A Batch-dynamic Suitor Algorithm for Approximating Maximum Weighted Matching

EUGENIO ANGRIMAN, Department of Computer Science, Humboldt-Universität zu Berlin, Germany
MICHAŁ BOROŃ, Visiting scholar at Humboldt-Universität zu Berlin, Germany
HENNING MEYERHENKE, Department of Computer Science, Humboldt-Universität zu Berlin, Germany

Matching is a popular combinatorial optimization problem with numerous applications in both commercial and scientific fields. Computing optimal matchings w.r.t. cardinality or weight can be done in polynomial time; still, this task can become infeasible for very large networks. Thus, several approximation algorithms that trade solution quality for a faster running time have been proposed. For networks that change over time, fully dynamic algorithms that efficiently maintain an approximation of the optimal matching after a graph update have been introduced as well. However, no semi- or fully dynamic algorithm for (approximate) maximum weighted matching has been implemented.

In this article, we focus on the problem of maintaining a 1/2-approximation of a maximum weighted matching (MWM) in fully dynamic graphs. Limitations of existing algorithms for this problem are (i) high constant factors in their time complexity, (ii) the fact that none of them supports batch updates, and (iii) the lack of a practical implementation, meaning that their actual performance on real-world graphs has not been investigated. We propose and implement a new batch-dynamic 1/2-approximation algorithm for MWM based on the Suitor algorithm and its local edge domination strategy [Manne and Halappanavar, IPDPS 2014]. We provide a detailed analysis of our algorithm and prove its approximation guarantee. Despite having a worst-case running time of $O(n + m)$ for a single graph update, our extensive experimental evaluation shows that our algorithm is much faster in practice. For example, compared to a static recomputation with sequential Suitor, single-edge updates are handled up to $10^5\times$ to $10^6\times$ faster, while batches of 10^4 edge updates are handled up to $10^2\times$ to $10^3\times$ faster.

CCS Concepts: • **Theory of computation → Dynamic graph algorithms;**

Additional Key Words and Phrases: Dynamic matching, dynamic approximation, suitor algorithm

ACM Reference format:
Eugenio Angriman, Michał Boroń, and Henning Meyerhenke. 2022. A Batch-dynamic Suitor Algorithm for Approximating Maximum Weighted Matching. *J. Exp. Algorithmics* 27, 1, Article 6 (July 2022), 41 pages.
https://doi.org/10.1145/3529228

This work is partially supported by German Research Foundation (DFG) Grant No. ME 3619/3-2 within Priority Programme 1736 *Algorithms for Big Data* and by DFG Grant No. ME 3619/4-1 (*Accelerating Matrix Computations for Mining Large Dynamic Complex Networks*).

Authors' addresses: E. Angriman and H. Meyerhenke, Department of Computer Science, Humboldt-Universität zu Berlin, Berlin, Germany, Rudower Chaussee 25, 12489; emails: {angrimae, meyerhenke}@hu-berlin.de; M. Boroń, Visiting scholar at Humboldt-Universität zu Berlin, Berlin, Germany, Rudower Chaussee 25, 12489; email: michal.s.boron@gmail.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.

1084-6654/2022/07-ART6 $15.00
https://doi.org/10.1145/3529228

1 INTRODUCTION

Context. Matching is a widely studied combinatorial optimization problem with numerous practical applications [12, Chapter 5]. Given a graph[1] $G = (V, E)$ with n vertices and m edges, a matching $M \subseteq E$ in G is a set of pairwise non-adjacent edges, i.e., all the vertices in G are incident to at most one edge in M. A matching is called *maximal* if no edge can be added to it without violating the matching property. It is called *maximum*, in turn, if there is no other matching with higher cardinality. Computing the **maximum cardinality matching (MCM)** of a graph can be done in $O(m\sqrt{n})$ time in general graphs with the algorithm by Micali and Vazirani [44], and in $O(n^\omega)$ time in planar graphs with the algorithm by Mucha and Sankowski [45], where $\omega < 2.373$ is the matrix multiplication exponent. On weighted graphs, the **maximum weighted matching (MWM)** is a matching with maximum edge weight. The fastest known algorithms for this problem are by Gabow [23] in $O(nm + n^2 \log n)$ time and (for very sparse graphs) by Galil [24] in $O(mn \log n)$ time. With some restrictions on the input, faster (but still superlinear) time complexities are possible. For a broader overview, we refer the reader to Section 2.2 and to References [12, 37, 38].

Motivation. Today, massive graph data sets are ubiquitous and executing an algorithm with superlinear running time on them can be prohibitively expensive. To mitigate long running times, it is common to resort to approximation. Preis's greedy algorithm [52] computes a 1/2-approximation of the matching with highest weight in $O(m)$ time; the same result is also achieved with the **path-growing algorithm (PGA)** by Drake and Hougardy [16]. The main disadvantage of these algorithms is that they are inherently sequential and thus cannot exploit another common acceleration strategy for massive data: parallelism. Birn et al. [11], in turn, provide a parallel implementation of the local max algorithm [30], which computes a maximal matching of an unweighted graph and a 1/2-approximation of the MWM of a weighted graph in $O(\log^2 n)$ expected time. Manne and Halappanavar introduced Suitor [40], a parallel 1/2-approximation algorithm based on local domination that outperforms previous strategies and is amenable to parallelism.

Real-world networks are not only large but often change over time [43]: edges are inserted, deleted, or change their weight. For example, Internet pages are constantly updated or friendships in social networks are continuously created and terminated. Even with a linear-time algorithm, it would be excessively expensive to recompute a (weighted) matching from scratch every time the graph changes. In recent years, several fully dynamic algorithms for both exact and approximate MCM [3, 5, 6, 8–10, 14, 27, 28, 31, 32, 47, 48, 53, 55] and MWM [1, 28, 57] have been proposed. These algorithms perform a static computation of the matching on an initial snapshot of the graph and exploit this information to update the matching more efficiently than a static rerun when the graph changes. Main limitations of existing fully dynamic algorithms are either a weaker quality guarantee (Reference [1]), or an expensive time complexity (see Reference [28]). To the best of our knowledge, none of the existing algorithms for fully dynamic MWM supports batch updates and—most importantly—none of them has been implemented in practice.

Contribution. In this article, we introduce a new algorithm that takes inspiration from Suitor [40] and maintains a 1/2-approximation of the MWM on fully dynamic graphs. The main idea is simple: we use the static Suitor algorithm to compute a matching on an initial snapshot of the graph. Then, after an edge update, our dynamic algorithm identifies the affected vertices (i.e., whose partner needs to be updated) and updates the matching accordingly.

Our implementation of this algorithm is the first for semi- or fully dynamic MWM. It supports multiple-edge insertions and removals in batches. For single-edge updates, our dynamic algorithm

[1] In this article, we also use the term "network" as a synonym for "graph".

has a worst-case time complexity of $O(n + m)$, whereas for batches with b edge updates it is $O(b(n + m))$. Although this does not improve the static time complexity, our dynamic algorithm performs remarkably well in practice: In our experiments, we evaluate its running time on real-world (complex and street) and synthetic networks with up to 2.5 billion edges. Our results show that, compared to a static recomputation (for lack of another meaningful MWM baseline) with (sequential) Suitor, our algorithm can handle single graph updates $10^5\times$ to $10^6\times$ faster and batches of 10^4 of such updates $10^2\times$ to $10^3\times$ faster. Furthermore, the time required by our dynamic algorithm for every considered batch size is always below a millisecond. Thus, our algorithm's implementation provides real-time capabilities even without parallelism.

2 PRELIMINARIES

2.1 Problem Definition and Notation

Let $G = (V, E, w)$ be a simple, undirected and weighted graph, where $n = |V|$ is the number of vertices, $m = |E|$ is the number of edges, $w : E \to \mathbb{R}_{>0}$, and $N(u)$ is the set of neighbors of vertex u. A matching in G is a subset of pairwise non-adjacent edges $M \subseteq E$. Alternatively, one can see a matching as a subgraph of G (restricted to the edges) with degree at most 1. A vertex is *matched* if it is incident to an edge in M; otherwise, it is called *unmatched* or *free*.

In the MWM problem, the objective is to compute a matching $M^\star$ that maximizes the sum of the edge weights.

Maximum-weight Matching
Input: Undirected weighted graph $G = (V, E, w)$.
Output: Matching $M^\star \subseteq E$ s.t. $\sum_{e \in M^\star} w(e)$ is maximal.

In the context of dynamic graphs, if any edge update[2] happens to G, then we denote by $G' = (V, E', w')$ the graph after the edge update. Similarly, we denote by $N'(u)$ the set of neighbors of a vertex u in G'. Given a matching M computed on G and a sequence of graph updates, our objective is to update M in G' faster (in terms of empirical running time) than recomputing a new matching in G' from scratch, while retaining the theoretical bound on the solution quality.

2.2 Related Work

In the following, we summarize relevant works concerning MCM and MWM in both static and dynamic settings.

Static Algorithms. Edmond's blossom algorithm [20] ($O(mn^2)$ time), and the later improved algorithm by Micali and Vazirani [44] ($O(m\sqrt{n})$ time) are two popular strategies based on augmenting paths to compute an MCM. Goldberg and Karzanov [26] propose a blocking skew-symmetric flow algorithm that achieves the same running time as Micali and Vazirani. More recent works use data reduction rules [35] or shrink-tree data structures [18] to achieve better running times in practice on sparse real-world networks. If we restrict the input to planar graphs, then the randomized algorithm by Mucha and Sankowski [45] computes an MCM in $O(n^\omega)$ time via Gaussian elimination, where $\omega < 2.373$ is the matrix multiplication exponent. Concerning the MWM problem, recall that the fastest known algorithms run in $O(nm + n^2 \log n)$ [23] and in $O(mn \log n)$ [24] time. Assuming integral edge weights, the algorithm by Duan et al. [19] takes $O(m\sqrt{n}\log(nW))$ time, while Sankowski [54] takes $\tilde{O}(Wn^\omega)$ time (if we restrict the input to bipartite graphs), where $\tilde{O}$ hides a polylogarithmic factor and W is the highest edge weight.

[2]Hereafter, we will denote with "edge update" an edge insertion, an edge removal, or an edge weight change.

On bipartite graphs, Sankowski [54] takes $\tilde{O}(Wn^{\omega})$ time.

Running a superlinear algorithm is often too expensive on large graphs, therefore several approximation algorithms with (nearly) linear running time have been introduced. A naïve greedy algorithm that iteratively adds to the matching the (heaviest) edge that does not violate the matching condition takes $O(m \log n)$ time and achieves a 1/2-approximation for both the MCM and MWM problems [4]. Preis [52] reduced the running time to $O(m)$ while retaining the same quality guarantee. Further strategies to obtain the bound 1/2 are the one by Manne and Bisseling [39] based on dominant edges, the (previously known) local max algorithm investigated by Birn et al. [11], the path growing algorithms by Drake and Hougardy [16, 17], the global paths algorithm by Maue and Sanders [41], and Suitor by Manne and Halappanavar [40].

Dynamic Algorithms. A trivial strategy maintains a 1/2-approximation of a maximal matching on dynamic graphs in $O(n)$ time per update by resolving all augmenting paths with length one. This result has been improved in several works. The update time was reduced for the first time to $O((n + m)^{\sqrt{2}/2})$ by Ivković and Lloyd [31]. The randomized algorithm by Onak and Rubinfeld [48] maintains an $O(1)$-approximation of an MCM in $O(\log^2 n)$ expected amortized update time; this result was further improved by Baswana et al. [6] who reduced the update time to $O(\log n)$ and the approximation ratio to 1/2. Solomon [55] further reduced the amortized update time of Baswana et al. from logarithmic to constant. Deterministic algorithms for approximate MCM have been presented first by Bhattacharya et al. [10], who maintain a $(3 + \varepsilon)$-approximate MCM in $\tilde{O}(\min(\sqrt{n}, m^{1/3}/\varepsilon))$ amortized update time; the update time was later reduced to constant but at the cost of a weaker $O(1)$-approximation guarantee [9]. In terms of worst-case bounds for MCM, the best known algorithms are the ones from Gupta and Peng [28] (which maintains a $(1 + \varepsilon)$-approximation with $O(\sqrt{m}/\varepsilon)$ update time), Neiman and Solomon [47] (which maintains a 3/2-approximation with $O(\sqrt{m})$ update time), and Bernstein and Stein [8] (which maintains a $(3/2+\varepsilon)$-approximation with $O(m^{1/4}/\varepsilon^{2.5})$ update time). The first $(2+\varepsilon)$-approximation algorithms in $O(poly \log n)$ update time were introduced independently by Charikar and Solomon [14] and by Arar et al. [3], whereas Grandoni et al. [27] gave a $(1 + \varepsilon)$-approximation algorithm limited to edge insertions. For graphs with constant neighborhood independence, Barenboim and Maimon [5] present an algorithm for MCM with deterministic $\tilde{O}(n)$ update time. For lax and eager algorithms, i.e., two subclasses of fully dynamic algorithms for maintaining a MCM, Kashyop and Narayanaswamy [32] prove a conditional lower bound of the update time that is sublinear in the number of edges.

Despite the vast variety of algorithms for dynamic MCM, very little effort has been invested in implementing them and evaluating their practical performance in real-world instances. Only recently, Henzinger et al. [29] evaluated dynamic algorithms for MCM in practice. These are the algorithms by Baswana et al. [7] (2-approximate MCM in $O(\sqrt{n})$ update time), by Neiman and Solomon [47] (3/2-approximate MCM in $O(\sqrt{m})$ update time), and two novel algorithms: one based on random walks and one that uses a depth-bounded blossom algorithm to find augmenting paths. Their experimental evaluation shows that (i) the optimal matching can be maintained more than 10× faster than a naïve static recomputation, (ii) the considered approximation algorithms are multiple orders of magnitude faster than the naïve static algorithm, and (iii) the extended random walk-based algorithms achieve the best practical performance.

Concerning the dynamic MWM problem, Anand et al. [1] propose a fully dynamic algorithm for maintaining an 8-MWM with an expected amortized time of $O(\log n \log C)$ per edge update, where C is the ratio between the maximum edge weight and the minimum edge weight of the graph. They also show that the approximation ratio can be reduced to 4.9108 without sacrificing performance by using geometric rounding. Gupta and Peng [28] maintain a $(1+\varepsilon)$-approximation

in $O(\sqrt{m}\varepsilon^{-2-O(1/\varepsilon)} \log W)$ update time in graphs with edge weights between 1 and W; their strategy runs the static algorithm from time to time, trims the graph into smaller equivalent graphs whenever possible and partitions the edges into geometrically shrinking intervals depending on their weights.

Stubbs and Williams [57] present metatheorems to show that if there exists an α-approximation algorithm for MCM with update time T, then there also exists an $2\alpha(1+\varepsilon)$-approximation algorithm for MWM with $O(\frac{T}{\varepsilon^2} \log^2 W)$ update time, where W is the maximum edge weight. The main idea relies on improving and extending the algorithm by Crouch and Stubbs [15], who addressed the dynamic MWM problem in the semi-streaming model. None of the aforementioned algorithms for MWM has been implemented.

For the semi-streaming model, Ghaffari and Wajc [25] describe a single-pass MWM algorithm with a $(2+\varepsilon)$-approximation ratio requiring $O(n \log n)$ bits, improving previous results [15, 21, 22, 42, 50]. Again, with the exception of Henzinger et al. [29], we are not aware of any implementations of all these dynamic algorithms.

2.3 The Static Suitor Algorithm

Since our dynamic algorithm is built on top of the static one called Suitor [40], we provide in the following a brief overview of this latter algorithm for self-containment purposes.

Recall that w is the weight function of the edges. By convention, if $\{u, v\} \notin E$ or if a vertex reference v is `null`, then $w(u, v)$ is 0. To guarantee a correct execution, the following total ordering of the edges incident to the same vertex u is enforced: if $\{u, x\}$ and $\{u, y\}$ have the same edge weight and $x < y$, then $w(u, x) < w(u, y)$. Suitor keeps two references for each vertex $u \in V$, $p(u)$ and $suitor(u)$, during the course of the algorithm. When the execution is finished, then

$$p(u) = \arg\max_{v \in N(u)} \{w(u, v) : \nexists x \in N(v) \text{ s.t. } p(x) = v \wedge w(x, v) > w(u, v)\}. \tag{1}$$

In other words, $p(u)$ is the neighbor v of u such that $w(u, v)$ is maximum and there is no vertex $x \in N(v)$ where $p(x) = v$ with $\{x, v\}$ dominating $\{u, v\}$. If no such v exists, then $p(u) =$ `null` (unmatched). Vice versa, $suitor(v)$ refers to the vertex (if any) that keeps a reference to v: $suitor(u) = v$ iff $p(v) = u$.

Algorithm 8 in Appendix A.1 shows the pseudocode of the Suitor algorithm: p and $suitor$ are initially set to `null` for every vertex in the graph; then, the recursive function FINDSUITOR (Algorithm 7) is called on each vertex u, which sets p according to Equation (1). The progress of the algorithm can be described as follows:

LEMMA 2.1. *[40, Lemma 3.2] Following each call to FINDSUITOR in Algorithm 8 from the loop over the vertices of G, $p(u)$ is set according to Equation (1) for each vertex u processed so far.*

Clearly, after the execution of the algorithm, the condition in Equation (1) is true for all vertices, and this leads us to the next property of the resulting matching:

LEMMA 2.2. *[40, Lemma 3.1] If $p(u)$ is set according to Equation (1) for each vertex $u \in V$, then $p(\cdot)$ defines the same matching as the greedy algorithm.*

PROOF. See Reference [33, Section 3.2]. The proof is for b-matching, but it contains the MWM problem by setting b to 1. □

Thus, regardless of the order in which the loop processes the vertices, Suitor is a deterministic algorithm that computes the same matching as the well-known greedy algorithm that adds

permissible edges in the order of decreasing weight. Due to our assumption of a total edge ordering, this matching is unique.

Algorithm 10 in Appendix A.2 shows an iterative version of Suitor, which does not use p any more but only *suitor*. An additional array *ws* stores for each vertex the value $w(u, suitor(u))$. We rewrite the condition in Equation (1) in terms of *suitor* as follows:

$$suitor(u) = \arg\max_{v \in N(u)} \{w(u,v) : \nexists y \in N(v) \text{ s.t. } suitor(v) = y \wedge w(y,v) > w(u,v)\}. \tag{2}$$

If no such vertex exists, then $suitor(u) =$ `null`. Similarly to its recursive counterpart, Algorithm 10 initializes *suitor* and *ws* to `null` and 0, respectively, for every vertex in the graph; then, for every $u \in V$, the algorithm calls the iterative function FindSuitor(u) (Algorithm 9). FindSuitor uses a variable *cur* to store the vertex that is seeking a new partner in the current iteration (i.e., vertex u in the recursive FindSuitor). In Lines 7–10 it determines whether there exists a neighbor v of *cur* that satisfies Equation (2) for *cur* and, if so, it stores v and $w(cur, v)$ into the *partner* and *heaviest* variables, respectively. In Line 12, if *heaviest* is 0, no such neighbor exists and the function terminates, because *done* is left to `true` –hence *cur* remains unmatched. Otherwise, *partner* was set to the matching partner of *cur* and *heaviest* to $w(cur, partner)$; in Lines 13–15, y stores the previous matching partner of *partner* (if any) before making *cur* the new matching partner of *partner* by setting $suitor(partner)$ to *cur* and $ws(partner)$ to *heaviest*. Then, in Lines 16–18, if y is not `null`, then *partner* had a previous potential matching partner y; therefore, y needs to seek a new potential matching partner and this is done by setting *cur* to y and *done* to `false`, which is equivalent to a recursive call of Algorithm 7.

3 DYNAMIC SUITOR ALGORITHM

In this section, we first describe how we extend the static Suitor algorithm [40] to also handle single-edge updates. Building upon that, we generalize our approach to multiple-edge updates in batches in Section 3.3.

We index variables of the Suitor algorithm with the superscript $^{(i)}$ (for intermediate) [or $^{(f)}$ for final, respectively] if they refer to the state directly after the edge change [or after the dynamic Suitor algorithm has been run], e.g., $suitor^{(i)}(u)$ [or $suitor^{(f)}(u)$]. The matching $M^{(i)}$ as well as other Suitor variables are derived from M (and the other counterparts) by taking the edge update on G into account. For example, if an edge e is deleted from G that is part of M, then $M^{(i)} = M \setminus \{e\}$. Our goal is to update/improve the intermediate matching $M^{(i)}$ efficiently, i.e., to avoid redundant computations when computing the final matching $M^{(f)}$. We will show that $M^{(f)}$ equals M', the matching computed by the static Suitor algorithm on G'. To this end, we define the notion of an *affected* vertex.

Definition 1. A vertex u is called *affected* iff $M^{(i)}$ violates Equation (2) for $suitor^{(i)}(u)$, i.e., iff $suitor^{(i)}(u) \neq \arg\max_{v \in N'(u)} \{w'(u,v) : \nexists y \in N'(v) \text{ s.t. } suitor^{(i)}(v) = y \wedge w'(y,v) > w'(u,v)\}$.

Our dynamic algorithm computes $M^{(f)}$ after an edge update by finding all the vertices affected by the edge update (FindAffected function in Algorithm 1) and by then updating their matching partner so that Equation (2) is satisfied (UpdateAffected function in Algorithm 2). If Equation (2) is satisfied for every vertex in G', then it follows from Lemma 2.2 that the resulting matching is (the unique) M'.

FindAffected (Algorithm 1) is an extended version of the FindSuitor function (Algorithm 9) used by the iterative Suitor algorithm; it uses a Boolean array *affected* to keep track of the affected vertices and pushes the affected vertices whose *suitor* and *ws* variables need to be updated onto a

ALGORITHM 1: Extended version of the FINDSUITOR function (Algorithm 9, Appendix A) that finds the affected vertices.

```
1:  function FINDAFFECTED(z)
2:  Input: Affected vertex z
3:  Output: Stack of affected vertices
4:      S_A ← empty stack                                  ▷ Stack of affected vertices.
5:      cur ← z
6:      done ← false
7:      repeat
8:          partner ← suitor(cur)
9:          heaviest ← ws(cur)
10:         for each x ∈ N(cur) do
11:             if not affected(x) and w(cur, x) > heaviest and w(cur, x) > ws(x) then
12:                 partner ← x
13:                 heaviest ← w(cur, x)
14:         done ← true
15:         if heaviest > ws(partner) then
16:             y ← suitor(partner)
17:             suitor(partner) ← cur
18:             ws(partner) ← heaviest
19:             S_A.push(partner)
20:             affected(partner) ← true
21:             if y ≠ null then
22:                 suitor(y) ← null
23:                 ws(y) ← 0
24:                 affected(y) ← true
25:                 cur ← y
26:                 done ← false
27:         else
28:             affected(cur) ← false
29:     until done
30:     return S_A
```

stack S_A. As in FINDSUITOR, *cur* is the vertex we are trying to find a matching partner for, *partner* is the preferred matching partner for *cur*, i.e., the vertex that satisfies Equation (2) for *cur* (if any), and $heaviest = w'(cur, partner)$. Then, in Lines 17 and 18, if a new matching partner for *cur* is found, then the $suitor^{(f)}(partner)$ and $ws^{(f)}(partner)$ variables are updated to *cur* and *heaviest*, respectively; additionally, *partner* is pushed onto S_A (Line 19). If *partner* is matched in M with another vertex y, then the edge $\{partner, y\}$ would violate the matching condition and thus needs to be removed from the matching; this is done in Lines 22 and 23 by "invalidating" the vertex y, i.e., setting the values of $suitor^{(f)}(y)$ and $ws^{(f)}(y)$ to `null` and 0, respectively. As in FINDSUITOR, in the next iteration, we seek for a new partner for the invalidated vertex y by updating *cur* to y. This procedure is repeated until either a new partner for *cur* cannot be found (i.e., *cur* is free in M'), or *partner* is free in M. By keeping track of the affected vertices, we guarantee that a previously invalidated vertex is not selected as new matching partner in future iterations of the loop (see Line 11). The stack S_A is later used by UPDATEAFFECTED (Algorithm 2) to match the affected vertices that were not updated by FINDAFFECTED (i.e., the ones that were stored in the *cur*

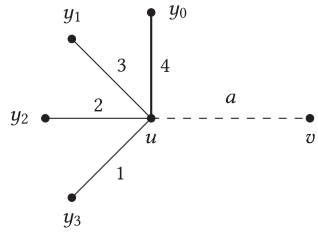


Fig. 1. Example illustration of Lemma 3.1. In this graph, we have that $M = \{\{u, y_0\}\}$, $suitor^{(i)}(u) = y_0$, $suitor^{(i)}(v) = \texttt{null}$, and a new edge $e = \{u, v\}$ with weight a has just been added. Clearly, if $e \in M'$, then Equation (2) holds only if $a > w(u, suitor^{(i)}(u)) = 4$. Vice versa, if $a > 4$, then Equation (2) holds only if $M' = \{\{u, v\}\}$.

ALGORITHM 2: Updates *suitor* and *ws* of the matching partners of the vertices in the stack S_A.

```
1: function updateAffected(S_A)
2: Input: Stack of affected vertices S_A
3:     while S_A is not empty do
4:         x ← S_A.pop()
5:         y ← suitor(x)
6:         suitor(y) ← x
7:         ws(y) ← ws(x)
8:         affected(x) ← false
9:         affected(y) ← false
```

variable) to their new partner (Lines 6 and 7). Once a pair of matched vertices has eventually been processed, they are not affected anymore and thus updateAffected marks them as unaffected (Lines 8 and 9). In the following, we show how these two functions are used in case of an edge insertion or an edge removal and that our dynamic algorithm yields a matching $M^{(f)}$ that equals the matching M' computed by the static Suitor algorithm on G'. Concerning edge weight updates, they can be handled as an edge removal followed by an edge insertion.

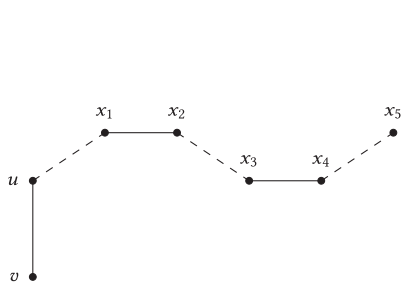
3.1 Edge Insertions

Let us first address the case in which an edge is inserted into G, i.e., $G' = (V, E \cup \{u, v\})$ with $e = \{u, v\} \notin E$.[3] Intuitively, this new edge will only be part of the new matching M' iff it is "a better deal" for both u and v. In other words: $e \in M'$ iff $w'(u, v)$ is heavier than both $w'(u, suitor^{(i)}(u))$ and $w(v, suitor^{(i)}(v))$ and is thus the dominant edge for both. We show this in the following lemma:

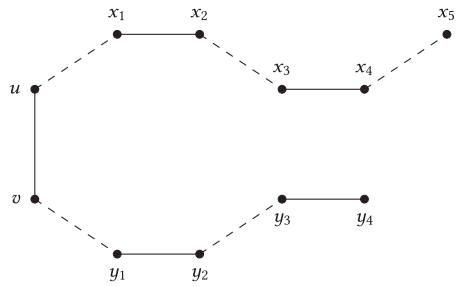
Lemma 3.1. *Let $G' = (V, E \cup e)$ with $e = \{u, v\} \notin E$.*
Then: $e \in M' \Leftrightarrow w'(u, v) > \max\{w(u, suitor^{(i)}(u)), w(v, suitor^{(i)}(v))\}$.

Proof. We develop the proof w.l.o.g. for u by assuming that $w(u, suitor^{(i)}(u)) > w(v, suitor^{(i)}(v))$; the proof for v is symmetric. Let $Y \subseteq N(u)$ be the set of vertices y that do not have any neighbor x such that $w(x, y) > \max\{w(x, suitor^{(i)}(x)), w(u, y)\}$, i.e., the set of vertices among which Equation (2) selects $suitor(u)$ as the vertex y with maximum $w(y, u)$. In the example in Figure 1, $Y = \{y_0, y_1, y_2, y_3\}$. We also define $Y^{(v)} := Y \cup \{v\}$.

[3]The intuition behind the proofs presented in this section share some similarities with the proof presented in Reference [51, Section A.1] for dynamic maximum cardinality *bipartite* matching.



(a) The affected vertices are covered by one alternating path P_v.



(b) The affected vertices are covered by two alternating paths P_u and P_v.

Fig. 2. Examples of alternating paths that cover the vertices affected by the insertion of an edge $\{u, v\}$. Solid lines and dashed lines represent edges in $M' \backslash M$ and edges in $M \backslash M'$, respectively. In Figure 2(a) there is only one alternating path, because u is matched in M and v is not, whereas in Figure 2(b) there are two, because both u and v are matched in M.

ALGORITHM 3: Dynamic Suitor algorithm for single-edge insertions.

Input: Graph $G' = (V, E \cup \{u, v\})$, an edge $\{u, v\} \notin E$.
Output: New matching $M^{(f)}$ on G'.

1: **if** $w'(u, v) > \max\{w'(u, suitor^{(i)}(u)), w'(v, suitor^{(i)}(v))\}$ **then**
2: $\quad affected(x) \leftarrow$ `false` $\forall x \in V$
3: $\quad$ **for** $z \in \langle v, u \rangle$ **do**
4: $\quad\quad affected(z) \leftarrow$ `true`
5: $\quad\quad S_A \leftarrow$ FINDAFFECTED(z)
6: $\quad\quad$ UPDATEAFFECTED$(S_A \backslash \{suitor^{(i)}(z)\})$
7: $M^{(f)} \leftarrow \{\{u, v\} \in E$ s.t. $(suitor(u) = v) \wedge (suitor(v) = u)\}$
8: **return** $M^{(f)}$

"$\Rightarrow$" From $e \in M'$, we have that $suitor'(v) = u$, and from Equation (2) it follows that $v = \arg\max_{y \in Y^{(v)}} w'(u, y)$. Therefore, since $suitor^{(i)}(u) \in Y^{(v)}$, $w'(u, v) > w'(u, suitor^{(i)}(u))$. In our example in Figure 1, it is clear that $\{u, v\} \in M' \Rightarrow a > 4$.

"$\Leftarrow$" By definition, we have that $suitor^{(i)}(u) = \arg\max_{y \in Y} w'(u, y)$ and that $suitor'(u) = \arg\max_{y \in Y^{(v)}} w'(u, y)$. By hypothesis, $w'(u, v) > w(u, suitor^{(i)}(u))$, and thus $suitor'(u) = \arg\max_{y \in Y^{(v)}} w'(u, y) = v$. The same holds if we exchange u and v in the argument. Therefore, $e \in M'$. This is also clear in Figure 1: if $a > 4$, then Equation (2) holds only if $\{u, v\} \in M'$. □

Algorithm 3 shows our dynamic algorithm for edge insertions. Given a newly added edge $e = \{u, v\}$, in Line 1 the algorithm excludes e if it does not satisfy Lemma 3.1, because e is then also not part of M'. As we will show later, all the vertices affected by an edge insertion lie on two alternating paths that start from u and v and that alternate edges in $M' \backslash M$ with edges in $M \backslash M'$, as shown in Figure 2. In the for loop, our algorithm finds the affected vertices that lie on these two alternating paths and updates their matching partner according to Equation (2). This is done as follows: the first vertex in the path is marked as affected, FINDAFFECTED updates the *suitor* and *ws* variables of the affected *partner* vertices and pushes them onto a stack S_A. Finally, UPDATEAFFECTED matches the vertices in S_A to their new partner. Note that, in Line 6, $suitor^{(i)}(z)$ is removed from S_A to avoid overwriting $suitor(z)$ in UPDATEAFFECTED; this information is needed in the next iteration of the for loop to find the affected vertices in the alternating path that starts from u.

We now analyze in more detail which vertices are affected by the insertion of an edge $e = \{u, v\} \notin E$ that satisfies Lemma 3.1 (and is thus in M') and how $M^{(f)}$ is computed starting from $M^{(i)}$. In the following, we split our analysis into three possible scenarios: both u and v are unmatched (Section 3.1.1), only one of u and v is matched (Section 3.1.2), and both u and v are matched (Section 3.1.3).

3.1.1 u and v Both Unmatched. Let us first cover the trivial case where both u and v are unmatched in M, i.e., there is no vertex in G that satisfies Equation (2) for both u and v. In the following lemma, we show that $M' = M^{(i)} \cup \{e\}$ and that u and v are the only affected vertices.

Lemma 3.2. *Let $e = \{u, v\} \notin E$. If both u and v are unmatched in M, then $M' = M \cup \{e\}$ and u and v are the only affected vertices.*

Proof. The proof is symmetric for u and v, we develop it for u. If u is unmatched in M, then $suitor^{(i)}(u) = \texttt{null}$ and there is no neighbor of u that satisfies Equation (2) in G. After the insertion of e, we have that $0 = w'(u, suitor^{(i)}(u)) < w'(u, v)$ and thus v satisfies Equation (2) for u. Further, all the neighbors of u in G are already matched, and matching u with v cannot invalidate Equation (2) for any of them. Therefore, $M' = M^{(i)} \cup \{e\}$ and no other vertex apart from u and v is affected. □

We now show that $M^{(f)}$ equals M', i.e., Equation (2) is fulfilled by $M^{(f)}$ for every vertex in G'. The condition in Line 1 in Algorithm 3 is clearly true, because both u and v are free. In findAffected(v), we have that $cur = v$ and $partner = u$. Thus, $suitor^{(f)}(u)$ is set to v and u is pushed onto S_A. findAffected(v) performs only one iteration, since $y = suitor^{(i)}(u) = \texttt{null}$, and updateAffected has no effect, since its input stack is empty. The next iteration of the for loop performs the same operations but with u and v swapped. Thus, the resulting matching is $M^{(f)} = M^{(i)} \cup \{e\} = M'$.

3.1.2 u Matched and v Unmatched. To analyze the case where just one of u and v is matched in M, our analysis assumes w.l.o.g. that u is matched in M and v is not; the other case is symmetric. We first describe how Algorithm 3 identifies all the affected vertices and then how it updates their matching partner.

In this scenario, u and v are not the only vertices affected by the insertion of e, because Equation (2) is violated also for $suitor^{(i)}(u)$. In particular, we show in the following lemma that the vertices affected by the insertion of e are covered by a simple (i.e., without loops) path that, starting from v, alternates edges in $M'\backslash M$ with edges in $M\backslash M'$ as shown in Figure 2(a). We show this in the following lemma.

Lemma 3.3. *Let $e = \{u, v\} \notin E$ be a newly inserted edge such that $e \in M'$. If u is matched in M and v is not, then all the vertices affected by the insertion of e are connected by a simple alternating path P_v that starts from v and that alternates edges in $M'\backslash M$ with edges in $M\backslash M'$. Further, the weights of the edges along P_v are decreasing, i.e., for each $e_1, e_2 \in P_v$ where e_1 precedes e_2 in P_v, we have that $w'(e_1) > w'(e_2)$.*

Proof. Clearly, $e \in M'\backslash M$ is the first edge in P_v. As shown in Figure 2(a), let $x_1 = suitor^{(i)}(u)$: $e \in M'$ implies that Equation (2) is violated for x_1 in G'. If no vertex $x_2 \in N'(x_1)$ satisfies Equation (2) for x_1, then x_1 remains unmatched in M' and no further vertex is affected. Otherwise, there exists another vertex $x_2 \in N'(x_1)$ that satisfies Equation (2) for x_1. In the former case, the alternating path has only two edges: e and $e_1 = \{u, x_1\} \in M\backslash M'$. In the latter case, $suitor^{(f)}(x_1) = x_2$ and $suitor^{(f)}(x_2) = x_1$. Hence, in addition to e_1, the alternating path has at least another edge $e_2 = \{x_1, x_2\} \in M'\backslash M$. By repeating with e_1 the same logic, we applied to e, it follows that the

vertices affected by the insertion of e lie on a path P_u that, starting from u, follows edges in M' and in M alternately.

What is left to be shown is that P_v is simple, which can be done similarly as in the final part of the proof of Reference [40, Lemma 3.2]. Note that, if x_1 is affected, then $w'(u, x_1) < w'(u, suitor^{(f)}(u) = v)$ and $w'(x_1, suitor^{(f)}(x_1) = x_2) < w'(u, x_1)$, and thus $w'(x_1, suitor^{(f)}(x_1)) < w'(u, suitor^{(f)}(u))$. In words, the weights of the edges along P_v are decreasing, because every time an affected vertex x_1 loses its matching partner $suitor^{(i)}(x_1)$, it holds: if it finds a new partner $suitor^{(f)}(x_1) = x_2$, then $w'(x_1, x_2)$ must be smaller than $w'(x_1, suitor^{(i)}(x_1))$. Therefore, it is not possible for x_1 to be matched in M' with a vertex that is already covered by P_v, which implies that P_v is simple. □

Note that, as shown in Figure 2(a), the alternating path $P_v = (u, v, x_1, \ldots, x_\ell)$ alternates vertices x_i that are matched in M' with a "worse partner" (in terms of edge weight) than the one they had in M (i.e., the ones where i is odd), and vertices x_i that are matched in M' with a "better partner" than the one they had in M (i.e., u, v, and the ones where i is even). Hereafter, we will call the former ones "downgraded" and the latter ones "upgraded." More formally:

Lemma 3.4. *For each vertex x_i in an alternating path $P_v = (v, u, x_1, \ldots, x_\ell)$, $1 \leq i \leq \ell$ holds: if i is odd, then $w'(x_i, suitor^{(f)}(x_i)) < w'(x_i, suitor^{(i)}(x_i))$ (i.e., x_i is downgraded); otherwise, i is even and $w'(x_i, suitor^{(f)}(x_i)) > w'(x_i, suitor^{(i)}(x_i))$ (i.e., x_i is upgraded).*

Proof. Every downgraded vertex x_i loses its initial matching partner $suitor^{(i)}(x_i)$, because $suitor^{(i)}(x_i)$ is matched in M' with another (upgraded) vertex—e.g., x_1 loses its initial matching partner u, because $\{u, v\} \in M'$. Note that, in P_v, the upgraded vertex matched with $suitor^{(i)}(x_i)$ in M' comes always earlier than x_i; from Lemma 3.3, we know that the weights of the edges along P_v are decreasing, and thus $w'(x_i, suitor^{(i)}(x_i)) < w'(x_i, suitor^{(f)}(x_i))$. Further, by construction of the alternating path, P_v alternates downgraded and upgraded vertices from x_1 on. Thus, knowing that x_1 is downgraded, all the remaining x_i with odd i are also downgraded.

If x_i is upgraded, then it is either one of u and v, or it is $suitor^{(f)}(x_{i-1})$ of the previous downgraded vertex x_{i-1} in P_v. Since by our hypothesis x_i is in P_v, we also have that $suitor^{(f)}(x_i) = x_{i-1}$, hence $w'(x_i, suitor^{(f)}(x_i) = x_{i-1}) > w'(x_i, suitor^{(i)}(x_i))$. □

Remark 1. Due to the total ordering of the edge weights, the alternating path P_v is unique and can be computed deterministically as the sequence of the vertices stored in the *cur* and *partner* variables of the FindSuitor(v) function (Algorithm 9).

From Lemma 3.3 it follows that, to compute M', we need to find the affected vertices in the alternating path P_v and update their $suitor^{(f)}$ and $ws^{(f)}$ values according to Equation (2). In the following, we show that Algorithm 3 finds all the vertices in P_v and updates their matching partner according to Equation (2), so that $M^{(f)}$ equals M'.

Proposition 3.5. *FindAffected(v) (Algorithm 1) computes $suitor^{(f)}$ and $ws^{(f)}$ according to Equation (2) for the upgraded vertices in P_v and pushes them onto a stack S_A.*

Proof. The function maintains the following loop invariant: *cur* is either v or an invalidated downgraded vertex $x_i \in P_v$; in the first case, *partner* is u; in the second case, if the condition in Line 15 is true, then *partner* is an improving vertex $x_{i+1} \in P_v$—otherwise both the loop and P_v stop. Maintaining the invariant guarantees that FindAffected(v) covers all affected vertices in P_v and that all the improving vertices x_i are updated according to Equation (2) and pushed onto S_A.

The invariant obviously holds in the first two iterations. In the first one, we have that $cur = v$, $partner = u$, and $y = suitor^{(i)}(u) = x_1$ (see Figure 2(a)). In the next one, $cur = x_1$ is invalidated

from the previous iteration, and thus it is a downgraded vertex in P_v that is seeking a new partner to replace the previous partner u; if a new partner x_2 is found, $partner = x_2$ is an upgraded vertex in P_v, thus $suitor^{(f)}(x_2)$ and $ws^{(f)}(x_2)$ are updated to x_1 and $w'(x_1, x_2)$, respectively, and it is pushed onto S_A (Lines 17–19). By applying the same logic to the remaining vertices in P_v, the invariant also holds for the remaining iterations of FINDAFFECTED(v). We remark that, by marking the vertices in P_v as affected, the function cannot iterate on the same vertices multiple times. The function terminates when either *partner* is a free vertex (hence $y =$ `null`), or there is no neighbor of *cur* that satisfies Equation (2), and thus *cur* remains unmatched in $M^{(f)}$. □

Note that, if P_v ends with a downgraded (hence, free) vertex x_ℓ, then FINDAFFECTED(v) invalidates it before terminating, i.e., $suitor^{(f)}(x_\ell)$ and $ws^{(f)}(x_\ell)$ are updated according to Equation (2). Therefore, x_ℓ is not affected anymore in G' and thus marked as unaffected (Line 28).

PROPOSITION 3.6. *After* UPDATEAFFECTED *finishes, all vertices in P_v satisfy Equation (2) in G'.*

PROOF. From Proposition 3.5, we know that, when UPDATEAFFECTED is called, all the upgraded vertices satisfy Equation (2) in G' and that they are stored in the stack S_A. UPDATEAFFECTED "completes" the matching $M^{(f)}$ by updating, for all the downgraded vertices $x_i \in P_v$ matched in M', their $suitor^{(f)}(x_i)$ and $ws^{(f)}(x_i)$ values to x_{i+1} and $w'(x_i, x_{i+1})$, respectively, and thus all vertices in P_v satisfy Equation (2) in G'. □

PROPOSITION 3.7. *If u is matched in M and v is not (or vice versa), then Algorithm 3 has a worst-case running time that is linear in the number of the affected vertices and in the sum of their degrees.*

PROOF. Algorithm 3 invokes FINDAFFECTED and UPDATEAFFECTED twice. The number of iterations of FINDAFFECTED is linear in the length of a simple alternating path that covers the vertices affected by the edge insertion (Lemma 3.3). In each iteration, FINDAFFECTED iterates over all the neighbors of the current vertex (Line 10) and all the other operations have constant time complexity.

UPDATEAFFECTED performs constant-time operations on each vertex in S_A. FINDAFFECTED pushes at most one vertex onto S_A in each iteration, and thus the worst-case time complexity of UPDATEAFFECTED is linear in the number of affected vertices. □

Remark 2. In the worst case, all vertices in the graph are affected by an edge insertion and then FINDAFFECTED performs at most n iterations and visits all edges twice. Therefore, the worst-case time complexity of Algorithm 3 is $O(n + m)$.

3.1.3 u and v Both Matched. To settle the final case, we show (i) that the affected vertices are covered by two alternating paths P_v and P_u and then (ii) that the matching $M^{(f)}$ computed by Algorithm 3 equals M'.

LEMMA 3.8. *Let $\{u, v\} \notin E$ be a newly inserted edge such that $e \in M'$. If both u and v are matched in M, then all the vertices affected by the insertion of e are connected by two simple alternating paths P_v and P_u with decreasing edge weights that start from v and u, respectively, and that alternate edges in $M' \setminus M$ and edges in $M \setminus M'$.*

PROOF. We proceed similarly as in Lemma 3.3. As shown in Figure 2(b), $e \in M' \setminus M$ is the first edge for both P_v and P_u. Thus, Equation (2) is violated for both $x_1 = suitor^{(i)}(u)$ and $y_1 = suitor^{(i)}(v)$ (due to the matching condition, $x_1 \neq y_1$). Consequently, the edges $e_{1,u} = \{u, x_1\}$ and $e_{1,v} = \{v, y_1\}$ are removed from $M^{(i)}$, which implies that $e_{1,u}, e_{1,v} \in M \setminus M'$. Now let x_2 and y_2 be the two vertices (if any) that satisfy Equation (2) for x_1 and y_1, respectively, in G'. Clearly,

we have that $\{x_1, x_2\}, \{y_1, y_2\} \in M' \backslash M$. If x_2 and/or y_2 are matched in M, then we can apply recursively to their matching partners the same logic as we did with x_1 and y_1. Hence, P_v and P_u are constructed exactly as described in Lemma 3.3 and thus they are both simple and have decreasing edge weights. □

In the following, we show that the vertices along P_v and P_u are computed in the two iterations of Algorithm 3. A crucial observation is that our dynamic algorithm computes one alternating path at a time—P_u is computed after P_v. Therefore, when a new *partner* vertex for *cur* is found while computing P_v (i.e., in FINDAFFECTED(v)), this might not be the actual vertex that satisfies Equation (2) for *cur* in G', because it is computed without considering the affected vertices in P_u—they are yet to be computed. If this is the case, then it results in a wrong computation of P_v and thus, immediately after the first iteration of Algorithm 3, some of the vertices covered by FINDAFFECTED(v) are in a "wrong state," i.e., Equation (2) is locally satisfied for them, but not for all the vertices in G'. However, we show later that, in this case, the second iteration of Algorithm 3 not only updates the affected vertices in P_u but also corrects the vertices that are in a wrong state, so that eventually all vertices in G' fulfill Equation (2) and thus $M^{(f)} = M'$. For this purpose, we expand our notation by denoting the values of the variables of Suitor and P_v immediately after the first iteration of Algorithm 3 with superscript $^{(ii)}$.

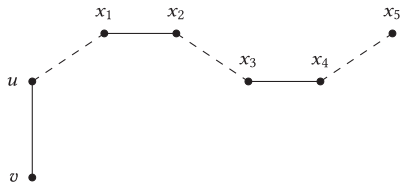
Clearly, if in $P_v^{(ii)}$ there are no vertices in a wrong state, then $P_v^{(ii)} = P_v$ and P_u is computed exactly in the same way as P_v (as described in Section 3.1.2) and no further analysis is required. Otherwise, the two paths intersect: let y_i be the last vertex in P_u before P_u intersects $P_v^{(ii)}$ and let x_j be the vertex in $P_v^{(ii)}$ adjacent to y_i (e.g., vertices x_5 and y_5 in Figure 3(b)). We observe that an intersection always happens when FINDAFFECTED(u) selects a vertex in $P_v^{(ii)}$ as new matching partner for the current vertex y_i in P_u. Therefore, y_i is always a downgraded vertex and x_j satisfies Equation (2) for y_i in G'. Depending on the position of x_j in $P_v^{(ii)}$, we identify three possible cases and treat them separately: (i) x_j is downgraded and it is the last vertex in $P_v^{(ii)}$ (as in Figure 3(b)), (ii) x_j is downgraded and internal in $P_v^{(ii)}$ (as in Figure 3(c)), and (iii) x_j is upgraded (as in Figure 3(d)).

Case (i)—x_j is downgraded and it is the last vertex in $P_v^{(ii)}$: In this case (see Figure 3(b)), x_j is free in $M^{(ii)}$, because $suitor^{(ii)}(y_i) = y_{i-1}$ and no other vertex in $N'(x_j)$ satisfies the condition in Line 11 of FINDAFFECTED(v) for x_j. Hence, x_j is in a wrong state, because it is not matched with y_i, its actual matching partner in M'. The wrong state of x_j is corrected by FINDAFFECTED(u): when P_u reaches y_i, $suitor^{(f)}(x_j)$ and $ws^{(f)}(x_j)$ are set to y_i and $w'(x_j, y_i)$, respectively. Also, x_j is pushed onto S_A, so that y_i will eventually be matched with x_j in UPDATEAFFECTED.

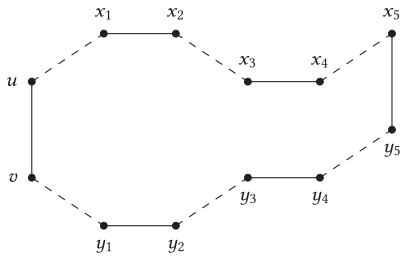
Case (ii)—x_j is downgraded and it is internal in P_u: Here x_j is in a wrong state for the same reason as in case (i). The main difference (see Figure 3(c)) is that $P_v^{(ii)}$ does not finish in x_j, because there exists a vertex $partner = x_{j+1} \in N'(x_j)$ that satisfies the conditions in Lines 11 and 15 in FINDAFFECTED(v). Thus, all the vertices in $P_v^{(ii)}$ after x_j are in a wrong state as well, because they are not affected, but their matching partner is updated by Algorithm 3. However, in the following lemma, we show that the second iteration of the algorithm matches x_j to its matching partner in M' and restores the original matching partners of the vertices in $P_v^{(ii)}$ after x_j.

LEMMA 3.9. *The second iteration of Algorithm 3 matches x_j with its matching partner in M' and restores the original matching partners of the vertices in $P_v^{(ii)}$ after x_j.*

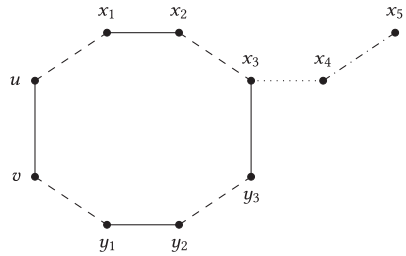
PROOF. As in case (i), x_j fulfills Equation (2) for y_i and, once FINDAFFECTED(u) reaches y_i, it matches x_j to y_i. Recall that the vertices in $P_v^{(ii)}$ after x_j are in a wrong state in $M^{(ii)}$, because



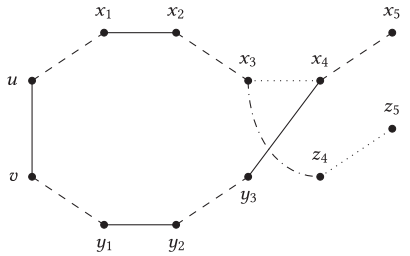
(a) Affected vertices immediately after the computation of $P_v^{(ii)}$ and before the computation of P_u. This represents $M^{(ii)}$, not $M^{(f)}$ since P_u is yet to be computed.



(b) Case (i): matching $M^{(f)}$ after P_u is computed. The last vertex x_5 in $P_v^{(ii)}$ is free in $M^{(ii)}$, P_u intersects $P_v^{(ii)}$ in x_5, and x_5 is matched with y_5 in $M^{(f)}$.



(c) Case (ii): matching $M^{(f)}$ after P_u is computed. P_u intersects $P_v^{(ii)}$ in x_3, which is internal in $P_v^{(ii)}$. $P_v^{(ii)}$ from x_4 on is undone.



(d) Case (iii): matching $M^{(f)}$ after P_u is computed. P_u intersects $P_v^{(ii)}$ in x_4 which is upgrade and continues with x_3.

Fig. 3. Examples of intersecting alternating paths computed by Algorithm 3 to update the matching after the insertion of an edge $\{u, v\}$. Figure 3(a) shows the status of the affected vertices after the computation of $P_v^{(ii)}$ and before the computation of P_u. Figures 3(b), (c), and (d) show the three possible cases of intersection between $P_v^{(ii)}$ and P_u. Dashed and solid edges have the same meaning as in Figure 2, dotted edges are in $M^{(ii)} \setminus M^{(f)}$, dash-dotted edges are in $M^{(f)} \setminus M^{(ii)}$.

their matching has been updated, although they are not affected by the edge insertion. Hence, we need to show that for these vertices, FINDAFFECTED(u) restores the matching partner they have in M—i.e., the same as in M', since they are not affected.

Let x_{j+1} be $suitor^{(i)}(x_j)$; in the iteration where *partner* is x_j, FINDAFFECTED(u) keeps iterating with $cur = x_{j+1}$. Hereafter, FINDAFFECTED(u) maintains the following invariant: *cur* is an upgraded vertex in $M^{(i)}$ and, if $heaviest > ws^{(ii)}(partner)$, then *partner* is the matching partner of *cur* in M; otherwise *cur* is free in M'. Due to the properties of the alternating path, we have that, in $M^{(i)}$, x_{j+1} is an upgraded vertex, because it is matched with x_j and $w'(x_j, x_{j+1}) > w'(x_{j+1}, x_{j+2})$. In M', in turn, we have that x_{j+1} is either free or matched with another vertex x_{j+2}. In the first case, FINDAFFECTED(u) stops at x_{j+1} and leaves it unmatched, because there is no other vertex in $N'(x_{i+1})$ that satisfies Equation (2). Otherwise, we need to show that FINDAFFECTED(u) selects x_{j+2} as matching partner for x_{j+1}. From Lemma 3.8, the vertices on $P_v^{(ii)}$ and P_u before x_{j+1} satisfy Equation (2) and thus cannot be selected as partners for x_{j+1}. Furthermore, x_{j+2} satisfies Equation (2) for x_{j+1} in G', and it is downgraded in $M^{(ii)}$ (hence $w'(x_{j+2}, suitor^{(ii)}(x_{j+2})) < w'(x_{i+1}, x_{j+2})$), which means that the conditions in Lines 11 and 15 hold. Also, the edge weights are decreasing along $P_v^{(ii)}$ (Lemma 3.8), implying that no other neighbor of x_{i+1} in $P_v^{(ii)}$ satisfies Equation (2) for x_{j+1} in G'; hence, $\{x_{i+1}, x_{i+2}\} \in M^{(f)}$. The same applies to the remaining vertices in $P_v^{(ii)}$. □

Case (iii)—x_j is upgraded. In this case (see Figure 3(d)), we have that $\{x_{j-1}, x_j\} \in M^{(ii)}$ and that $w'(y_i, x_j) > w'(x_{j-1}, x_j) > w'(x_j, x_{j+1})$. Thus, x_{j-1} is in a wrong state, because $\{y_i, x_j\} \in M^{(f)}$,

whereas the remaining part of $P_v^{(ii)}$ from x_j on is not necessarily wrong, because x_j is upgraded also in P_u. FINDAFFECTED(u) corrects x_{j-1} by continuing the alternating path from it: once $cur = y_i$ is matched with x_j, in the next iteration of FINDAFFECTED(u) we have that cur is x_{j-1}. Let z_j be the vertex that satisfies Equation (2) for x_{j-1} in G' (e.g., z_4 in Figure 3(d)). If no such vertex exists, then x_{j-1} is free in $M^{(f)}$ and FINDAFFECTED(u) stops; otherwise P_u continues with z_j.

We covered now all possible cases that can occur after an edge insertion. In the following lemma, we generalize our results.

PROPOSITION 3.10. *After an edge insertion, Algorithm 3 computes M'; its worst-case time complexity is $O(n+m)$.*

PROOF. The correctness of the resulting matching is shown for every possible case of edge insertion in Sections 3.1.1 to 3.1.3. Concerning the worst-case time complexity, from Proposition 3.7 and Remark 2 it follows that the first iteration of Algorithm 3 has $O(n+m)$ worst-case running time. The same also holds for the second iteration: due to Lemma 3.8, P_u is simple, so that its length is bounded by $O(n)$, and therefore the worst-case time complexity of Algorithm 3 is $O(n+m)$. □

Unfortunately, a bound using the number of affected vertices and their degrees as in Proposition 3.7 is not possible here due to the vertices in a "wrong" state.

3.2 Edge Removals

We now address the case in which an edge $\{u,v\} \in E$ is removed from G, i.e., $G' = (V, E\backslash\{u,v\})$.

LEMMA 3.11. *Let $G' = (V, E\backslash e)$ with $e = \{u,v\} \in E$. Then: u and v are affected $\Leftrightarrow$ $e \in M$.*

PROOF. We develop the proof w.l.o.g. for u, for v it is symmetric.

"$\Leftarrow$" If $e \in M$, then u is trivially affected, since $suitor^{(i)}(u) = v \notin N'(u)$; u thus violates Equation (2) in G'.

"$\Rightarrow$" Let $Y \subseteq N(u)$ be defined as in Lemma 3.1, i.e., as the set of neighbors y of u among which Equation (2) selects $suitor(u)$ as the vertex y with maximum $w(u,y)$. Further, let $Y^{(v)} := Y\backslash\{v\}$. Let us assume for sake of contradiction that $e \notin M$. Hence, $suitor^{(i)}(u) = \arg\max_{y\in Y} w(u,y) \neq v$ and, in G', this suitor does not change by having removed v from the neighborhood of u. Hence, we have that $suitor'(u) = \arg\max_{y\in Y^{(v)}} w'(u,y) = suitor^{(i)}(u)$, meaning that u is not affected, which contradicts our hypothesis. □

Algorithm 4 shows our dynamic algorithm for edge removals. According to Lemma 3.11, Line 1 excludes all the removals where the removed edge $e = \{u,v\}$ is not in M. Similar to edge insertions, we show later that the vertices affected by an edge removal lie on two alternating paths that start from u and v and that alternate edges in $M'\backslash M$ with edges in $M\backslash M'$. If $e \in M$, then both u and v are affected downgraded vertices for which we have to find a new partner. First, Lines 3 and 4 invalidate them. Then, as for edge insertions in Algorithm 3, the for loop in Line 5 uses FINDAFFECTED to compute the alternating paths of the affected vertices and to update the matching partner of the upgraded vertices in the path. Then, it uses UPDATEAFFECTED to update the matching for the downgraded vertices in the path.

LEMMA 3.12. *Let $e = \{u,v\} \in M$. If e is removed from G, then all the vertices affected by the removal of e are connected by two simple alternating paths P_u and P_v with decreasing edge weights that start from u and v, respectively, and that alternate edges in $M'\backslash M$ and edges in $M\backslash M'$.*

PROOF. As we did in Lemma 3.8, we need to show that the construction of P_u and P_v is equivalent to the construction of an alternating path that connects the vertices affected by an edge insertion as described in Lemma 3.3. The only difference is that, after an edge removal, the alternating paths

ALGORITHM 4: Dynamic Suitor algorithm for single-edge removals.

Input: Graph $G = (V, E)$, an edge $(u, v) \in E$.
Output: New matching $M^{(f)}$ on G'.

1: **if** $suitor(u) = v$ **then** ▷ Check if u and v are matched together.
2: $affected(x) \leftarrow$ `false` $\forall x \in V$
3: $suitor(u) \leftarrow$ `null`; $suitor(v) \leftarrow$ `null`
4: $ws(u) \leftarrow 0$; $ws(v) \leftarrow 0$
5: **for** $z \in \langle u, v \rangle$ **do**
6: $affected(z) \leftarrow$ `true`
7: $S_A \leftarrow$ FINDAFFECTED(z)
8: UPDATEAFFECTED(S_A)
9: $M^{(f)} \leftarrow \{\{u, v\} \in E$ s.t. $(suitor(u) = v) \wedge (suitor(v) = u)\}$
10: **return** $M^{(f)}$

start from a downgraded vertex rather than from an upgraded vertex. We develop our proof w.l.o.g. for u, for v it is symmetric.

Due to Lemma 3.12, u is affected and downgraded. If there exists a vertex x_1 that satisfies Equation (2) for u in G', then x_1 is upgraded and $e_1 = \{u, x_1\} \in M' \backslash M$; otherwise, u remains free in G' and it is the only vertex in P_u. Similarly, if x_1 is matched in M with a vertex x_2, then x_2 is affected and downgraded, $e_2 = \{x_1, x_2\} \in M \backslash M'$, and we can apply to x_2 the same logic we applied to u; otherwise, x_1 is free in G and P_u has only one edge e_1. Thus, as in Lemma 3.3, the resulting path P_u alternates edges in $M' \backslash M$ and edges in $M \backslash M'$ as well as downgraded and upgraded vertices. □

Thus, the vertices affected by an edge removal are covered by alternating paths as the ones shown in Figure 2, with the difference that the solid lines represent edges in $M \backslash M'$ and the dashed lines represent edges in $M' \backslash M$.

PROPOSITION 3.13. *After Algorithm 4 finishes, the resulting matching $M^{(f)}$ equals M' in G'.*

PROOF. Algorithm 4 works analogously to Algorithm 3, namely, it uses FINDAFFECTED and UPDATEAFFECTED to find the affected vertices along an alternating path and to compute their $suitor^{(f)}(\cdot)$ and $ws^{(f)}(\cdot)$. From Lemma 3.12, we know that the vertices affected by an edge removal are covered by two alternating paths as described in Lemma 3.3. Thus, the correctness of Algorithm 4 follows from the correctness of Algorithm 3. □

PROPOSITION 3.14. *The worst-case running time of Algorithm 4 is $O(n + m)$.*

As argued in Proposition 3.13, Algorithm 4 works analogously to Algorithm 3. Thus, they have the same worst-case time complexity $O(n + m)$ (also see Proposition 3.10).

3.3 Multiple-edge Updates

Our dynamic algorithms for single-edge updates can be generalized to batches of edge updates. The main idea is to run the algorithms for single-edge updates multiple times on the updated graph G'. When doing this, we might modify the *suitor* and *ws* variables multiple times. Thus, in this section, we use superscript $^{[i]}$ to denote the values of these variables after we ran the algorithm for single-edge updates i times. Hence, if the total number of updates in the batch is b, then $^{[b]}$ is equivalent to $^{(f)}$.

Note that the crucial difference to an algorithm that updates the matching after every single-edge insertion is that our algorithm runs directly on the graph G' that already includes a batch B

ALGORITHM 5: Dynamic Suitor algorithm for a batch of edge insertions.

Input: Graph $G' = (V, E \cup B)$, batch of edge insertions $B = \{\{u, v\}$ s.t. $\{u, v\} \notin E\}$.
Output: New matching $M^{(f)}$ on G'.

1: $affected(u) \leftarrow$ `false` $\forall u \in V$
2: $i \leftarrow 0$
3: $suitor^{[i]}(u) \leftarrow suitor^{(i)}(u)\ \forall\ u \in V$
4: $ws^{[i]}(u) \leftarrow ws^{(i)}(u)\ \forall\ u \in V$
5: **for** $\{u, v\} \in B$ **do**
6: **if** $w'(u, v) > \max\{w'(u, suitor^{[i]}(u)), w'(v, suitor^{[i]}(v))\}$ **then**
7: **for** $z \in \langle u, v \rangle$ **do**
8: $affected(z) \leftarrow$ `true`
9: $S_A \leftarrow$ FINDAFFECTED$_B(z)$ ▷ Edge weights in P_z must be decreasing as described in Section 3.3.1.
10: UPDATEAFFECTED$(S_A \backslash \{suitor^{[i]}(z)\})$
11: $i \leftarrow i + 1$
12: $M^{(f)} \leftarrow \{\{u, v\} \in E$ s.t. $(suitor(u) = v) \wedge (suitor(v) = u)\}$
13: **return** $M^{(f)}$

of edge updates. Thus, the intermediate matching computed by our algorithm after $i < |B| = b$ iterations is not necessarily the matching that Suitor computes on the initial graph with the first i edge updates in the batch. As we explain in Sections 3.3.1 and 3.3.2, this allows our algorithm to update vertices affected by different edge updates in the same iteration; albeit this does now lower the time complexity, it results in better practical performances (see Section 4.3.3).

3.3.1 Multiple-edge Insertions. Algorithm 5 shows our dynamic algorithm to handle a batch $B = \{\{u, v\}$ s.t. $\{u, v\} \notin E\}$ of edge insertions, which essentially applies Algorithm 3 to every individual edge in B. For every edge $e = \{u, v\} \in B$, Algorithm 5 checks if $e \in M^{[i+1]}$ (Line 6) and, if so, it computes the values of $suitor^{[i]}$ and $ws^{[i]}$ of the vertices in G' affected by the insertion of e as done by Algorithm 3. As in Section 3.1, these vertices lie along two alternating paths P_u and P_v that alternate edges in $M^{[i+1]} \backslash M^{[i]}$ and edges in $M^{[i]} \backslash M^{[i+1]}$. In addition to their first edge, we allow P_u and P_v to include further edges in B whose weight is lower compared to any other preceding edge in the alternating path. This condition is necessary to ensure the correctness of the resulting matching: if an alternating path does not have decreasing edge weights, this could result in violations of Equation (2) for some vertices after Algorithm 5 finishes. Furthermore, allowing the alternating paths to include more than one edge in B makes our batch-dynamic algorithm more competitive in practice than an algorithm that only handles single-edge insertions (see Section 4.3.3).

Remark 3. Let P be an alternating path computed by FINDAFFECTED in the ith iteration of Algorithm 5. If the edge weights of P are not decreasing, then Equation (2) could be violated for some vertices in $M^{(f)}$.

Figure 4 shows a simple example when such a violation occurs. Let us assume that $B = \{\{u, v\}, \{y_3, y_4\}\}$, that Algorithm 5 is computing P_u with FINDAFFECTED(u), and that $\{y_3, y_4\}$ is yet to be processed by Algorithm 5 in the loop in Line 5. Once P_u reaches y_5 (by adding $\{y_4, y_5\} \in B$), y_5 cannot choose y_1 as matching partner, because y_1 is already in P_u, and thus FINDAFFECTED marked it as affected and does not consider it as a potential matching partner for y_5 (see Line 11 in Algorithm 1). Thus, Equation (2) is violated for y_5 and y_1; further, UPDATEAFFECTED matches

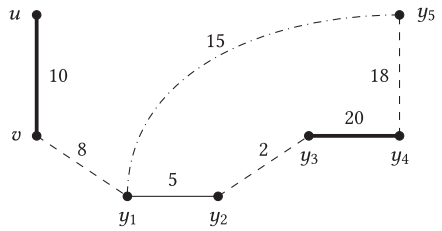


Fig. 4. Example of alternating path (including edge weights) with non-decreasing edge weights where Equation (2) is violated for y_5. Thick solid edges are in B, solid edges are in $M^{[i+1]} \setminus M^{[i]}$, and dashed edges are in $M^{[i]} \setminus M^{[i+1]}$. The dash-dotted edge shows the violation: assuming that y_1 satisfies Equation (2) for y_5, FINDAFFECTED ignores it, because when *cur* is y_5, y_1 is marked as affected and thus not considered as a potential partner for y_5 (see Line 15).

together y_3 and y_4, thus the next iteration of Algorithm 5 does not have any effect and the violation of Equation (2) remains in $M^{(f)}$.

Consequently, in case of a batch of edge insertions, we enforce FINDAFFECTED to discard heavier edges than the ones that are already part of the alternating path and, to distinguish it from the function in Algorithm 1, we denote it as FINDAFFECTED$_B$ (Algorithm 11, Appendix A.3). As we prove in Lemma 3.15, this guarantees that, for every vertex x in an alternating path computed in the ith iteration of Algorithm 5, if $suitor^{[i+1]}(x)$ does not satisfy Equation (2) in $M^{[i+1]}$, then the vertex y that satisfies Equation (2) for x in $M^{[i+1]}$ is such that $\{x, y\} \in B$ and $\{x, y\}$ is yet to be processed by Algorithm 5. Hence, once Algorithm 5 finishes, all vertices in G' satisfy Equation (2), and thus the resulting matching $M^{(f)}$ (i.e., $M^{[b]}$) equals M'.

LEMMA 3.15. *Let P be an alternating path computed in the i-th iteration of Algorithm 5. For every vertex $x \in P$ such that $suitor^{[i+1]}(x)$ does not satisfy Equation (2) in $M^{[i+1]}$ (if any), let y be the vertex that satisfies Equation (2) in $M^{[i+1]}$ for x. Then $\{x, y\}$ is in B and it is yet to be processed by Algorithm 5 in the for loop in Line 5.*

PROOF. We first show that $\{x, y\} \in B$: in case of single-edge insertions, as shown in Lemma 3.8, the resulting alternating paths have always decreasing edge weights. Therefore, an alternating path computed by FINDAFFECTED (Algorithm 1) can have non-decreasing edge weights only if multiple edges are added to G at once—as shown in the example in Figure 4; such edges are excluded by FINDAFFECTED$_B$ in Algorithm 5 (Line 9). Thus, $suitor^{[i+1]}(x)$ does not satisfy Equation (2) in $M^{[i+1]}$ only if $\{x, y\}$ is in B and it is discarded by FINDAFFECTED$_B$.

Further, if y satisfies Equation (2) for x in $M^{[i+1]}$, then we have that $w'(x, y) > \max\{w'(x, suitor^{[i]}(x)), w'(y, suitor^{[i]}(y))\}$. Thus, if $\{x, y\}$ was processed by Algorithm 5 in an earlier iteration than i, then FINDAFFECTED$_B$ would have matched together x and y. Since $\{x, y\} \notin M^{[i]}$, $\{x, y\}$ is yet to be processed by Algorithm 5. □

3.3.2 Multiple-edge Removals. As shown in Algorithm 6, a batch of edge removals $B \subseteq E$ is handled similarly to batch insertions, namely, we apply Algorithm 4 to every edge in B. For every vertex z adjacent to the current edge $e = \{u, v\} \in B$, we check in Line 7 if $suitor^{[i]}(z)$ violates Equation (2) in $M^{[i]}$ due to the removal of e. If so, then we update it in Lines 8–12 as done in Algorithm 4. Otherwise, z has already been updated in a previous iteration of the algorithm and no further action needs to be done.

PROPOSITION 3.16. *Let P be an alternating path computed in the ith iteration of the outermost for-loop in Algorithm 6. Every vertex $x \in P$ satisfies Equation (2) in G'.*

ALGORITHM 6: Dynamic Suitor algorithm for a batch of edge removals.

Input: Graph $G' = (V, E \cup B)$, batch of edge removals $B \subseteq E$.
Output: New matching $M^{(f)}$ on G'.

1: $affected(u) \leftarrow$ `false` $\forall u \in V$
2: $i \leftarrow 0$
3: $suitor^{[i]}(u) \leftarrow suitor^{(i)}(u)\ \forall\ u \in V$
4: $ws^{[i]}(u) \leftarrow ws^{(i)}(u)\ \forall\ u \in V$
5: **for** $\{u, v\} \in B$ **do**
6: **for** $z \in \langle u, v \rangle$ **do**
7: **if** $suitor^{[i]}(z) = \{u, v\} \backslash \{z\}$ **then**
8: $suitor^{[i+1]}(z) \leftarrow$ `null`
9: $ws^{[i+1]}(z) \leftarrow 0$
10: $affected(z) \leftarrow$ `true`
11: $S_A \leftarrow$ findAffected(z)
12: updateAffected(S_A)
13: $i \leftarrow i + 1$
14: $M^{(f)} \leftarrow \{\{u, v\} \in E$ s.t. $(suitor(u) = v) \wedge (suitor(v) = u)\}$
15: **return** $M^{(f)}$

Proof. Conversely to batches of edge insertions, in case of a batch of edge removals, all the alternating paths computed in Lines 11 and 12 of Algorithm 6 have decreasing edge weights, because no edge is added to the graph. The new matching partner of every vertex $x \in P$ is chosen by findAffected according to Equation (2) and thus once Algorithm 6 finishes, all vertices in G' satisfy Equation (2). □

From Proposition 3.16 it follows that, after Algorithm 6 finishes, $M^{(f)}$ equals M'. Note that an alternating path computed by Algorithm 6 can update vertices adjacent to other removed edges in B; similarly to Algorithm 5, this does not improve the worst-case time complexity of the algorithm but, as reported in Section 4.3.3, makes it faster in practice.

We can combine Algorithms 5 and 6 to handle batches with both edge insertions and removals, with the only difference that we have to use findAffected$_B$ instead of findAffected in Algorithm 6. In this way, we guarantee that every alternating path P computed by our algorithm has decreasing edge weights and thus, as shown in Lemma 3.15, for each vertex $x \in P$ either $suitor^{[i]}(x)$ satisfies Equation (2) in G' or x is adjacent to an edge update in B that is yet to be processed by our algorithm.

Corollary 3.17. *Let B be a batch with $|B| = b$ edge updates. Our dynamic algorithms compute M' in $O(b(n + m))$ worst-case time complexity.*

As shown in Propositions 3.10 and 3.14, the worst-case time complexity of Algorithms 3 and 4 is $O(n + m)$. Thus, after a batch with b edge updates, $M^{(f)} = M'$ is computed in $O(b(n + m))$ time.

3.4 Implementation

We implement SortSuitor, i.e., the variant of Suitor where the adjacency "list" of every vertex is sorted by decreasing edge weight so that every vertex considers a neighbor as matching partner at most once [40]. The adjacency lists are implemented with a dynamic array of dynamic arrays. If the additional preprocessing cost is not taken into account, then Manne and Halappanavar show empirically that SortSuitor is faster than Suitor. We implement this by keeping, for each vertex

u in the graph, an additional index in the adjacency list of u that indicates the next vertex in the adjacency list of u to be considered as potential matching partner for u. Such indices are stored in the array `nextCandidate`; they are incremented in each iteration of the for loop in Line 12 in Algorithm 11 (Appendix A.3), which is interrupted as soon as a new matching partner is found (Line 19).

On dynamic graphs, we need to update the adjacency lists and `nextCandidate` before running both SortSuitor and our dynamic algorithm. In case of b edge insertions, we insert the new edges into the sorted edge lists. Under the reasonable assumption that the newly added edges are inserted at the back of every adjacency list, we sort the new edges and we merge the first (already sorted) part of the adjacency list with the second one. With this strategy the adjacency list of each vertex $x \in V$ can be sorted in $O(\deg(x) + \delta_x \log \delta_x)$, where δ_x is the number of edges in B adjacent to x. When rerunning SortSuitor, for each vertex in G', `nextCandidate` is updated to the first (i.e., heaviest) edge in the adjacency list. In our dynamic algorithm, in turn, for each vertex u adjacent to an edge insertion, `nextCandidate`$[u]$ is updated so that in G' it indicates the same edge as in G. This prevents FINDAFFECTED$_B$ from computing paths with non-decreasing edge weight (as required by Algorithm 5), because every vertex x in an alternating path, when seeking a new partner, can only consider edges that are lighter than $ws^{[i]}(x)$. Newly inserted edges adjacent to x and heavier than $ws^{[i]}(x)$ are taken into account by updating the neighbor index of the current vertex x to the first (i.e., heaviest) edge in the adjacency list of x.

In case of b edge removals, the adjacency lists are updated with the same time complexity as edge insertions—for each vertex $x \in V$, the index of an edge in B adjacent to x can be found in $O(\log \deg(x))$ time and all removed edges adjacent to x can be deleted from the adjacency list in $O(\deg(x))$ time. Concerning the neighbor indices, in SortSuitor they are updated to the first edge in the adjacency list, whereas in our dynamic algorithm this is done only for the vertices adjacent to a removed edge.

4 EXPERIMENTS

We conduct experiments to compare the performance of our dynamic algorithms against the static Suitor algorithm in computing an MWM in fully dynamic graphs with single or multiple-edge updates.

4.1 Settings

All algorithms are implemented in C++, and they use the NetworKit [56] graph APIs. All experiments are conducted on a Linux machine equipped with 192 GB of RAM and an Intel Xeon Gold 6126 CPU with two sockets, 12 cores each (24 cores in total) at 2.6 GHz. Because we want to evaluate the *algorithmic* speedup (i.e., ratio between the sequential running times) of our dynamic algorithms against Suitor, in all our experiments, we only use one core. All the experiments are managed by the SimexPal [2] software to ensure reproducibility; they are executed on both real-world graphs and randomly generated instances—see Tables 1 and 2 in Appendix B. All the complex networks in Table 1 are downloaded from the KONECT [36] repository; the road networks, in turn, are downloaded from OpenStreetMap [49]. From the road networks, we build the pedestrian routing graph using RoutingKit,[4] and choose the geographic distance as weight function. Further weighted networks are downloaded from SuiteSparse.[5] Synthetic networks are generated using the R-MAT [13] and the random hyperbolic[6] models. For the R-MAT model, we use the Graph500 [46]

[4] https://github.com/RoutingKit/RoutingKit.
[5] https://sparse.tamu.edu/.
[6] The random hyperbolic model generates networks with a power-law degree distribution.

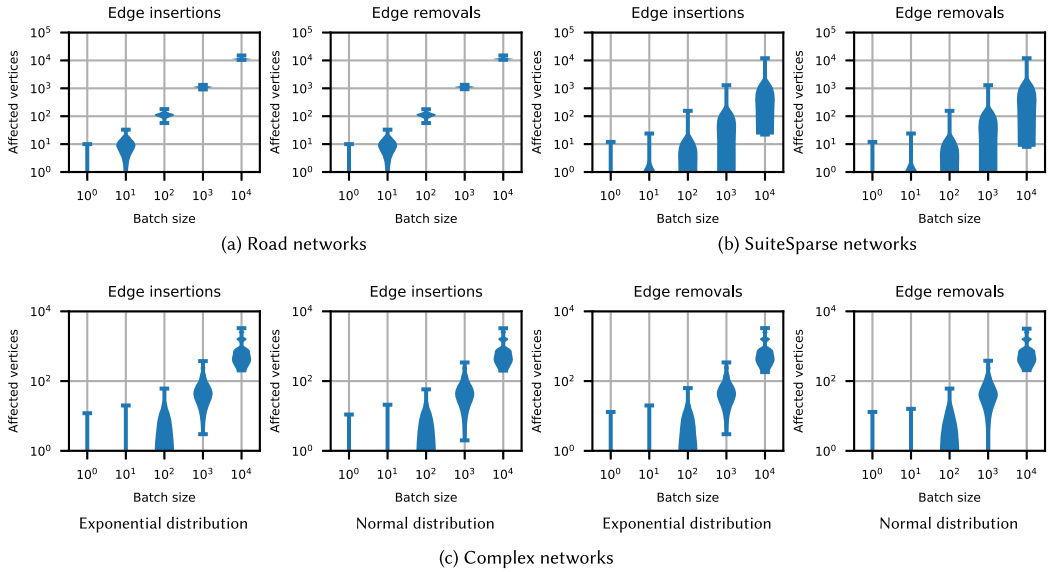


Fig. 5. Violin plots showing the number of affected vertices in the real-world networks of Table 1 in Appendix B. For complex networks, edge weights are randomly generated using a normal and an exponential distribution.

parameter setting (i.e., edge factor 16, $a = 0.57$, $b = 0.19$, $c = 0.19$, and $d = 0.05$), and the generator from Khorasani et al. [34]. For the random hyperbolic model, we use the generator from von Looz et al. [58] within NetworKit; we set the average degree to 20, and the exponent of the power-law distribution to 3. Detailed statistics about synthetic networks are reported in Table 2 in Appendix B. Experiments on synthetic networks are repeated five times, in each one, we generate the network using a different random seed—this results in a different graph for every experiment.

Because the real-world complex networks and the synthetic networks are initially unweighted, we generate edge weights using a normal distribution with mean 1 and standard deviation 0.5, and an exponential distribution with parameter 1. Experiments with random edge weights are repeated five times; in each one we generate random weights using a different random seed.

For each tested graph, we either add or remove a batch of edges selected uniformly at random and run the dynamic algorithm after each batch update. We repeat this process 100 times. For batch insertions, we first remove a random batch of edges from the original graph and re-add them back, whereas for removals, we first add a batch of edges and then remove them. Therefore, after every batch of graph updates the resulting graph G' is always the same, and thus we need to run the static Suitor algorithm only once on G', regardless of the batch size.

4.2 Affected Vertices

We first analyze how many vertices are affected by a batch of edge updates according to Definition 1. The number of affected vertices is summarized in Figure 5 for real-world networks and in Figure 6 for synthetic networks (detailed results are reported in Tables 3 to 10 in Appendix C.1). In road networks, the number of affected vertices is on average moderately higher than the batch size for both edge insertions and removals; intuitively, a random edge update is more likely to update the matching of its adjacent vertices if their degree is low, and road networks are the ones with lowest average degree—see Appendix B. Also, as explained in Section 3, updating

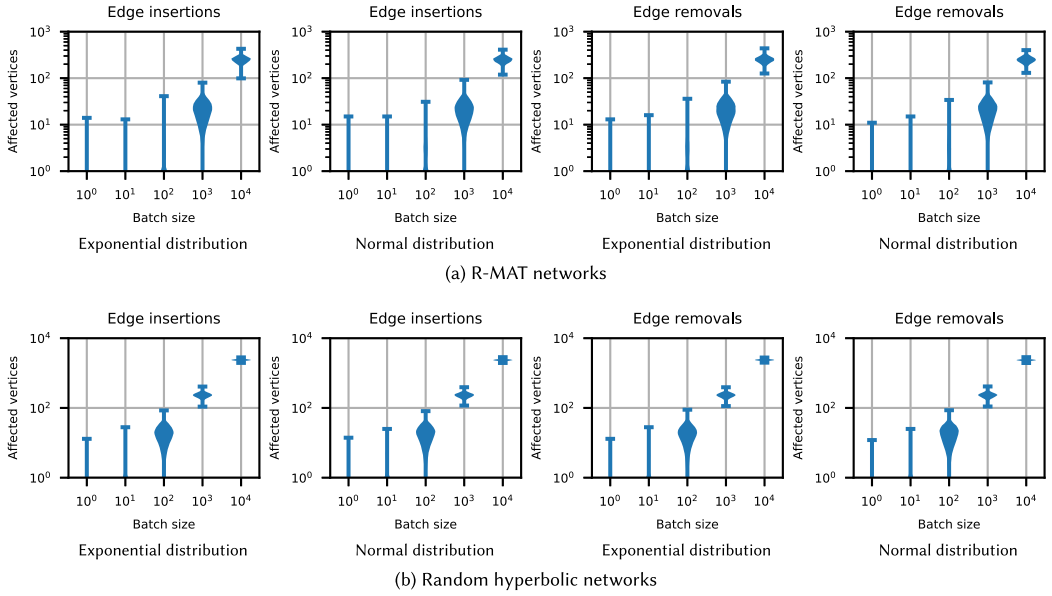


Fig. 6. Violin plots showing the number of affected vertices in the synthetic networks of Table 2 in Appendix B. Edge weights are randomly generated using a normal and an exponential distribution.

the matching of two vertices might also impact the matching of other vertices, which explains why in road networks we have an higher number of affected vertices w.r.t. the batch size.

Regarding complex networks (both real-world and synthetic), their average degree is higher than road networks and, as expected, the number of affected vertices is lower—it is on average one order of magnitude smaller than the batch size. Results do not change notably between the two distributions of edge weights.

4.3 Speedups on the Static Algorithm

We now evaluate the algorithmic speedup of our dynamic Suitor algorithm against a static recomputation, both on real-world and on synthetic networks. Because both the dynamic and the static algorithm need to sort the adjacency lists of the vertices after a batch of edge updates, we discard this step in the speedup computation (i.e., we only compare the running time of both algorithms after the adjacency lists have been sorted). As shown in Figure 9 in Appendix C.4, in terms of running time this preprocessing step is almost negligible as it always takes less than 6%—but mostly less than 2%—of the overall running time of the static Suitor algorithm.

Detailed speedup results are reported in Tables 11 to 18 in Appendix C.2. Running times in seconds are reported in Tables 19 to 22 in Appendix C.3.

4.3.1 Speedups on Real-world Networks. Figure 7(a) summarizes the speedup on road networks. For single-edge insertions and removals, the dynamic algorithm is on average 5 orders of magnitude faster than a static recomputation (geometric mean). As we consider larger batches, the number of affected vertices increases, and thus the dynamic algorithm becomes slower. Nevertheless, the geometric mean of the speedup is still higher than 10^3 for batches with up to 10^3 edge updates. For batches of 10^4 edge insertions and removals, the geometric mean of the speedup is still 196.1× and 310.4×, respectively.

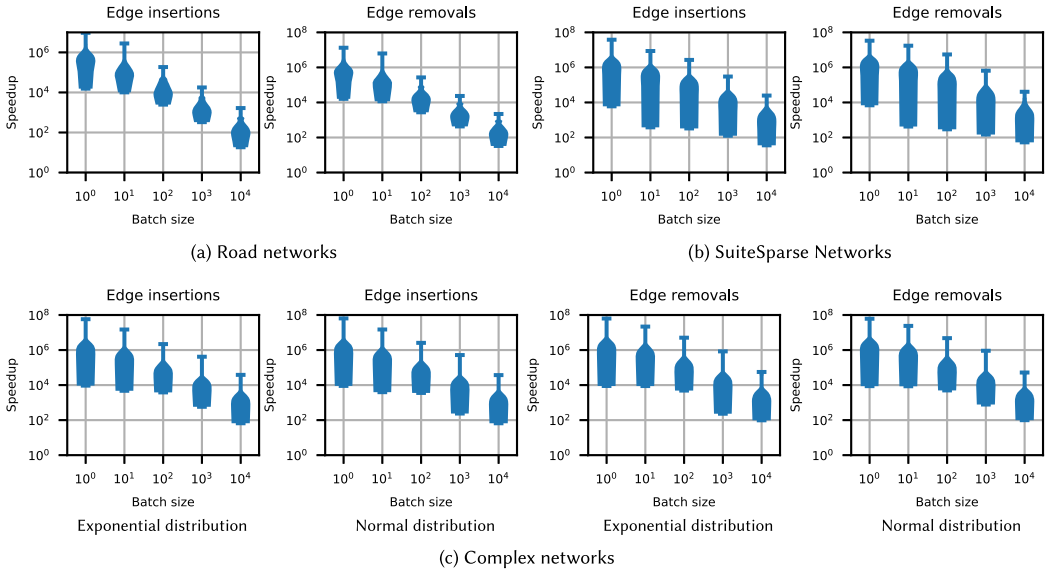


Fig. 7. Violin plots showing the speedups of the dynamic algorithm over a static recomputation over the real-world networks of Table 1 in Appendix B. For complex networks, edge weights are randomly generated using a normal and an exponential distribution.

Concerning complex networks, our dynamic algorithm performs even better: the geometric mean of the speedup is always greater than 10^6 for single-edge updates, and greater than 10^4 for batches of up to 10^3 edge edge updates (Figures 7(b) and 7(c)). For batches of 10^4 edge updates with edge weights generated using a normal distribution, the dynamic algorithm is on average 1,362.1× and 2,135.7× faster than a static recomputation, respectively; using an exponential distribution to generate edge weights yields similar speedups: 1,352.5× for edge insertions and 2,114.1× for edge removals.

As discussed in Section 4.2, better speedups on complex networks can be explained by the fact that the number of affected vertices on complex networks are on average lower compared to road networks, and therefore the dynamic algorithm needs to perform less work. Further, these results show that the worst-case time complexity of our algorithms is very pessimistic compared to their practical performance, and thus that the length of the alternating paths described in Section 3—which determine the running time of our algorithms—is, in practice, usually only a small fraction of the number of vertices in the graph.

Our speedup results are comparable to the ones achieved by Henzinger et al. for MCM [29]: their dynamic algorithms are roughly 10^5× faster than a static recomputation with an *optimal* MCM algorithm. Note that they compare against an exact algorithm, which is speed-wise a weaker baseline than an approximate algorithm. This advantage, however, may be compensated by the fact that their comparisons are run on rather small networks (25K vertices), where higher speedups are more difficult to obtain.

4.3.2 Speedups on Synthetic Networks. Results on R-MAT and random hyperbolic networks are shown in Figures 8(a) and 8(b), respectively. Because speedups results on synthetic networks are not subject to substantial variation, we report the geometric mean of the speedups in bar plots instead of violin plots. Compared to the static Suitor algorithm, for both models and for both distributions of the edge weights, our dynamic algorithm is 5 to 6 orders of magnitude faster on

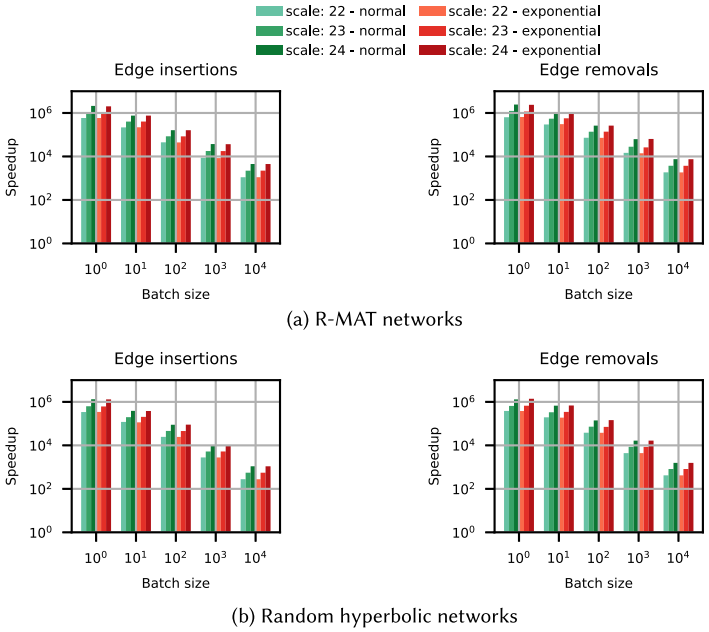


Fig. 8. Geometric mean of the speedups of the dynamic algorithm over a static recomputation over the synthetic networks of of Table 2 in Appendix B. The considered graphs have 2^s vertices, where s is the scale shown in the legend. Edge weights are randomly generated using a normal and an exponential distribution.

single-edge updates, and 3 to 5 orders of magnitude faster on batches with up to 10^3 edge updates. Concerning batches of 10^4 edge updates, the speedup for edge insertions and removals on R-MAT networks is always at least 2,228.7× and 3,719.2×, respectively, and always at least 550.5× and 811.6×, respectively, on random hyperbolic networks.

From Figures 8(a) and 8(b), we can also see that, for every batch size, the speedups increase with the size of the networks. A possible interpretation of this result is that, as explained in Section 4.3.1, even though our algorithms have a worst-case time complexity of $O(n+m)$ for a single-edge update (see Sections 3.1 and 3.2), in a real-world scenario this is too pessimistic and the algorithm is instead much faster. As we have shown in Section 4.2, in complex networks edge updates either do not change the matching of any vertex in the graph (and thus they are handled in constant time), or they affect a very small number of vertices, leading to short processing times.

4.3.3 Speedup of Batch Updates on Single Updates. Finally, we measure the speedup of our batch-dynamic algorithm against the more naïve approach of handling the edge updates in the batch one by one. We perform these experiments for batches of size $b = 100$ edge updates. As described in Section 3.3, although the worst-case time complexity of the two algorithms is the same, in a real-world scenario, we observe that the batch-dynamic algorithm is faster than the naïve one. On road networks (Table 1, Appendix B), the batch-dynamic algorithm is on average 4.3× and 5.3× faster than the naïve one (geometric mean) on batches of edge insertions and removals, respectively. Regarding complex networks (Table 1, Appendix B), when edge weights are generated with a normal distribution, the speedups on batches of edge insertions and removals are 7.7× and 10.2×, respectively; the results for edge weights drawn from an exponential distribution are similar: 7.5× and 10.7× for batches of edge insertions and removals, respectively.

5 CONCLUSIONS

We have developed and implemented a batch-dynamic 1/2-approximation algorithm for MWM based on the Suitor algorithm by Manne and Halappanavar [40]. Our dynamic algorithm updates the matching results from an initial static computation quickly after a batch of edge updates, leading to results that are equivalent to the static algorithm's. Our experimental data show that it can handle in less than a millisecond batch sizes of up to 10^4, thus providing real-time capabilities. Compared to a static recomputation, our dynamic algorithm is 2 to 6 orders of magnitude faster, depending on the input network and on the batch size; further, our speedup results are comparable to the ones achieved by Henzinger et al. for the related dynamic MCM problem [29].

The main reason of such high speedups is that the running time of our dynamic algorithms is determined by the length of the alternating paths. In the worst case, these paths can contain all vertices in the graph. However, as explained in Section 4.3, in practice these paths are much shorter. Conversely, even in a best-case scenario, the complexity of the static Suitor algorithm is linear in the size of the input network.

A possible extension of our dynamic algorithms left for future work is to improve the quality of the solution by adapting the *two-round approach* [40] to dynamic graphs.

APPENDICES

A PSEUDOCODES

Here, we show the pseudocode of the Suitor algorithm by Manne and Halappanavar [40]: the recursive Suitor is shown in Algorithms 7 and 8 (Appendix A.1) while the iterative Suitor is shown in Algorithms 9 and 10 (Appendix A.2). Further, in Appendix A.3, we report the FINDAFFECTED$_B$ function.

A.1 Recursive Suitor

ALGORITHM 7: Recursive FINDSUITOR function.

```
1: function FINDSUITOR(u)
2:     p(u) ← arg max_{v∈N(u)} {w(u,v) : w(u,v) > w(v, suitor(v))}
3:     if p(u) ≠ null then
4:         y ← suitor(u)
5:         suitor(p(u)) ← u
6:         if y ≠ null then
7:             FINDSUITOR (y)
```

ALGORITHM 8: Static recursive Suitor algorithm [40].

Input: Graph $G = (V, E)$.

```
1: for each u ∈ V do
2:     p(u) ← null
3:     suitor(u) ← null
4: for each u ∈ V do
5:     FINDSUITOR(u)
```

A.2 Iterative Suitor

ALGORITHM 9: Iterative FINDSUITOR function.

```
1:  function FINDSUITOR(u)
2:      cur ← u
3:      done ← false
4:      repeat
5:          partner ← suitor(cur)
6:          heaviest ← ws(cur)
7:          for each v ∈ N(cur) do
8:              if w(cur, v) > heaviest and w(cur, v) > ws(v) then
9:                  partner ← v
10:                 heaviest ← w(cur, v)
11:         done ← true
12:         if heaviest > 0 then
13:             y ← suitor(partner)
14:             suitor(partner) ← cur
15:             ws(partner) ← heaviest
16:             if y ≠ null then
17:                 cur ← y
18:                 done ← false
19:     until done is true
```

ALGORITHM 10: Static iterative Suitor algorithm [40].

```
1: for each u ∈ V do
2:     suitor(u) ← null
3:     ws(u) ← 0
4: for each u ∈ V do
5:     FINDSUITOR (u)
```

A.3 Function findAffected for Batches of Edge Updates

ALGORITHM 11: Generalization of the findAffected function (Algorithm 1, Section 3) for batches of edge updates.

```
1:  function findAffected(z)
2:  Input: Affected vertex z
3:  Output: Stack of affected vertices
4:      S_A ← empty stack                                   ▷ Stack of affected vertices.
5:      nextCandidate[z] ← 0
6:      done ← false
7:      cur ← z
8:      repeat
9:          partner ← suitor(cur)
10:         heaviest ← ws(cur)
11:         found ← false
12:         for i ← nextCandidate[cur] to deg(cur) do
13:             x ← adjList[cur][i]          ▷ i-th neighbor in the adjacency list of vertex cur.
14:             nextCandidate[cur] ← i + 1
15:             if not affected(x) and w(cur, x) > heaviest and w(cur, x) > ws(x) then
16:                 partner ← x
17:                 heaviest ← w(cur, x)
18:                 found ← true
19:                 break
20:         done ← true
21:         if found then
22:             y ← suitor(partner)
23:             suitor(partner) ← cur
24:             ws(partner) ← heaviest
25:             S_A.push(partner)
26:             affected(partner) ← true
27:             if y ≠ null then
28:                 suitor(y) ← null
29:                 ws(y) ← 0
30:                 affected(y) ← true
31:                 cur ← y
32:                 done ← false
33:         else
34:             affected(cur) ← false
35:     until done
36:     return S_A
```

B INSTANCE STATISTICS

Tables 1 and 2 reports detailed statistics about the real-world and synthetic instances used in our experiments.

Table 1. Real-world Instances used in the Experiments

Complex Networks

Graph	ID	$\|V\|$	$\|E\|$	Avg. Deg.
hyves	hy	1,402,673	2,777,419	4.0
com-youtube	cy	1,134,890	2,987,624	5.3
flixster	fx	2,523,386	7,918,801	6.3
youtube-u-growth	yg	3,223,589	9,375,374	5.8
flickr-growth	fg	2,302,925	22,838,276	19.8
livejournal-links	ll	5,204,176	48,709,621	18.7
soc-LiveJournal1	lj	4,846,609	68,475,391	14.1
orkut-links	ol	3,072,441	117,184,899	76.3
dimacs10-uk-2002	di	18,483,186	261,787,258	28.3
wikipedia_link_en	we	13,593,032	437,167,958	32.2
twitter	tw	41,652,230	1,468,364,884	35.3
twitter_mpi	tm	52,579,682	1,963,263,507	37.3
friendster	fs	68,349,466	2,586,147,869	37.8

Road Networks

Graph	ID	$\|V\|$	$\|E\|$	Avg. Deg.
belgium	be	1,216,902	1,563,642	2.6
czech-republic	cz	1,713,252	2,181,152	2.5
finland	fi	2,177,796	2,639,775	2.4
austria	au	2,621,866	3,082,590	2.4
canada	ca	3,795,591	4,780,472	2.5
poland	po	5,567,642	7,200,814	2.6
italy	it	6,339,229	7,818,183	2.5
great-britain	gb	7,108,301	8,358,289	2.4
france	fr	11,063,911	13,785,539	2.5
russia	ru	10,984,765	14,079,238	2.6
germany	ge	15,918,055	20,266,409	2.5
dach	da	20,207,259	25,398,909	2.5
africa	af	23,975,266	31,044,959	2.6
us	us	41,256,068	51,271,328	2.5
asia	as	57,736,107	72,020,649	2.5

SuiteSparse Networks

Graph	ID	$\|V\|$	$\|E\|$	Avg. Deg.
human_gene2	hg	14,340	9,027,024	1,259.0
mouse_gene	mg	45,101	14,461,095	641.3
mawi_201512012345	m2	18,571,154	19,020,160	2.0
GAP-road	gr	23,947,347	28,854,312	2.4
mawi_201512020000	m3	35,991,342	37,242,710	2.1
cage15	cg	5,154,859	47,022,346	18.2
mawi_201512020030	m1	68,863,315	71,707,480	2.1
GAP-twitter	gt	61,578,415	1,202,513,046	39.1
GAP-web	gw	50,636,151	1,810,063,330	71.5

Hereafter, we will refer to every instance by its "ID." For complex networks, edge weights are randomly generated using either a normal or an exponential distribution.

Table 2. R-MAT and Random Hyperbolic Networks Used in the Experiments

R-MAT Networks

Graph	$\|V\|$	$\|E\|$	Avg. Deg.
rmat-22	2^{22}	67,108,864	32.0
rmat-23	2^{23}	134,217,728	32.0
rmat-24	2^{24}	268,435,456	32.0

Random Hyperbolic Networks

Graph	$\|V\|$	$\|E\|_{\min}$	$\|E\|_{\text{avg}}$	$\|E\|_{\max}$	Avg. Deg.
hyp-22	2^{22}	41,876,800	41,951,095.8	42,013,293	20.0
hyp-23	2^{23}	83,705,169	83,830,659.2	83,928,747	20.0
hyp-24	2^{24}	167,562,625	167,697,480.2	167,902,689	20.0

Every network is generated five times with a different random seed. For a fixed number of vertices the random hyperbolic generator [58] generates networks with different number of edges; thus, we report the minimum, the average, and the maximum number of edges in the $|E|_{\min}$, $|E|_{\text{avg}}$, and $|E|_{\max}$ columns, respectively. Edge weights are randomly generated using either a normal or an exponential distribution.

C ADDITIONAL EXPERIMENTAL RESULTS

C.1 Affected Vertices

Tables 3 to 8 report the average number of vertices affected by a batch of $b = \{1, \ldots, 10^4\}$ edge insertions or edge removals on all the considered instances.

Table 3. Average Number of Affected Vertices in the Road Networks of Table 1 in Appendix B

	Edge insertions					Edge removals				
Graph	Average #of affected vertices					Graph	Average #of affected vertices				
	$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
be	1.2	12.0	113.5	1,143.7	12,598.5	be	1.3	12.0	113.6	1,144.9	12,945.1
cz	1.3	11.9	116.0	1,135.4	12,179.6	cz	1.5	11.9	116.2	1,135.9	12,383.4
fi	0.9	10.2	109.9	1,121.2	11,983.8	fi	1.0	10.2	109.9	1,123.1	12,158.1
au	0.8	11.3	116.2	1,156.7	12,236.6	au	1.0	11.3	116.2	1,160.1	12,413.1
ca	1.1	12.3	110.2	1,116.2	11,477.2	ca	1.3	12.3	110.2	1,120.9	11,564.0
po	1.1	11.6	113.7	1,120.6	11,295.4	po	1.3	11.6	113.7	1,121.8	11,388.3
it	1.3	10.8	114.2	1,149.6	11,651.4	it	1.6	10.8	114.2	1,149.2	11,697.5
gb	1.0	11.1	116.0	1,150.1	11,726.8	gb	1.1	11.1	116.0	1,150.2	11,792.9
fr	0.8	11.0	116.8	1,156.5	11,539.0	fr	0.9	11.0	116.9	1,158.3	11,592.1
ru	1.2	11.3	112.0	1,096.9	11,117.6	ru	1.4	11.3	111.9	1,097.1	11,153.0
ge	1.1	11.0	113.1	1,129.5	11,356.4	ge	1.1	11.0	113.1	1,129.5	11,383.1
da	1.1	12.3	114.4	1,132.9	11,421.6	da	1.3	12.3	114.4	1,132.9	11,439.9
af	0.8	10.7	111.3	1,088.6	10,981.8	af	0.9	10.7	111.3	1,088.4	10,998.5
us	0.9	10.6	109.1	1,103.7	11,072.7	us	1.0	10.6	109.1	1,103.2	11,081.2
as	0.9	10.9	111.4	1,096.9	11,039.0	as	1.1	10.9	111.4	1,096.9	11,043.5

Table 4. Average Number of Affected Vertices in the SuiteSparse Networks of Table 1 in Appendix B

	Edge insertions					Edge removals				
Graph	Average #of affected vertices					Graph	Average #of affected vertices				
	$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hg	0.0	0.1	1.3	9.2	97.7	hg	0.0	0.1	1.3	9.2	98.8
mg	0.0	0.2	0.8	12.4	128.5	mg	0.0	0.2	0.8	12.2	135.5
m2	0.1	0.4	5.0	55.0	556.8	m2	0.1	0.4	5.0	55.1	557.3
gr	0.9	10.5	112.0	1,144.7	11,523.9	gr	0.9	10.5	112.1	1,143.9	11,533.1
m3	0.1	0.6	5.3	47.3	462.3	m3	0.1	0.6	5.3	47.3	462.5
cg	0.2	1.4	14.9	165.3	1,772.3	cg	0.2	1.4	14.9	163.9	1,773.3
m1	0.1	0.4	4.5	43.4	405.4	m1	0.1	0.4	4.5	43.4	405.1
gt	0.1	0.3	3.3	32.2	330.8	gt	0.1	0.3	3.3	32.2	330.5
gw	0.0	0.2	3.9	35.9	348.4	gw	0.0	0.2	3.9	35.9	348.3

Table 5. Average Number of Affected Vertices in the Complex Networks of Table 1 in Appendix B

Edge insertions						Edge removals					
Graph	Average #of affected vertices					Graph	Average #of affected vertices				
	$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	0.09	1.25	11.60	121.27	1,458.87	hy	0.15	1.24	11.96	119.92	1,461.82
cy	0.24	2.57	24.97	252.53	2,745.08	cy	0.31	2.70	24.65	249.34	2,734.40
fx	0.03	0.37	3.71	38.66	398.10	fx	0.04	0.49	3.70	37.78	398.93
yg	0.29	2.42	23.59	238.19	2,437.38	yg	0.19	2.52	24.55	238.09	2,431.13
fg	0.05	0.85	8.57	82.59	826.29	fg	0.06	0.88	7.86	80.81	819.98
ll	0.14	1.65	16.12	157.23	1,587.56	ll	0.15	1.43	16.23	156.64	1,586.20
lj	0.10	1.85	16.73	162.72	1,625.15	lj	0.10	1.54	16.31	163.42	1,627.85
ol	0.11	0.82	8.17	75.39	761.71	ol	0.04	0.77	7.42	77.20	758.17
di	0.09	0.74	7.36	75.44	745.72	di	0.06	0.71	7.60	76.17	745.27
we	0.01	0.53	4.80	48.77	483.54	we	0.00	0.44	5.05	48.32	484.18
tw	0.03	0.25	3.26	32.70	325.47	tw	0.03	0.38	3.32	32.46	326.62
tm	0.03	0.26	3.13	29.73	297.10	tm	0.01	0.35	2.82	29.07	301.16
fs	0.07	0.54	5.61	58.07	580.53	fs	0.01	0.81	6.04	57.13	576.43

Edge weights are generated using a normal distribution.

Table 6. Average Number of Affected Vertices in the Complex Networks of Table 1 in Appendix B

Edge insertions						Edge removals					
Graph	Average #of affected vertices					Graph	Average #of affected vertices				
	$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	0.08	1.08	12.28	121.90	1,462.90	hy	0.10	1.16	11.88	121.57	1,465.79
cy	0.20	2.29	24.98	253.57	2,758.08	cy	0.19	2.38	24.98	253.28	2,766.21
fx	0.05	0.41	3.96	38.35	400.75	fx	0.05	0.34	3.74	38.19	398.68
yg	0.21	2.52	23.97	239.36	2,447.33	yg	0.24	2.46	23.80	240.16	2,453.34
fg	0.05	0.78	8.09	81.13	827.56	fg	0.04	0.85	7.95	81.96	830.43
ll	0.12	1.57	16.35	158.27	1,597.29	ll	0.19	1.60	15.72	160.37	1,592.24
lj	0.16	1.99	16.77	162.35	1,637.20	lj	0.17	1.81	16.88	162.95	1,636.81
ol	0.06	0.82	8.24	74.27	767.03	ol	0.05	0.86	7.81	75.83	762.64
di	0.07	0.66	7.43	74.23	754.21	di	0.07	0.68	8.00	73.94	751.44
we	0.04	0.41	5.12	48.18	486.44	we	0.04	0.34	4.85	48.05	487.52
tw	0.02	0.25	3.34	32.67	329.47	tw	0.05	0.35	3.19	32.34	327.00
tm	0.04	0.41	3.36	30.91	299.93	tm	0.08	0.33	3.33	30.86	303.08
fs	0.08	0.74	5.74	58.48	585.62	fs	0.09	0.69	5.90	59.63	585.80

Edge weights are generated using an exponential distribution.

Table 7. Average Number of Affected Vertices in R-MAT Networks (Table 2, Appendix B)

Edge insertions						Edge removals					
Graph	Average #of affected vertices					Graph	Average #of affected vertices				
	$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	0.03	0.32	3.16	31.60	317.28	rmat-22	0.04	0.34	3.04	31.23	316.55
rmat-23	0.03	0.31	2.93	28.64	291.35	rmat-23	0.01	0.25	2.84	29.13	289.61
rmat-24	0.02	0.29	2.71	27.60	272.77	rmat-24	0.02	0.27	2.69	27.27	273.35

Edge weights are generated using a normal distribution.

Table 8. Average Number of Affected Vertices in R-MAT Networks (Table 2, Appendix B)

Edge insertions						Edge removals					
Graph	Average #of affected vertices					Graph	Average #of affected vertices				
	$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	0.03	0.31	3.32	31.71	318.74	rmat-22	0.03	0.31	3.23	31.63	318.52
rmat-23	0.03	0.26	3.02	28.79	292.93	rmat-23	0.04	0.26	2.87	28.98	292.65
rmat-24	0.02	0.29	2.69	27.20	273.99	rmat-24	0.04	0.24	2.77	27.43	274.55

Edge weights are generated using an exponential distribution.

Table 9. Average Number of Affected Vertices in Random Hyperbolic Networks (Table 2, Appendix B)

Edge insertions						Edge removals					
Graph	Average #of affected vertices					Graph	Average #of affected vertices				
	$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	0.22	2.38	24.08	236.89	2,381.54	hyp-22	0.24	2.28	23.96	235.66	2,383.13
hyp-23	0.20	2.46	23.43	237.93	2,376.37	hyp-23	0.25	2.53	23.42	236.70	2,376.14
hyp-24	0.24	2.24	23.41	236.56	2,369.67	hyp-24	0.26	2.22	24.05	237.51	2,372.14

Edge weights are generated using a normal distribution.

Table 10. Average Number of Affected Vertices in Random Hyperbolic Networks (Table 2, Appendix B)

Edge insertions						Edge removals					
Graph	Average #of affected vertices					Graph	Average #of affected vertices				
	$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	0.21	2.49	24.15	238.28	2,396.69	hyp-22	0.26	2.52	23.95	237.98	2,400.01
hyp-23	0.20	2.27	23.98	238.23	2,388.64	hyp-23	0.22	2.25	24.16	238.31	2,392.24
hyp-24	0.19	2.38	23.53	238.20	2,387.47	hyp-24	0.22	2.35	23.50	238.74	2,388.09

Edge weights are generated using an exponential distribution.

C.2 Speedups

Tables 11 to 18 report the geometric mean of the speedups of the dynamic Suitor algorithm over a static recomputation on the instances reported in Appendix B. Results are averaged over 100 batches with $\{1, \ldots, 10^4\}$ edge updates.

Table 11. Geometric Mean of the Speedups of the Dynamic Algorithm Over a Static Recomputation on the Road Networks of Table 1

Edge insertions

Graph	Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
be	$1.3 \cdot 10^5$	$2.4 \cdot 10^4$	$3.6 \cdot 10^3$	$4.0 \cdot 10^2$	$3.1 \cdot 10^1$
cz	$1.4 \cdot 10^5$	$3.4 \cdot 10^4$	$5.0 \cdot 10^3$	$5.6 \cdot 10^2$	$4.5 \cdot 10^1$
fi	$1.8 \cdot 10^5$	$4.2 \cdot 10^4$	$5.9 \cdot 10^3$	$6.5 \cdot 10^2$	$5.3 \cdot 10^1$
au	$2.3 \cdot 10^5$	$6.0 \cdot 10^4$	$8.0 \cdot 10^3$	$8.7 \cdot 10^2$	$7.1 \cdot 10^1$
ca	$3.1 \cdot 10^5$	$6.1 \cdot 10^4$	$1.0 \cdot 10^4$	$9.9 \cdot 10^2$	$8.4 \cdot 10^1$
po	$3.8 \cdot 10^5$	$1.1 \cdot 10^5$	$1.7 \cdot 10^4$	$1.8 \cdot 10^3$	$1.5 \cdot 10^2$
it	$4.8 \cdot 10^5$	$1.2 \cdot 10^5$	$1.8 \cdot 10^4$	$1.9 \cdot 10^3$	$1.7 \cdot 10^2$
gb	$6.3 \cdot 10^5$	$1.1 \cdot 10^5$	$2.0 \cdot 10^4$	$2.0 \cdot 10^3$	$1.8 \cdot 10^2$
fr	$9.8 \cdot 10^5$	$1.9 \cdot 10^5$	$2.9 \cdot 10^4$	$3.4 \cdot 10^3$	$3.1 \cdot 10^2$
ru	$6.6 \cdot 10^5$	$1.6 \cdot 10^5$	$2.7 \cdot 10^4$	$3.1 \cdot 10^3$	$2.8 \cdot 10^2$
ge	$1.4 \cdot 10^6$	$2.7 \cdot 10^5$	$4.4 \cdot 10^4$	$5.0 \cdot 10^3$	$4.5 \cdot 10^2$
da	$1.4 \cdot 10^6$	$3.0 \cdot 10^5$	$5.1 \cdot 10^4$	$6.4 \cdot 10^3$	$5.7 \cdot 10^2$
af	$1.6 \cdot 10^6$	$2.8 \cdot 10^5$	$4.7 \cdot 10^4$	$5.5 \cdot 10^3$	$5.1 \cdot 10^2$
us	$2.4 \cdot 10^6$	$4.7 \cdot 10^5$	$7.1 \cdot 10^4$	$9.0 \cdot 10^3$	$8.7 \cdot 10^2$
as	$3.8 \cdot 10^6$	$6.0 \cdot 10^5$	$9.4 \cdot 10^4$	$1.3 \cdot 10^4$	$1.2 \cdot 10^3$
geom. mean	$6.0 \cdot 10^5$	$1.3 \cdot 10^5$	$2.0 \cdot 10^4$	$2.2 \cdot 10^3$	$2.0 \cdot 10^2$

Edge removals

Graph	Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
be	$1.5 \cdot 10^5$	$3.0 \cdot 10^4$	$4.2 \cdot 10^3$	$4.9 \cdot 10^2$	$4.3 \cdot 10^1$
cz	$2.1 \cdot 10^5$	$4.0 \cdot 10^4$	$6.1 \cdot 10^3$	$6.9 \cdot 10^2$	$6.5 \cdot 10^1$
fi	$2.4 \cdot 10^5$	$4.9 \cdot 10^4$	$7.6 \cdot 10^3$	$8.4 \cdot 10^2$	$7.7 \cdot 10^1$
au	$3.6 \cdot 10^5$	$7.1 \cdot 10^4$	$9.8 \cdot 10^3$	$1.1 \cdot 10^3$	$1.0 \cdot 10^2$
ca	$3.4 \cdot 10^5$	$7.3 \cdot 10^4$	$1.2 \cdot 10^4$	$1.4 \cdot 10^3$	$1.4 \cdot 10^2$
po	$4.6 \cdot 10^5$	$1.3 \cdot 10^5$	$2.1 \cdot 10^4$	$2.4 \cdot 10^3$	$2.4 \cdot 10^2$
it	$4.3 \cdot 10^5$	$1.5 \cdot 10^5$	$2.3 \cdot 10^4$	$2.7 \cdot 10^3$	$2.6 \cdot 10^2$
gb	$6.6 \cdot 10^5$	$1.7 \cdot 10^5$	$2.6 \cdot 10^4$	$3.0 \cdot 10^3$	$3.0 \cdot 10^2$
fr	$1.5 \cdot 10^6$	$2.6 \cdot 10^5$	$4.2 \cdot 10^4$	$4.7 \cdot 10^3$	$4.8 \cdot 10^2$
ru	$7.1 \cdot 10^5$	$2.4 \cdot 10^5$	$3.7 \cdot 10^4$	$4.6 \cdot 10^3$	$4.6 \cdot 10^2$
ge	$1.4 \cdot 10^6$	$3.6 \cdot 10^5$	$6.5 \cdot 10^4$	$7.4 \cdot 10^3$	$7.4 \cdot 10^2$
da	$1.8 \cdot 10^6$	$5.2 \cdot 10^5$	$7.8 \cdot 10^4$	$9.2 \cdot 10^3$	$9.5 \cdot 10^2$
af	$1.7 \cdot 10^6$	$4.5 \cdot 10^5$	$7.4 \cdot 10^4$	$8.7 \cdot 10^3$	$8.8 \cdot 10^2$
us	$2.9 \cdot 10^6$	$7.7 \cdot 10^5$	$1.3 \cdot 10^5$	$1.5 \cdot 10^4$	$1.5 \cdot 10^3$
as	$3.9 \cdot 10^6$	$10.0 \cdot 10^5$	$1.8 \cdot 10^5$	$2.1 \cdot 10^4$	$2.1 \cdot 10^3$
geom. mean	$7.1 \cdot 10^5$	$1.7 \cdot 10^5$	$2.8 \cdot 10^4$	$3.2 \cdot 10^3$	$3.1 \cdot 10^2$

Results are averaged over 100 batches with $b \in \{1, \ldots, 10^4\}$ edge updates.

Table 12. Geometric Mean of the Speedups of the Dynamic Algorithm Over a Static Recomputation on the SuiteSparse Networks of Table 1

Edge insertions

Graph	Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hg	$4.4 \cdot 10^4$	$1.6 \cdot 10^4$	$2.6 \cdot 10^3$	$3.4 \cdot 10^2$	$5.2 \cdot 10^1$
mg	$6.7 \cdot 10^4$	$2.5 \cdot 10^4$	$5.2 \cdot 10^3$	$4.9 \cdot 10^2$	$6.7 \cdot 10^1$
m2	$6.3 \cdot 10^5$	$3.2 \cdot 10^5$	$7.1 \cdot 10^4$	$1.1 \cdot 10^4$	$1.3 \cdot 10^3$
gr	$1.6 \cdot 10^6$	$2.6 \cdot 10^5$	$3.6 \cdot 10^4$	$4.6 \cdot 10^3$	$4.5 \cdot 10^2$
m3	$1.2 \cdot 10^6$	$4.5 \cdot 10^5$	$1.2 \cdot 10^5$	$2.2 \cdot 10^4$	$2.5 \cdot 10^3$
cg	$4.7 \cdot 10^5$	$1.8 \cdot 10^5$	$3.1 \cdot 10^4$	$3.5 \cdot 10^3$	$3.5 \cdot 10^2$
m1	$2.1 \cdot 10^6$	$9.1 \cdot 10^5$	$2.2 \cdot 10^5$	$3.7 \cdot 10^4$	$4.4 \cdot 10^3$
gt	$1.9 \cdot 10^7$	$6.2 \cdot 10^6$	$1.6 \cdot 10^6$	$2.0 \cdot 10^5$	$2.2 \cdot 10^4$
gw	$8.1 \cdot 10^6$	$3.8 \cdot 10^6$	$1.0 \cdot 10^6$	$1.4 \cdot 10^5$	$9.7 \cdot 10^3$
geom. mean	$9.3 \cdot 10^5$	$3.4 \cdot 10^5$	$7.2 \cdot 10^4$	$9.5 \cdot 10^3$	$1.1 \cdot 10^3$

Edge removals

Graph	Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hg	$1.4 \cdot 10^5$	$3.8 \cdot 10^4$	$6.1 \cdot 10^3$	$6.4 \cdot 10^2$	$8.5 \cdot 10^1$
mg	$2.2 \cdot 10^5$	$6.0 \cdot 10^4$	$8.1 \cdot 10^3$	$8.7 \cdot 10^2$	$1.0 \cdot 10^2$
m2	$6.8 \cdot 10^5$	$4.0 \cdot 10^5$	$1.2 \cdot 10^5$	$2.2 \cdot 10^4$	$2.7 \cdot 10^3$
gr	$1.8 \cdot 10^6$	$4.3 \cdot 10^5$	$6.0 \cdot 10^4$	$7.0 \cdot 10^3$	$7.2 \cdot 10^2$
m3	$1.3 \cdot 10^6$	$7.1 \cdot 10^5$	$2.2 \cdot 10^5$	$4.5 \cdot 10^4$	$5.4 \cdot 10^3$
cg	$8.8 \cdot 10^5$	$3.2 \cdot 10^5$	$4.0 \cdot 10^4$	$4.3 \cdot 10^3$	$4.2 \cdot 10^2$
m1	$2.3 \cdot 10^6$	$1.4 \cdot 10^6$	$4.2 \cdot 10^5$	$7.9 \cdot 10^4$	$1.1 \cdot 10^4$
gt	$2.1 \cdot 10^7$	$1.0 \cdot 10^7$	$2.4 \cdot 10^6$	$3.2 \cdot 10^5$	$3.5 \cdot 10^4$
gw	$1.5 \cdot 10^7$	$5.6 \cdot 10^6$	$1.6 \cdot 10^6$	$2.3 \cdot 10^5$	$1.7 \cdot 10^4$
geom. mean	$1.5 \cdot 10^6$	$5.8 \cdot 10^5$	$1.2 \cdot 10^5$	$1.7 \cdot 10^4$	$1.8 \cdot 10^3$

Results are averaged over 100 batches with $b \in \{1, \ldots, 10^4\}$ edge updates.

Table 13. Geometric Mean of the Speedups of the Dynamic Algorithm Over a Static Recomputation on the Complex Networks of Table 1 in Appendix B

Edge insertions

Graph	Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	$1.3 \cdot 10^5$	$5.0 \cdot 10^4$	$1.0 \cdot 10^4$	$1.2 \cdot 10^3$	$1.3 \cdot 10^2$
cy	$1.4 \cdot 10^5$	$4.2 \cdot 10^4$	$6.4 \cdot 10^3$	$7.6 \cdot 10^2$	$7.9 \cdot 10^1$
fx	$1.5 \cdot 10^5$	$7.2 \cdot 10^4$	$1.7 \cdot 10^4$	$2.1 \cdot 10^3$	$2.5 \cdot 10^2$
yg	$3.1 \cdot 10^5$	$1.0 \cdot 10^5$	$2.0 \cdot 10^4$	$2.4 \cdot 10^3$	$2.6 \cdot 10^2$
fg	$3.0 \cdot 10^5$	$1.1 \cdot 10^5$	$2.0 \cdot 10^4$	$3.5 \cdot 10^3$	$4.2 \cdot 10^2$
ll	$9.2 \cdot 10^5$	$3.1 \cdot 10^5$	$7.5 \cdot 10^4$	$8.7 \cdot 10^3$	$9.6 \cdot 10^2$
lj	$1.1 \cdot 10^6$	$2.5 \cdot 10^5$	$4.7 \cdot 10^4$	$7.2 \cdot 10^3$	$8.5 \cdot 10^2$
ol	$1.2 \cdot 10^6$	$5.0 \cdot 10^5$	$7.3 \cdot 10^4$	$1.1 \cdot 10^4$	$1.4 \cdot 10^3$
di	$1.5 \cdot 10^6$	$7.3 \cdot 10^5$	$1.7 \cdot 10^5$	$2.1 \cdot 10^4$	$2.2 \cdot 10^3$
we	$3.8 \cdot 10^6$	$1.1 \cdot 10^6$	$1.8 \cdot 10^5$	$2.5 \cdot 10^4$	$4.2 \cdot 10^3$
tw	$1.4 \cdot 10^7$	$4.5 \cdot 10^6$	$8.2 \cdot 10^5$	$1.5 \cdot 10^5$	$2.1 \cdot 10^4$
tm	$1.7 \cdot 10^7$	$6.0 \cdot 10^6$	$1.1 \cdot 10^6$	$1.5 \cdot 10^5$	$2.6 \cdot 10^4$
fs	$2.3 \cdot 10^7$	$9.4 \cdot 10^6$	$1.5 \cdot 10^6$	$2.7 \cdot 10^5$	$3.1 \cdot 10^4$
geom. mean	$1.2 \cdot 10^6$	$4.2 \cdot 10^5$	$7.9 \cdot 10^4$	$1.1 \cdot 10^4$	$1.4 \cdot 10^3$

Edge removals

Graph	Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	$1.6 \cdot 10^5$	$7.8 \cdot 10^4$	$1.8 \cdot 10^4$	$2.2 \cdot 10^3$	$2.3 \cdot 10^2$
cy	$1.6 \cdot 10^5$	$6.2 \cdot 10^4$	$1.1 \cdot 10^4$	$1.2 \cdot 10^3$	$1.2 \cdot 10^2$
fx	$1.9 \cdot 10^5$	$1.0 \cdot 10^5$	$3.5 \cdot 10^4$	$4.8 \cdot 10^3$	$5.7 \cdot 10^2$
yg	$4.3 \cdot 10^5$	$1.4 \cdot 10^5$	$2.9 \cdot 10^4$	$3.7 \cdot 10^3$	$4.0 \cdot 10^2$
fg	$3.2 \cdot 10^5$	$1.5 \cdot 10^5$	$3.2 \cdot 10^4$	$5.7 \cdot 10^3$	$6.4 \cdot 10^2$
ll	$1.0 \cdot 10^6$	$5.0 \cdot 10^5$	$9.2 \cdot 10^4$	$1.3 \cdot 10^4$	$1.3 \cdot 10^3$
lj	$1.4 \cdot 10^6$	$5.1 \cdot 10^5$	$7.4 \cdot 10^4$	$1.0 \cdot 10^4$	$1.2 \cdot 10^3$
ol	$1.6 \cdot 10^6$	$7.5 \cdot 10^5$	$1.4 \cdot 10^5$	$1.6 \cdot 10^4$	$2.0 \cdot 10^3$
di	$2.2 \cdot 10^6$	$1.1 \cdot 10^6$	$2.8 \cdot 10^5$	$3.6 \cdot 10^4$	$3.8 \cdot 10^3$
we	$4.6 \cdot 10^6$	$1.7 \cdot 10^6$	$3.1 \cdot 10^5$	$4.1 \cdot 10^4$	$6.4 \cdot 10^3$
tw	$1.8 \cdot 10^7$	$6.1 \cdot 10^6$	$1.4 \cdot 10^6$	$2.2 \cdot 10^5$	$3.5 \cdot 10^4$
tm	$2.3 \cdot 10^7$	$8.2 \cdot 10^6$	$2.0 \cdot 10^6$	$2.6 \cdot 10^5$	$4.0 \cdot 10^4$
fs	$3.0 \cdot 10^7$	$1.3 \cdot 10^7$	$2.0 \cdot 10^6$	$3.5 \cdot 10^5$	$4.5 \cdot 10^4$
geom. mean	$1.5 \cdot 10^6$	$6.3 \cdot 10^5$	$1.3 \cdot 10^5$	$1.8 \cdot 10^4$	$2.1 \cdot 10^3$

Edge weights are generated using a normal distribution. Results are averaged over 100 batches with $b \in \{1, \ldots, 10^4\}$ random edge updates.

Table 14. Geometric Mean of the Speedups of the Dynamic Algorithm Over a Static Recomputation on the Complex Networks of Table 1 in Appendix B

Edge insertions

Graph	Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	$1.2 \cdot 10^5$	$5.1 \cdot 10^4$	$9.5 \cdot 10^3$	$1.2 \cdot 10^3$	$1.3 \cdot 10^2$
cy	$1.4 \cdot 10^5$	$4.1 \cdot 10^4$	$6.6 \cdot 10^3$	$7.7 \cdot 10^2$	$7.9 \cdot 10^1$
fx	$1.5 \cdot 10^5$	$7.0 \cdot 10^4$	$1.7 \cdot 10^4$	$2.1 \cdot 10^3$	$2.5 \cdot 10^2$
yg	$3.2 \cdot 10^5$	$1.0 \cdot 10^5$	$2.0 \cdot 10^4$	$2.4 \cdot 10^3$	$2.6 \cdot 10^2$
fg	$2.7 \cdot 10^5$	$1.1 \cdot 10^5$	$2.0 \cdot 10^4$	$3.4 \cdot 10^3$	$4.2 \cdot 10^2$
ll	$1.0 \cdot 10^6$	$3.1 \cdot 10^5$	$6.6 \cdot 10^4$	$8.9 \cdot 10^3$	$9.6 \cdot 10^2$
lj	$1.0 \cdot 10^6$	$2.4 \cdot 10^5$	$4.7 \cdot 10^4$	$8.4 \cdot 10^3$	$8.5 \cdot 10^2$
ol	$1.3 \cdot 10^6$	$4.8 \cdot 10^5$	$6.8 \cdot 10^4$	$1.1 \cdot 10^4$	$1.4 \cdot 10^3$
di	$1.5 \cdot 10^6$	$7.1 \cdot 10^5$	$1.7 \cdot 10^5$	$2.1 \cdot 10^4$	$2.2 \cdot 10^3$
we	$3.6 \cdot 10^6$	$1.1 \cdot 10^6$	$1.9 \cdot 10^5$	$2.5 \cdot 10^4$	$4.2 \cdot 10^3$
tw	$1.6 \cdot 10^7$	$4.5 \cdot 10^6$	$8.3 \cdot 10^5$	$1.3 \cdot 10^5$	$2.1 \cdot 10^4$
tm	$1.5 \cdot 10^7$	$6.0 \cdot 10^6$	$1.1 \cdot 10^6$	$1.5 \cdot 10^5$	$2.7 \cdot 10^4$
fs	$2.3 \cdot 10^7$	$8.7 \cdot 10^6$	$1.4 \cdot 10^6$	$2.3 \cdot 10^5$	$3.0 \cdot 10^4$
geom. mean	$1.2 \cdot 10^6$	$4.2 \cdot 10^5$	$7.7 \cdot 10^4$	$1.1 \cdot 10^4$	$1.4 \cdot 10^3$

Edge removals

Graph	Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	$1.7 \cdot 10^5$	$8.1 \cdot 10^4$	$1.7 \cdot 10^4$	$2.2 \cdot 10^3$	$2.3 \cdot 10^2$
cy	$1.8 \cdot 10^5$	$6.6 \cdot 10^4$	$1.0 \cdot 10^4$	$1.2 \cdot 10^3$	$1.2 \cdot 10^2$
fx	$1.9 \cdot 10^5$	$1.2 \cdot 10^5$	$3.5 \cdot 10^4$	$4.6 \cdot 10^3$	$5.7 \cdot 10^2$
yg	$3.9 \cdot 10^5$	$1.4 \cdot 10^5$	$2.9 \cdot 10^4$	$3.7 \cdot 10^3$	$4.0 \cdot 10^2$
fg	$2.9 \cdot 10^5$	$1.5 \cdot 10^5$	$3.1 \cdot 10^4$	$5.4 \cdot 10^3$	$6.3 \cdot 10^2$
ll	$1.1 \cdot 10^6$	$4.8 \cdot 10^5$	$8.8 \cdot 10^4$	$1.2 \cdot 10^4$	$1.3 \cdot 10^3$
lj	$1.3 \cdot 10^6$	$4.9 \cdot 10^5$	$7.3 \cdot 10^4$	$1.0 \cdot 10^4$	$1.2 \cdot 10^3$
ol	$1.5 \cdot 10^6$	$6.9 \cdot 10^5$	$1.4 \cdot 10^5$	$1.8 \cdot 10^4$	$2.0 \cdot 10^3$
di	$2.0 \cdot 10^6$	$1.1 \cdot 10^6$	$2.7 \cdot 10^5$	$3.7 \cdot 10^4$	$3.7 \cdot 10^3$
we	$4.1 \cdot 10^6$	$1.8 \cdot 10^6$	$3.1 \cdot 10^5$	$4.0 \cdot 10^4$	$6.5 \cdot 10^3$
tw	$1.8 \cdot 10^7$	$6.6 \cdot 10^6$	$1.4 \cdot 10^6$	$2.5 \cdot 10^5$	$3.5 \cdot 10^4$
tm	$2.1 \cdot 10^7$	$8.2 \cdot 10^6$	$1.9 \cdot 10^6$	$2.5 \cdot 10^5$	$4.4 \cdot 10^4$
fs	$2.6 \cdot 10^7$	$1.2 \cdot 10^7$	$2.2 \cdot 10^6$	$4.3 \cdot 10^5$	$3.6 \cdot 10^4$
geom. mean	$1.4 \cdot 10^6$	$6.4 \cdot 10^5$	$1.3 \cdot 10^5$	$1.8 \cdot 10^4$	$2.1 \cdot 10^3$

Edge weights are generated using an exponential distribution. Results are averaged over 100 batches with $b \in \{1, \ldots, 10^4\}$ random edge updates.

Table 15. Geometric Mean of the Speedups of the Dynamic Algorithm Over a Static Recomputation on R-MAT Networks of Table 2 in Appendix B

Edge insertions

Graph	Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	$5.9 \cdot 10^5$	$2.2 \cdot 10^5$	$4.5 \cdot 10^4$	$8.7 \cdot 10^3$	$1.1 \cdot 10^3$
rmat-23	$1.1 \cdot 10^6$	$4.0 \cdot 10^5$	$8.5 \cdot 10^4$	$1.8 \cdot 10^4$	$2.2 \cdot 10^3$
rmat-24	$2.1 \cdot 10^6$	$7.6 \cdot 10^5$	$1.6 \cdot 10^5$	$3.7 \cdot 10^4$	$4.5 \cdot 10^3$
geom. mean	$1.1 \cdot 10^6$	$4.0 \cdot 10^5$	$8.5 \cdot 10^4$	$1.8 \cdot 10^4$	$2.2 \cdot 10^3$

Edge removals

Graph	Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	$6.4 \cdot 10^5$	$3.0 \cdot 10^5$	$7.3 \cdot 10^4$	$1.5 \cdot 10^4$	$1.9 \cdot 10^3$
rmat-23	$1.2 \cdot 10^6$	$5.5 \cdot 10^5$	$1.4 \cdot 10^5$	$2.8 \cdot 10^4$	$3.7 \cdot 10^3$
rmat-24	$2.4 \cdot 10^6$	$1.0 \cdot 10^6$	$2.6 \cdot 10^5$	$6.3 \cdot 10^4$	$7.5 \cdot 10^3$
geom. mean	$1.2 \cdot 10^6$	$5.5 \cdot 10^5$	$1.4 \cdot 10^5$	$3.0 \cdot 10^4$	$3.7 \cdot 10^3$

Edge weights are generated using a normal distribution. Results are averaged over 100 batches with $b \in \{1, \ldots, 10^4\}$ random edge updates.

Table 16. Geometric Mean of the Speedups of the Dynamic Algorithm Over a Static Recomputation on R-MAT Networks of Table 2 in Appendix B

Edge insertions

Graph	Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	$5.9 \cdot 10^5$	$2.2 \cdot 10^5$	$4.4 \cdot 10^4$	$8.4 \cdot 10^3$	$1.1 \cdot 10^3$
rmat-23	$1.1 \cdot 10^6$	$4.0 \cdot 10^5$	$8.4 \cdot 10^4$	$1.8 \cdot 10^4$	$2.2 \cdot 10^3$
rmat-24	$2.0 \cdot 10^6$	$7.6 \cdot 10^5$	$1.6 \cdot 10^5$	$3.7 \cdot 10^4$	$4.5 \cdot 10^3$
geom. mean	$1.1 \cdot 10^6$	$4.1 \cdot 10^5$	$8.4 \cdot 10^4$	$1.8 \cdot 10^4$	$2.2 \cdot 10^3$

Edge removals

Graph	Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	$6.5 \cdot 10^5$	$3.1 \cdot 10^5$	$7.2 \cdot 10^4$	$1.4 \cdot 10^4$	$1.8 \cdot 10^3$
rmat-23	$1.2 \cdot 10^6$	$5.7 \cdot 10^5$	$1.4 \cdot 10^5$	$2.6 \cdot 10^4$	$3.7 \cdot 10^3$
rmat-24	$2.4 \cdot 10^6$	$1.0 \cdot 10^6$	$2.6 \cdot 10^5$	$6.4 \cdot 10^4$	$7.5 \cdot 10^3$
geom. mean	$1.2 \cdot 10^6$	$5.7 \cdot 10^5$	$1.4 \cdot 10^5$	$2.9 \cdot 10^4$	$3.7 \cdot 10^3$

Edge weights are generated using a exponential distribution. Results are averaged over 100 batches with $b \in \{1, \ldots, 10^4\}$ random edge updates.

Table 17. Geometric Mean of the Speedups of the Dynamic Algorithm Over a Static Recomputation on Random Hyperbolic Networks of Table 2 in Appendix B

Edge insertions

Graph	Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	$3.4 \cdot 10^5$	$1.2 \cdot 10^5$	$2.5 \cdot 10^4$	$2.8 \cdot 10^3$	$2.8 \cdot 10^2$
hyp-23	$6.3 \cdot 10^5$	$2.0 \cdot 10^5$	$4.6 \cdot 10^4$	$5.2 \cdot 10^3$	$5.5 \cdot 10^2$
hyp-24	$1.3 \cdot 10^6$	$3.9 \cdot 10^5$	$8.9 \cdot 10^4$	$1.0 \cdot 10^4$	$1.1 \cdot 10^3$
geom. mean	$6.6 \cdot 10^5$	$2.1 \cdot 10^5$	$4.7 \cdot 10^4$	$5.3 \cdot 10^3$	$5.5 \cdot 10^2$

Edge removals

Graph	Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	$3.8 \cdot 10^5$	$1.9 \cdot 10^5$	$3.8 \cdot 10^4$	$4.4 \cdot 10^3$	$4.2 \cdot 10^2$
hyp-23	$6.5 \cdot 10^5$	$3.3 \cdot 10^5$	$7.2 \cdot 10^4$	$8.6 \cdot 10^3$	$8.2 \cdot 10^2$
hyp-24	$1.3 \cdot 10^6$	$6.7 \cdot 10^5$	$1.4 \cdot 10^5$	$1.7 \cdot 10^4$	$1.6 \cdot 10^3$
geom. mean	$6.9 \cdot 10^5$	$3.5 \cdot 10^5$	$7.3 \cdot 10^4$	$8.5 \cdot 10^3$	$8.1 \cdot 10^2$

Edge weights are generated using a normal distribution. Results are averaged over 100 batches with $b \in \{1, \ldots, 10^4\}$ random edge updates.

Table 18. Geometric Mean of the Speedups of the Dynamic Algorithm Over a Static Recomputation on Random Hyperbolic Networks of Table 2 in Appendix B

Edge insertions

Graph	Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	$3.4 \cdot 10^5$	$1.1 \cdot 10^5$	$2.4 \cdot 10^4$	$2.8 \cdot 10^3$	$2.8 \cdot 10^2$
hyp-23	$6.2 \cdot 10^5$	$2.0 \cdot 10^5$	$4.6 \cdot 10^4$	$5.2 \cdot 10^3$	$5.5 \cdot 10^2$
hyp-24	$1.3 \cdot 10^6$	$3.8 \cdot 10^5$	$9.0 \cdot 10^4$	$1.0 \cdot 10^4$	$1.1 \cdot 10^3$
geom. mean	$6.5 \cdot 10^5$	$2.1 \cdot 10^5$	$4.6 \cdot 10^4$	$5.3 \cdot 10^3$	$5.5 \cdot 10^2$

Edge removals

Graph	Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	$3.8 \cdot 10^5$	$1.9 \cdot 10^5$	$3.8 \cdot 10^4$	$4.4 \cdot 10^3$	$4.2 \cdot 10^2$
hyp-23	$6.7 \cdot 10^5$	$3.5 \cdot 10^5$	$7.1 \cdot 10^4$	$8.5 \cdot 10^3$	$8.2 \cdot 10^2$
hyp-24	$1.4 \cdot 10^6$	$6.8 \cdot 10^5$	$1.4 \cdot 10^5$	$1.7 \cdot 10^4$	$1.6 \cdot 10^3$
geom. mean	$7.1 \cdot 10^5$	$3.5 \cdot 10^5$	$7.3 \cdot 10^4$	$8.5 \cdot 10^3$	$8.1 \cdot 10^2$

Edge weights are generated using a exponential distribution. Results are averaged over 100 batches with $b \in \{1, \ldots, 10^4\}$ random edge updates.

C.3 Running Times

Tables 19 to 26 report the average running time in seconds of both the static and dynamic Suitor algorithms. Results are averaged over 100 batches with $\{1, \ldots, 10^4\}$ edge updates. As explained in Section 4.1, on every run of our experiments we create a batch of edge insertions or removals by preemptively removing or adding a batch of edges selected uniformly at random. Thus, after every batch of graph updates, the resulting graph G' (i.e., the graph on which we measure the running time of both the static and the dynamic algorithms) is always the same (and we need to run the static Suitor algorithm only once, regardless of the batch size).

Table 19. Average Running Time in Seconds of the Static and Dynamic Suitor Algorithms for 100 Batches of $b \in \{1, \ldots, 10^4\}$ Edge Updates on the Road Networks of of Table 1

Edge insertions

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
be	0.07	$5.5 \cdot 10^{-7}$	$3.0 \cdot 10^{-6}$	$2.1 \cdot 10^{-5}$	$1.9 \cdot 10^{-4}$	$2.0 \cdot 10^{-3}$
cz	0.10	$7.2 \cdot 10^{-7}$	$3.1 \cdot 10^{-6}$	$2.1 \cdot 10^{-5}$	$1.9 \cdot 10^{-4}$	$2.1 \cdot 10^{-3}$
fi	0.11	$6.2 \cdot 10^{-7}$	$2.7 \cdot 10^{-6}$	$1.9 \cdot 10^{-5}$	$1.8 \cdot 10^{-4}$	$2.0 \cdot 10^{-3}$
au	0.16	$6.9 \cdot 10^{-7}$	$2.7 \cdot 10^{-6}$	$2.0 \cdot 10^{-5}$	$1.9 \cdot 10^{-4}$	$2.2 \cdot 10^{-3}$
ca	0.20	$6.5 \cdot 10^{-7}$	$3.3 \cdot 10^{-6}$	$2.0 \cdot 10^{-5}$	$2.1 \cdot 10^{-4}$	$2.4 \cdot 10^{-3}$
po	0.36	$9.8 \cdot 10^{-7}$	$3.4 \cdot 10^{-6}$	$2.2 \cdot 10^{-5}$	$2.0 \cdot 10^{-4}$	$2.3 \cdot 10^{-3}$
it	0.41	$8.6 \cdot 10^{-7}$	$3.3 \cdot 10^{-6}$	$2.2 \cdot 10^{-5}$	$2.2 \cdot 10^{-4}$	$2.3 \cdot 10^{-3}$
gb	0.45	$7.2 \cdot 10^{-7}$	$4.3 \cdot 10^{-6}$	$2.3 \cdot 10^{-5}$	$2.3 \cdot 10^{-4}$	$2.4 \cdot 10^{-3}$
fr	0.72	$7.7 \cdot 10^{-7}$	$3.9 \cdot 10^{-6}$	$2.6 \cdot 10^{-5}$	$2.1 \cdot 10^{-4}$	$2.1 \cdot 10^{-3}$
ru	0.67	$1.0 \cdot 10^{-6}$	$4.4 \cdot 10^{-6}$	$2.5 \cdot 10^{-5}$	$2.1 \cdot 10^{-4}$	$2.3 \cdot 10^{-3}$
ge	1.17	$8.8 \cdot 10^{-7}$	$4.5 \cdot 10^{-6}$	$2.7 \cdot 10^{-5}$	$2.3 \cdot 10^{-4}$	$2.5 \cdot 10^{-3}$
da	1.46	$1.1 \cdot 10^{-6}$	$4.6 \cdot 10^{-6}$	$3.0 \cdot 10^{-5}$	$2.3 \cdot 10^{-4}$	$2.5 \cdot 10^{-3}$
af	1.27	$8.4 \cdot 10^{-7}$	$4.7 \cdot 10^{-6}$	$2.6 \cdot 10^{-5}$	$2.3 \cdot 10^{-4}$	$2.5 \cdot 10^{-3}$
us	2.29	$9.8 \cdot 10^{-7}$	$5.0 \cdot 10^{-6}$	$3.3 \cdot 10^{-5}$	$2.5 \cdot 10^{-4}$	$2.6 \cdot 10^{-3}$
as	3.20	$8.6 \cdot 10^{-7}$	$5.5 \cdot 10^{-6}$	$3.4 \cdot 10^{-5}$	$2.5 \cdot 10^{-4}$	$2.5 \cdot 10^{-3}$

Edge removals

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
be	0.07	$4.7 \cdot 10^{-7}$	$2.5 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.6 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
cz	0.10	$5.0 \cdot 10^{-7}$	$2.6 \cdot 10^{-6}$	$1.7 \cdot 10^{-5}$	$1.6 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
fi	0.11	$4.7 \cdot 10^{-7}$	$2.3 \cdot 10^{-6}$	$1.5 \cdot 10^{-5}$	$1.4 \cdot 10^{-4}$	$1.4 \cdot 10^{-3}$
au	0.16	$4.5 \cdot 10^{-7}$	$2.3 \cdot 10^{-6}$	$1.6 \cdot 10^{-5}$	$1.6 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
ca	0.20	$5.9 \cdot 10^{-7}$	$2.7 \cdot 10^{-6}$	$1.6 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
po	0.36	$8.1 \cdot 10^{-7}$	$2.9 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
it	0.41	$9.6 \cdot 10^{-7}$	$2.8 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
gb	0.45	$6.9 \cdot 10^{-7}$	$2.8 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$1.4 \cdot 10^{-3}$
fr	0.74	$5.0 \cdot 10^{-7}$	$2.9 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
ru	0.67	$9.6 \cdot 10^{-7}$	$2.9 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.4 \cdot 10^{-4}$	$1.4 \cdot 10^{-3}$
ge	1.17	$8.7 \cdot 10^{-7}$	$3.3 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
da	1.49	$8.5 \cdot 10^{-7}$	$2.9 \cdot 10^{-6}$	$1.9 \cdot 10^{-5}$	$1.6 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
af	1.26	$7.8 \cdot 10^{-7}$	$2.7 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.4 \cdot 10^{-4}$	$1.4 \cdot 10^{-3}$
us	2.30	$8.1 \cdot 10^{-7}$	$3.0 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
as	3.19	$8.5 \cdot 10^{-7}$	$3.3 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$

The columns "Static" and "Dynamic" report the average time (in seconds) for the static and for the dynamic algorithm, respectively.

Table 20. Average Running Time in Seconds of the Static and Dynamic Suitor Algorithms for 100 Batches of $b \in \{1, \ldots, 10^4\}$ Edge Updates on the SuiteSparse Networks of Table 1

Edge insertions

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hg	0.01	$2.0 \cdot 10^{-7}$	$5.3 \cdot 10^{-7}$	$3.2 \cdot 10^{-6}$	$2.5 \cdot 10^{-5}$	$1.6 \cdot 10^{-4}$
mg	0.01	$2.2 \cdot 10^{-7}$	$6.0 \cdot 10^{-7}$	$2.9 \cdot 10^{-6}$	$2.7 \cdot 10^{-5}$	$2.3 \cdot 10^{-4}$
m2	0.40	$6.5 \cdot 10^{-7}$	$1.3 \cdot 10^{-6}$	$5.7 \cdot 10^{-6}$	$3.4 \cdot 10^{-5}$	$3.2 \cdot 10^{-4}$
gr	1.09	$6.5 \cdot 10^{-7}$	$4.1 \cdot 10^{-6}$	$3.0 \cdot 10^{-5}$	$2.6 \cdot 10^{-4}$	$2.4 \cdot 10^{-3}$
m3	0.74	$6.5 \cdot 10^{-7}$	$1.5 \cdot 10^{-6}$	$6.1 \cdot 10^{-6}$	$3.5 \cdot 10^{-5}$	$3.0 \cdot 10^{-4}$
cg	0.27	$5.9 \cdot 10^{-7}$	$1.5 \cdot 10^{-6}$	$8.7 \cdot 10^{-6}$	$8.0 \cdot 10^{-5}$	$7.3 \cdot 10^{-4}$
m1	1.39	$6.5 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$	$6.4 \cdot 10^{-6}$	$3.9 \cdot 10^{-5}$	$3.1 \cdot 10^{-4}$
gt	10.13	$5.6 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$6.3 \cdot 10^{-6}$	$4.9 \cdot 10^{-5}$	$4.1 \cdot 10^{-4}$
gw	6.77	$9.4 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$6.3 \cdot 10^{-6}$	$4.8 \cdot 10^{-5}$	$6.8 \cdot 10^{-4}$

Edge removals

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hg	0.01	$6.3 \cdot 10^{-8}$	$2.2 \cdot 10^{-7}$	$1.4 \cdot 10^{-6}$	$1.4 \cdot 10^{-5}$	$1.1 \cdot 10^{-4}$
mg	0.01	$6.5 \cdot 10^{-8}$	$2.4 \cdot 10^{-7}$	$1.8 \cdot 10^{-6}$	$1.7 \cdot 10^{-5}$	$1.6 \cdot 10^{-4}$
m2	0.40	$6.0 \cdot 10^{-7}$	$1.0 \cdot 10^{-6}$	$3.3 \cdot 10^{-6}$	$1.7 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$
gr	1.10	$6.6 \cdot 10^{-7}$	$2.8 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.4 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
m3	0.76	$5.9 \cdot 10^{-7}$	$1.1 \cdot 10^{-6}$	$3.5 \cdot 10^{-6}$	$1.7 \cdot 10^{-5}$	$1.4 \cdot 10^{-4}$
cg	0.27	$3.1 \cdot 10^{-7}$	$8.4 \cdot 10^{-7}$	$6.9 \cdot 10^{-6}$	$6.6 \cdot 10^{-5}$	$6.5 \cdot 10^{-4}$
m1	1.34	$6.4 \cdot 10^{-7}$	$9.4 \cdot 10^{-7}$	$3.2 \cdot 10^{-6}$	$1.6 \cdot 10^{-5}$	$1.3 \cdot 10^{-4}$
gt	10.28	$4.5 \cdot 10^{-7}$	$1.0 \cdot 10^{-6}$	$4.4 \cdot 10^{-6}$	$3.3 \cdot 10^{-5}$	$2.8 \cdot 10^{-4}$
gw	6.51	$4.2 \cdot 10^{-7}$	$1.2 \cdot 10^{-6}$	$4.1 \cdot 10^{-6}$	$2.8 \cdot 10^{-5}$	$3.8 \cdot 10^{-4}$

The columns "Static" and "Dynamic" report the average time (in seconds) for the static and for the dynamic algorithm, respectively.

Table 21. Average Running Time in Seconds of the Static and Dynamic Suitor Algorithms for 100 Batches of $b \in \{1, \ldots, 10^4\}$ Edge Updates on the Complex Networks of Table 1

Edge insertions

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	0.08	$6.0 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$	$7.8 \cdot 10^{-6}$	$6.4 \cdot 10^{-5}$	$5.6 \cdot 10^{-4}$
cy	0.06	$4.5 \cdot 10^{-7}$	$1.5 \cdot 10^{-6}$	$9.9 \cdot 10^{-6}$	$8.3 \cdot 10^{-5}$	$7.3 \cdot 10^{-4}$
fx	0.11	$7.5 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$	$6.8 \cdot 10^{-6}$	$5.1 \cdot 10^{-5}$	$3.9 \cdot 10^{-4}$
yg	0.21	$7.1 \cdot 10^{-7}$	$2.1 \cdot 10^{-6}$	$1.1 \cdot 10^{-5}$	$9.0 \cdot 10^{-5}$	$7.9 \cdot 10^{-4}$
fg	0.19	$6.4 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$9.6 \cdot 10^{-6}$	$5.3 \cdot 10^{-5}$	$4.1 \cdot 10^{-4}$
ll	0.69	$7.6 \cdot 10^{-7}$	$2.3 \cdot 10^{-6}$	$9.4 \cdot 10^{-6}$	$8.0 \cdot 10^{-5}$	$6.7 \cdot 10^{-4}$
lj	0.68	$6.5 \cdot 10^{-7}$	$2.9 \cdot 10^{-6}$	$1.5 \cdot 10^{-5}$	$9.7 \cdot 10^{-5}$	$6.6 \cdot 10^{-4}$
ol	0.80	$7.0 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$	$1.1 \cdot 10^{-5}$	$7.4 \cdot 10^{-5}$	$5.2 \cdot 10^{-4}$
di	1.19	$7.9 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$7.1 \cdot 10^{-6}$	$5.6 \cdot 10^{-5}$	$5.0 \cdot 10^{-4}$
we	2.15	$5.9 \cdot 10^{-7}$	$2.1 \cdot 10^{-6}$	$1.2 \cdot 10^{-5}$	$8.9 \cdot 10^{-5}$	$4.4 \cdot 10^{-4}$
tw	11.23	$8.9 \cdot 10^{-7}$	$2.5 \cdot 10^{-6}$	$1.4 \cdot 10^{-5}$	$7.6 \cdot 10^{-5}$	$4.7 \cdot 10^{-4}$
tm	14.45	$8.9 \cdot 10^{-7}$	$2.5 \cdot 10^{-6}$	$1.3 \cdot 10^{-5}$	$1.0 \cdot 10^{-4}$	$5.0 \cdot 10^{-4}$
fs	21.51	$1.0 \cdot 10^{-6}$	$2.5 \cdot 10^{-6}$	$1.5 \cdot 10^{-5}$	$7.6 \cdot 10^{-5}$	$6.0 \cdot 10^{-4}$

Edge removals

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	0.08	$5.1 \cdot 10^{-7}$	$1.0 \cdot 10^{-6}$	$4.6 \cdot 10^{-6}$	$3.5 \cdot 10^{-5}$	$3.2 \cdot 10^{-4}$
cy	0.06	$4.1 \cdot 10^{-7}$	$1.0 \cdot 10^{-6}$	$6.0 \cdot 10^{-6}$	$5.4 \cdot 10^{-5}$	$4.9 \cdot 10^{-4}$
fx	0.11	$5.9 \cdot 10^{-7}$	$1.1 \cdot 10^{-6}$	$3.3 \cdot 10^{-6}$	$2.3 \cdot 10^{-5}$	$1.8 \cdot 10^{-4}$
yg	0.22	$5.1 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$	$7.6 \cdot 10^{-6}$	$5.8 \cdot 10^{-5}$	$5.2 \cdot 10^{-4}$
fg	0.19	$6.0 \cdot 10^{-7}$	$1.2 \cdot 10^{-6}$	$6.0 \cdot 10^{-6}$	$3.3 \cdot 10^{-5}$	$2.7 \cdot 10^{-4}$
ll	0.69	$6.9 \cdot 10^{-7}$	$1.4 \cdot 10^{-6}$	$7.6 \cdot 10^{-6}$	$5.5 \cdot 10^{-5}$	$4.8 \cdot 10^{-4}$
lj	0.68	$5.2 \cdot 10^{-7}$	$1.4 \cdot 10^{-6}$	$9.6 \cdot 10^{-6}$	$6.8 \cdot 10^{-5}$	$4.7 \cdot 10^{-4}$
ol	0.80	$5.2 \cdot 10^{-7}$	$1.1 \cdot 10^{-6}$	$5.7 \cdot 10^{-6}$	$5.2 \cdot 10^{-5}$	$3.7 \cdot 10^{-4}$
di	1.19	$5.6 \cdot 10^{-7}$	$1.1 \cdot 10^{-6}$	$4.3 \cdot 10^{-6}$	$3.4 \cdot 10^{-5}$	$3.0 \cdot 10^{-4}$
we	2.15	$4.9 \cdot 10^{-7}$	$1.3 \cdot 10^{-6}$	$7.3 \cdot 10^{-6}$	$5.4 \cdot 10^{-5}$	$2.8 \cdot 10^{-4}$
tw	11.46	$7.0 \cdot 10^{-7}$	$1.9 \cdot 10^{-6}$	$8.0 \cdot 10^{-6}$	$5.3 \cdot 10^{-5}$	$2.9 \cdot 10^{-4}$
tm	14.47	$6.5 \cdot 10^{-7}$	$1.8 \cdot 10^{-6}$	$7.4 \cdot 10^{-6}$	$5.5 \cdot 10^{-5}$	$3.3 \cdot 10^{-4}$
fs	20.99	$7.3 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$1.1 \cdot 10^{-5}$	$5.8 \cdot 10^{-5}$	$4.2 \cdot 10^{-4}$

The columns "Static" and "Dynamic" report the average time (in seconds) for the static and for the dynamic algorithm, respectively. Edge weights are generated using a normal distribution.

Table 22. Average Running Time in Seconds of the Static and Dynamic Suitor Algorithms for 100 Batches of $b \in \{1, \ldots, 10^4\}$ Edge Updates on the Complex Networks of Table 1

Edge insertions

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	0.08	$6.5 \cdot 10^{-7}$	$1.5 \cdot 10^{-6}$	$8.3 \cdot 10^{-6}$	$6.4 \cdot 10^{-5}$	$5.7 \cdot 10^{-4}$
cy	0.06	$4.6 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$	$9.7 \cdot 10^{-6}$	$8.3 \cdot 10^{-5}$	$7.4 \cdot 10^{-4}$
fx	0.11	$7.3 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$	$6.7 \cdot 10^{-6}$	$5.0 \cdot 10^{-5}$	$3.9 \cdot 10^{-4}$
yg	0.21	$6.8 \cdot 10^{-7}$	$2.1 \cdot 10^{-6}$	$1.1 \cdot 10^{-5}$	$8.8 \cdot 10^{-5}$	$7.9 \cdot 10^{-4}$
fg	0.19	$7.1 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$9.5 \cdot 10^{-6}$	$5.5 \cdot 10^{-5}$	$4.1 \cdot 10^{-4}$
ll	0.69	$7.0 \cdot 10^{-7}$	$2.3 \cdot 10^{-6}$	$1.1 \cdot 10^{-5}$	$7.7 \cdot 10^{-5}$	$6.7 \cdot 10^{-4}$
lj	0.68	$7.1 \cdot 10^{-7}$	$2.9 \cdot 10^{-6}$	$1.5 \cdot 10^{-5}$	$8.2 \cdot 10^{-5}$	$6.6 \cdot 10^{-4}$
ol	0.80	$6.5 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$1.2 \cdot 10^{-5}$	$7.4 \cdot 10^{-5}$	$5.2 \cdot 10^{-4}$
di	1.18	$8.0 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$7.1 \cdot 10^{-6}$	$5.6 \cdot 10^{-5}$	$5.0 \cdot 10^{-4}$
we	2.14	$6.3 \cdot 10^{-7}$	$2.0 \cdot 10^{-6}$	$1.2 \cdot 10^{-5}$	$8.9 \cdot 10^{-5}$	$4.4 \cdot 10^{-4}$
tw	11.41	$7.9 \cdot 10^{-7}$	$2.6 \cdot 10^{-6}$	$1.4 \cdot 10^{-5}$	$9.1 \cdot 10^{-5}$	$4.6 \cdot 10^{-4}$
tm	14.66	$1.0 \cdot 10^{-6}$	$2.5 \cdot 10^{-6}$	$1.3 \cdot 10^{-5}$	$1.1 \cdot 10^{-4}$	$4.8 \cdot 10^{-4}$
fs	21.69	$1.0 \cdot 10^{-6}$	$2.6 \cdot 10^{-6}$	$1.6 \cdot 10^{-5}$	$9.3 \cdot 10^{-5}$	$6.5 \cdot 10^{-4}$

Edge removals

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	0.08	$4.7 \cdot 10^{-7}$	$9.8 \cdot 10^{-7}$	$4.5 \cdot 10^{-6}$	$3.5 \cdot 10^{-5}$	$3.2 \cdot 10^{-4}$
cy	0.06	$3.7 \cdot 10^{-7}$	$9.7 \cdot 10^{-7}$	$6.2 \cdot 10^{-6}$	$5.4 \cdot 10^{-5}$	$4.9 \cdot 10^{-4}$
fx	0.11	$6.0 \cdot 10^{-7}$	$9.6 \cdot 10^{-7}$	$3.2 \cdot 10^{-6}$	$2.3 \cdot 10^{-5}$	$1.8 \cdot 10^{-4}$
yg	0.22	$5.6 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$	$7.6 \cdot 10^{-6}$	$5.9 \cdot 10^{-5}$	$5.2 \cdot 10^{-4}$
fg	0.19	$6.5 \cdot 10^{-7}$	$1.3 \cdot 10^{-6}$	$6.1 \cdot 10^{-6}$	$3.5 \cdot 10^{-5}$	$2.7 \cdot 10^{-4}$
ll	0.69	$6.2 \cdot 10^{-7}$	$1.5 \cdot 10^{-6}$	$8.1 \cdot 10^{-6}$	$5.8 \cdot 10^{-5}$	$4.8 \cdot 10^{-4}$
lj	0.68	$5.6 \cdot 10^{-7}$	$1.5 \cdot 10^{-6}$	$9.8 \cdot 10^{-6}$	$6.9 \cdot 10^{-5}$	$4.7 \cdot 10^{-4}$
ol	0.80	$5.5 \cdot 10^{-7}$	$1.2 \cdot 10^{-6}$	$6.0 \cdot 10^{-6}$	$4.6 \cdot 10^{-5}$	$3.7 \cdot 10^{-4}$
di	1.18	$6.1 \cdot 10^{-7}$	$1.1 \cdot 10^{-6}$	$4.5 \cdot 10^{-6}$	$3.3 \cdot 10^{-5}$	$3.0 \cdot 10^{-4}$
we	2.16	$5.6 \cdot 10^{-7}$	$1.2 \cdot 10^{-6}$	$7.2 \cdot 10^{-6}$	$5.5 \cdot 10^{-5}$	$2.9 \cdot 10^{-4}$
tw	11.47	$7.2 \cdot 10^{-7}$	$1.8 \cdot 10^{-6}$	$7.9 \cdot 10^{-6}$	$4.7 \cdot 10^{-5}$	$2.8 \cdot 10^{-4}$
tm	14.62	$7.5 \cdot 10^{-7}$	$1.9 \cdot 10^{-6}$	$7.9 \cdot 10^{-6}$	$5.7 \cdot 10^{-5}$	$2.9 \cdot 10^{-4}$
fs	21.26	$8.2 \cdot 10^{-7}$	$1.9 \cdot 10^{-6}$	$1.0 \cdot 10^{-5}$	$5.0 \cdot 10^{-5}$	$5.0 \cdot 10^{-4}$

The columns "Static" and "Dynamic" report the average time (in seconds) for the static and for the dynamic algorithm, respectively. Edge weights are generated using an exponential distribution.

Table 23. Average Running Time in Seconds of the Static and Dynamic Suitor Algorithms for 100 Batches of $b \in \{1, \ldots, 10^4\}$ Edge Updates on the R-MAT Networks of Table 2

Edge insertions

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	0.41	$7.1 \cdot 10^{-7}$	$1.9 \cdot 10^{-6}$	$9.3 \cdot 10^{-6}$	$4.8 \cdot 10^{-5}$	$3.5 \cdot 10^{-4}$
rmat-23	0.84	$8.1 \cdot 10^{-7}$	$2.1 \cdot 10^{-6}$	$1.0 \cdot 10^{-5}$	$4.8 \cdot 10^{-5}$	$3.6 \cdot 10^{-4}$
rmat-24	1.73	$8.3 \cdot 10^{-7}$	$2.3 \cdot 10^{-6}$	$1.1 \cdot 10^{-5}$	$4.7 \cdot 10^{-5}$	$3.6 \cdot 10^{-4}$

Edge removals

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	0.41	$6.6 \cdot 10^{-7}$	$1.4 \cdot 10^{-6}$	$5.7 \cdot 10^{-6}$	$2.8 \cdot 10^{-5}$	$2.1 \cdot 10^{-4}$
rmat-23	0.85	$6.9 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$	$6.2 \cdot 10^{-6}$	$3.1 \cdot 10^{-5}$	$2.1 \cdot 10^{-4}$
rmat-24	1.72	$7.2 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$6.7 \cdot 10^{-6}$	$2.8 \cdot 10^{-5}$	$2.2 \cdot 10^{-4}$

The columns "Static" and "Dynamic" report the average time (in seconds) for the static and for the dynamic algorithm, respectively. Edge weights are generated using a normal distribution.

Table 24. Average Running Time in Seconds of the Static and Dynamic Suitor Algorithms for 100 Batches of $b \in \{1, \ldots, 10^4\}$ Edge Updates on the R-MAT Networks of Table 2

Edge insertions

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	0.41	$7.1 \cdot 10^{-7}$	$1.9 \cdot 10^{-6}$	$9.4 \cdot 10^{-6}$	$5.0 \cdot 10^{-5}$	$3.5 \cdot 10^{-4}$
rmat-23	0.85	$7.9 \cdot 10^{-7}$	$2.1 \cdot 10^{-6}$	$1.0 \cdot 10^{-5}$	$4.9 \cdot 10^{-5}$	$3.6 \cdot 10^{-4}$
rmat-24	1.73	$8.7 \cdot 10^{-7}$	$2.3 \cdot 10^{-6}$	$1.1 \cdot 10^{-5}$	$4.8 \cdot 10^{-5}$	$3.6 \cdot 10^{-4}$

Edge removals

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	0.41	$6.4 \cdot 10^{-7}$	$1.4 \cdot 10^{-6}$	$5.8 \cdot 10^{-6}$	$3.0 \cdot 10^{-5}$	$2.1 \cdot 10^{-4}$
rmat-23	0.85	$7.4 \cdot 10^{-7}$	$1.5 \cdot 10^{-6}$	$6.2 \cdot 10^{-6}$	$3.3 \cdot 10^{-5}$	$2.1 \cdot 10^{-4}$
rmat-24	1.73	$7.4 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$6.7 \cdot 10^{-6}$	$2.7 \cdot 10^{-5}$	$2.2 \cdot 10^{-4}$

The columns "Static" and "Dynamic" report the average time (in seconds) for the static and for the dynamic algorithm, respectively. Edge weights are generated using an exponential distribution.

Table 25. Average Running Time in Seconds of the Static and Dynamic Suitor Algorithms for 100 Batches of $b \in \{1, \ldots, 10^4\}$ Edge Updates on the Random Hyperbolic Networks of Table 2

Edge insertions

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	0.21	$6.4 \cdot 10^{-7}$	$1.8 \cdot 10^{-6}$	$8.8 \cdot 10^{-6}$	$7.7 \cdot 10^{-5}$	$7.5 \cdot 10^{-4}$
hyp-23	0.43	$6.8 \cdot 10^{-7}$	$2.2 \cdot 10^{-6}$	$9.3 \cdot 10^{-6}$	$8.2 \cdot 10^{-5}$	$7.5 \cdot 10^{-4}$
hyp-24	0.85	$6.5 \cdot 10^{-7}$	$2.2 \cdot 10^{-6}$	$9.7 \cdot 10^{-6}$	$8.3 \cdot 10^{-5}$	$7.6 \cdot 10^{-4}$

Edge removals

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	0.22	$5.7 \cdot 10^{-7}$	$1.1 \cdot 10^{-6}$	$5.7 \cdot 10^{-6}$	$4.9 \cdot 10^{-5}$	$5.0 \cdot 10^{-4}$
hyp-23	0.43	$6.6 \cdot 10^{-7}$	$1.3 \cdot 10^{-6}$	$5.9 \cdot 10^{-6}$	$5.0 \cdot 10^{-5}$	$5.1 \cdot 10^{-4}$
hyp-24	0.85	$6.7 \cdot 10^{-7}$	$1.3 \cdot 10^{-6}$	$6.1 \cdot 10^{-6}$	$5.2 \cdot 10^{-5}$	$5.3 \cdot 10^{-4}$

The columns "Static" and "Dynamic" report the average time (in seconds) for the static and for the dynamic algorithm, respectively. Edge weights are generated using a normal distribution.

Table 26. Average Running Time in Seconds of the Static and Dynamic Suitor Algorithms for 100 Batches of $b \in \{1, \ldots, 10^4\}$ Edge Updates on the Random Hyperbolic Networks of Table 2

Edge insertions

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	0.21	$6.3 \cdot 10^{-7}$	$1.9 \cdot 10^{-6}$	$8.8 \cdot 10^{-6}$	$7.7 \cdot 10^{-5}$	$7.5 \cdot 10^{-4}$
hyp-23	0.43	$7.0 \cdot 10^{-7}$	$2.1 \cdot 10^{-6}$	$9.4 \cdot 10^{-6}$	$8.2 \cdot 10^{-5}$	$7.6 \cdot 10^{-4}$
hyp-24	0.85	$6.6 \cdot 10^{-7}$	$2.3 \cdot 10^{-6}$	$9.6 \cdot 10^{-6}$	$8.3 \cdot 10^{-5}$	$7.6 \cdot 10^{-4}$

Edge removals

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	0.22	$5.7 \cdot 10^{-7}$	$1.1 \cdot 10^{-6}$	$5.7 \cdot 10^{-6}$	$4.9 \cdot 10^{-5}$	$5.0 \cdot 10^{-4}$
hyp-23	0.43	$6.5 \cdot 10^{-7}$	$1.2 \cdot 10^{-6}$	$6.1 \cdot 10^{-6}$	$5.0 \cdot 10^{-5}$	$5.1 \cdot 10^{-4}$
hyp-24	0.85	$6.2 \cdot 10^{-7}$	$1.3 \cdot 10^{-6}$	$5.9 \cdot 10^{-6}$	$5.2 \cdot 10^{-5}$	$5.3 \cdot 10^{-4}$

The columns "Static" and "Dynamic" report the average time (in seconds) for the static and for the dynamic algorithm, respectively. Edge weights are generated using an exponential distribution.

C.4 Preprocessing Time

Figure 9 shows the percentage of running time spent by the static Suitor algorithm in sorting the adjacency lists after a batch of edge updates w.r.t. the overall running time.

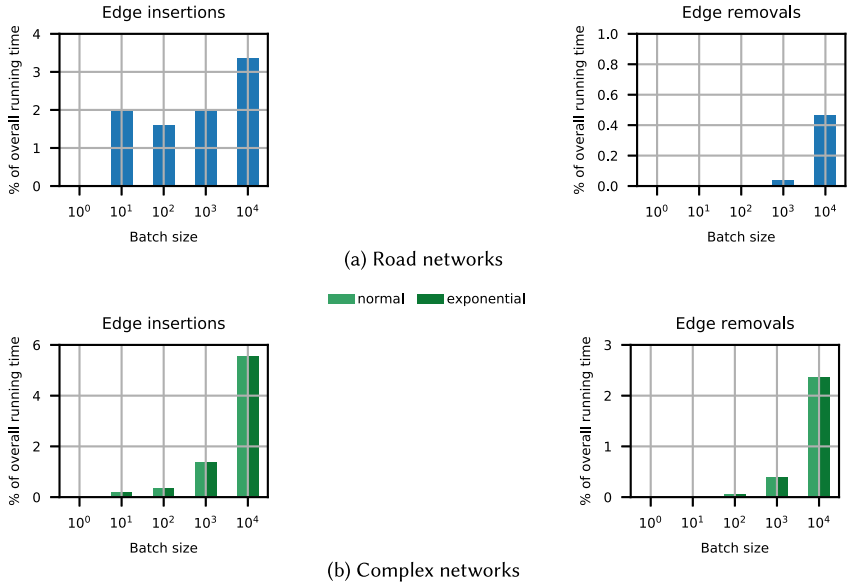


Fig. 9. Percentage of time spent by the static Suitor algorithm for the preprocessing step (i.e., sorting the adjacency lists after a batch of edge updates) w.r.t. the overall running time of the algorithm.

REFERENCES

[1] Abhash Anand, Surender Baswana, Manoj Gupta, and Sandeep Sen. 2012. Maintaining approximate maximum weighted matching in fully dynamic graphs. In *Proceedings of the IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'12) (LIPIcs)*, Deepak D'Souza, Telikepalli Kavitha, and Jaikumar Radhakrishnan (Eds.), Vol. 18. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 257–266. https://doi.org/10.4230/LIPIcs.FSTTCS.2012.257

[2] Eugenio Angriman, Alexander van der Grinten, Moritz von Looz, Henning Meyerhenke, Martin Nöllenburg, Maria Predari, and Charilaos Tzovas. 2019. Guidelines for experimental algorithmics: A case study in network analysis. *Algorithms* 12, 7 (2019), 127.

[3] Moab Arar, Shiri Chechik, Sarel Cohen, Cliff Stein, and David Wajc. 2018. Dynamic matching: Reducing integral algorithms to approximately-maximal fractional algorithms. In *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP'18)*. 7:1–7:16.

[4] David Avis. 1983. A survey of heuristics for the weighted matching problem. *Networks* 13, 4 (1983), 475–493. https://doi.org/10.1002/net.3230130404

[5] Leonid Barenboim and Tzalik Maimon. 2019. Fully dynamic graph algorithms inspired by distributed computing: Deterministic maximal matching and edge coloring in sublinear update-time. *ACM J. Exp. Algor.* 24, 1 (2019), 1.14:1–1.14:24. https://doi.org/10.1145/3338529

[6] Surender Baswana, Manoj Gupta, and Sandeep Sen. 2018. Fully dynamic maximal matching in $O(\log(n))$ update time (Corrected Version). *SIAM J. Comput.* 47, 3 (2018), 617–650.

[7] Surender Baswana, Manoj Gupta, and Sandeep Sen. 2018. Fully dynamic maximal matching in O(log n) update time (Corrected Version). *SIAM J. Comput.* 47, 3 (2018), 617–650. https://doi.org/10.1137/16M1106158

[8] Aaron Bernstein and Cliff Stein. 2016. Faster fully dynamic matchings with small approximation ratios. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 692–711.

[9] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. 2018. Deterministic fully dynamic data structures for vertex cover and matching. *SIAM J. Comput.* 47, 3 (2018), 859–887.

[10] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. 2016. New deterministic approximation algorithms for fully dynamic matching. In *Proceedings of the 48th Annual ACM Symposium on Theory of Computing*. 398–411.

[11] Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. 2013. Efficient parallel and external matching. In *Proceedings of the 19th International Conference on Parallel Processing (Euro-Par'13) (Lecture Notes in Computer Science)*, Felix Wolf, Bernd Mohr, and Dieter an Mey (Eds.), Vol. 8097. Springer, 659–670. https://doi.org/10.1007/978-3-642-40047-6_66

[12] Rob H. Bisseling. 2020. *Parallel Scientific Computation: A Structured Approach Using BSP*. Oxford University Press, New York, NY.

[13] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the SIAM International Conference on Data Mining*. SIAM, 442–446.

[14] Moses Charikar and Shay Solomon. 2017. Fully dynamic almost-maximal matching: Breaking the polynomial barrier for worst-case time bounds. Retrieved from http://arxiv.org/abs/1711.06883.

[15] Michael Crouch and Daniel M. Stubbs. 2014. Improved streaming algorithms for weighted matching, via unweighted matching. In *Proceedings of the Conference on Approximation, Randomization, and Combinatorial Optimization: Algorithms and Techniques (APPROX/RANDOM'14)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[16] Doratha E. Drake and Stefan Hougardy. 2003. Linear time local improvements for weighted matchings in graphs. In *Proceedings of the International Workshop on Experimental and Efficient Algorithms*. Springer, 107–119.

[17] Doratha E. Drake and Stefan Hougardy. 2003. A simple approximation algorithm for the weighted matching problem. *Inform. Process. Lett.* 85, 4 (2003), 211–213.

[18] Andre Droschinsky, Petra Mutzel, and Erik Thordsen. 2020. Shrinking trees not blossoms: A recursive maximum matching approach. In *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX'20)*, Guy E. Blelloch and Irene Finocchi (Eds.). SIAM, 146–160. https://doi.org/10.1137/1.9781611976007.12

[19] Ran Duan, Seth Pettie, and Hsin-Hao Su. 2018. Scaling algorithms for weighted matching in general graphs. *ACM Trans. Algor.* 14, 1 (2018), 8:1–8:35. https://doi.org/10.1145/3155301

[20] Jack Edmonds. 1965. Paths, trees, and flowers. *Canadian J. Math.* 17, 3 (1965), 449–467.

[21] Leah Epstein, Asaf Levin, Julián Mestre, and Danny Segev. 2011. Improved approximation guarantees for weighted matching in the semi-streaming model. *SIAM J. Discrete Math.* 25, 3 (2011), 1251–1265.

[22] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. 2005. On graph problems in a semi-streaming model. *Theor. Comput. Sci.* 348, 2–3 (2005), 207–216.

[23] Harold N. Gabow. 1990. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, David S. Johnson (Ed.). SIAM, 434–443. http://dl.acm.org/citation.cfm?id=320176.320229.

[24] Zvi Galil. 1986. Efficient algorithms for finding maximum matching in graphs. *ACM Comput. Surv.* 18, 1 (1986), 23–38. https://doi.org/10.1145/6462.6502

[25] Mohsen Ghaffari and David Wajc. 2019. Simplified and space-optimal semi-streaming $(2+\varepsilon)$-approximate matching. In *Proceedings of the Symposium on Simplicity in Algorithms*, Vol. 69.

[26] Andrew V. Goldberg and Alexander V. Karzanov. 2004. Maximum skew-symmetric flows and matchings. *Math. Program.* 100, 3 (2004), 537–568. https://doi.org/10.1007/s10107-004-0505-z

[27] Fabrizio Grandoni, Stefano Leonardi, Piotr Sankowski, Chris Schwiegelshohn, and Shay Solomon. 2019. $(1+\varepsilon)$-approximate incremental matching in constant deterministic amortized time. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'19)*, Timothy M. Chan (Ed.). SIAM, 1886–1898. https://doi.org/10.1137/1.9781611975482.114

[28] Manoj Gupta and Richard Peng. 2013. Fully dynamic (1+ e)-approximate matchings. In *Proceedings of the 54th Annual IEEE Symposium on Foundations of Computer Science (FOCS'13)*. IEEE Computer Society, 548–557. https://doi.org/10.1109/FOCS.2013.65

[29] Monika Henzinger, Shahbaz Khan, Richard Paul, and Christian Schulz. 2020. Dynamic matching algorithms in practice. In *Proceedings of the 28th Annual European Symposium on Algorithms (ESA'20) (LIPIcs)*, Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders (Eds.), Vol. 173. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 58:1–58:20. https://doi.org/10.4230/LIPIcs.ESA.2020.58

[30] Jaap-Henk Hoepman. 2004. Simple distributed weighted matchings. Retrieved from http://arxiv.org/abs/cs.DC/0410047.

[31] Zoran Ivković and Errol L. Lloyd. 1993. Fully dynamic maintenance of vertex cover. In *Proceedings of the 19th International Workshop Graph-Theoretic Concepts in Computer Science (LNCS)*, Vol. 790. Springer, 99–111.

[32] Manas Jyoti Kashyop and N. S. Narayanaswamy. 2020. Lazy or eager dynamic matching may not be fast. *Inform. Process. Lett.* 162 (2020), 105982. https://doi.org/10.1016/j.ipl.2020.105982

[33] Arif M. Khan, Alex Pothen, Md. Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Fredrik Manne, Mahantesh Halappanavar, and Pradeep Dubey. 2016. Efficient approximation algorithms for weighted b-matching. *SIAM J. Sci. Comput.* 38, 5 (2016). https://doi.org/10.1137/15M1026304

[34] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Scalable SIMD-efficient graph processing on GPUs. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT'15)*. 39–50.

[35] Viatcheslav Korenwein, André Nichterlein, Rolf Niedermeier, and Philipp Zschoche. 2018. Data reduction for maximum matching on real-world graphs: Theory and experiments. In *Proceedings of the 26th Annual European Symposium on Algorithms (ESA'18) (LIPIcs)*, Yossi Azar, Hannah Bast, and Grzegorz Herman (Eds.), Vol. 112. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 53:1–53:13. https://doi.org/10.4230/LIPIcs.ESA.2018.53

[36] Jérôme Kunegis. 2013. KONECT: The Koblenz network collection. In *Proceedings of the 22nd International World Wide Web Conference (WWW'13)*, Leslie Carr, Alberto H. F. Laender, Bernadette Farias Lóscio, Irwin King, Marcus Fontoura, Denny Vrandecic, Lora Aroyo, José Palazzo M. de Oliveira, Fernanda Lima, and Erik Wilde (Eds.). International World Wide Web Conferences Steering Committee/ACM, 1343–1350. https://doi.org/10.1145/2487788.2488173

[37] Eugene L. Lawler. 2001. *Combinatorial Optimization: Networks and Matroids*. Courier Corporation.

[38] László Lovász and Michael D. Plummer. 2009. *Matching Theory*. Vol. 367. American Mathematical Soc.

[39] Fredrik Manne and Rob H. Bisseling. 2007. A parallel approximation algorithm for the weighted maximum matching problem. In *Proceedings of the International Conference on Parallel Processing and Applied Mathematics*. Springer, 708–717.

[40] Fredrik Manne and Mahantesh Halappanavar. 2014. New effective multithreaded matching algorithms. In *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 519–528.

[41] Jens Maue and Peter Sanders. 2007. Engineering algorithms for approximate weighted matching. In *Proceedings of the International Workshop on Experimental and Efficient Algorithms*. Springer, 242–255.

[42] Andrew McGregor. 2005. Finding graph matchings in data streams. In *Proceedings of the Conference on Approximation, Randomization and Combinatorial Optimization: Algorithms and Techniques*. Springer, 170–181.

[43] Aranyak Mehta, Amin Saberi, Umesh V. Vazirani, and Vijay V. Vazirani. 2005. AdWords and generalized on-line matching. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*. IEEE Computer Society, 264–273. https://doi.org/10.1109/SFCS.2005.12

[44] Silvio Micali and Vijay V. Vazirani. 1980. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *Proceedings of the 21st Symposium on Foundations of Computer Science*. IEEE Computer Society, 17–27. https://doi.org/10.1109/SFCS.1980.12

[45] Marcin Mucha and Piotr Sankowski. 2006. Maximum matchings in planar graphs via gaussian elimination. *Algorithmica* 45, 1 (2006), 3–20. https://doi.org/10.1007/s00453-005-1187-5

[46] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. 2010. Introducing the graph 500. *Cray Users Group* 19 (2010), 45–74.

[47] Ofer Neiman and Shay Solomon. 2015. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Trans. Algor.* 12, 1 (2015), 1–15.

[48] Krzysztof Onak and Ronitt Rubinfeld. 2010. Maintaining a large matching and a small vertex cover. In *Proceedings of the 42nd ACM Symposium on Theory of Computing*. 457–464.

[49] OpenStreetMap contributors. 2017. Planet dump. Retrieved from https://planet.osm.org; https://www.openstreetmap.org.

[50] Ami Paz and Gregory Schwartzman. 2017. A $(2 + \varepsilon)$-approximation for maximum weight matching in the semi-streaming model. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2153–2161.

[51] Daniël Maria Pelt and Rob H. Bisseling. 2015. An exact algorithm for sparse matrix bipartitioning. *J. Parallel Distrib. Comput.* 85 (2015), 79–90. https://doi.org/10.1016/j.jpdc.2015.06.005

[52] Robert Preis. 1999. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science (STACS'99) (Lecture Notes in Computer Science)*, Christoph Meinel and Sophie Tison (Eds.), Vol. 1563. Springer, 259–269. https://doi.org/10.1007/3-540-49116-3_24

[53] Piotr Sankowski. 2007. Faster dynamic matchings and vertex connectivity. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'07)*. 118–126.

[54] Piotr Sankowski. 2009. Maximum weight bipartite matching in matrix multiplication time. *Theor. Comput. Sci.* 410, 44 (2009), 4480–4488. https://doi.org/10.1016/j.tcs.2009.07.028

[55] Shay Solomon. 2016. Fully dynamic maximal matching in constant update time. In *Proceedings of the IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS'16)*. IEEE, 325–334.

[56] Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. 2016. NetworKit: A tool suite for large-scale complex network analysis. *Netw. Sci.* 4, 4 (2016), 508–530. https://doi.org/10.1017/nws.2016.20

[57] Daniel Stubbs and Virginia Vassilevska Williams. 2017. Metatheorems for dynamic weighted matching. In *Proceedings of the 8th Innovations in Theoretical Computer Science Conference (ITCS'17) (LIPIcs)*, Christos H. Papadimitriou (Ed.), Vol. 67. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 58:1–58:14. https://doi.org/10.4230/LIPIcs.ITCS.2017.58

[58] Moritz von Looz, Mustafa Safa Özdayi, Sören Laue, and Henning Meyerhenke. 2016. Generating massive complex networks with hyperbolic geometry faster in practice. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC'16)*. IEEE, 1–6. https://doi.org/10.1109/HPEC.2016.7761644

Received April 2021; revised February 2022; accepted March 2022