Brief Announcement: Non-Blocking Dynamic Unbounded Graphs with Worst-Case Amortized Bounds

Bapi Chatterjee
Institute of Science and Technology, Klosterneuburg, Austria

Sathya Peri
Indian Institute of Technology, Hyderabad, India

Muktikanta Sa
Télécom SudParis – Institut Polytechnique de Paris, France

Abstract
This paper reports a new concurrent graph data structure that supports updates of both edges and vertices and queries: Breadth-first search, Single-source shortest-path, and Betweenness centrality. The operations are provably linearizable and non-blocking.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms

Keywords and phrases concurrent data structure, linearizability, non-blocking, directed graph, breadth-first search, single-source shortest-path, betweenness centrality

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.52


Funding This work was partially funded by National Supercomputing Mission, Govt. of India under the project “Concurrent and Distributed Programming primitives and algorithms for Temporal Graphs”(DST/NSM/R&D_Exascale/2021/16).

1 Introduction
Dynamic graph data structures with concurrent query operations and updates can readily boost important real-world applications such as social networks [6], semantic-web [5], biological networks [10], blockchains [3], and many others. The existing libraries of graph queries, which support dynamic updates, for example, Stinger [12], GraphOne [16], GraphTinker [15], Kineograph [9], Graphitau [14], Kickstarter [18], Aspen [11], etc. face limitations such as blocking concurrency, no native support for vertex updates, and high memory-footprint.

In this paper, we describe the design and implementation of a graph data structure, which provides (a) three useful operations – breadth-first search (BFS), single-source shortest-path (SSSP), and betweenness centrality (BC), (b) dynamic updates of edges and vertices concurrent with the operations, (c) non-blocking progress with linearizability [13], and (d) a light memory footprint. We call it PANIGRAMHAM a: Practical Non-blocking Graph Algorithms. In a nutshell, we implement a concurrent non-blocking dynamic directed graph data structure as an adjacency-list formed by a composition of lock-free sets: a lock-free hash-table and multiple lock-free binary search trees (BSTs). The set of outgoing edges \( E_v \) from a vertex \( v \in V \) is implemented by a BST, whereas, \( v \) itself is a node of the hash-table. Addition/removal of a vertex translates to the same operation in the lock-free hash-table,

* Panigramham is the Sanskrit translation of Marriage, which undoubtedly is a prominent event in our lives resulting in networks represented by graphs.

© Bapi Chatterjee, Sathya Peri, and Muktikanta Sa; licensed under Creative Commons License CC-BY 4.0
35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 52; pp.52:1–52:4
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
Brief Announcement: Non-Blocking Dynamic Unbounded Graphs

To analyze the trade-off between consistency and performance, in addition to the presented framework interface operation for graph queries. We implement an ADT $\mathcal{Q}$ specialized to the requirements of the three queries $\text{BFS}$, $\text{SSSP}$, $\text{BC}$ – implemented by specialized partial snapshots. In a dynamic concurrent setting, we apply multi-scan/validate [1] to ensure the linearizability and non-blocking progress. We prove that these operations are non-blocking. The empirical results show the effectiveness of our algorithms.

Figure 1 Framework interface operation for graph queries.

whereas, addition/removal of an edge translates to the same operation in a lock-free BST. The operations – BFS, SSSP, BC – are implemented by specialized partial snapshots. In a dynamic concurrent setting, we apply multi-scan/validate [1] to ensure the linearizability of a partial snapshot. We prove that these operations are non-blocking. The empirical results show the effectiveness of our algorithms.

### 2 PANIGRAM

**Algorithm Overview.** We implement an ADT $\mathcal{A} = \mathcal{F} \cup \mathcal{P}$, wherein the set operations $\mathcal{F} := \{\text{PutV}, \text{RemV}, \text{GetV}, \text{PutE}, \text{RemE}, \text{GetE}\}$ use lock-free hash-table and BST and the queries $\mathcal{P} := \{\text{BFS}, \text{SSSP}, \text{BC}\}$ use partial snapshot. To de-clutter the presentation, we encapsulate the three queries in a unified framework with an interface operation $\text{Op}$ – presented in pseudo-code in Figure 1. $\text{Op}$ is specialized to the requirements of the three queries. We have explained the pseudo-code using line-comments in Figure 1. For detail of the ADT operations please see the full version [8], wherein we also present their proofs of linearizability and non-blocking progress.

**Experimental Results and Discussion.** We experimentally evaluate our non-blocking graph against two well-known existing batch analytics methods: (a) Stinger [12], and (b) Ligra [17]. To analyze the trade-off between consistency and performance, in addition to the presented...
Given a graph $G = (V,E)$, denote $|V| = n$, $|E| = m$, $\max_{v \in V}(\delta_v) = \delta$, where $\delta_v$ is the degree of vertex $v$. Define the (static) state of a graph $G$ as a tuple $S_G = (n,m,\delta)$. Let $X$ be a concurrent execution given as a set of operations. Thus, for an $o \in X$, type$(o) \in \mathcal{A}$, where type$(o)$ denotes the type of $o$ and $\mathcal{A}$ is the ADT. Let $I_o$ and $C_o$ be the internal contention [2] and point contention [4], respectively, for an $o \in X$. Denote $\tilde{I}_o = (I_o - 1)$, the total number of concurrent operation calls other than $o$ itself (those responsible for a possible cost escalation) that were invoked between the invocation and response of $o$. Denote the worst-case cost of $o$, given $o$ is invoked at an atomic time point when state of $G$ was $S_G$ by $W_{o,S_G}$. $W_{o,S_G}$ for each operation type is given in Table 1 of [8]. The states of $G$, being tuples, are ordered by dictionary order. In a dynamic setting, $W_{o,S_G}$ is upper-bounded by the worst-case cost of $o$ as performed in a static setting over the worst-case state, during the lifetime of $o$, of $G$. It can be shown that the worst-case state of $G$ that $o$ can encounter is $S_{G,o} = (O(n + \tilde{I}_o), O(m + \tilde{I}_o), O(\delta + \tilde{I}_o))$. 

Figure 2 Latency of the executions containing OP: (1) BFS ((a), (b), and (c)) on a graph of size $|V| = 131K$ and $|E| = 2.44M$, (2) SSSP ((d), (e), and (f)) on a graph of size $|V| = 8K$ and $|E| = 80K$, and (3) BC ((g), (h), and (i)) on a graph of size $|V| = 16K$ and $|E| = 160K$. X-axis and Y-axis units are the number of threads and time in second, respectively.
Theorem 1. Denote $\mathcal{D} = \{BFS, SSSP, BC\}$, $\mathcal{M}_V = \{PUT, REM\}$, $\mathcal{M}_E = \{PUT_E, REM_E\}$, and $\mathcal{M} = \mathcal{M}_V \cup \mathcal{M}_E$. Let $\delta_e$ be the degree of vertex whose edge modification happens. Denote $X_{\mathcal{M}} = \{o \in X \mid \text{type}(o) \in \mathcal{M}, \mathcal{M} \subseteq X\}$, where $\mathcal{M}$ is the ADT as defined before. Let $I_{\mathcal{M}}$ and $C_{\mathcal{M}}$ denote the interval and point contentions, respectively, of $o$ pertaining to the operation calls $o \in \{X_{\mathcal{M}} \cup \{\}\}$). Accordingly, $I_{\mathcal{M}} = I_{\mathcal{M}} \mathcal{M} - 1$. Considering the queries $q \in \mathcal{Q}$ performed by $PG-Cn$, the worst-case amortized cost per operation $A_X$ for an execution $X$ s.t. type($o$) $\in \mathcal{M} \cup \{q\} \forall o \in X$, and $q \in \mathcal{Q}$ is $A_X = A_X(\mathcal{M}) + \sum_{o \in X_{\mathcal{M}}} W_{o, S_{G,o}} + |X| \sum_{o \in X_{\mathcal{M}}} W_{o, S_{G,o}} + \sum_{o \in X_{\mathcal{M}}} O(\delta_e)$.

The proof of Theorem 1 is available in the full version [8].

References