψης του τι αποτελεί κοινότητα καθιστούν το έργο της ανίχνευσης κοινοτήτων ιδιαίτερα απαιτητικό.

Μια σχετική πρόκληση είναι η ανίχνευση των πιθανώς επικαλυπτόμενων κοινοτήτων ενός κόμβου σε ένα γράφο μεγάλης κλίμακας, ένα πρόβλημα αρκετά συνηθισμένο στα σύγχρονα κοινωνικά δίκτυα όπως το Facebook και το LinkedIn. Σε αυτή την εργασία προτείνουμε μια κλιμακούμενη τοπική προσέγγιση ανίχνευσης κοινοτήτων για να ανιχνεύουμε

αποτελεσματικά τις κοινότητες συγκεκριμένων κόμβων σε ένα δίκτυο. Ο στόχος μας είναι να αποκαλύψουμε τις ομάδες που σχηματίζονται γύρω από τους κόμβους (χρήστες) αξιοποιώντας τις σχέσεις των διαφορετικών πλαισίων στα οποία δραστηριοποιούνται.

Οι τοπικές προσεγγίσεις ανίχνευσης κοινοτήτων δέχονται σαν είσοδο έναν κόμβο ή μια μικρή ομάδα κόμβων και περιορίζουν το χώρο αναζήτησης σε ένα υπογράφο του οποίου οι κόμβοι βρίσκονται τοπολογικά κοντά στην είσοδο αυτή, επιτυγχάνοντας έτσι αποτελεσματικότητα και υψηλή απόδοση. Εμείς εστιάζουμε στην περίπτωση που η είσοδος αποτελείται από έναν μόνο κόμβο. Η περίπτωση αυτή παρουσιάζει ιδιαίτερο ενδιαφέρον καθώς συγκεντρώνει την προσοχή γύρω από ένα κόμβο όπως ακριβώς και τα περισσότερα σύγχρονα κατανεμημένα συστήματα επεξεργασίας γράφων που υιοθετούν το μοντέλο Pregel. Επομένως, η κατανεμημένη εκδοχή της επιτυγχάνεται ιδιαίτερα απλοϊκά. Παράλληλα, η συγκεκριμένη περίπτωση είναι ενδιαφέρουσα γιατί προκαλεί προβλήματα σε παλαιότερες προσεγγίσεις εξαιτίας της ύπαρξης επικαλυπτόμενων κοινοτήτων. Έχοντας μόνο έναν κόμβο ως σημείο αναφοράς, οι προσεγγίσεις αυτές συχνά αδυνατούν να ανιχνεύσουν τις επικαλυπτόμενες κοινότητες ως διακριτές αλλά αντιθέτως τις αναγνωρίζουν συνολικά ως μία κοινότητα.

Ο αλγόριθμός μας, με όνομα LDLC, μετρά την ομοιότητα των ζευγαριών των συνδέσμων στο γράφο καθώς και το βαθμό της συμμετοχής τους σε διαφορετικά πλαίσια. Για να το πετύχει αυτό χρησιμοποιεί μία πρόσφατα προτεινόμενη μετρική διασποράς. Έπειτα, καθορίζει τη σειρά με την οποία πρέπει να ομαδοποιηθούν οι σύνδεσμοι για να σχηματιστούν οι κοινότητες ακολουθώντας ιεραρχική συσταδοποίηση. Η προσέγγισή μας είναι απασχλημένη από περιορισμούς που αντιμετώπιζαν προηγούμενες τεχνικές, όπως η ανάγκη για πολλαπλούς κόμβους αρχικοποίησης μιας κοινότητας, ή η ανάγκη συγχώνευσης πολλαπλών επικαλυπτόμενων κοινοτήτων σε μία.

Στην πειραματική μας αξιολόγηση χρησιμοποιούμε πλήθος μεγάλης κλίμακας δικτύων πραγματικού κόσμου που συνοδεύονται από κοινότητες αντικειμενικής αλήθειας (ground-truth). Τα ευρήματά μας δείχνουν πως ο αλγόριθμος που προτείνουμε ξεπερνά σημαντικά τις επιδόσεις μεθόδων τεχνολογίας αιχμής τόσο σε ακρίβεια όσο και σε αποτελεσματικότητα. Επιπλέον, η μετρική διασποράς που χρησιμοποιούμε συντελεί στην παραγωγή ιδιαίτερα αναλυτικών δένδρογραμμάτων τα οποία περιγράφουν λεπτομερώς τη δομή των κοινοτήτων του κόμβου υπό εξέταση.

Ανίχνευση κοινοτήτων σε συστήματα ρευμάτων: Το πρόβλημα της ανίχνευσης κοινοτήτων συγκεντρώνει εξαιρετικό ενδιαφέρον και στο πλαίσιο των δικτύων που αναπαρίστανται με τη χρήση ρευμάτων γράφων. Τέτοια ρεύματα επιτρέπουν το χειρισμό δικτύων η αναπαράσταση των οποίων στην κύρια μνήμη είναι αδύνατη εξαιτίας του μεγέθους τους.

Σε αυτή την εργασία προτείνουμε έναν αλγόριθμο ανίχνευσης κοινοτήτων για ρεύματα γράφων ο οποίος επεκτείνει μικρά σύνολα κόμβων σε κοινότητες. Θεωρούμε ως είσοδο ένα πλήθος συνόλων κόμβων κάθε ένα από τα οποία περιγράφει μια κοινότητα που μας ενδιαφέρει. Τα σύνολα αυτά πρέπει να επεκταθούν με τους κόμβους εκείνους που αποτελούν κομμάτι της αντίστοιχης κοινότητας. Θεωρούμε επίσης ένα ρεύμα από ακμές το οποίο επεξεργαζόμαστε χωρίς να διατηρούμε στη μνήμη ολόκληρη τη δομή του αντίστοιχου γράφου. Αντιθέτως, διατηρούμε ελάχιστη πληροφορία σχετικά με τους κόμβους του

και τις κοινότητες που αναζητούμε. Συγκεκριμένα, πέρα από τις κοινότητες που σχηματίζουμε καθώς επεξεργαζόμαστε το ρεύμα, διατηρούμε για κάθε κόμβο το πλήθος των γειτόνων του, αλλά και μια μετρική που ορίζουμε και η οποία ποσοτικοποιεί την πιθανότητα συμμετοχής του σε κάθε μία από τις κοινότητες που αναζητούμε.

Πέρα από την καινοτόμα προσέγγισή μας, αναπτύσσουμε μια τεχνική που βελτιώνει σημαντικά την ακρίβεια του αλγορίθμου μας και προτείνουμε ένα νέο αλγόριθμο συσταδοποίησης που επιτρέπει την αυτόματη ανίχνευση του μεγέθους των κοινοτήτων που αναζητούμε. Η πρώτη τεχνική φροντίζει ώστε οι κόμβοι που είναι περισσότερο εδραιωμένοι σε μια υπό σχηματισμό κοινότητα να ωθούν με μεγαλύτερη ισχύ τους γείτονές τους στην κοινότητα αυτή, σε σχέση με τους λιγότερο εδραιωμένους κόμβους που επίσης αποτελούν κομμάτι της κοινότητας. Με αυτόν τον τρόπο η εστίαση του αλγορίθμου δεν απομακρύνεται από την κοινότητα που αναζητούμε. Επιπλέον, ο καινοτόμος αλγόριθμος συσταδοποίησης που προτείνουμε χρησιμοποιεί την κατανομή των τιμών κάθε κόμβου όσον αφορά τη μετρική μας για τη συμμετοχή του στην κοινότητα προς ανίχνευση ώστε να αποφασίσει ποιοι κόμβοι δεν αποτελούν κομμάτι της κοινότητας.

Η πειραματική μας αξιολόγηση γίνεται σε ένα πλήθος μεγάλης κλίμακας δικτύων πραγματικού κόσμου που συνοδεύονται από κοινότητες αντικειμενικής αλήθειας. Στην αξιολόγηση αυτή δείχνουμε πως η προτεινόμενη προσέγγιση επιτυγχάνει ακρίβεια αντίστοιχη ή και καλύτερη αυτής των μεθόδων τεχνολογίας αιχμής που λειτουργούν σε ολόκληρη τη δομή του γράφου. Πέραν αυτού, τόσο ο χρόνος εκτέλεσης όσο και οι απαιτήσεις σε μνήμη που επιτυγχάνουμε είναι πραγματικά αξιοσημείωτες. Τέλος, εξακριβώνουμε πειραματικά πως οι δύο επιπρόσθετες τεχνικές που προτείνουμε για βελτίωση της ακρίβειας του αλγορίθμου μας και αυτόματη ανίχνευση του μεγέθους των κοινοτήτων που αναζητούμε είναι εξαιρετικά αποδοτικές.

Δυναμική δειγματοληψία ποιοτικού περιεχομένου σε ρεύματα κοινωνικής δραστηριότητας: Ο τρομακτικός ρυθμός με τον οποίο παράγεται πληροφορία στις υπηρεσίες κοινωνικής δικτύωσης φέρνει στην επιφάνεια σημαντικές προκλήσεις σε εφαρμογές όπως η σύσταση περιεχομένου, η εξόρυξη γνώμης, η ανάλυση συναισθημάτων και η ανίχνευση αναδυόμενων ειδήσεων. Η επεξεργασία ενός πλήρους ρεύματος δραστηριότητας ενός κοινωνικού δικτύου σε πραγματικό χρόνο είναι συχνά απαγορευτική, τόσο σε κόστος αποθήκευσης όσο και σε υπολογιστικό κόστος.

Μια πιθανή προσέγγιση επίλυσης αυτού του προβλήματος είναι η δειγματοληψία της δραστηριότητας ώστε να χρησιμοποιηθεί μόνο το δείγμα ως είσοδος σε εφαρμογές όπως οι προαναφερθείσες. Στην παρούσα εργασία μελετούμε το πρόβλημα της δειγματοληψίας αναρτήσεων σε ένα ρεύμα δραστηριότητας ενός κοινωνικού δικτύου, οι οποίες ανήκουν σε εκείνους τους χρήστες που είναι πιο πιθανό να αναρτούν πληροφορία με *επιρροή*. Επομένως, μια σημαντική πρόκληση που καλούμαστε να αντιμετωπίσουμε είναι η ανίχνευση των χρηστών αυτών. Προγενέστερες προσεγγίσεις βασίζουν τη λειτουργία τους σε στατικές λίστες εγκεκριμένων χρηστών, το περιεχόμενο των οποίων συμπεριλαμβάνεται στο τελικό δείγμα. Η χρήση στατικών λιστών όμως καθιστά αδύνατη την προσαρμογή σε εξελισσόμενες τάσεις στο ρεύμα της δραστηριότητας.

Για το λόγο αυτό προχωρήσαμε χτίζοντας σε μια μετρική που έχει προταθεί στη βιβλιο-

γραφία στα πλαίσια της εξόρυξης εξειδικευμένων χρηστών. Προσαρμόσαμε τη μετρική κατάλληλα ώστε να βρίσκει εφαρμογή σε ένα ευρύ πεδίο κοινωνικών δικτύων, και τη χρησιμοποιούμε δυναμικά, καθώς δηλαδή συλλέγουμε το δείγμα. Εξ' όσων γνωρίζουμε, η προσέγγισή μας είναι η πρώτη που προσαρμόζεται με το χρόνο ώστε να λαμβάνει υπόψη της εξελισσόμενες τάσεις στο ρεύμα της δραστηριότητας. Έτσι, επιτυγχάνουμε τη συλλογή υψηλής ποιότητας περιεχομένου από χρήστες τους οποίους εναλλακτικές στατικές μέθοδοι αγνοούν.

Τα αποτελέσματα της αξιολόγησής μας πάνω σε δύο δημοφιλή κοινωνικά δίκτυα και έως μισό δισ. αναρτήσεις δείχνουν πως η προσέγγισή μας υπερέχει έναντι προγενέστερων σε ανάκληση και ακρίβεια, ενώ προσφέρει και σημαντικά ακριβέστερη σειρά κατάταξης των αποτελεσμάτων.

Εμπειρική μελέτη των δυναμικών διαμόρφωσης γνώμης στα κοινωνικά δίκτυα: Η διαμόρφωση γνώμης σε ένα κοινωνικό πλαίσιο είναι ένα πρόβλημα που έχει απασχολήσει ευρέως την ερευνητική κοινότητα και ιδιαίτερα το πεδίο της κοινωνιολογίας. Στις μέρες μας, κι εξαιτίας της δημοφιλίας των κοινωνικών δικτύων, ο όγκος της διαθέσιμης σχετικής πληροφορίας είναι μεγαλύτερος από ποτέ. Μπορούμε συνεπώς να μελετήσουμε σε μεγάλη κλίμακα τη συμπεριφορά των ανθρώπων και να αναλύσουμε μεταξύ άλλων τη διαδικασία της διαμόρφωσης γνώμης σε ένα κοινωνικό πλαίσιο.

Έχει παρατηρηθεί πως σε ένα κοινωνικό περιβάλλον οι άνθρωποι συχνά σχηματίζουν τη γνώμη τους συναρτήσει της γνώμης των φίλων τους, σε μια προσπάθεια ίσως να τονίσουν τις κοινές τους πεποιθήσεις. Χρησιμοποιώντας ένα δημοφιλές κοινωνικό δίκτυο, αναλύουμε τη δραστηριότητα των χρηστών και καταλήγουμε πως η κοινωνική αλληλεπίδραση όντως επηρεάζει τη γνώμη των συμμετεχόντων. Στο πλαίσιο αυτό έχουν προταθεί διάφορα μοντέλα που επιδιώκουν να συλλάβουν τη συμπεριφορά των χρηστών. Στην παρούσα εργασία χρησιμοποιούμε ένα τέτοιο μοντέλο, που θεωρεί πως η γνώμη των συμμετεχόντων καθορίζεται από τις γνώμες που εκφράζουν οι άνθρωποι με τους οποίους έρχονται σε επαφή, χωρίς όμως οι συμμετέχοντες τελικά να εκφράζουν απαραίτητα την ίδια γνώμη. Μοντελοποιώντας λοιπόν την αλληλεπίδραση των χρηστών με χρήση της εργαλειοθήκης της θεωρίας παιγνίων και υλοποιώντας έναν κατανεμημένο αλγόριθμο που εφαρμόζεται επαναληπτικά έως ότου οι γνώμες των χρηστών συγκλίνουν, δείχνουμε πως καταλήγουμε σε σημεία ισορροπίας Nash που είναι ενδεικτικά της αληθινής συμπεριφοράς των χρηστών.

Αποτελέσματα των παραπάνω ερευνητικών προσπαθειών δημοσιεύθηκαν στον πλέον έγκριτο περιοδικό διαχείρισης δεδομένων (IEEE TKDE - [85]) και παρουσιάστηκαν σε κορυφαία συνέδρια της ίδιας περιοχής (ACM CIKM 2016 - [84], IEEE Big Data 2016 & 2017 - [80, 81, 82], AAAI ICWSM 2016 - [83]) Η διατριβή χαρακτηρίζεται από πρωτοτυπία και πληρότητα.

To my brother, mother & father.

For the Fat Lady.

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to my advisor, Prof. Alex Delis, for his continuous guidance as well as for the pressure he put on me, that helped me keep my progress on schedule. His faith in me has been extremely flattering, motivating and reassuring and I will be forever indebted to him for the kind words he often shared with me. I am also in debt to him for the many wonderful acquaintances I might not have made without him, among which are the two other members of my advisory committee, Prof. Mema Roussopoulos and Prof. Alexandros Ntoulas. I would like to offer my special thanks to Prof. Mema Roussopoulos for her eagerness to help me and her overall mentality. I am also particularly grateful for the many chances I had to collaborate with Prof. Alexandros Ntoulas, who has provided me with brilliant ideas and enthusiastic encouragement.

Besides the members of my advisory committee, the following exceptional individuals have played a vital role in my becoming a researcher. First and foremost, I would like to express my very great appreciation to Dr. Katia Papakonstantinou whose pure love for research and passion for her work has pretty much defined my research mentality. I am also grateful for the many times we have collaborated, as working with her has been inspiring and has kept me constantly motivated to improve. I would also like to thank Dr. Michael Sioutis for his useful critiques on my research as well as for creating a friendly atmosphere between the members of my first office in the University of Athens, which was at the time packed with extraordinary talented researchers. Dr. Nikos Leonardos has provided me with advice that has been greatly appreciated. His words always are, as you rarely encounter someone who is as brilliant and as humble. Finally, I wish to acknowledge two colleagues that have helped me shape my work ethic: Katerina El Raheb, who I look up to for her professionalism, integrity and talent, and Alexandros Antoniadis, who questions everything in pursuit of his goals.

I sincerely hope that all of you “will think of me again someday”.

CONTENTS

1. INTRODUCTION	31
1.1 Contribution	31
1.2 Outline	32
1.3 Credits	33
2. MEMORY-OPTIMIZED DISTRIBUTED GRAPH PROCESSING	35
2.1 Related Work	37
2.2 Background	39
2.2.1 Pregel	40
2.2.2 Apache Giraph	41
2.2.3 Properties of Real-World Graphs	42
2.2.4 Codings for Graph Compression	42
2.3 Overview of our approach	43
2.3.1 Representations based on consecutive out-edges	44
2.3.1.1 BVEdges	44
2.3.1.2 IntervalResidualEdges	45
2.3.2 IndexedBitArrayEdges	46
2.3.3 VariableByteArrayWeights	48
2.3.4 RedBlackTreeEdges	49
2.4 Experimental Evaluation	51
2.4.1 Experimental Setting	52
2.4.2 Space Efficiency Comparison	52
2.4.3 Execution Time Comparison	53
2.4.3.1 PageRank Computation	53
2.4.3.2 Shortest Paths Computation	54
2.4.3.3 Comparison using small-scale graphs	55
2.4.3.4 Comparison using large-scale graphs	56
2.4.3.5 Initialization time comparison	58
2.4.3.6 Comparison when using weighted graphs	59
2.4.3.7 Comparison when performing mutations	60
2.5 Conclusion	62
3. UNCOVERING LOCAL HIERARCHICAL LINK COMMUNITIES AT SCALE	63
3.1 Background	65
3.1.1 Egonet	65

3.1.2	Tie Strength Measures	66
3.1.2.1	Embeddedness	66
3.1.2.2	Jaccard similarity coefficient	66
3.1.2.3	Absolute and Recursive Dispersion	66
3.1.3	Partition Density	67
3.1.4	Networks in our Dataset	68
3.2	Local Dispersion-aware Link Communities	68
3.2.1	Egonet Coverage Ratio	69
3.2.2	Effective Detection of Local Hierarchical Overlapping Communities	69
3.2.2.1	Local hierarchical link communities	71
3.2.2.2	Building on dispersion-based measures	73
3.2.3	Our Proposed LDLC Algorithm	73
3.2.4	Reducing the Search Space	76
3.3	Experimental Evaluation	76
3.3.1	Experimental Setting	77
3.3.2	Qualitative Evaluation	78
3.3.3	Evaluation via Ground-Truth	80
3.3.4	Execution Time Comparison	82
3.3.5	Impact of Dispersion on the Resulting Hierarchical Community Structure	82
3.3.6	Impact of Sampling Technique	83
3.4	Related Work	85
3.5	Conclusion	87
4.	COMMUNITY DETECTION VIA SEED SET EXPANSION ON GRAPH STREAMS	89
4.1	Community Detection via Seed-Set Expansion on Graph Streams	91
4.1.1	Problem formulation	91
4.1.2	Space complexity	92
4.1.3	Our CoEuS Algorithm for Streaming Community Detection	94
4.1.4	Reckoning in edge quality w.r.t. each community	96
4.1.5	Size of the community	98
4.2	Experimental Evaluation	100
4.2.1	Experimental Setting	100
4.2.2	Impact of the Edge Quality Variation	102
4.2.3	Evaluation of Automatic Size Determination	102
4.2.4	Comparison against state-of-the-art non-streaming local community detection algorithms	103
4.2.4.1	F1-score comparison	104
4.2.4.2	Execution time and space efficiency comparison	104
4.3	Related Work	106
4.4	Conclusion	107
5.	ADAPTIVELY SAMPLING AUTHORITATIVE CONTENT FROM SOCIAL AC-	
	TIVITY STREAMS	109
5.1	Identifying Authorities in Streams	110
5.1.1	Network of Authorities from Social Activity	110
5.1.2	Ranking the Authorities	111

5.1.3	Limitations of Static Lists of Authorities	113
5.2	Rhea: Stream Sampling for Authoritative Content	115
5.2.1	Maintaining User Information	115
5.2.1.1	Frequent Items	115
5.2.1.2	Reducing the Processing Overhead through Sampling	116
5.2.2	Ranking Authorities	117
5.2.3	Filtering-out Non-relevant Activity	118
5.2.4	The Proposed Rhea Algorithm	118
5.3	Experimental Evaluation	120
5.3.1	Experimental Setting	121
5.3.2	Recall, Precision, and F1-score Comparison	121
5.3.3	Evaluation of Ranked Retrieval Results	122
5.3.3.1	Evaluation using Spearman's ρ	123
5.3.3.2	Evaluation using NDCG	123
5.3.4	Impact of Techniques and Parameters	124
5.3.4.1	Varying the Value of Probability p	125
5.3.4.2	Removing the Filtering Step	125
5.3.4.3	Impact of the Capacity of the Top-K-Heap	126
5.4	Related Work	127
5.5	Conclusion	129
6.	ON THE IMPACT OF SOCIAL COST ON OPINION DYNAMICS	131
6.1	Model	132
6.2	Empirical analysis	133
6.2.1	Information propagation and reproductive ratio	134
6.2.2	Frequent cascade patterns	134
6.3	Experimental evaluation	134
6.3.1	Simulation methodology	135
6.3.2	Experiments using real-world data	138
6.4	Conclusion	139
7.	CONCLUSION AND OPEN DIRECTIONS	141
	ABBREVIATIONS - ACRONYMS	143
	REFERENCES	151

LIST OF FIGURES

Figure 1: A graph partitioned on a vertex basis in a distributed environment. Each vertex maintains a list of its out-edges.	35
Figure 2: The <i>Pregel</i> programming model: workers compute in parallel the vertices' actions at every <i>superstep</i> and messages between iterations are synchronized using a barrier before every <i>superstep</i> commences.	40
Figure 3: Giraph's adjacency-list representations: <i>ByteArrayEdges</i> (a) and <i>HashMapEdges</i> (b).	41
Figure 4: The storage of neighbors in <i>BVEEdges</i> , detailed in Example 1. $\gamma(x)$ and $\zeta(x)$ denote the γ and ζ encodings of x respectively.	45
Figure 5: The storage of neighbors in <i>IntervalResidualEdges</i> , detailed in Example 2. $(x)_2$ is the binary representation of x	46
Figure 6: A bit-array representation of an adjacency list and the storage of these neighbors in <i>IndexedBitArrayEdges</i> , detailed in Example 3. $(x)_2$ denotes the binary representation of x	47
Figure 7: The storage of edge weights using <i>VariableByteArrayWeights</i> . Weights of neighbors are held in variable-byte encoding. Two weights (32 and 378) are represented using only one and two bytes, respectively. <i>VariableByteArrayWeights</i> can extend <i>BVEEdges</i> and <i>IntervalResidualEdges</i> to support weighted graphs.	49
Figure 8: The storage of neighbors in <i>RedBlackTreeEdges</i> . Neighbors' ids are inserted as keys to a <i>red-black tree</i> . For weighted graphs each node would additionally maintain a variable to hold the weight.	50
Figure 9: Execution time (in minutes) of PageRank algorithm for <i>indochina-2004</i> using a setup of 2, 4, and 8 workers.	55
Figure 10: Execution time (in minutes) for each <i>superstep</i> of the PageRank algorithm for the graph <i>uk-2005</i> using 5 workers. <i>ByteArrayEdges</i> performance fluctuates due to extensive garbage collection.	57
Figure 11: Execution time (in minutes) of the PageRank algorithm for the graph <i>uk-2005</i> using 5 and 4 workers. <i>IntervalResidualEdges</i> and <i>IndexedBitArrayEdges</i> outperform <i>ByteArrayEdges</i> which fails to complete execution with 4 workers.	58
Figure 12: Initialization time (in seconds) for graphs <i>indochina-2004</i> (using 2 workers) and <i>uk-2005</i> (using 5 workers). There is notable performance gain on large-scale graphs over <i>ByteArrayEdges</i> when using <i>IndexedBitArrayEdges</i> or <i>IntervalResidualEdges</i> . <i>BVEEdges</i> is the slowest of the representations examined.	59

Figure 13: Execution time (in minutes) of the ShortestPaths algorithm for a single vertex, on the graph <i>uk-2005</i> , using a setup of 5 and 4 workers.	60
Figure 14: Execution time (in minutes) of an algorithm performing a random number of mutations on the graph <i>hollywood-2011</i> using 5 workers, for a varying number of maximum mutations allowed.	61
Figure 15: Illustration of the social circles of an individual. Her family, co-workers, basketball buddies and friends from college are distinct yet overlapping communities.	63
Figure 16: Egonet coverage ratio for the ground-truth communities of graphs provided by SNAP. We show that the coverage ratio for all graphs, with the exception of <i>Orkut</i> , is above 87%. The ratio is lower for <i>Orkut</i> due to its large average ground-truth community size.	70
Figure 17: Social communities in the egonet of an individual (10) in a social network. Using a force-directed layout we can easily identify two well-connected groups of acquaintances. A special tie between (10) and (6) is evident, as (6) is the only vertex having links (in red) towards both communities.	71
Figure 18: The hierarchical link structure of the malformed community that results when performing link clustering in the egonet of Figure 17 using Equation (3.6), and cutting at the level of optimal partition density. The similarity of link (6,10) with link (8,10) leads to a community that groups numerous nodes that are not linked with each other.	72
Figure 19: The egonet of a node in the <i>DBLP</i> graph. LEMON's detected community (19a) features, among others, all the nodes in the egonet. Numbers indicate the LEMON's ranking of the nodes according to their likelihood of belonging to the detected community. LDLC uses hierarchical link clustering in the egonet of the target node and penalizes the links with nodes exhibiting high dispersion to come up with two communities, colored teal and pink (19b).	79
Figure 20: Impact of the use of recursive dispersion on the number of merges that occur until LDLC terminates. When using recursive dispersion (LDLC) the number of total merges increases significantly. Thus, the resulting dendrogram reveals the hierarchical community structure in greater detail.	83
Figure 21: LDLC results on the networks of our dataset when using a random sampling technique and our k-largest sampling technique. Impact is negligible for the four first networks as very few or no egonets surpass 100 nodes. However, the k-largest sampling technique outperforms the random technique for <i>orkut</i> and <i>friendster</i>	84
Figure 22: A stream comprising the edges of an undirected graph and a set of communities initialized with a few seed nodes. For every edge of the stream we wish to evaluate whether the adjacent nodes belong to the communities we examine.	90

Figure 23: Count-Min sketch update process.	94
Figure 24: Ranking of nodes according to their community participation values and the partitioning that Algorithm 6 makes to come up with a community automatically for a random citation network community.	98
Figure 25: F1-score comparison for CoEuS when incrementing community degree by 1 (CoEuS ₁) and by community degree of the adjacent node (CoEuS _{cp}). The variation of CoEuS _{cp} clearly improves the F1-score for all graphs our dataset. The improvement is impressive for graphs <i>orkut</i> and <i>dblp</i>	101
Figure 26: F1-score comparison between LEMON and CoEuS.	103
Figure 27: Deriving a network of authorities from a social activity stream. Potential authorities may be identified by applying measures on the resulting weighted directed graph.	112
Figure 28: Precision@K 28a and Spearman's ρ 28b results for the authorities extracted from the tweets of September 2009, using the rankings resulting from the tweets of the three subsequent months. Both metrics reveal that the correlation between rankings of authorities according to the tweets of subsequent months weakens significantly with time.	113
Figure 29: Count-Min Sketch update process.	116
Figure 30: <i>Recall</i> comparison between our approach and a baseline for our two datasets (T, S0) when querying for the tweets of the top-100, 250, 500, 750, and 1,000 authorities of the stream.	120
Figure 31: <i>Precision</i> comparison between our approach and a baseline for our two datasets (T, S0) when querying for the tweets of the top-100, 250, 500, 750, and 1,000 authorities of the stream.	121
Figure 32: <i>F1-score</i> comparison between our approach and a baseline for our two datasets (T, S0) when querying for the tweets of the top-100, 250, 500, 750, and 1,000 authorities of the stream.	122
Figure 33: Comparison of Rhea and WhiteList on Spearman's ρ for Twitter (T) and StackOverflow (S0).	123
Figure 34: Comparison of Rhea and WhiteList on NDCG for Twitter (T) and StackOverflow (S0).	125
Figure 35: Impact of probability p on <i>NDCG</i>	126
Figure 36: Impact of the filtering step of Rhea on <i>precision</i>	127
Figure 37: Impact of the capacity of the <i>Top-K-Heap</i> on <i>F1-score</i>	128
Figure 38: Illustration of the model of [48]. Individuals' hold internal opinions and form their expressed opinions by additionally considering their friends' expressed opinions due to social interaction.	132
Figure 39: Top-20 cascades that occurred in 50 randomly selected <i>stories</i> of the <i>digg</i> dataset, ordered by frequency.	135
Figure 40: Cumulative <i>precision/recall</i> curve for the two configurations of our model and a <i>Ridge regression</i> classifier for all the <i>stories</i> of our dataset.	138

LIST OF TABLES

Table 1: Dataset of our experimental setting with a total of ten publicly available web and social network graphs [20, 18].	52
Table 2: Memory requirements of Giraph’s ByteArrayEdges and our three out-edge representations for the small and large-scale graphs of our dataset. Requirements of BV [20] in a centralized setting are also listed to provide an indication of the compressibility potential of each graph.	53
Table 3: Graphs of our dataset reaching up to 1.8 billion edges.	67
Table 4: F1 Score comparison.	80
Table 5: Execution time comparison.	80
Table 6: Graphs of our dataset reaching up to 1.8 billion edges.	100
Table 7: Execution time comparison between CoEuS and LEMON for all the graphs of our dataset. CoEuS is remarkably fast, even for the largest network of our dataset and clearly outperforms LEMON.	105
Table 8: Comparison of space requirements between CoEuS and LEMON for all the graphs of our dataset. CoEuS uses two Count-Min sketches to hold a graph’s elements and therefore its requirements depend only on the desired approximation quality of the sketches. LEMON maintains the adjacency lists of a graph and thus requires significantly more space. . . .	106
Table 9: Top-10 authorities for the tweets of 3 months.	114

1. INTRODUCTION

Real-life systems involving interacting objects are typically modeled as graphs and can often grow very large in size. A multitude of contemporary applications heavily involves such graph data and has driven to research directions that allow for efficient handling of large scale networks. Two prominent such directions are distributed graph processing and streaming graph algorithms.

The tremendous growth of the Web graph has driven Google to introduce Pregel, a scalable platform with an API that allows for expressing arbitrary graph algorithms. Pregel is a distributed graph processing system that powers the computation of PageRank and has served as an inspiration to many systems that adopted its programming model. One such system is Apache Giraph which originated as the open-source counterpart of Pregel. Giraph is maintained by developers of Facebook that use it to analyze Facebook's social graph. Pregel-like systems follow a vertex-centric approach and address the task of in-memory batch processing of large scale graphs [60]. Communication details are abstracted away from the developers that implements algorithms for such systems. The latter offer APIs that allow for specifying computations with regard to what each vertex of the graph needs to compute whereas edges serve the purpose of transmitting results from one vertex to another. The input graph is loaded on start-up and the entire execution takes place in-memory. Consequently, the execution of a graph algorithm in a Pregel-like system depends on the available memory and will fail if the later is not sufficient enough to fit the graph.

The ever-increasing size of real-world networks has also motivated the design of algorithms that process massive graphs in the data stream model [97]. More specifically, the input of algorithms in this model is defined by a stream of data which usually comprises the edges of the graph. Therefore, graph stream algorithms are a perfect fit for problems dealing with networks that are formed as we attempt to analyze them, e.g., the network describing the activity taking place in a social networking site. However, many challenges arise in a streaming setting that need to be addressed when designing respective techniques. A streaming graph algorithm processes the stream in the order it arrives and each element of the stream must be processed immediately or stored as it will not become available again. In addition, the size of the stream and the speed in which its elements arrive do not allow for persisting the stream in its entirety. Therefore, processing cannot occur at a later stage.

1.1 Contribution

In this thesis we focus on both distributed and streaming graph processing techniques. We begin by investigating the memory usage patterns that contemporary distributed graph processing systems adopt. We observe that graph compression techniques have not been considered in the design of the representations that distributed systems employ. There-

fore, we build on compression techniques that assume centralized execution and provide numerous novel compact representations that are fitting for all Pregel-like systems. Our structures offer memory-optimization regardless of the algorithm that is to be executed, and enable the successful execution of algorithms in settings that state-of-the-art systems fail to terminate. We continue by studying a problem that has received considerable attention in the past, yet is still extremely relevant as previously proposed approaches fail to handle the massive volume of today’s real-world graphs. In particular, we address the problem of community detection and our contribution is twofold as we propose a vertex-centric and a streaming approach. We follow the trend of seed-set expansion methods in which small sets of nodes are expanded to communities. Our techniques offer impressive results with regards to all accuracy, memory usage and execution time. Next, we consider the stream of real-time activity of a social networking site and investigate ways of deriving the interesting part out of it based on network properties. More specifically, we use the interactions of the site’s users to construct a network of authorities and assess whether each particular element of activity in the stream is interesting. This approach enables applications in numerous fields to exploit in real-time the enormous amount of information that is made available online everyday without being overwhelmed by the volume of the information. Finally, we investigate yet another field of study in the area of graph mining, namely opinion formation. We adopt a well-studied model and employ a distributed graph processing system to evaluate whether the predicted behavior of the users of a real social network according to this model matches the actual behavior these users.

1.2 Outline

The rest of this thesis is organized as follows:

In Chapter 2 we propose three compressed adjacency list representations that may fit any Pregel-like distributed graph processing system, a compressed representation that adds support for weighted graphs, and a compact tree-based representation that allows for efficient mutations on the graphs’ elements.

Chapter 3 introduces LDLC, a local community detection algorithm that reveals effectively and efficiently the community structure around a given node in a network. Our algorithm is vertex-centric and can easily be executed in a distributed setting.

In Chapter 4 we contribute a second community detection algorithm, termed CoEuS. CoEuS operates on graph streams and offers remarkable improvements in terms of execution time and memory usage when compared to state-of-the-art approaches, while also achieving equivalent accuracy.

Chapter 5 deals with the problem of sampling the interesting part of an activity stream of a social networking site. To this end, we propose a dynamic streaming technique that evaluates the authoritativeness of the network’s users and samples only those elements of the stream whose users appear to have an impact in the network.

Chapter 6 studies the problem of opinion formation in a social context. We model user

interactions based on the game theory framework, implement a distributed graph algorithm for this model, and execute it using a dataset of a popular social network to show that the repeated averaging process results to Nash equilibria which are illustrative of how users really behave.

Finally, Chapter ?? concludes this thesis and discusses possible future directions.

1.3 Credits

The contents of this thesis are based on published work that was conducted in collaboration with others. Specifically:

- Realizing Memory-Optimized Distributed Graph Processing [85]: joint research with Katia Papakonstantinou and Alex Delis.
- Memory-Optimized Distributed Graph Processing through Novel Compression Techniques [84]: joint research with Katia Papakonstantinou and Alex Delis.
- Scalable link community detection: A local dispersion-aware approach [80]: joint research with Alexandros Ntoulas and Alex Delis.
- CoEuS: Community detection via seed-set expansion on graph streams [81]: joint research with Alexandros Ntoulas and Alex Delis.
- Rhea: Adaptively sampling authoritative content from social activity streams [82]: joint research with Alexandros Ntoulas and Alex Delis.
- On the Impact of Social Cost in Opinion Dynamics [83]: joint research with Katia Papakonstantinou.

2. MEMORY-OPTIMIZED DISTRIBUTED GRAPH PROCESSING

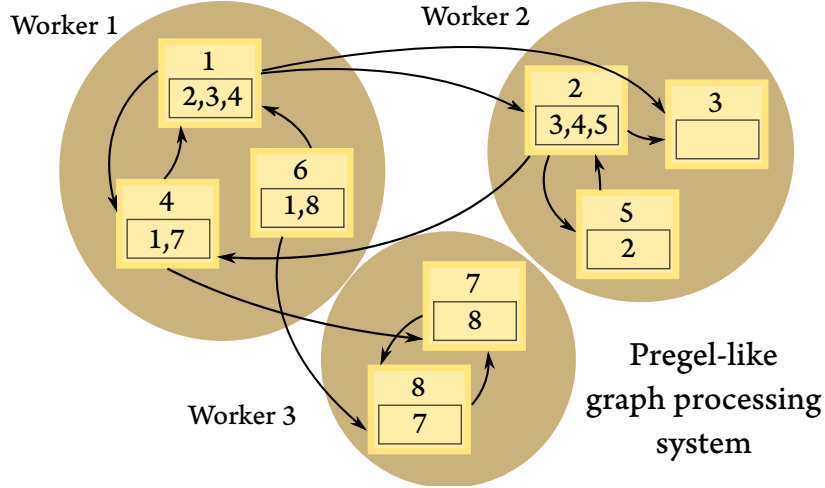


Figure 1: A graph partitioned on a vertex basis in a distributed environment. Each vertex maintains a list of its out-edges.

The proliferation of web applications, the explosive growth of social networks, and the continually-expanding WWW-space have led to systems that routinely handle voluminous data modeled as graphs. *Facebook* has over 1 billion active users [32] and *Google* has long reported that it has indexed unique URLs whose number exceeds 1 trillion [3]. This ever-increasing requirement in terms of graph-vertices has led to the realization of a number of distributed graph-processing approaches and systems [1, 112, 89, 121]. Their key objective is to efficiently handle large-scale graphs using predominantly commodity hardware [60].

Most of these approaches parallelize the execution of algorithms by dividing graphs into partitions [115, 133] and assigning vertices to workers (i.e., machines) following the “*think like a vertex*” programming paradigm introduced with *Pregel* [93]. However, recent studies [60, 28] point out that the so-far proposed frameworks [1, 112, 89, 121] fail to handle the unprecedented scale of real-world graphs as a result of ineffective, if not right out poor, memory usage [60]. Thereby, the space requirements of real-world graphs have become a major memory bottleneck.

Deploying space-efficient graph representations in a vertex-centric distributed environment to attain memory optimization is critical when dealing with web-scale graphs and remains a challenge. Figure 1 illustrates a graph partitioned over three workers. Every vertex is assigned to a *single* node and maintains a list of its out-edges. For example, vertices 1, 4, and 6 are assigned to worker 1, and vertex 1 maintains out-edges towards vertices 2, 3, and 4. This partitioning hardens the task of compression as vertices become unaware of the physical node their neighbors ultimately find themselves in. Related efforts have exclusively focused on providing a compact representation of a graph in a centralized machine environment [20, 10, 31, 26, 87]. In such single-machine settings,

we can exploit the fact that vertices tend to exhibit similarities. However, this is infeasible when graphs are partitioned on a vertex basis, as each vertex must be processed independently of other vertices. Furthermore, to achieve memory optimization, we need representations that allow for mining of the graph’s elements *without decompression*; this decompression would unfortunately necessitate additional memory to accommodate the resulting unencoded representation.

A noteworthy step towards memory optimization was taken by *Facebook* when it adopted Apache Giraph [1] for its graph search service; the move yielded both improved performance and scalability [32]. However, *Facebook*’s improvements regarding memory optimization entirely focused on a more careful implementation for the representation of the out-edges of a vertex [32]; the redundancy due to properties exhibited in real-world graphs was not exploited.

Here, we investigate approaches that help realize compact representations of out-edges in (weighted) graphs of web-scale while following the Pregel paradigm. The vertex placement policy that Pregel-like systems follow necessitates for storing the out-edges of each vertex independently as Figure 1 depicts. This policy preserves the *locality of reference* property, known to be exhibited in real-world graphs [108, 18], and enables us to exploit in this work, patterns that arise among the out-edges of a *single* vertex. We cannot however utilize similarities among out-edges of different vertices, for we are unaware of the partition each vertex is placed into.

Our first technique, termed BVEges, applies all methods proposed in [20] that can effectively function with the vertex placement policy of Pregel in a distributed environment. BVEges primarily focuses on identifying intervals of consecutive out-edges of a vertex and employs universal codings to efficiently represent them. To facilitate access without imposing the significant computing overheads of BVEges, we propose IntervalResidualEdges, which holds the corresponding values of intervals in a non-encoded format. We facilitate support of weighted graphs with the use of a parallel array holding variable-byte encoded weights, termed VariableByteArrayWeights. Additionally, we propose IndexedBitArrayEdges, a novel technique that considers the out-edges of each vertex as a single row in the adjacency matrix of the graph and indexes only the areas holding edges using byte sized bit-arrays. Finally, we propose a fourth space-efficient tree-based data structure termed RedBlackTreeEdges, to improve the trade-off between memory overhead and performance of algorithms requiring mutations of out-edges.

Our experimental results with diverse datasets indicate significant improvements on space-efficiency for all our proposed techniques. We reduce memory requirements up-to 5 times in comparison with currently applied methods. This eases the task of scaling to *billions of vertices per machine* and so, it allows us to load much larger graphs than what has been feasible thus far. In settings where earlier approaches were also capable of executing graph algorithms, we achieve significant performance improvements in terms of time of up-to 41%. We attribute this to our introduced memory optimization as less time is spent for garbage collection. These findings establish our structures as the undisputed preferable option for web graphs, which offer compression-friendly orderings, or any other type of graph after the application of a reordering that favors its compressibility. Last

but not least, we attain a significantly improved trade-off between space-efficiency and performance of algorithms requiring mutations through a representation that uses a tree structure and does not depend on node orderings.

In summary, our contributions in this chapter are that we: I) offer space-efficient representations of the out-edges of vertices and their respective weights, II) allow fast mining (in-situ) of the graph elements without the need of decompression, III) enable the execution of graph algorithms in memory-constrained settings, and IV) ease the task of memory management, thus allowing faster execution.

2.1 Related Work

Our work lies in the intersection of distributed graph processing systems and compressed graph representations. In this regard, we outline here pertinent aspects of these two areas: i) well-established graph processing systems and the challenges they face when it comes to memory management, and ii) state-of-the-art space-conscious representation of real-world graphs.

Google’s proprietary Pregel [93] is a graph processing system that enables scalable batch execution of iterative graph algorithms. As the source code of Pregel is not publicly available, a number of graph processing systems that follow the same data flow paradigm have emerged. Apache Giraph [1] is such an open-source Java implementation with contributions from *Yahoo!* and *Facebook*, that operates on top of HDFS. Our work focuses on Pregel-like systems and extends Giraph’s implementation. Therefore, we provide a short discussion on both Pregel and Giraph in Section 2.2.1. GPS [112] is a similar Java open-source system that introduces an optimization for high-degree vertices: as the degrees of graphs created by human activity are heavy-tail distributed, certain vertices have an “extreme number” of neighbors and stall the synchronization at every iteration. To overcome this deficiency, GPS proposes the large adjacency list partitioning (*LALP*) technique. Pregel+ [121] is implemented in C++ and uses MPI processes as workers to achieve high efficiency. Moreover, Pregel+ features two additional optimizations. The first is the mirroring of vertices, an idea similar to that of *LALP*. The second is a request-respond API which simplifies the process of a vertex requesting attributes from other vertices and merges all requests from a machine to the same vertex into a single request. Unlike the aforementioned distributed graph processing systems that follow Pregel’s BSP execution model, some approaches employ asynchronous execution [89, 54, 59]. GraphLab [89] is such an example that also adopts a shared memory abstraction. PowerGraph [54] is included in GraphLab and mitigates the problem of high-degree vertices by following an edge-centric model bundle. Han and Daudjee [59] extend Giraph with their Barrierless Asynchronous Parallel (BAP) computational model to reduce the frequency of global synchronization barriers and message staleness. GraphX [55] is an embedded graph processing system build on top of the very successful Apache Spark [130] distributed dataflow system. GraphX has received notable attention, partly due to the widespread adoption of the Spark framework. However, a recent comparison [65] against Giraph shows that the latter is able to handle

50x larger graphs than GraphX and is more efficient even on smaller graphs. Our work is orthogonal to these approaches as we introduce compressed adjacency list representations that can be readily applied to all above systems. Several *Facebook* optimizations contributed to Giraph are reported in [32]. Significant improvements are realized through a new representation of out-edges which serializes the edges of every vertex into a byte-array. However, this representation does not entail any memory optimization through compression. MOCGraph [135] is a Giraph extension focused on improving scalability by reducing the memory footprint. This is achieved through the *message online computing* model according to which messages are digested on-the-fly. The MOCGraph approach is also orthogonal to our work, as MOCGraph focuses solely on the memory footprint of messages exchanged, whereas our focus is on representation of the graph. Deca [90] transparently decomposes and groups objects into byte-arrays to significantly reduce memory consumption and achieves impressive execution time speed ups. Our techniques achieve compression over Giraph’s graph representation that already uses byte-arrays and resides in memory for the entire execution of algorithms. However, Deca can offer additional benefits through the optimization of memory consumption with regard to objects other than the graph representation, such as the messages exchanged.

As the size of graphs continues to grow numerous efforts focus on shared-memory or secondary storage architectures. Shun et al. [114] consider compression techniques that can be applied on a shared-memory graph processing system and manage to halve space usage at the cost of slower execution when memory is not a bottleneck. Gemini [136] achieves surprising efficiency by using MPI and performing updates directly on shared-memory graph data, instead of passing messages between cores on the same socket. Our focus is on shared-nothing distributed computing architectures, in which certain techniques of [114] and [136] are inapplicable. GraphChi [73], FlashGraph [134], and Graphene [88] maintain graph data on disks and achieve reasonable performance, having very modest requirements. However, no effort is spent on compressing the graph data. Moreover, our approach does not impose any limitations on the execution time of in-memory distributed graph processing systems, or sacrifice the ease of programming and fault tolerance that go along with the Pregel paradigm.

The increasing number of proposed graph processing systems initiated research concerning their performance. Lu et al. [91] experiment with the number of vertices in a graph and report that GPS and GraphLab run out of memory in settings where Giraph and Pregel+ manage to complete execution. In [28], Cai et al. find that both Giraph and GraphLab face significant memory-related issues. Han et al. [60] carry out a comparative performance study that includes among others, Giraph, GraphLab and GPS. The asynchronous mode of GraphLab is reported to have poor scalability and performance due to the overhead imposed by excessive locking. Moreover, the optimization of GPS for high degree vertices offers little performance benefit. These findings motivated us to use the implementation of Giraph as a basis for this work. [60] notes that Giraph is much improved compared with its initial release, yet, it still demonstrates noteworthy space deficiencies. We note that this is also the case in Giraph’s only subsequent release since, i.e., 1.2, as it does not introduce any additional out-edge representations providing improved space- or time-

efficiency. Therefore, in this chapter we investigate compact representations to further reduce Giraph’s space requirements. Lastly, [60] additionally reports that Giraph’s new adjacency list representation is not suitable for algorithms featuring mutations (i.e., additions and/or deletions). To this effect, we have opted to investigate structures that do not necessarily favor mutations.

The field of graph compression has yielded significant research results after the work presented in [108]. Randall et al. exploit the *locality of reference* as well as the *similarity property* that is unveiled in web graphs when their links are sorted lexicographically. The seminal work on web graph compression is that of Boldi and Vigna [20], who introduce a number of sophisticated techniques as well as a new coding to further reduce the bits per link ratio. Several following efforts [26, 31, 10] managed to present improved results with regard to space but not access time of the graph’s elements. Brisaboa et al. [26] introduce a graph compression approach that uses the adjacency matrix representation of the graph, instead of adjacency lists. A tree structure is used to hold the areas of the adjacency matrix that do actually represent edges. As real-world graphs are sparse, these areas are a very small part of the original matrix. However, there is also a cost in maintaining the in-memory tree structure. In [86, 87], this cost is amortized by representing a specific area around the diagonal of the adjacency matrix without the use of an index and the remaining elements of the graph through an adjacency list representation. All the above approaches focus on providing a compact representation of a graph that can be loaded in the memory of a *single* machine. Hence, the techniques used exploit the presence of all edges in a centralized computing node, which is not suitable for distributed graph processing systems. To the best of our knowledge, our approach is the first to consider compressed graph representations for Pregel-like systems offering distributed execution.

GBASE [66] is the only approach we are aware of that considers compressed graph representations in a distributed environment in general. GBASE uses block compression to efficiently store graphs by splitting the respective adjacency matrices into regions. The latter are compressed using several methods including *Gzip* and *Gap Elias’- γ* encoding. We should note, however, that GBASE does not follow the established by now “think like a vertex” model we have adopted in this work. In addition, GBASE aims at minimizing the storage and I/O cost and its techniques require full decompression of multiple blocks for the extraction of the out-edges of a single vertex. In contrast, we seek to minimize the overall memory requirements and, thus, we cannot apply the techniques used in GBASE; doing so would require at least equivalent amount of memory with non-compressed structures.

A preliminary version of our work appeared in [84]. Here, we propose new representations for weights of out-edges and algorithms requiring mutations. Further, we evaluate our techniques through the execution of additional Pregel algorithms and carry out the entire range of our experimentation using settings that suppress the overhead generated from system logging activity.

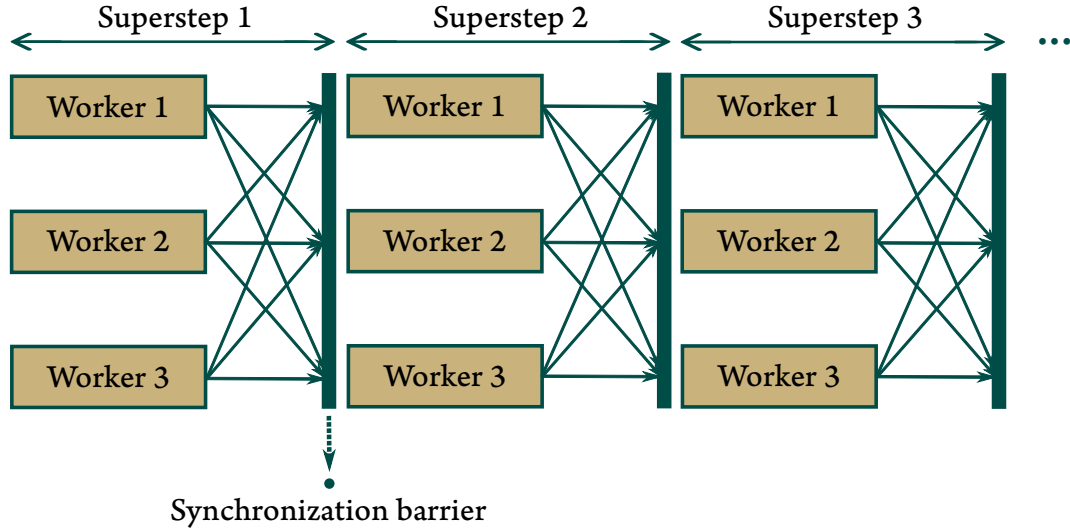


Figure 2: The Pregel programming model: workers compute in parallel the vertices' actions at every *superstep* and messages between iterations are synchronized using a barrier before every *superstep* commences.

2.2 Background

Key Pregel concepts and structures used for representing adjacency lists by the Apache Giraph make up the foundation upon which we develop our proposed techniques. In this section, we outline both Pregel and Giraph, present empirically-observed properties of real-world graphs, and offer definitions for the encodings to be used by our suggested compression techniques.

2.2.1 Pregel

Pregel [93] is a computational model suitable for large scale graph processing, inspired by the *Bulk Synchronous Parallel (BSP)* programming model. Pregel encourages programmers to “*think like a vertex*” by following a vertex-centric approach. The input to a Pregel algorithm is a directed graph whose vertices, along with their respective out-edges, are distributed among the machines of a computing cluster. Pregel algorithms are executed as a sequence of iterations, termed *supersteps*. During a *superstep*, every vertex independently computes a user-defined list of actions and sends messages to other vertices, to be used in the following *superstep*. Therefore, edges serve as communication channels for the transmission of results. A synchronization barrier between *supersteps* ensures that all messages are delivered at the beginning of the next *superstep*. A vertex may vote to halt at any *superstep* and will be reactivated upon receiving a message. The algorithm terminates when all vertices are halted and there are no messages in transit. This programming model is illustrated in Figure 2.

Pregel loads the input graph and performs all associated computations in-memory. The-

reby, Pregel only supports graphs whose edges entirely fit in main-memory. Regarding the management of out-edges, the basic operations provided by Pregel API are the initialization of adjacency lists, the retrieval of the out-edges, and mutations of out-edges, i.e., additions and removals. For example in Figure 1, vertex 1 maintains a list of its neighbors: 2, 3, and 4; Pregel algorithms need to be able to initialize such a list, retrieve its elements, and possibly add or remove elements.

2.2.2 Apache Giraph

The *Apache Software Foundation* has spear-headed the implementation of Giraph [1], an open-source implementation of Pregel that operates atop HDFS and uses *map-only* Hadoop jobs for its computations. The project has been in rapid development since *Facebook* released its own graph search service based on an earlier Giraph release. A key *Facebook* contribution is related to the system’s memory optimization. Giraph used separate Java objects for all data types that needed to be maintained, including the out-edge representation (`HashMapEdges`). A new representation, namely `ByteArrayEdges`, significantly reduced the memory usage as well as the number of objects being maintained by serializing edges as byte arrays instead of instantiating native Java objects. Below, we outline these two widely used Giraph data structures to highlight their difference when it comes to memory usage. We note that Giraph’s configuration allows for specifying the representation of out-edges to be used and maintains an object of the respective class for each vertex of the graph. Extending Giraph with a new out-edge representation is as simple as writing your own class that implements the `OutEdges` interface.

- `ByteArrayEdges`: The default Giraph structure for holding the out-neighbors of a vertex is that of `ByteArrayEdges` [32]. This representation is realized as a byte array, in which target vertex ids and their respective weights are held consecutively, as Figure 3(a) illustrates. The bytes required per out-neighbor are determined by the data type used for its id and weight; for integer numbers $4+4=8$ bytes are required. `ByteArrayEdges` are impractical for algorithms involving mutations as they deserialize all out-edges to perform a removal.
- `HashMapEdges`: An earlier and more “memory-hungry” representation for holding the out-neighbors of a vertex is `HashMapEdges`. This representation is backed by a hash-table that maps target vertex ids to their respective weights as Figure 3(b) illustrates. `HashMapEdges` offer constant time mutations to the adjacency list of a vertex but are very inefficient space-wise. In particular, the memory cost of maintaining out-edges is up to 10 times larger with `HashMapEdges` than it is with `ByteArrayEdges` [32].

2.2.3 Properties of Real-World Graphs

Over the last two decades, studies of real-world graphs have led to the identification of common properties that the graphs in question exhibit [108, 20, 76]. In this context, we

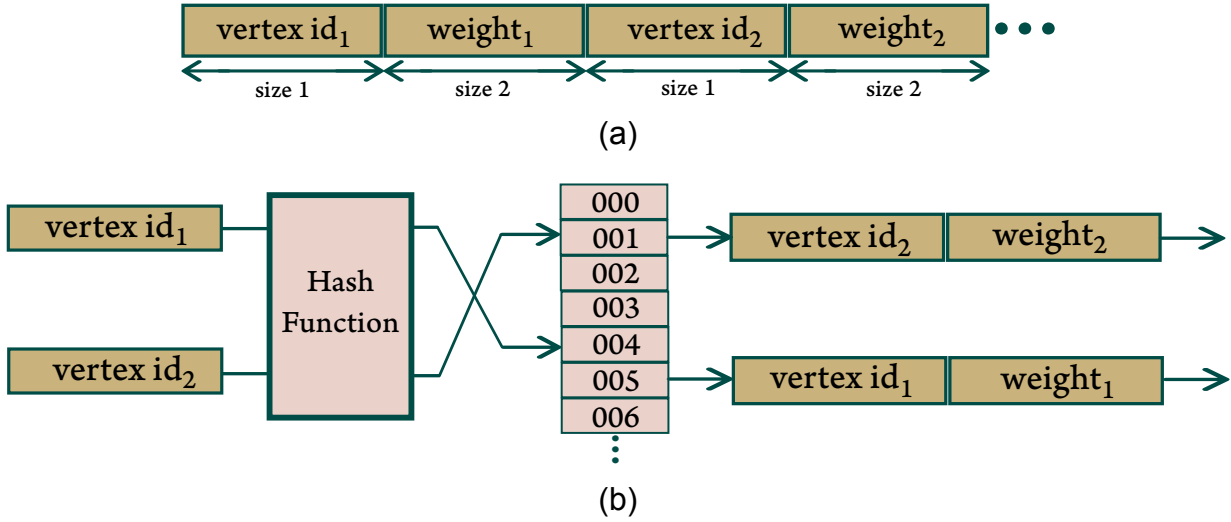


Figure 3: Giraph's adjacency-list representations: ByteArrayEdges (a) and HashMapEdges (b).

list here four empirically-observed properties of real-world graphs that are frequently encountered and allow for effective compression. We begin with two such properties of web graphs that occur when their vertices are ordered lexicographically by URL [108, 20]. We note that even though there is no equivalent way of ordering vertices of other types of graphs, the same properties arise when we apply appropriate reordering algorithms [19, 18, 57]. Then, we list two additional properties observed in realistic graphs from various domains, related to the distribution of node degrees and edge weights. More specifically, the following properties of real-world graphs may be exploited:

- *Locality of reference*: this property states that the majority of the edges of a graph link vertices that are close to each other in the order.
- *Similarity (or copy property)*: vertices that are close to each other in the order tend to have many common out-neighbors.
- *Heavy-tailed distributed degrees*: a constrained number of vertices demonstrate high-degree, whereas the majority of vertices exhibit low-degree. Consequently, graphs created by human activity are generally sparse.
- *Right-skewed weight distributions*: Statistical analysis of weighted graphs shows that the weights of edges are right-skewed distributed [15].

2.2.4 Codings for Graph Compression

In order to compress the data in our structure, we can use various encoding approaches; below, we provide the pertinent definitions of codings we employ in Section 2.3.1.1: *Elias'* γ and ζ codings. We also furnish the definitions of baseline unary and minimal binary coding that help define the first two codings. Let x denote a positive integer, b its binary representation and l the length of b . The aforementioned codings are defined as follows:

- a) *Unary coding*: the unary coding of x consists of $x - 1$ zeros followed by a 1, e.g., the unary coding of 2 is 01.
- b) *Minimal binary coding* over an interval $[21]$: consider the interval $[0, z - 1]$ and let $s = \lceil \log z \rceil$. If $x < 2^s - z$ then x is coded using the x -th binary word of length $s - 1$ (in lexicographical order), otherwise, x is coded using the $(x - z + 2^s)$ -th binary word of length s . As an example, the minimal binary coding of 8 in $[0, 56 - 1]$ is 010000, as $8 = 2^{\lceil \log 56 \rceil} - 56$ and therefore we need the $8 - 56 + 2^6 = 16$ -th binary word of length 6.
- c) *Elias' γ coding*: the γ coding of x consists of l in unary, followed by the last $l - 1$ digits of b , e.g., b of 2 is 10, thus l in unary is 01 and the γ coding of 2 is 010.
- d) *ζ coding* with parameter k [21]: given a fixed positive integer k , if $x \in [2^{hk}, 2^{(h+1)k} - 1]$, its ζ_k -coding consists of $h + 1$ in unary, followed by a minimal binary coding of $x - 2^{hk}$ in the interval $[0, 2^{(h+1)k} - 2^{hk} - 1]$. As an example, 16 is ζ_3 -coded to 01010000, as $16 \in [2^3, 2^6 - 1]$, thus $h = 1$ and the unary of $h + 1 = 2$ is 01, and the minimal binary coding of $16 - 2^3$ over the interval $[0, 2^6 - 2^3 - 1]$ is 010000, as shown above.

In the context of graph compression, Elias' γ coding is preferred for the representation of rather small values of x , whereas ζ coding is more proper for potentially large values. Handling zero is achieved by adding 1 before coding and subtracting 1 after decoding. In the following representations, when no coding is mentioned, the unencoded binary representation is being used.

2.3 Overview of our approach

In this section we describe in detail the space-efficient data structures we suggest for the representation of a vertex's neighbors in a graph. We first propose three compressed out-edge representations that enable the efficient execution of graph algorithms in modest settings. Then, we extend these representations to additionally support weighted graphs, by providing a structure to hold the weights of out-edges. Finally, we propose a compact tree-based out-edge representation that provides significant space and efficiency earnings, favors mutations and offers type-flexibility.

Some centralized graph compression methods, as [20], focus on the compression of the adjacency lists, while others, for example [26], are based on the compact representation of the adjacency matrices. In this work, as we follow a vertex-centric approach, we consider both of these approaches at a vertex level. In particular, we are unable to exploit certain properties that centralized graph compression methods use, such as the *similarity* property, as each vertex in Pregel is unaware of the information present in other vertices. However, we are able to take into account all the other properties described in Section 2.2.3.

2.3.1 Representations based on consecutive out-edges

A common property of graphs created by human activity is *locality of reference*: Vertices, adhering to the orderings mentioned in Section 2.2.3, tend to neighbor with vertices of similar ids. This property is evident through the adjacency lists of the graphs of our dataset, all of which tend to have a lot of neighbors with *consecutive* ids.

We can exploit this property by applying a technique similar to the one introduced in [20]. In particular, [20] distinguishes between the neighbors whose ids form some *interval* of consecutive ids, and the rest. To reconstruct all the edges of the *intervals*, only the leftmost neighbor id and the length of the *interval* need to be kept. This information is further compressed using gap *Elias' γ coding*. The remaining out-edges, termed *residuals*, are compressed using *ζ coding*.

We build on these ideas and introduce two compressed representations that exploit *locality of reference* in a similar fashion but are applicable to Prege1-like systems. We consider that neighbors are sorted according to their id, as the case is in [20]. We also note that both of our structures based on consecutive out-edges do not favor mutations, as any addition or removal of an edge would require a complete reconstruction of the compact representation to discover the new set of intervals and residuals.

2.3.1.1 BVEEdges

Our first representation, namely BVEEdges, focuses solely on compressing the neighbors of a vertex, at the cost of computing overheads. Therefore, we simply adjust the method of Boldi and Vigna [20] to the restrictions imposed by Prege1. In particular, we use the ideas of distinguishing *intervals* and *residuals*, as well as applying appropriate codings on them. The compressed data structure discussed in [20] considers the whole graph and exploits the current vertex's id during compression. However, the vertex id is not available in the level where adjacency lists are kept in the Prege1 model. To overcome this issue, we use the first neighbor id we store in our structure as a reference to proceed with gap encoding. As the case is with [20], we use *Elias' γ coding* for *intervals*, and *ζ coding* for *residuals*. *Elias' γ coding* is most preferable for *intervals* of at least 4 elements [20]; shorter *intervals* are more compactly stored as *residuals*. We note here that [20] uses *copy lists* to exploit the *similarity property*. However, using *copy lists* in a vertex-centric distributed environment is infeasible.

Definition 1 (BVEEdges) *Given a list l of a node's neighbors, BVEEdges is a sequence of bits holding consecutively: the γ -coded number of intervals in l of length at least 4; for the first such interval, the smallest neighbor id in it and the γ -coded difference of the interval length minus 4; for each of the rest of the intervals, the difference of the smallest neighbor id in it minus the largest neighbor id of the previous interval decreased by one; a ζ coding for each of the remaining neighbors, its argument being either the difference x between the current node's id and the previous node id which was encoded to be stored in the sequence minus 1, or, in case $x < 0$, the quantity $2|x| - 1$.*

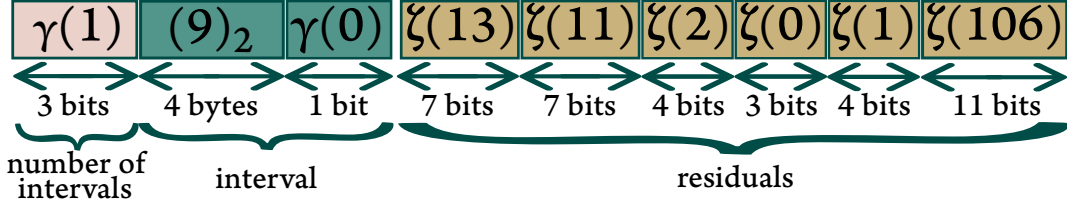


Figure 4: The storage of neighbors in *BVEges*, detailed in Example 1. $\gamma(x)$ and $\zeta(x)$ denote the γ and ζ encodings of x respectively.

Example 1 Consider the following sequence of neighbors to be represented: (2, 9, 10, 11, 12, 14, 17, 18, 20, 127). We employ *BVEges* as illustrated in Figure 4. Here, there is only one interval of length at least equal to 4: [9..12]. We first store the number of intervals using γ coding. Then, we store the leftmost id of the interval, i.e., 9, using its unencoded binary representation. We proceed with storing a representation of the length of the interval to enable the recovery of the remaining elements. In particular, we store the γ coding of the difference of the interval length minus the minimum interval length, which is $4 - 4 = 0$ in our case. Then, we append a representation for the residual neighbors. For each residual, we store the ζ coding of the difference of its id with the id of the last node stored, minus 1 (as each id appears at most once in the neighbors' list). The residual id 2 is smaller than the smallest id of the first interval, so we store the residual neighbor 2 as $\zeta(13)$, since $2|2 - 9| - 1 = 13$, and the residual 14 as $\zeta(11)$, since $14 - 2 - 1 = 11$. Similarly, we store 17, 18, 20 and 127 as $\zeta(2)$, $\zeta(0)$, $\zeta(1)$ and $\zeta(106)$, respectively.

The respective values computed in each step are written using a bit stream. This, combined with the fact that values have to be encoded, renders the operation costly. We also investigated the idea of treating all neighbors as *residuals* to examine if the re-construction of *intervals* was more expensive. However, we experimentally found that the resulting larger bit stream offered worse access time.

Accessing the out-edges of a vertex requires the following procedure: first, we read the number of *intervals*. For the first interval, we read the id of the smallest neighbor in it and decode its length. For each of the rest of the intervals, we construct the smallest neighbor id by adding to the next γ -coded value the largest neighbor id of the previous interval incremented by one, and decode its length. After we process the specified number of *intervals*, we decode the *residuals* one by one.

2.3.1.2 IntervalResidualEdges

Our second compressed out-edge representation, namely *IntervalResidualEdges*, also incorporates the idea of using *intervals* and *residuals*. However, we propose a different structure to avoid costly bit stream I/O operations. In particular, we keep the value of the leftmost id of an *interval* unencoded, along with a byte that is able to index up to 256 consecutive neighbors. *Residuals* are then also kept unencoded. Clearly, any consecutive neighbors of length at least equal to 2 are represented more efficiently using an *interval*

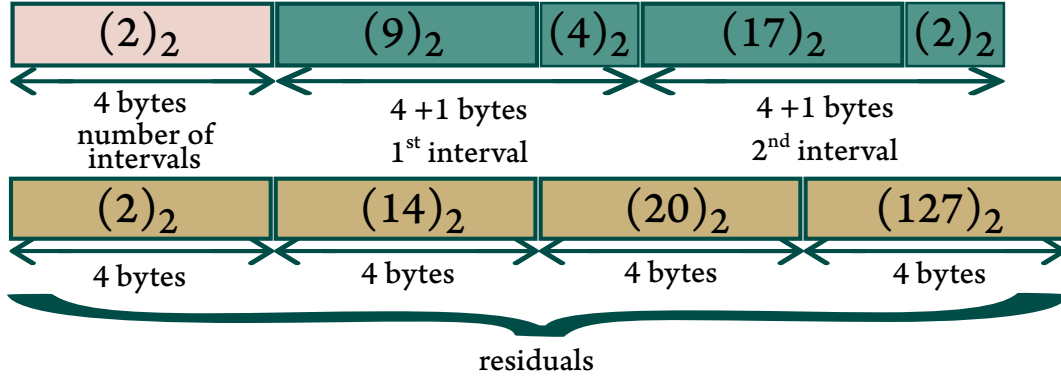


Figure 5: The storage of neighbors in *IntervalResidualEdges*, detailed in Example 2. $(x)_2$ is the binary representation of x .

rather than two or more *residuals*. Therefore, we set the minimum interval length with *IntervalResidualEdges* equal to 2. Due to the *locality of reference* property, this dedicated byte of each *interval* allows us to compress the adjacency list significantly, while also avoiding the use of expensive encodings and bit streams.

Definition 2 (*IntervalResidualEdges*) Given a list l of a node's neighbors, *IntervalResidualEdges* is a sequence of bytes holding consecutively: the number of intervals in l ; the smallest neighbor id and the length of each such interval; the id of each of the remaining neighbors.

Example 2 The representation of the aforementioned sequence of neighbors (2, 9, 10, 11, 12, 14, 17, 18, 20, 127) using *IntervalResidualEdges* is illustrated in Figure 5. In this case there are two intervals of at least 2 consecutive neighbors, namely [9..12] and [17, 18]. We first store the number of intervals, and then use one 5-byte element for each interval, consisting of a 4-byte representation of the smallest neighbor id in it (i.e., 9 and 17), plus a byte holding the number of neighbors in this interval (4 and 2 respectively). Finally we append a 4-byte representation for each residual neighbor.

This representation delivers its elements through the following procedure: we first read the number of *intervals*; while there are still unread *intervals*, we read 5-bytes, i.e., the leftmost element of the *interval* and its length, and recover one by one the elements of the *interval*. When all out-edges that are grouped into *intervals* are retrieved, we read in the *residuals* directly as integers.

2.3.2 IndexedBitArrayEdges

Our first two representations are based on the presence of *consecutivity* among the neighbors of a vertex. Here we propose a representation termed *IndexedBitArrayEdges*, that takes advantage of the *concentration* of edges in *specific areas* of the adjacency matrix, regardless of whether these edges are in fact consecutive. With *IndexedBitArrayEdges*

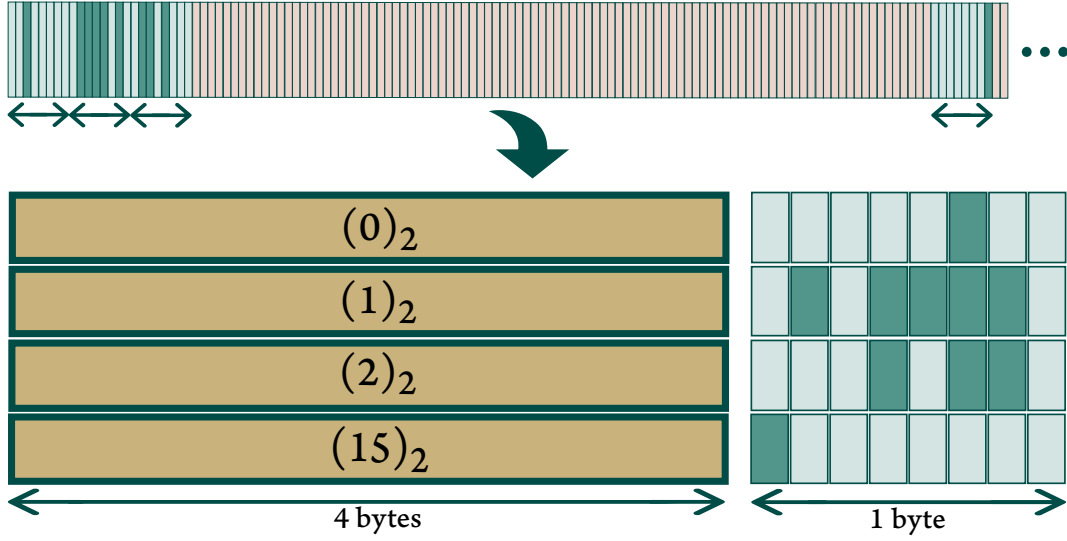


Figure 6: A bit-array representation of an adjacency list and the storage of these neighbors in *IndexedBitArrayEdges*, detailed in Example 3. $(x)_2$ denotes the binary representation of x .

we use a single byte to depict eight possible out-neighbors. Using a byte array, we construct a data structure of 5-byte elements, one for each interval of neighbor ids having the same quotient by 8. The first 4 bytes of each element represent the quotient, while the last one serves as a set of 8 flags indicating whether each possible edge in this interval really exists. As the neighbor ids of each node tend to concentrate within a few areas, the number of intervals we need to represent is small and the compression achieved is exceptional.

Definition 3 (IndexedBitArrayEdges) Given a bit-array r representing a list of a node's neighbors, *IndexedBitArrayEdges* is a sequence of 5-byte elements, each one holding an octet of r that contains at least one 1: the first 4 bytes hold the distance in r of the first bit of the octet from the beginning of r ; the last one holds the octet.

Example 3 The representation of the aforementioned sequence of neighbors (2, 9, 10, 11, 12, 14, 17, 18, 20, 127) using *IndexedBitArrayEdges* is illustrated in Figure 6. In the top part we see the bit-array r representation of this adjacency list. The quotient and remainder of each node id divided by 8 give us the approximate position (octet) and the exact position of the node in r , respectively; hence, as depicted in the bottom part of Figure 6, the neighbors are grouped in four sets: $\{2\}$, $\{9, 10, 11, 12, 14\}$, $\{17, 18, 20\}$, $\{127\}$. All ids in each set share the same quotient when divided by 8, which will be referred as index number henceforth. For instance, the index number of the third set is 2, and is stored in the first part of the third element, denoted by $(2)_2$. Moreover, the remainders of the ids 17, 18 and 20 divided by 8 are 1, 2, and 4 respectively, and so the 2nd, 3rd and 5th flags from the right side of the same element are set to 1 to depict these neighbors.

Accessing the out-edges of a vertex is performed as follows: First, we read a 5-byte element. Then, we recover out-edges from the flags of its last byte and reconstruct the

neighbor ids using the first 4 bytes. After we examine all flags of the last byte, we proceed by reading the next 5-byte element and repeat until we retrieve all out-edges.

We note that `IndexedBitArrayEdges` is able to represent graphs that are up to 8 times larger than the maximum size achieved with `ByteArrayEdges` and 32-bit integers. Hence, we expect its space-efficiency against `ByteArrayEdges` will be even more evident when dealing with a graph of this size. In addition, this representation is clearly more suitable for supporting mutations as opposed to our other two suggested techniques. The addition of an edge in the graph requires us to search linearly the 5-byte elements to ascertain whether we have already indexed the corresponding byte. If that is the case, we merely have to change a single flag in that byte. Otherwise, we have to append a 5-byte element at the end of the structure with the new index number (4-bytes) plus one byte with one –specific– flag set to 1. Obviously, `IndexedBitArrayEdges` does not require that the out-edges are sorted by their id, an assumption that our two other compressed representations make. To remove an edge from the structure, we again have to search for the element with the corresponding index, and set a specific flag to 0. In the case of ending up with a completely empty byte, removing the 5-byte element would be costly. However, this cost is imposed only when elements are left completely empty. Hence, removals are more efficient than with `ByteArrayEdges`, in which the cost is imposed for every out-edge removal. Moreover, there is no inconsistency in keeping the element in our representation, only some memory loss which can be addressed via marking elements when they empty so that they be used in a subsequent neighbor addition.

2.3.3 VariableByteArrayWeights

Our proposed `BVEEdges` and `IntervalResidualEdges` consider ordered lists of neighbors. Thus, they can be easily modified to support weighted graphs through the use of an additional array, holding the respective weights of the neighbors. This array could simply adapt the format of `ByteArrayEdges` and maintain only the weight of each neighbor in its uncompressed binary format. However, statistical analysis of weighted graphs has shown that the weights of edges exhibit right-skewed distributions [15, 96]. Therefore, there is strong potential for memory optimization in using a compressed weight representation.

Variable-byte coding [120] uses a sequence of bytes to provide a compressed representation of integers. In particular, when compressing an integer n , the seven least significant bits of each byte are used to code n , whereas the most significant bit of each byte is set to 0 in the last byte of the sequence and to 1 if further bytes follow. Hence, variable-byte coding uses $\lfloor \log_{128}(n) \rfloor + 1$ bytes to represent an integer n . The advantage of this approach over the more compact bitwise compression schemes, such as Golomb-Rice, is the significantly faster decompression time it offers due to byte-alignment. In particular, Scholer et al. [113] show that when using the variable-byte coding scheme, queries are executed twice as fast as with bitwise codes, at a small loss of compression efficiency.

Definition 4 (VariableByteArrayWeights) *Given a list l of a node's edge weights sorted according to the id of their respective neighbor, `VariableByteArrayWeights` is a sequence*

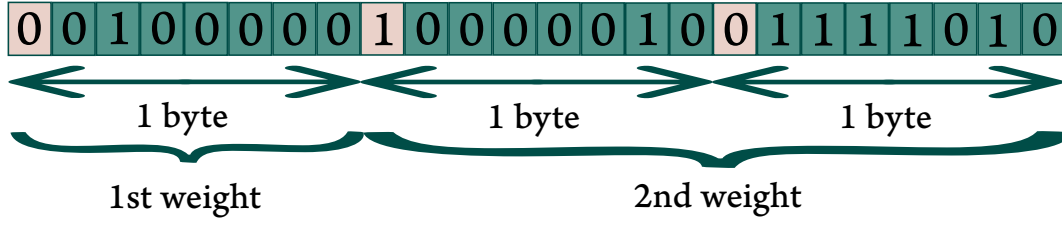


Figure 7: The storage of edge weights using `VariableByteArrayWeights`. Weights of neighbors are held in variable-byte encoding. Two weights (32 and 378) are represented using only one and two bytes, respectively. `VariableByteArrayWeights` can extend `BVEEdges` and `IntervalResidualEdges` to support weighted graphs.

of bytes holding consecutively the weights in variable-byte coding.

Example 4 Consider the following sequence of edge weights to be represented: (32, 378). Figure 7 provides an illustration of the parallel array using variable-byte coding that we extend `BVEEdges` and `IntervalResidualEdges` with, to support weighted graphs. The weight of the first neighbor is represented using only one byte, and thus the most significant bit of the latter is set to 0. In contrast, the second weight requires two bytes, the first of which has its most significant bit set to 1, to signify that the following byte is also part of the same weight.

Extracting the weight of a neighbor is as simple as reading a sequence of bytes until reaching one with the most significant bit set to 0, and using the 7 least significant bits of each byte in the sequence to decode the weight.

2.3.4 RedBlackTreeEdges

Compressed representations essentially limit the efficiency of performing mutations. Even the non-compressed `ByteArrayEdges` representation is impractical when executing algorithms involving mutations of edges [60]. This is due to the excessive time required to perform a removal, as all out-edges need to be deserialized. However, we can achieve mutation efficiency without the overwhelming memory overhead induced when using `HashMapEdges`. Java’s `HashMap` objects use a configurable number of buckets in their hash-table, which doubles once their entries exceed a percentage of their current capacity. Giraph sets the initial capacity of each `HashMap` to be equal to the number of out-edges of the corresponding adjacency list, thus ending up with significantly more buckets than what is needed at initialization. Furthermore, the iterator of out-edges for this representation requires additional $O(n)$ space.

The space wasted when using a `HashMap` due to empty hash-table buckets and additional memory requirements for iterating elements motivated us to implement a *red-black tree*-based representation¹ offering the same type flexibility provided by `HashMapEdges`. Even

¹Java’s `TreeMap` uses an unnecessary parent reference.

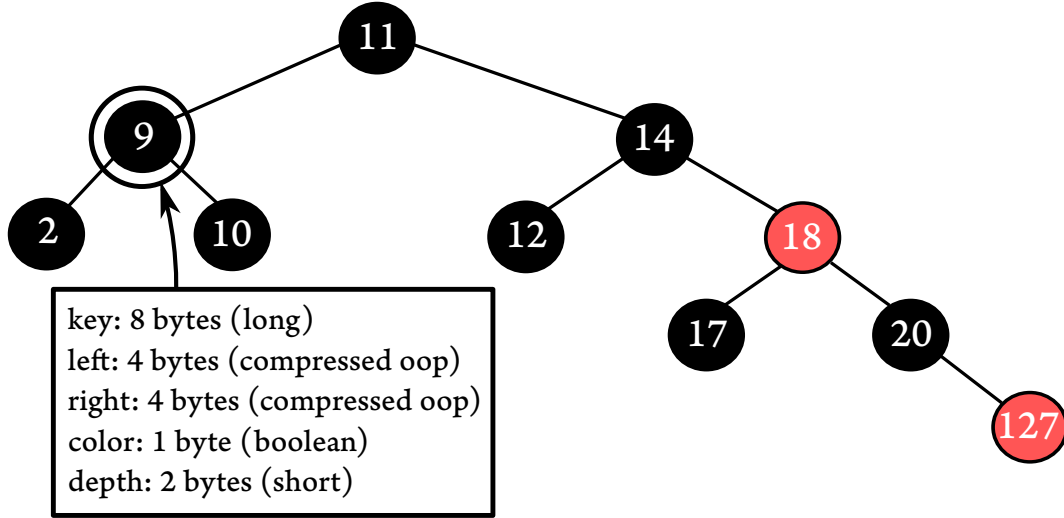


Figure 8: The storage of neighbors in `RedBlackTreeEdges`. Neighbors' ids are inserted as keys to a red-black tree. For weighted graphs each node would additionally maintain a variable to hold the weight.

though the individual entries of the tree require more space, we noticed that the total memory used is reduced by more than 15% for graphs of our dataset, as our tree does not waste space for empty buckets. These savings can be significantly enhanced through the use of primitive data types. Moreover, using Morris' tree traversal algorithm [99], we can iterate through the out-edges without additional cost in space. Based on these observations, we developed `RedBlackTreeEdges`, a space-efficient representation that favors mutations and offers type-flexibility.

Definition 5 (`RedBlackTreeEdges`) Given a list l of a node's neighbors, each one potentially associated with an edge weight, *RedBlackTreeEdges* is a red-black tree which uses the id of a neighbor as a key. The nodes of the tree comprise two references to their left and right child, a boolean for the color of the node, a short for its depth, and two variables using primitive data types for the id and the weight. The memory requirements of the key and the weight depend on the id's respective primitive data type. The references on the left and right children require 4 bytes each—when the maximum heap size for each worker is less than 32GB and thus compressed ordinary object pointers (oops) can be used—or 8 bytes each otherwise.

Example 5 The representation of the aforementioned sequence of neighbors (2, 9, 10, 11, 12, 14, 17, 18, 20, 127) using *RedBlackTreeEdges* is illustrated in Figure 8. We observe that the neighbors' ids are inserted as keys to a red-black tree. For each id, a tree node is created and holds the id as a key, references to the left and right child of the node, the color of the node, and the depth of the node. In case of a weighted graph, each node additionally maintains a variable to hold the weight. In this example, we consider that keys are long integers, and thus require 8 bytes. In addition compressed oops can be used, so the references to the left and right child need 4 bytes each. The graph is unweighted so

no bytes are required for the weights, and finally, for the color and the depth of the node, 1 and 2 bytes are needed, respectively, as is always the case.

The use of a *red-black tree* instead of a *hash-table* allows us to access the neighbors without inducing further costs space-wise, and to avoid resizing as neighbors are added or removed. This leads to significantly less memory requirements than with `HashMapEdges`, without forgoing the efficiency of performing mutations. The use of primitive data types instead of generic types necessitates defining suitable Java classes for the input graph; however, this is insignificant when compared to our space earnings. Instead of reducing the total memory requirements 15 percentage points we are able to achieve significantly higher savings, as we will show in our experimental evaluation. Besides, the majority of publicly available graphs use *integer* ids, and *Facebook* uses *long integers*, which are represented at the same cost with `RedBlackTreeEdges`, due to JVM alignment. We also note that the ordering of the vertices' labels does not impact the performance of this representation which is applicable to graphs with in-situ node labelings.

2.4 Experimental Evaluation

We implemented our techniques using Java and compared their performance against Giraph's out-edge representations using a number of publicly available and well-studied web and social network graphs [20, 18], reaching up to 2 billion edges. Our implementation is available online.² We first present the dataset and detail the specifications of the machines used in our experiments. Then, we proceed with the evaluation of our out-edge representations by answering the following questions:

- How much more space-efficient is each of our three compressed out-edge representations compared to Giraph's default representation?
- Are our techniques competitive speed-wise when memory is not a concern?
- How much more efficient are our compressed representations when the available memory is constrained?
- Can we execute algorithms for large graphs in settings where it was not possible before?
- Is our compressed weight representation able to induce additional gains?
- What are the benefits of using our tree-based out-edge representation instead of Giraph's fastest representation for algorithms involving mutations?

²<https://goo.gl/hJlG8H>

graph	vertices	edges	type
<i>uk-2007-05@100000</i>	100,000	3,050,615	web
<i>uk-2007-05@1000000</i>	1,000,000	41,247,159	web
<i>ljournal-2008</i>	5,363,260	79,023,142	social
<i>indochina-2004</i>	7,414,866	194,109,311	web
<i>hollywood-2011</i>	2,180,759	228,985,632	social
<i>uk-2002</i>	18,520,486	298,113,762	web
<i>arabic-2005</i>	22,744,080	639,999,458	web
<i>uk-2005</i>	39,459,925	936,364,282	web
<i>twitter-2010</i>	41,652,230	1,468,365,182	social
<i>sk-2005</i>	50,636,154	1,949,412,601	web

Table 1: Dataset of our experimental setting with a total of ten publicly available web and social network graphs [20, 18].

2.4.1 Experimental Setting

Our dataset consists of 10 web and social network graphs of different sizes. The properties of these graphs are detailed in Table 6. We ran our experiments on a Dell PowerEdge R630 server with an Intel®Xeon® E5-2630 v3, 2.40 GHz with 8 cores, 16 hardware threads and a total of 128GB of RAM. Our cluster comprises eight virtual machines running Xubuntu 14.04.02 with Linux kernel 3.16.0-30-generic and 13GB of virtual RAM. On this cluster we set up Apache Hadoop 1.0.2 with 1 master and 8 slave nodes and a maximum per machine JVM heap size of 10GB. Lastly, we used Giraph 1.1.0 release.

2.4.2 Space Efficiency Comparison

We present here our results regarding space efficiency for the web and social network graphs of our dataset. We compare our methods involving compression with the one discussed in [32], viz. `ByteArrayEdges`, which is currently the default Giraph representation for out-edges. To measure the memory usage we loaded each graph using a fixed capacity Java array list to hold the adjacency lists, dumped the heap of the JVM and used the Eclipse Memory Analyzer³ to retrieve the total occupied memory.

Table 2 lists the memory required by the four representations examined here and the representation of [20] in MB. We observe that all our proposed compression techniques have significantly reduced memory requirements compared to `ByteArrayEdges`. As was expected, `BVE` edges, which essentially also serves as a yardstick to measure the performance of our structures that focus on access-efficiency, outperforms all representations as far as space-efficiency is concerned. In particular, depending on the graph, its memory requirements are always less than 40% of the requirements of `ByteArrayEdges`, and reach much smaller figures in certain cases, e.g., 20.08% for *hollywood-2011*. However,

³<https://eclipse.org/mat/>

graph	Byte- ArrayEdges	BVEEdges (BV)	IntervalRe- sidualEdges	IndexedBit- ArrayEdges
<i>uk-2007-05@100000</i>	22.61 MB	6.41 MB (0.96 MB)	7.92 MB	8.91 MB
<i>uk-2007-05@1000000</i>	279.16 MB	67.36 MB (10.54 MB)	82.7 MB	97.79 MB
<i>ljournal-2008</i>	866.36 MB	386.73 MB (117.68 MB)	497.52 MB	648.52 MB
<i>indochina-2004</i>	1,511.67 MB	442.34 MB (48.03 MB)	646.03 MB	554.23 MB
<i>hollywood-2011</i>	1,381.91 MB	287.53 MB (145.85 MB)	613.52 MB	676.88 MB
<i>uk-2002</i>	2,733.6 MB	1,092.82 MB (116.39 MB)	1,224.07 MB	1,255.67 MB
<i>arabic-2005</i>	4,820.09 MB	1,428.97 MB (187.58 MB)	1,674.75 MB	1,849.83 MB
<i>uk-2005</i>	7,401.88 MB	2,383.54 MB (279.45 MB)	2,728.74 MB	2,928.81 MB
<i>twitter-2010</i>	11,189.88 MB	4,628.48 MB (2,600.07 MB)	7,127.76 MB	8,888.50 MB
<i>sk-2005</i>	14,829.64 MB	4,889.85 MB (607.92 MB)	5,657.79 MB	6,354.17 MB

Table 2: Memory requirements of Giraph’s ByteArrayEdges and our three out-edge representations for the small and large-scale graphs of our dataset. Requirements of BV [20] in a centralized setting are also listed to provide an indication of the compressibility potential of each graph.

we observe that our novel IntervalResidualEdges as well as the less restrictive IndexedBitArrayEdges, both of which do not impose any computing overheads, also manage to achieve impressive space-efficiency.

2.4.3 Execution Time Comparison

In this section, we present results regarding the execution times of Pregel algorithms using our compressed out-edge representations. Reported timings for all our results are averages of multiple executions.

2.4.3.1 PageRank Computation

PageRank is a popular algorithm employed by many applications that run on top of real world-networks, with (web page/social network users) *ranking* and *fake account detection* being typical examples.

A Pregel implementation of PageRank is shown in Function computePageRank. In our experimental setting MAX_SUPERSTEPS is set to 30 and α is set to 0.85. Every vertex executes the function computePageRank at each *superstep*. The graph is initialized so that in *superstep* 0 all vertices have value equal to $\frac{1}{|V|}$. In each of the first 30 (i.e., 0 to 29) *supersteps*, each vertex sends along each out-edge its current PageRank value divided by the number of out-edges (line 9). From *superstep* 1 and on, each vertex computes its PageRank value v_{vertex} as shown in line 6. When *superstep* 30 is reached, no further messages are sent, each vertex votes to halt, and the algorithm terminates.

We expect that any Pregel algorithm not involving mutations would exhibit similar behavior

Function: computePageRank(vertex, messages)

```

1 begin
2   if superstep ≥ 1 then
3     sum ← 0;
4     foreach message ∈ messages do
5       sum ← sum + message;
6      $v_{\text{vertex}} \leftarrow \frac{1-\alpha}{|V|} + \alpha \times \text{sum}$ ;
7   if superstep < MAX_SUPERSTEPS then
8      $d_{\text{vertex}} \leftarrow \text{degree}(\text{vertex})$ ;
9     sendMessageToAllOutEdges( $\frac{v_{\text{vertex}}}{d_{\text{vertex}}}$ );
10  else
11    voteToHalt();

```

Function: computeShortestPaths(vertex, messages)

```

1 begin
2   if superstep == 0 then
3     vertex.setValue(∞);
4   minDist ← isSource(vertex) ? 0 : ∞;
5   for message in messages do
6     minDist ← min(minDist, message);
7   if minDist < vertex.getValue() then
8     vertex.setValue(minDist);
9     for edge in vertex.getEdges() do
10      sendMessage(edge, minDist + edge.getValue());
11  voteToHalt();

```

for the different representations with the one reported here for PageRank, as it would also feature the same set of actions regarding out-edges, i.e., initialization and retrieval.

2.4.3.2 Shortest Paths Computation

Single-source Shortest Paths algorithms [30] focus on finding a shortest path between a single source vertex and every other vertex in the graph, a problem arising in numerous applications.

A Pregel implementation of Shortest Paths is shown in Function computeShortestPaths. Initially, the value associated with each vertex is initialized to infinity, or a constant larger than any feasible distance in the graph from the source vertex. Then, using the temporary variable *minDist* the function examines cases that may update this value. There are two such cases: *i*) if the vertex is the source vertex the distance is set to **zero**, and *ii*) if the vertex receives a message with a smaller value than the one it currently holds, the distance is updated accordingly. When a vertex updates its value it must also send a message to

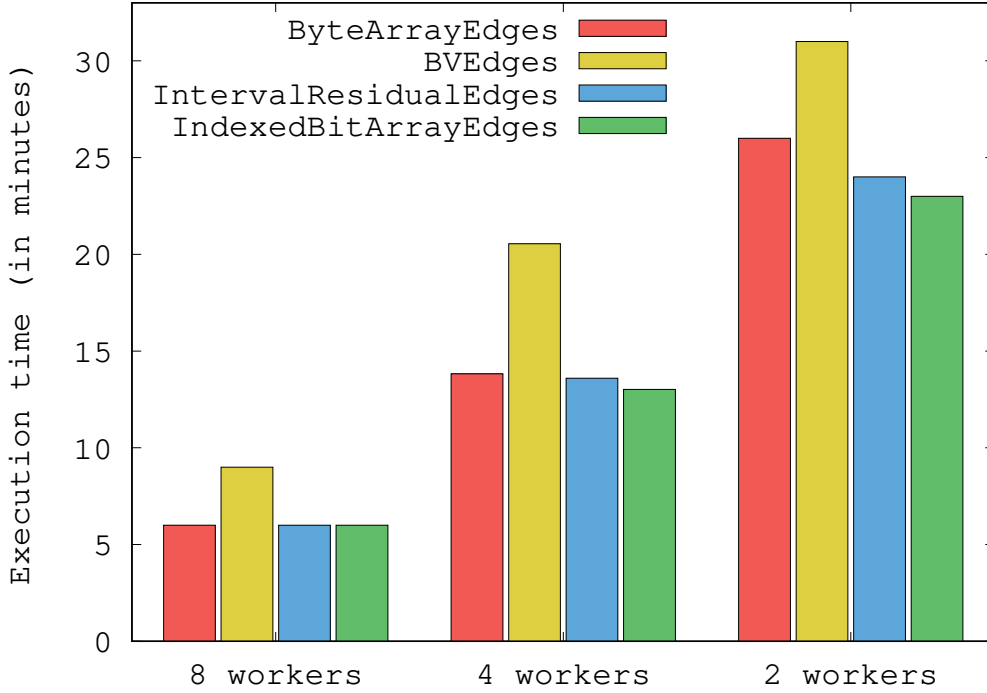


Figure 9: Execution time (in minutes) of PageRank algorithm for *indochina-2004* using a setup of 2, 4, and 8 workers.

all its out-neighbors to notify them about the newly found path. Each message is set to the updated distance of the vertex plus the weight of the edge that connects the vertex with the respective neighbor. Finally, the vertex votes to halt and remains halted until a message reaches it. The algorithm terminates when all vertices are halted, at which time each vertex holds the value of the shortest path to the source vertex.

The Shortest Paths algorithm involves the same operations as the PageRank algorithm, but additionally serves the purpose of evaluating our techniques on weighted graphs.

2.4.3.3 Comparison using small-scale graphs

We begin our access time comparison by investigating the performance of our three compressed out-edge representations, as well as that of Giraph’s `ByteArrayEdges`. Figure 9 depicts the results of all four techniques when executing the PageRank algorithm for the graph *indochina-2004*. We run experiments on setups of 2, 4, and 8 workers and present the results of the total time needed for each representation.

We observe that `IndexedBitArrayEdges` and `IntervalResidualEdges` do not impose any latency in the process. In fact, using either of our two novel representations we achieve execution times for all three setups that are better than those of `ByteArrayEdges`. The performance gain becomes more notable as we limit the number of available workers. `BVEEdges` is inferior speed-wise due to the computationally expensive access of the out-

edges offered through this structure which requires decoding *Elias- γ* and *ζ -coding* values. This indicates that the computing overheads imposed by the *state-of-the-art* techniques of [20] are not negligible and simply adopting them proves to be inefficient. We note that this graph is fairly small for all our setups and its memory requirements are not a bottleneck for any of the representations we examine. However, the messages that are exchanged during the execution of the algorithm need in total more than 65GB of memory. Thus, garbage collection needs to take place in the setups of 2 and 4 workers.

For graphs which are equivalent to or smaller than *indochina-2004* the performance is similar. In particular, for all three setups *IndexedBitArrayEdges* and *IntervalResidualEdges* managed to execute the PageRank algorithm faster than *ByteArrayEdges* was able to. On the contrary, *BVEEdges* required more time for each *superstep*.

2.4.3.4 Comparison using large-scale graphs

We further examine the performance of our representations using setups where memory does not suffice for the needs of the execution of PageRank. This forces the JVM to work too hard and results in wasting a significant proportion of the total processing time performing garbage collection. Hence, the overall performance degrades extremely. In particular, we examine the behavior of all four representations for the graph *uk-2005*, using a setup of 5 workers, i.e., the smallest possible setup that can handle the execution of PageRank using *ByteArrayEdges*.

The merits of memory optimization in the execution of *Pregel* algorithms for large scale graphs are evident in Figure 10. In particular, Figure 10 depicts the time needed for each *superstep* of the execution of PageRank for the *uk-2005* graph with each one of the four space-efficient out-edge representations. We observe that *BVEEdges* requires significantly more time than our other two representations for every *superstep*, as was the case with small-scale graphs. In particular, using *BVEEdges* most *supersteps* require more than 3 minutes each, whereas using our other two representations most *supersteps* need about 2.5 minutes each. We also see, however, that in this setup the execution with *Byte-ArrayEdges* tends to fluctuate in performance, and consequently performs worse than our slowest structure, i.e., *BVEEdges*. The increased memory requirements of Giraph's default implementation, result in an unstable pace during the execution of PageRank, as it needs to perform garbage collection very frequently to accommodate the memory objects required in every *superstep*. *IndexedBitArrayEdges* and *IntervalResidualEdges* were able to handle every *superstep* at a steady pace and greatly outperformed *ByteArrayEdges*, requiring 2.45 and 2.46 minutes of execution per *superstep*, respectively, when in fact *ByteArrayEdges* needed 4.03. Our most compact structure, i.e., *BVEEdges* required 3.13 minutes per *superstep* to run the PageRank algorithm, which is also significantly faster than Giraph's default representation.

The performance difference of the four representations with regard to the total execution time of PageRank for the graph *uk-2005* is even more evident in Figure 11. The executions using *IndexedBitArrayEdges* and *IntervalResidualEdges* are faster by 40.63% and

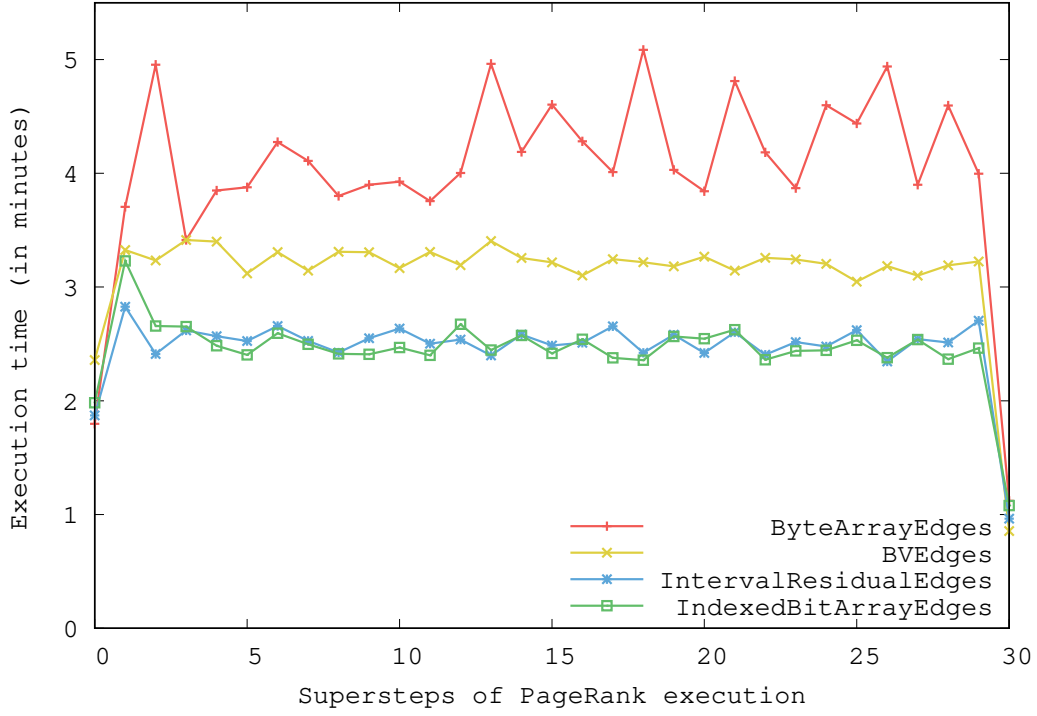


Figure 10: Execution time (in minutes) for each *superstep* of the PageRank algorithm for the graph *uk-2005* using 5 workers. ByteArrayEdges performance fluctuates due to extensive garbage collection.

40.01% than the one with ByteArrayEdges, respectively.

We further evaluate the performance of the four representations by executing PageRank for the same graph using only 4 workers. As already mentioned, the execution with ByteArrayEdges on this setup fails as the garbage collection overhead limit is exceeded, i.e., more than 98% of the total time is spent doing garbage collection. Our proposed implementations, however, are able to execute PageRank for the *uk-2005* graph despite the limited resources. The total time needed by our three representations is also illustrated in Figure 11. We observe that under these settings IndexedBitArrayEdges, IntervalResidualEdges, and BVEges need 212.65, 221.27, and 230.47 minutes, respectively. As we can see in Table 2, IndexedBitArrayEdges requires more memory than IntervalResidualEdges to represent the out-edges of *uk-2005*. However, the retrieval of out-edges using IndexedBitArrayEdges is more memory-efficient than using IntervalResidualEdges, which results in it being 4% faster under these settings.

We note that for the *uk-2005* graph, PageRank execution requires the exchange of messages that surpass 313GB of memory in total.

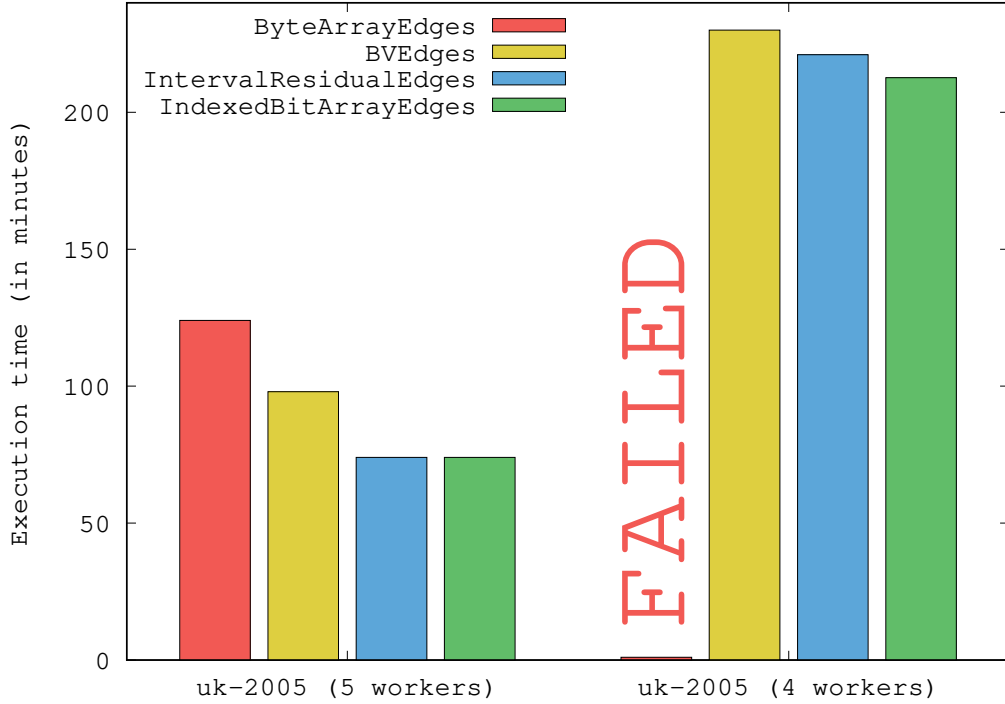


Figure 11: Execution time (in minutes) of the PageRank algorithm for the graph *uk-2005* using 5 and 4 workers. IntervalResidualEdges and IndexedBitArrayEdges outperform ByteArrayEdges which fails to complete execution with 4 workers.

2.4.3.5 Initialization time comparison

Having measured the execution time of the PageRank algorithm using small- and large-scale graphs we now report the initialization time the different representations need. Figure 12 illustrates a comparison between our three space-efficient out-edge structures and ByteArrayEdges in two different setups. In particular, we first examine the loading time for the relatively small graph *indochina-2004* when using two workers. We observe that there are negligible differences between ByteArrayEdges, IntervalResidualEdges, and IndexedBitArrayEdges, with the former being the slowest and the latter being the fastest. In contrast, BVEEdges is significantly slower than all other representations. Furthermore, we see in Figure 12 that when loading a larger graph, i.e., *uk-2005*, the performance of the different structures varies considerably. Again, IndexedBitArrayEdges is the fastest approach, followed by IntervalResidualEdges, ByteArrayEdges, and BVEEdges, but in this setting there is obvious disparity in the initialization performance.

We note that the graph loading time is negligible compared to the execution time of the PageRank algorithm. For instance, for graph *uk-2005* using 5 worker nodes, IndexedBitArrayEdges requires 121.64 seconds to initialize the graph, whereas the execution time for this setting is over 70 minutes using any of the representations examined here. However, the significant performance gain induced when using IntervalResidualEdges and IndexedBitArrayEdges can have a notable impact in algorithms requiring less execution

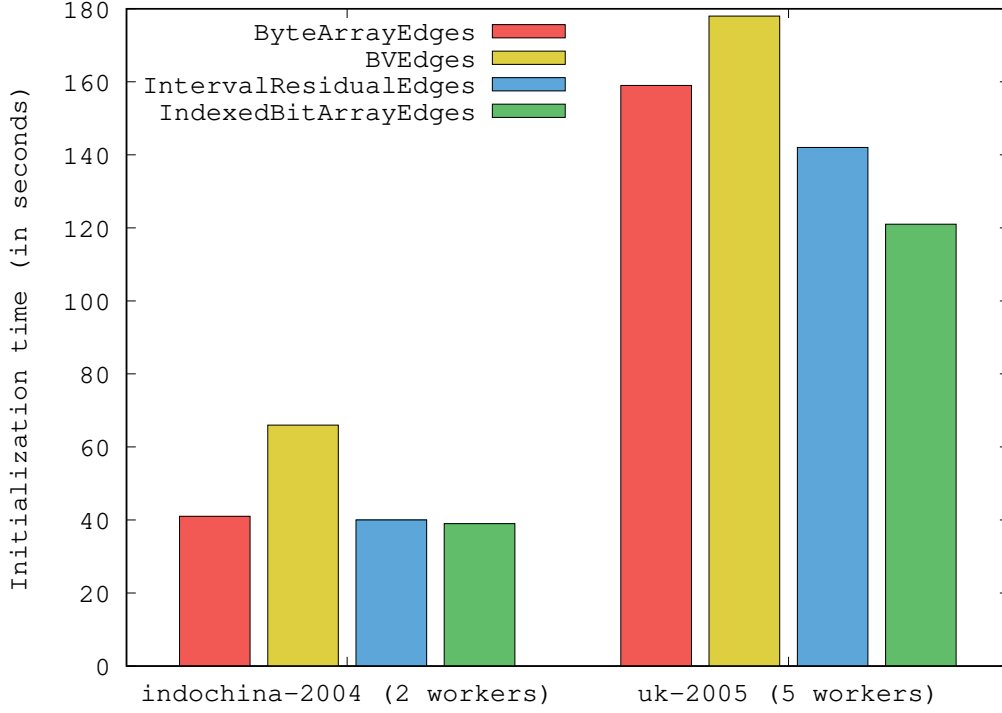


Figure 12: Initialization time (in seconds) for graphs *indochina-2004* (using 2 workers) and *uk-2005* (using 5 workers). There is notable performance gain on large-scale graphs over `ByteArrayEdges` when using `IndexedBitArrayEdges` or `IntervalResidualEdges`. `BVEEdges` is the slowest of the representations examined.

time.

2.4.3.6 Comparison when using weighted graphs

We continue our experimental evaluation by measuring the time needed for the execution of a Pregel algorithm that operates on weighted graphs. In particular, we present performance results for the different compact out-edge representations when executing the Shortest Paths algorithm, as described through Function `computeShortestPaths`. Being that all the graphs of our dataset are unweighted, we assign random weights exhibiting a Zipf distribution⁴ on the edges of graph *uk-2005*. Then, we proceed with the execution of the algorithm using `ByteArrayEdges`, `BVEEdges` and `IntervalResidualEdges` in setups of 5 and 4 workers. For `BVEEdges` and `IntervalResidualEdges` we additionally use the `VariableByteArrayWeights` representation to hold the weights of edges.

Figure 13 illustrates a comparison of the results we obtain with our representations against Giraph’s `ByteArrayEdges`. We observe that using `IntervalResidualEdges` we are able to execute the Shortest Paths algorithm more than 1.5 minutes faster than using `ByteArrayEdges`.

⁴We used the `numpy.random.zipf` function from NumPy’s random sampling library to generate weights for the graph using $\alpha = 2$.

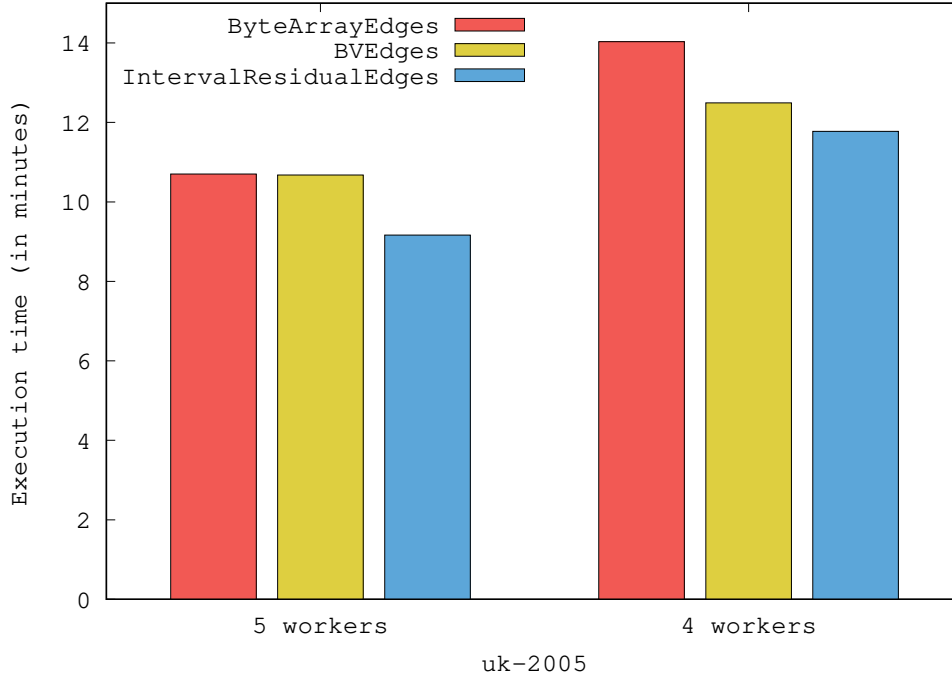


Figure 13: Execution time (in minutes) of the ShortestPaths algorithm for a single vertex, on the graph *uk-2005*, using a setup of 5 and 4 workers.

Edges in the setup of 5 workers. The significant savings in execution time are due to the limited memory usage of `IntervalResidualEdges` and `VariableByteArrayWeights`. Our `BVEEdges` behaves similarly to `ByteArrayEdges` as the computation overhead involved in accessing the edges counterbalances the merits of space-efficiency this representation offers. Furthermore, Figure 13 shows the respective results for the setup of 4 workers. We see that as we limit the available memory resources, the performance gains of our representations become more evident. In particular, `BVEEdges` is clearly also preferable than `ByteArrayEdges` in this setting being more than 1.5 minute faster. Moreover, `IntervalResidualEdges` is able to terminate 2.26 minutes faster than `ByteArrayEdges`.

We note that `VariableByteArrayWeights` requires additional 1,957.25MB of memory to hold the weights of out-edges, whereas `ByteArrayEdges` needs 5,074.36MB. Moreover, `IndexedBitArrayEdges` does not presume that the ids of out-edges are sorted, and thus, cannot support weighted graphs through `VariableByteArrayWeights`. For this reason we do not include `IndexedBitArrayEdges` in this experiment.

2.4.3.7 Comparison when performing mutations

All the aforementioned experiments focus on space-efficient structures that are applicable on algorithms that do not involve additions or removals of out-edges. However, oftentimes graph algorithms need to perform mutations on the vertices' neighbors. To this end, we examine here the performance of our novel `RedBlackTreeEdges` representation, against

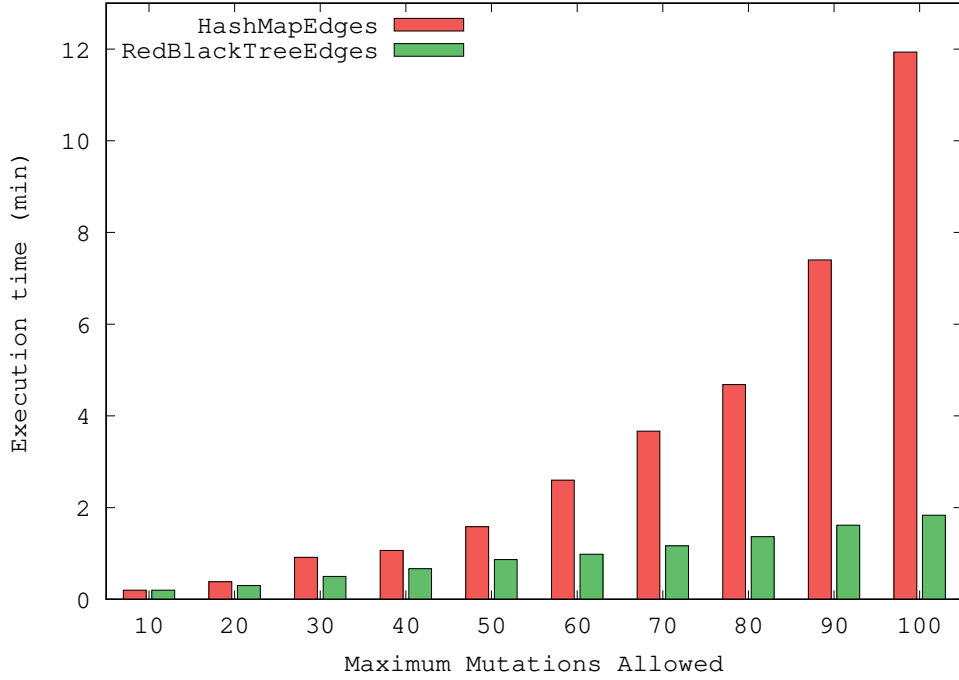


Figure 14: Execution time (in minutes) of an algorithm performing a random number of mutations on the graph *hollywood-2011* using 5 workers, for a varying number of maximum mutations allowed.

Giraph’s `HashMapEdges`. Both structures provide type flexibility, support weighted graphs, and can operate on graphs with in-situ node labelings.

We consider here an input graph which uses long integers for the ids of its nodes⁵ and the execution of a simple algorithm that performs additions and removals of out-edges on this graph. In particular, we executed over *hollywood-2011*—the largest graph we were able to load using `HashMapEdges`—an algorithm of two *supersteps*. The first one performs a random number of insertions of out-edges, and the second one removes them.

The initialization phase, in which the graph is loaded in memory, is faster using our novel tree-based structure. `RedBlackTreeEdges` needs 33.64 seconds to do so, whereas `HashMapEdges` requires 38.73 seconds. Moreover, Figure 14 depicts the execution time needed by the two representations when varying the number of maximum insertions/deletions allowed in our algorithm. We observe that when the number of mutations is low, the time spent using the two representations is equivalent. However, as the number of mutations grows and more memory is needed for the representation of the graph, the performance of `HashMapEdges` deteriorates significantly, and `RedBlackTreeEdges` proves to be clearly superior.

We note that `RedBlackTreeEdges` requires less than half of the space that `HashMapEdges` needs to load the graph *hollywood-2011* in memory. In particular, `RedBlackTreeEdges` uses 7,079.6MB of memory, whereas `HashMapEdges` uses 19,323.8MB.

⁵We examine the case of long integer ids as this is the data type used by *Facebook*, Giraph’s most significant contributor.

2.5 Conclusion

In this chapter, we propose and implement three compressed out-edge representations for distributed graph processing, termed `BVEEdges`, `IntervalResidualEdges`, and `IndexedBitArrayEdges`, a variable-byte encoded representation of out-edge weights, termed `VariableByteArrayWeights`, for compact support of weighted graphs, and a compact tree-based representation that favors mutations, termed `RedBlackTreeEdges`. We focus on the vertex-centric model that all Pregel-like graph processing systems follow and examine the efficiency of our structures by extending one such system, namely Apache Giraph. Our techniques build on empirically-observed properties of real-world graphs that are exploitable in settings where graphs are partitioned on a vertex basis. In particular, we capitalize on the sparseness of such graphs, as well as the *locality of reference* property they exhibit. We cannot, however, exploit the *similarity* property as vertices are unaware of any information regarding other vertices.

All our representations offer significant memory optimizations that are applicable to any distributed graph compressing system that follows the Pregel paradigm. `BVEEdges`, which is based on *state-of-the-art* graph compression techniques, achieves the best compression but offers relatively slow access time to the graph's elements. Our `IntervalResidualEdges` and `IndexedBitArrayEdges` representations outperform Giraph's most efficient representation, namely `ByteArrayEdges`, and are able to execute algorithms over large-scale graphs under very modest settings. Furthermore, our representations are clearly superior than `ByteArrayEdges` when memory is an issue, and are capable of successfully performing executions in settings where Giraph fails due to memory requirements. Our compressed out-edge representations are also shown to allow for efficient execution of weighted graph algorithms, through `VariableByteArrayWeights`, a variable-byte encoding based representation of out-edge weights. Finally, through our evaluation regarding algorithms involving mutations we show that the performance of `RedBlackTreeEdges` is equivalent to that of `HashMapEdges` when memory is sufficient, and shows significant improvements otherwise.

3. UNCOVERING LOCAL HIERARCHICAL LINK COMMUNITIES AT SCALE

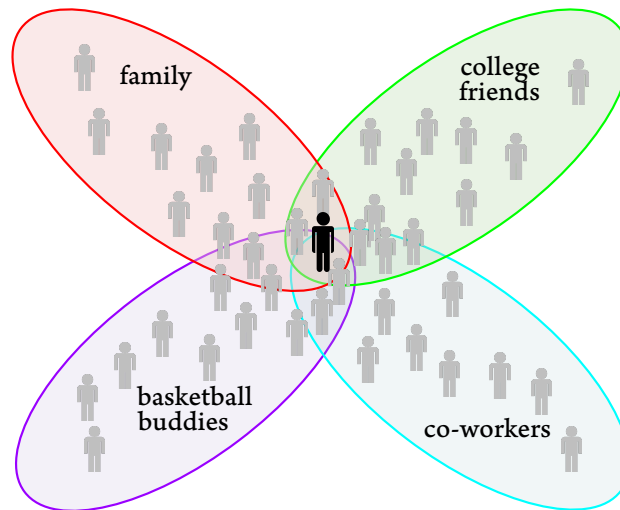


Figure 15: Illustration of the social circles of an individual. Her family, co-workers, basketball buddies and friends from college are distinct yet overlapping communities.

The neurons in our brains, the proteins in live cells, the powerplants of an electrical grid, and the users of an online social networking service, are all entities of *complex systems* that play a vital role in our daily lives. Networks are a powerful tool for modeling relations and interactions between the components of such complex systems. Respective real-world networks are often massive; yet they exhibit a high level of order and organization, which allows the study of common properties they exhibit, such as the power-law degree distribution and the small-world structure [37, 42]. Another important property that real-world networks exhibit is the presence of community structure [51]. At a high level, communities are groups of nodes that share a common functional property or context, e.g., two people that attended the same school, or two movies with the same actor. In several cases communities in a network are distinct; consider for example the fans of different basketball teams. However, it is often the case that communities overlap. Figure 15 illustrates the communities of an individual in a social network, i.e., her family, co-workers, basketball buddies and friends from college. It is obvious that the communities may overlap in different ways. For example, a co-worker may also be a basketball buddy and a friend from college. Such overlapping communities may have a complex structure of connections that are not easy to discern and are certainly more challenging to identify compared to non-overlapping ones.

Effectively extracting the community structure of a node in a network has many useful applications:

- We can provide more informative and engaging social network feeds by better understanding the membership of an individual to various organizational groups.

- We can suggest common friends of an individual to connect because they share mutual interests.
- We can create match-making algorithms for online players based on the similarity of their game play.
- We can identify groups of customers with similar behavior and enhance the efficiency of recommender systems.

Early community detection approaches focused either on grouping the nodes of a network or on searching for links that should be removed to separate the clusters [45]. However, these approaches did not consider the fact that communities may overlap, and ultimately could not provide an accurate representation of a network's community structure. Algorithms that followed [7, 41, 53, 119, 123, 125] allow for nodes to belong to several overlapping communities by employing techniques such as link clustering, matrix factorization, and personalized PageRank vectors. Still, these approaches are not applicable to the massive graphs of the *Big Data* era, as they focus on the *entire* graph structure and do not scale with regards to both execution time and memory consumption. Recent efforts have therefore shifted the focus from the global structure to a local view of the network [61, 70, 72, 79]. More specifically, such approaches locally expand a set of target nodes in the community of interest, instead of uncovering the communities of the entire network.

Seed set expansion approaches employ techniques such as random walks to estimate the likelihood of a node to participate in the target community, and manage to scale to large networks [61, 70, 72, 79]. These approaches consider that overlaps between communities are sparsely connected whereas the areas where communities overlap are denser than the actual communities. However, studies of real-world networks show that two nodes are more likely to be connected if they share multiple communities in common [127]. For example, people belonging to both the co-workers and basketball buddies communities of Figure 15, are expected to know each other with high probability. Hence, as the overlapping area is in fact denser than the actual communities, seed set expansion approaches are driven towards nodes that reside in the overlap. In addition to this, all scalable methods require *multiple seeds* to avoid detecting multiple overlapping communities as a single one. This constitutes a challenge, as it is usually the case that we are interested in all communities of a single node, instead of seeking one community involving multiple predefined nodes. Finally, seed set expansion approaches are shown to perform well when detecting relatively large communities, whereas high quality communities are in fact small [127].

Here, we focus on the neighbors of a single node in the network, i.e., its *egonet*, and aim at extracting the –possibly overlapping– communities of this node. We build upon the ideas of *link clustering* [7, 41] and employ *similarity* measures that allow for effectively handling densely connected overlaps between communities. Our intuition is that when grouping pairs of links we should capture the *extent* to which a link belongs to multiple overlapping communities. To this end, we utilize a dispersion-based tie-strength measure that helps us quantify the participation of a link's adjacent nodes to more than one community. Our

approach is both *efficient* and *scalable* as we focus on local parts of graphs comprising a target node and its neighbors. As we show in our experimental evaluation, we produce a more accurate and intuitive representation of the community structure around a node for a number of real-world networks.

In summary, we make the following contributions:

- We propose a local community detection algorithm that effectively reveals the overlapping nature of real-world network communities of individual target nodes.
- We operate with less input from the user (a single seed vs a set of multiple seeds) and generate communities of equal or better quality.
- We experimentally evaluate our algorithm against state-of-the-art approaches using publicly available networks. Our results show that our approach significantly outperforms current methods using popular evaluation metrics.
- We reduce the execution time notably, by focusing on the neighborhood of a node and thus, manage to handle billion-edge scale graphs.
- We provide a detailed view of the rich hierarchical structure of the derived community.

3.1 Background

In this section we review some basic principles and definitions for our work. First, we provide the definition of the *egonet* and subsequently we discuss measures that are used to estimate the strength of *ties* in networks. Finally, we give the definition of *partition density* and detail the dataset used in our study.

3.1.1 Egonet

Graph mining applications focusing on large-scale graphs are often based on local neighborhoods of nodes [52]. The set of nodes that are one hop away from a given node allows for a variety of analyses that build intuition about that node and its role. Focusing on local neighborhoods of nodes enables respective applications to scale effortlessly to large networks as the task in hand can be executed in parallel for all nodes in the network. In the context of social networks, this one hop neighborhood is frequently called the *egonet* of a user. Figure 15 depicts such an egonet of an individual and the overlapping communities she is part of.

In this work, we address the challenge of extracting efficiently the community structure formed by the nodes adjacent to a *single* target node. The networks we are interested are often massive and, thus, our approach should scale to graphs of extreme volume. To this end, we investigate ground-truth communities of real-world networks and in particular,

whether these communities are recoverable using egonets alone. Eventually, we focus on the egonets of target nodes to significantly reduce the search space of our algorithm.

3.1.2 Tie Strength Measures

The *closeness* between nodes in a network and its impact on the network's dynamics has been studied extensively [58, 95]. Understanding the complex nature of interacting objects calls for quantifying the *strength* of ties to distinguish the connections of particular importance. We outline here the tie strength measures that we employ in the context of this work:

3.1.2.1 Embeddedness

Intuitively, a large number of shared neighbors between nodes indicates a *strong* tie, whereas a few mutual neighbors indicate a *weak* tie. Therefore, a frequently used measure to estimate the tie strength between two nodes u and v is *embeddedness*, formally defined as:

$$emb(u, v) = |N_+(u) \cap N_+(v)| \quad (3.1)$$

where $N_+(u)$ is the set of nodes adjacent to u .

In the case of social networks, individuals operating on a common context are more likely to share joint activities with each other, as opposed to people that do not share this particular context [43]. Therefore, *embeddedness* can effectively be applied for the identification of couples [44].

3.1.2.2 Jaccard similarity coefficient

The Jaccard similarity coefficient is a frequently used measure of similarity of two sets and is defined as the size of their intersection divided by the size of their union. In the case of two nodes u and v in a network, the Jaccard similarity coefficient can be applied on the respective sets of neighbors, $N_+(u)$ and $N_+(v)$, as follows:

$$J(u, v) = \frac{|N_+(u) \cap N_+(v)|}{|N_+(u) \cup N_+(v)|} \quad (3.2)$$

3.1.2.3 Absolute and Recursive Dispersion

The task of identifying spouses or romantic partners in a social network is also the focus of [12]. Backstrom and Kleinberg address this challenge through the use of *dispersion*-based measures. They analyze real data from Facebook and conclude that high dispersion is indeed present, not only to spouses or romantic partners, but to people who share multiple relevant aspects of their social environment in general.

Table 3: Graphs of our dataset reaching up to 1.8 billion edges.

Graphs	Type	Nodes	Edges	Average Degree	Av. Community Size
DBLP	Co-authorship	317,080	1,049,866	3.31	22.45
Amazon	Co-purchasing	334,863	925,872	2.76	13.49
Youtube	Social	1,134,890	2,987,624	2.63	14.59
LiveJournal	Social	3,997,962	34,681,189	8.67	27.80
Orkut	Social	3,072,441	117,185,083	38.14	215.72
Friendster	Social	65,608,366	1,806,067,135	27.53	46.81

Formally, we consider the egonet G_u of u in G and define *absolute dispersion* as:

$$disp(u, v) = \sum_{\substack{s, t \in C_{uv} \\ s < t}} d_v(s, t) \quad (3.3)$$

where C_{uv} is the set of common neighbors of u and v in G_u , and $d_v(s, t)$ is a distance function equal to 1 when s and t are not directly linked themselves and have no common neighbors in G_u other than u and v , and 0 otherwise.

Experiments show that for a fixed value of $disp(u, v)$, increased embeddedness is a negative predictor of whether v is close to u [12]. Thus, absolute dispersion is more effective when normalized using embeddedness. In addition, the performance of dispersion is found to strengthen when applying it recursively as follows. First, we consider $x_v = 1$ for all neighbors v of u . Then, we iteratively update x_v using the formula:

$$x_v = \frac{\sum_{w \in C_{ij}} x_w^2 + 2 \sum_{\substack{s, t \in C_{ij} \\ s < t}} d_v(s, t) x_s x_t}{emb(u, v)} \quad (3.4)$$

The value produced after the *third iteration* of (3.4) is empirically found to perform the best [12]. We will refer to this value as *recursive dispersion* of v in G_u for the rest of this chapter, and use it to identify pairs of nodes that operate in multiple common contexts.

3.1.3 Partition Density

Agglomerative community detection algorithms provide us with a dendrogram describing the hierarchical organization pattern of communities [102, 45, 7]. To obtain meaningful communities from the dendrogram it is necessary to determine the level at which to cut the tree at. To this end, Ahn et al. [7] introduced the measure of *partition density* D , that is formally defined as follows:

$$D = \frac{2}{|E|} \sum_{c \in C} e_c \frac{e_c - (n_c - 1)}{(n_c - 2)(n_c - 1)} \quad (3.5)$$

where C is the set of communities discovered, e_c is the number of links in a community $c \in C$, and n_c is the number of nodes all the links in e_c touch.

We can come up with the optimal value of D by examining its value at each step of the hierarchical clustering process. Cutting the dendrogram at the level that produces the optimal value of D is shown to effectively derive meaningful and relevant communities. In addition, partition density is suitable for large-scale graphs as it does not suffer a resolution limit like *modularity* [46], being that every term in Equation (3.5) is local in each community c .

3.1.4 Networks in our Dataset

Evaluating and comparing communities detected by different algorithms is not a trivial task. Large networks exhibit extremely complex organization and cannot be visualized in meaningful ways. However, we can measure the accuracy of a community detection algorithm given the presence of *ground-truth* communities [127].

In this work, we employ *all six* of the real-world networks with available ground-truth communities that are provided by the Stanford Network Analysis Project (SNAP).¹ In particular, our evaluation is based on the top-5,000 highest quality communities of each of these networks [124]. Table 6 provides the details of our dataset.

DBLP is a co-authorship network and ground-truth is formed from authors who published in the same journal or conference. *Amazon* is a product co-purchasing network and the annotated communities associated with it are based on the categories of the products. Finally, *Youtube*, *LiveJournal*, *Orkut*, and *Friendster* are all social networks, and the respective ground-truth communities are user groups that have been formed in these networks. We note that Table 6 features a graph that exceeds 1.8 billion edges, namely *Friendster*. We also see that, the average community size of most networks is relatively small, with the exception of *Orkut* with an average size of 215.72.

3.2 Local Dispersion-aware Link Communities

In this section we describe in detail our approach for local community detection. We commence by examining the coverage ratio of egonets on the ground-truth communities of the networks in our dataset. We then discuss the difficulties that existing methods based on seed set expansion and link clustering face due to the nature of real-world overlapping communities. We show that the use of dispersion-based measures of tie strength can alleviate such issues. Then, we present our algorithm, termed LDLC, in detail. Finally, we discuss a novel sampling technique to effectively reduce the search space of our algorithm.

¹<https://snap.stanford.edu/data/#communities>

3.2.1 Egonet Coverage Ratio

Community detection methods that focus on the *global structure* of graphs fail to scale to the massive volume that real-world networks reach, i.e., millions of nodes and billions of edges. We aim at detecting communities for large-scale graphs efficiently. To this end, we focus on the *local* structure of a node in the network. Studies of real-world networks show that community members tend to organize themselves around hub nodes that are linked with most of the nodes in the community [127]. We begin discussing our approach by investigating ground-truth communities of real-world networks, and in particular, the fraction of the nodes they comprise that is part of *egonets* of nodes that belong to the respective communities.

We report in Figure 16 the coverage ratio of egonets regarding the ground-truth communities of the networks of our dataset. For every ground-truth community of all six networks of our dataset, we examined the coverage of the egonets of each of the nodes belonging to the community. The average coverage ratio depicted in Figure 16, results from the egonets of the nodes with the largest coverage for each ground-truth community. We observe that the coverage ratio is very high for all networks, ranging from 87% to 97%, with the exception of *Orkut* at slightly under 67%. The lower coverage ratio of *Orkut* is attributed to the larger average community size of this network. Empirical observations [127] show that high quality communities usually consist of no more than 100 nodes, whereas the average community size of *Orkut* is more than twice as high and remains low even when using the 2-step geodesic neighborhood of nodes, as reported in [61].

The large coverage ratio of egonets on ground-truth communities verifies our hypothesis that high quality communities can be detected when focusing on egonets of nodes. This allows us to significantly reduce the scale of our search by focusing only on a small part of a possibly extremely large network. Even in the case of nodes exhibiting large degrees, dealing with the respective egonets instead of the global structure of the graph is beyond comparison with regard to efficiency. Space complexity is also reduced greatly, as the memory footprint of the egonet is insignificant when compared to the whole network.

3.2.2 Effective Detection of Local Hierarchical Overlapping Communities

Investigations on the structure of real-world networks have revealed that there is an increasing relationship between the number of shared communities and the probability of nodes being linked with an edge [127]. Hence, the nodes residing in overlapping parts of communities are more densely connected than the nodes residing in the non-overlapping parts. Moreover, *connector* nodes, i.e., nodes linked with most of the members of a community, belong to the overlap [127].

Recent local community detection methods [61, 70, 79] expand seed sets by utilizing the dynamics of random walks initiating from the seeds. The participation of a node in the target local community is determined by the corresponding probability score that results from these random walks. Naturally, nodes that reside in the dense overlapping area of multiple

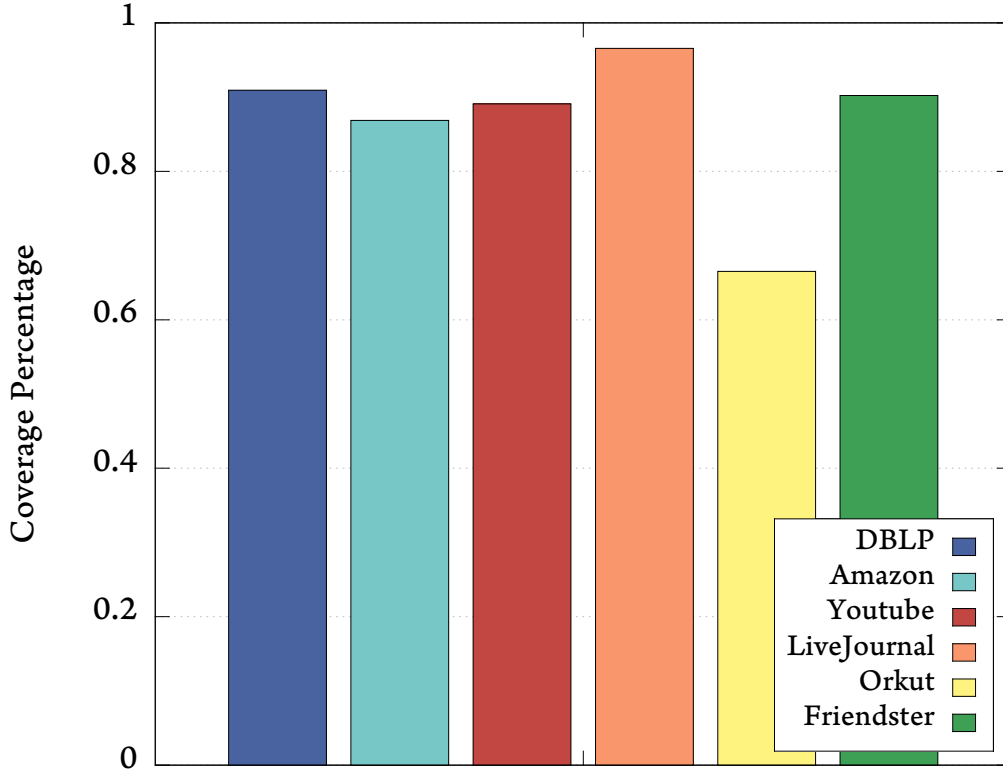


Figure 16: Egonet coverage ratio for the ground-truth communities of graphs provided by SNAP. We show that the coverage ratio for all graphs, with the exception of *Orkut*, is above 87%. The ratio is lower for *Orkut* due to its large average ground-truth community size.

communities of a particular node, have high probability scores for random walks starting off this node. In addition to this, nodes outside the overlapping area that are selected in the resulting community due to their probability scores, do not necessarily belong to the same community, as each random walk starting from a seed node is likely to reach a different community. Hierarchical link clustering approaches focusing on the global network structure [7, 41] examine the similarity of pairs of links, and thus, avoid grouping nodes that actually belong to disparate communities. However, such approaches do consider that communities in whole are more densely connected than their overlapping parts [126]. Therefore, these approaches are also unable to handle overlaps appropriately.

Figure 17 illustrates the egonet of an individual (**10**) in a social network. We use this egonet here as a *toy example*. The use of a force-directed layout enables us to easily identify the organizational groups shaped around this node. In particular, we observe that the neighbors of node **10** form two well-connected groups. We also notice, that the only node in the egonet that maintains links (red-colored) with nodes of both groups other than **10**, is **6**. The relationship between nodes **10** and **6** is a particular case of a strong tie which is frequent in networks and has to be considered when identifying overlapping communities. Node **6** acts as a *connector* in the egonet of **10** and is linked with nodes

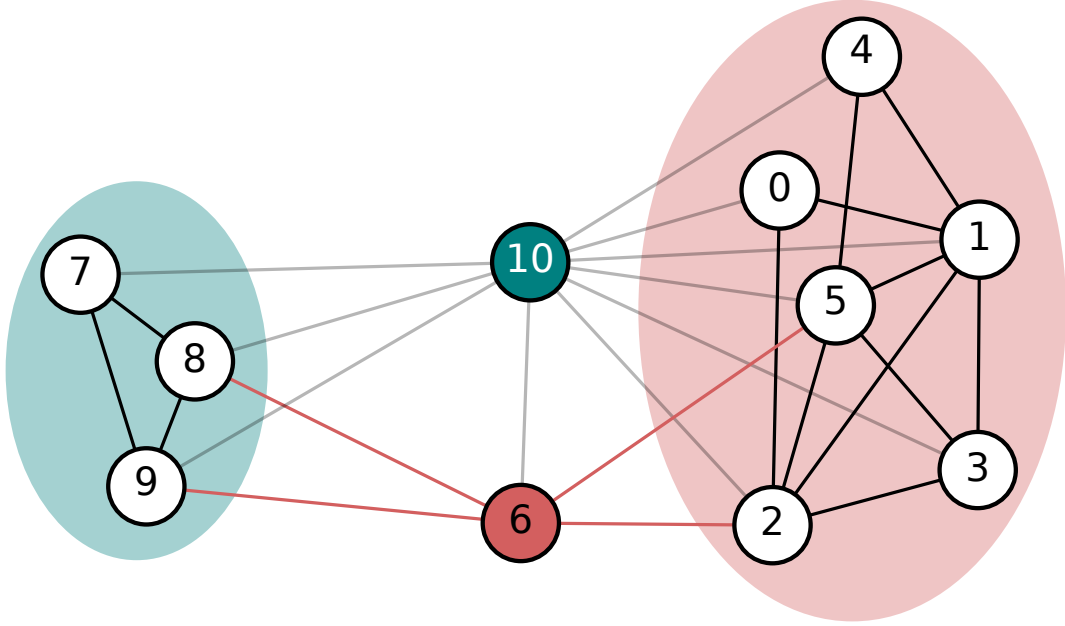


Figure 17: Social communities in the ego network of an individual (10) in a social network. Using a force-directed layout we can easily identify two well-connected groups of acquaintances. A special tie between (10) and (6) is evident, as (6) is the only vertex having links (in red) towards both communities.

that are not themselves well-connected, as they belong to different organizational groups.

3.2.2.1 Local hierarchical link communities

We address the task of local community detection by merging pairs of links in the ego network of a target node. Links often demonstrate a particular relation, e.g., a friendship between two nodes, whereas nodes are usually part of multiple groups. Thus, by grouping links –instead of nodes– we allow for the participation of nodes into multiple overlapping communities. To quantify the relevance of two edges e_{wu} and e_{wv} sharing a common node w , we can properly adopt the Jaccard similarity coefficient in the context of links [7]. Using the common node of the two links provides no additional information, and may introduce bias, depending on the degree of w . Therefore, using Equation (3.2), we can define the similarity of the pair (e_{wu}, e_{wv}) as:

$$J(e_{wu}, e_{wv}) = J(u, v) = \frac{|N_+(u) \cap N_+(v)|}{|N_+(u) \cup N_+(v)|} \quad (3.6)$$

where u and v are both adjacent to w .

After quantifying the similarity of all pairs of links in the ego network of Figure 17 that share a common node using Equation (3.6), we can build a hierarchy of communities through agglomerative clustering. More specifically, we proceed by merging pairs of links by descending order of similarity. Finally, we cut the resulting dendrogram at the level of optimal

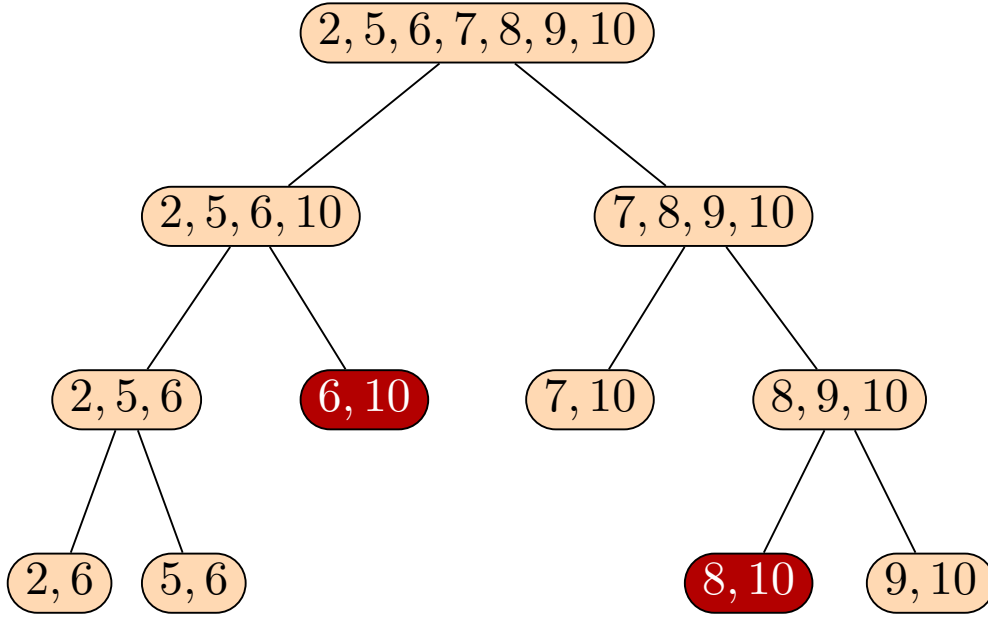


Figure 18: The hierarchical link structure of the malformed community that results when performing link clustering in the egonet of Figure 17 using Equation (3.6), and cutting at the level of optimal partition density. The similarity of link (6,10) with link (8,10) leads to a community that groups numerous nodes that are not linked with each other.

partition density, and the communities we come up with are:

- a) {0, 1, 2, 3, 4, 5, 10}
- b) {6, 7, 8, 9, 10}
- c) {2, 5, 6, 7, 8, 9, 10}

We see that the first two communities are *well-knit*, i.e., the respective sub-graph is quite dense. However, we also observe that the third community coalesces numerous nodes that are not linked together (2, 5 with 7, 8, 9). This is an effect of the Jaccard similarity coefficient that is used for quantifying the similarity of pairs of links, as this measure is unable to capture *how well* the neighbors of two nodes are interconnected. Indeed, the hierarchical link structure of this particular community, as portrayed through Figure 18, highlights evidently the cause of this undesired behavior. We see that the third community actually results from the coalescence of two well-separated clusters (2, 5, 6, 10 and 7, 8, 9, 10). This grouping occurs due to the similarity of links (6, 10) and (8, 10). These links are in fact similar when considering the Jaccard similarity coefficient; however they belong to an overlapping area of different communities that Equation (3.6) is unable to take into account and consequently reveal. In addition, as the distance between two clusters is determined by a single pair, grouping large clusters with each other results in a dendrogram that reveals very little about the hierarchical structure of the community.

3.2.2.2 Building on dispersion-based measures

Estimating the relevance of pairs of links in the presence of dense community overlaps calls for measures that take into account the extent to which the neighbors of two nodes are interconnected. Dispersion-based measures address this challenge and therefore fit perfectly in the task of overlapping community detection. Through their use, we are able to *single out connector nodes* that lie in overlapping parts of communities. For example, using Equation (3.3) we obtain that node **6** exhibits the highest absolute dispersion in the egonet of **10** with a value of 4. Hence, we can employ dispersion-based measures to favor groupings of pairs of links with adjacent nodes that share a lot of common neighbors (high Jaccard similarity coefficient) only if these neighbors are also well interconnected (low recursive dispersion). In this way, connector nodes are involved in groupings at a *higher* level of the resulting dendrogram, which then depicts more accurately the hierarchical structure of the communities in the egonet. In particular, we propose the use of the recursive dispersion measure *along with* the Jaccard similarity coefficient in order estimate the relevance of pairs of links. More formally, we define the similarity S of two pairs of links (e_{wu}, e_{wv}) to be:

$$S(e_{wu}, e_{wv}) = \frac{J(e_{wu}, e_{wv})}{rec(u) + rec(v) + rec(w)} \quad (3.7)$$

where $rec(u)$ is the recursive dispersion of u in the egonet of the target node.

Returning on the example of Figure 17 and the egonet of node 10, if we apply hierarchical link clustering using Equation (3.7) as a measure of similarity instead of (3.6), we come up with the following communities:

- a) {0, 1, 2, 3, 4, 5, 10}
- b) {7, 8, 9, 10}
- c) {2, 5, 6, 10}
- d) {6, 8, 9, 10}

We observe that the nodes of all communities are much more well-connected. Moreover, node **6** is featured in two communities, in which every two distinct vertices are adjacent, i.e., they form *cliques*. Evidently, through Equation (3.7), we are able to penalize the high dispersion that node **6** exhibits in this egonet. Links featuring this node are now merged at a higher level of the resulting dendrogram, and thus, we avoid forming communities featuring nodes of different organizational groups.

3.2.3 Our Proposed LDLC Algorithm

We present here the LDLC algorithm for finding local hierarchical overlapping communities in large-scale graphs. LDLC is an agglomerative clustering algorithm whose aim is to reveal

Algorithm 1: LDLC(G, u)

input : An undirected network $G = (V, E)$,
and a node $u \in V$.

output : A dendrogram depicting the hierarchical (possibly overlapping) communities of G_u .

```

1 begin
2    $G_u(V_u, E_u) \leftarrow \text{egonet}(G, u);$ 
3    $rec \leftarrow \text{dict}();$ 
4   foreach  $v \in G_u, v \neq u$  do
5      $rec[v] \leftarrow 1;$ 
6   for  $iteration \leftarrow 1$  to 3 do
7     foreach  $v \in V_u, v \neq u$  do
8        $rec[v] \leftarrow \frac{\sum_{w \in C_{uv}} rec[w]^2 + 2 \sum_{s, t \in C_{uv}, s < t} d(s, t) rec[s] rec[t]}{emb(u, v)};$ 
9    $similarities \leftarrow \text{min\_heap}();$ 
10  for  $k \in V_u$  do
11    for  $(e_{ik}, e_{jk}) \leftarrow \text{combinations}(N_+(k), 2)$  do
12       $J(e_{ik}, e_{jk}) \leftarrow \frac{|N_+(i) \cap N_+(j)|}{|N_+(i) \cup N_+(j)|};$ 
13       $S(e_{ik}, e_{jk}) \leftarrow \frac{J(e_{ik}, e_{jk})}{rec[i] + rec[j] + rec[k]};$ 
14       $similarities \leftarrow (1 - S(e_{ik}, e_{jk}), (e_{ik}, e_{jk}));$ 
15  foreach  $(similarity, (e_{ij}, e_{jk})) \in similarities$  do
16     $\text{join\_clusters}(e_{ik}, e_{jk});$ 
17    if  $\text{len}(\text{clusters}) == 1$  then
18      break;

```

the hierarchical structure of the possibly overlapping communities of a single target node in a network. LDLC uses Equation (3.7) to determine the similarity of all pairs of links in the egonet of the target node in the network that share a common node. These pairs of links are merged progressively in ranking order according to their similarity. The groupings result in a dendrogram that depicts the hierarchical organization of the communities the target node belongs to. To derive the actual communities, we may cut this dendrogram at the level that produces the optimal partition density (Equation (3.5)), or alternatively, we can cut it at the level that produces the desired number of communities.

Algorithm 1 outlines our suggested LDLC. The input of our algorithm comprises an undirected graph $G(V, E)$ and a single target node $u \in V$. The output of LDLC is a dendrogram depicting the rich hierarchical structure of the local communities of node u .

Loading the egonet and initializing the communities: We start by loading in memory the egonet of u , i.e., node u , its adjacent nodes, and the edges among them (Line 2). Depending on the network representation, this process can be quite costly. For example, using an edge-list or an adjacency-list stored in a file would necessitate two passes over the files, to identify the neighbors of u as well as the neighbors of u 's neigh-

bors. To alleviate this cost we focus instead on space-efficient in-memory representations [2, 20, 87, 84, ?]. Such approaches allow for fast neighbor queries as they maintain adjacency lists in-memory, usually in sorted order. Thus, to come up with the egonet of u , we first retrieve the adjacency list of u , and then the adjacency lists of u 's neighbors. For each of the latter adjacency lists we keep only those nodes that are neighbors of u , by applying intersection with the adjacency list of u . This operation ends up with u 's egonet as it discards all nodes that are not adjacent to u , as well as the respective edges. Having loaded the egonet, we proceed by initializing the communities. More specifically, we consider every edge $e \in E_u$ to be a community of its own, with the two adjacent nodes as its members.

Computing the recursive dispersion of u 's neighbors: Lines 3–8 compute the recursive dispersion of all neighbors of u , $v \in V_u$. We map the respective recursive dispersion values of the neighbors in a dictionary (associative array) that uses the nodes as keys (Line 3). We first assign a value of 1 for all nodes (Lines 4–5) and then we apply Equation (3.4) for three iterations (Lines 6–8). This process results in the values of recursive dispersion, as defined in [12] and detailed in Section 3.1.2.3.

Computing the similarities of pairs of links: Our next objective is to come up with the similarity of all pairs of links in the egonet that share a common node. To this end, for every node in the egonet we examine the similarity of all possible pairs of its links (Lines 9–14). The use of a *min-heap* allows us to maintain the similarities of pairs of links sorted (Line 9). We first calculate the distance of two links using the Jaccard similarity coefficient (Line 12), and then we balance this distance using the previously calculated recursive dispersion measure, as specified in Equation (3.7). In particular, we divide the value of Jaccard similarity coefficient with the sum of the recursive dispersion of the nodes involved in the links (Line 13). Finally, we insert the resulting similarity value in the heap holding the similarities of all pairs (Line 14).

Creating the dendrogram: The last step of our algorithm is to merge the pairs of links and come up with the dendrogram that portrays the hierarchical structure of the local communities of node u . More specifically, we iterate through our heap holding the sorted similarities of pairs of links, and group the pairs one by one (Lines 15–16). At every grouping stage, we keep track of the action that takes place to allow for the construction of the dendrogram. Moreover, we monitor the current partition density using Equation (3.5), to determine the best level at which to cut the dendrogram at. When the dendrogram is built, i.e., when we are left with a single cluster, LDLC terminates (Lines 17–18). The resulting dendrogram we come up with reveals the overlapping nature of the network's communities through a rich and intuitive hierarchical structure.

Analysis of LDLC: The running time of our algorithm depends on the calculations needed to come up with the intersection and the union of the neighbors of all pairs of nodes in the egonet. The former is needed for the calculation of both recursive dispersion and similarity, whereas the latter is needed just for calculating similarity. There are totally $\binom{|V_u|}{2}$ pairs of nodes in the egonet, where $|V_u|$ is the number of nodes in the egonet. Both the intersection and union operations require linear time, as we consider representations of sorted adjacency lists. Consequently, the running time of LDLC is $O(|V_u|^3)$. When it

comes to egonets of nodes with large degrees, this order may become unmanageable; hence, we continue with a discussion on how we can reduce our search space and be efficient even in such cases.

3.2.4 Reducing the Search Space

LDLC operates on the egonets of a target node, as community detection in the global structure of the network is prohibitively expensive for large-scale graphs. However, detecting communities in the egonets of certain nodes in the network may have equivalent cost. In particular, many real-world networks, such as the Internet router graph [42], the World Wide Web graph [14, 27, 4], and citation graphs [109], are known to exhibit power law degree distributions and a few of their nodes exhibit extremely large degrees. Therefore, the size of the respective egonets of these nodes may be comparable to the size of the network.

Uncovering the community structure of nodes with large egonets efficiently calls for a sampling technique that is applied on the egonet to reduce the search space. To this end, a straightforward approach is to perform random sampling, i.e., to pick uniformly at random a subset of the nodes in the egonet, and apply LDLC on the respective subgraph that comprises these nodes. Such an approach would successfully reduce the time needed to execute our algorithm; however, a random sample of the neighborhood of a node exhibiting high degree is likely include many disparate nodes.

Algorithm 2 outlines an alternative sampling technique. Instead of including nodes in our sample at random, we maintain the most *involved* nodes of the egonet. More specifically, we use a min-heap (Line 2) that holds in its root the inserted node with the smallest degree. First, we insert in the min-heap the first k nodes of the egonet (Lines 3 – 4). Then, for the rest of the nodes in the egonet (Line 5), we examine whether their degree is larger than that of the node in the root of the min-heap (Line 6). If so, we remove the node in the root and insert the current node in the min-heap (Line 7 – 8). After we have iterated through all nodes in the egonet, the min-heap will hold the nodes with the largest degrees in the network, which is the outcome of Algorithm 2.

We employ this sampling technique in our experimental section for egonets that surpass 100 nodes and examine its effectiveness through a comparison against a random sampling approach. We note that communities with more than 100 nodes are reported to be of poor quality [127].

3.3 Experimental Evaluation

We compare LDLC against three prominent community detection algorithms based on seed-set expansion, namely LEMON [79], LOSP [61], and HeatKernel [70]. All three above algorithms perform *local* community detection and thus, allow for comparison with our approach in a similar setting. We first discuss the specifications of our experimental setting.

Algorithm 2: $k\text{-largest}(G, u)$

input : The egonet of $u \in G$, $G_u = (V_u, E_u)$,
and maximum size k .

output : A sample V'_u of V_u , where $N_+(v)$ in the top- k in $G_u \forall v \in V'_u$.

```

1 begin
2    $V'_u \leftarrow \text{min\_heap}()$ ;
3   for  $i \leftarrow 1$  to  $k$  do
4      $V'_u \leftarrow V_u[i]$ 
5   for  $i \leftarrow k + 1$  to  $|V_u|$  do
6     if  $V_u[i] > V'_u.\text{low}()$  then
7       delete  $V'_u[1]$ ;
8        $V'_u \leftarrow V_u[i]$ 
9   return  $V'_u$ ;

```

Then, we proceed with the evaluation of our LDLC by answering the following questions:

- How well does LDLC overcome the need of constraints other methods have, such as requiring multiple seeds to avoid mixing-up multiple overlapping communities, and detecting mostly large communities?
- How well does LDLC perform in detecting communities of real-world networks compared to other methods?
- How efficient is LDLC when compared to other local community detection approaches?
- What is the impact of dispersion-based measures on the quality of the derived hierarchical community structures?
- How effective is our sampling algorithm when compared with a random sampling approach?

3.3.1 Experimental Setting

Our dataset comprises six social, co-authorship, and co-purchasing networks of different sizes, the details of which are outlined in Section 3.1.4. We implemented LDLC using Python 2.7 and the Snap.py interface² of the SNAP system [2]. Our algorithm is publicly available.³ We conducted our timing experiments on a Dell PowerEdge R630 server with an Intel®Xeon® E5-2630 v3, 2.40 GHz with 8 cores, and a total of 256GB of RAM. Our approach could be easily run in parallel as each node can act unilaterally, but we restricted to using only one core to refrain from treating the rest of the approaches unfairly.

²<https://snap.stanford.edu/snappy/index.html>

³<https://bitbucket.org/panagiotisl/ldlc>

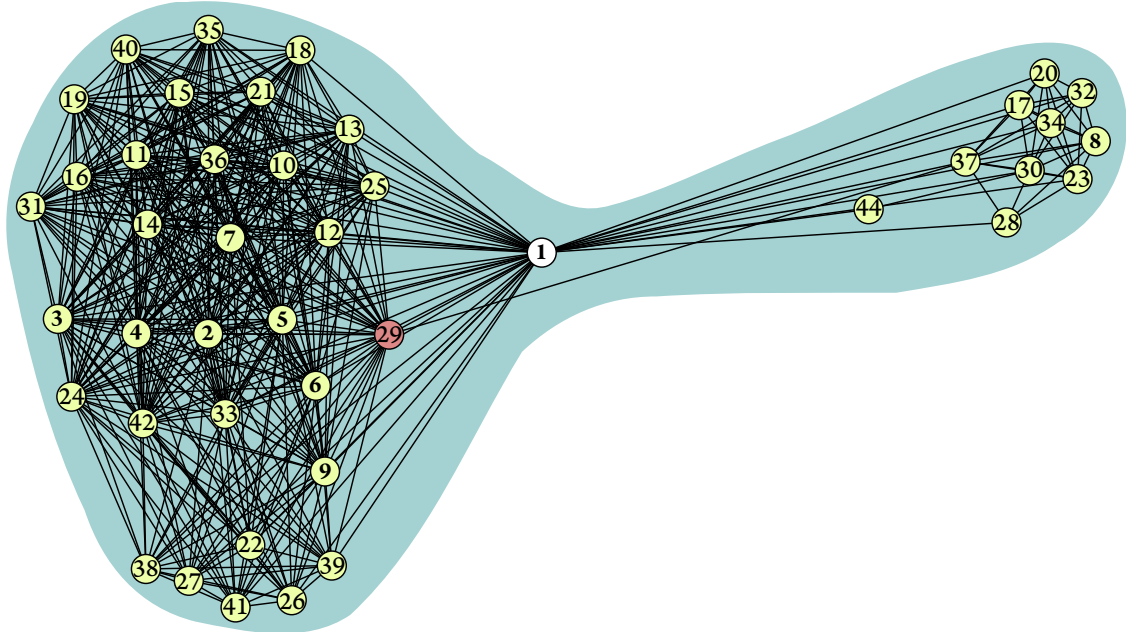
3.3.2 Qualitative Evaluation

We begin our discussion on experimental results by illustrating the behavior of our LDLC against LEMON, when discovering the communities of a target node in the *DBLP* co-authorship network.

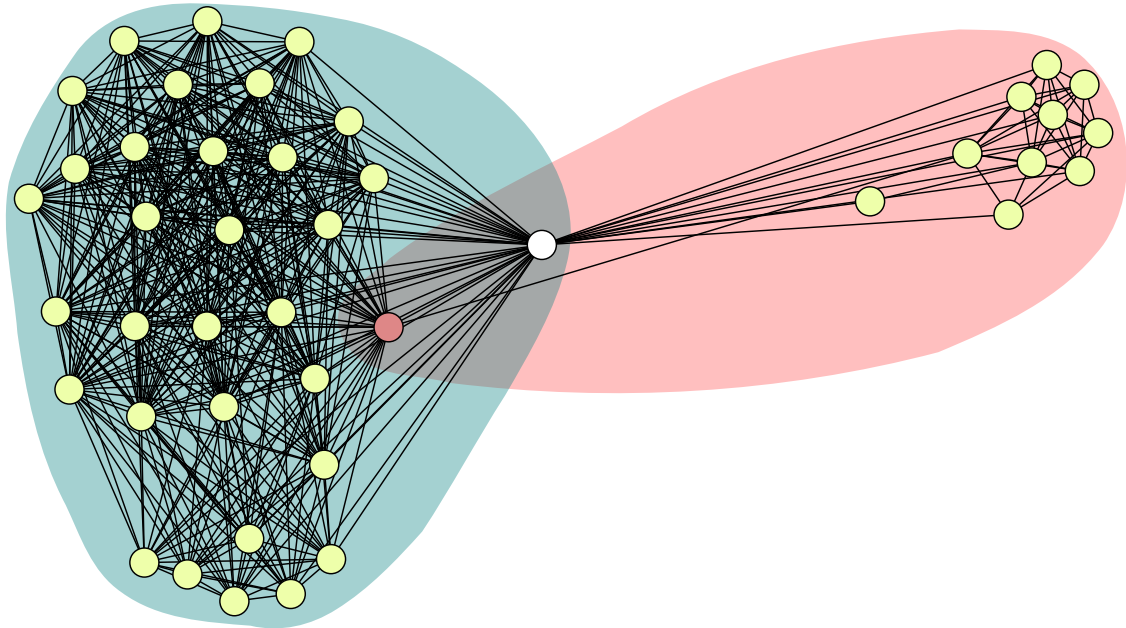
Figure 19 depicts the egonet of the target node which we use as a seed to both algorithms (white colored node), as well as the communities detected by the two algorithms. The force-directed layout we use to enhance the visualization, reveals that the nodes form two well-connected groups. The nodes of the grouping in the right actually belong to one of *DBLP*'s high quality ground-truth communities to which none of the nodes of the left grouping belongs to. Moreover, we observe, that the pink colored node is part of the left group but features a link with a node that is part of the right group and is not connected with any of the pink node's neighbors other than the white node. This results to a high value of absolute dispersion for the pink node in the egonet of the white node.

Figure 19a illustrates part of the community that is detected using LEMON. In particular, providing the white colored node as a seed to LEMON, results in a detected community of 81 nodes in total, featuring all the neighbors of the seed node, as well as nodes that are only connected to the target's neighbors. The numbers on the nodes in Figure 19a indicate their ranking according to their likelihood to belong to the target community. We observe that the community detected by LEMON exhibits certain unexpected or undesired attributes. First, high quality ground-truth communities are reported to be much smaller than the community detected by LEMON. In particular, the high quality communities of *DBLP* have an average community size of 22.45 nodes, as shown in Table 6. The community of Figure 19a however, is more than 3 times as large. Second, using the target node as the single seed results in the participation in the detected community of nodes that belong to different social circles. In particular, LEMON performs random walks starting from the target node to calculate the likelihood of a node belonging to the detected community. Naturally, nodes of different social circles are likely to exhibit high likelihood and LEMON is unable to distinguish between the different and possibly overlapping communities of the target node. This behavior is evident in Figure 19a. We observe that nodes ranked from 2 to 7 according to their likelihood, reside in the middle part of the left well-connected group of the seed's neighbors. The node that LEMON adds to the community immediately after, ranked 8th, does not share a single link with these nodes, and clearly belongs to another community. Similarly, LEMON continues to add nodes in the detected community from diverse areas around the seed node, until it meets a stopping criterion. Therefore, we see that LEMON favors nodes that reside in dense areas *regardless of their relevance to one another*. Overcoming this issue would require *multiple cherry-picked* seeds that would increase the likelihood of nodes that are actually part of the same community. This is equally true for other methods that employ random walks for seed set expansion, including LOSP.⁴ Last, the pink colored node continues to exhibit high dispersion in the community detected by LEMON, as the community features its link with a node of the cluster on the right

⁴As the authors show in [61] (Figure 2) the presence of three seeds is essential to enable LOSP to distinguish between two overlapping cliques.



(a) LEMON



(b) LDLC

Figure 19: The egonet of a node in the *DBLP* graph. LEMON's detected community (19a) features, among others, all the nodes in the egonet. Numbers indicate the LEMON's ranking of the nodes according to their likelihood of belonging to the detected community. LDLC uses hierarchical link clustering in the egonet of the target node and penalizes the links with nodes exhibiting high dispersion to come up with two communities, colored teal and pink (19b).

Table 4: F1 Score comparison.

Algorithm	DBLP	Amazon	Youtube	LiveJournal	Orkut	Friendster
LDLC	0.843	0.894	0.560	0.876	0.438	0.688
LEMON [79]	0.525	0.910	0.190	-	0.170	-
LOSP [61]	0.691	0.845	0.413	0.674	0.216	-
HeatKernel [70]	0.257	0.325	0.177	0.131	0.055	0.078

Table 5: Execution time comparison.

Algorithm	DBLP	Amazon	Youtube	LiveJournal	Orkut	Friendster
LDLC	0.0063 sec	0.0007 sec	0.0048 sec	0.1471 sec	0.3742 sec	0.0642 sec
LEMON	9.2781 sec	9.9206 sec	12.2579 sec	-	13.1432 sec	-
LOSP	0.38 sec	0.04 sec	3.85 sec	1.47 sec	4.74 sec	-

side of Figure 19a.

Figure 19b illustrates the communities discovered in the egonet of the white colored node using LDLC. We cut the tree produced by LDLC at the level that produces the optimal partition density and observe that our algorithm detects two communities, depicted with pink and teal color, respectively. The pink community has a size of 12 nodes, and the teal community a size of 33 nodes. The average size of the two communities of LDLC (22.5) is very close to the average size of the ground-truth communities of this network. Both detected communities are well-connected. In addition, the pink-colored community is a very accurate detection of an actual ground-truth community. Finally, the pink-colored node is featured in both detected communities and does not exhibit high dispersion in either community.

We saw here that previous approaches may not detect communities well in situations like the one that we described in this qualitative evaluation. Of course, there are other examples where previous approaches can accurately identify communities. Our goal was to show the strengths of our method through a concrete example. To measure performance more objectively, we now turn to comparing the accuracy of previous local community detection techniques and LDLC through the use of our ground truth datasets.

3.3.3 Evaluation via Ground-Truth

Evaluating and comparing communities detected by different algorithms is not a trivial task. Large networks exhibit extremely complex organization and cannot be visualized in meaningful ways. However, we can measure the accuracy of a community detection algorithm given the presence of ground-truth communities [127]. In particular, we can quantify the similarity of a detected community C and a ground-truth community T using F1 score, which is defined as:

$$F_1(C, T) = \frac{2 * Precision(C, T) * Recall(C, T)}{Precision(C, T) + Recall(C, T)} \quad (3.8)$$

where precision is the fraction of detected nodes that are relevant and recall is the fraction of relevant nodes that are retrieved:

$$Precision(C, T) = \frac{|C \cap T|}{|C|} \quad (3.9)$$

$$Recall(C, T) = \frac{|C \cap T|}{|T|} \quad (3.10)$$

As there is no standard way of selecting a seed, we followed the procedure performed in [70]. We execute LDLC for all ground-truth communities of each network of our dataset, using every single node as an individual seed. For each ground-truth community, we kept the seed that produced the community with the highest F1 score. Table 4 shows the average F1 score of LDLC for all ground-truth communities of each network. In addition, we present results of 3 state-of-the-art local community detection algorithms on the same datasets. In particular, we used the publicly available implementation of LEMON⁵ to perform experiments through the same exhaustive procedure. We also executed LEMON using 3 random seeds as suggested in [79]. The results we obtained are worse than the ones reported in [79] for *both* cases, as the optimal initialization setting of LEMON differs for the various networks of our dataset. Therefore, we opt to present in Table 4 the results reported in [79] instead. We also include the results on the same dataset of LOSP, as reported in [61], and HeatKernel from [70]. We note that the results of LOSP and HeatKernel are obtained using a subset of only 500, and 100 ground-truth communities for each network, respectively.

We see in Table 4 that our LDLC manages to outperform all three state-of-the-art algorithms for all the networks of our dataset, with the exception of the *Amazon* co-purchasing graph for which LEMON is slightly better. The average F1 score of LDLC is significantly larger for all other networks, and the improvement is more evident on the social networks of our dataset, i.e., *Youtube*, *LiveJournal*, *Orkut*, and *Friendster*. For *DBLP*, *Youtube*, and *LiveJournal* the results of LDLC are impressive and much more accurate than all three other methods. Regarding *Orkut*, accurate detection is a particularly hard task, as the size of the ground-truth communities is unusually large in this network. Nonetheless, LDLC is much more effective than the other methods. The friendship graph of *Friendster* almost reaches 2 billion edges, and both LEMON and LOSP have failed to report results for this network due memory consumption. We are able to operate on the *Friendster* network despite its size, as LDLC employs a memory-efficient representation (SNAP). HeatKernel also manages to load graphs of this scale by using pylibbvg.⁶ We see in Table 4 that LDLC is able to achieve an F1 score of 0.688, which clearly outperforms HeatKernel. The results regarding the *Amazon* network differentiate due to the particular nature of its communities, which allows all 4 algorithms to achieve high accuracy. More specifically, *Amazon* is a co-purchasing network and, thus, does not feature any connector nodes [127]. In addition,

⁵<https://github.com/YixuanLi/LEMON>

⁶<https://pypi.python.org/pypi/pylibbvg>

the overlapping ground-truth communities of *Amazon* are usually nested communities, that are subsets of other communities [127].

3.3.4 Execution Time Comparison

We further evaluate LDLC as far as the execution time is concerned. In particular, for every graph of our dataset we executed LDLC for 5,000 random nodes of the graph, and report here the average execution time needed. We perform the same experiment using LEMON.

Table 7 shows the results we obtained for the two algorithms and restates the results as reported in [61] for LOSP. We observe that LDLC is significantly faster than both other methods. In particular, we are able to respond *in real-time* for the communities of all the graphs of our dataset, including *Friendster* that comprises 1,806,067,135 edges. We see in Table 7 that LDLC significantly outperforms both LEMON and LOSP with regard to execution time. This is expected, as LDLC operates only on the egonet of a target node. To produce the egonet we simply need to apply intersection on the sets of neighbors of all neighbors of the target node. Instead, LEMON and LOSP perform multiple random walks to generate a local neighborhood around the target node, a procedure that is much more costly timewise. In addition, the local neighborhood of LEMON or LOSP is usually significantly larger than the egonet of the target node. Therefore, LDLC is applied on a much smaller portion of the original graph, compared to LEMON and LOSP. We note, that the average execution time of LDLC for the *Friendster* graph is smaller than that for *LiveJournal* and *Orkut*, as the egonets of the first are sparser. Thus, LDLC has to iterate over fewer pairs of links in the grouping phase for the graph of *Friendster* and terminates faster.

3.3.5 Impact of Dispersion on the Resulting Hierarchical Community Structure

LDLC builds on hierarchical link clustering and dispersion-based measures to detect the communities of a single node in its egonet. Having discussed the accuracy and efficiency of our algorithm we investigate here its effectiveness with regard to deriving a detailed hierarchical community structure. The toy example discussed in Section 3.2.2 shows that there are cases in which relying strictly on the Jaccard similarity coefficient may result in grouping overlapping communities at an early stage. More specifically, clusters featuring multiple links are likely to be grouped with each other due to the similarity of a single pair at the low levels of the respective dendrogram.

We attempt here to quantify the extent of this trend as well as the impact of recursive dispersion on it. In particular, we consider two settings for LDLC: i) the first one ($LDLC_{nd}$) employs Equation (3.6) to estimate the similarity of pairs of links, whereas ii) the second one (LDLDC) employs Equation (3.7). Then, we investigate for every ground truth community of every network of our dataset the total merges involving the final cluster, i.e., the one featuring all links of the egonet. We use the number of total merges as a quality function, as it is indicative of the height the dendrogram reaches, and thus, of the detail we achieve with regard to the resulting community structure.

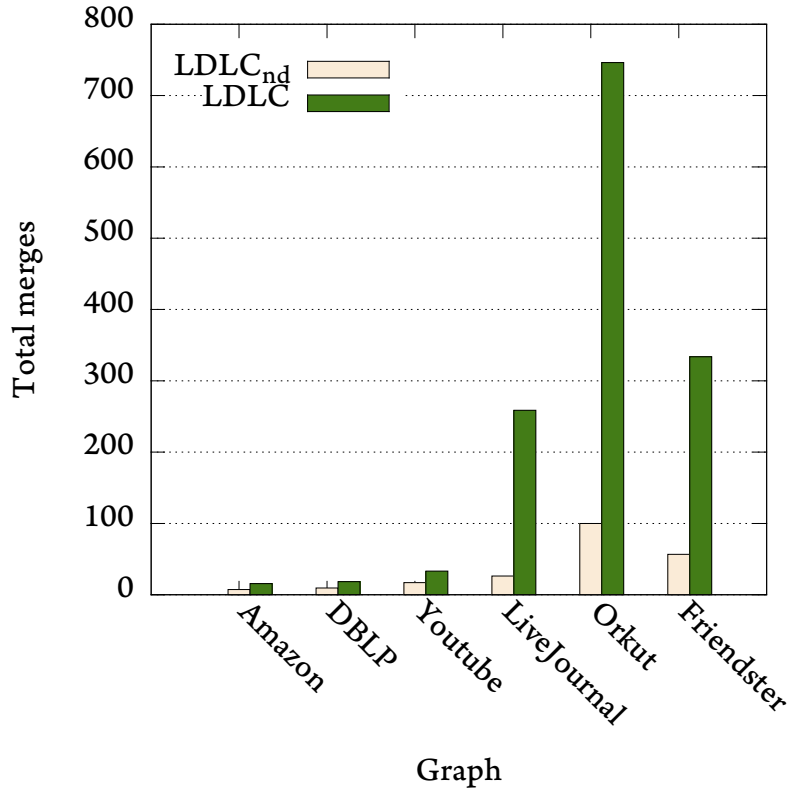


Figure 20: Impact of the use of recursive dispersion on the number of merges that occur until LDLC terminates. When using recursive dispersion (LDLC) the number of total merges increases significantly. Thus, the resulting dendrogram reveals the hierarchical community structure in greater detail.

Figure 20 illustrates a comparison between the two settings for all networks of our dataset. We see that the first setting consistently leads to *shorter* dendrograms when compared to the ones resulting using the second setting. Evidently, the use of recursive dispersion results in dendrograms with significantly richer structure. Through Equation (3.7) LDLC successfully delays the grouping of pairs of links exhibiting high dispersion in the egonet. Thus, our algorithm reveals the hierarchical community structure in greater detail. The impact is noticeable in all networks as we end up with at least twice as much merges using the second setting. The total number of merges depends on network properties such as the average degree and the average community size. For *dblp*, both these properties exhibit small values and thus the number of total merges that occur is small for both settings. In contrast, for *orkut* both these properties exhibit large values and the number of total merges that occur is large for both settings.

3.3.6 Impact of Sampling Technique

We complete our experimental section by evaluating the effectiveness of our sampling technique. More specifically, if the target node has a large egonet LDLC obtains a sample

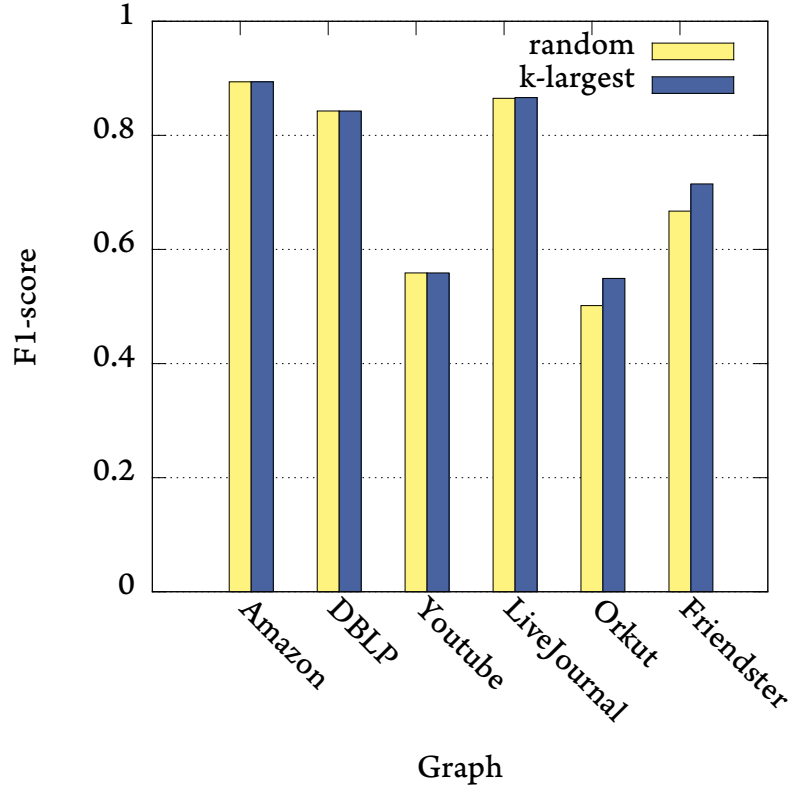


Figure 21: LDLC results on the networks of our dataset when using a random sampling technique and our k-largest sampling technique. Impact is negligible for the four first networks as very few or no egonets surpass 100 nodes. However, the k-largest sampling technique outperforms the random technique for *orkut* and *friendster*.

of the egonets and operates on the sample. Algorithm 2 outlines this process in which k nodes exhibiting the largest degrees in the egonet are retrieved efficiently. We examine here the effectiveness of this approach by comparing it with a sampling technique that selects k nodes of the egonet uniformly at random.

Figure 21 illustrates the F1-score results we obtain for all networks of our datasets when applying each of the two techniques for nodes with egonets larger than 100 nodes. There are very few or no such nodes in our results for the four smaller networks of the dataset, i.e., *amazon*, *dblp*, *youtube*, and *livejournal*. Thus, the difference in performance between the two techniques is negligible for these networks. However, for *orkut* and *friendster* several communities included in our results resulted from target nodes with egonets comprising more than 100 nodes and were sampled with the use of the two techniques. We observe that using random sampling we achieve an F1-score of 0.50 for *orkut* and 0.67 for *friendster*. The respective values when using k-largest sampling are 0.55 for *orkut* and 0.71 for *friendster*. That is, sampling the most involved nodes of the egonet instead of sampling uniformly at random has a significant impact on the accuracy we achieve.

3.4 Related Work

The problem of identifying communities emanates from research on graph partitioning, which has been active since the 1970s [68]. Girvan and Newman, with their seminal paper on community detection [51], build on Freeman's *betweenness centrality* measure [47] and define *edge betweenness* as the number of shortest paths between pairs of vertices that run along an edge. Using this measure, they iteratively remove the edges with high betweenness, as they have a tendency to connect different clusters, and thus, reveal the underlying community structure of a network. The algorithm is computationally expensive, yet this work sparked significant research in the field of community detection [45].

Many clustering methods aim at maximizing *modularity*, a measure introduced by Newman and Girvan [101]. Modularity captures the quality of a specific proposed division of a network into communities, by examining how higher the internal cluster density is than the external cluster density. One such method is that of Clauset et al. [34]. There, the proposed algorithm discovers a hierarchical community structure and identifies the best level to cut the tree at as the one that produces the division that maximizes modularity. Blondel et al. [17] propose *Louvain*, another greedy modularity maximization algorithm. Nodes are iteratively aggregated into communities as long as such a move locally improves modularity. Methods of this class are known to suffer from a resolution limit [46].

Another popular direction in the field of community detection, is the use of *random walks*. Pons and Latapy [107] use random walks to measure the similarity between vertices. In another line of work, Infomap [110] finds the shortest multilevel description of a *random walker* to get a hierarchical clustering of the network.

The previous methods, hierarchically nested or else, do not take into account the fact that communities in networks may overlap [105]. Palla et al. [105], propose the *Clique Percolation Method*, a local approach based on *k-cliques*. Overlaps between communities are allowed as a given node can be part of several *k-clique* percolation clusters at the same time. A revolutionary idea in overlapping community detection was introduced in two approaches that were developed almost simultaneously [7, 41]. The core of these approaches is that instead of focusing on grouping nodes, communities should be formed by considering groups of links. This allows for a natural incorporation of overlaps between communities while also retaining a hierarchical community structure. Ahn et al. [7] additionally report a comparison of their proposed algorithm with previous approaches, proving that it outperforms all of them.

Later research efforts focused on providing more scalable approaches. Coscia et al. [36] use *egonet* analysis methods and propose *DEMON* that allows nodes to vote for the communities they see locally in an effort to improve the quality of overlapping partitions. Yang and Leskovec [123] report that, contrary to previous belief, community overlaps are more densely connected than the non-overlapping parts. This relaxes the assumption that governed all previous efforts on overlapping community detection. Building on their empirical observations, they also propose *BIGCLAM* [125], a community detection method that uses matrix factorization to detect communities. *BIGCLAM* requires as an input the number of

communities to look for, or else should be guided with the minimum and maximum number of communities as well as the number of tries it should make. Gleich and Seshadhri [53] formalized the problem of community detection as finding vertex sets with small *conductance*, where conductance of a cluster is a measure of the probability that a one-step random walk starting in that cluster, leaves the cluster. They proposed the use of personalized PageRank vectors to identify communities with good conductance scores. A similar approach is investigated in [119], where a number of alternative seeding phases before the use of personalized PageRank vectors is examined. However, minimizing conductance leads to the identification of dense areas of a network as single communities, when they are in fact overlapping parts of multiple communities [126]. These approaches are more efficient than previous overlapping methods but fail to handle massive scale graphs.

Recent approaches depart from the direction of detecting communities on the *global* graph structure. Instead, they detect *local* communities in time functional to the size of the community, and provide support for large scale graphs. Kloster and Gleich [70] propose a deterministic local algorithm to compute heat kernel diffusion and study the communities it produces. The authors compare with PageRank diffusion on real-world datasets and report that their approach is able to detect smaller, more accurate communities, with slightly worse conductance. Li et al. [79] propose LEMON that uses seeds to perform short random walks and form an approximate invariant subspace termed *local spectra*. Then, LEMON looks for the minimum 1-norm vector in the span of this *local spectra* such that the seeds are in its support. Building on the findings of LEMON, He et al. propose LOSEP [61] that is additionally able to detect small communities and performs better when initiated with a single seed. In another line of work, Metwally et al. [98] employ general purpose clustering algorithms to detect click rings that launch advertising traffic fraud attacks. However, their techniques are not applicable on *single* graphs, as they use *multi-faceted* graphs, where each *facet* is a set of edges that represents the relationships between the nodes in a specific context. Our approach focuses on local communities but employs hierarchical clustering of pairs of links in the egonet of a target node, using *tie strength* measures that effectively handle networks with dense overlapping parts of communities. Thus, we efficiently reveal a more accurate hierarchical community structure in large scale networks.

A preliminary version of our work appeared in [80]. In this extended version:

- We delve into the merits of employing dispersion-based measures with regard to the hierarchical community structure provided by our algorithm. We show that such measures result in richer hierarchical structures through experimentation on our entire dataset.
- We propose a sampling technique that reduces the search space of our algorithm by focusing on the most *involved* nodes of an egonet. This technique allows us to maintain the efficiency of our algorithm in cases when target nodes exhibit large degrees in the network.
- We compare our sampling technique against a random sampling technique and show that our approach is very effective.

- We carry out the entire range of our experimentation using our sampling technique and report updated results with regards to both accuracy and execution time.

In another line of work [81], we apply community detection via seed set expansion on graph-streams. Our approach, termed CoEuS allows for local community detection without a need for in-memory representation of the network, as we employ a one-pass streaming approach and effectively determines the size of communities automatically. However, CoEuS requires more input target nodes than LDLC.

3.5 Conclusion

In this chapter, we propose and develop LDLC, a novel local community detection algorithm for large scale graphs. LDLC focuses on the egonet of a target node in the network and performs hierarchical agglomerative clustering on the egonet's pairs of links. We investigate measures that evaluate the strength of ties in networks, building on the notion that mutual neighbors of nodes may be or may be not well interconnected. The nodes involved in ties that belong in the second category, act as connector nodes between overlapping communities. Therefore, in a hierarchical approach they should be considered for grouping when the higher levels of the respective dendrogram are forming. We achieve that, by using the recursive dispersion measure to balance the similarity of two links and prioritize the grouping of pairs of links with mutual neighbors that function in a single context. Consequently, our approach is able to handle overlapping communities appropriately and provides increased accuracy, while also revealing the rich hierarchical structure of the communities of a node in the network. We compare LDLC with three state-of-the-art local community detection methods to highlight the effectiveness of our approach when handling overlapping areas of multiple communities. Moreover, we examine the accuracy of all algorithms against ground-truth communities and find that LDLC significantly outperforms all of them for a wide range of publicly available networks. Our timing experiments showcase that LDLC additionally offers improved efficiency and scales to large scale graphs. Finally, we discuss the merits of employing dispersion-based measures, as well as applying a sampling technique we introduce on the egonets of target nodes.

4. COMMUNITY DETECTION VIA SEED SET EXPANSION ON GRAPH STREAMS

Graph structures attract significant attention as they allow for representing entities of various domains as well as the relationships these entities entail. Real-world networks are commonly portrayed using graphs and are often massive. Despite their size, such networks exhibit a high level of order and organization, a property frequently referred to as community structure [51]. Nodes tend to organize into densely connected groups that exhibit weak ties with the rest of the graph. We refer to such groups as communities, whereas the task of identifying them is termed *community detection*.

Community detection is a fundamental problem in the study of networks and becomes more relevant with the prevalence of online social networking services such as Twitter and Facebook. Identifying the social communities of an individual enables us to perform recommendations for new connections. Moreover, by better understanding the membership of an individual to various organizational groups, we can provide more informative and engaging social network feeds. In addition to social networks, community detection is successfully applied to numerous other types of networks, such as biological or citation networks. In the former, we are particularly interested in inferring communities of interacting proteins, whereas in the latter we wish to uncover relationships between disciplines or the citation patterns of authors [45].

In the last two decades a plethora of community detection methods has been proposed. Initially, the focus has been on non-overlapping communities [17, 34, 101, 107]. More recent approaches, however, allow for nodes to belong to more than one community [7, 41, 53, 119, 123, 125]. Still, these approaches are not applicable to the massive graphs of the Big Data era, as they focus on the *entire* graph structure and do not scale with regards to both execution time and memory consumption. Recent efforts manage to scale as far as execution time is concerned by focusing on the local structure and expanding exemplary seeds-sets into communities [70, 79, 61, 80]. Such a seed-set expansion setting can be applied to numerous real world applications, e.g., given a few researchers focusing on Big Data we can use a citation network to detect their colleagues in the same field. However, the space requirements of such algorithms rapidly become a concern due to the unprecedented size now reached by real-world graphs. The latter have become difficult to represent in-memory even in a distributed setting [84].

An increasingly popular approach for massive graph processing is to consider a *data stream model*, in which the stream comprises the edges of a graph [97]. This is a new direction in the field of community detection and to the best of our knowledge no prior approach has considered such a setting without imposing restrictions on the order in which edges are made available [62, 128]. In this chapter, we propose CoEuS, a novel community detection algorithm that is fully applicable on graph streams. Figure 22 depicts such a graph stream whose edges arrive at no particular order. CoEuS is initialized with seed-sets of nodes that define different communities, such as the three sets depicted with the circles of Figure 22. As edges arrive, we can process them but we cannot afford to

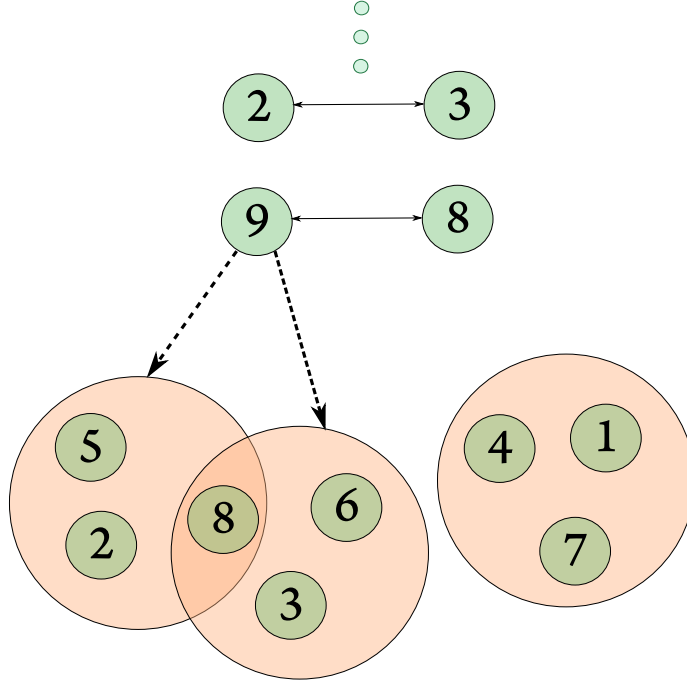


Figure 22: A stream comprising the edges of an undirected graph and a set of communities initialized with a few seed nodes. For every edge of the stream we wish to evaluate whether the adjacent nodes belong to the communities we examine.

keep them all in-memory. Therefore, CoEuS maintains rather limited information about the adjacent nodes of each edge and their participation in the communities in question. This information is kept using probabilistic data structures to further reduce the memory requirements of our algorithm. In addition to our original idea for community detection in graph streams, we propose two algorithms to enhance the effectiveness of CoEuS. The first one focuses on better quantifying the quality of each edge w.r.t. to a community. The second one is a novel clustering algorithm that allows for automatically determining the size of the resulting communities, in spite of the absence of the graph structure.

Our experimental results on various large scale real-world graphs show that CoEuS is extremely competitive with regard to *accuracy* against approaches that employ the entire graph structure and cannot operate on graph streams. More specifically, CoEuS can process with just a few MBs, graphs that prior approaches fail to handle on a machine with 16GB of RAM. Moreover, CoEuS is able to derive the communities in question inordinately faster. For instance we show that CoEuS is more than 17 times faster for the largest graph we could process with previously suggested approaches. More importantly, CoEuS is able to return its resulting communities *on demand* at any time as we process the graph stream. This is particularly important, as even if we could afford to use space linear to the number of a graph's edges, no other approach is able to update communities as new edges arrive with no additional *significant* computational cost.

In summary, we make the following contributions:

- We propose CoEuS, a novel community detection algorithm that can operate on a graph stream. To the best of our knowledge this is the first community detection algorithm that uses space sublinear to the number of edges and does not impose any restrictions on the order in which edges arrive in the stream.
- We develop a variation of our algorithm to better quantify the quality of each edge w.r.t. a community and verify that it improves the accuracy of CoEuS impressively.
- We suggest a novel clustering algorithm that allows for automatically determining the size of the resulting communities of CoEuS.
- We experimentally evaluate the accuracy of our algorithm and show that it is extremely competitive with prior approaches that cannot operate on graph streams and require the presence of the entire graph structure. In addition, we show that both the execution time and space requirements of CoEuS are astonishingly low.

4.1 Community Detection via Seed-Set Expansion on Graph Streams

This section first formulates the problem we target in this work. Then, we discuss the space requirements of our algorithm, and present our novel approach for streaming community detection. Lastly, we propose two enhancements to our algorithm, that greatly improve its effectiveness and efficiency.

4.1.1 Problem formulation

Consider a streaming sequence of unique unordered pairs $e = \{u, v\}$ where $u, v \in V$. Such a stream $S = \langle e_1, e_2, \dots, e_m \rangle$ naturally defines an undirected unweighted graph $G = \{V, E\}$, where V is the set of vertices $\{v_1, v_2, \dots, v_n\}$ and E is the set of undirected edges $\{e_1, e_2, \dots, e_m\}$. Given a community seed-set $K = \{k_1, k_2, \dots, k_l\} \in V$, our goal is to extend it to a community C . Figure 22 shows such a graph stream with two visible arriving edges, and three seed-sets that are to be extended to communities.

A community is generally thought to be a set of nodes of a graph that are tightly connected to each other and maintain very few ties with the rest of the graph's nodes [101]. However, there is no universal definition of what communities are, and thus, there exists a plethora of different approaches in detecting them. A widely used [53, 79, 92, 119] quality function in the field of community detection is the *conductance* of a community. More specifically, conductance $\phi(C)$ of a community C is formally defined as:

$$\phi = \frac{adj(C, V \setminus C)}{\min(adj(C, V), adj(V \setminus C, V))}, \quad (4.1)$$

where:

$$adj(C_i, C_j) = |\{(u, v) \in E : u \in C_i, v \in C_j\}|.$$

Several methods try to detect communities exhibiting low conductance, in an effort to come up with a set of nodes with a limited number of ties to nodes outside of the community. However, tracking the conductance of all possible communities as we process the edges of stream is inefficient with regard to both time and space. Instead, we introduce here *community participation* $cp(u)$ of a node u in a community, that measures a node's u participation level in a community. In particular the community participation of node u in community C is defined as:

$$cp(u) = \frac{|\{(u, v) \in E : v \in C\}|}{|\{(u, v) \in E\}|}, \quad (4.2)$$

i.e., community participation of a node in a community is the fraction of its adjacent nodes in the graph that are part of the community. Our intuition is that including nodes exhibiting high values of cp to a community C will result to a low value of *conductance* for the community. To this end, our approach employs Eq. (4.2) to detect communities. We note, however, that the use of a particular quality function, such as conductance or community participation, does not hinder in any way the evaluation of our approach against community detection methods using different quality functions. Such an evaluation is possible, as there exist publicly available networks with ground-truth communities. Our experimental setting features numerous such networks that allow us to verify the efficiency of different algorithms.

4.1.2 Space complexity

The motivation behind graph stream algorithms lies in the fact that many real-world networks nowadays reach sizes that are simply too large. Thus, graph algorithms are unable to store and process the respective graphs in their entirety [97]. In contrast, graph stream algorithms process a stream comprising the edges of the graph in the order in which these edges arrive over time using *limited memory*.

Earlier streaming community detection approaches have successfully revealed the community structure of graphs streams with limited memory requirements. However, the latter were minimized at the expense of additional constraints on the order in which the edges of the stream arrive. In particular, Yun et al. [128] consider a data stream model in which rows of the adjacency matrix of the graph are revealed sequentially. In such a setting we are aware at any moment of all neighbors of certain nodes. Thus, we can apply community detection with partial information on the subgraphs as they are revealed to us. Memory requirements are kept low as we can discard at each step all information that was made available in earlier steps. Moreover, SCoDa [62] considers a setting in which the edges of the graph stream arrive as if we picked them uniformly at random. This allows for estimating whether a newly arriving edge is an intra-community or an inter-community edge and enables SCoDa to achieve space complexity that is linear to the number of the graph's nodes. However, picking an edge of the graph uniformly at random requires that we already possess the graph in its entirety. The latter assumption is not true for graph streams.

We consider a more practical scenario of a streaming setting in which the edges of a graph arrive at no particular order. Thus, we cannot discard information in ways similar to the techniques in [62, 128]. We focus instead on estimating the participation level of each node of the graph in each of the communities we examine, according to Eq. (4.2). In this context, we need to keep track of the following aspects as we process a graph stream:

- a) **degrees**: the total number of nodes each node in the graph is adjacent to, i.e., the degree of each node in the graph,
- b) **community degrees**: the degree of each node in the subgraph of each community, and
- c) **communities**: the nodes that comprise each community we examine.

Essentially, if $|C'|$ is the number of communities we examine, the above information can be kept in-memory using $|C'|$ sets (one set for each community we examine), and $|V|(|C'| + 1)$ integers ($|C'| + 1$ integers for each node of the graph). More specifically, we need one integer for the degree of each node in the graph, and one integer for each community we examine to hold the degree of the node in the subgraph that comprises the nodes of the community. Given that the number of communities we examine can be large we decided to use Count-Min sketches to hold the $|C'| + 1$ integers.

The Count-Min sketch [35] is a well-known sublinear space data structure for the representation of high-dimensional vectors. Count-Min sketches allow fundamental queries to be answered efficiently and with strong accuracy guarantees. They are particularly useful for summarizing data streams as they are capable of handling updates at high rates. A Count-Min sketch uses a two-dimensional array of w columns and d rows, where $w = \lceil \frac{\epsilon}{\delta} \rceil$, $d = \lceil \frac{\ln(1)}{\delta} \rceil$, and the error in answering a query is within a factor of ϵ with probability δ . A total of d pairwise independent hash functions is also used, each one associated with a row of the array.

Figure 29 illustrates the update process of a Count-Min sketch for our specific problem. Consider that an edge (u, v) arrives in the stream and as $v \in C$ we need to increase the number of adjacent nodes u has in community C . Thus, we form a unique *id* using the indices of the node and the community and create an update $(i : u, 1)$, indicating that the count of $i : u$ should be incremented by 1. The array *count* is updated as follows: for each row j of *count* we apply the corresponding hash function to obtain a column index $k = h_j(i : u)$ and increment the value in row j , column k of the array by 1, i.e., $count[j, k] + 1$. This allows us to retrieve at any time an (over)estimation of the count of an event $i : u$ using the least value in the array for $i : u$, i.e., $\hat{a}_{i:u} = \min_j count[j, h_j(i : u)]$.

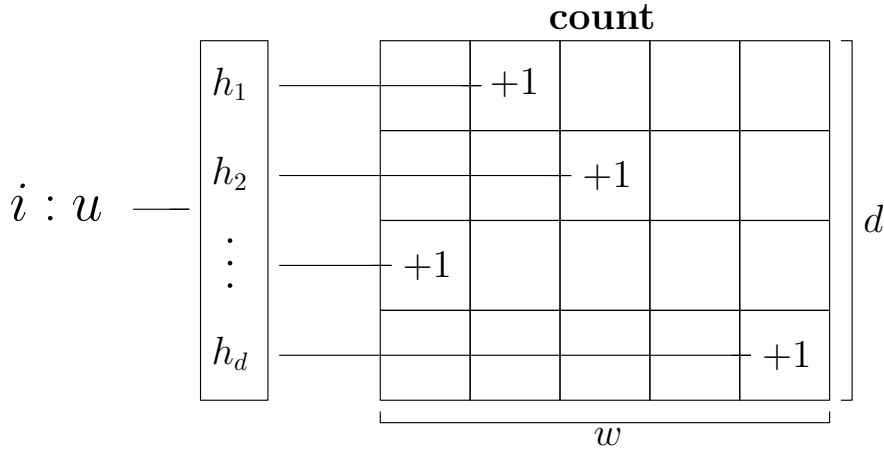


Figure 23: Count-Min sketch update process.

4.1.3 Our CoEuS Algorithm for Streaming Community Detection

In this section, we discuss the details of our algorithm for streaming community detection, termed CoEuS.¹ The pseudocode of CoEuS is given in Algorithm 3.

Input/Output: CoEuS takes as its input two parameters: The first one is a set of community seed-sets $K' = \{K_1, K_2, \dots, K_s\}$, where each $K_i = \{k_1, k_2, \dots, k_l\} \in V$. The second one is a stream $S = \langle e_1, e_2, \dots, e_m \rangle$, where $e_i \in E$, and E is the set of edges of the undirected graph $G = \{V, E\}$ that S defines. CoEuS processes the edges of the graph stream to extend each of the seed-sets in K' to a community. Thus, the output of CoEuS is the set of communities $C' = \{C_1, C_2, \dots, C_s\}$, with community C_i corresponding to seed-set K_i . This output is available *on-demand* at all times as we process the stream.

Initialization: The first step of CoEuS is to initialize the communities using the seed-sets (Lines 2-6). This is a simple procedure in which we create an additional set for each of the community seed-sets, to hold the nodes of the respective communities. The seed-sets and the community sets enable us to query efficiently at any time whether a node is a seed or a member of a community. Using Figure 22 as an example, consider that we wish to detect three communities. CoEuS is initiated with three seed-sets that describe these communities, namely $\{2, 5, 8\}$, $\{3, 6, 8\}$, and $\{1, 4, 7\}$. In this setting, CoEuS creates three community sets that comprise these nodes.

Stream processing: After initializing the communities, CoEuS is ready to process the stream (Lines 7-21). Due to the size of the network, we consider that maintaining the whole graph is prohibitive. Instead, we focus on the degree of each node in the graph as well as its degree in each community, and the nodes that comprise each community. For each incoming edge of the stream, we first increment by 1 the degree of each of the adjacent nodes in the graph (Lines 8-9). Then, for each community we wish to extend, we examine whether each of the adjacent nodes is a member of the community. If this is

¹In Greek mythology Coeus was the Titan of intellect, the axis of heaven around which the constellations revolved and probably of heavenly oracles.

Algorithm 3: CoEuS(S, K')

```

input : A set of community seed-sets  $K'$ ,
        and a graph stream  $S$ .
output: A set of communities  $C'$ .
1 begin
2   foreach  $K \in K'$  do
3      $C \leftarrow \{\}$ ;
4     foreach  $k \in K$  do
5        $C[k] = 1$ ;
6      $C'.put(C)$ ;
7   while  $\exists(u, v) \in S$  do
8      $degree_V[u] + = 1$ ;
9      $degree_V[v] + = 1$ ;
10    foreach  $C \in C'$  do
11      if  $u \in C$  then
12         $degree_C[v] + = 1$ ;
13      if  $v \in C$  then
14         $degree_C[u] + = 1$ ;
15      if  $u \in C$  then
16         $C.put(v)$ ;
17      if  $v \in C$  then
18         $C.put(u)$ ;
19       $processedElements + = 1$ ;
20      if  $processedElements \bmod W == 0$  then
21         $C \leftarrow prune(C, s, degree_V, degree_C)$ ;

```

the case, we increment the community degree of the other node. In addition, if the other node is not a member of the community, we add the node to the community (Lines 11-18). Going back to the example of Figure 22, with the arrival of edge $\{9, 8\}$ CoEuS will first increment the degree of both nodes 8 and 9 by 1. After that, CoEuS will examine for every community if nodes 9, or 8 are members of the community. This is true regarding node 8 for two communities. Therefore, CoEuS will increment the community degree of node 9 by 1 for both communities. In addition, CoEuS will add node 9 to both communities that node 8 belongs to.

As the diameters that real-world networks exhibit are small and in many cases decrease as the network grows [76], the communities CoEuS detects through the above process often grow considerably in size. However, we wish to focus on nodes that are tightly connected to each other for each community. To this end, we additionally consider a window of size W . During a window, the communities may grow freely in size, as new edges arrive.

Algorithm 4: pruneComm

```

1 Function pruneComm ( $C, s, degree_V, degree_C$ )
2   minheap  $\leftarrow []$ ;
3   foreach  $c \in C$  do
4      $cp(c) = \frac{degree_C[c]}{degree_V[c]}$ ;
5     if minheap.size() <  $s$  then
6       minheap.push( $(c, cp(c))$ );
7     else if  $cp(c) > minheap[0]$  then
8       minheap.pop();
9       minheap.push( $c, cp(c)$ );
10  return set(minheap);

```

However, when the window closes, CoEuS prunes all communities and keeps only the most *highly involved* nodes of each community (Lines 20-21). This process is detailed with Algorithm 4 and function *pruneComm*, which uses Eq. 4.2 to evaluate each node's participation level to community C . For each node $c \in C$ we calculate $cp(c)$ (Line 4). Then, we use a min-heap to hold the nodes with the highest community participation values. If the size of the min-heap is currently below s , i.e., the size at which we want to prune the community, we push the node and its community participation value to the min-heap (Lines 5-6). Otherwise, we examine whether the community participation value of the current node is higher than that of the minimum value in the min-heap (Line 7). If so, we pop the latter out of the min-heap, and push the current node in it (Lines 8-9). The function outputs a set that comprises the nodes that remained in the min-heap after examining all the nodes of the community (Line 10). CoEuS prunes communities to a size of 100, as related studies state that quality communities do not surpass 100 nodes [127]. Moreover, CoEuS uses a window of 10,000 edges, a value derived via extensive exploratory testing that consistently works well.

Termination: CoEuS can be stopped at will, as the member nodes of each community are available at any moment. In the pseudocode of Algorithm 3, we consider a finite stream and CoEuS terminates when all elements of the stream have been processed. However, CoEuS can handle infinite streams as well. Besides, all nodes of each community are associated with a community participation value that CoEuS may include in its output. The higher this value is, the more certain we are that the respective node is part of the community.

4.1.4 Reckoning in edge quality w.r.t. each community

For every edge of a graph stream, CoEuS examines whether an adjacent node is a member of a community. If so, CoEuS increments the respective community degree of its adjacent node by 1. This procedure takes under consideration the number of adjacent

Algorithm 5: addToCommByEdgeQuality

```

1 Procedure addToCommByEdgeQuality
2   foreach  $C \in C'$  do
3     if  $u \in C$  then
4        $\text{degree}_C[v] + = \frac{\text{degree}_C[u]}{\text{degree}_v[u]}$ ;
5     if  $v \in C$  then
6        $\text{degree}_C[u] + = \frac{\text{degree}_C[v]}{\text{degree}_v[v]}$ ;
7     if  $u \in C$  then
8        $C.\text{put}(v)$ ;
9     if  $v \in C$  then
10       $C.\text{put}(u)$ ;

```

nodes each node has in a community to estimate the participation of the node in the latter. However, we do not consider the level of involvement of the adjacent nodes in the community. All nodes included in a community provide increments of 1 to all of their adjacent nodes, regardless of how well-established the former are in the community. We discuss here a variation of CoEuS to improve over a simple community degree measure by taking into account the edge quality of nodes w.r.t. each community. This variation is reminiscent of PageRank, that employs the network's link structure to improve over the in-degree measure [25].

Our variation employs Eq. (4.2), instead of incrementing the community degree of a node by 1 for all of its adjacent nodes that are members of a community. Eq. (4.2) is equal to the fraction of the adjacent nodes of a node that are also members of the community in question. This fraction is essentially an estimation of the probability that a one-step random walk starting from the node will lead to a node that is a member of the community in question. Therefore, the value of Eq. (4.2) for each node grows with its *involvement* in the community. If this value is high, then the probability that an adjacent node is a member of the community is also high. Incrementing the community degree of a node using the value of Eq. (4.2) of its adjacent node instead of 1, enables CoEuS to *maintain its focus* in the community. In particular, this variation favors nodes that are adjacent to well-established members of the community, as such nodes receive a significant increment to their community degree. In contrast, nodes that exhibit low values of Eq. (4.2) provide insignificant increments to the participation levels of their adjacent nodes. Thus, the potential of nodes exhibiting low values of Eq. (4.2) to shift the focus of the community is limited.

Algorithm 5 details the above outlined approach, and can replace Lines 9-17 of Algorithm 3. The difference in functionality is in Lines 4 and 6 of Algorithm 5, which increment the participation level of a node in the community using an estimation of Eq. (4.2) for the respective adjacent node.

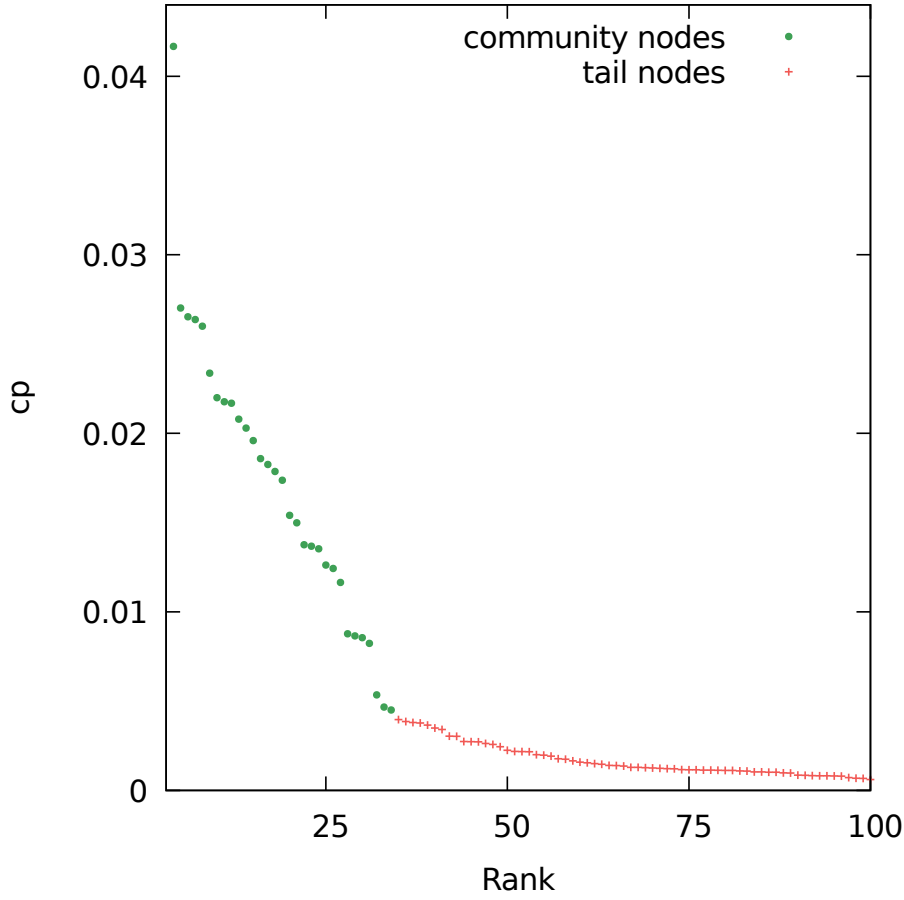


Figure 24: Ranking of nodes according to their community participation values and the partitioning that Algorithm 6 makes to come up with a community automatically for a random citation network community.

4.1.5 Size of the community

CoEuS associates each node included in the expanded community with a community participation value. However, the size of an actual community might be smaller than the one CoEuS examines. Therefore, CoEuS needs to additionally solve the issue of determining the size of a community automatically and removing any irrelevant nodes.

Algorithm 6 details *dropTail*, a procedure that identifies nodes irrelevant to the community formed with CoEuS and removes them. In this regard, *dropTail* utilizes the community participation values of the nodes included in the community, and allows for fully *automatic, on-demand* removal of irrelevant nodes. More specifically, irrelevant nodes exhibit weak ties to the actual community and thus, their respective community participation values are insignificant when compared to the values of other nodes included in the community. This is evident in Figure 24 that illustrates the community participation values of nodes included in a community of a real-world graph, as found by CoEuS. We observe that ordering nodes according to their community participation values results to a clearly visible

Algorithm 6: dropTail

```

1 Procedure dropTail
2    $\hat{C} \leftarrow \text{reverseSort}(C)$ ;
3    $\text{totalDifference} \leftarrow 0$ ;
4    $\text{previous} \leftarrow 0$ ;
5   foreach  $c \in \hat{C}$  do
6     if  $\text{previous} > 0$  then
7        $\text{totalDifference} \leftarrow \text{cp}(c) - \text{previous}$ ;
8        $\text{previous} \leftarrow \text{cp}(c)$ ;
9    $\text{averageDifference} \leftarrow \frac{\text{totalDifference}}{\hat{C}.\text{size()}-1}$ ;
10   $\text{previous} \leftarrow 0$ ;
11  foreach  $c \in \hat{C}$  do
12    if  $\text{previous} > 0$  then
13       $\text{difference} \leftarrow \text{cp}(c) - \text{previous}$ ;
14       $\text{previous} \leftarrow \text{cp}(c)$ ;
15      if  $\text{difference} < \text{averageDifference}$  then
16         $\hat{C}.\text{remove}(c)$ ;
17    else
18      break;

```

tail. The distribution of community participation values varies, depending on both the graph and the community in question. Thus, setting a constant *threshold* value and discarding nodes that exhibit lower community participation values to remove such tails is not an option. Instead, we need to adjust to each particular community and isolate the nodes that belong to the tail through *clustering*. To do so, *dropTail* calculates the average distance between two consecutive nodes with regard to their ranking by their associated community participation values (Lines 5-9). Then, *dropTail* iteratively examines the value distance of two nodes in this ranking, starting from the last node. When this distance is found to be larger than the average distance of nodes, *dropTail* stops, as it has spotted a significant gap between the values of two consecutive nodes (Lines 11-18). The result of this process for our example is also illustrated in Figure 24. The average distance between consecutive nodes w.r.t. the ranking by community participation value is 0.00043. The first node from the end that exhibits a gap larger than that from its predecessor is the one ranked 35th. Therefore, *dropTail* considers that the tail of irrelevant nodes begins from the 35th node (depicted using red crosses), and the actual community is formed by the first 34 nodes (depicted using green dots).

We note that seed nodes exhibit relatively large community participation values and their inclusion in this process is experimentally found to include more relevant nodes in the tail. Thus, *dropTail* does not consider seed nodes, but only those nodes that have been added

Table 6: Graphs of our dataset reaching up to 1.8 billion edges.

Graphs	Type	Nodes	Edges	Average Degree
Amazon	Co-purchasing	334, 863	925, 872	2.76
DBLP	Co-authorship	317, 080	1, 049, 866	3.31
Youtube	Social	1, 134, 890	2, 987, 624	2.63
LiveJournal	Social	3, 997, 962	34, 681, 189	8.67
Orkut	Social	3, 072, 441	117, 185, 083	38.14
Friendster	Social	65, 608, 366	1, 806, 067, 135	27.53

to the community during its expansion process.

4.2 Experimental Evaluation

We proceed by evaluating the performance of CoEuS on a range of networks from various domains. Our experiments measure the impact of the novel techniques of our algorithm and feature comparisons against state-of-the-art community detection approaches that use the entire graph. We first discuss the specification of our experimental setting. Then, we proceed with the evaluation of CoEuS by answering the following questions:

- What is the impact of employing the edge quality variation of CoEuS with regard to its *accuracy* in detecting communities?
- Is CoEuS able to automatically determine the *size* of a detected community using our novel *dropTail* clustering procedure?
- Is the accuracy achieved through CoEuS *comparable* to that of state-of-the-art local community detection methods that use the entire graph?
- What are the merits of CoEuS with regard to *execution time* as well as *space efficiency* when compared to prior efforts?

4.2.1 Experimental Setting

Our dataset comprises the six publicly available social, co-authorship, and co-purchasing networks listed in Table 6.² The respective graphs reach up to 1.8 billion edges and possess ground-truth communities which allow for quantifying the accuracy of community detection algorithms. To ensure a fair comparison against a state-of-the-art algorithm [79] we have adopted its experimental setting and use the top-5000 ground-truth communities of each network that possess the highest quality according to [77], after enforcing a minimum community size of 20.

²<https://snap.stanford.edu/data/#communities>

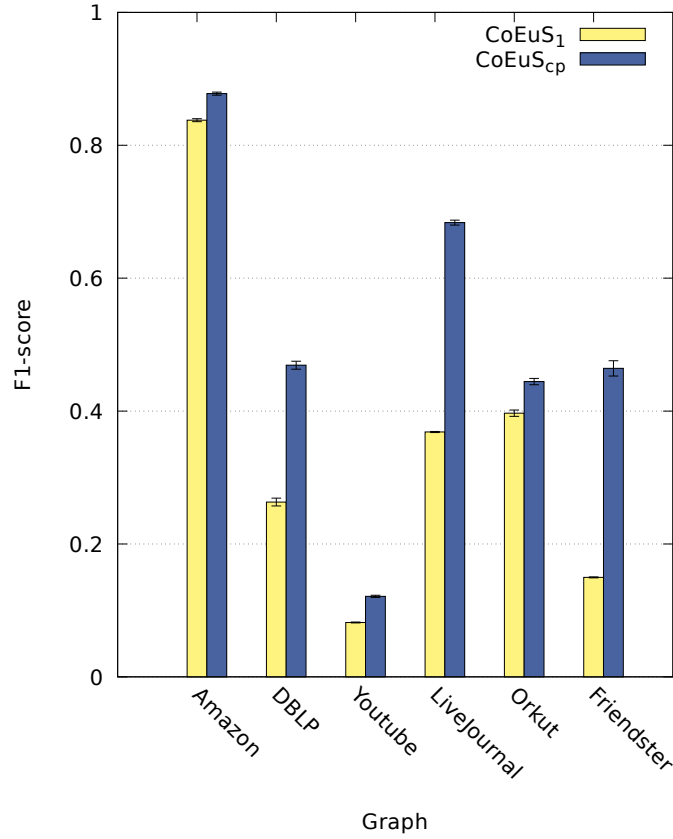


Figure 25: F1-score comparison for CoEuS when incrementing community degree by 1 (CoEuS₁) and by community degree of the adjacent node (CoEuS_{cp}). The variation of CoEuS_{cp} clearly improves the F1-score for all graphs our dataset. The improvement is impressive for graphs *orkut* and *dblp*.

We implemented CoEuS using Java 8. Our implementation, as well as execution tests that enable the reproducibility of our results are publicly available.³ The experiments were carried out on a machine with an Intel® Core™ i5-4590, with a CPU frequency of 3.30GHz, a 6MB L3 cache and a total of 16GB DDR3 1600MHz RAM and the Linux Xubuntu 14.04.5 (Trusty Tahr) x86 64 OS. To maintain node and community degrees we employ in all our experiments Count-Min sketches. The latter are initialized with the following parameters: *i*) $d = 7$, and *ii*) $w = 200,000$, so that we obtain 99% confidence that $\epsilon < 0.00001$. Our evaluation assumes that three random nodes of each ground-truth community are provided to each algorithm as an input seed-set. To measure the accuracy of each algorithm we use the average F1-score achieved for the communities of each graph. All results reported are averages of multiple executions (for various random seed-sets and permutations of the order of edges) and are accompanied with their respective 95% confidence intervals.

4.2.2 Impact of the Edge Quality Variation

We begin by studying the behavior of CoEuS when considering our two different techniques of incrementing the community degree of a node. We denote our algorithm as CoEuS₁ when the community degree of each node is incremented by 1 for every adjacent node found in the community (Algorithm 3) and CoEuS_{cp} when the community degree of each node is incremented by the community degree of the adjacent node (Algorithm 3 with the edge quality variation of Algorithm 5).

Figure 25 illustrates a comparison between CoEuS₁ and CoEuS_{cp} on their accuracy on detecting the ground-truth communities of the networks in Table 6. It is clearly evident that the edge quality variation we introduce to CoEuS heavily impacts the ability of our algorithm to accurately retrieve the members of a community. We see that CoEuS_{cp} achieves an increased F1-score compared to CoEuS₁ for all graphs included in our dataset. The improvement for graphs *dblp*, *livejournal*, and *friendster* is particularly impressive, increasing from 0.263 to 0.469, from 0.369 to 0.684, and from 0.15 to 0.464, respectively. Significant improvements with regard to F1-score are also achieved for graphs *orkut*, *amazon*, *youtube*, for which the F1-scores increase from 0.397 to 0.444, from 0.838 to 0.878, and from 0.082 to 0.121, respectively.

These results verify emphatically that the variation of Algorithm 5 successfully *favors* nodes that are actual members of the community in question, and *penalizes* nodes that exhibit weak ties with the community, when incrementing their respective community degrees. Thus, the resulting communities are much more accurate than the ones detected when relying entirely on Algorithm 3.

We note that this experiment considers for both CoEuS₁ and CoEuS_{cp} as size of each resulting community the size of the respective ground-truth community. We now proceed with the evaluation of our automatic size determination clustering algorithm (Algorithm 6), as we cannot assume that the size of a community is known a priori.

4.2.3 Evaluation of Automatic Size Determination

Community detection via seed-set expansion calls for a stopping criterion for the expanding process. CoEuS employs two techniques to limit the expansion of each community. The first one, i.e., Algorithm 4, is a pruning procedure that is periodically applied to reduce the size of the community. The second one, i.e., Algorithm 6, is a novel clustering algorithm that is applied on the resulting community of CoEuS to separate the nodes that exhibit weak ties with the community and should be removed. In this experiment we evaluate the effectiveness of our clustering algorithm by comparing the average F1-score of CoEuS_{cp} and CoEuS_{cp-auto}; for CoEuS_{cp} we assume that the size of each community is known a priori, whereas CoEuS_{cp-auto} automatically derives the size of a community using Algorithm 6.

³<https://github.com/panagiotisl/CoEuS>

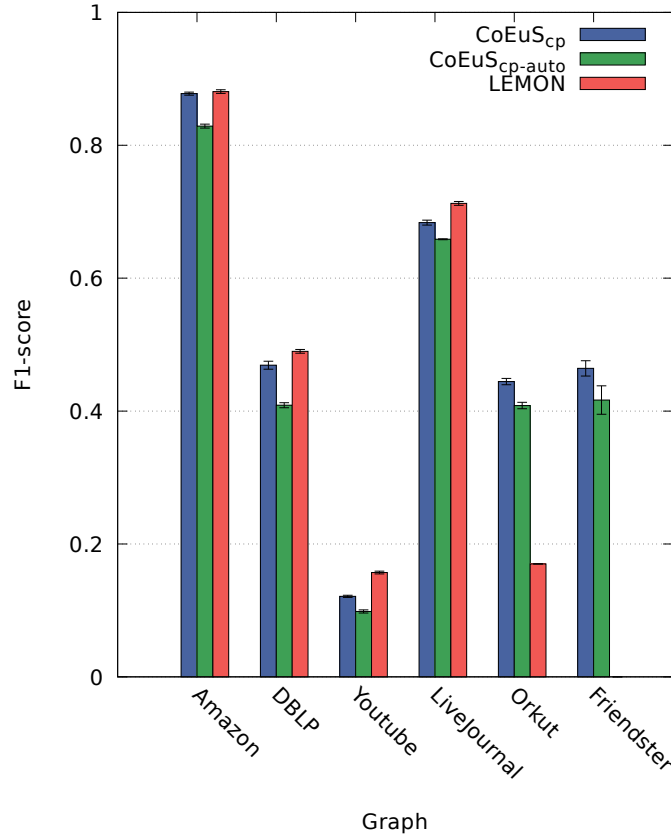


Figure 26: F1-score comparison between LEMON and CoEuS.

The first two bars of Figure 26 illustrate the F1-scores achieved through CoEuS_{cp} and CoEuS_{cp-auto} when detecting the ground-truth communities of the networks of our dataset. As we can see, our *dropTail* clustering algorithm is able to offer impressive performance, as the difference between the F1-score of CoEuS_{cp} is in most cases negligible. More specifically, the difference in F1-score is under 0.06 for all networks of our dataset and 0.04 on average. This result strongly highlights the effectiveness of Algorithm 6 to determine the size of a community automatically. We also note that Algorithm 6 is extremely efficient, both time- and space-wise, requiring only two passes over each resulting community (about 100 nodes), without any access to the graph's elements. In contrast, other size determination techniques such as the one employed in [79] necessitate calculations of complex community quality measures like that of Eq. (4.1) for every possible size of each community and require the presence of the entire graph.

4.2.4 Comparison against state-of-the-art non-streaming local community detection algorithms

After evaluating the impact of Algorithms 5 and 6 on the accuracy of CoEuS, and having verified their effectiveness in detecting communities in a variety of networks, we now pro-

ceed with comparing our graph stream algorithm against state-of-the-art non-streaming local community detection algorithms. Our comparison focuses on LEMON as it is shown in [79] to outperform all [70, 71, 119], whereas the more recent SCODA [62] reports significantly lower F1-scores and does not allow overlaps.

4.2.4.1 F1-score comparison

We begin the comparison of $\text{CoEuS}_{\text{cp-auto}}$ with LEMON as far as their accuracy on detecting communities is concerned. We initialize LEMON with three random seeds of each ground-truth community of our dataset and report here averages of multiple executions. The third bar of Figure 26 illustrates the F1-scores achieved with LEMON for all six networks. Note, that we were unable to retrieve results for the two largest networks of our datasets due to LEMON's memory requirements. However, we include here the results reported for *orkut* in [79].

As we can see, using the entire graph, LEMON is usually able to slightly outperform $\text{CoEuS}_{\text{cp-auto}}$ for small graphs. However, our algorithm is extremely competitive w.r.t. accuracy, despite the fact that it operates on a graph stream setting. More specifically, The average F1-score difference of $\text{CoEuS}_{\text{cp-auto}}$ with LEMON for the four smaller graphs of our dataset is 0.061. Regarding *orkut*, $\text{CoEuS}_{\text{cp-auto}}$ is far more accurate achieving an F1-score of 0.408 against LEMON's 0.17. Finally, our algorithm is able to achieve a notable F1-score of 0.417 for the largest graph of our dataset, which LEMON fails to handle due to its size.

These results are particularly impressive as the graph stream setting that CoEuS adheres to, is much more restrictive than the setting LEMON and other prior seed-set expansion methods operate on. CoEuS processes each edge of the graph as it is becoming available and maintains very limited information for each node and community. Hence, it is surprising that our algorithm achieves comparable accuracy to methods that utilize the entire graph structure. Furthermore, it is evident in Figure 26 that our effective novel graph stream techniques enable CoEuS to easily scale to large graphs, which other community detection methods fail to handle.

4.2.4.2 Execution time and space efficiency comparison

Having shown that CoEuS is competitive or better than state-of-the-art non-streaming algorithms as far as accuracy is concerned, we now report results concerning *execution time* and *space* efficiency.

Table 7 illustrates a comparison on the execution time between $\text{CoEuS}_{\text{cp-auto}}$ and LEMON. Regarding $\text{CoEuS}_{\text{cp-auto}}$, we consider a streaming setting in which we process every edge of each graph to update our structures and expand the communities in question. Similarly for LEMON, we read the entire graph in-memory, as only then we can sequentially expand each community in question. We use only one CPU core for both algorithms as parallel execution is not an option for LEMON which is written in Python. The results concern the

Table 7: Execution time comparison between CoEuS and LEMON for all the graphs of our dataset. CoEuS is remarkably fast, even for the largest network of our dataset and clearly outperforms LEMON.

Graphs	CoEuS	LEMON
Amazon	0.0458 sec	3.1197 sec
DBLP	0.0575 sec	7.2756 sec
Youtube	0.176 sec	11.3834 sec
LiveJournal	1.573 sec	28.14 sec
Orkut	7.5171 sec	—
Friendster	158.6547 sec	—

average time needed for one community of each network. As we can see in Table 7, our algorithm is extremely faster. In particular, CoEuS_{cp-auto} is able to detect a community in the four smaller networks in time less than 2 seconds, whereas LEMON needs up to 28.14 seconds. Moreover, CoEuS is able to scale to networks reaching billions of edges requiring only an impressive total of 158.66 seconds for a community of the *friendster* network.

We note that even though these results clearly show that CoEuS is considerably faster than prior approaches, they are not indicative of CoEuS's speed in a real streaming setting. In particular, CoEuS is able to return the communities in question *on-demand* as we process the stream in *real-time*. The measurements reported in Table 7 additionally consider edge processing, which we expect that in a real-life setting will happen faster than edges are made available. In this regard, the actual response time of CoEuS in a streaming setting is *in the order of milliseconds* regardless of the size of the graph. Yet, the results of Table 7 indicate that CoEuS is a very attractive option even for non-streaming settings.

Table 8 shows a comparison between the two algorithms on their space requirements concerning the graph structure. CoEuS_{cp-auto} employs two Count-Min sketches which are initialized in our experiments to require the same space regardless of the graph. Therefore, the space requirements are independent of the size of the graph and depend only on the desired approximation quality of the sketches and the number of communities in question. In contrast, LEMON needs to maintain in-memory the entire graph structure, using adjacency lists. Both algorithms require additional space to hold the communities we seek; however, this space is linear to the number of the communities and fairly insignificant compared to the graph structure.

It is evident that the space requirements of CoEuS are remarkably low. Even the largest graph of our dataset, reaching up to 1.8 billion edges is handled appropriately with just 21.36MB. In contrast, the space requirements of LEMON grow with the number of edges of a graph. The largest graph we are able to handle is *livejournal* with 34 million edges for which more than 2,500MB are needed. The two largest graphs of our dataset result in memory errors.

Table 8: Comparison of space requirements between CoEuS and LEMON for all the graphs of our dataset. CoEuS uses two Count-Min sketches to hold a graph’s elements and therefore its requirements depend only on the desired approximation quality of the sketches. LEMON maintains the adjacency lists of a graph and thus requires significantly more space.

Graphs	CoEuS	LEMON
Amazon	21.36MB	155.74MB
DBLP	21.36MB	156.49MB
Youtube	21.36MB	457.62MB
LiveJournal	21.36MB	2,652.99MB
Orkut	21.36MB	—
Friendster	21.36MB	—

4.3 Related Work

Our work lies in the intersection of community detection over streaming graphs and local community detection via seed-set expansion. We outline here pertinent aspects of these two areas.

Local community detection via seed-set expansion: Numerous recent approaches depart from the direction of working on the entire graph structure. Instead, they focus on detecting *local* communities in time functional to the size of the community and are thus able to support large scale graphs. Such approaches usually operate using a seed-set of nodes which they expand to a community. Kloster and Gleich [70] propose a deterministic local algorithm to compute heat kernel diffusion and study the communities it produces when initiated with seed nodes. The authors compare with PageRank diffusion on large scale real-world graphs and report that using Heat Kernel diffusion they are able to detect smaller, more accurate communities, with slightly worse conductance. LEMON [79] also uses seeds to perform short random walks and forms an approximate invariant subspace termed *local spectra*. Then, LEMON looks for the minimum 1-norm vector in the span of this *local spectra* such that the seeds are in its support. To determine the size of the community, the authors of [79] employ the measure of conductance. In particular, they measure the conductance of the community as they increase its size, and stop at the first relative minimum conductance encountered. Both the above approaches are similar to our setting as they expand a seed-set of nodes into a community. However, neither of them is able to handle graph streams. LDLC [80] focuses on egonets of nodes in networks and performs hierarchical link clustering to detect *all* the overlapping communities of a node. In addition, LDLC uses a measure of *dispersion* to detect nodes that share multiple communities and thus avoids to group together overlapping parts of communities. Our CoEuS is different as it uses a seed-set of nodes and expands it into a single community.

Streaming community detection: Yun et al. [128] consider settings in which the size of the network is so large that maintaining the respective graph is prohibitive. Thus, they study the problem of clustering the nodes of a graph to communities in a streaming setting where rows of the adjacency matrix of the graph are revealed sequentially. They

propose an online algorithm with space complexity that grows sub-linearly with the size of the network. Our streaming setting does not assume that rows of the adjacency matrix are completely revealed to us. Instead, we consider that edges involving any node of the graph may arrive at any moment. Moreover, we are unaware of the size of the graph, which grows with time. Zakrzewska and Bader [131] propose a dynamic seed set expansion algorithm for community detection. In particular, they consider that edges may be inserted to or removed from the graph dynamically and detect the local community of a seed set by incrementally adjusting to the changes of the graph. The latter allows for faster execution compared to an algorithm that requires re-computation after every update at the cost of slightly worse community quality. Our approach is different as we assume that we cannot maintain the whole graph in-memory, whereas the incremental adjustments that [131] performs do impose such a requirement. Moreover, we suggest a significantly more cost-effective recomputation of the local community at every step. Hollocau et al. [62] consider an edge streaming setting and assign all the nodes of a graph to non overlapping communities using only two integers per node that hold: i) the node's degree, and ii) the current community index assigned to the node. Their work is heavily based on the observation that if we pick uniformly at random an edge of the graph, this edge is more likely to link nodes of the same community, than nodes from distinct communities. This is expected to be true as nodes tend to be more connected within a community than across communities, thus, if we process edges in a random order we expect many intra-community edges to arrive before the inter-community edges. However, this requires that we already hold the graph in its entirety and we are able to select its edges one by one uniformly at random. We operate on the more practical assumption that the edges of the graph arrive at no particular order.

4.4 Conclusion

In this chapter we propose and develop CoEuS, a novel graph stream community detection algorithm that expands seed-sets of nodes into communities. To the best of our knowledge CoEuS is the first streaming algorithm that performs community detection using space sublinear to the number of edges without imposing any restrictions in the order in which edges arrive in the stream. CoEuS processes a stream of edges and maintains limited information about the respective graph, concerning the nodes' degrees, the participation of nodes into communities and the nodes that comprise each community we seek. In addition to CoEuS, we propose two algorithms to improve the effectiveness of our approach. The first one places emphasis on the quality of an edge w.r.t. a community and is able to better preserve the focus of a community as the latter is expanding. The second one allows for automatic on-demand determination of the size of a community through a novel clustering technique, tailored to the needs of CoEuS.

We compare CoEuS with a non-streaming local community detection method that reportedly outperforms other recent approaches. Using large-scale networks from various domains we show that CoEuS is able to offer accuracy that is equivalent to or better than that of methods exploiting the entire graph, even though it operates on a graph stream. The

two algorithms we propose to enhance CoEuS, contribute enormously to its effectiveness and efficiency, by improving its accuracy and allowing for *real-time* determination of the size of each community. Furthermore, we examine the requirements of CoEuS and show that our algorithm is clearly superior than prior approaches with regard to both execution time and space used. Therefore, our CoEuS algorithm proves to be not only an extremely accurate graph stream algorithm, but a very attractive option for large-scale community detection in general.

5. ADAPTIVELY SAMPLING AUTHORITATIVE CONTENT FROM SOCIAL ACTIVITY STREAMS

The tremendous scale of content generation in online social networks brings several challenges to applications such as content recommendation, opinion mining, sentiment analysis, or emerging news detection, all of which have an inherent need to mine this content in real time. As an example, the daily volume of new *tweets* posted by users of Twitter surpasses 500 million.¹ However, not all generated online social activity is useful or interesting to all applications. Using Twitter again as an example, more than 90% of its posts is actually conversational and of interest strictly limited to a handful of users, or spam [50]. Therefore, applications such as emerging news detection that operate on the entire stream, spend a lot of computational cycles as well as storage in processing posts that are not very useful.

One way to solve this problem is, instead of processing the social activity stream in its entirety, to take a sample of the activity and operate on the sample. Through sampling, our goal is to still capture the important and interesting parts of the activity stream, while reducing the amount of data that we would have to process. To this end, one obvious approach is to perform random sampling, i.e., randomly pick a subset of the activity stream and use that in the respective application. A more effective approach however, is to sample content published in the activity stream only from the users that are considered authoritative (or *authorities*).² By sampling the posts of authoritative users from the stream, we are reportedly [129] more likely to produce samples that are of *high-quality*, with limited conversational content and less spam.

The challenge in sampling high quality content from a social activity stream lies therefore in identifying authoritative users. Existing work deploys white-lists of users that are likely to produce authoritative content [49, 50, 116, 129] and samples their activity. Although such approaches have been shown to work well for certain applications, we will show experimentally that they are unable to cope with the dynamic nature of a social activity stream where, for example, new users emerge as authorities and old ones fade out. Other prior efforts on identifying authoritative users in social networks (not streams) have focused on computing a relative ranking of users based on network attributes [6, 23, 24, 64, 103, 132]. We build on the findings of such approaches to identify authorities likely to produce useful content; our approach is different however, as we cannot presume that the complete structure of the social network is available, nor that we can afford to process the network offline.

We operate with the more practical assumption that we have incomplete access to the social network. In other words, we do not know which users exist in the network but we simply observe some partial activity from a social activity stream. Our goal is to produce high quality samples from such streams that will still be as useful as possible compared to being able to access the entirety of the social network and the activity within.

¹<http://www.internetlivestats.com/twitter-statistics/>

²We use terms *authoritative users* and *authorities* interchangeably.

We propose Rhea,³ an adaptive algorithm for sampling authoritative social activity content. Rhea forms a *network of authorities* as it processes a stream and includes in its sample only the content published by the top- K authorities in this network. Given a social activity stream with user interactions (e.g., answers in Q&A sites or mentions in the case of Twitter) we create a weighted graph used to quantify user authoritativeness. To deal with the potentially enormous amount of items that we encounter in the stream and limit memory blowup, we construct a highly compact, yet extremely efficient sketch-based novel data structure to maintain the authoritative users of the network. Our experimental results with half a billion posts from two popular social networks show significant improvements with regard to various binary and ranked retrieval measures over previous approaches. Rhea is able to sample significantly more *relevant* documents, with *higher precision* and remarkably more accurate *ranking* compared to sampling based on static white-lists of authoritative users. Our approach is generic and can be used with any online social activity stream, as long as we can observe indicators of authoritativeness in the stream.

In summary, we make the following contributions:

- a) We propose Rhea, a stream sampling algorithm, that employs network-based measures to dynamically elicit authoritative content of social activity. To the best of our knowledge, this is the first work that addresses the problem of dynamically sampling the posts of authoritative users from a social activity stream.
- b) We evaluate Rhea with datasets reaching up to half a billion posts from two popular social networks and show that it outperforms contemporary approaches with regard to precision, recall, and ranking accuracy.
- c) We empirically demonstrate that static white-lists cannot always capture *temporal* changes in rankings of authorities, and thus, are not an appropriate choice when sampling authoritative content from streams.

5.1 Identifying Authorities in Streams

5.1.1 Network of Authorities from Social Activity

Streams of social activity reveal very little about the respective network structure. Depending on the social network, users may perform certain actions like “posting” messages or “liking” content other users have posted. For example in Twitter, tweets may mention another user’s *@username*, in Facebook users may tag another user, in LinkedIn users can make endorsements, while in Q&A sites such as StackOverflow, users can provide answers to other users’ questions. The aforementioned actions (mentions, endorsements, answers, etc.) as well as their direction may often be considered as indications of importance, and can be used to form a network of authorities from the respective stream. More

³Rhea was the Titaness daughter of the earth goddess Gaia and the sky god Uranus. Her name stands for “she who flows”.

specifically, users receiving numerous mentions or regularly providing answers, without reciprocating these actions with the same frequency, may be deemed as *important* in the network [118].

To illustrate the process of deriving a network of authorities from social activity, we provide an example of a stream featuring the three tweets depicted in Figure 27. In the first element of the stream, *@user1* creates a mention to user *@SLAM* by retweeting a post of that user regarding the injury of a basketball player. Then, the same user retweets some additional information on the same story from the same source. These posts appear in the *feeds* of the users that follow *@user1*. Soon, *@user2* posts a reply to *@user1* and reports that another source (*@SI*) has also confirmed the story. Overall, there are 4 mentions in this stream, most of which offer valuable *evidence* regarding user importance. However, one of the mentions (to *@user1*) is actually only a reply; the respective tweet is conversational and the user simply intends to notify another user. Similarly, in a Q&A site, providing answers is usually an indication of authoritativeness, even though some answers may be inaccurate.

Figure 27 also depicts the actual process of forming a network of authorities out of this particular social activity stream. The network is represented as a directed weighted graph. For each mention in the stream we create an edge from the source node (i.e., user) to the receiving node. If the edge is already present, we increase the respective weight by 1. Using the 3 tweets of our example we can detect a total of 4 nodes. We observe that one of the nodes stands out with regard to weighted in-degree (*@SLAM*). However, we also see that based on weighted in-degree alone, we cannot differentiate between receiving mentions indicating importance and replies. To this end, we can additionally utilize the weighted out-degree to quantify the extent to which these actions are reciprocated, as we discuss next.

5.1.2 Ranking the Authorities

Numerous prior efforts have utilized network structure to identify authorities and exploit the content they produce [6, 23, 64, 132]. Although in our setting we cannot recover the complete network structure, there are usually indications of expertise inherent in the social activity stream that we can utilize. When a user mentions another user in *Twitter* she is either acknowledging the authority of the latter, or trying to engage in a conversation. Both these actions typically imply that the initiating user considers herself less authoritative than the target user. On the other hand, receiving a mention is often an indicator of importance for the recipient. Similarly, asking questions in Q&A sites is usually a negative indicator of authoritativeness, whereas providing answers is a positive one. Therefore, one way to capture this balance is to compute the fraction of the difference between these indicators of importance.

Zhang et al. [132] focus on Q & A communities and propose *z-score*, a measure that builds on positive and negative predictors of *expertise*. The *z-score* of user u is formally defined

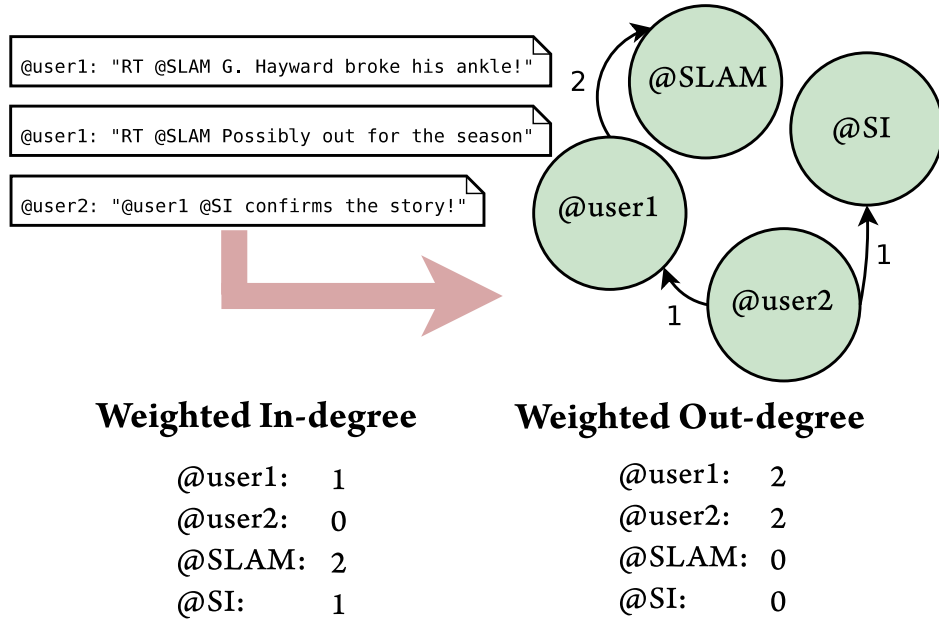


Figure 27: Deriving a network of authorities from a social activity stream. Potential authorities may be identified by applying measures on the resulting weighted directed graph.

as:

$$z(u) = \frac{a(u) - q(u)}{\sqrt{a(u) + q(u)}} \quad (5.1)$$

where, $a(u)$ is the number of questions u has answered and $q(u)$ is the number of questions u has asked. Through crowdsourcing Zhang et al. show that *z-score* outperforms measures such as the *in-degree* as well as sophisticated approaches based on *PageRank* [74] and *HITS* [69] when identifying distinguished users in social networks. We build on this finding and propose *auth-value*, a generalized version of *z-score* for a wide range of social networks, that we formally define as:

$$auth(u) = \frac{in(u) - out(u)}{\sqrt{in(u) + out(u)}} \quad (5.2)$$

where, $in(u)$ is the weighted in-degree of u in the network of authorities and $out(u)$ is her respective weighted out-degree. Thus, our *auth-value* measure enables us to extract the authoritative users of a network in which social activity does not necessarily imply user *expertise*. As the effectiveness of *z-score* against other measures has been previously exhibited [132], we rely on Eq. (5.2) to measure authoritativeness and our focus is on applying it effectively in a streaming setting.

Taking into account both positive and negative predictors of importance through Eq. (5.2) allows us to differentiate between authorities and frequent posters. In particular, users who are frequently mentioned in conversational tweets or provide (possibly incorrect) answers to numerous questions, are also expected to make a lot of mentions to other users or frequently ask questions, and will be penalized by Eq. (5.2) for doing so. More specifically, such users are expected to exhibit an *auth-value* that is negative or close to zero.

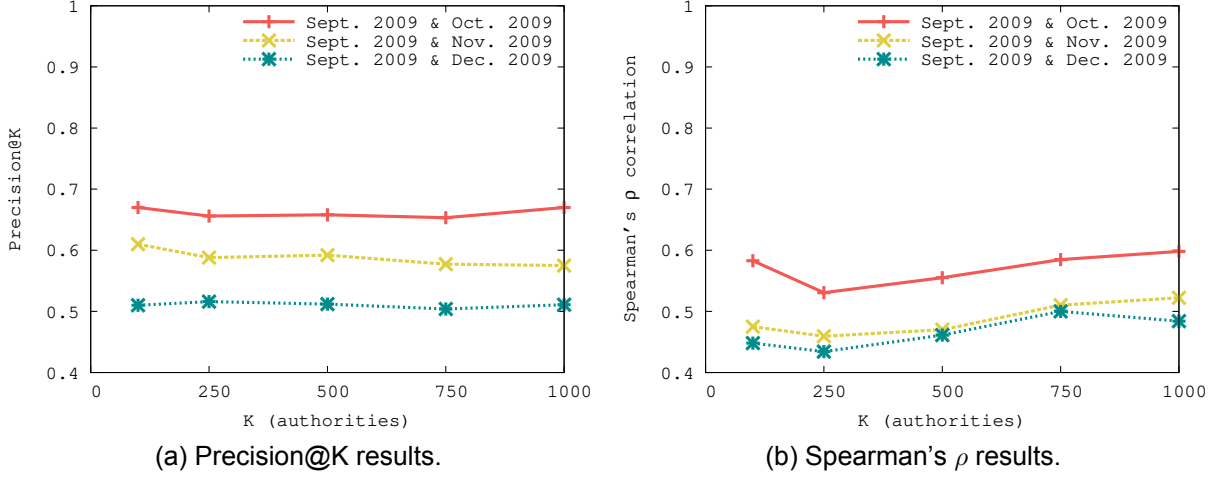


Figure 28: Precision@K 28a and Spearman's ρ 28b results for the authorities extracted from the tweets of September 2009, using the rankings resulting from the tweets of the three subsequent months. Both metrics reveal that the correlation between rankings of authorities according to the tweets of subsequent months weakens significantly with time.

In contrast, authoritative users who receive much more mentions than they give or answer significantly more questions than they ask will exhibit high *auth-values*. We note, that $auth(u)$ is susceptible to spam-farms that may attempt to boost the values of certain users; however, this is the case with alternative network measures as well, e.g., *in-degree*, *PageRank*, or *HITS*. Thus, we consider that fighting web-spamming is beyond the scope of our work. Moreover, we use the notion of authoritativeness to describe influential contributors of a network *regardless* of the diversity of topics discussed. Our focus is on the entire activity and thus, our goal is to distinguish the highly influential players *overall*, as the case is with prior stream sampling efforts [50].

5.1.3 Limitations of Static Lists of Authorities

Previous approaches on sampling the activity of authoritative users from social streams employ *white-lists* of authorities extracted from user annotated content [49, 50, 116, 129]. In particular, social networks often enable users to create *lists* that group together distinguished users. We can form a white-list of authorities by including users with considerable appearances in such user-generated lists [49]. Although this approach can work well in some cases, static white-lists may often be outdated, featuring inactive accounts or users that are no longer receiving attention. Activity in social networks is highly *dynamic* and authorities tend to *rise and fall* with time. To quantify how dynamic social activity is, we used Twitter posts from 3 consecutive months. We created a white-list for each month, that comprises the most authoritative users according to their *auth-values*, and examined the similarity of these white-lists.

Table 9 shows the top-10 authorities for these 3 months. We observe that even at the

Table 9: Top-10 authorities for the tweets of 3 months.

	October 2009		November 2009		December 2009	
	user u	auth(u)	user u	auth(u)	user u	auth(u)
1	justinbieber	393.885	justinbieber	448.815	justinbieber	433.185
2	donniewahlberg	358.286	donniewahlberg	249.988	nickjonas	249.558
3	tweetmeme	263.103	revrunwisdom	242.807	revrunwisdom	222.571
4	revrunwisdom	237.964	tweetmeme	195.379	donniewahlberg	202.996
5	mashable	229.650	addthis	186.282	tweetmeme	183.603
6	addthis	212.325	ddlovato	181.720	jonasbrothers	182.882
7	ddlovato	204.910	luansantanaevc	167.514	addthis	181.403
8	jordanknight	191.045	jordanknight	167.197	omgfacts	154.136
9	jonasbrothers	175.054	jonasbrothers	165.520	mashable	153.616
10	lilduval	174.616	mashable	164.496	johncmayer	147.241

first 10 positions user rankings vary across different months. For instance, user *ddlovato*⁴ started from the 7th spot in October, moved up to the 6th spot in November, before dropping off from the list in December. This is an example of a user that receives increasing attention over time before eventually being surpassed by other emerging users later on. Similarly, user *luansantanaevc*⁵ appeared in the first 10 positions only for the tweets of November. This is an example of a user that received attention *temporarily*. More importantly, this user started posting tweets using a second account (*luansantana*) on January 19th, 2011, without deactivating the first account. Hence, we observe that white-lists can be *unstable* and quickly become *out-of-date*.

To further quantify the volatility of rankings in white-lists, we examined their similarity over time based on the percentage of the users appearing in the list of September 2009 that also appeared in the lists of the 3 subsequent months. More specifically, we consider the ranking resulting from the tweets of September to be the *ground-truth*, and calculate the *Precision@K* achieved in the following 3 months. Figure 28a depicts the *Precision@K* results for $100 \leq K \leq 1,000$ for October, November, and December, respectively. For October (the immediately following month), less than 70% of the authorities of September are also identified as authorities, for all values of K examined. As expected, *Precision@K* deteriorates very quickly in the following months. For the same range of K , we get that $0.57 < \text{Precision@K} \leq 0.61$ for November, and $0.50 < \text{Precision@K} \leq 0.51$ for December. Overall, *Precision@K* remains relatively stable as we increase K and deteriorates as we increase the time interval from the ground truth. We additionally measure the rank correlation using Spearman's ρ with the similar setting of considering September as the ground truth. Figure 28b illustrates Spearman's ρ results for the same pairs of months. There is moderate correlation between the rankings of users, which remains stable as we

⁴*ddlovato* is the account of American singer/actress Demi Lovato:

<https://twitter.com/ddlovato>.

⁵*luansantanaevc* is the old account of Brazilian singer Luan Santana:

<https://twitter.com/luansantanaevc>.

increase K . However, the rank correlation also weakens as we increase the time interval from the ground-truth.

Our findings strongly suggest that white-lists are inappropriate when extracting authoritative users from streams, as social activity is dynamic; instead, we need an adaptive algorithm. We proceed by presenting such an algorithm and measuring its effectiveness against contemporary white-list based approaches.

5.2 Rhea: Stream Sampling for Authoritative Content

In this section we present Rhea, an adaptive sampling algorithm for authoritative content from social activity streams. More formally, Rhea seeks to produce a sample \hat{S} of a stream S such that $\forall s \in S$ whose respective user is in the $top-K$ authorities of the network according to Eq. (5.2), $s \in \hat{S}$.

This endeavor involves three main challenges: 1) Online social networks are ever increasing and users publishing content may surpass 1 billion [33]. Hence, maintaining user information as we process the stream may be costly in both memory requirements and computational time. 2) Ranking users according to their authoritativeness and classifying their content as relevant or non-relevant, often requires reckoning in multiple measures. 3) Finally, many elements we opt to include in the sample as we process the stream may actually be published by non-authorities. Thus, we need to filter out posts that mistakenly lurked in our sample. In this section, we discuss the individual pieces of Rhea that address these three challenges and then present our algorithm.

5.2.1 Maintaining User Information

5.2.1.1 Frequent Items

Rhea maintains a *limited view* of the social network based on the social activity stream. In particular, Rhea is aware of the weighted in- and out-degrees of each user in the stream, as depicted in the weighted directed graph of Figure 27. In practice, we expect that an enormous number of users will participate in the activity stream of an online social network. Efficiently mapping their respective weighted in- and out-degrees with structures such as hash tables would require memory that far surpasses that of a modern day computer. Moreover, resizing such hash tables would be necessary to maintain new users encountered in the stream, and would eventually cause serious bottlenecks in terms of CPU cycles.

The Count-Min sketch [35] is a well-known and widely-used [5, 106] sublinear space data structure for the representation of high-dimensional vectors. Count-Min sketches allow fundamental queries to be answered efficiently and with strong accuracy guarantees. It is particularly useful for summarizing data streams as it is capable of handling updates at high rates. The sketch uses a two-dimensional array of w columns and d rows, where

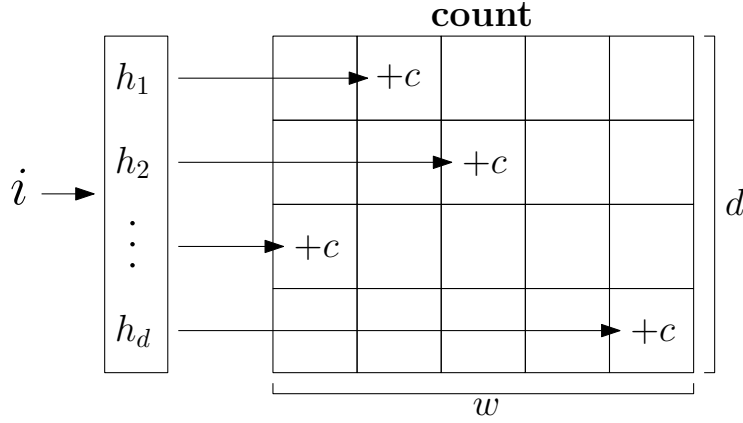


Figure 29: Count-Min Sketch update process.

$w = \lceil \frac{e}{\epsilon} \rceil$, $d = \lceil \frac{\ln(1/\delta)}{\delta} \rceil$, and the error in answering a query is within a factor of ϵ with probability δ . A total of d pairwise independent hash functions is also used, each one associated with a row of the array. Figure 29 illustrates the update process of a Count-Min sketch for our specific problem. Consider that an update (i, c) arrives, indicating that user's i count should be incremented by c . The array *count* is updated as follows: for each row j of *count* we apply the corresponding hash function to obtain a column index $k = h_j(i)$ and increment the value in row j , column k of the array by c , i.e., $\text{count}[j, k] + = c$. This allows for retrieving at any time an (over)estimation of the count of an event i using the least value in the array for i , i.e., $\hat{a}_i = \min_j \text{count}[j, h_j(i)]$.

Rhea keeps track of both positive and negative indicators of importance. Thus, we employ two Count-Min sketches to compactly maintain both these indicators for all users appearing in a stream.

5.2.1.2 Reducing the Processing Overhead through Sampling

Palguna et al. [104] show that a *uniform random sample with replacement* of enough size is able to guarantee with strong accuracy that *i)* elements that occur with frequency more than θ in the stream occur with frequency more than $(1 - \frac{\epsilon}{2})\theta$ in the sample and *ii)* elements that occur with frequency less than $(1 - \epsilon)\theta$ in the stream occur with frequency less than $(1 - \frac{\epsilon}{2})\theta$ in the sample, where $\theta \in [0, 1]$ and $\epsilon \in [0, 1]$. In addition, they experimentally show that the behavior of the *Bernoulli sampling scheme* is very similar and primarily influenced by the sample size alone. Obviously, we are unable to use *uniform random sample with replacement*, as elements of the stream are only seen once. However, we can employ the *Bernoulli sampling scheme*. In particular, we can include each element of the stream in our authorities' network formation process with probability p and exclude the element with probability $1 - p$, independently of other elements, where $p \in (0, 1]$. This allows us to reduce the computational overhead of Rhea without sacrificing its effectiveness. We thoroughly investigate the impact of p in our evaluation to come up with the size of the sample that will facilitate our set requirements.

Algorithm 7: $\text{put}(Top-K\text{-Heap}, key, value)$

input : A $Top-K\text{-Heap}$ structure and a key associated with a $value$ to be inserted in the $Top-K\text{-Heap}$.

output : The updated $Top-K\text{-Heap}$.

```

1 begin
2   if  $Top-K\text{-Heap.size}() < K$  then
3     if  $Top-K\text{-Heap.contains}(key)$  then
4        $Top-K\text{-Heap.replace}(key, value);$ 
5     else
6        $Top-K\text{-Heap.push}(key, value);$ 
7   else
8     if  $Top-K\text{-Heap.contains}(key)$  then
9        $Top-K\text{-Heap.replace}(key, value);$ 
10    else if  $value > Top-K\text{-Heap.low}()$  then
11       $Top-K\text{-Heap.pop}();$ 
12       $Top-K\text{-Heap.push}(key, value);$ 
13  return  $Top-K\text{-Heap};$ 

```

5.2.2 Ranking Authorities

Count-Min sketches provide answers to point and dot product queries with strong accuracy guarantees. Using two such sketches, we are able to approximate the number of positive and negative indicators of importance a user exhibits. This is enough to provide us with an approximation of a user's *auth-value* through Eq. (5.2). However, we are not interested in the *absolute* value of $auth(u)$. Rather, we wish to know at any time whether a user's value is among the top- K overall. To this end, we employ a structure we term $Top-K\text{-Heap}$ to hold user elements with associated *auth-values*. A $Top-K\text{-Heap}$ *puts* an element in the structure if its value is larger than the minimum value currently on the structure or the structure holds less than K elements. In case the element is already inserted, we update its value accordingly; otherwise, we first remove the element with the smallest value. Thus, a $Top-K\text{-Heap}$ holds a maximum of K elements. In addition, duplicate values are allowed, as users may exhibit the same *auth-value*.

A *min-heap* [11] allows for duplicate values and enables us to examine the minimum element of our structure in *constant* time. In addition, *min-heaps* support insertion of elements or removal of the minimum element in *logarithmic* time. Therefore, if the element is not present in the structure, we can place it in a *min-heap* and remove the root holding the minimum value in *logarithmic* time. However, examining if an element is already in a *min-heap* takes *linear* time. To alleviate this problem, we additionally employ a *hash-table* to hold the inserted elements, which allows for examining the presence of an element in our $Top-K\text{-Heap}$ in *constant* time. We note that K is insignificant compared to the total number of users, and the cost of using an additional *hash-table* is negligible.

Algorithm 7 details the insertion in a $Top-K\text{-Heap}$. Lines 2-6 concern the case when

the *Top-K-Heap* holds less than K elements. If the new element is already inserted we replace its value (Line 4), i.e., we remove the old element from the *min-heap* and the *hash-table* and insert the new one with the updated value ($O(\log n)$). If the new element is not in the structure, we simply place it inside (Line 6), i.e., we insert it to both the *min-heap* and the *hash-table* ($O(\log n)$). Lines 7-12 are executed in the case when the *Top-K-Heap* holds exactly K elements. If the latter is true and the element to be added is already inserted, we replace its value as before (Line 9). However, if the new element is not already in the *Top-K-Heap*, we examine if its value is larger than the minimum value on the *Top-K-Heap* (Line 10), and remove the root of the *Top-K-Heap* before inserting it. This requires us to access the minimum value ($O(1)$), remove the root of the *min-heap* ($O(\log n)$) and the respective element in the *hash-table* ($O(1)$), and then insert the new element in the *min-heap* ($O(\log n)$) and the *hash-table* ($O(1)$). We note that Eq. (5.2) may both increase or decrease as elements appear in the stream. Therefore, when we update an element in the *Top-K-Heap* with a value that is smaller than the one previously held, it might be the case that the element should no longer be part of the top- K . However, as we update the *Top-K-Heap* with every element that appears on the stream, the element that would actually belong to the top- K will claim its position at its next appearance, and will be included in the sample.

5.2.3 Filtering-out Non-relevant Activity

Rhea makes decisions based on what appears to be optimal at the time. During stream processing, Rhea may deem as a top- K authority a user that temporarily exhibits a high *auth-value* but is actually not among the top- K overall for the particular stream. Thus, posts of non-authorities may end up in our sample, i.e., we lose in *precision*. Similarly, Rhea might stumble upon posts of an authority that is not yet identified as such. This will lead to relevant posts being excluded from our sample, i.e., we lose in *recall*. For this latter case, we are unable to improve our *recall* at a later stage, as the elements that we discard from the stream are lost. However, for the former case we can perform a post-processing step to filter-out non-relevant posts using the more refined classification model that is formed after seeing a good portion of the stream. For each document included in our sample, Rhea examines the respective user that published it. If the user is contained in our *Top-K-Heap*, we keep the document in the sample; otherwise, we discard it. In our evaluation, we investigate the impact of this technique in detail.

5.2.4 The Proposed Rhea Algorithm

Algorithm 8 outlines our proposed Rhea method for stream sampling. Rhea processes a stream S with elements of social activity. Each element contains some content and is associated with a user and a timestamp. Rhea takes as its input parameters K and p , that specify the amount of authorities whose activity we wish to sample, and the probability according to which we process an element in the stream to form our network of authorities,

Algorithm 8: Rhea(S, K, p)

input : A stream S , a parameter $K > 0$ and a probability $p \in (0, 1]$.
output : A set $\hat{S} \subset S$ containing elements whose respective users are likely to be among the top- K w.r.t. to the *auth-value*.

```

1 begin
2    $Top\text{-}K\text{-heap} \leftarrow \emptyset$ ;
3    $CMSin \leftarrow \emptyset$ ;
4    $CMSout \leftarrow \emptyset$ ;
5   foreach  $s \in S$  do
6     if  $random(0, 1] < p$  then
7        $(in, out) \leftarrow extractIndicators(s.message)$ ;
8        $CMSin[in]++ = 1$ ;
9        $CMSout[out]++ = 1$ ;
10     $auth_{user} \leftarrow \frac{CMSin[s.user] - CMSout[s.user]}{\sqrt{CMSin[s.user] + CMSout[s.user]}}$ ;
11    if  $auth_{user} > Top\text{-}K\text{-heap}.low()$  then
12       $put(Top\text{-}K\text{-heap}, user, auth_{user})$ ;
13       $\hat{S}.put(s)$ ;
14  foreach  $s \in \hat{S}$  do
15    if  $s.user \notin Top\text{-}K\text{-heap}$  then
16       $\hat{S}.remove(s)$ ;
17  return  $\hat{S}$ ;

```

respectively. The output is a sample of S containing elements whose respective users are likely among the top- K w.r.t. the *auth-value*.

Rhea begins by initializing the structures to be used while processing the stream (Lines 2-4), i.e., a *Top-K-Heap* to hold the current K users with the highest *auth-value* in the stream, and two Count-Min sketches to maintain the weighted in- and out-degrees of each user. Then, we process the elements of the stream (Line 5), a phase that involves two actions:

Creating the Network of Authorities (Lines 6-9):

We apply a *Bernoulli sampling scheme* and use an element of the stream with probability $p \in (0, 1]$ to extract positive and negative indicators of importance (Line 6-7). The extracted indicators are used to update the two Count-Min sketches (Lines 8-9).⁶ Hence, sketches $CMSin$ and $CMSout$ keep track of the weighted in- and out-degrees of the users of the formed authorities' network, respectively.

Stream Sampling for Authoritative Content (Lines 10-13):

First, we derive an approximation of the *auth-value* of the respective user of the current element of the stream (Line 10). Then, we compare with the lowest value in the *Top-K-Heap* to decide whether the current user is an authority, and thus, her activity must be

⁶Depending on the stream an element may contain more than one positive or negative indicators of importance. We consider this in our implementation but we omit it from the presentation of our algorithm for simplicity.

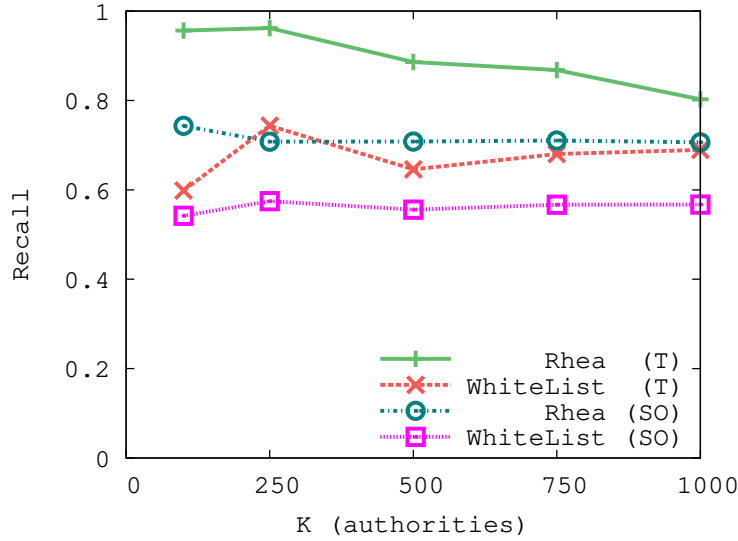


Figure 30: Recall comparison between our approach and a baseline for our two datasets (T, SO) when querying for the tweets of the top-100, 250, 500, 750, and 1,000 authorities of the stream.

sampled or not (Line 11). If the user is classified as an authority, we update the *Top-K-Heap* with the *auth-value* of the user (Line 12), and include the element in our sample (Line 13).

Finally, Rhea features a post-processing step to improve the quality of the sample by filtering-out elements that were wrongly considered as relevant while we were processing the stream (Lines 14-16). This step processes the elements of the sample and removes all those whose respective users are not in the *Top-K-Heap*.

5.3 Experimental Evaluation

We implemented⁷ Rhea using Java. Our evaluation is based on two datasets: *i*) one that comprises 467 million *tweets* from 20 million users of Twitter (T), covering a period from June 2009 to December 2009 [122], and *ii*) one that consists of 263,540 answers to 83,423 questions posted by 26,752 users of StackOverflow (SO), between February 18, 2009 and June 7, 2009 [39]. We first present the details of our experimental setting. Then, we proceed with the evaluation of Rhea by answering the following questions:

- How does Rhea compare against white-list based sampling in terms of *recall*, *precision*, and *F1-score*?
- Is Rhea able to assess the ranking relevance of the sampled documents?
- What is the impact of the parameters involved in the execution of Rhea?

⁷Source code and reproducible tests: <https://github.com/panagiotisl/rhea>

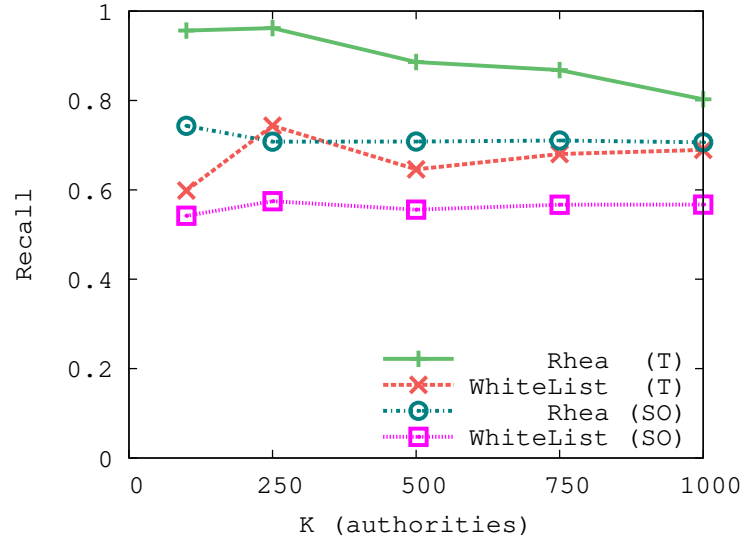


Figure 31: *Precision* comparison between our approach and a baseline for our two datasets (T, SO) when querying for the tweets of the top-100, 250, 500, 750, and 1,000 authorities of the stream.

5.3.1 Experimental Setting

For our `Twitter` dataset we include in our stream all tweets published during August 2009 – December 2009, i.e., $|S| = 411,778,304$. Similarly, $|S| = 131,768$ for our `StackOverflow` dataset. We did not use all available elements for the stream, as we also needed a sufficiently large part of the dataset to create static white-lists for a method based in [50] that we compare against. We created the white-lists using the *auth-value* rankings that result from all the available user activity occurring before the activity of the stream. The respective approach decides to sample elements from the stream based on a single criterion: whether the user publishing the element is part of the white-list or not. For the rest of this work we will refer to this method as `WhiteList`. The elements of both our datasets are timestamped which enables us to replay them chronologically. Unless stated otherwise, `Rhea` is initialized using the following parameters: *i*) $p = 0.2$, to use 20% of the stream’s elements to extract mentions, and *ii*) $d = 7, w = 20,000$, which gives us 99% confidence that $\epsilon < 0.0001$.

5.3.2 Recall, Precision, and F1-score Comparison

We commence our evaluation by comparing the performance of `Rhea` against `WhiteList` with regard to *recall*, *precision* and *F1-score* measures. For each element (tweet or answer) we include in our samples, we make a binary assessment concerning the user who posted it. If the user is among the top- K according to the ground-truth, we mark the element as relevant; otherwise, we consider the element to be non-relevant.

We observe in Figure 30 that `Rhea` significantly outperforms `WhiteList` with regard to *recall* for both datasets. That is, `Rhea` is able to include *more relevant* documents than

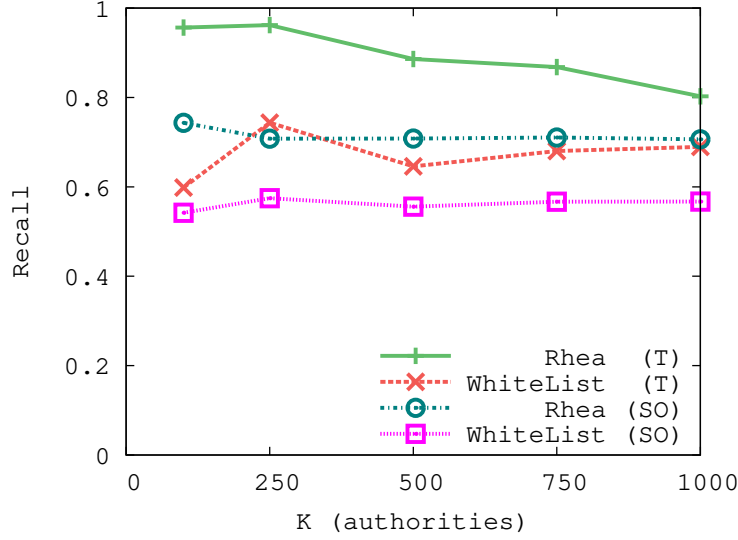


Figure 32: *F1*-score comparison between our approach and a baseline for our two datasets (T, SO) when querying for the tweets of the top-100, 250, 500, 750, and 1,000 authorities of the stream.

WhiteList in its sample. In particular, more than 74% of the relevant documents is included in the sample of Rhea for both datasets even for $K = 1,000$, whereas, WhiteList’s recall drops as low as 0.55. Figure 31 illustrates the *precision* achieved by Rhea and WhiteList. Rhea behaves much better than WhiteList for the StackOverflow dataset, achieving almost perfect *precision*. For Twitter, we observe that both methods initially behave similarly. This is because a few very active non-authorities that are mistakenly taken as authorities may heavily impact *precision* for small values of K . However, as K grows Rhea significantly outperforms WhiteList for Twitter as well. Finally, we illustrate the results of both methods regarding *F1-score*, i.e., the harmonic mean of *precision* and *recall*, in Figure 32. We observe that our approach achieves an *F1-score* that is above 0.8 for StackOverflow and close to 0.8 for Twitter regardless of K . In contrast, using a static white-list, the *F1-score* is much lower and ranges between 0.54 and 0.7.

5.3.3 Evaluation of Ranked Retrieval Results

Recall, *precision*, and *F1-score* measures are appropriate for sets of documents that have no ranking information associated to them. The binary assessment we make to classify an element as relevant or non-relevant does not consider the significance of the element with regard to its respective user’s authoritativeness. However, we are keenly interested in *ranking quality*. To this end, we employ two additional measures that take under consideration the level of relevance of each element, namely *Spearman’s ρ* and *Normalized Discounted Cumulative Gain (NDCG)*.

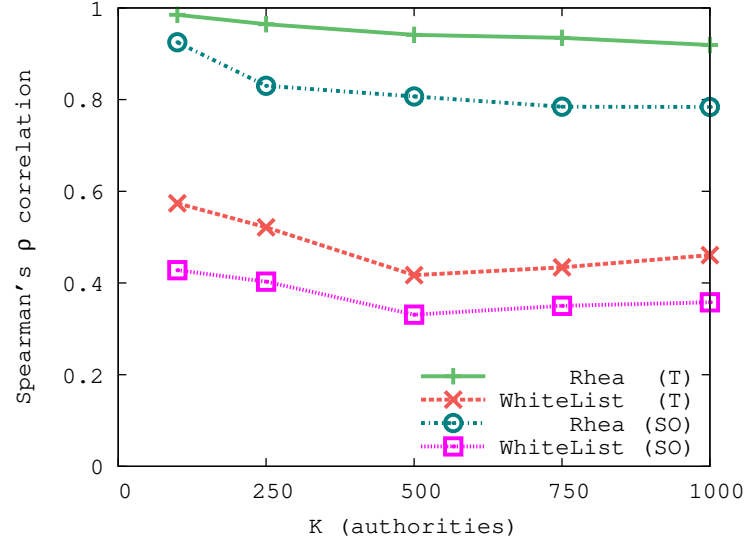


Figure 33: Comparison of Rhea and WhiteList on Spearman's ρ for Twitter (T) and StackOverflow (SO).

5.3.3.1 Evaluation using Spearman's ρ

We first investigate the rank correlation between the ground-truth *auth-values* resulting from all the elements of the stream and the *auth-values* derived from each of the two methods examined in this chapter. Figure 33 depicts the *Spearman's* rank correlation for the top- K users of the ground-truth with their respective rankings for Rhea and WhiteList. In particular, we assigned the rank of 1 to the user with the highest *auth-value* in the ground-truth and increased the rank as we proceeded to users with lower *auth-value*, until the K^{th} user. Then, we created pairs with the rankings of the users that occur when using Rhea and WhiteList. We observe that there is an *extremely strong correlation* for Rhea, i.e., our approach is able to adapt and derive the order of the top- K authorities very accurately. In contrast, WhiteList exhibits *moderate* to *weak* correlation. These results do exhibit in unambiguous terms the superiority of Rhea over contemporary white-list methods, as in addition to higher *precision* and *recall*, our adaptive algorithm captures *much more accurately* the level of importance of each user.

5.3.3.2 Evaluation using NDCG

Next, we use *NDCG*, a measure, suitable for situations of non-binary notions of relevance [94]. *NDCG* is evaluated over some number K of top results. We consider rel_i to be the graded relevance of the result at position i . Then, the *discounted cumulative gain* (*DCG*) at K is defined as:

$$DCG_K = rel_1 + \sum_{i=2}^K \frac{rel_i}{\log_2(i)} \quad (5.3)$$

From Eq. (5.3) we observe that DCG reduces the graded relevance value of each result logarithmically proportional to its respective position in the ranking, to penalize highly relevant documents that appear lower than their actual position [117]. Our goal is not only to retrieve a ranked list of users according to their authoritativeness, but also retrieve their social activity. Therefore, for our purpose we propose an extension of Eq. (5.3) that considers the *recall* for each user i :

$$DCG_K = rel_1 * recall_1 + \sum_{i=2}^K \frac{rel_i * recall_i}{\log_2(i)} \quad (5.4)$$

$NDCG$ results after normalizing the cumulative gain at each position for a given K as follows:

$$NDCG_K = \frac{DCG_K}{IDCG_K} \quad (5.5)$$

where $IDCG_K$ is the maximum possible (ideal) DCG for the given set of relevances:

$$IDCG_K = rel_1 + \sum_{i=2}^{|REL|} \frac{rel_i}{\log_2(i)} \quad (5.6)$$

and $|REL|$ stands for the ordered list of relevant documents up to position K .

We consider that the elements of user i have a relevance $rel_i = K + 1 - rank(i)$, where $rank(i)$ is the ranking of the users according to their ground-truth *auth-value*. Thus, the elements of the user with the highest *auth-value* have a relevance of K , whereas those of the user with the K^{th} highest ground-truth ranking have a relevance of 1.

Figure 34 illustrates the results of Rhea and WhiteList with regard to $NDCG$ for different values of K . We observe that Rhea again *significantly outperforms* the WhiteList method for both datasets. The latter performs *poorly* with regard to $NDCG$ as its value is penalized severely when assigning low rankings to highly relevant users. A vital observation here is the improved performance of Rhea on $NDCG$ compared to *recall* (Fig. ???). From this we can induce that the few relevant documents that Rhea is unable to retrieve, are usually not of high relevance. If that was the case, the $NDCG$ results would be worse than those measuring *recall*. This is particularly important; we are interested in sampling the elements of the top- K users in the stream, and thus, we are generally more keen on retrieving the elements of the most relevant users. Figure 34 shows that Rhea is *very effective* in doing so.

5.3.4 Impact of Techniques and Parameters

In this section, we investigate the impact of Rhea's techniques and parameters using the largest of our two datasets, namely Twitter. First, we examine the performance of Rhea when altering the probability p of examining a tweet of the stream S to extract mentions and form the *network of authorities*. Second, we quantify the importance of the filtering step of the Rhea algorithm (Lines 14-16 of Algorithm 8). Third, we vary the size of the *Top-K-Heap* to examine its impact on *F1-score*. Our findings are in agreement with those that come up using the StackOverflow dataset, but we omit the latter due to limited space.

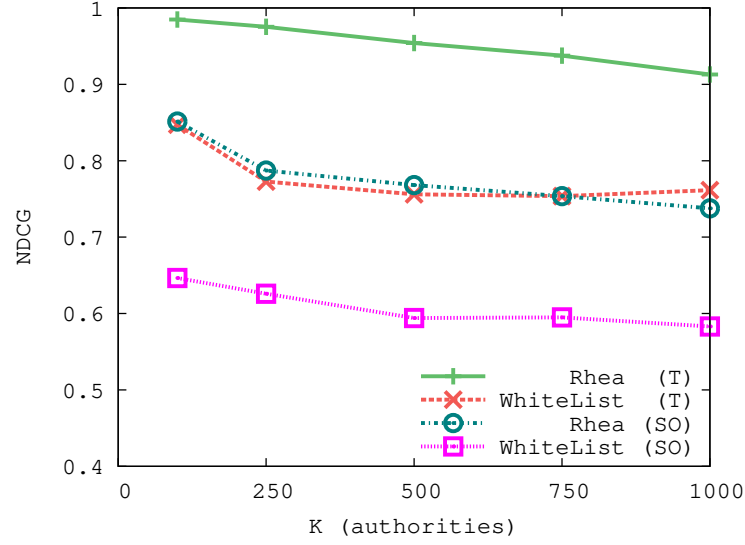


Figure 34: Comparison of Rhea and WhiteList on NDCG for Twitter (T) and StackOverflow (SO).

5.3.4.1 Varying the Value of Probability p

Rhea involves a random sampling subprocedure, that selects to use with some probability $p \in (0, 1]$ an element of the stream to form the network of authorities. This process significantly reduces the computational overhead of Rhea as we use $|S| * p$ elements of the stream, instead of $|S|$. We examine here the impact this probability has on the results of Rhea with regard to *NDCG*. Figure 35 depicts the performance of our algorithm in settings where p is equal to 0.01, 0.05, 0.1, 0.2, and 1, respectively. We observe that using a sample of 20% of the stream’s elements we are able to achieve performance that is almost as good as that of using the *entire* stream. Moreover, we observe negligible differences when reducing p to 0.1 or 0.05. In fact, the impact of probability p is noticeable only when p is extremely low. Finally, even though using 1% of the elements leads to worse performance, the *NDCG* results we get for Rhea still outperform WhiteList *significantly*. We note that using $p = 0.2$ instead of $p = 1$ greatly reduces processing time. For example, we drop from 3,844 to 2,533 seconds for $K = 100$. For $p = 0.01$ Rhea terminates after 2,189 seconds, slightly over WhiteList that needs 2,040 seconds.

5.3.4.2 Removing the Filtering Step

Rhea samples elements from the stream in a greedy fashion. Therefore, elements of users that are only temporarily part of the top- K authorities manage to end up in our sample. However, when the sampling process is over, we are aware of a final set of top- K authorities, that we have experimentally shown to be a very accurate representation of the actual list of authorities. Hence, we are able to filter-out the elements that in retrospect should not have been collected, by iterating over the sampled elements. We note that $\hat{S} \ll S$, so this operation is inexpensive. Figure 36 compares the performance of Rhea

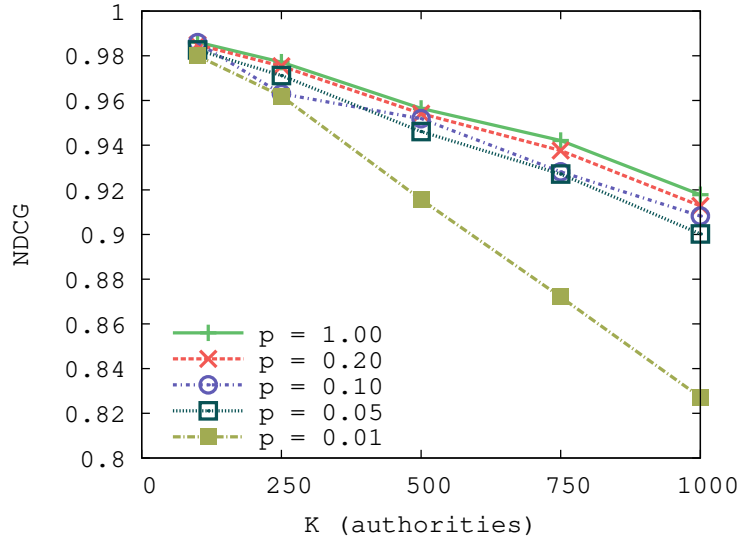


Figure 35: Impact of probability p on $NDCG$.

when the filtering step is on (Rhea) and off (Rhea-NF). We opt to report the *precision* value, as *recall*, *Spearman's ρ* , and *NDCG* results are all unaffected by this modification. We observe that the difference in *precision* performance is indeed significant. In particular, the difference is over 25 percentage points for $K = 1,000$, and is never less than 10 percentage points for any K examined.

5.3.4.3 Impact of the Capacity of the Top-K-Heap

We complete our exploration on the parameters of Rhea by examining the impact of the size of the structure we use for holding the stream's current list of authorities. Rhea maintains a heap of authorities induced from the social activity occurring in the stream. This heap has a maximum capacity that enables us to decide on whether to temporarily include an element in our sample or permanently discard it. The size of this heap in our experiments is set to K , i.e., the number of authorities whose activity we aim to include in our sample.

We investigate here an approach that may potentially increase our *recall*. Our intuition is that some authoritative users are “*late bloomers*”, i.e., their importance is not visible until later than expected. Rhea is unable to recover the *entire* activity of such users as it is unaware at the time of sampling of the *final* ranking of each user. However, we may opt to include the activity of more users while sampling, and eventually hold on to the elements produced by those who we believe are the top- K authorities. Figure 37 illustrates a comparison of the performance of Rhea when using a *Top-K-Heap* of capacity K and $2K$, respectively. We use the *F1-score* measure as the capacity of this structure impacts both *recall* and *precision*. We observe that the *F1-score* of Rhea when using a capacity of $2K$ is slightly worse. More specifically, our *recall* is improved as we include the activity of more users during sampling. However, using a larger capacity also leads to including

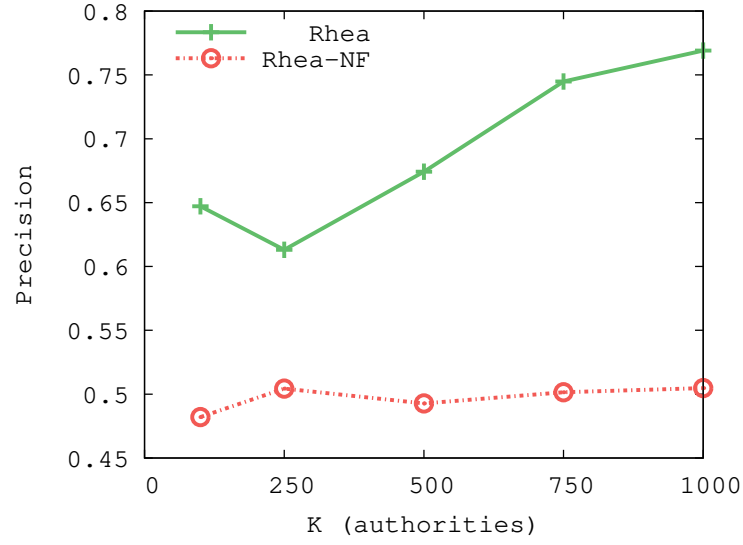


Figure 36: Impact of the filtering step of Rhea on *precision*.

more false positives in our sample. Therefore, we do indeed notice an improvement in *recall*, but it is accompanied with significantly worse *precision*.

5.4 Related Work

Our work lies in the intersection of social activity stream sampling and authoritative social network users identification. Here, we briefly discuss pertinent efforts in these two areas.

Social Activity Stream Sampling: Related research efforts have mainly focused on the *Twitter* microblogging service due to its immense popularity and low latency access to its stream of activity. Ghosh et al. [50] compare random samples of *Twitter* with samples that are taken using a white-list of users. Their motivation is to avoid the large amount of spam, non-topical and conversational tweets that random sampling preserves. The first set of tweets was acquired through the *Streaming API*, while the second is created using tweets from half million white-listed users. The white-list is derived using *Twitter Lists* [49], i.e., user generated lists of prominent *Twitter* accounts. The random sample features a substantially larger population of users, whereas the white-list sample's tweets are extremely more popular. Moreover, the quality of the tweets of the white-listed users is found to be superior. In particular, about 90% of the random sample's *tweets* are conversational, whereas 43% of the white-list sample's tweets contain useful information on a certain topic. Our work is similar to [50] as we also sample streaming social activity content. However, our work does not rely on static white-lists and our focus is not on a specific social network.

Palguna et al. [104] come up with a theoretical formulation for sampling *Twitter* data. They investigate the number of tweets that is needed to come up with a representative sample using random sampling with replacement. To decide on how representative a

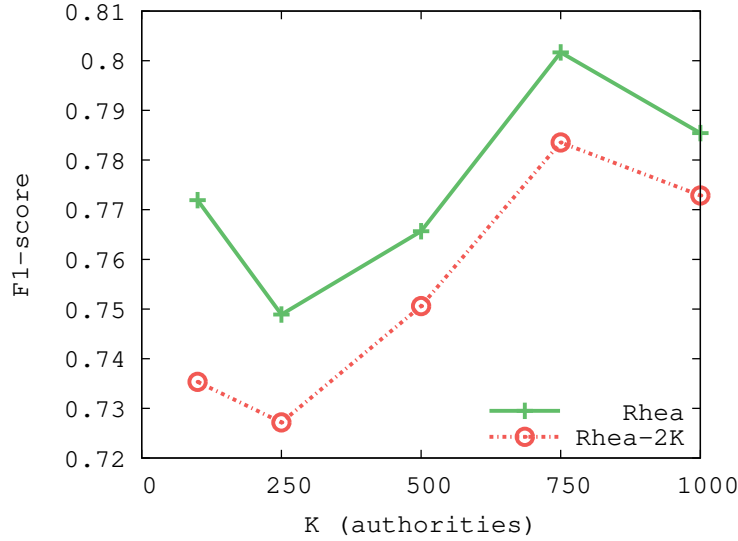


Figure 37: Impact of the capacity of the *Top-K-Heap* on *F1-score*.

sample is, they examine how the frequency of elements in the streams correlates in the sample and the original data. In addition, they examine the case of going through tweets one-by-one and sampling each tweet independent of others with probability p . They show that this behaves similarly to random sampling with replacement and is primarily influenced by the size of the sample. We build on this very last result to speed-up the formation of our network of authorities.

Research efforts have also focused on the quality of the samples offered directly from Twitter. Morstatter et al. [100] perform a comparison of Twitter’s Streaming API sample and Twitter’s Firehose to examine the impact of the sampling technique of the first. They compare the top *hashtags* of the two datasets, as well as those of random samples taken from the Firehose dataset, and find that the random samples find the top hashtags more consistently than the Streaming API. Moreover, a comparison of topics in the two datasets is performed, using LDA, which shows that decreased coverage in the Streaming API data causes variance in the discovered topics.

Mining streams of social activity is challenging due to the implicit network structure within the stream that ought to be considered along with the content. Aggarwal and Subbian [5] focus on clustering and event detection using social streams. They show that using both the content and the linkage information has numerous advantages. Node counts for individual clusters are handled by employing Count-Min sketches [35]. We also apply Count-Min sketches to summarize counting information of social streams. However, we do not deal with clustering or event detection; rather, we focus on sampling content published by authorities.

Authoritative Users in Online Social Networks: Zhang et al. [132] investigate different network-based ranking algorithms to identify prominent users in a online social network. They show that relative expertise can be automatically determined through structural in-

formation of the network, as they find that network-based algorithms perform nearly as good as crowd-sourcing. In addition, they report that simple measures behave at least as good as complex algorithms. In particular, they come up with *z-score*, a measure that considers both the question and answer patterns of a user in a Q & A community, that best captures the relative expertise of users in a network. In this chapter, we rely on the findings of [132] on the effectiveness of *z-score* and propose a generalized version of this measure to identify the top- K authorities in any social activity stream. Agichtein et al. [6] exploit various kinds of community feedback to export high quality content from social media. Among else, they use quality ratings on the content. Pal and Counts [103] use probabilistic clustering and a within-cluster ranking procedure to identify topical authorities on *Twitter*. In an effort to exclude users with high visibility they use nodal features, such as the in-degree. In [24], Bozzon et al. focus on finding topical experts in various popular social networking sites. Their approach takes into account user activity as well as profile information. We operate on a streaming setting and decide whether new content is useful as it becomes available. Therefore, certain aspects of the aforementioned approaches, such as exploiting user ratings, in-degrees, and profile information are not applicable. Ghosh et al. [49] propose *Cognos*, which distinguishes authoritative *Twitter* users using the frequency at which they are included in *Twitter Lists*. This approach assumes the presence of user annotated information indicating importance, whereas in this chapter we consider the task of sampling a stream without any prior knowledge. Moreover, we show that static white-list approaches get outdated very quickly and are unable to identify newly emerging authorities. Rybak et al. [111] also point out that authoritativeness is not static. However, they do not deal with stream sampling. Instead, they focus on a co-authorship network and create timestamped profiles of user importance.

5.5 Conclusion

In this chapter, we propose and implement *Rhea*, the first reported effort to realize adaptive behavior for sampling authoritative content from social activity streams. We commence by exposing the *dynamic* nature of this task which calls for approaches different from employing static white-lists of authoritative users. Then, we proceed by addressing the challenges involved in our dynamic approach. *Rhea* employs Count-Min sketches to compactly maintain both positive and negative indicators of importance of all users appearing in a social activity stream. We additionally propose a novel structure termed *Top-K-Heap*, to efficiently query for the top- K authoritative users in the stream, using their relative ranking resulting from their *auth-value*. The latter allows for identifying authoritative users independently of the underlying social network. To reduce the processing overhead of extracting indicators of importance from social activity streams, *Rhea* opts to include in this process each element of the stream with probability p . Finally, *Rhea* features a post-processing step that reevaluates content included in the sample, using the more refined classification model that is available after reading the whole stream.

We compare *Rhea* with a static white-list approach using two datasets reaching up to half a billion posts. We show that *Rhea* exhibits significantly improved performance with

regard to both *recall* and *precision*. The superiority of Rhea is even more evident when comparing on ranking accuracy, using the Spearman's ρ and *NDCG* measures. Finally, we investigate the effect of various parameters of Rhea and ascertain its improved efficiency and effectiveness.

6. ON THE IMPACT OF SOCIAL COST ON OPINION DYNAMICS

An ever-increasing amount of social activity information is available today, due to the exponential growth of online social networks. The structure of a network and the way the interaction among its users impacts their behavior has received significant interest in the sociology literature for many years. The availability of such rich data now enables us to analyze user behavior and interpret sociological phenomena at a large scale [8].

Social influence is one of the ways in which social ties may affect the actions of an individual, and understanding its role in the spread of information and opinion formation is a new and interesting research direction that is extremely important in social network analysis. The existence of social influence has been reported in psychological studies [67] as well as in the context of online social networks [22]. The latter usually allow users to endorse articles, photos or other items, thus expressing shortly their opinion about them. Each user has an internal opinion, but since she receives a feed informing her about her friends' endorsements, her expressed (or overall) opinion may well be influenced by her friends' opinions. This process may lead to a consensus.

The most notable example of studying consensus formation due to information transmission is the DeGroot model [38]. This model considers a network of individuals with an opinion which they update using the average opinion of their friends, eventually reaching a shared opinion. In [48] the notion of an individual's internal opinion is added, which, unlike her expressed opinion, is not altered due to social interaction. Such a setting is illustrated in Figure 38 where an individual's expressed opinion results from her friend's expressed opinions as well as her internal belief. This model captures more accurately the fact that consensus is rarely reached in real word scenarios. The popularity of a specific article, for instance, may vary largely between different communities in a social network. This fact gives rise to the study of the lack of consensus, and the quantification of the social cost that is associated with disagreement [16]; the authors here consider a game where the utilities are the users' social costs and perform repeated averaging to get the Nash equilibrium. The resulting models of opinion dynamics in which consensus is not in general reached allow for testing against real-world datasets, and enable the verification of influence existence.

An approach towards verifying the existence of influence against real-world data based on general models is that of [8]. However, the models proposed are probabilistic and the correlation that is present on the data is not attributed to *influence*. Investigating game theoretic models of networks against real data is crucial in understanding whether the behavior they portray depicts an illustration that is close to the real picture.

Our contributions

We study the spreading of opinions in social networks, using a variation of the DeGroot model [48] and the corresponding game detailed in [16]. We perform an extensive analysis on a large sample of a popular social network and highlight its properties to indicate its appropriateness for the study of influence. The observations we make verify our intu-

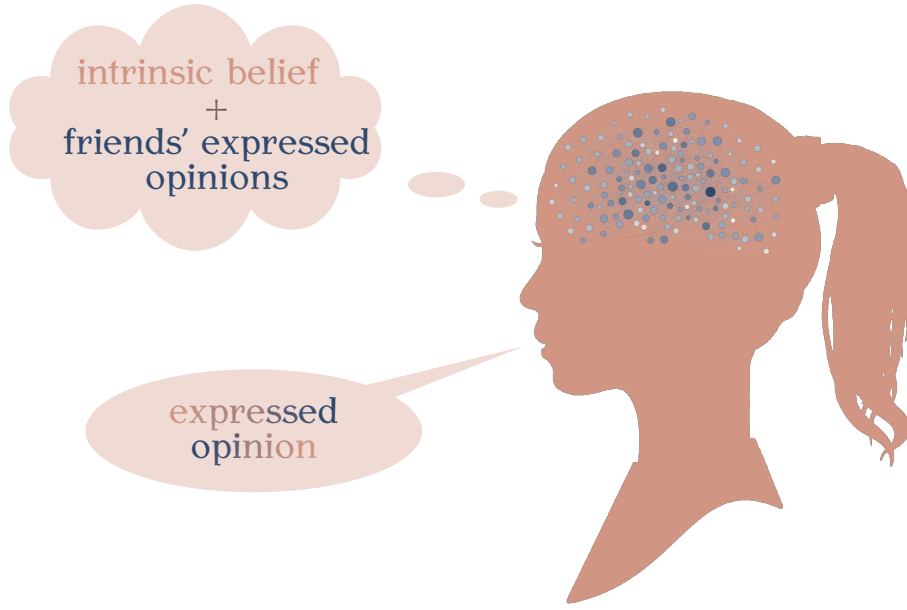


Figure 38: Illustration of the model of [48]. Individuals' hold internal opinions and form their expressed opinions by additionally considering their friends' expressed opinions due to social interaction.

itions regarding the source and presence of social influence. Furthermore, we initialize instances of games using real data and use repeated averaging to calculate their Nash equilibrium. We experimentally show that our model, when properly initialized, is able to mimic the original behavior of users and captures the social cost affecting their activity more accurately than a classification model utilizing the same information.

6.1 Model

We study a setting in which a group of individuals (also called users) are members of a social network, and investigate the impact of social influence on their opinions on some issue. We are concerned with users that perform, within this network, a certain action for the *first* time. Consider for example a social network in which a user can *endorse* articles and *get informed* about her friends' endorsements. We examine whether this information modifies the users' *opinions* on related subjects, e.g., their preferences in articles. We represent the social network as a *directed* graph G . Each node of G corresponds to a user, and there is an edge from node i to j iff user i gets informed about the actions of j .

We assume that if a user endorses an article after some friend of her has done so, the endorsement results from influence. In order to identify adjustments in users' opinions, we have to observe the system for a certain period of time. We keep trace of the endorsements of each user (from which we infer her opinion) and the endorsements of their friends, and compare their opinions in the initial state of the system to their opinions in the final state. The comparison illustrates whether some opinion has changed under social

influence.

We model the users' opinions using the notions of [16] and [48]. Each user i maintains a persistent intrinsic belief s_i and an overall (or expressed) opinion z_i : s_i remains constant, while z_i is updated iteratively, during the game, through averaging. In what follows, *opinion* refers to the overall opinion. We assume that for each user i that endorsed some article prior to all of her friends it is $s_i = 1$, otherwise $s_i = 0$. The same heuristic is used in [29]. z_i is a real number, representing the probability that i endorses the article. At each time step user i updates z_i to minimize the social cost of disagreement with her friends, using the formula:

$$z_i = \frac{s_i + \sum_{j \in N(i)} w_{ij} z_j}{1 + \sum_{j \in N(i)} w_{ij}} \quad (6.1)$$

where $N(i)$ denotes the set of nodes that i follows and w_{ij} expresses the strength of the influence of j on i . According to our intuition, the influence of j on i regarding a specific article is strong if i generally respects j 's opinion and/or j is authoritative on the article under consideration. We therefore define $w_{ij} = a_{ij} b_j$, where a_{ij} expresses how much j influences i in general, and b_j expresses the expertise of user j on the topic of the article. The weights w_{ij} are used to distinguish between close and distant friends, and are real numbers that may be greater than 1, as a user may value the opinion of a friend higher than her intrinsic belief.

A user's opinion in a social network may also change due to reasons other than social influence. Our model does not capture those cases.

6.2 Empirical analysis

We studied the behavior of users in a social network and the impact of social influence on their actions in its context by analyzing a sample of Digg¹, a social news aggregator, to which we will refer to as the *digg* dataset in the following [75].

Digg allows users to submit links to news stories and vote them up (*digg*). A user is also able to follow other members and track the *stories* they recently voted for. The *digg* dataset consists of the votes of the 3,553 most popular stories of June 2009, and the directed social graph depicting the followers of each voter. A total of 3,018,196 votes from 139,409 users and 1,731,658 follower edges is available. The probability distribution of the users' follows as well as the distribution of votes per user are heavy-tailed, indicating that most of the activity can be attributed to a small number of users. Such heavy-tailed activity patterns have been tightly connected with many aspects of human behavior [13].

¹Digg: <http://digg.com>

6.2.1 Information propagation and reproductive ratio

Information can travel through many paths in a social network and identifying *word-of-mouth* hops that form social cascades is a rather infeasible task. To differentiate the users of *digg* who endorsed a *story* due to social influence from the ones that acted freely, we adopt the heuristic used in [29] and consider an endorsement to have propagated from user i to user j if j endorsed a *story* after i did, and j followed i before endorsing the *story*. If multiple users i satisfy these conditions, we assume that the propagation was caused by all of them. Those users j that endorsed a *story* having no users i influencing them are considered as *seeders*.

In epidemiological models, the *reproductive ratio*, denoted R_0 , is used to measure the potential for disease spread in a population [9]. If $R_0 > 1$, an infected individual is expected to infect more than one other individuals and the infection will be able to spread in a population, otherwise the infection will die out. We observe that in more than 92% of the stories of the *digg* dataset the total ‘infections’ were less than the number of initial seeders, indicating that the reproductive ratio is well below 1.

We also examine the distribution of the total cascades caused by every individual for every *story* of the dataset, which appears to be heavy-tailed. This verifies our intuition about the presence of authoritative users, although the average transmission probability is quite small. However, it is worth noting that influence varies depending on the *story*, and even the users that frequently trigger cascades tend to be more effective in certain stories than in others. Therefore a good estimation of b_j can only occur when examining it in the context of a *single topic*.

6.2.2 Frequent cascade patterns

To further study the complex collective behavior attributed to the interaction of social network users, we mined the frequent cascade patterns occurring in the *digg* dataset. We picked 50 *stories* at random and formed a graph by creating a node for every $\{voter_id, story_id\}$ pair and an edge from v_{ir} to v_{jr} if voter i endorsed a story r before voter j did. Figure 39 illustrates the 20 most frequent cascade patterns met, extracted using Grami [40].

We observe that the spread of information for the *stories* of our dataset exhibits small chain- and tree-like cascades, as was the case with most of the datasets examined in [78]. However, splits are much more infrequent than collisions in our dataset, as opposed to the datasets of [78]. This is also an indication of a relatively small *reproductive ratio* R_0 in *digg*.



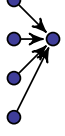
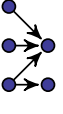



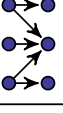



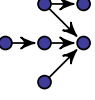






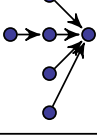
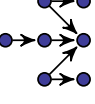
Rank	Pattern	Rank	Pattern	Rank	Pattern	Rank	Pattern
1		6		11		16	
2		7		12		17	
3		8		13		18	
4		9		14		19	
5		10		15		20	

Figure 39: Top-20 cascades that occurred in 50 randomly selected *stories* of the *digg* dataset, ordered by frequency.

6.3 Experimental evaluation

Having verified our intuitions regarding the adoption of an opinion due to social influence, we apply (6.1) on real-world data to examine its fitting performance with respect to our findings. We conduct experiments on the cascade graphs of the *digg* dataset *stories* to answer:

- How much more improved is the precision of (6.1) when distinguishing the crucial aspects of social interaction related to the spread of influence?
- How does (6.1) perform against a linear regression model?

6.3.1 Simulation methodology

We perform repeated averaging in our model until it converges to the unique Nash equilibrium. To initialize our model we apply the following set-up for each *story* we examine:

- We consider that every user of *digg* that endorsed a *story* before any of the members she follows did so, has a strongly positive intrinsic opinion about it. However, we cannot hypothesize on the intrinsic opinion of users that voted up a *story* after at least

Algorithm 9: Repeated Averaging algorithm

```

1 initialization of  $s_i$ ,  $a_{ij}$ , and  $b_j$  for each  $i, j$ ;
2 foreach  $i$  do  $z_i = s_i$ ;
3 while not converged do
4   foreach  $i$  do  $z_i^{new} = \frac{s_i + \sum_{j \in N(i)} a_{ij} * b_j * z_j}{1 + \sum_{j \in N(i)} a_{ij} * b_j}$ ;
5    $z_i = z_i^{new}$ 
6 end
7 for  $threshold \leftarrow 0$  to 1 do
8   calculate recall and precision;
9 end
10 plot the precision-recall curve;

```

one of the users they follow did so, as their behavior can be attributed to numerous causes. Hence, we consider for user i : $s_i = 1$, if i voted a story before any user she follows, and 0 otherwise.

- Regarding the influential strength of j on i , w_{ij} , we consider two variants:
 - (i) A straight-forward approach where users are equally authoritative on all *stories* of *digg*, i.e., $b_j = 1$ for every j . Additionally, users are equally influenced by all the members they follow and are not influenced at all by the rest of the users, i.e., $a_{ij} = 1$ if user i follows user j , and 0 otherwise.
 - (ii) An approach that follows our intuition that the influence of j on i increases with the ratio of votes of j that i followed and builds on the findings reported during our empirical analysis. We specify a_{ij} by how frequently j influences i , using information about the total influence of j on i to compute the influence on a certain story:

$$a_{ij} = \frac{\text{\# times } i \text{ is influenced by } j}{\text{\# votes of } j} \quad (6.2)$$

Moreover, we quantify b_j by how authoritative user j is for the article under consideration:

$$b_j = \frac{\text{\# users influenced by } j \text{ in this story}}{\text{\# followers of } j}, \quad (6.3)$$

thus, capturing the expertise of each user per story.

Algorithm 9 outlines our approach. We perform repeated averaging with both configurations until we reach convergence, to calculate the unique Nash equilibrium of the corresponding games. At the state of convergence, the expressed opinions of the users are given values in $[0, 1]$. Deciding whether a value stands for endorsement or not calls for the use of a threshold. We examine the trade-off between *precision* and *recall* by varying the threshold value to obtain the respective curves, where *precision* is the fraction of users predicted as endorsers that actually voted up, while *recall* is the fraction of users that voted up that are predicted to do so.

Algorithm 10: compute

```

1  $sumw \leftarrow 0$ ;
2  $sumwz \leftarrow 0$ ;
3 foreach  $(z, w) \in \text{messages}$  do
4    $sumw \leftarrow sumw + w$ ;
5    $sumwz \leftarrow sumwz + w \cdot z$ 
6 end
7  $diff \leftarrow |value.left - \frac{value.right + sumwz}{1 + sumw}|$ ;
8  $value.left \leftarrow \frac{value.right + sumwz}{1 + sumw}$ ;
9 aggregate( $DIFF, diff$ );
10 if  $getAggregatedValue(DIFF) < tolerance$  then
11   voteToHalt();
12 end
13 else
14   sendMessageToAllInEdges( $value.left, edge.weight$ );
15 end

```

Distributed algorithm: Applying graph algorithms on real-world networks is often prohibitive because of the massive volume that the latter may reach. *Pregel* [93] is a computational model suitable for large scale graph processing due to its vertex-centric approach. Pregel encourages programmers to “*think like a vertex*”, and distributes the vertices of the network, and along with them the execution of the algorithm, among the machines of a computing cluster.

Algorithm 10 is a Pregel version of Algorithm 9 and details the actions that need to be taken by each vertex in the network in every iteration (*superstep*) of the execution. We consider that each vertex $v \in V$ holds a pair of values (z_v, s_v) , where z_v is the *opinion* of v and s_v the internal opinion of v . Moreover, the edges of the network are weighted.

The node initially calculates its updated value according to Eq. (6.1), as well as its difference from the previous value of the node (Lines 1-7). In the first superstep no messages have been received and thus, z_i is equal to s_i . However, in subsequent supersteps the value is updated according to the averaged values of the nodes’ neighbors. After, the new value is calculated the node provides the difference to an aggregator (Line 8), that allows for checking whether a condition is satisfied in the whole graph. In our case the condition is whether the total difference in the previous superstep is smaller than the specified *tolerance*, and thus, we can assume that we have reached convergence (Line 9). If so, the nodes vote to halt and the algorithm terminates (Line 10). If not, the nodes send their new values to all nodes that have an edge towards them, and the process continues until we reach convergence (Lines 11-12).

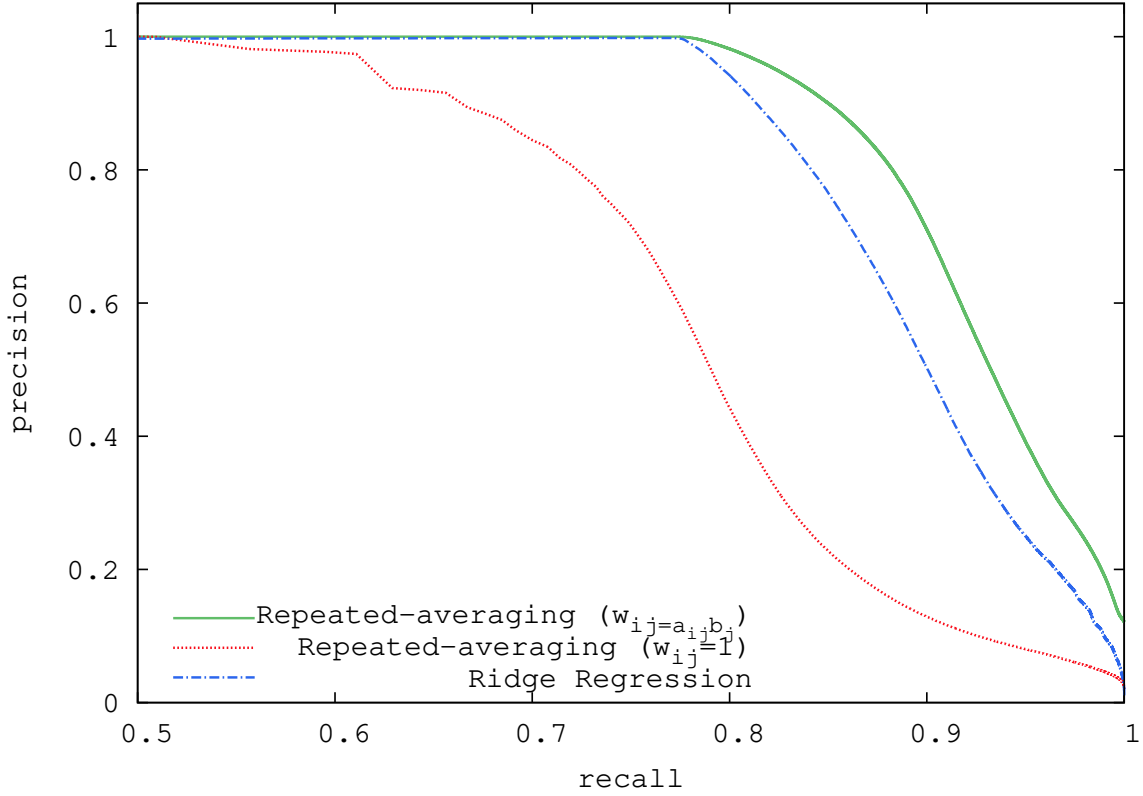


Figure 40: Cumulative *precision/recall* curve for the two configurations of our model and a *Ridge regression* classifier for all the *stories* of our dataset.

Ridge regression: We also use *Ridge regression*² to train a regression model that predicts user actions and compare against it. To this end, we use the following independent variables: (i) s_i , (ii) the sum of (6.2) for every j friend of i , and (iii) the sum of (6.3) for every j friend of i .

6.3.2 Experiments using real-world data

We performed extensive simulations using our methodology on different stories of the *digg* dataset, and obtained *precision-recall* curves for the two configurations of our model. Our distributed implementation with Spark’s GraphX [56] is publicly available.³ Furthermore, we obtained the respective curves that occur through the use of the *Ridge regression* model.

Figure 40 illustrates a cumulative *precision/recall* curve for all the *stories* of our dataset. In particular, we have applied the two configurations of our model to each *story* of our dataset and used the predicted values to come up with a *precision/recall* curve for all of

²We employ ***sklearn.linear_model.Ridge***: <https://tinyurl.com/yd7v5qky>.

³<https://bitbucket.org/network-analysis/social-cost>

them. It is obvious that the second configuration of our model captures much more accurately the true activity of *digg* in comparison with our simplistic setting. For instance, for 90% *recall* our model achieves on average about 58.12 percentage points higher precision. This verifies our belief that the opinions of all users in *digg* can be approximated by applying (6.1), given that the social influence imposed by users of the network is weighted appropriately. We note that the different *stories* may vary in terms of popularity or number of cascades. However, the second configuration consistently outperforms the first one significantly, regardless of these properties.

Moreover, we see in Figure 40 that our model consistently outperforms the *Ridge regression* model, and the improvement remains relatively stable as the desired recall increases. In particular, we observe that for low *recall* the *Ridge regression* model behaves similarly with our model, as both predict exclusively users with positive intrinsic opinion will vote a *story*. However, as we adjust the threshold to acquire higher *recall*, the trade-off with precision is worse for the *Ridge regression* model. For instance, for 90% *recall* our model achieves on average about 41.33% better *precision*, i.e., 20.77 percentage points higher precision. This improvement is due to the fact that our model additionally captures the opinion adjustments during the averaging process.

We note here that the deviation from the original activity of *digg* for all methods is not surprising. Our hypothesis regarding the internal opinion of its users for each *story* is convenient for conducting experiments but may well be mistaken for quite a lot of them.

6.4 Conclusion

In this work we specified the details which allow for a better formalization of the social effects on a variation of the DeGroot model. To verify our intuitions on the causes of social influence, we performed a comprehensive analysis on a real-life popular social network. In addition to this, we initialized several instances of the dataset and applied repeated averaging on them using a distributed graph processing algorithm, to calculate the state where our model converges. As it is shown in [16], this state is the unique Nash equilibrium of the game defined by the individual cost functions. We presented results comparing this state with the original actions of the network's members. Our findings show that a properly initialized instance following our model, converges to a Nash equilibrium that closely mimics the original social activity of a real-world dataset. Therefore, we verified that users act according to the social cost described above.

7. CONCLUSION AND OPEN DIRECTIONS

In this thesis we study two research directions that allow for handling large-scale graphs, i.e., *distributed graph processing* and *streaming graph algorithms*. Our focus is on improving contemporary distributed systems, introducing novel techniques for important and challenging graph processing problems, and employing scalable platforms to empirically study real-world networks.

In Chapter 2 we discuss how we can extend Apache Giraph with novel memory-optimized structures that are applicable to any distributed graph compressing system that follows the Pregel paradigm. Our representations are able to execute algorithms over large-scale graphs under very modest settings and greatly outperform earlier representations when memory is an issue.

Chapters 3 and 4 focus on the problem of community detection and propose a vertex-centric and a streaming technique, respectively. Both approaches consider the setting of seed-set expansion where a number of small seed-sets of nodes is given as an input and the challenge is to expand these sets into communities. Our results show that our techniques offer impressive improvements over the state-of-the-art with regards to accuracy, execution time and memory usage.

In Chapter 5 we deal with another challenge that arises when dealing with a graph stream. In particular, we address the task of sampling the content posted by authoritative social network users from a stream of social activity. To the best of our knowledge our approach is the first to consider a dynamic setting and we are able to outperform previously proposed white-list based methods with regards to all recall, precision and ranking accuracy.

Finally, in Chapter 6 we empirically analyze a popular social network and implement a distributed graph processing algorithm to calculate the state where a well-studied opinion formation model converges. Our findings show that when initializing our model properly, it converges to a Nash equilibrium that closely mimics the original social activity of a real-world dataset.

Large-scale graph processing poses many challenges and thus, there are numerous possible directions for future research. Naturally, improving existing results would be interesting. In addition, distributed graph processing systems depend on much more than just the in-memory representations of graphs. For example the messages that are exchanged during the iterations of distributed algorithm execution also consume a significant amount of memory. Therefore, limiting memory requirements of Pregel-like systems with regards to aspects other than the structures depicting the graphs' elements is also desirable. Regarding our community detection techniques, we believe that a drift from the currently available ground-truth communities depicting metadata groups [63] to communities that better portray the functional roles of a network's nodes would be an interesting future direction. Such communities will allow for a more accurate comparison of community detection techniques. To this end, we can collect data from social network groups where membership signifies affinity. Finally, with regard to our effort on opinion formation, hav-

ing verified that opinion dynamics of a real life social network can be accurately captured through network interaction models, we can proceed with estimating the price of anarchy in such a network. Furthermore, we can examine ways to reduce the social cost of these networks.

Hopefully, these directions will be explored in our future research.

ABBREVIATIONS - ACRONYMS

BSP	Bulk synchronous parallel
CoEuS	Community detection via seed-set Expansion on graph Streams
JVM	Java virtual machine
LALP	Large adjacency list partitioning
LDLC	Local Dispersion-aware Link Communities
MPI	Message Passing Interface
NDCG	Normalized discounted cumulative gain
SNAP	Stanford Network Analysis Project

REFERENCES

- [1] Apache Giraph. <http://giraph.apache.org/>.
- [2] Stanford Network Analysis Project. <https://snap.stanford.edu/>.
- [3] We knew the web was big.... <http://googleblog.blogspot.ca/2008/07/we-knew-web-was-big.html>.
- [4] Lada A Adamic and Bernardo A Huberman. Power-law distribution of the world wide web. *science*, 287(5461):2115–2115, 2000.
- [5] Charu C. Aggarwal and Karthik Subbian. Event detection in social streams. In *SDM 2012*, pages 624–635.
- [6] Eugene Agichtein, Carlos Castillo, Debora Donato, Aristides Gionis, and Gilad Mishne. Finding high-quality content in social media. In *WSDM 2008*, pages 183–194.
- [7] Yong-Yeol Ahn, James P Bagrow, and Sune Lehmann. Link communities reveal multiscale complexity in networks. *Nature*, 466(7307):761–764, 2010.
- [8] Aris Anagnostopoulos, Ravi Kumar, and Mohammad Mahdian. Influence and correlation in social networks. In *SIGKDD*, pages 7–15, Las Vegas, Nevada, USA, 2008.
- [9] Roy M Anderson and Robert McCredie May. *Infectious diseases of humans*. Oxford University Press, 1991.
- [10] Alberto Apostolico and Guido Drovandi. Graph compression by BFS. *Algorithms*, 2(3):1031–1044, 2009.
- [11] M. D. Atkinson, Jörg-Rüdiger Sack, Nicola Santoro, and Thomas Strothotte. Min-max heaps and generalized priority queues. *Commun. ACM*, 29(10):996–1000, 1986.
- [12] Lars Backstrom and Jon Kleinberg. Romantic partnerships and the dispersion of social ties: A network analysis of relationship status on facebook. In *Proc. of the 17th ACM Conf. on Computer Supported Cooperative Work & Social Computing*, pages 831–841, 2014.
- [13] Albert-Laszlo Barabasi. The origin of bursts and heavy tails in human dynamics. *Nature*, 435(7039):207–211, 2005.
- [14] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.
- [15] A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani. The architecture of complex weighted networks. *Proc. of the National Academy of Sciences of the United States of America*, 101(11):3747–3752, 2004.
- [16] David Bindel, Jon M. Kleinberg, and Sigal Oren. How bad is forming your own opinion? In *FOCS*, pages 57–66, 2011.
- [17] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.
- [18] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *Proc. of the 20th Int. Conf. on World Wide Web, Hyderabad, India, March 28 - April 1*, pages 587–596, 2011.
- [19] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. Permuting web and social graphs. *Internet Mathematics*, 6(3):257–283, 2009.
- [20] Paolo Boldi and Sebastiano Vigna. The webgraph framework I: compression techniques. In *Proc. of the 13th Int. Conf. on World Wide Web, New York, NY, USA, May 17-20*, pages 595–602, 2004.
- [21] Paolo Boldi and Sebastiano Vigna. The webgraph framework II: codes for the world-wide web. In *Proc. of the 2004 Data Compression Conference, March 23-25, Snowbird, UT, USA*, page 528, 2004.
- [22] Robert M Bond, Christopher J Fariss, Jason J Jones, Adam DI Kramer, Cameron Marlow, Jaime E Settle, and James H Fowler. A 61-million-person experiment in social influence and political mobilization. *Nature*, 489(7415):295–298, 2012.

- [23] Mohamed Bouguessa and Lotfi Ben Romdhane. Identifying authorities in online communities. *ACM Trans. Intell. Syst. Technol.*, 6(3):30:1–30:23, 2015.
- [24] Alessandro Bozzon, Marco Brambilla, Stefano Ceri, Matteo Silvestri, and Giuliano Vesci. Choosing the right crowd: expert finding in social networks. In *EDBT '13*, pages 637–648.
- [25] Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks*, 30(1-7):107–117, 1998.
- [26] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. k2-Trees for Compact Web Graph Representation. In *String Processing and Information Retrieval*, volume 5721, pages 18–30. 2009.
- [27] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. *Computer networks*, 33(1):309–320, 2000.
- [28] Zhuhua Cai, Zekai J. Gao, Shangyu Luo, Luis Leopoldo Perez, Zografoula Vagena, and Christopher M. Jermaine. A comparison of platforms for implementing and running very large scale machine learning algorithms. In *Proc. of the Int. Conf. on Management of Data, Snowbird, UT, USA, June 22-27*, pages 1371–1382, 2014.
- [29] Meeyoung Cha, Alan Mislove, and Krishna P Gummadi. A measurement-driven analysis of information propagation in the flickr social network. In *WWW*, pages 721–730, 2009.
- [30] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73(2):129–174, 1996.
- [31] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *Proc. of the 15th Int. Conf. on Knowledge Discovery and Data Mining, Paris, France, June 28 - July 1*, pages 219–228, 2009.
- [32] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One Trillion Edges: Graph Processing at Facebook-Scale. *Proc. of the VLDB Endowment*, 8(12):1804–1815, 2015.
- [33] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.
- [34] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.
- [35] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.
- [36] Michele Coscia, Giulio Rossetti, Fosca Giannotti, and Dino Pedreschi. DEMON: a local-first discovery method for overlapping communities. In *Proc. of the 18th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 615–623, 2012.
- [37] Ithiel de Sola Pool and Manfred Kochen. Contacts and influence. *Social networks*, 1(1):5–51, 1978.
- [38] Morris H DeGroot. Reaching a consensus. *Journal of the ASA*, 69(345):118–121, 1974.
- [39] Christopher DuBois. StackOverflow Data. <https://www.ics.uci.edu/~dubois/stackoverflow/>, jun 2009.
- [40] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment*, 2014.
- [41] TS Evans and R Lambiotte. Line graphs, link partitions, and overlapping communities. *Physical Review E*, 80:016105, 2009.
- [42] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *ACM SIGCOMM computer communication review*, volume 29, pages 251–262. ACM, 1999.
- [43] Scott L Feld. The focused organization of social ties. *American journal of sociology*, pages 1015–1035, 1981.
- [44] Diane H Felmlee. No couple is an island: A social network perspective on dyadic stability. *Social Forces*, 79(4):1259–1287, 2001.
- [45] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.
- [46] Santo Fortunato and Marc Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007.

- [47] Linton C Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.
- [48] Noah E Friedkin and Eugene C Johnsen. Social influence and opinions. *Journal of Mathematical Sociology*, 15(3-4):193–206, 1990.
- [49] Saptarshi Ghosh, Naveen Kumar Sharma, Fabrício Benevenuto, Niloy Ganguly, and P. Krishna Gummadi. Cognos: crowdsourcing search for topic experts in microblogs. In *SIGIR '12*, pages 575–590.
- [50] Saptarshi Ghosh, Muhammad Bilal Zafar, Parantapa Bhattacharya, Naveen Kumar Sharma, Niloy Ganguly, and P. Krishna Gummadi. On sampling the wisdom of crowds: random vs. expert sampling of the twitter stream. In *CIKM'13*, pages 1739–1744.
- [51] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proc. of the National Academy of Sciences*, 99(12):7821–7826, 2002.
- [52] David F. Gleich and Michael W. Mahoney. Mining large graphs. In Peter Bühlmann, Petros Drineas, Michael Kane, and Mark van de Laan, editors, *Handbook of Big Data*, Handbooks of modern statistical methods, pages 191–220. CRC Press, 2016.
- [53] David F Gleich and C Seshadhri. Vertex neighborhoods, low conductance cuts, and good seeds for local community methods. In *Proc. of the 18th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 597–605, 2012.
- [54] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proc. of the 10th USENIX Symposium on Operating Systems Design and Implementation, Hollywood, CA, USA, October 8-10*, pages 17–30, 2012.
- [55] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proc. of the 11th USENIX Conference on Operating Systems Design and Implementation*, pages 599–613, Berkeley, CA, USA, 2014.
- [56] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 599–613, 2014.
- [57] Oshini Goonetilleke, Danai Koutra, Timos Sellis, and Kewen Liao. Edge labeling schemes for graph data. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, SSDBM '17*, pages 12:1–12:12, New York, NY, USA, 2017. ACM.
- [58] Mark S Granovetter. The strength of weak ties. *American journal of sociology*, pages 1360–1380, 1973.
- [59] Minyang Han and Khuzaima Daudjee. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *Proc. VLDB Endow.*, 8(9):950–961, May 2015.
- [60] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, and Tianqi Jin. An Experimental Comparison of Pregel-like Graph Processing Systems. *Proc. of the VLDB Endowment*, 7(12):1047–1058, 2014.
- [61] Kun He, Yiwei Sun, David Bindel, John E. Hopcroft, and Yixuan Li. Detecting overlapping communities from local spectral subspaces. In *IEEE International Conference on Data Mining, Atlantic City, NJ, USA*, pages 769–774, 2015.
- [62] A. Holloco, J. Maudet, T. Bonald, and M. Lelarge. A linear streaming algorithm for community detection in very large networks. *ArXiv e-prints*, March 2017.
- [63] Darko Hric, Richard K Darst, and Santo Fortunato. Community detection in networks: Structural communities versus ground truth. *Physical Review E*, 90(6):062805, 2014.
- [64] Pawel Jurczyk and Eugene Agichtein. Discovering authorities in question answer communities by using link analysis. In *CIKM 2007*, pages 919–922.
- [65] Maja Kabiljo, Dionysis Logothetis, Sergey Edunov, and Avery Ching. A comparison of state-of-the-art graph processing systems. <https://code.facebook.com/posts/319004238457019/a-comparison-of-state-of-the-art-graph-processing-systems/>.
- [66] U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. Gbase: an efficient analysis platform for large graphs. *VLDB J.*, 21(5):637–650, 2012.

- [67] Herbert C Kelman. Compliance, identification, and internalization: Three processes of attitude change. *Journal of conflict resolution*, pages 51–60, 1958.
- [68] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 49(2):291–307, 1970.
- [69] Jon M. Kleinberg. Hubs, authorities, and communities. *ACM Comput. Surv.*, 31(4es):5, 1999.
- [70] Kyle Kloster and David F. Gleich. Heat kernel based community detection. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, New York, NY, USA, pages 1386–1395, 2014.
- [71] Isabel M. Kloumann and Jon M. Kleinberg. Community membership identification from small seed sets. In *Proc. of the 20th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 1366–1375.
- [72] Isabel M. Kloumann and Jon M. Kleinberg. Community membership identification from small seed sets. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 1366–1375, 2014.
- [73] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10*, pages 31–46.
- [74] Page Lawrence, Brin Sergey, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford University, 1998.
- [75] Kristina Lerman, Rumi Ghosh, and Tawan Surachawala. Social contagion: An empirical study of information spread on digg and twitter follower graphs. *arXiv preprint arXiv:1202.3162*, 2012.
- [76] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *Proc. of the 11th Int. Conf. on Knowledge Discovery and Data Mining, Chicago, Illinois, USA, August 21-24*, pages 177–187, 2005.
- [77] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Statistical properties of community structure in large social and information networks. In *Proc. of the 17th Int. Conf. on World Wide Web, WWW '08*, pages 695–704, 2008.
- [78] Jure Leskovec, Ajit Singh, and Jon Kleinberg. Patterns of influence in a recommendation network. In *Advances in Knowledge Discovery and Data Mining*, pages 380–389, 2006.
- [79] Yixuan Li, Kun He, David Bindel, and John E Hopcroft. Uncovering the small community structure in large networks: A local spectral approach. In *Proc. of the 24th Int. Conf. on World Wide Web*, pages 658–668, 2015.
- [80] Panagiotis Liakos, Alexandros Ntoulas, and Alex Delis. Scalable link community detection: A local dispersion-aware approach. In *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*, pages 716–725, 2016.
- [81] Panagiotis Liakos, Alexandros Ntoulas, and Alex Delis. COEUS: community detection via seed-set expansion on graph streams. In *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*, pages 676–685, 2017.
- [82] Panagiotis Liakos, Alexandros Ntoulas, and Alex Delis. Rhea: Adaptively sampling authoritative content from social activity streams. In *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*, pages 686–695, 2017.
- [83] Panagiotis Liakos and Katia Papakonstantinou. On the impact of social cost in opinion dynamics. In *Proceedings of the Tenth International Conference on Web and Social Media, Cologne, Germany, May 17-20, 2016.*, pages 631–634, 2016.
- [84] Panagiotis Liakos, Katia Papakonstantinou, and Alex Delis. Memory-optimized distributed graph processing through novel compression techniques. In *Proc. of the 25th ACM Int. Conf. on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*, pages 2317–2322.
- [85] Panagiotis Liakos, Katia Papakonstantinou, and Alex Delis. Realizing memory-optimized distributed graph processing. *IEEE Trans. Knowl. Data Eng.*, 30(4):743–756, 2018.
- [86] Panagiotis Liakos, Katia Papakonstantinou, and Michael Sioutis. On the effect of locality in compressing social networks. In *Proc. of the 36th Eur. Conf. on IR Research, Amsterdam, The Netherlands, April 13-16*, pages 650–655, 2014.

- [87] Panagiotis Liakos, Katia Papakonstantinou, and Michael Sioutis. Pushing the Envelope in Graph Compression. In *Proc. of the 23rd ACM Int. Conf. on Information and Knowledge Management*, pages 1549–1558, Shanghai, China, 2014.
- [88] Hang Liu and H. Howie Huang. Graphene: Fine-grained io management for graph computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 285–300, Santa Clara, CA, 2017. USENIX Association.
- [89] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *Proc. of the VLDB Endowment*, 5(8):716–727, 2012.
- [90] Lu Lu, Xuanhua Shi, Yongluan Zhou, Xiong Zhang, Hai Jin, Cheng Pei, Ligang He, and Yuanzhen Geng. Lifetime-based memory management for distributed data processing systems. *PVLDB*, 9(12):936–947, 2016.
- [91] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. Large-scale distributed graph computing systems: An experimental evaluation. *Proc. VLDB Endow.*, 8(3):281–292, November 2014.
- [92] Michael W. Mahoney, Lorenzo Orecchia, and Nisheeth K. Vishnoi. A local spectral method for graphs: With applications to improving graph partitions and exploring data graphs locally. *J. Mach. Learn. Res.*, 13(1):2339–2365, August 2012.
- [93] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data, Indianapolis, Indiana, USA, June 6-10*, pages 135–146, 2010.
- [94] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.
- [95] Peter V Marsden and Karen E Campbell. Measuring tie strength. *Social forces*, 63(2):482–501, 1984.
- [96] Mary McGlohon, Leman Akoglu, and Christos Faloutsos. Weighted graphs and disconnected components: patterns and a generator. In *Proc. of the 14th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008*, pages 524–532.
- [97] Andrew McGregor. Graph stream algorithms: a survey. *SIGMOD Record*, 43(1):9–20.
- [98] Ahmed Metwally, Jia-Yu Pan, Minh Doan, and Christos Faloutsos. Scalable community discovery from multi-faceted graphs. In *IEEE Int. Conf. on Big Data, Santa Clara, CA, USA*, pages 1053–1062, 2015.
- [99] Joseph M Morris. Traversing binary trees simply and cheaply. *Information Processing Letters*, 9(5):197–200, 1979.
- [100] Fred Morstatter, Jürgen Pfeffer, Huan Liu, and Kathleen M. Carley. Is the Sample Good Enough? Comparing Data from Twitter’s Streaming API with Twitter’s Firehose. In *ICWSM 2013*.
- [101] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69(2):026113, February 2004.
- [102] Newman, M. E.J. Detecting community structure in networks. *Eur. Phys. J. B*, 38(2):321–330, 2004.
- [103] Aditya Pal and Scott Counts. Identifying topical authorities in microblogs. In *WSDM 2011*, pages 45–54.
- [104] Deepan Subrahmanian Palguna, Vikas Joshi, Venkatesan T. Chakaravarthy, Ravi Kothari, and L. Venkata Subramaniam. Analysis of sampling algorithms for twitter. In *IJCAI 2015*, pages 967–973.
- [105] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043):814–818, 2005.
- [106] Odysseas Papapetrou, Minos N. Garofalakis, and Antonios Deligiannakis. Sketch-based querying of distributed sliding-window data streams. *PVLDB*, 5(10):992–1003, 2012.
- [107] Pascal Pons and Matthieu Latapy. Computing communities in large networks using random walks. In *Computer and Information Sciences-ISCIS 2005*, pages 284–293. 2005.
- [108] Keith H. Randall, Raymie Stata, Janet L. Wiener, and Rajiv Wickremesinghe. The link database: Fast access to graphs of the web. In *Proc. of the 2002 Data Compression Conference, 2-4 April, Snowbird, UT, USA*, pages 122–131, 2002.
- [109] Sidney Redner. How popular is your paper? an empirical study of the citation distribution. *The European Physical Journal B-Condensed Matter and Complex Systems*, 4(2):131–134, 1998.

- [110] Martin Rosvall and Carl T Bergstrom. Multilevel compression of random walks on networks reveals hierarchical organization in large integrated systems. *PloS one*, 6(4):e18209, 2011.
- [111] Jan Rybak, Krisztian Balog, and Kjetil Nørnvåg. Expertise: tracking expertise over time. In *SIGIR '14*, pages 1273–1274.
- [112] Semih Salihoglu and Jennifer Widom. GPS: a graph processing system. In *Proc. of the 25th Int. Conf. on Scientific and Statistical Database Management, Baltimore, MD, USA, July 29 - 31, 2013*, pages 22:1–22:12, 2013.
- [113] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. of the 25th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, August 11-15, 2002, Tampere, Finland*, pages 222–229.
- [114] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *2015 Data Compression Conference, DCC 2015, Snowbird, UT, USA, April 7-9*, pages 403–412, 2015.
- [115] Johan Ugander and Lars Backstrom. Balanced Label Propagation for Partitioning Massive Graphs. In *Proc. of the 6th ACM Int. Conf. on Web Search and Data Mining, Rome, Italy, February 4-8*, pages 507–516, 2013.
- [116] Claudia Wagner, Vera Liao, Peter Pirolli, Les Nelson, and Markus Strohmaier. It's not in their tweets: Modeling topical expertise of twitter users. In *PASSAT 2012, and SocialCom 2012*, pages 91–100.
- [117] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, and Tie-Yan Liu. A theoretical analysis of NDCG type ranking measures. In *COLT 2013*, pages 25–54.
- [118] Stanley Wasserman and Katherine Faust. *Social network analysis: Methods and applications*, volume 8. Cambridge University Press, 1994.
- [119] Joyce Jiyoung Whang, David F Gleich, and Inderjit S Dhillon. Overlapping community detection using seed set expansion. In *Proc. of the 22nd ACM Int. Conf. on Information & Knowledge Management*, pages 2099–2108, 2013.
- [120] Hugh E. Williams and Justin Zobel. Compressing integers for fast file access. *Comput. J.*, 42(3):193–201, 1999.
- [121] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Effective Techniques for Message Reduction and Load Balancing in Distributed Graph Computation. In *Proc. of the 24th Int. Conf. on World Wide Web, Florence, Italy, May 18-22, 2015*, pages 1307–1317, 2015.
- [122] Jaewon Yang and Jure Leskovec. Patterns of temporal variation in online media. In *WSDM 2011*, pages 177–186.
- [123] Jaewon Yang and Jure Leskovec. Community-affiliation graph model for overlapping network community detection. In *Proc. of the 12th IEEE International Conference on Data Mining*, pages 1170–1175, 2012.
- [124] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *Proc. of the 12th IEEE International Conference on Data Mining*, pages 745–754, 2012.
- [125] Jaewon Yang and Jure Leskovec. Overlapping community detection at scale: a nonnegative matrix factorization approach. In *Proc. of the 6th ACM int. Conf. on Web Search and Data Mining*, pages 587–596, 2013.
- [126] Jaewon Yang and Jure Leskovec. Overlapping communities explain core–periphery organization of networks. *Proc. of the IEEE*, 102(12), 2014.
- [127] Jaewon Yang and Jure Leskovec. Structure and overlaps of ground-truth communities in networks. *ACM Transactions on Intelligent Systems and Technology*, 5(2):26, 2014.
- [128] Se-Young Yun, Marc Lelarge, and Alexandre Proutière. Streaming, memory limited algorithms for community detection. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3167–3175, 2014.
- [129] Muhammad Bilal Zafar, Parantapa Bhattacharya, Niloy Ganguly, Saptarshi Ghosh, and Krishna P. Gummadi. On the wisdom of experts vs. crowds: Discovering trustworthy topical news in microblogs. In *CSCW 2016*, pages 437–450.

- [130] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proc. of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10–10, Berkeley, CA, USA, 2010.
- [131] Anita Zakrzewska and David A. Bader. A dynamic algorithm for local community detection in graphs. In *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, ASONAM 2015, Paris, France, August 25 - 28, 2015*, pages 559–564, 2015.
- [132] Jun Zhang, Mark S. Ackerman, and Lada A. Adamic. Expertise networks in online communities: structure and algorithms. In *WWW 2007*, pages 221–230.
- [133] Angen Zheng, Alexandros Labrinidis, Panos K. Chrysanthis, and Jack Lange. Argo: Architecture-aware graph partitioning. In *2016 IEEE Int. Conf. on Big Data, Washington DC, USA, December 5-8, 2016*, pages 284–293, 2016.
- [134] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, Santa Clara, CA, 2015. USENIX Association.
- [135] Chang Zhou, Jun Gao, Binbin Sun, and Jeffrey Xu Yu. Mocgraph: Scalable distributed graph processing using message online computing. *Proc. VLDB Endow.*, 8(4):377–388, December 2014.
- [136] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4*, pages 301–316.