Bachelorarbeit

ENGINEERING ALGORITHMS FOR FULLY DYNAMIC MAXIMAL MATCHING

Verfasser

Richard Paul

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, 2018

Studienkennzahl lt. Studienblatt:	A 033 521
Fachrichtung:	Informatik - Scientific Computing
Betreuerin:	Prof. Dr. Monika Henzinger
Co-Betreuer	Dr. Christian Schulz
	Shahbaz Khan, PhD

Abstract

We present our results of an extensive experimental evaluation of different fully dynamic maximal matching algorithms using real-world fully dynamic graphs and fully dynamic graphs created from real-world static graphs. We examine the algorithms presented from Baswana, Gupta and Sen [17], which performs edge updates in $O(\sqrt{n})$ time and maintains a 2-approximate maximum matching, from Neiman and Solomon [35], which takes $O(\sqrt{n+m})$ time to maintain a 3/2-approximate maximum matching, as well as a naive, greedy algorithm and some random walk-based algorithms, which take up between $O(n)$ and $O(1/\epsilon)$ time and do mostly maintain 2-approximate maximum matchings.

As a result, the naive algorithm computes edge updates averagely fastest, whereas the algorithm by Neiman and Solomon achieves the averagely best matching sizes. We presume, that the results regarding update time arise from the rather small problem sizes. By improving some of our algorithms to detect and resolve augmenting paths during edge insertion we are able to achieve similarily good results as Neiman and Solomon

Further we conclude that random walks can be a mean to achieve matching sizes above the guaranteed 2-approximate maximum matching, however at the cost of increased update time. We also show, that small improvements can significantly improve the performance of random walk based algorithms. One particular algorithm, which does not guarantee any particular lower bound of the matching size, does turn out to perform significantly worse than the other algorithms.

Contents

1 Introduction

1.1 Motivation

In graph theory a matching of a graph $G = (V, E)$, with $n = |V|$ and $m = |E|$, is a subset of edges $M \subseteq E$ where every vertex has maximally one edge incident to it. Finding an arbitrary matching for a graph is a trivial task, since a single edge already qualifies as a matching. However finding matchings that satisfy more specific properties is not as trivial. The most well-researched matching problem is the search for maximum cardinality matchings in a graph [17].

Computing maximum matchings in a static graph is a well researched topic with algorithmic solutions developed in the 1960's by Jack Edmonds [23] and later improvements from Micali and Vazirani in the 1980's [31]. Many graphs derived from real-world applications however tend to be dynamic graphs [17], meaning that over time they undergo changes like loss and creation of new edges. The most naive way to solve this dynamic maximum matching problem is to recompute the matching every time an update on the graph has occured using a static algorithm. This however is obviously a wasteful approach regarding the running time. The aim therefore is to find algorithms, that can maintain a maximal matching based on the previously computed matchings and which can compute an update in way less time than the best static algorithm would take. Until now the algorithm providing the best upper bound on time complexity to compute the exact dynamic maximum matching size exactly is a randomized algorithm by Sankowski [41] which performs an update in $O(n^{1.495})$ time. Therefore it has become a relevant topic developing algorithms which try to maintain an approximate maximum matching in a dynamic graph [17], [28], [35], [42], [20], [25]. This problem is further referred to as dynamic maximal matching problem. The computation of dynamic matchings is often needed in combinatorial optimization problems, especially from operations research or market design [14], [16].

Traditionally, the development of algorithms is done in a theoretical manner, resulting in mathematically proveable asymptotic bounds of complexity. For some algorithms however, observing this asymptotic behaviour may be only possible for unrealistically large problem sizes, as complexity ignores constant factors. [34], [38]. A modern approach is known and described as *algorithm engineering* [38] or *experimental algorithmics* [34], which tries to evaluate algorithms in an experimental manner using real-world data, models and assumptions in order to gain more practical knowledge about performance and behaviour of algorithms. The insight achieved using this approach may then lead to further refinement[38].

1.2 Contribution and Methods

In this work, we describe and examine different dynamic maximal matching algorithms, which we implemented and evaluated experimentally. The experiments are performed on fully dynamic real-world graphs and fully dynamic graphs constructed from real-world graphs, taken from the Koblenz network collection [29]. We compare the experimental results of a naive algorithm, two more sophisticated algorithms from Baswana, Gupta and Sen [17] and Neiman and Solomon [35] and different random-walk based algorithms. As a result, we identify the naive algorithm, which has runtime complexity $O(n)$, as the fastest algorithm on our test data. We identify Neiman-Solomn as the algorithm to compute the largest matchings on our test data and approve experimentally, that eliminating all augmenting paths of length 3 does have a significant impact on the matching size. Further we show that sufficiently large random walks can be a mean to detect augmenting paths and to obtain relatively good results, comparable to the results achieved by the Neiman-Solomon algorithm, although this happens at the cost of an increased update time. By examining the experimental results of our algorithms in detail, we try to give valuable insight in order to help to improve different solutions for the dynamic maximal matching problem.

1.3 Structure

This work is structured as follows. After clarifying basic terms and concepts in Section 2, we discuss some related work in Section 3. In Section 4 we present the different dynamic algorithms, that we examine throughout this paper. The main part of this paper is in Section 5 and especially in Section 5.3, where we present the results of our experimental evaluation of the previously introduced and discussed algorithms. Our conclusion is presented in Section 6.

2 Preliminaries

2.1 Matchings

In graph theory a *matching* M of a simple graph $G = (V, E)$ is defined as a subset of edges $M \subseteq E$ where no vertex from V is incident to more than one edge from M. In other words, if we construct the graph $G' = (V, M)$, then all vertices in V have either degree 0 (we call them *free* or *unmatched*) or 1 (we call them *unfree* or *matched*). For every vertex u with degree of 1, we call the vertex v at the other end of the incident edge the *mate* of u, which we denote as $\text{mate}(u) = v$. For an unmatched vertex u we define $\text{mate}(u) = \perp$. Note that we call an graph without self-loops and without parallel edges a *simple graph.* The *cardinality* or *size* of a matching is simply the cardinality of the edge subset M.

An *augmenting path* is defined as a cycle-free path in the graph G, that starts and ends on a *free* vertex and where edges from M alternate with edges from $E \setminus M$. The *trivial augmenting path* is a single edge, that has both its endpoints unmatched. Throughout this paper, we call such an edge a *free* edge. If we take an augmenting path and resolve it by matching every unmatched edge and unmatching every matched edge, we increase the matching cardinality by one.

Further we call a matching *maximal*, if there is no edge in E, that we could add to M without corrupting the previously introduced matching condition. We call a matching a *maximum matching*, if it is maximal in size among all other valid matchings for some particular graph. The cardinality of a maximum matching is called the *matching number* $\nu(G)$. As proven in [19] any maximal matching without *augmenting paths* is a maximum matching.

We call an *a-approximate* maximum matching a matching, that contains at least $\frac{\nu(G)}{a}$ number of edges. Hopcroft and Karp [27] showed that any maximal matching with no augmenting paths of length at most $2k - 3$ is a $(k/(k-1))$-approximate matching. By setting $2k - 3 = 1 \Rightarrow k = 2$ we can show that any maximal matching is therefore at least a 2-approximate maximum matching.

2.2 Dynamic Graphs and Sequences

Unlike the classical maximal matching problem, which tries to compute a matching for static graphs, we focus in this work on *dynamic graphs*, where the number of vertices is fixed, but edges can appear and disappear. The algorithms examined do not recompute the matching for every instance of the graph at a given moment of time i, but try to maintain a maximal matching over time. We call a *sequence* S a sequence of edge updates, which can be either insertions or deletions. A sequence S of length k is a k-tupel of 3-tupels (m, u, v), where m denotes the input mode (0 for deletion, 1 for insertion), and u and v denote the edge. Note that G is undirected and $(u, v) = (v, u)$ holds. We denote a single sequence step for a given point in time i as S_i.

Furthermore let $\mathcal{G} = \{G_0, G_1, \ldots, G_k, G_{k+1}\}$ be a dynamic graph, where G_i denotes the graph constructed from applying the sequence step S_{i-1} on G_{i-1}.

Note here that G_0 is the graph without edges, therefore we obtain $k+1$ static graph instances from a sequence of length k. Our definition of dynamic graphs does consider only the edge set to be dynamic, whereas the set of vertices remains the same for all static instances $G_i \in \mathcal{G}$. Therefore we simply write V or $\mathcal{V}$ to denote the node set of the dynamic graph $\mathcal{G}$ as well as every static instance G_i. On the other hand the edge set $\mathcal{E}$ of the dynamic graph $\mathcal{G}$ is defined as $\mathcal{E} = \{E_0, E_1, \ldots, E_k, E_{k+1}\}$, where E_i is the edge set of the static graph instance G_i. Also let $\mathcal{M} = \{M_0, M_1, \ldots, M_i, \ldots, M_k, M_{k+1}\}$ be the set of matchings for the according graph G_i for some point in time i.

According to [22] we can handle dynamic graph problems in three different settings depending on the kind of updates allowed. As stated in [21], changes happen more frequently on edges than on vertices, hence usually edge updates are considered in dynamic graph problems. A setting where only edge insertions are allowed is called an *incremental* setting, whereas a problem where only edge deletions are allowed is called *decremental*. If both updates are allowed we call the setting *fully dynamic*.

In this paper we address the problem of maintaining a maximal matching in a fully dynamic graph, therefore a graph, where edges can be added and deleted from the graph. Accordingly all algorithms presented can handle edge insertions as well as edge deletions.

3 Related Work

Throughout the last decades the interest in computing structures in fully dynamic graphs has grown and many different problems like maintaining minimum spanning trees or connectivity information are well-researched, providing algorithms which solve the problems in polylogarithmic time [22], [26]. Regarding the dynamic maximum matching problem, the best algorithm to obtain the size of a maximum matching in a fully dynamic setting is a randomized algorithm by Sankowski [41]. A trivial solution in order to obtain a maximum matching in a fully dynamic undirected graph is known to require $O(m) \subset O(n^2)$ update time [24]. As no better solutions have been discovered so far, interest has grown in computing approximated maximum matchings. Hopcroft and Karp [27] showed, that any matching without augmenting paths of length $2k-3$ is a $(k/(k-1))$-approximate maximum matching. Hence, by resolving all trivial augmenting paths of length 1, we receive a maximal matching, which is a 2-approximate maximum matching. This circumstance has been a key concept when developing fully dynamic approximate maximum matching algorithms [17], [28], [35], [42].

The first algorithm Baswana, Gupta and Sen present in [17] maintains a 2-approximate maximum matching in amortized $O(\sqrt{n})$, however they improve upon their own result and present a further algorithm, which maintains a 2-approximate maximum matching in $O(\log n)$ time, both however are randomized. Solomon [42] presented a randomized algorithm which improves even further upon the update time by Baswana, Gupta and Sen and maintains a 2-approximate maximum matching in constant amortized update time $O(1)$. In contrary Neiman and Solomon [35] presented a deterministic algorithm, which maintains a 3/2-approximate maximum matching by guaranteeing, that there exists no augmenting path of length 3 throughout the matching and which takes $O(\sqrt{m+n})$ time. Gupta and Peng [25] improved upon this with a deterministic algorithm, which maintains a $(1+\epsilon)$-approximate maximum matching in $O(\sqrt{m}\epsilon^{-2})$, Bernstein and Stein [20] presented a deterministic algorithm, which maintains a $(3/2+\epsilon)$-approximate maximum matching in only $O(m^{1/4}\epsilon^{-2.5})$ time and claim to be the first ones to achieve an approximation factor above 2 in $O(\sqrt{n})$ time, since $m^{1/4} \in O(\sqrt{n})$. Kashyop and Narayanaswamy [28] just recently published a randomized algorithm, which combines the approaches of Neiman and Solomon [35] and Baswana, Gupta and Sen [17] and by doing so achieve to maintain a 3/2-approximate maximum matching in $O(\sqrt{n})$ amortized update time.

Despite this variety of different algorithms, to the best of our knowledge, there has been no effort made so far, to implement and evaluate these algorithms in an experimental manner, using algorithm engineering techniques [38]. Although there exist quite numerous randomized algorithms for the dynamic maximal matching problem, we do not know about any attempts, to use random walks as a mean to improve matching quality.

4 Algorithms

In this section we give an overview over the implemented and revised fully dynamic algorithms. We begin with the most naive algorithm, which is of complexity $O(n)$. Further we revise a random walk algorithm with different variations, that range from complexity $O(1/\epsilon)$ to $O(n)$ and finally two algorithms by Baswana, Gupta and Sen and Neiman and Solomon. All these algorithms work on fully dynamic graphs and assume that the graph is empty at $i = 0$ and that for every sequence step S_i only one insertion or deletion is processed.

4.1 A Naive Approach

In this section we describe a naive algorithm to maintain a maximal matching in a dynamic graph in $O(n)$ time.

4.1.1 Edge Insertion

The most naive approach to handle an edge insertion is to check, if both endpoints are free and, if so, add the edge to the matching, otherwise simply ignore it. For a sequence, that consists of insertions only and is therefore pure incremental, this approach maintains a maximal matching.

Lemma 4.1. *Given the graph $G_i = (V, E)$ and a matching M_i, that is maximal with respect to G_i, the naive algorithm will maintain a maximal matching M_{i+1} for G_{i+1} when adding an arbitrary edge (u, v), $u, v \in V$.*

Proof. Given a maximal matching M_i on a graph $G_i = (V, E_i)$. In such a graph, every free vertex u has all its neighbours matched, as otherwise M_i would not be maximal. Therefore the only way to create a new unmatched edge, which has both its endpoints free, is to insert an edge (u, v) to $G_{i+1} = (V, E_i \cup \{(u, v)\})$, where both vertices u, v are unmatched. The naive insertion algorithm described will match any newly inserted edge (w, x), if both its endpoints are free, hence (u, v) will be added to the matching $M_{i+1} = M_i \cup \{(u, v)\}$, which is then maximal on G_{i+1}. Since $M_0 = \{\}$ is a maximal matching for $G_0 = (V, \{\})$, this algorithm will maintain a maximal matching for an arbitrary long incremental edge update sequence performed on an initially empty graph. □

4.1.2 Edge Deletion

When removing an edge (u, v), we also have to distinguish two cases: An edge, that is to be removed, can either be matched or unmatched. Removing an unmatched edge has neither an effect on the matching size nor does it change the state of an incident vertex, therefore we can simply remove it from the graph. However removing a matched edge does leave the two incident and previously matched vertices free as well as it does decrease the size of the matching by one. These vertices are of special intereset, since, if at least one of them has unmatched neighbours, freeing them has created at least one edge, that could be added to the matching without corrupting the matching condition. Therefore the remaining matching would not be maximal anymore.

In order to fix this issue, we have to check the surroundings of the freed vertices u and v. Checking if there exists a free vertex by scanning through all neighbours of a vertex $w \in u, v$ and if applicable match it, is the most simple way to at least assure that our matching remains maximal. This approach takes $O(\deg(u) + \deg(v))$ time, where $\deg(u)$ is the vertex degree of a vertex u and the vertices u, v are the endpoints to the removed edge. The vertex degree in a simple graph is at most $n - 1$, where $n = |V|$. Therefore runtime complexity is

$$O(\deg(u) + \deg(v)) \subset O(n).$$

4.1.3 Complexity and Approximation

As already mentioned, computing an update for the maintained matching costs $O(1)$, if we handle an edge insertion, and $O(n)$, if we handle an edge deletion. The outlined naive approach maintains a maximal matching in a deterministic way. A maximal matching is known to be a 2-approximate maximum matching, which means that it contains at least 1/2 the amount of edges contained in any maximum matching.

4.2 Random Walk Methods

As a simple heuristic we extended the naive algorithm by performing random walks whenever an edge is removed in order to detect augmenting paths. We implemented different versions of this random walk algorithm which we will further explain as follows.

Our random walk algorithms handle edge insertions exactly like the naive algorithm does. As shown in the previous Section 4.1.1 this approach maintains a maximal matching in $O(1)$ time.

4.2.1 Edge Deletion and Random Walk

As mentioned in the naive algorithm, deleting a matched edge (u, v) leaves the two endpoints u and v free. This can give rise to unmatched edges with free endpoints iff there exists a free neighbour for at least one vertex from $\{u, v\}$. If such an unmatched edge exists, that could be added to the matching M after the deletion of (u, v) without conflicting with the matching condition, the matching M is not maximal. We already explained how to maintain the matching maximal in $O(n)$ time.

If the freed vertices have no free neighbours and the matching before the edge deletion was maximal, then the matching remains maximal. However a free vertex may be starting point to an augmenting path of arbitrary length. Finding such augmenting paths of arbitrary length k in a naive manner costs $O(n^k)$ time and is therefore not a trivial task. We now present our random walk approach and four variants which are used as an heuristic approach to find augmenting paths and increase the matching size in order to achieve a better matching quality than the naive algorithm does.

Random Walk: As an heuristic approach we do a random walk where we assume the walked path to be an augmenting path. We will finish our walk after a maximum of $k = 1/\epsilon$ steps latest, where ϵ is an algorithm parameter. Starting at a free vertex u, which at $k = 0$ is one of the vertices freed from the deletion of the matched edge, we randomly choose a neighbour w of u. If this neighbour is free, then we match the edge (u, w) and our random walk has finished. Otherwise if w is matched, then we unmatch $(w, \text{mate}(w))$ and match (u, w). Note that $u \neq \text{mate}(w)$ since u is free in the beginning and therefore $\text{mate}(u) = \perp$, but $\text{mate}(\text{mate}(w)) = w$ and $w \neq \perp$. Afterwards $\text{mate}(w)$ remains free, therefore we continue our random walk, but starting the next step from $\text{mate}(w)$. Since the decision on where to continue the random walk is done randomly, we do not have to scan anyhow through the neighbours of a vertex. Note that we provide the necessary data structure to retrieve a single neighbour from a vertex in $O(1)$ time. A step in our random walk therefore costs $O(1)$ time, a complete update $O(k) = O(1/\epsilon)$ time.

A closer look at the presented random walk algorithm reveals, that this algorithm cannot guarantee some lower bound for the matching quality. Consider the following scenario.

Example 4.2.1. *There exists a free vertex x which has a neighbour w which is matched with v and which is neighbour of u. Our random walk is at its penultimate step, which starts at vertex u. Note that the vertices u, v, w, x form an augmenting path of length 3. Our random walk now chooses v randomly from its neighbours. Since v is matched with w, we unmatch (v, w) and match (u, v). Our random walk has now performed k steps, it simply breaks out from the recursion. The matching calculated from this update is not maximal, since the two adjacent vertices w and x remain free and the edge (w, x) unmatched.*

max-random-walk: The presented issue leads us to our first variation of the random walk algorithm. Instead of letting the random walk just suddenly end, we settle the last vertex z naively by scanning through its neighbours for a free vertex. This guarantees, that any matchable edge incident to z will be matched, but takes $O(\deg(z))$ time. As we already examined in Section 4.1.2 this is theoretically of cost $O(n)$.

$\sqrt{m}$-random-walk: As a second variant we determined the size of the random walk $k = 1/\epsilon$ in dependence of the number of edges present in the graph at time of the update. The maximum length of the random walk is set to $k = \sqrt{m}$, with $m = |E|$. Further this variant does also settle the last node before breaking out from the recursion.

low-degree-settle: A third variant checks the degree of every vertex u before continuing its random walk of length $k = 1/\epsilon$. If $\deg(u) < 1/\epsilon$, then the algorithms tries to settle the vertex naively by scanning through the neighbours of u. If settling the vertex was successful, then the we break out from the recursion. Otherwise we keep performing the random walk. The last vertex of our random walk is settled naively as in variant 1, regardless of its degree.

extended-naive: Our fourth variant basically extends the naive algorithm by a conditional random walk. When a matched edge gets removed, the algorithm tries to settle both endpoints naively just as the naive algorithm does. This assures, that the matching remains maximal. However if a vertex remains free after scanning through all its neighbours for a new mate, then it could be the starting point to an augmenting path. Therefore we perform a random walk of length $k = 1/\epsilon$, to possibly detect and augment it. As before, we perform a naive settle at the last step of our random walk in order to guarantee a maximal matching.

4.2.2 Complexity and Approximation

We outlined in the previous section, that our basic variant of the random walk algorithm does not guarantee any particular matching quality. The probability of the described scenario to actually occur seems quite small, wherefore it will be interesting, how the algorithm will perform in terms of matching size in the experiments. This lack of a performance guarantee is compensated by the runtime complexity of $O(1/\epsilon)$.

It is the declared goal of the **max-random-walk** algorithm to guarantee that our matching remains maximal. As stated previously in Section 2 any maximal matching is a 2-approximate maximum matching. This guarantee happens at cost of runtime complexity, which is increased to $O(1/\epsilon + n)$.

Since the **$\sqrt{m}$-random-walk** of the random walk algorithm is actually just a differently parametrized variant, that deduces the length of the random walk k from the number of edges m present in the graph as $k = \sqrt{m}$, runtime complexity is $O(\sqrt{m} + n)$. Settling the end vertex of the random walk in a naive manner assures, that our matching is maximal and therefore a 2-approximate maximum matching.

Regarding update time of the **low-degree-settle** algorithm, querying the degree of a vertex can be accomplished in $O(1)$ time. Performing a naive settle on a vertex is of $O(\deg(u))$ cost. Therefore it costs $O(1/\epsilon)$ time, if we try to perform a naive settle on a vertex during our random walk. Yet if we do not find a vertex during the random walk that we settle, settling the last vertex at the end of the random walk costs $O(n)$ time. Total update time therefore costs $O(n + 1/\epsilon)$ time. We suspect that probability of actually having to perform all $k = 1/\epsilon$ steps to be low. Performing naive settle on the last step of our random walk assures, that our matching is maximal after any update.

Since the **extended-naive** algorithm is basically the naive algorithm with the extension of performing a conditional random walk, when no mate was found by scanning naively through the neighbours of a vertex u, we have runtime complexity $O(n + 1/\epsilon)$. As in the previous variants, we perform a naive settle on the last vertex of the random walk, wherefore this algorithm guarantees the matching to be maximal after any update. Time complexity remains unchanged, since $O(2n + 1/\epsilon) \in O(n + 1/\epsilon)$.

4.3 Randomized algorithm by Baswana, Gupta and Sen

Baswana, Gupta and Sen presented an randomized algorithm in [17], that maintains a 2-approximate maximal matching in a dynamic graph in *amortized* $O(\sqrt{n})$ time with high probability. This algorithm shows how to improve the runtime of $O(\deg(u) + \deg(v))$ for the earlier mentioned naive approach, where we scan all neighbours of a freed vertex in order to find an appropriate new mate. This improvement is achieved by introducing a concept of *ownership* for edges and maintaining a partition of the set of vertices into two disjunct sets based on the concept of ownership.

4.3.1 Levels and Ownership of Edges

Baswana, Gupta and Sen partition the set of vertices into two levels 0 and 1 based upon the number of edges *owned* by a vertex. An edge is always owned by at least one of its endpoints. If both endpoints are at level 0, both own the edge. If only one endpoint is at level 1, this endpoint owns the edge. If both endpoints are at level 1, then simply the edge mentioned first will own the edge. A new edge (u, v) with $\mathit{level}(u) = \mathit{level}(v) = 1$ inserted will therefore be owned by the vertex u. $\mathcal{O}_u$ denotes the set of edges owned by a vertex u. Derived from the partition of the vertices into two levels, Baswana, Gupta and Sen also define the level of an edge (u, v) as $\mathit{level}(u, v) = \max(\mathit{level}(u), \mathit{level}(v))$.

Furthermore the following invariants are introduced:

Invariant 1. *Every vertex at level 1 is matched. Every free vertex at level 0 has all its neighbours matched.*

Invariant 2. *Every vertex at level 0 owns less then $\sqrt{n}$ edges at any moment of time.*

Invariant 3. *Both endpoints of every matched edge are at the same level.*

4.3.2 Edge Insertion

If an edge (u, v) gets inserted and at least one endpoint is at level 1, the edge can not be naively added to the matching as because of Invariant 1 every vertex at level 1 is matched. In this case, in order to maintain the data structures needed to represent ownership of edges, we add the edge (u, v) to $\mathcal{O}_u$ if $\mathit{level}(u) = 1$ or to $\mathcal{O}_v$ if $\mathit{level}(v) = 1$. Note, that if both vertices are at level 1, the edge will be assigned to u only.

Now if no endpoint of an ede (u, v) is at level 1, then the edge will be owned by both endpoints. If both endpoints are free the edge will further be added to the matching. Note that the behaviour of the algorithm matches the behaviour of the naive algorithm so far. The matching maintained is therefore maximal. Adding the edge (u, v) to the sets $\mathcal{O}_u$ and $\mathcal{O}_v$ obviously increases the number of

```
1: procedure Random-Settle($u$)
2:     $y \leftarrow$ randomly chosen element from $\mathcal{O}_u$
3:     for all $x \in \mathcal{O}_y$ do
4:         remove $y$ from $\mathcal{O}_y$
5:     end for
6:     $z \leftarrow$ mate($y$)
7:     if $z \neq$ null then
8:         $M \leftarrow M \setminus \{(y, z)\}$
9:     end if
10:    $M \leftarrow M \cup \{(u, y)\}$
11:    level($u$) $\leftarrow 1$, level($y$) $\leftarrow 1$
12:    return $z$
13: end procedure
```

Figure 1: *Pseudocode for procedure* Random-Settle

edges owned by u and v. This may contradict Invariant 2. If at least one set $\mathcal{O}_u$ or $\mathcal{O}_v$ exceeds the threshold of $\sqrt{n}$ in size, the vertex $w \in u, v$ with the higher number of owned edges will be *repaired*. Repairing a vertex w is done by calling the procedure Random-Settle on w. Since $|\mathcal{O}_w| > \sqrt{n}$ the vertex w will be risen to level 1 at the end of the procedure Random-Settle. Hence we have to remove all edges (w, x), more precisely the vertex w, from $\mathcal{O}_x$ for all vertices $x \in \mathcal{O}_u$, since an edge can not be owned by two vertices at level 1. Note that for any vertex u the set $\mathcal{O}_u$ holds the endpoints for all edges owned by u.

The procedure Random-Settle (see Figure 1) performed on a vertex u picks a mate y for u randomly from the set of owned edges $\mathcal{O}_u$. In order to maintain Invariant 3 y will be risen to level 1 at the end of the procedure. Therefore we have to do the same we did with u earlier, namely remove the vertex y from $\mathcal{O}_x$ for all vertices $x \in \mathcal{O}_y$. We do so because we stated earlier, that an edge at level 1 is always owned by only one of its endpoints. Further if y was matched, we unmatch $(y, \text{mate}(y))$, so that we can then match (u, y). Afterwards we set the level of both vertices u and y to 1. In the end we return the previous mate of y. If y was free, we obviously return null.

After performing Random-Settle on u, Invariant 2 is reestablished. Note that it suffices to handle only u, even when both endpoints of the inserted edge (u, v) own more than $\sqrt{n}$ edges after inserting (u, v). This is because when u rises to level 1, it takes sole ownership of the edge (u, v). The size of $\mathcal{O}_v$ is therefore reduce by one again and back at its inital size. Naturally the same holds the other way around where only v would be processed.

Furthermore the procedure may leave two vertices free, namely the mate of u from before performing Random-Settle on u, if u was matched, as well as the vertex z returned by the procedure. In order to guarantee, that the matching is maximal, the algorithm tries to settle both vertices mate(u) and z by scanning through the set of owned edges in search for a free vertex, which can then become the mate of mate(u) and z respectively. Note the difference to the settle performed by the naive algorithm examined in Section 4.1.2, where we scan through all neighbours.

4.3.3 Edge Deletion

An edge (u, v) can be matched or unmatched when it gets deleted. As we know from the naive algorithm, an unmatched edge being deleted does not affect the matching. For this particular algorithm we can also see, that it does not violate any of the invariants. Note that Invariant 2 does not state, that no vertex at level 1 is allowed to own less than $\sqrt{n}$ edges.

Obviously no vertex can own the edge (u, v) after it got removed from the graph. It follows that we have to update the according data structures $\mathcal{O}_u$ and $\mathcal{O}_v$ in any case, whether the edges was matched or not. If the edge was not matched, this is the only processing that needs to be done. Otherwise, if the edge was matched, removing it will leave both endpoints u and v free. This might lead to a potential violation of Invariant 1.

We distinguish two cases: If the edge was at level 0, we settle both endpoints u and v naively. Note that because of Invariant 3 we know that level(u) = level(v) and therefore can handle both endpoints in the same way. Now if the edge (u, v) was at level 1, we handle each endpoint $w \in \{u, v\}$ by passing ownership about all edges (w, x) at level 1 to the respective endpoint z. This decreases the size of the set $\mathcal{O}_w$. If the size $|\mathcal{O}_w|$ still exceeds or is equal to $\sqrt{n}$, the vertex has to remain at level 1 because of Invariant 2 and to be settled (according to Invariant 1). We settle a vertex at level 1 by performing the earlier introduced routine RANDOM-SETTLE on it (see Figure 1). Similar to edge insertion before, the vertex returned by the routine, if it is not NULL, as well as the previous mate of w, if there was one, have to be settled naively. In contrast, if the size $|\mathcal{O}_w|$ is now less than $\sqrt{n}$, the vertex can fall back to level 0. Since every edge at level 0 is owned by both its endpoints, we have to add every owned edge (w, x), that is at level 0, to the set of owned edges of the respective vertex x. Afterwards we perform a naive settle on the vertex w to guarantee that Invariant 1 holds. Adding edges to the ownership of x as we did before may increase the size of the set $|\mathcal{O}_x|$ so that it exceeds or is equal to $\sqrt{n}$ and hence violates Invariant 2. Therefore we scan through all owned edges (w, x) and if the size $|\mathcal{O}_x|$ is greater or equal to $\sqrt{n}$, we repair this issue by first performing RANDOM-SETTLE on the vertex x and then settling the vertex returned by the rountine as well as the previous mate of the vertex x on which RANDOM-SETTLE was performed naively, given that they are not NULL. We would like to mention, that the particular task of settling the previous mate of x is not mentioned in the original paper by Baswana, Gupta and Sen in the section about edge deletion. In order to maintain Invariant 1, it is however crucial to do so.

4.3.4 Approximation and Complexity

The algorithm proposed by Baswana, Gupta and Sen guarantees that the matching is a 2-approximate maximal matching after every update. Processing an insertion is done quite similiar to the naive approach, except the processing which is triggered, when a vertex rises to level 1. In thise case the algorithm is able to detect an augmenting path up to length 5. This is when the vertex u was matched before we perform Random-Settle on it and further the vertex z returned by the procedure as well as the previous mate of u, which has to be free after Random-Settle, are both settled successfully. For an incremental sequence this may increase the quality of the matching over the naive algorithm, however there is no guarantee that this case occurs.

The same case can occur when a matched edge at level 1 is deleted, falls to level 0 and thus forces the endpoint of an owned vertex to rise to level 1, because then Random-Settle has to be performed on this vertex and the previously mentioned scenario can arise. Again, this may lead to an increase of the matching quality in a particular scenario, but does not affect the lower bound on the matching quality.

The fact that the matching is maximal is guaranteed by Invariant 1. Since no vertex at level 1 is free, all free vertices have to be at level 0. Since no vertex at level 0 has a free neighbour at level 0, there is no edge that we could add to the matching, that would increase its size. The matching is therefore maximal and a 2-approximate maximum matching.

A key concept of the presented algorithm in order to reduce time complexity is to settle vertices u, that have *small* $\mathcal{O}_u$ using the naive approach and settling those with *large* $\mathcal{O}_u$ in a randomized manner. In this particular case the threshold between *small* and *large* is $\sqrt{n}$. This approach reduces the costs of settling a vertex naively to $O(\sqrt{n})$. However if we examine the procedure Random-Settle, we can easily see, that it is of cost $O(n)$. In the particular case where an edge at level 1 is removed, the vertex u on which Random-Settle is performed may own up to n edges. If further none of these edges other endpoints is at level 1, then we will not remove any edges from the set of edges owned by u. The loop in line 3 in Figure 1 will therefore run for $\mathcal{O}_u = n$ times.

Baswana, Gupta and Sen present a sophisticated analysis of their algorithm in order to prove that it achieves *amortized* runtime of $O(\sqrt{n})$. Reproducing their analysis would go beyond the scope of this work. Please refer to their paper [17] for detailed examination of the runtime.

4.4 Deterministic Algorithm by Neiman and Solomon

Unlike Baswana, Gupta and Sen in [17], whose algorithm is randomized, Neiman and Solomon [35] show a deterministic algorithm for maintaining a maximal matching in a dynamic graph. Their approach guarantees, that the matching calculated is a 3/2-approximate maximum matching and that update time is

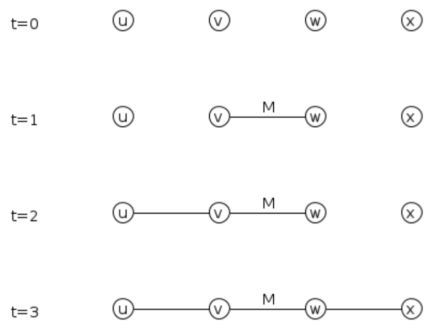


Figure 2: *Illustration of Example 4.4.1*

of $O(\sqrt{m})$ in *worst case*, where m denotes the number of edges present in the graph in the moment of the update.

4.4.1 Edge insertion

As mentioned earlier, any matching without augmenting paths of length at most $2k-3$ is a $(k/(k-1))$-approximate maximum matching, meaning that the matching contains at least $(k-1)/k$ fraction of the matching number $\nu(G)$. Remember that the matching number is the size of any maximum matching. Therefore a 3/2-approximate matching, as Neiman and Solomon guarantee it, is free of any augmenting paths of length at most 3, which follows from setting $k=3$ for the above equations.

In order to guarantee the approximation of the matching, we have to assure already after each insertion step, that no augmenting path of length 3 is present. The earlier mentioned, naive approach does only exclude the presence of augmenting paths of length 1, which is the equivalent to assuring that the matching is maximal. Consider the following example illustrated in Figure 2 for a more detailed explanation.

Example 4.4.1. *Our dynamic graph $\mathcal{G}$ in this minimal example consists of $\mathcal{V}=\{u,v,w,x\}$ and the dynamic edge set $\mathcal{E}=(E_0)$ with $E_0=\{\}$ on which we apply the sequence $S=((1,v,w),(1,u,v),(1,w,x))$. Note that our matching in the beginning is empty, therefore $M_0=\{\}$. Using the naive approach applying $(1,v,w)$ to the graph, where the 1 signs that we add the edge in this step, results in the matching $M_1=\{(v,w)\}$, since both vertices were free. If we now apply the next two sequence steps, the naive approach will not match any further edges, since for $(1,u,v)$ v is already matched, and for $(1,w,x)$ w is already matched. Therefore the matching after applying the last sequence step is $M_3=\{(v,w)\}$. It is easy to see, that we also have created an augmenting path of length 3, starting at vertex u and ending on x.*

Neiman and Solomon address this issue by scanning the surroundings of any edge, that cannot be simply added to the matching for the reason that one endpoint of the edge is already matched. More specifically, if an edge (u, v) cannot be added to the matching, because e.g. u is already matched, we scan the neighbours of the mate of u, which we call $u' = \text{mate}(u)$, for a free vertex x. By providing appropriate data structures this can be achieved in $O(\sqrt{n})$ time, which we will examine further in Section 4.4.4. If such a free vertex exists, we have found an augmenting path of length 3, which we augment. This increases the matching size by one. Further an edge insertion, where both endpoints are free, is processed as we already know, namely by simply adding this edge to the matching. Although detecting a vertex as free can be achieved in $O(1)$ time, matching an edge however entails updating several data structures, which takes $O(\sqrt{n+m})$ time as we will explain in a bit (see Section 4.4.4). An edge with both endpoints matched does not entail any further processing than the one needed to detect them as unfree. Reconsider the earlier Example 4.4.1, but this time using Neiman and Solomons approach to update the matching.

Example 4.4.2. *As before let the dynamic graph be* $\mathcal{G} = (V, \mathcal{E}) = (V, (E_0)) = (\{u, v, w, x\}, (\{\}))$ *and the sequence* $S = ((1, v, w), (1, u, v), (1, w, x))$. *We now apply S on $\mathcal{G}$, where i denotes the step and at each step we apply S_i.*

i=0: Since for the first step $M_0 = \{\}$ and therefore $v, w \notin M_0$ apply, the first edge (v, w) is added directly to the matching.

i=1: In the next step, we add (u, v) to $\mathcal{G}$. Since v is already matched, we check the surroundings of $\text{mate}(v)$ *for a free neighbour, however there exists none because the only neighbours of v are u, which is v's mate, and w, which is the other endpoint of the just added edge.*

i=2: In the last step we add (w, x). Again one endpoint is free, namely x, and the other one, w, is already matched. We scan the surroundings of $v = \text{mate}(w)$ *and detect u as free neighbour of v. Therefore we have found an augmenting path starting at x and ending at u. By inverting the augmenting path we receive the matching $M_3 = \{(u, v), (w, x)\}$.*

Note that for this particular example the matching M_3 is a perfect *maximum matching. We call a matching* perfect *if there is exists no free vertex.*

4.4.2 Edge Deletion

As mentioned in Section 4.1.2 the process of edge deletion may create new augmenting paths of length 1, which are simply edges with two free endpoints, but moreover may also create other augmenting paths of arbitrary length. In oder to tackle the augmenting paths of length 1, the algorithm from Neiman and Solomon checks for both freed vertices whether they have free neighbours and if so matches the freed vertices with those free neighbours. This is done in $O(\sqrt{m})$ time. We examine this improvement in runtime over the naive approach with runtime $O(n)$ more thoroughly in Section 4.4.4.

We now focus on guaranteeing that no augmenting paths of length at most 3 are present in the graph. First we will explain how augmenting paths of length 3 can rise from an edge deletion using the following example.

Example 4.4.3. *Let $\mathcal{G}$ be a dynamic graph with $\mathcal{V} = \{u, v, w, x, y\}$ and $\mathcal{E} = \{E_0\}$, where $E_0 = \{(u, v), (v, w), (w, x), (x, y)\}$ and let $M_0 = \{(u, v), (w, x)\}$. Note that this graph and the respective matching can be easily obtained from applying the sequence $R = ((1, u, v), (1, w, x), (1, v, w), (1, x, y))$ to an empty dynamic graph $\mathcal{G}' = (\mathcal{V}, (\{\}))$.*

We now apply a single sequence step $S_0 = (0, u, v))$ to the dynamic graph $\mathcal{G}$. Removing (u, v) from the graph entails removing the edge also from the matching. This leaves the node u free, but isolated, wherefore it can not be matched again. v at the other hand is then starting point to an augmenting path of length 3, which ends at the free node y. Note that the matching $M_1 = \{(v, w)\}$ is maximal, but only a 2-approximate to a maximum matching.

Neiman and Solomon approach this by checking both freed vertices u and v, if they are starting points to augmenting paths. This however does only happen, if the vertex degree is not more than a threshold of $\sqrt{2m}$, where m denotes the amount of edges present in the graph at the moment of the update. For simplicity we will explain the further behaviour of the algorithm for the node u, the processing for v is anyhow the same. Finding an augmenting path is done by scanning through all neighbours w of u and checking if $\text{mate}(w)$ has a free neighbour. Naively this would take $O(\min(n, m))$ time, since the runtime complexity of checking all neighbours is dominated by the vertex degree. However as we mentioned previously, this approach is only taken for vertices with a degree less than $\sqrt{2m}$, time complexity is therefore reduced to $O(\sqrt{2m})$. The alternative behaviour will be explained subsequently. Now if an augmenting path has been found, we can invert it in $O(\log n)$ time (see Section 4.4.4), hence the update time taken from this routine is $O(\sqrt{m})$.

For vertices with degree greater than $\sqrt{2m}$, we find a surrogate z for u with degree of at most $\sqrt{2m}$ who is the mate of a neighbour of u. We find such a surrogate by scanning linear through the neighbours of u, retrieving $z = \text{mate}(w), w \in N(u)$, where $N(u)$ denotes the set of neighbours of u, and then checking if $\deg(z) \leq \sqrt{2m}$. Neiman and Solomon claim, that finding such a surrogate z is done in at most $\sqrt{2m}$ steps, since otherwise the sum of degrees in the graph would be more than $\sqrt{2m} \cdot \sqrt{2m} = 2m$, which is impossible. Now that a surrogate vertex z is found, we unmatch the edge (w, z), match (u, w) and handle z then just as u and v were handled before. Since it is guaranteed, that $\deg z < \sqrt{2m}$, an infinite loop can be foreclosed.

4.4.3 Approximation

As we tried to outline throughout the section, the goal of the algorithm is to maintain a maximal matching that is free from augmenting paths of length at most 3. According to [27], this approach guarantees that the matching maintained is a 3/2-approximate maximum matching.

4.4.4 Complexity

In order to bound runtime complexity and guarantee deterministic behaviour, Neiman and Solomon use and maintain the following data structures.

- The matching M is saved in an AVL-tree which suppports insertion and deletion in $O(\log n)$ time.
- For each vertex $x \in \mathcal{V}$ an AVL-tree $N(x)$ is maintained, that holds all neighbours of x.
- A custom data structure $F(x)$ for each vertex $x \in \mathcal{V}$, that contains all free neighbours of x and supports insertion, deletion and querying, if a free neighbour exists, in $O(1)$ time and further allows retrieving a free node in $O(\sqrt{n})$ time.
- Finally an addressable maximum heap F_{max}, which contains all free vetices indexed by their degree. This data structure supports insertion, deletion and updating keys in $O(\log n)$ time, as well retrieving the vertex with highest degree in $O(1)$ time.

As mentioned earlier, detecting an augmenting path or detecting free vertices can be achieved in $O(\sqrt{n})$ or $O(1)$ time. However matching an edge is not as trivial as it might appear as it does entail updates on the data structure $F(w)$ for all $w \in N(x)$ for all $x \in \{u, v\}$. Naively this would mean update costs of $O(\deg(u) + \deg(v))$ time, which is $O(n)$, since the maximum degree for a vertex in a simple graph is $n-1$, where n. In order to bound this update time, Neiman and Solomon introduce two invariants, that state that:

Invariant 4. *All free vertices have degree at most* $\sqrt{2n+2m}$.

Invariant 5. *All vertices, that became free in round* i *have degree at most* $\sqrt{m}$.

The handling of Invariant 5 has been already introduced, but implicitly. The action of finding a surrogate z with $\deg(z) \leq \sqrt{2m}$ for a freed vertex u with $\deg(u) > \sqrt{2m}$, where u remains matched by matching the edge $(u, \text{mate}(z))$, guarantees that any vertex freed in round i has degree not more than $\sqrt{2m}$.

Invariant 4 however has to be handled explicitly by identifying *problematic* vertices, that are *close* to violate Invariant 4. They call a vertex *problematic*, if it is free and its degree exceeds $\sqrt{2m}$. Neiman and Solomon prove, that it is sufficient to handle one vertex per sequence step additionally to the two vertices, which are already handled by being involved in the performed edge update. This one vertex is the vertex with maximal degree from all free vertices, which can be obtained from the maximum heap F_{max} in $O(\log n)$ time (including removing this vertex from the heap as well as restoring the heap condition). This problematic vertex x is then handled like a vertex u of a freed edge (u, v) with $\deg(u) > \sqrt{2m}$ is handled. This is, we find a surrogate z for x, where $z = \text{mate}(y)$, with $y \in N(x)$, unmatch (z, y) in order to be able to match (x, y) and let the surrogate vertex z possibly become free. This however could create new augmenting paths of length 3 which is why we then call the same procedure, which we use to find augmenting paths from a freed vertex after deletion of a matched edge, to assure that no such augmenting path exists.

5 Experimental Evaluation

In this section we present how we evaluated the implemented algorithms experimentally and what results we gathered. Our tests were performed on real-world graphs taken from the KONECT database [29]. Since unfortunately, there are only few fully dynamic graphs available on KONECT, we also present methods, that we used to create fully dynamic sequences based upon static or incremental dynamic real-world graphs. We evaluate runtime and matching quality of the different algorithms in comparison to each other but also in comparison to the theoretically elaborated upper and lower bounds for runtime and matching quality respectively.

5.1 Implementation and Environment

In order to evaluate the previously presented algorithms experimentally, we implemented them in C++11 using g++ version 5.5.0 as compiler. As a base for writing the code we used the KaHIP project [40]. Unfortunately this project cannot be seen as an extension to KaHIP. The whole code is available at GitHub[1] and is licensed under GPLv3.0[2].

Our experiments consisted of running different sequences, which will be further described in Section 5.2.2 & 5.3. The sequences were processed by the different algorithms listed in Table 2 and data about time taken per update, graph edge cardinality, matching edge cardinality, average vertex degree and average vertex degree of matched vertices was gathered all r steps, where r is a custom parameter. In order to avoid corruption of data, especially the running time, caused by external events on the test machine, every experiment was run several times and the runtime was averaged. Because of the randomized nature of some of the algorithms, we also averaged the matching sizes compute in each run.

As a benchmark we used the Global Path Algorithm [30], [39] for maximal matching in static graphs to calculate the matchings for the static instances G_i of the dynamic graph $\mathcal{G}$ at the points i in time, where we also gathered the above mentioned data.

As a test machine we used a system running Ubuntu 18.04.1 LTS on 4 Intel Xeon cores with 2.20GHz, with 16GB RAM and 100GB disc space. Execution of experiments was done using `gnu screen` [44] sessions and also using `gnu parallel` [43] in order to execute several sequential experiments in parallel. This however turned out to be a problem for very big input sequences (about 10M steps), causing memory issues. Regarding the exhaustive use of RAM, we would like to mention, that we could not find any memory leaks using `valgrind` [36] on our test programs.

[1]`https://github.com/ripaul/dynamic_matching`
[2]`https://www.gnu.org/licenses/gpl-3.0.html`

5.2 Input Graphs and Sequences

We present some different approaches used to create fully dynamic sequences from static graphs. Further we give an overview about the real-world graph instances from which we created dynamic sequences in order to run experiments.

5.2.1 Creating Dynamic Sequences

We used several graphs from the *Koblenz network collection* (*KONECT*) [29] as test data. Unfortunately only few of the provided graphs match our requirements of being dynamic, undirected, loop-free and without parallel edges. Since most of the graphs from KONECT are static, we used the following approaches to create dynamic graphs from the static ones provided.

addition-only: As the name already implies this approach does construct the static input graph by subsequently adding the edges of the static graph to the dynamic one in order of appearance. If the input graph provides time stamps as some of the KONECT graphs do, the edges are first sorted by time stamp and then added. The maximal size of such a sequence is therefore the size of the input sequence. Note that any unprocessed KONECT graph can be seen as a sequence.

random-step: Before a new sequence step is randomly set to either be an edge insertion or an edge deletion. If it is an edge insertion the next edge in order of appearance or if applicable sorted by time stamps is added to the sequence. Furthermore we maintain an array that contains all edges, that are present in the graph after the recently added sequence step. Hence if the next step is to be an edge deletion, an edge is randomly taken from the array of present edges added as edge deletion to the sequence. Further it gets deleted from the array of edges present in the graph. The maximal size of such a sequence is $2 \cdot l$, where l is the length of the input sequence.

sliding-window: This approach takes an additional parameter ws to determine the window size of the sliding window. For the first ws sequence steps edges are added in order of appearance or sorted by time stamp. Note that we use 0-based counting. Afterwards every edge (u, v) added at step k is removed again at step $i = ws + k$. For a input sequence of length l, that we convert into a sliding-window sequence, no edges are left to be inserted at step $2 \cdot l - ws$. After this point we continue by deconstructing the graph again by removing edges for the next ws steps as before. The overall maximum length is then $2 \cdot l$ for a given input sequence of length l, further the window size ws must not exceed the sequence length l.

simple-outage: As suggested from Bergamini and Meyerhenke [18] we created sequences, where we remove a randomly chosen edge (u, v) from the graph and then reinsert it in the next step. This approach simulates the outage of a network connection. Similar to the sliding-window method, the first ws sequence steps

Name	$n \approx \lvert V\rvert$	$\approx k$	Type	Reference
dbpedia	4M	13.8M	S	[15], [1]
edit-enwiktionary	2.2M	9M	T	[12], [46]
edit-frwiktionary	1.9M	7.4M	T	[13], [46]
facebook-wosn-links	63.7k	817k	T	[45], [2]
flickr-growth	2.3M	33.1M	T	[32], [3]
link-dynamic-dewiki	2.2M	86.3M	D	[6], [37]
link-dynamic-frwiki	2.2M	59M	D	[7], [37]
link-dynamic-itwiki	1.2M	34.8M	D	[8], [37]
link-dynamic-nlwiki	1M	20M	D	[9], [37]
link-dynamic-plwiki	1M	25M	D	[10], [37]
link-dynamic-simplewiki	100k	1.6M	D	[11], [37]
livejournal-groupmemberships	13.9M	112.3M	S	[4], [33]
orkut-groupmemberships	14.3M	327M	S	[5], [33]

Table 1: *Input graphs from KONECT. k denotes the sequence length. For all graphs, except the dynamic ones, $m = |E| = k$ is valid. Regarding the type, S stands for a static graph instance, T for graphs with time stamps and D for real fully dynamic graphs.*

fill the graph up with edges. In the following phase we continue as already mentioned by removing and reinserting a randomly chosen edge. This method can be used to create arbitrary long sequences, however the window size ws must not exceed the length of the input sequence l.

pooled-outage: As an extension to the simple-outage method this approach does not delete and then reinsert only one edge, but right after the insertion phase creates a pool of size ps by removing ps randomly selected edges. Afterwards we randomly decide for each step whether an edge from the pool is reinserted or another edge is deleted from the graph and therefore added to the pool. The edge cardinality is then expected to be approximately $ws - ps$, where ws is the parameter determining the window size. Like the simple-outage method this approach can be used to create arbitrary long sequences, again the parameter ws must not exceed the length of the input sequence l.

5.2.2 Input Graphs and Parameters

In Table 1 we list all graphs that we either used to create dynamic sequences or that already were real dynamic graphs and could therefore be also processed natively.

Because of the earlier mentioned problem of exhaustive memory usage, we could not run our test programs on the full sized instances, but had to truncate them. We did this by creating sequences with up to 250 thousand vertices and about 8 million sequence steps.

The algorithms by Baswana, Gupta and Sen and Neiman and Solomon use thresholds to determine between *low* and *high* degree vertices. In order to test the behaviour of the algorithms for a dense dynamic graph, we also created

Algorithm	Abbreviation	ϵ
Baswana, Gupta and Sen	bgs	-
Naive	naive	-
Neiman and Solomon	ns	-
Random Walk	$\mathrm{rw}_{0.5}$	0.5
	$\mathrm{rw}_{0.1}$	0.1
	$\mathrm{rw}_{0.01}$	0.01
max-random-walk	$\mathrm{mrw}_{0.5}$	0.5
	$\mathrm{mrw}_{0.1}$	0.1
	$\mathrm{mrw}_{0.01}$	0.01
$\sqrt{m}$-random-walk	$\mathrm{mrw}_{\sqrt{m}}$	-
low-degree-settle	$\mathrm{lds}_{0.5}$	0.5
	$\mathrm{lds}_{0.1}$	0.1
	$\mathrm{lds}_{0.01}$	0.01
extended-naive	$\mathrm{en}_{0.5}$	0.5
	$\mathrm{en}_{0.1}$	0.1
	$\mathrm{en}_{0.01}$	0.01

Table 2: *Algorithms and respective parametrization tested throughout the experiments.*

sequences using the pooled-outage approach with only up to 5000 vertices but up to 6.25 million edges present in the graph.

The random walk algorithm as well as its variants 1,3 and 4 take a parameter ϵ to determine the maximal length of the random walk $k = 1/\epsilon$. We tested the random walk algorithm and each of the mentioned variants with the following three different $\epsilon = 0.5, 0.1, 0.01$, which results in maximal length $k = 2, 10, 100$ respectively. An overview about all algorithms tested during experiments is given in Thable 2.

5.3 Results

In this section we present our results from running the algorithms presented in Section 4 on different sequences. Our main interest lies in the runtime and the matching size achieved by the different algorithms. The graphs used for creating the fully dynamic sequences, that we used in our experiments, were all taken from KONECT [29] and are listed in Table 1.

We performed our first experiments on sequences created using the sliding-window and simple-outage approaches. However the results gathered from those experiments appeared rather odd. Although some of these odd phenomenas were caused by bugs in our testing software, we also concluded that the used approaches are rather weak in order to simulate real dynamic graphs as they always do alternating insertions and deletions, which is a quite predictable pattern. As an improvement we switched to using the pooled-outage approach, as it gives the whole sequence a more randomized structure.

graph	$\approx n_{\mathcal{G}}$	$\approx \bar{m}_{\mathcal{G}}$	$\approx \mathrm{d}_{\mathcal{G}}$	$\approx \max(\lvert\mathcal{M}\rvert)$	by	$\approx \min(\bar{t}_u)$	by
dbpedia	150k	185.8k	2.77	8777	ns	9 ms	naive
enwikt	142.8k	305.3k	4.66	3762	ns	7.21 ms	naive
frwikt	149.8k	318.3k	4.6	606	ns	5.83 ms	naive
facebook	56.1k	501.2k	18.46	24209	$\mathrm{mrw}_{\sqrt{m}}$	11.81 ms	naive
flickr	42.8k	463.6k	23.15	1845	ns	7.84 ms	$\mathrm{rw}_{0.5}$
dewiki	60.3k	279.7k	10.14	8090	ns	8.38 ms	naive
frwiki	65.9k	337.9k	11	15377	ns	9.9 ms	naive
itwiki	71.5k	367.1k	10.94	17500	$\mathrm{mrw}_{\sqrt{m}}$	10.76 ms	naive
nlwiki	62.8k	335.9k	11.34	17581	ns	9.81 ms	naive
plwiki	49.6k	336k	14.44	13723	ns	9.76 ms	naive
simplewiki	48.9k	251.7k	11.02	15514	ns	9.33 ms	naive
livejournal	150k	500.6k	7.48	8276	ns	8.62 ms	naive
orkut	112.7k	501k	9.69	20059	ns	10.19 ms	naive

Table 3: *Properties and results of pooled-outage sequences.* $m_{\mathcal{G}}$ *denotes average amount of edges present throughout the sequence,* $\mathrm{d}_{\mathcal{G}}$ *denotes the average degree throughout the sequence.*

5.3.1 Experiments on Pooled-Outage Sequences

As we stated before, we created pooled-outage sequences for our further tests from the graphs in Table 1 with up to 150k vertices, up to 600k edges present in the graph and approximately 8 million sequence steps. All of these sequences were processed by all algorithms from Table 2. As mentioned in Section 5.2.2 the basic random walk algorithm as well as the max-random-walk, low-degree-settle and extended-naive variants were run with three different parameters $\epsilon = 0.5, 0.1, 0.01$. In order to give a general overview we listed detailed properties of the sequences as well as the results from running all algorithms on the according sequence in Table 3. Further we show the largest average matching size $max(|\bar{M}|)$ achieved and the best average update time $min(\bar{t}_u)$ achieved as well as the corresponding algorithms. In Figure 3 we plotted the distribution of the results computed from the sequences from Table 3. Every data point in this plot represents the result computed by an algorithm on a particular sequence. The coordinates (x, y) of a data point in the plot are computed as follows.

Let i be a particular experiment, then

$$x = \frac{\min(\bar{t}_{u_i})}{\bar{t}_{u_i}}, \qquad y = \frac{|\bar{\mathcal{M}}_i|}{\max(|\bar{\mathcal{M}}_i|)}.$$

The large circles mark the average quality of the respective algorithm, the diameter of the circle represents to the variance of all results computed by the algorithm.

As a main result we observed that the naive algorithm outperforms all other algorithms in terms of runtime, as one can easily see in Table 3. This is rather odd since theoretically speaking the naive algorithm has worse runtime complexity than the algorithms by Baswana, Gupta and Sen and Neiman and Solomon. However the more sophisticated algorithms maintain additional data strucutes, which of course takes additional time. The naive algorithm on the other hand is very light-weight in comparison and does not need any additional data structure apart from those needed to store the graph and the matching.

If we take a look at the average degrees of the sequences, we can see that they are quite low with the maximal average degree being only $23.15 \approx \frac{n}{1848}$. This

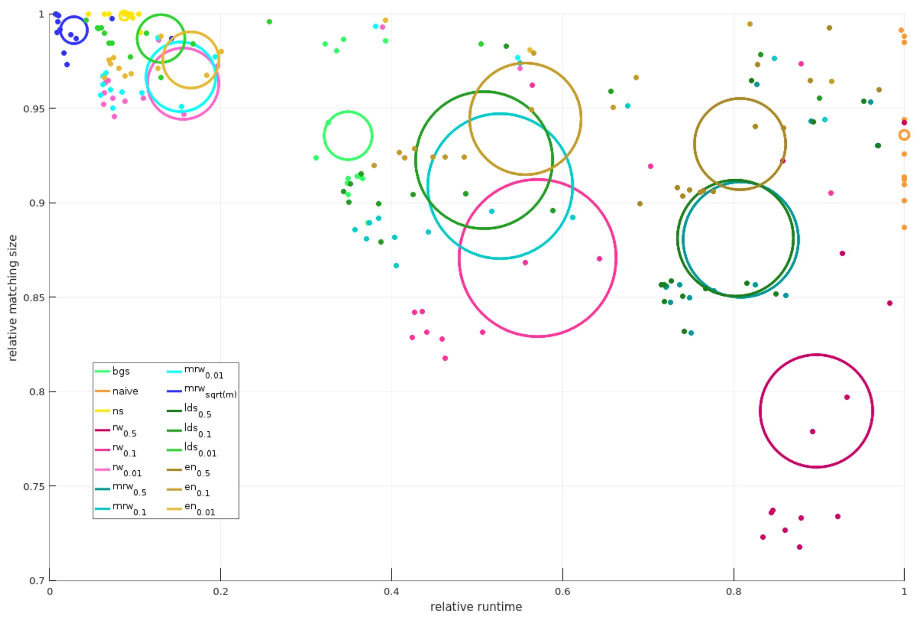
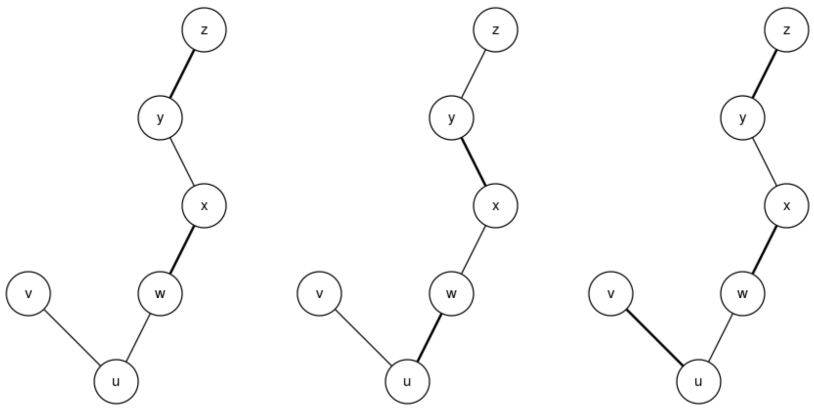


Figure 3: *Distribution of all experiment results on sequences from Table 3. Matching size and runtime are relative to the respective* best *result, large circle represent average quality and variance of results of the respective algorithm.*

may be a cause why the naive algorithm performs so well, since its expected runtime at any point in time i is $O(\mathrm{d}(G_i))$, where $\mathrm{d}(G_i)$ denotes the average vertex degree for the graph G_i. As already mentioned earlier, Baswana, Gupta and Sen and Neiman and Solomon use thresholds to distinguish *low* degree vertices from *high* degree vertices. These thresholds are crucial for the theoretical runtime complexity presented by the authors. The algorithms should perform best in comparison to the naive algorithm when many vertices exceed these thresholds because then $O(\sqrt{m+n}) < O(\mathrm{d}(G_i))$ (for Neiman and Solomon) and $O(\sqrt{n}) < O(\mathrm{d}(G_i))$ (for Baswana, Gupta and Sen) become more likely.

Deduced from this observation we performed further experiments, where we tried to achieve particularly high average vertex degrees. We present the results later in the paper in Section 5.3.3.

Another interesting observation is the particular bad performance of the basic random walk algorithm with $\epsilon = 0.5$ in terms of matching size in Figure 3. Although the algorithm is the second fastest, the matching size is also the worst among all algorithms. This fact can be explained by the issue, that the matching maintained by the basic random walk algorithm is not maximal, as we already examined in Section 4.2.1 and Example 4.2.1. We see, that the matching size of the random walk algorithm increases with the length of the random walk $k = 1/\epsilon$, however we can also clearly see, that it always performs worse than any



(a) $|M| = 2$ (b) $|M| = 2$ (c) $|M| = 3$

Figure 4: *In* (a) *u was the endpoint of a deleted matched edge, wherefore it remains free right after the edge deletion. In* (b) *we see a possible outcome of random walk variant 1, in* (c) *the outcome of variant 4 for the same scenario. Matched edges are indicated as bold line.*

of the random walk variants, which guarantee to maintain a maximal matching.

From the last observation follows also, that the attempt of improving the matching quality made by the random walk variants is quite successful. We can see in the scatter plot in Figure 3 that the max-random-walk variant, which extends the basic random walk algorithm by settling the last vertex naively, does always return a larger average matching size. However the extended-naive does return even better results. A random walk performed by the max-random-walk algorithm always carries the risk, that the vertex u, on which the random walk was started, could have been settled with some neighbour v but is instead matched with a vertex w, that was already matched and was therefore freed. In Figure 4a we illustrated this initial scenario. If the last vertex z of such a random walk remains free, because it has no free neighbour, the matching size remains the same. Further such a random walk creates a new augmenting path starting at its last vertex z and ending at the free neighbour v of the vertex u, where the random walk started (see Figure 4b). The extended-naive algorithm tries to match the freed vertices of a removed matched edge naively and only if a vertex could not be matched with some new mate, a random walk is performed on that vertex. By doing so, it reduces the number of previously mentioned random walks, that don't increase the matching size and therefore anticipates the creation of the mentioned augmenting path. The result for our illustrated scenario can be seen in Figure 4c. The extended-naive algorithm scans through the neighbours of u, finds v as a free neighbour and therefore matches the edge (u, v).

Regarding the particular good runtime of the naive algorithm in Figure 3 we can assume that performing a naive settle is averagely achieved faster than a

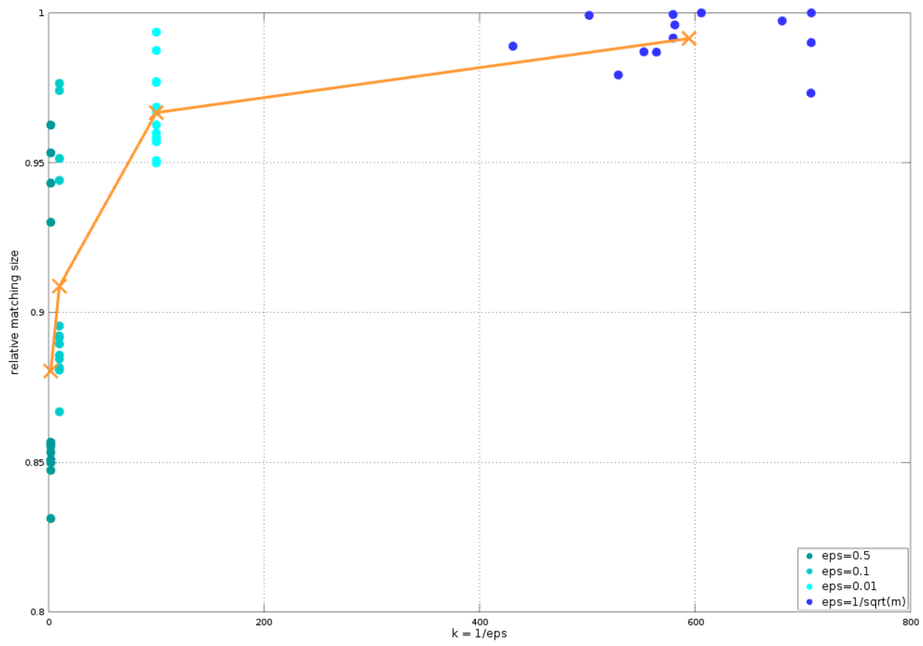


Figure 5: *Matching size results and average matching size from random walk algorithms in dependence of length of the random walk k.*

random walk, at least on the particular test graphs. This explains the runtime improvement of the extended-naive variant over the max-random-walk variant, since extended-naive performs less random walks than max-random-walk.

Although quite expectable we find it noteworthy that increasing the length of the random walks of the different random walk algorithms does result in an improved matching size. This indicates that the declared goal of finding augmenting paths of arbitrary length $l < k$ in order to increase the matching size is quite successful. In Figure 5 we visualized the matching size and average matching size of the random walk algorithm results as well as the respective ϵ used to achieve those results. We expect the matching quality to increase with smaller ϵ, since a high maximal random walk length does give the chance of finding more augmenting paths. Obviously an arbitrary random walk of length k cannot detect an augmenting path of length $l = k + 1$. As we can see, with increasing length of the random walk the average matching quality converges asymptotically, most probably towards the size of the maximum matching.

However this improvement comes at the cost of increased update time per sequence step as we can deduce from Figure 3. We raise the question, if we can determine some ϵ experimentally for which further decrease, and hence increase of the random walk length, does not improve the matching size significantly. In Section 5.3.4 we peformed further experiments where we ran the random walk algorithms with a variety of different ϵ in order to answer the raised question.

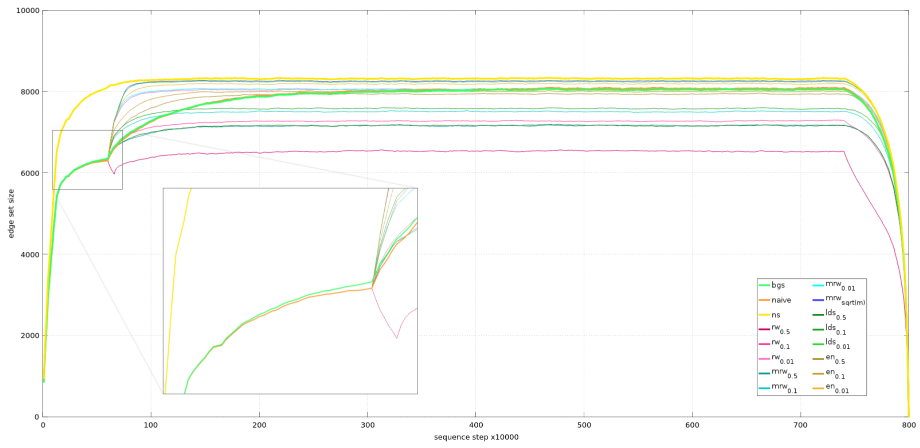


Figure 6: *Average matching size over time for the* dewiki *sequence from Table 3. Each algorithm was run 5 times, results were averaged. Neiman-Solomon and Baswana-Gupta-Sen are highlighted.*

As stated in Table 3 and observable in Figure 3 the algorithm by Neiman and Solomon does averagely compute the best matching sizes for the given test graphs. This is most probably because of the fact, that it is the only algorithm of the ones we tested, that guarantees that no augmenting path of length at most 3 exists. Note that all other algorithms do only guarantee the matching to be a 2-approximate maximum matching and therefore to be free of trivial augmenting paths of length 1.

In Figure 6 we plotted the matching size per sequence step for the experiment on the *dewiki* sequence from Table 3. This figure shows a very common result, that we achieved from running our algorithms on the *pooled-outage* sequences described in Section 2.2. For the first 600k steps we perform only additions, then for a very short subsequence of 60k steps we perform only deletions in order to create the pool, afterwards we start doing insertions and deletions randomly, either from the pool into the graph or the other way around.

A noteworthy phenomena we would like to mention is the gap between the matching size calculated by the Neiman-Solomon algorithm and all other algorithms. As we already mentioned, the Neiman-Solomon algorithm is the only one to assure the absence of augmenting paths of length at most 3 and therefore guaranteeing a 3/2-approximate maximum matching. This obviously has to hold for a pure incremental sequence as well as for a fully dynamic sequence. The gap between the result from the Neiman-Solomon algorithm and the other algorithms indicates that this additional effort does have a significant impact on the matching quality.

In contrary almost all the other algorithms, except Baswana, Gupta and Sen, do handle edge insertions in the naive manner described in Section 4.1.1. This does guarantee us a maximal matching and therefore 2-approximate maximum cardinality matching in $O(1)$ time. This naive approach works deterministically, which is the reason for the exact similarity of the result of all random walk algorithms as well as the naive algorithm for the pure incremental phase.

By having a look to the close-up in Figure 6, we can see that the result of Baswana, Gupta and Sen does improve slightly over the result of the naive algorithm already during the pure incremental phase. This can be explained by recalling Invariant 2, which states, that every vertex at level 0 owns less then $\sqrt{n}$ edges at any moment of time. Naturally the set of owned edges can increase when new edges are inserted in the graph and therefore also exceed $\sqrt{n}$. If this threshold is exceeded, the algorithm fixes the issue by moving the *dirty* vertex, which violates the invariant, to level 1, which again is done by performing the routine RANDOM-SETTLE (see Figure 1) on it. As we examined in Section 4.3.4, calling RANDOM-SETTLE can detect augmenting paths up to length 3 and augment them. This results in an increased matching size over the naive insertion process.

Another observation worth remarking is the sudden change in matching quality for all algorithms except Neiman-Solomon after the pure incremental phase of the *pooled-outage* sequence at about 600k steps. On the one hand we have the random walk algorithms that perform mainly a steep improvement in the matching size as soon as the sequence starts to contain edge deletions as well as edge insertions. We can observe that the slope increases with smaller ϵ and therefore larger maximal random walk length $k = \frac{1}{\epsilon}$. On the other hand we have the naive algorithm as well as the algorithm by Baswana, Gupta and Sen, which improve more slowly, but grow to outperform the random walk algorithms with $\epsilon \geq 0.1$.

Generally, we assume the improvement that sets in when first edge deletions occur to be caused by the fact, that the naive insertion process which is used by these algorithms can create maximal matchings with a high number of augmenting paths. Our assumption is encouraged by the previously mentioned gap between Neiman-Solomon and the other algorithms, which is caused by the exclusion of augmenting paths of length 3. In such a matching, where no effort has been made to detect and exclude augmenting paths, the number of augmenting paths present in the graph obviously increases. Generally speaking, the deletion of a matched edge can lead to the scenario, where we delete the middle edge of an augmenting path of length 3. The probability of occurence of such a scenario increases with the number of augmenting paths present in the graph, which is, as we presume, quite *high* if no effort has been made, to resolve any augmenting paths. In this case the naive algorithm as well as Baswana-Gupta-Sen and the extended-naive algorithm do find new mates for the freed endpoints of the deleted edge. This results in an improved matching size of $|M_{i+1}| = |M_i| + 1$.

The random walks do have the capability of finding augmenting paths up to length $k = \frac{1}{\epsilon}$, but since they try to detect these augmenting paths in a randomized manner, there is obviously no guarantee, that an augmenting path is found even if there exists one. However the steep slope in matching size of the random walk algorithms does indicate, that the approach of performing random walks does indeed help to find more augmenting paths and therefore to increase the matching size significantly. Further we can also see that more augmenting paths are found with a larger random walk length k, which confirms the observations from Figure 5 addressing the effect of the parameter ϵ on the resulting matching quality.

We explain the slower increase in matching size for Baswana-Gupta-Sen and the naive algorithm by the fact, that obviously the number $c_{l \geq 3}$ of augmenting paths with length $l \geq 3$ is at least as high as the number $c_{l=3}$ of augmenting paths with length exactly 3, but more probably does exceed $c_{l=3}$ significantly. This alone does not suffice to explain the stronger increase of the random walk algorithms over Baswana-Gupta-Sen and the naive one, since the random walk algorithms do not necessarily have to find such an augmenting path even if there exists one, whereas Baswana-Gupta-Sen and the naive algorithm will find an augmenting path of length 3, if there exists one starting from the respectively handled vertices. However the results of our experiments imply, that the probability of finding an augmenting path of arbitrary length $l \leq k$ with a random walk of maximal length k is at this point indeed higher than the probability, that a matched edge is deleted, which was the middle edge of an augmenting path of length 3.

As we mentioned, we can see in Figure 6 Baswana-Gupta-Sen and the naive algorithm outperform the random walk algorithms with $\epsilon \geq 0.1$ approximately between the sequence steps 2 and 5 million. Throughout this subsequence we have approximately the same amount of edge insertions and deletions. We suspect the reason, that Baswana-Gupta-Sen and the naive algorithm start to outperform the random walk algorithms to lay in the previously mentioned probability of finding augmenting paths of arbitrary length $l \leq k$. By finding augmenting paths and resolving them, we reduce the number of free vertices by two for every augmenting path resolved. This obviously reduces the probability of finding an augmenting path, since there can be maximally $n_{f,i}/2$ augmenting paths being resolved, where we use $n_{f,i}$ to denote the number of free vertices in the graph G_i. At some point the probability of finding further augmenting paths of length $l \leq k$ using random walks of maximal length k does therefore seem to get so low, that no further improvement in matching size can be observed, although there might still exist free vertices. Although Baswana-Gupta-Sen and the naive algorithm cannot detect augmenting paths of length $l > 3$, the detection of an augmenting path with length 3 of which the middle edge is being deleted is guaranteed, whereas the random walk algorithms can detect such an augmenting path only with probability $\frac{free(u)}{\deg(u)}$, where $free(u)$ returns the number of free vertices in the neighbourhood of u. Apparently the probability to find an augmenting path or a free neighbour for a freed vertex randomly is smaller than the probability of deleting such a middle edge of an augmenting path with decreasing n_f, which causes the Baswana-Gupta-Sen and naive algorithm to pass the random-walk algorithms with $\epsilon \geq 0.1$.

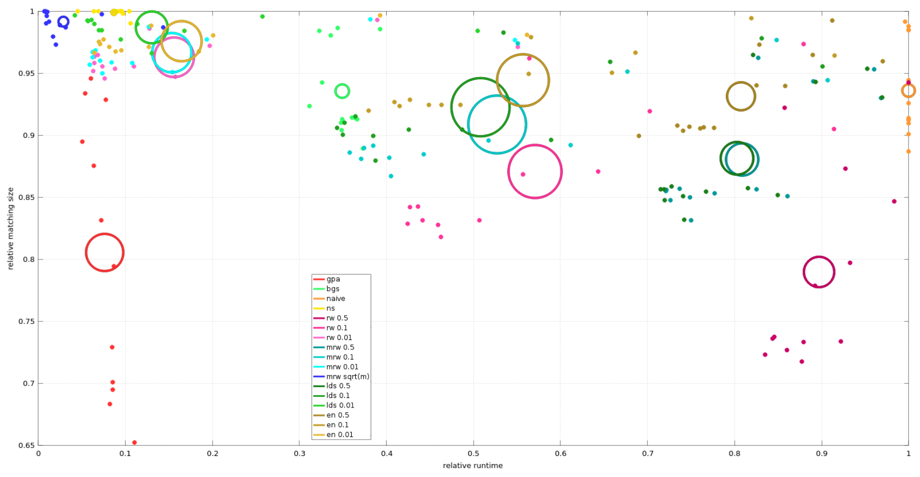


Figure 7: *Distribution of all experiment results on sequences from Table 3. Matching size and runtime are relative to the respective* best *result, large circles represent average quality and variance of results of the respective algorithm. Results from the static Global Path Algorithm are included here.*

5.3.2 Comparing With Static Algorithm GPA

Developing algorithms to dynamically update maximal matchings on a dynamic graph is obviously motivated by the fact, that recomputing the matching after each edge update from scratch seems to be a wasteful approach. Hence, we compared our dynamic algorithms to the results computed by the GPA algorithm proposed by Maue and Sanders in [30] by using the GPA implementation by Sanders and Schulz [39].

We computed the matchings using the GPA algorithm on static snapshots of the dynamic graphs used for testing the dynamic algorithms. These snapshots were obtained after performing some fixed number of edge updates, which in our case was 10k updates for most experiments.

In Figure 7 we present the results of the previous Section 5.3.1 but include the static results in order to compare them. As we can see, its results rank in quality as well as in update time below almost all dynamic algorithms. For the update time this is quite expectable. Note that GPA was only performed every 10k sequence steps. Recomputing the matching using it after every step would therefore take 10k times longer.

Regarding the matching sizes, the results in Figure 7 do not match the expectations. GPA does compute a $\frac{1}{2}$-approximate maximum matching in the worst case, but achieves better results empirically according to [30]. However all our algorithms except Neiman-Solomon and the extended insertion algorithms from Section 5.3.6 do also compute $\frac{1}{2}$-approximate maximum matchings, still Figure 7 implies that the dynammic algorithms return better results regarding matching quality. A reason for the averagely better performance of the dynamic

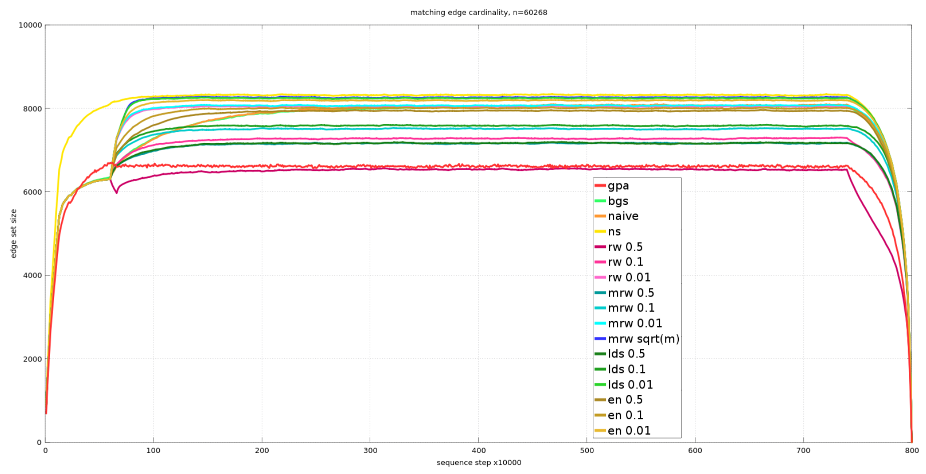


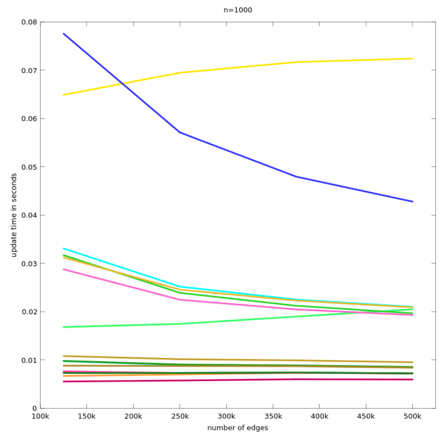
Figure 8: *Average matching size over time for the* dewiki *sequence from Table 3 with the results from the static Global Path Algorithm included.*

algorithms can be found when looking at the matching size growth of particular experiments. In Figure 8 we show the result of the paritcular experiment on the *dewiki* sequence from Table 3 with the GPA results included. We can see that during the pure incremental phase the GPA algorithm is able to compute a better matching than the algorithms using the naive insertion process. However at the point, where the edge deletions set in, which we already identified as a moment of high interest in Section 5.3.1, the dynamic algorithms start outperforming the static one. This reveals a key advantage of the dynamic algorithms, as they have the ability to improve previously computed matchings, whereas the static algorithm can not improve a previously computed matching since it starts its computations from scratch each time. Further the particular shape of our pooled-outage sequences favours some of our algorithms, since during the long phase of mixed insertions and deletions, the dynamic algorithms do outrun the static algorithm by steadly improving their results and also the average matching size throughout the whole sequence is therefore shifted in there favour, which gives us then the results as presented in Figure 7.

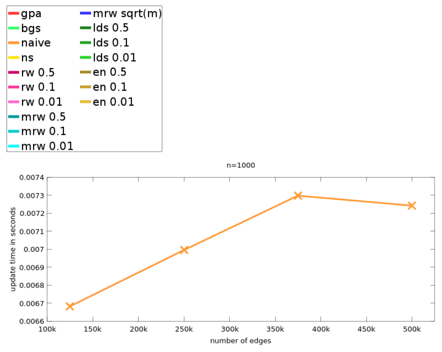
Overall this comparison outlines very clear, that recalculating the static matching after every edge update is a wasteful approach. Further our experiments imply, that the dynamic algorithms are able to compute larger matchings than at least the GPA algorithm by steadily improving their results over time.

5.3.3 High Average Degree Sequences

We observed that the naive algorithm performs best on our test sequences regarding update time. This does not match the expected update times regarding the theoretical time complexity of the respective algorithms. We suspect that our test graphs might be too small and not dense enough to observe the theo-



(a) *Average update time for all algorithms.*



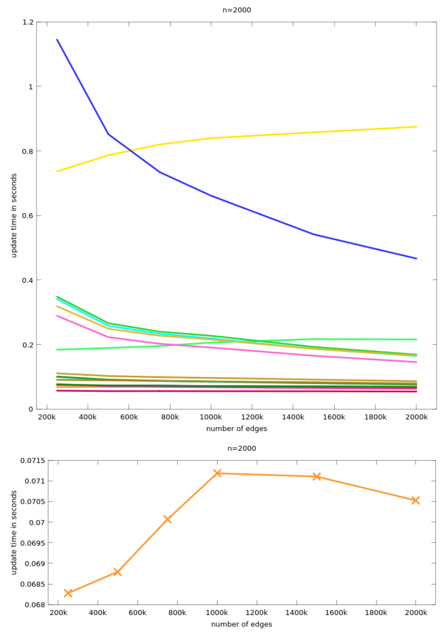
(b) *Average update time for naive algorithm only.*

Figure 9: *Average running time throughout five randomly created pooled-outage sequences with constant $n = 1000$ and increasing density.*

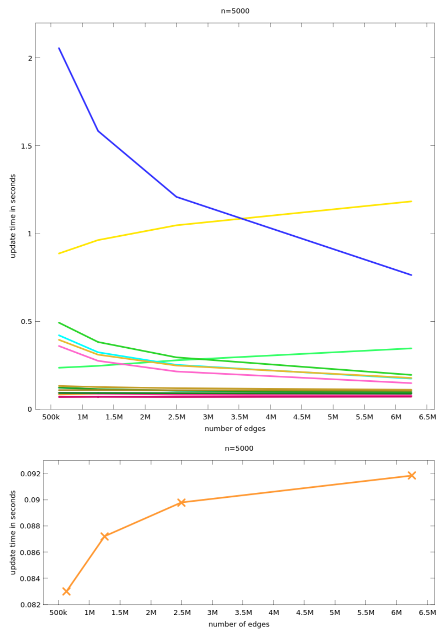
retical disadvantage of the naive algorithm compared to Neiman-Solomon and Baswana-Gupta-Sen. In order to examine the scalability of the algorithms regarding update time, we tried to create test sequences, which achieve a particular high density.

In an undirected simple graph with n vertices, the maximum degree of a single vertex can not exceed $n - 1$. The maximal number of edges in a simple graph is therefore $\frac{n(n-1)}{2}$. We tried creating pooled-outage sequences from the real world graphs from the KONECT data base, but where not able to create high average degree sequences due to the overall low average degree of the real world graphs. We therefore switched to creating high average degree sequences randomly with equally distributed edges. Our first experiments were run on pooled-outage sequences with $n = 1000$, $m_i \leq 125k, 250k, 375k, 499.5k$ edges and 2 million sequence steps. This results in an expected average edge degree $\mathrm{d}_{G_i} \approx \frac{n}{4}, \frac{n}{2}, \frac{3n}{4}, n - 1 = 250, 500, 750, 999$ respectively.

In Figure 9a we show the average update time achieved by the different algorithms and with growing number of edges present in the graph. A close look at the naive algorithm as provided in Figure 9b shows, that the update time of the naive algorithms grows with increasing density. Regarding Figure 9a the update time of the naive algorithms seems to be almost constant in comparison with the update times of the other algorithms. Although an increase in update time of the naive algorithm with increasing density was expected and can be observed here, we also expected to see the naive algorithm scale worse than Baswana-Gupta-Sen and Neiman-Solomon. Figure 9a show, that the two algorithms scale worse than the naive algorithm in this particular scenario. We conjecture, that this observations comes from the overall small problem size.



(a) $n = 2000$



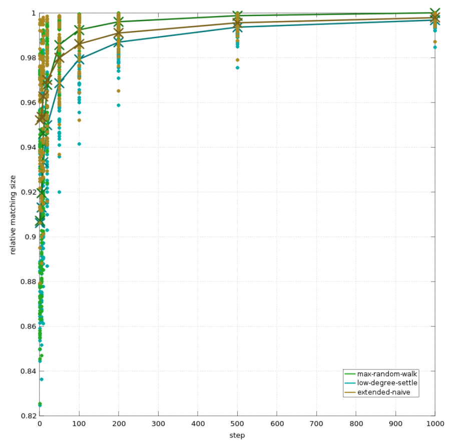
(b) $n = 5000$

Figure 10: *Average running time throughout five randomly created pooled-outage sequences with constant* $n = 2000, 5000$ *respectively and increasing density. See Figure 9 for legend.*

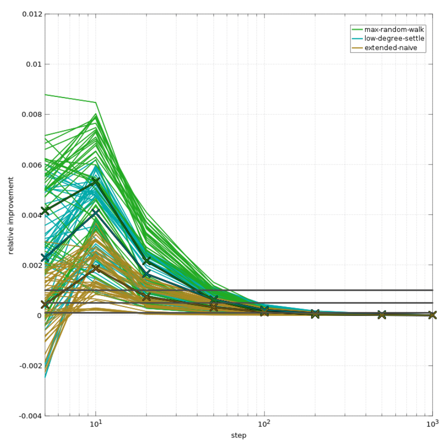
In order to further examine the runtime complexity of the algorithms, we performed experiments on larger graphs with more vertices and comparable average vertex degrees. We created pooled-outage sequences with $n' = 2000$, $m'_i \leq 250k, 500k, 750k, 1000k, 1500k, 2000k$ and 4 to 6 million sequence steps and also with $n'' = 5000$, $m''_i \leq 625k, 1250k, 2500k, 6250k$ and 6 to 20 million sequence steps. For the sequences with $n' = 2000$, this resulted in sequences with an expected average degree of $\mathrm{d}_{G'_i} \approx \frac{n'}{8}, \frac{n'}{4}, \frac{3n'}{8}, \frac{n'}{2}, \frac{3n'}{4}, n' - 1 = 250, 500, 750, 1000, 1500, 1999$ and for $n'' = 5000$ in sequences with an expected average degree of $\mathrm{d}_{G''_i} \approx \frac{n''}{20}, \frac{n''}{10}, \frac{n''}{5}, \frac{n''}{2} = 250, 500, 1000, 2500$.

Figure 10 present the results on those larger test sequences. The results are quite similar to the earlier results shown in Figure 9. Compared to Baswana-Gupta-Sen and Neiman-Solomon, the naive algorithm seems to have constant update time regardless of the average vertex degree. The closer look on the update time of the naive algorithm reveals, that its update time actually grows with increasing average vertex degree. However we observe the same phenomena as in the previous experiments, where the algorithms by Baswana-Gupta-Sen and Neiman-Solomon still scale worse than the naive algorithm.

Noteworthy are the peaks, that we can see in the update time of the naive algorithm in Figure 9b and 10a. The update time does drop for average degrees



(a) *Growth of the relative matching size per random walk length.*



(b) *Relative improvement of the matching per additional random walk step.*

Figure 11: *Results of running the sequences from Table 3 with the random walk algorithms* ***low-degree-settle****,* ***extended-naive*** *and* ***max-random-walk*** *multiple times with increasing random walk lengths.*

of more than $\frac{n}{2}$, where n is the number of vertices in the graph. However an high vertex degree does not necessarily mean, that the naive algorithm scans through all neighbours. Instead the algorithm will break as soon as a free neighbour which is to become the new mate of a freed vertex is found. The probability of finding a free neighbour grows, if a free neighbours is added to some vertex u, since the probability, that a randomly picked neighbour of u is free, is $p = \frac{free(u)}{\deg(u)}$, where $free(u)$ denotes the number of free neighbours of u. We conjecture, that a growing vertex degree $\deg(u)$ does increase the probability of finding a free neighbour for u and that this increased probability has a reducing effect on the update time. Apparently from some average vertex degree on this effect outweighs the increased time complexity caused by a higher vertex degree.

Further we can see a significant improvement in the achieved update time of the random walk algorithms with increasing density. This is an interesting result as runtime complexity of all random walk algorithms except the basic variant is $O(n)$. We presume that also this improvement comes from an increase in probability of finding a free vertex in a high average degree graph.

5.3.4 Determining an Optimal ϵ

We observed that longer random walks result in an improved matching size, however we also observed, that the length of the random walk is a key factor for the running time of our random walk algorithms and further that the matching size seems to grow asymptotically against some upper limit. The latter makes clearly sense as the size of any maximal matching can not exceed the size of a maximum matching $\nu(G)$. Increasing the length of the random walk $l = \frac{1}{\epsilon}$ arbitrarily is not meaningful, as it increases the average update time of our algorithms and further the relative improvement on the matching size achieved by each additional step does drop with increasing l.

We performed extended experiments on the test sequences from Table 3 in order to determine some ϵ up to which it makes sense to increase the length of the random walk. In order to do so we determined three thresholds $\varphi_1 = 0.001$, $\varphi_2 = 0.0005$ and $\varphi_3 = 0.0001$. Using these thresholds, we want to examine from which point on the relative improvement achieved by an additional random walk step does drop below the defined thresholds. For these experiments, we only ran the random walk algorithms **max-random-walk**, **low-degree-settle** and **extended-naive**, each one parametrized with $\epsilon = 0.5, 0.2, 0.1, 0.05, 0.02, 0.01, 0.005, 0.002, 0.001$.

In Figure 11a we present a figure quite similar to Figure 5, where we presented the relative matching size achieved with the respective random walk length. The observations, that we already could do in the earlier figure, do come out even clearer for this experiment. We can see that with increasing random walk length l, the matching size does grow asymptotically against some upper limit. For the first approximately 50 to 100 steps, we can easily see a significant improvement in matching quality between the different random walk lengths.

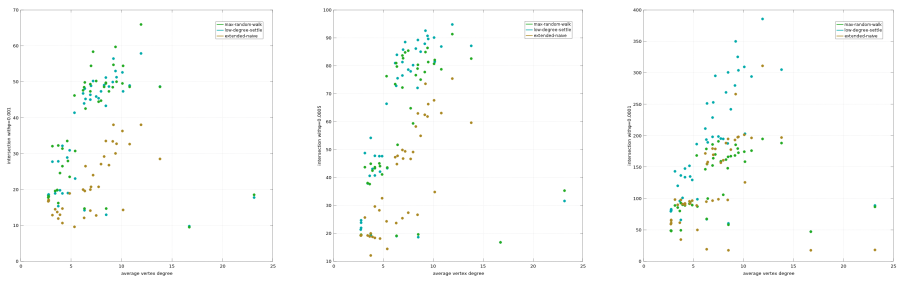
In Figure 11b we present the relative improvement in matching size for by additional random walk step. Since we did not perform experiments for all random walk lengths between 2 and 1000, we interpolated the results linearly. Let y_a and y_b be the relative matching sizes achieved by random walks of length a and b respectively with $a < b$, then

$$Y = \frac{y_b - y_a}{b - a}$$

is the the relative improvement achieved per random walk step between the lengths b and a. We can see that for most experiments the relative improvement per additional step increases at least for the first 100 steps. Afterwards the relative improvement drops. Further we also display the average relative improvement for each of the three algorithms as well as the thresholds φ_1, φ_2 and φ_3. We can see, that the average relative improvment of the extended-naive algorithm does drop below φ_1 after a random walk of approximately 20 steps, the other two algorithms drop below φ_1 at approximately 40 steps.

Obviously the relative improvement does not drop below the thresholds at the same step for all experiments. In Figure 12 we present the random walk length l_φ at which the relative improvement drops below the corresponding threshold φ as well as the respective average vertex degree for every experiment in a scatter plot.

The scatter plots imply, that there exists a correlation between the average vertex degree of the input graph and the optimal random walk length. A first assumption led us to examine if there exists a correlation between the average



(a) $\varphi_1 = 0.001$ (b) $\varphi_2 = 0.0005$ (c) $\varphi_3 = 0.0001$

Figure 12: *Relation between the average vertex degree throughout the test sequence and the optimal random walk length l_φ.*

number of edges and/or the number of vertices in the graph, as this would have indicated, that larger graphs require longer random walks in order to compute a good matching. We could however find no such correlation when comparing the optimal random walk length with the respective properties.

We conjecture, that the actual vertex degrees are not equally distributed at the average vertex degree and further presume, that therefore in real-world graphs regions of density above and also below the average vertex degree exist. A random walk, that enters a dense region of the graph has a high probability to stay in this dense area, as presumably most edges of a vertex in such a region do lead to other vertices of the same region whereas only few edges lead either to a sparse or another dense region. Thus a free vertex in such a region, which is what we try to find by performing a random walk, is more likely to be detected than in a sparse region. As in our *pooled-outage* test graphs the pool of edges being inserted and removed is actually constant, there is presumably only small change in the distribution of dense and sparse regions. Hence we further presume, that after performing a long sequence of approximately equally distributed insertions and deletions, most free vertices and therefore augmenting paths within dense regions are found and resolved as presumably most random walks will at some point enter a dense region and will most likely find a free vertex in such a region as long as there exist some. Increasing the random walk length increases the probability to find a way out of the dense region into regions, where free vertices may still be found and hence the matching size be further improved. This corresponds the results seen in 12, which imply that with increasing average vertex degree of a dynamic graph, the length of the random walk in order to further improve the matching size grows.

This theory however needs further examination. Performing similar experiments on randomly generated graphs with nearly equally distributed edges between the set of vertices might give further insight, as such graphs do not have regions, which differ largely in density. Also tracking the average degree per step of a large number of random walks could give further insight, since we expect the degree per step to stay on an approximately high level above the

graph	$\approx n_{\mathcal{G}}$	$\approx \bar{m}_{\mathcal{G}}$	$\approx \mathrm{d}_{\mathcal{G}}$	$\approx \max(\lvert\mathcal{M}\rvert)$	by	$\approx \min(\bar{t}_u)$	by
dewiki	250k	545.9k	7.99	7124	ns	6.03 ms	$\mathrm{rw}_{0.5}$
frwiki	250k	893.6k	11.19	20307	$\mathrm{mrw}_{\sqrt{m}}$	6.51 ms	naive
simplewiki	100.3k	362.4k	11.38	19279	$\mathrm{mrw}_{\sqrt{m}}$	6.44 ms	naive

Table 4: *Properties and results of the real dynamic test sequences.* $\bar{m}_{\mathcal{G}}$ *denotes the average amount of edges present throughout the sequence,* $\mathrm{d}_{\mathcal{G}}$ *denotes the average degree throughout the sequence.*

average vertex degree after entering a dense region. Further we could also try to create heat maps indicating which regions of the graph have been visited more often than others.

5.3.5 Experiments on Real Dynamic Graphs

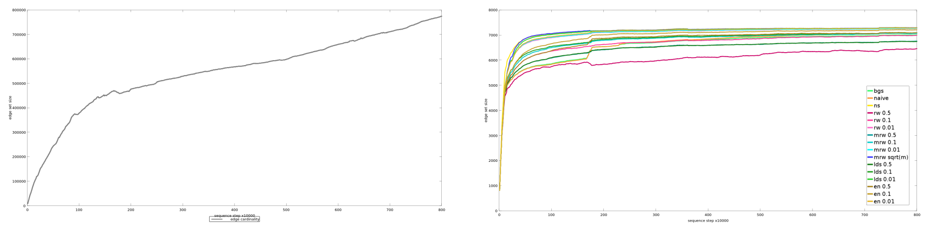
Most of the graphs we used to create test sequences were static or only incremental dynamic graphs, wherefore we used different approaches to create fully dynamic sequences as we presented in Section 5.2.1. In this section we examine the results of our algorithms when ran on real-world fully dynamic graphs and compare the results with the observations made when running them on the rather synthetic pooled-outage sequences.

The only real-world fully dynamic graphs found on KONECT are a group of graphs that represent the evolution of hyperlinks between articles on Wikipedia of different languages. As we mentioned in Section 5.1, we encountered problems with exhaustive memory consumption, which caused some experiments to fail. As a conclusion, we truncated the real input graphs in order to only process the first 250k vertices and further only the first 8M sequence steps. Unfortunately, for the graphs *itwiki*, *nlwiki* and *plwiki* the experiments still failed, wherefore we focused on the remaining experiments on the graphs *dewiki*, *frwiki* and *simplewiki*.

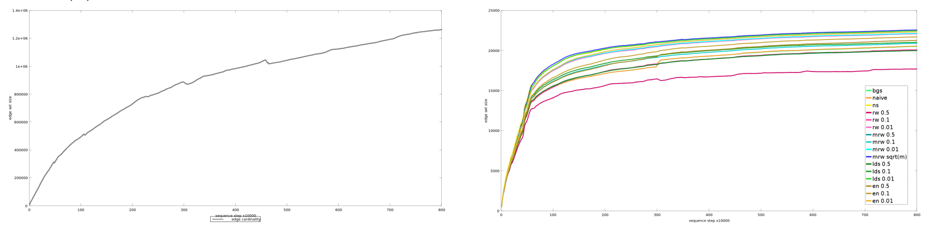
Like the results for the pooled-outage sequences in Table 3 the results in Table 4 show that the naive algorithm is mostly the fastest one. Unlike the previous results the algorithm $\sqrt{m}$**-random-walk** does achieve most often the best average matching quality.

In Figure 13a, 13b and 13c we plotted the amount of edges present in the graph as well as the matching size throughout the sequence. A main difference between the pooled-outage sequences and all of the real-world fully dynamic sequences presented here, lays in the fact that there exists no long and strictly distinguishable incremental phase in the beginning of the real dynamic sequences. Instead first edge deletions start to happen already very soon in the sequence. Throughout the whole sequence we have an overall increase of number of edges, but always also edge deletions. Furthermore we can see peaks in the number of edges after which short subsequences occur, in which edge deletions outweigh.

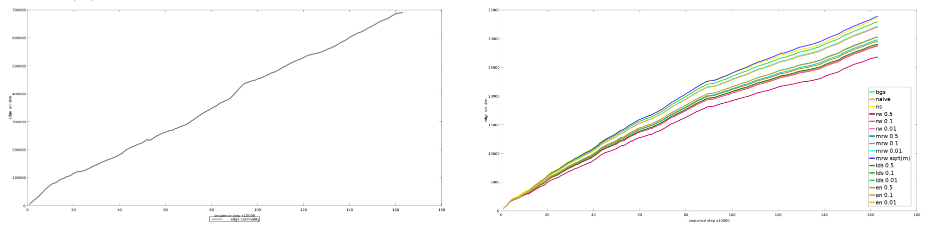
Regarding the matching size we can see that the gap in the beginning of the sequence between the result of Neiman-Solomon and the other algorithms, that we observed for pooled-outage sequences in Section 5.3.1, is not as noticeably as before. In fact having a closer look at the initial phase, shows a still noticeable



(a) *Edge cardinality and matching size for* dewiki *throughout the sequence.*



(b) *Edge cardinality and matching size for* frwiki *throughout the sequence.*



(c) *Edge cardinality and matching size for* simplewiki *throughout the sequence.*

Figure 13: *Edge cardinality and matching size for real dynamic graphs.*

improvement of Neiman-Solomon against the other algorithms, however not as distinctive as for the pooled-outage sequences.

This observation can be explained with the phenomena of matching size improvement caused by edge deletions, that we already described in Section 5.3.1. The gap in matching size between Neiman-Solomon and the other matchings results from many *unluckily* matched edges, that cause the creation of augmenting paths of length 3. With increasing length of the pure incremental subsequence, the size of the gap increase, because more and more edges are matched *unluckily.* As we concluded in Section 5.3.1, the edge deletion process of all algorithms except Neiman-Solomon can lead to an improvement of the matching size by detecting and resolving such augmenting paths of length 3. Since for our real-world fully dynamic sequences edge insertions and deletions happen in a mixed ratio from the beginning on, such *unluckily* matched edges may be detected shortly after their creation, which results in a smaller gap between the matching size from Neiman-Solomon and the other algorithms.

Another noteworthy difference to the results from Section 5.3.1 is the performance of Baswana-Gupta-Sen and the naive algorithm. For the pooled-outage sequences we observed the matching sizes of Baswana-Gupta-Sen and the naive algorithm to grow remarkably and outperform random walk algorithms with $\epsilon \geq 0.1$. This growth is only observable for the results of the *dewiki* sequence in Figure 13a, although less distinctive. Further we cannot observe this kind of growth for the results of the *frwiki* and *simplewiki* sequences.

Unlike the pooled-outage sequences the real dynamic sequences do not have a long subsequence, where edge insertions and deletions happen almost equally often. Instead we can see from Figure 13 that edge insertions overweigh the deletions overall. As we stated before, edge insertions handled in the naive manner described in Section 4.1.1 do have a probability to create augmenting paths. Therefore it does not come to the situation, where so many augmenting paths have been resolved, that the probability to find further, does grow so small, it hardly occurs and has no more significant effect on growth of the matching size, since new augmenting paths are most probably created constantly throughout the sequence.

As a conclusion to this observation, we conjecture, that a long subsequence, where edge insertions and deletions happen in an almost equal ratio, is a particular good scenario for the naive algorithm as well as the algorithm from Baswana, Gupta and Sen [17].

5.3.6 Improving Edge Insertion

In Section 5.3.1 and 5.3.5 we discussed the phenomena, that the naive insertion routine as performed by all algorithms except Neiman-Solomon (and actually also Baswana-Gupta-Sen, though the change is quite small), seem to create a lot of augmenting paths of length 3. Although this insertion routine guarantees a maximal matching, we could observe that the edge deletion routines have a significant effect on the matching size. In Figure 6 the mentioned effect is es-

Insertion	Deletion	Abbreviation	ϵ
naive-search	naive	ei-n-naive	-
combined	naive	ei-n+rw-naive	0.5
			0.1
			0.05
random-walk	extended-naive	ei-rw-en	0.5
			0.1
			0.05
combined	extended-naive	ei-n+rw-en	0.5
			0.1
			0.05

Table 5: *Algorithms which we obtained from combining an extended insertion routine with the naive and extended-naive algorithms from Section 4.*

pecially noticeable. Up until sequence step 600k, edges are only being inserted. As soon as edge deletions set in, the matching size improves drastically. Motivated by these observations we tried to improve the insertion routines of our algorithms by combining elements of the edge deletion routines.

The following two approaches apply only, when an edge can not be matched, because at least one of its endpoints are already matched. If both vertices u, v of an edge (u, v), that is being inserted, are free, we simply match the edge and end the update.

naive-search: In Figure 2 we presented how the naive edge insertion process creates augmenting paths of length 3. As a first improvement on the naive insertion process we extended routine like Neiman and Solomon do an as we explained in Example 4.4.2. Consider the scenario in Figure 2 at $t = 2$. If we add the edge (w, x) to the graph, we cannot match it, since w is already matched. We search for an augmenting path of length 3 by retrieving the mate of w, which is $v = \text{mate}(w)$ and then scanning through its neighbours for a free vertex, which can become the new mate. In this example, we find u, therefore match (u, v) and afterwards we are able to match (w, x). This process eliminates the augmenting path of length 3. If no free neighbour of v is available, we rematch (v, w). Further, if both vertices are already matched, we perform this search for an augmenting path on both vertices w, x of the edge (w, x) being inserted. Finally, when both endpoints have been handled, we check if the vertices w, x are free and if so, we match them. Retrieving the mate of w can be done in $O(1)$ time, whereas scanning through the neighbours of $v = \text{mate}(w)$ does cost $O(\deg(v)) = O(n)$ time.

random-walk: As an alternative approach we tried to improve the insertion process by adding random walks whenever an edge cannot be matched. Reconsider the example in Figure 2 at $t = 2$. Again we add the edge (w, x), but are not able to match it, because w is already matched. We retrieve the mate of w, which is $v = \text{mate}(w)$, unmatch the edge (v, w) and then start our random

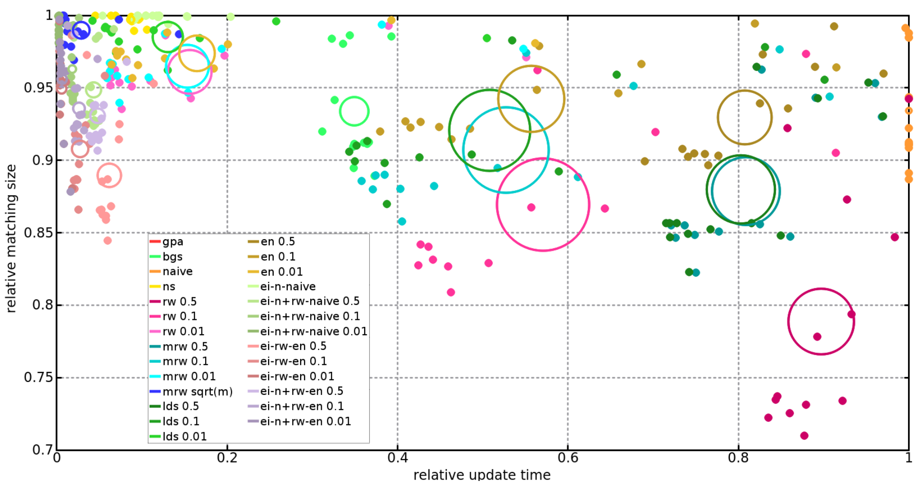


Figure 14: *Results from performing the extended insertion algorithms on the sequences from Table 3. Previous results from Figure 3 included as reference.*

walk from v. In this example, there is a 50% chance, that the random walk will lead us back to w, which is obviously not a good solution. We handle this by choosing a surrogate in case we randomly choose the vertex w, which is either the subsequent vertex or the previous one. If no surrogate exists, we act as if there was no vertex to randomly choose and end the random walk. After the random walk has finished, we check if w and x are still free and if so, we match them. As before we handle both endpoints w, x of the edge (w, x) being inserted in the same way, if both are already matched.

We used the random walk as the extended-naive algorithm uses it, which means, that at the last step of the random walk, we try to settle the last vertex naively, if we couldn't find a mate otherwise. This ensures, that the matching maintained remains maximal.

combined: A third approach consisted of simply combining the two previously mentioned approaches to improve the edge insertion routine. The naive-search approach does find an augmenting path of length 3, if inserting the respective edge will create one, however it is not capable to handle longer augmenting paths. On the contrary the random-walk algorithm has no guarantee of really finding any augmenting paths, but can detect augmenting paths of length $l < \frac{1}{\epsilon}$, where ϵ is the parameter determining the random walk length as introduced in Section 4.2. Quite similar to the extended-naive algorithm, we combined both approaches by first performing naive-search and, if naive-search was not successful, also performing random-walk afterwards.

In order to test our algorithms, we implemented the insertion routines, combined and parametrized them with our test algorithms from Section 4 as described in

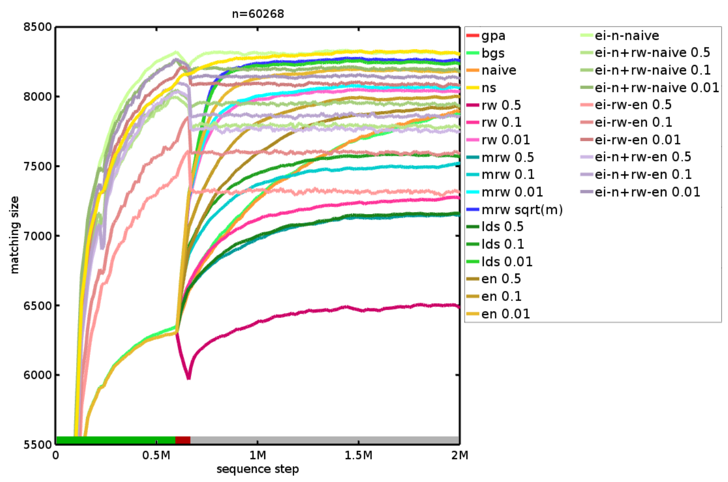


Figure 15: *Close-up at the the results from performing the algorithms from Section 4 as well as the extended inseriton algorithms on the* dewiki *sequence from Table 3.*

Table 5. We then let the algorithms run over the test sequences presented in Table 3.

In Figure 14 we present the results of our extended insertion algorithms on the sequences from Table 3. For better comparison, we also plotted the previous results from Figure 3. We see in Figure 14 that all algorithms move on the x-axis below a relative runtime of 0.2, which means the naive algorithm achieves a speedup of at least 5 over the extended insertion algorithms. It is quite obvious, that these algorithms are slower than the previously examined ones, as the extended insertion process is more time consuming than the naive inseriton process.

Regarding the size of the calculated matchings, the results do not match our expectations, as we hoped to further improve the matching size by eliminating as many augmenting paths as soon as possible. However in Figure 14 we observe, that all extended insertion algorithms which perform a random walk, do achieve worse results than most other algorithms, which have a comparable runtime. In contrary the naive-search extended insertion algorithm **ei-n-naive** without any conditional random walk does achieve the best results, not only among the extended insertion algorithms, but also when compared to the Neiman-Solomon algorithm. It achieves similarily good matching sizes, but is averagely a bit faster than Neiman-Solomon. We presume, that this slight advance is rooted in the relatively small problem size, for which we cannot observe the speedup, we would expect from the Neiman-Solomon algorithm, which has worst-case time complexity of $O(\sqrt{n+m})$.

We examined the bad performance of the approach of using random walks in the edge insertion process in order to detect and resolve augmenting paths, by studying the results of particular experiments. In Figure 15 we present the results of the extended insertion algorithms performed on the *dewiki* sequence from Table 3. We see, that all extended insertion algorithms actually achieve the task of computing better matchings than the naive insertion process during the pure incremental subsequence. During the short subsequence, where the edge pool is being created and we thus perform a pure decremental subsequence, the matching size of all extended insertion algorithms drops again. This is expectable up to some point, as probably many of the augmenting paths, which the naive insertion process does create and which lead to the jumpy improvement as described in Section 5.3.1, have been eliminated by the extended insertion. However the matching size of some of the extended insertion algorithms using random walks drop below the level, that the algorithms with naive insertion reach after the jumpy improvement. As a cause we suspect, that performing random walks during insertion and during deletion conflicts somehow. This theory is fortified by the fact, that the extended insertion variants, which use a naive-search, perform averagely better than the variants using only a random-walk. However, we have no more precise theory by hand and this issue has to be examined further.

Another odd observation is the decline in matching size from the extended insertion algorithms with random-walk during the pure incremental phase. By tracking the particular scenarios, in which such a decline happens, we found that these declines are caused by the *creation* of augmenting paths. This happens, when an edge being inserted has at least one matched endpoint. The random-walk approach does try to match this edge at any cost, hence it will make its endpoints free by unmatching the corresponding matched edges starting from the unfree endpoints, match the newly inserted edge and start random walks from the previous mates of the freed endpoints. If this random walk however does not find an augmenting path, it will end on a free vertex, which it cannot settle. In such a case, we have created a new augmenting path reaching from the last vertex of the random walk to the freed previous mate of the endpoints of our newly inserted edge.

There are however means to improve the random walk algorithms in order to prevent such situations. For example we could compare the matching size before and after performing the random walk and in case, we got a decline, undo the random walk. Further, it might be also a profitable approach to use a random walk similar to the one used in the low-degree-settle algorithm, which performs naive searches on every vertex along the random walk, which has degree below $\frac{1}{\epsilon}$ or maybe some other, more profitable vertex degree. Another interesting variation could be creating an algorithm, that performs random walks only on edge deletion and excludes augmenting paths of length 3 during insertion by performing a naive-search, as random walks in order to handle edge deletions turned out to be a profitable approach.

6 Conclusion

In this thesis we presented different random walk-based algorithms for solving dynamic maximal matching problems. We implemented these random walk algorithms along a naive algorithm and two more sophisticated algorithms from Neiman and Solomon [35] and Baswana, Gupta and Sen [17] in order to perform an extensive experimental evaluation of all these algorithms.

As a result of our experiments, we have seen that the naive algorithm outperforms all other algorithms in matters of runtime, computing also fairly good results. We have not been able to observe the algorithms from Neiman, Solomon or Baswana, Gupta and Sen achieve any speedup over the naive algorithm, although they have better runtime complexity. We assume, that this phenomena is due to the rather small problem size of the experiments we performed, which had maximally up to 250k vertices and 800k edges.

Throughout most of our experiments, we have seen that Neiman-Solomon computes the largest matchings and further we approved that this advantage comes from guaranteeing there exists no augmenting paths of length 3. We achieved approving this by extending the insertion process of the naive and extended-naive algorithms described in this paper with different approaches to search and resolve augmenting paths during edge insertion. All resulting new algorithms achieved better performances for pure incremental edge update sequences than the naive insertion process, but some approaches showed to drop in performance below algorithms, that perform the naive insertion, as soon as the sequence becomes fully dynamic. A naive algorithm combined with a naive search for augmenting paths of length 3 at edge insertion was able to outperform Neiman-Solomon in matters of times as well as sometines in matching size.

Further, we compared our results with the static Global Path Algorithm [30], [39]. Most of our algorithms did averagely compute larger matchings for the same input, however all algorithms showed to be able to compute a single edge update dynamically way faster than the static algorithm. We therefore approve experimentally, that maintaining a matching dynamically does give us a speedup over computing the matching from scratch after every edge update.

In a detailed examination of the effect of the random walk length on matching size, we discovered a correlation between the average vertex degree throughout a dynamic graph and the optimal length of a random walk in order to achieve good matching size.

6.1 Future Work

During the work on this thesis a lot of code has been created in order to perform the mentioned and descibed experiments. This code might come in handy for further research and turning it into valuable, publishable and reusable software is therefore intended.

The detailed information we gathered especially about random walks motivates further refinement, as on the one hand we showed, that random walks can be a mean to detect augmenting paths and therefore improve matching quality, on the other hand they come with different disadvantages. We presented two algorithms, which improved the basic random walk idea significantly by applying only small improvements. Further improvements might be achieved likewise. As random walks are a well researched topic, there is most likely a lot of literature to come in handy to further improve, analyse and evaluate them in our context. Also we would like to combine the random walk ideas with the more sophisticated algorithms from Baswana, Gupta and Sen, in order to try to further improve them.

As we could not get so many real-world fully dynamic graphs from the Koblenz network collection [29], we presented different approaches of creating dynamic edge update sequences from static real-world graphs. We performed most of our experiments using such an approach, which we called *pooled-outage*. However the experiments, that we where able to perform on real-world fully dynamic graphs imply, that the approach we chose does not create real-world-like sequences. Further we found that this issue has an impact especially on averaged resutls, so it might be a valuable task to improve approaches to construct more realistic dynamic graphs from static ones.

While working on this thesis, new results have been published by Kashyop and Narayanaswamy [28], which present an algorithm for fully dynamic 3/2-approximate maximum matching in $O(sqrtn)$ update time. This algorithm is basically a combination of the algorithm by Baswana, Gupta and Sen and the deterministic search for an augmenting path of length 3 as Neiman and Solomon present it and as we have applied to the naive and random walk algorithms. We will implement and evaluate this algorithm as we did throughout this work.

References

[1] Dbpedia network dataset – KONECT. `http://konect.uni-koblenz.de/networks/dbpedia-all`, Apr. 2017.

[2] Facebook friendships network dataset – KONECT. `http://konect.uni-koblenz.de/networks/facebook-wosn-links`, Apr. 2017.

[3] Flickr network dataset – KONECT. `http://konect.uni-koblenz.de/networks/flickr-growth`, Apr. 2017.

[4] Livejournal network dataset – KONECT. `http://konect.uni-koblenz.de/networks/livejournal-groupmemberships`, Apr. 2017.

[5] Orkut network dataset – KONECT. `http://konect.uni-koblenz.de/networks/orkut-groupmemberships`, Apr. 2017.

[6] Wikipedia, de (dynamic) network dataset – KONECT. `http://konect.uni-koblenz.de/networks/link-dynamic-dewiki`, Apr. 2017.

[7] Wikipedia, fr (dynamic) network dataset – KONECT. `http://konect.uni-koblenz.de/networks/link-dynamic-frwiki`, Apr. 2017.

[8] Wikipedia, it (dynamic) network dataset – KONECT. `http://konect.uni-koblenz.de/networks/link-dynamic-itwiki`, Apr. 2017.

[9] Wikipedia, nl (dynamic) network dataset – KONECT. `http://konect.uni-koblenz.de/networks/link-dynamic-nlwiki`, Apr. 2017.

[10] Wikipedia, pl (dynamic) network dataset – KONECT. `http://konect.uni-koblenz.de/networks/link-dynamic-plwiki`, Apr. 2017.

[11] Wikipedia, simple en (dynamic) network dataset – KONECT. `http://konect.uni-koblenz.de/networks/link-dynamic-plwiki`, Apr. 2017.

[12] Wiktionary (en) network dataset – KONECT. `http://konect.uni-koblenz.de/networks/edit-enwiktionary`, Apr. 2017.

[13] Wiktionary (fr) network dataset – KONECT. `http://konect.uni-koblenz.de/networks/edit-frwiktionary`, Apr. 2017.

[14] Akbarpour, M., Li, S., and Gharan, S. O. Dynamic matching market design. *CoRR abs/1402.3643* (2014).

[15] Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., and Ives, Z. DBpedia: A nucleus for a web of open data. In *Proc. Int. Semantic Web Conf.* (2008), pp. 722–735.

[16] Baccara, M., Lee, S., and Yariv, L. Optimal dynamic matching. CEPR Discussion Papers 12986, C.E.P.R. Discussion Papers, 2018.

[17] Baswana, S., Gupta, M., and Sen, S. Fully dynamic maximal matching in $O(\log n)$ update time. *CoRR abs/1103.1109* (2011).

[18] Bergamini, E., Crescenzi, P., D'Angelo, G., Meyerhenke, H., Severini, L., and Velaj, Y. Improving the betweenness centrality of a node by adding links. *CoRR abs/1702.05284* (2017).

[19] Berge, C. Two theorems in graph theory. *Proceedings of the National Academy of Sciences 43*, 9 (1957), 842–844.

[20] Bernstein, A., and Stein, C. Faster fully dynamic matchings with small approximation ratios. In *Proceedings of the Twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2016), SODA '16, Society for Industrial and Applied Mathematics, pp. 692–711.

[21] Bhattacharya, S., Chakrabarty, D., and Henzinger, M. Deterministic fully dynamic approximate vertex cover and fractional matching in $O(1)$ amortized update time. *CoRR abs/1611.00198* (2016).

[22] Demetrescu, C., Finocchi, I., and Italiano, G. F. Dynamic graphs. In *Handbook of Data Structures and Applications.*, D. P. Mehta and S. Sahni, Eds. Chapman and Hall/CRC, 2004.

[23] Edmonds, J. Paths, trees and flowers. *CANADIAN JOURNAL OF MATHEMATICS* (1965), 449–467.

[24] Gupta, M., and Khan, S. Simple dynamic algorithms for maximal independent set and other problems. *CoRR abs/1804.01823* (2018).

[25] Gupta, M., and Peng, R. Fully dynamic $(1+\epsilon)$-approximate matchings. *CoRR abs/1304.0378* (2013).

[26] Holm, J., de Lichtenberg, K., Thorup, M., and Thorup, M. Polylogarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM 48*, 4 (July 2001), 723–760.

[27] Hopcroft, J. E., and Karp, R. M. A $n^{5/2}$ algorithm for maximum matchings in bipartite. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)* (Oct 1971), pp. 122–125.

[28] Kashyop, M. J., and Narayanaswamy, N. S. Fully dynamic 3/2-approximate maximum cardinality matching in $O(\sqrt{n})$, update time. *CoRR abs/1810.01073* (2018).

[29] Kunegis, J. Konect: The koblenz network collection. In *Proceedings of the 22Nd International Conference on World Wide Web* (New York, NY, USA, 2013), WWW '13 Companion, ACM, pp. 1343–1350.

[30] Maue, J., and Sanders, P. Engineering algorithms for approximate weighted matching. In *Experimental Algorithms* (Berlin, Heidelberg, 2007), C. Demetrescu, Ed., Springer Berlin Heidelberg, pp. 242–255.

[31] Micali, S., and Vazirani, V. V. An $O(|E||V|^{1/2})$ algoithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)* (Oct 1980), pp. 17–27.

[32] Mislove, A., Koppula, H. S., Gummadi, K., Druschel, P., and Bhattacharjee, B. Growth of the Flickr social network. In *Proc. Workshop on Online Social Networks* (2008), pp. 25–30.

[33] Mislove, A., Marcon, M., Gummadi, K. P., Druschel, P., and Bhattacharjee, B. Measurement and analysis of online social networks. In *Proc. Internet Measurement Conf.* (2007).

[34] Moret, B. M. E. Towards a discipline of experimental algorithmics. In *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges, Proceedings of a DIMACS Workshop, USA, 1999* (1999), M. H. Goldwasser, D. S. Johnson, and C. C. McGeoch, Eds., vol. 59 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, DIMACS/AMS, pp. 197–214.

[35] Neiman, O., and Solomon, S. Deterministic algorithms for fully dynamic maximal matching. *CoRR abs/1207.1277* (2012).

[36] Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not. 42*, 6 (June 2007), 89–100.

[37] Preusse, J., Kunegis, J., Thimm, M., Gottron, T., and Staab, S. Structural dynamics of knowledge networks. In *Proc. Int. Conf. on Weblogs and Social Media* (2013).

[38] Sanders, P. *Algorithm Engineering – An Attempt at a Definition.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 321–340.

[39] Sanders, P., and Schulz, C. Engineering Multilevel Graph Partitioning Algorithms. In *Proceedings of the 19th European Symposium on Algorithms* (2011), vol. 6942 of *LNCS*, Springer, pp. 469–480.

[40] Sanders, P., and Schulz, C. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)* (2013), vol. 7933 of *LNCS*, Springer, pp. 164–175.

[41] Sankowski, P. Faster dynamic matchings and vertex connectivity. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2007), SODA '07, Society for Industrial and Applied Mathematics, pp. 118–126.

[42] Solomon, S. Fully dynamic maximal matching in constant update time. *CoRR abs/1604.08491* (2016).

[43] Tange, O. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine 36*, 1 (Feb. 2011), 42–47.

[44] Team, G. S. *GNU Screen: The Virtual Terminal Manager*. Samurai Media Limited, United Kingdom, 2015.

[45] Viswanath, B., Mislove, A., Cha, M., and Gummadi, K. P. On the evolution of user interaction in Facebook. In *Proc. Workshop on Online Social Networks* (2009), pp. 37–42.

[46] Wikimedia Foundation. Wikimedia downloads. `http://dumps.wikimedia.org/`, January 2010.