Dynamic Graph Algorithms with Restricted Update Sequences

Master Thesis of

Julian Schmidhuber

At the Department of Informatics and Mathematics
Chair of Theoretical Computer Science

Reviewers: Prof. Dr. Ignaz Rutter
Prof. Dr. Dirk Sudholt

Time Period: 28th April 2025 – 28th October 2025

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Passau, October 2, 2025

Abstract

Dynamic graph algorithms maintain an *underlying graph* in a sequence of *updates* and *queries* about properties of the graph. In this thesis, we consider dynamic graph algorithms where the sequence of updates is restricted. An example for such a restriction is the *stack dynamic update sequence*, where a remove operation may only remove the last added edge of the graph. We define such update sequences based on commonly used data structures, and introduce a notion of complexity comparing the different update sequences based on whether one update sequence can emulate another one. We present results that some update sequences can emulate others, as well as show negative results that this is not possible for some pairs of update sequences. This allows defining a partial order on update sequences, where some update sequences are *generalizations* of *specialized* update sequences, and some update sequences are *incomparable*.

Based on our definitions, we then discuss how specific problems can be solved in this restricted model. We investigate the connectivity, 2-edge connectivity and biconnectivity problems, and show that our algorithms can in many cases beat the current best dynamic algorithms that do not put restrictions on updates. In one case, we are able to show optimality of our algorithm. We also explore some ideas how other problems could be solved in this model.

For some update sequences, we can infer the order in which the edges of the current graph will be removed. Using this information, we define the *lifetime* of graph substructures. This allows us to define new classes of problems returning the *longest* or *shortest lifetime substructures* of the current underlying graph. This includes for example returning the longest lifetime path between any two vertices, or returning the longest lifetime triangle of the graph. We are able to solve some of these problems, both based on previous results in the thesis and results given in related work in the area of dynamic subgraph counting.

Deutsche Zusammenfassung

Dynamische Graphalgorithmen verwalten einen *zugrundeliegenden Graphen* während einer Sequenz von *Änderungen* und *Anfragen* über Eigenschaften des Graphen. In dieser Arbeit betrachten wir dynamische Graphalgorithmen bei denen die Reihenfolge der Änderungen eingeschränkt ist. Ein Beispiel für solch eine Einschränkung sind *Stapel-dynamische Änderungssequenzen*, bei denen die Löschung einer Kante immer die zuletzt hinzugefügte Kante löscht. Wir definieren solche Änderungssequenzen basierend auf häufig verwendeten Datenstrukturen, und führen den Begriff der *Komplexität* ein, um verschiedene Änderungssequenzen, basierend auf der Fähigkeit andere Änderungssequenzen zu emulieren, zu vergleichen. Wir zeigen, dass solch eine Emulierung bei manchen Paaren von Änderungssequenzen möglich ist, aber zeigen auch Resultate, dass dies bei manchen Änderungssequenzen nicht möglich ist. Dadurch können wir eine partielle Ordnung von Änderungssequenzen definieren, bei denen es *generellere* und *spezialisierte* Sequenzen gibt, und bei denen manche Paare *unvergleichbar* sind.

Basierend auf unseren Definitionen betrachten wir wie verschiedene Probleme in diesen eingeschränkten Modell gelöst werden können. Wir betrachten Konnektivität, 2-Kanten-Konnektivität und Bikonnektivität, und zeigen, dass unsere Algorithmen in vielen Fällen die besten Algorithmen schlagen können in denen keine solchen

Einschränkungen angenommen werden. In einem Fall können wir zeigen, dass unser Algorithmus optimal ist. Wir diskutieren außerdem Ideen wie andere Probleme in diesem Modell gelöst werden könnten.

Für manche Änderungssequenzen können wir die Reihenfolge, in welcher die Kanten des aktuellen Graphen entfernt werden, folgern. Basierend auf dieser Information definieren wir die *Lebensdauer* von Graphstrukturen. Dies erlaubt uns eine neue Klasse von Problemen zu definieren, welche die Graphstrukturen mit längster oder kürzester Lebensdauer des zugrundeliegenden Graphen zurückgeben. Solch ein Problem ist beispielsweise das Auffinden des Pfades zwischen zwei Knoten mit längster Lebensdauer, oder das Finden des Dreiecks mit längster Lebensdauer. Wir können einige solcher Probleme lösen, basierend sowohl auf vorhergehenden Resultaten dieser Arbeit als auch Resultaten in verwandten Arbeiten bezüglich dynamischer Subgraphenzählung.

Contents

1. Introduction

Dynamic graph algorithms maintain an *underlying graph* in a sequence of online *updates*, like edge insertion and deletion, as well as answer *queries* about properties of the current underlying graph, for example whether two vertices are connected. Updates and queries are collectively called *operations*. Dynamic graph algorithms are an important tool in practice, for example a survey by Sahu et al. [SMS+20] showed that about 65% of graphs from the surveyed researchers and practitioners changed over time. Example applications of dynamic graphs include communication networks, graphics or assembly planning [EGI99]. This interest in dynamic graphs can also be seen in research, like in the survey by Hanauer et al. [HHS22] listing recent work for connectivity, reachability, shortest path and many other problems.

Dynamic graph algorithms can always be implemented by maintaining the underlying graph, and for every query recomputing the answer from scratch. This leads to a constant *update time* and the same *query time* as the runtime of the fastest static algorithm. It is also often possible to implement dynamic graph algorithms such that the solutions to all possible queries are precomputed with an update. For connectivity, this means computing the connected components for an update, and then for queries looking up whether the two vertices are in the same connected component in constant time. Using these two trivial methods of creating dynamic graph algorithms, one can therefore get a dynamic graph algorithm for connectivity with $O(1)$ update and $O(n+m)$ query time, and another algorithm with $O(n+m)$ update time and $O(1)$ query time. Depending on the ratio between updates and queries, one of the algorithms may be preferred over the other. But if there are approximately as many updates as queries, both algorithms still have a linear runtime per operation. The research in dynamic graph algorithms therefore tries to find intermediate algorithms, ideally algorithms on the Pareto front of the update and query time. As an example for connectivity, Huang et al. [HHKP17] have given an algorithm that supports insertion and deletion of edges that has expected amortized $O(\log n(\log\log n)^2)$ update and $O(\log n/\log\log\log n)$ query time, which is currently the best algorithm in regard to the runtime per operation for the same number of updates and queries of the dynamic graph. Further algorithms for the connectivity problem with different tradeoffs for update and query times are described in Chapter 5.

The current research on dynamic graph algorithms mainly differentiates between three kinds of algorithms: *Fully dynamic* algorithms support both the insertion and deletion of single edges as updates, while *incremental* or *decremental* algorithms support only insertion or only deletion of single edges, respectively. While incremental and decremental algorithms

are less powerful compared to fully dynamic algorithms, they often significantly improve on the update and query time. As an example, incremental connectivity can be solved using the union-find data structure in amortized $O(\alpha(n))$ time per operation [EGI99], which is impossible to achieve for fully dynamic algorithms [PD05]. There are also some other models of updates used in the literature, for example graphs in the sliding window model or algorithms for emergency planning, which are discussed in further detail in Chapter 3.

In this thesis, we investigate restricting the sequence of updates for dynamic graphs. This can be seen as an intermediate concept between fully dynamic and incremental or decremental algorithms. We consider for example dynamic graphs where edge deletion operations always remove the last inserted edge from the graph. We call this a *stack dynamic update sequence*. Analogously, the *queue dynamic update sequence* always removes the edge that was inserted first into the graph. As an example, consider the sequence of updates that first inserts the edge e_1, then e_2, and finally removes an edge. One can interpret this sequence of updates as both a stack and queue dynamic update sequence. The edge that is removed is in both cases uniquely defined: For the stack dynamic update sequence, the removed edge is e_2, while in the queue dynamic update sequence it is e_1. Another update sequence, which we call the *priority queue dynamic update sequence*, adds a parameter to every edge when it is inserted specifying the order in which the edges will be removed from the graph. This thesis formally defines such restricted update sequences on the basis of commonly used data structures. We show that some update sequences can emulate others, such that we call some update sequences a *generalization* of *specialized* update sequences. Furthermore, we also prove that this is not possible for some pairs of update sequences, making these update sequences *incomparable*. This allows us to define a partial order on update sequences, indicating the complexity of different update sequences. We also discuss how different problems can be solved for such restricted update sequences, like connectivity, 2-edge connectivity and biconnectivity. We show that restricting the sequence of updates can lead to algorithms that beat the respective fully dynamic algorithms. As an example, we give a priority queue dynamic algorithm for the connectivity problem in $O(\log n)$ update and query time. Compared to the fully dynamic algorithm by Huang et al. [HHKP17], our algorithm therefore improves on the update time, while having a slightly worse query time. This priority queue dynamic connectivity algorithm is also shown to be optimal.

With the additional information about the order in which edges will be removed for the priority queue dynamic update sequence, we furthermore define a new class of problems, which we call *lifetime problems*. These problems are dynamic graph problems where the *longest lifetime* or *shortest lifetime substructures* of the current underlying graph needs to be returned. Examples of such problems include returning the longest lifetime path between any two vertices, or returning the longest lifetime triangle in the graph. We can solve some lifetime problems with the tools developed in this thesis, as well as related work done in the area of subgraph counting.

This thesis is structured as follows: Chapter 2 gives the prerequisite knowledge required to understand the thesis. Chapter 3 introduces some related work regarding dynamic graph models. Related work specific to certain dynamic graph problems is spread throughout the thesis when we discuss the problems in our model as well as when we consider future work for other dynamic graph problems with restricted update sequences. Chapter 4 formally defines our notion of restricted update sequences. It furthermore gives the concept of complexity for the different update sequences. We show how different update sequences can be compared in regard to their complexity, as well as show that some update sequences are incomparable in this model. This chapter also contains some simple initial results that follow directly from related work. With Chapter 5, we start to study how restricted update sequences can be used to solve dynamic problems, specifically in this chapter the connectivity problem. We show that it is possible to implement an $O(\log n)$ update and

query time algorithm for many restricted update sequences, beating the time per operation of the current best algorithm with amortized expected $O(\log n(\log\log n)^2)$ update time and $O(\log n/\log\log\log n)$ query time for fully dynamic update sequences [HHKP17]. We also give a proof that our algorithm for the priority queue dynamic update sequence is optimal. To end this chapter, we also give results for other problems that directly follow from our improvements from the connectivity problem. In Chapter 6, we consider the 2-edge connectivity and biconnectivity problems. We again show that it is possible to improve the runtime per operation compared to the current best fully dynamic algorithms for these problems for some restricted runtime sequences. Chapter 7 defines lifetime problems for update sequences where we know in which order the edges will be deleted. We show how our results from Chapter 5 can be used to solve such problems, and give further results based on dynamic substructure counting as found in the literature. Chapter 8 gives inspiration for future work to be done in the area of restricted dynamic update sequences. Chapter 9 concludes this thesis.

2. Prerequisites

In this section, we give the prerequisite knowledge required to understand this thesis. This thesis assumes familiarity with the basic graph-theoretic definitions, like graphs, connected components or spanning forests. Let $G = (V, E)$ be a graph, $n = |V|$, $m = |E|$. The graphs considered in this thesis are in all cases simple, which means they do not contain self-loops and have at most one edge between pairs of distinct vertices. If F is a spanning forest of G, denote the path between two connected vertices $u, v \in V$ in F as $\pi_F(u, v)$. If the spanning forest is clear from the context, we simply write $\pi(u, v)$. For non-forest edges $e = uv \in E \setminus F$, we also write $\pi(e)$ for $\pi(u, v)$. For $l \in \mathbb{N}$, write $[l]$ for $\{1, \ldots, l\}$. We note that all of our runtimes assume the WORD-RAM computation model with $O(\log n)$ word size. This in particular means that one can store references to vertices and edges in constant time and space. Furthermore, one can do arithmetic operations on integers with an absolute value bounded by a polynomial function in n in constant time.

To start with, we give an informal definition of dynamic graphs. A more detailed definition is given in Chapter 4. As already described, dynamic graphs maintain an underlying graph under a series of updates and queries. The series of operations is given online, which means that the data structure does not have knowledge about which operations will be executed in the future. In this model, one usually differentiates between the time required to update the data structure, and the time required to answer a query. For example the current best fully dynamic connectivity algorithm takes an amortized, expected $O(\log n(\log \log n)^2)$ time to update the data structure, and $O(\log n / \log \log \log n)$ time to query whether two vertices are connected [HHKP17]. In this thesis, fully dynamic updates are the insertion and deletion of a single edge in the underlying graph. Other notions of updates are also possible, for example inserting a set of edges all sharing one endpoint and removing an arbitrary set of edges [Rod03, RZ04], or inserting and removing hyperedges into hypergraphs [BCH17]. In many cases, it is also possible to extend the dynamic algorithms to allow for inserting or deleting vertices with no incident edges [EGI99, HRS92, ES09, GK18]. The type of queries supported by dynamic graphs depends on the specific problem. A query may have parameters, and return a value. The type of parameters and returned values depend on the problem. A query could for example ask whether two vertices of the underlying graph are connected.

Unless mentioned otherwise, this thesis assumes that the initial graph is the empty graph with n vertices and no edges. Due to this assumption, it is possible to initialize the dynamic

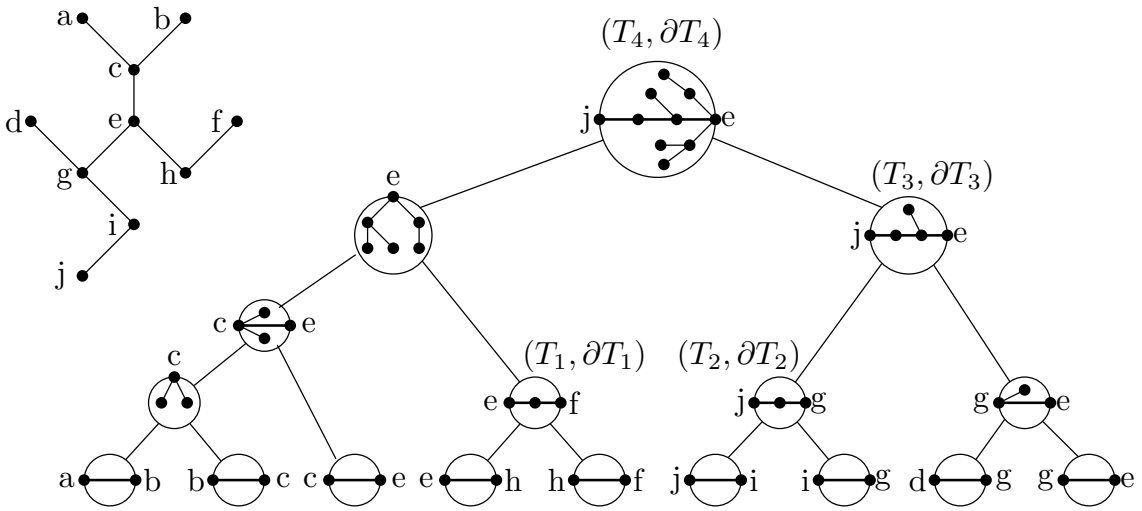

Figure 2.1: A tree and a possible representation as a top tree. Circular nodes represent clusters that represent the subtree depicted in the node and have boundary vertices labelled beside the cluster. Cluster paths for path clusters are drawn bold.

graph data structure in time linear in its required space. This thesis therefore ignores preprocessing time, and focuses on the update and query time of the dynamic graphs. If the initial given graph is non-empty, one can simply build up the underlying graph by executing a sequence of edge insertions. This may lead to a worse runtime compared to a dedicated preprocessing algorithm.

We now introduce top trees, as given by Alstrup et al. [AHLT05]. A top tree is a representation of a dynamic tree, allowing for efficiently merging and splitting trees, as well as maintaining information about the trees. We only give a coarse definition of top trees, for more details refer to the introduction by Alstrup et al. [AHLT05]. An example tree and one possible representation as a top tree is given in Figure 2.1.

Let F be a forest. A top tree is a binary tree of *clusters*, which are pairs $(T, \partial T)$ of subtrees T of F, and at most two *boundary vertices* ∂T contained in T. Leaf clusters of the top tree with root $(T, \partial T)$ then correspond to the edges in T. Furthermore, sibling clusters intersect in exactly one vertex, which is a boundary vertex of both sibling clusters. The subtree of non-leaf clusters is the union of the subtrees of the children, and its boundary vertices are a subset of the union of the boundary vertices of the children. We call $(T, \partial T)$ a *path cluster* if $|\partial T| = 2$, and a *point cluster* if $|\partial T| = 1$. If u and v are the boundary vertices of a path cluster, we call $\pi(u, v)$ the *cluster path*.

Two top trees can be joined by inserting an edge, and split into two by removing an edge. Furthermore, top trees support an expose(u, v) operation, where u and v are vertices of the underlying tree that modifies the top tree such that u and v are boundary vertices of the root cluster. It is shown by Alstrup et al. [AHLT05] that all these operations can be implemented in worst-case logarithmic time. Some implementations of top trees, like splay top trees by Holm et al. [HRR23], only have an amortized logarithmic runtime.

Information about trees is then stored inside clusters. As shown by Alstrup et al. [AHLT05], it suffices to describe how this information can be updated when splitting and joining clusters. Joining and splitting top trees as well as exposing nodes can then be implemented by logarithmically many join and split operations of top tree clusters.

We now give the results regarding top trees required for this thesis. Most of these results were directly given by Alstrup et al. [AHLT05].

Lemma 2.1. *Top trees support the following operations in $O(\log n)$ time each:*

1. *Maintain a dynamic forest in regard to edge insertion and deletion.*
2. *Test whether two vertices are in the same tree.*
3. *Find the maximum weight edge on the path between two vertices.*
4. *Add a common weight to all edges on the path between two vertices.*
5. *For two vertices and a weight, set the weight of all edges on the path between the vertices to the maximum of its previous weight and the given weight.*

Furthermore, the following operations can be implemented:

6. *Return all edges on the path between two vertices in $O(\log n + l)$ time, where l is the number of returned edges.*
7. *Return the j-th vertex on the path between two vertices in $O(\log n + j)$ time.*
8. *Return all edges contained in the forest in $O(l)$ time, where l is the number of returned edges.*

Proof. Use top trees as described above and by Alstrup et al. [AHLT05]. Operations 1–4 where directly shown by Alstrup et al. [AHLT05], operation 5 was given by Holm et al. [HdLT98]. We therefore only show operations 6–8.

We start with the operations 6 and 7. We show that we can maintain for each path cluster a doubly linked list of the vertices in the cluster path, excluding the boundary vertices. The path between two vertices can then be returned by exposing the two vertices in logarithmic time, and then walking along the linked list and the two boundary vertices. Querying the vertex with a specific offset can easily be done by only walking the specified distance. We note that this linked list is shared between path clusters in order to allow for an efficient implementation of joining and splitting clusters. Furthermore, each vertex is contained in at most one linked list. A vertex can therefore keep a reference to its position in a linked list. As an example, note that in the top tree in Figure 2.1 there are four clusters that have at least one intermediate vertex in its cluster path; these have been given the names $(T_1, \partial T_1)$ to $(T_4, \partial T_4)$. By sharing the linked list between path clusters, this leads to two linked lists: One containing only h, which is due to the cluster of the subtree $(T_1, \partial T_1)$, and one containing ig, which is shared by the clusters $(T_2, \partial T_2)$, $(T_3, \partial T_3)$ and $(T_4, \partial T_4)$.

We now describe how clusters are joined and split in constant time. By Alstrup et al. [AHLT05], this already suffices to give an algorithm with logarithmic runtime for maintaining the cluster path for every path cluster.

When joining two path clusters at a shared boundary vertex that is not a boundary vertex of the resulting cluster, add the shared boundary vertex in between the linked lists of both child clusters. To join two path clusters together where the boundary vertices of the resulting clusters are the same as the boundary vertices of one of the children, the cluster path is the same as the one of that child. To join one path and a point cluster to a path cluster, the resulting cluster path is just the cluster path of the path cluster child. When joining clusters to a point cluster nothing needs to be done. It is easy to see the correctness of this algorithm, as well as the constant runtime.

To split a path cluster with two path cluster children at a common boundary vertex that is not a boundary vertex of the parent cluster, remove this common vertex from the linked list producing two linked lists. These linked lists correspond to the path clusters of the two children. To split a path cluster into two path clusters when there exists a child with the same boundary vertices as the parent, the cluster path of that child is the same as the cluster path of the parent. When splitting a path cluster into a path and a point cluster, the cluster path of the resulting path cluster is the same as the cluster path of the split cluster. If splitting a point cluster, nothing needs to be done. It is again easy to see correctness and the constant runtime.

We now consider operation 8. As mentioned above, the leafs of a top tree correspond to the edges of the underlying tree. One can therefore list the leafs of a top tree using a simple depth-first search. As the top tree is a binary tree, it therefore contains more leafs than internal nodes. Therefore, the runtime of the traversal is linear in the number of edges returned. □

We also note that many of these operations can alternatively be implemented using dynamic trees introduced by Sleator and Tarjan [ST81].

3. Related Work

In this section, we discuss related work for dynamic graphs. We also compare their models with restricted update sequence we investigate in this thesis.

We have already introduced dynamic graphs in Chapters 1 and 2. Refer to the survey by Hanauer et al. [HHS22] for more details about dynamic graphs. We introduce related work specific to problems in Chapters 5–7 when we investigate these problems in our model. Furthermore, we give related work for other problems in Chapter 8, where we investigate other problems restricted dynamic update sequences could be applied to. The model of restricted update sequences this thesis investigates is a special case of dynamic graphs. This allows our algorithm to have a faster runtime, while only being applicable in certain contexts.

An already commonly used restriction of dynamic graphs is the sliding window model. Here, a sequence of edges is given online, and the underlying graph is built from the constant number of edges last inserted. Examples of using this model are given by Crouch et al. [CMS13]. Note that they are interested in minimizing the memory usage, while this thesis focuses on runtime. We show in Chapter 4 that this restriction of dynamic graphs can also be modelled by us, and show that the queue dynamic update sequence can emulate dynamic graphs in the sliding window model.

Another restriction was introduced by Peng and Rubinstein [PR23], the deletions-look-ahead model. In this model, the order of deletions among the edges in the current underlying graph is known. They in particular showed a reduction from incremental algorithms to algorithms in the deletions-look-ahead model. We describe this reduction in Chapter 4, and show that their model corresponds to our priority queue dynamic update sequence. We apply their reduction a few times in this thesis, but show that specialized priority queue dynamic algorithms can do better in many cases.

Offline dynamic graphs assume that the entire sequence of updates is already given in advance, while for ordinary dynamic graphs the sequence of operations is given online. This is done for example by Eppstein [Epp94], who compute the MSTs for a sequence of graphs under weight change operations as well as edge insertions and deletions. For a sequence of k updates, they show how to compute the minimum spanning forest for all graphs in $O(k \log n)$ time. We show in Chapter 4 that our model of priority queue dynamic update sequences generalizes offline dynamic graphs. The changes to edge weights can also be done in a continuous way, like done for kinetic problems. This includes research done by Agarwal et al. [AEGH98], who investigate computing the sequence of minimum spanning

forests of the graph where edge weights change continuously. We won't be able to model these continuous changes in this thesis. In this offline model, it can also make sense to query whether a property holds for every graph in a specified range, or for at least one graph. This is done in graph timelines like investigated by Lacki and Sankowski [LS13], who gave an algorithm that can check whether two vertices are connected in any or all graphs in an interval. We also won't model this in the thesis.

In contrast to dynamic graphs, which only allow a single edge to be inserted and deleted at a time, batch dynamic graph algorithms allow inserting and deleting multiple edges at once, as well as batching multiple query operations. This is done for example by Acar et al. [AABD19], who investigate connectivity in a parallelized, batch dynamic setting. Algorithms for the parallel setting are given as a thread directed acyclic graph (DAG), where each vertex represents an instruction. An instruction can only run if all its predecessors finished running. For the runtime, one then differentiates between the *work*, which is the number of vertices, and the *depth*, which is the length of the longest path in the DAG. They then give an algorithm that updates in $O(\log n \log(1 + \frac{n}{\Delta}))$ expected amortized work per edge insertion and deletion and $O((\log n)^3)$ depth with high probability, where Δ is the average batch size of all deletions. It answers a batch of k queries in $O(k \log(1 + \frac{n}{k}))$ work and $O(\log n)$ depth with high probability. While our modelling approach of restricted update sequences introduced in Chapter 4 could also be used in the batch dynamic setting, this is not investigated in this thesis. Restricted batch dynamic updates could be further investigated in future work as noted in Chapter 8.

Related to batch dynamic algorithms are also algorithms for emergency planning. Emergency planning is used for example by Henzinger and Neumann [HN16], who investigate connectivity in this model. In their paper, each vertex of the graph is either turned on or off. An update then changes the state of up to d vertices, after which connectivity queries are asked in the subgraph of active vertices. After this update, the graph may not be updated again. If one would want to update the graph again, one needs to reinitialize the entire data structure. For any $c \in \mathbb{N}$, the algorithm presented by Henzinger and Neumann [HN16] can solve connectivity in the emergency planning scenario in $\tilde{O}(n_{\text{off}}^2 d^{1-\frac{2}{c}} m_{\text{on}} n_{\text{on}}^{\frac{1}{c}-\frac{1}{c\log(2d)}} (\log n_{\text{on}})^2)$ preprocessing time, $O(d^{2c+6}(\log n_{\text{on}}^2)\log\log n_{\text{on}})$ update time and $O(d^2)$ query time, where n_{on}, n_{off} and m_{on} are the number of vertices turned on, off and edges turned on in the initial graph respectively. This model could be used for example by an internet service provider, where the vertices are hubs which can communicate with each other. An emergency can then turn off some of the hubs, which can potentially be mitigated by turning on backup hubs. The algorithm for connectivity in emergency planning can then be used to find out which hubs can still communicate with each other after such an emergency. This restriction to batch dynamic algorithms can also be modelled by us, as mentioned in Chapter 4.

Another specialization of dynamic graphs investigated in the literature includes dynamic graph algorithms with lookahead, like considered by Sankowski and Mucha [SM10] for transitive closure. In this model, for some $\varepsilon > 0$, the algorithm has access to the next n^ε updates or operations executed on the dynamic graph. Their algorithm can then answer reachability queries in a directed graph in $O(n^{\omega(1,1,\varepsilon)-\varepsilon})$ time per operation with one-sided error. Here, $\omega(1, 1, \varepsilon)$ is the exponent of the runtime required for the multiplication of an $n \times n$ with an $n \times n^\varepsilon$ matrix. We are not able to model this in our restricted dynamic graphs.

Dynamic graph algorithms can also be investigated with predictions, like done by van den Brand et al. [vdBFNP24]. These predictions include information about future updates, for example the time at which an edge is inserted or removed from the graph. Note that, unlike restricted dynamic graphs investigated in this thesis, these predictions may be wrong. This

means that the runtime of their algorithms is in particular parameterized by the error the prediction makes. As an example, they have given incremental and decremental algorithms for the all-pairs shortest paths problem that handles updates in $\tilde{O}(1)$ time and queries in $\tilde{O}(\eta^2)$ time, where η is the error measure bounded by the maximum time between the predicted and actual insertion or deletion of any edge. Depending on the error of the prediction, this algorithm can therefore beat fully dynamic connectivity algorithms, but may also be worse if the quality of the predictions is low. We are not able to model this uncertainty in our restricted dynamic graphs.

An orthogonal concept to dynamic graphs are temporal graphs [WCH$^+$14, CFQS10, ES22, Mic15, HFL15, NTM$^+$12, TSM$^+$10]. Here, a sequence of graphs sharing the same vertex set is given in one data structure, and one wants to query overall properties for the entire data structure. Examples include whether a message can be routed between vertices while maintaining temporal constraints. In a simplified view, this means that every graph in the sequence is active for exactly one timestamp and taking an edge proceeds time by one timestamp. A path between two vertices then takes an edge in the first graph in the first timestamp, an edge in the second graph in the second timestamp and so on until the destination is reached. This model can also be made more complex by specifying for every graph in the sequence how long it is active for, and annotating for each edge how much time it takes to pass through the edge. One can also allow waiting some time at a vertex, until for example a desired edge exists. One can also consider different kinds of temporal paths, like the earliest-arrival path, latest-departure path, fastest path or shortest path [WCH$^+$14]. We do not model this concept in this thesis.

4. Update Sequences

This thesis focuses on restricting the sequence of updates possible for dynamic graph algorithms. In that regard, the sequences referenced in this thesis are now formally introduced. We also show how the update sequences relate to each other in terms of their complexity based on one update sequence being able to emulate another. An overview of the most important update sequences and how they can emulate each other is given in Figure 4.1. We furthermore give a few initial results by interpreting commonly used models from the literature in our model, and give simple algorithms for problems in the context of the restricted update sequences that directly follow from related work.

4.1 Formalization

Informally, an update sequence specifies how the underlying graph of a dynamic graph algorithm over a fixed set of vertices V changes over time. We start by describing update sequences commonly used in the literature. Afterwards, we give a formalization of dynamic update sequences. It is then shown how this formalization can be applied to already existing update sequences, and we furthermore use the formalization to give new examples on restricted update sequences.

A fully dynamic update sequence allows for an arbitrary ordering of edge insertions and deletions. An edge may only be inserted if it is not currently in the underlying graph, and deleted only if it is currently in the underlying graph. This is the most general update sequence given in this thesis. It is also the most commonly investigated update sequences in the literature.

Other commonly used restricted versions of the fully dynamic update sequence include the incremental update sequence, which only allows for insertion of edges, and the decremental update sequence, which only allows deletion of edges. The decremental update sequence assumes that the initial graph is the complete graph with n vertices. Both of these sequences are necessarily bounded in length by $\frac{n(n-1)}{2}$ for undirected and $n(n-1)$ for directed underlying simple graphs.

This thesis now introduces other restricted update sequences, based on common data structures for maintaining sets of elements. In the previous definitions it was handy to describe the difference between the two underlying graphs due to an update. Now, the underlying graph is described directly by the set of edges contained within it.

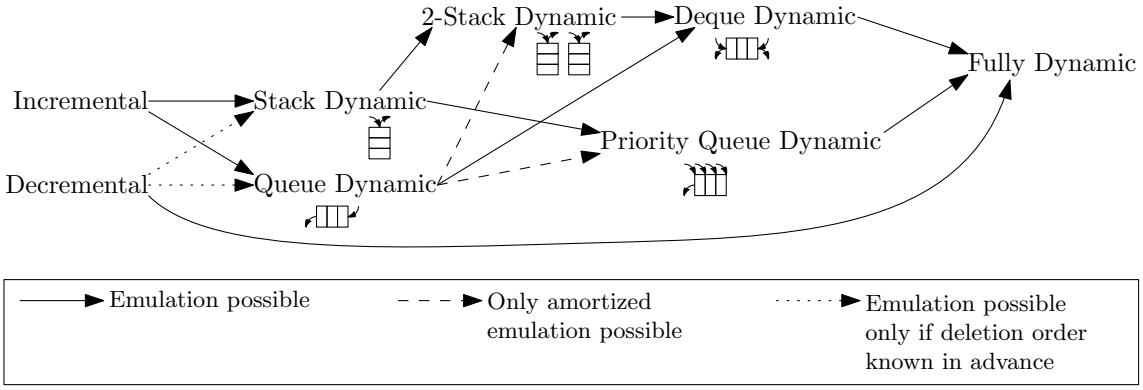

Figure 4.1: Important update sequences and the transitive reduction on how they can emulate each other, assuming that the runtime of the emulation is bounded. A dashed line signifies an amortized emulation where no worst-case emulation is possible. A dotted line signifies the emulation only holds if the deletion order of the decremental graph is known in advance. There may be further amortized-only emulations missing.

Definition 4.1. *Let $\mathcal{D}$ be a data structure containing a set of edges, where $D \in \mathcal{D}$ is an instance of the data structure containing edges $E(D)$. Let $\mathcal{O}$ be the set of update operations allowed for the data structure, where every* $\mathrm{op} \in \mathcal{O}$ *is a function* $\mathrm{op} : \mathcal{D} \times \mathcal{P} \to \mathcal{D} \cup \{\perp\}$. *Here, $\mathcal{P}$ is an arbitrary set of parameters, and* $\mathrm{op}(D, P) = \perp$ *if the update cannot be executed with the given instance and parameter. Assume a $D_0 \in \mathcal{D}$ exists and is unique with $E(D_0) = \emptyset$.*

Let $\mathrm{op}_i \in \mathcal{O}$, $P_i \in \mathcal{P}$ *for* $i \in [l]$ *and* $l \in \mathbb{N}$. *Let* $D_i = \mathrm{op}_i(D_{i-1}, P_i)$ *if* $D_{i-1} \neq \perp$, *and* $D_i = \perp$ *if* $D_{i-1} = \perp$. *We call* $\mathrm{op}_1, \ldots, \mathrm{op}_l$ *with the respective parameters* $P_1, \ldots, P_l$ *a* valid $\mathcal{D}$-dynamic update sequence *if* $D_l \neq \perp$. *After the update* op_i, *the* underlying graph *is* $(V, E(D_i))$.

Note that this definition does not restrict how large the change done by a single update operation is. But in all cases this thesis considers, at most a constant number of edges is added and removed from the underlying graph in a single update. It is also easy to generalize this definition to allow for other changes done to the graph, for example inserting or removing vertices. In most cases, this thesis assumes that the sequence of updates is valid and omits mentioning that it is valid. Furthermore, the update operations are usually only defined for a subset of parameters based on the current edge set, operations with different parameters return $\perp$. This also allows defining a separate parameter set for different operations. We also allow operations to not take any arguments, which can formally be done by allowing any parameter but ignoring it for the computation. In most cases, dynamic graph algorithm running on a restricted update sequence need to maintain the current data structure instance itself to know the set of edges contained in the underlying graph. If this is the case, this needs to be included in the time required to update the dynamic graph data structure. We call dynamic graph algorithms that accept a $\mathcal{D}$-dynamic update sequence *$\mathcal{D}$-dynamic algorithms*, analogous to what is usually done in the literature for fully dynamic, incremental or decremental algorithms.

As an example, we first show how to apply this definition to the *fully dynamic update sequence* for undirected underlying graphs. The data structure for the fully dynamic update sequence is the set of edges itself. In that case $\mathcal{D} = 2^{\binom{V}{2}}$ corresponds to all graphs with

vertex set V. An instance $D \in \mathcal{D}$ corresponds to the graph with edges $E(D) = D$. The fully dynamic update sequence allows for insertion and deletion of edges. These can be modelled as follows:

$$\text{insert}(D, e) = \begin{cases} D \cup \{e\} & \text{if } e \notin D \\ \perp & \text{otherwise} \end{cases}$$
$$\text{remove}(D, e) = \begin{cases} D \setminus \{e\} & \text{if } e \in D \\ \perp & \text{otherwise} \end{cases}$$

Then $\mathcal{O} = \{\text{insert}, \text{remove}\}$ with $\mathcal{P} = \binom{V}{2}$. From this definition, the definitions of the incremental and decremental update sequences directly follow: The *incremental update sequence* has $\mathcal{O} = \{\text{insert}\}$, while the *decremental update sequence* has $\mathcal{O} = \{\text{remove}\}$ with the additional change that $D_0 = \binom{V}{2}$.

This model is now directly used to define update sequences that are used throughout this thesis. We start with defining the *stack dynamic update sequence* as an initial example for a restricted update sequence. A stack is a data structure that allows adding an element using the push operation, and removing the most recently added element using the pop operation. The underlying data structure for the stack dynamic update sequence is the stack, which means that $\mathcal{D}$ is the set of all stacks containing edges. This update sequence supports the update operations $\mathcal{O} = \{\text{insert}, \text{remove}\}$, with $\mathcal{P} = \binom{V}{2}$, defined as follows:

$$\text{insert}(D, e) = \begin{cases} \text{push}(D, e) & \text{if } e \notin D \\ \perp & \text{otherwise} \end{cases}$$
$$\text{remove}(D) = \begin{cases} \text{pop}(D) & \text{if } E(D) \neq \emptyset \\ \perp & \text{otherwise} \end{cases}$$

Note that the remove operation does not have a parameter. This is as the edge that is removed is uniquely defined as the topmost edge of the stack. As mentioned in the discussion about the definition of $\mathcal{D}$-dynamic update sequences, this is just a shorthand notation for allowing any edge as a parameter but ignoring the parameter. Alternatively, the remove operation could also be modelled to take an edge as a parameter; if the given edge is the topmost edge of the stack, this edge is removed, otherwise $\perp$ needs to be returned. It is however easy to see that this alternative model does not change the difficulty of the update sequence: One can convert an algorithm solving some stack dynamic problem for the original model to one solving the modified model by first checking the topmost edge of the stack, and either removing it or returning $\perp$ depending on the edge given as a parameter. In fact, using the tools developed in the next section, one could easily proof that the two models of the stack dynamic update sequence are equivalent. We omit the details of the proof.

Analogously, the *queue dynamic update sequence* can be defined. This allows pushing an edge, which is not yet in the queue, to the end of the queue. Removing an edge always happens at the front of the queue. One can furthermore define the *priority queue dynamic update sequence*, where edges have a priority and removal always removes the lowest priority edge. More formally, insertion of edges uv requires an additional parameter t_{uv}. Based on

this parameter, the data structure maintains an injective mapping $\tau : E \to \mathbb{R}, uv \mapsto t_{uv}$. We call $\tau(uv)$ the *lifetime* of uv. The remove operation then removes the edge with minimum lifetime. It is important that the mapping τ is injective, otherwise the remove operation would not be uniquely defined. This in particular means that insertions with additional parameter t_{uv} should be rejected if that means that the mapping would not be injective anymore.

It is easy to see that all incremental update sequences are also valid stack and queue dynamic update sequences. If the deletion order of the edges is known in advance, the same also holds for the decremental update sequence by correctly initializing the stack and queue. Intuitively, the priority queue dynamic update sequence is a generalization of both the stack and queue dynamic update sequences, as a priority queue can emulate both a stack and a queue by choosing an appropriate lifetime parameter for insertions. We formalize this intuition in Section 4.2.

The last of the basic update sequences to introduce is the *deque dynamic update sequence*. In that update sequence, edges may be inserted or removed at two ends of a deque. This is done by having two parameters for insertion, and one for removal: One parameter for insertion specifying the edge to insert, and a separate parameter for insertion and removal specifying at which end the deque is modified. Intuitively, the deque dynamic update sequence is a generalization of both the stack and queue dynamic update sequences by choosing appropriate ends for insertion and removal of edges. We show this intuition later. Note that the deque dynamic update sequence is the first of the restricted sequences introduced where the next edge to remove is not given implicitly by the data structure. It is not obvious how the deque and priority queue dynamic update sequences relate to each other in terms of complexity. We show in Section 4.2 that they are indeed incomparable, at least for our formalized notion of complexity based on update sequences emulating each other.

4.2 Emulation

We have already seen a few examples on how some update sequences intuitively relate to each other in the previous section. We now formally define this relation based on update sequences emulating each other. This formalization is then used to show the intuitions given in Section 4.1. We also study minor modifications of the update sequences defined above, and show how some modifications strictly increase the complexity of the update sequences, while other modifications do not make the update sequence more complex.

Definition 4.2 (Emulation)**.** *Let $\mathcal{D}$ and $\mathcal{D}'$ be data structures each containing a set of edges with update operations $\mathcal{O}$ and $\mathcal{O}'$ and parameters $\mathcal{P}$ and $\mathcal{P}'$. An* emulation routine emulating $\mathcal{D}$-dynamic update sequences using $\mathcal{D}'$-dynamic update sequences *is an algorithm that translates an online sequence of updates from $\mathcal{O}$ with their respective parameters to a sequence of updates in $\mathcal{O}'$ and their parameters such that the edge set of both data structures is the same after execution of their respective updates and every update in the original sequence is translated to a constant number of updates in the emulated sequence.*

Formally, this can be defined as a function $\phi : \mathcal{S} \times \mathcal{O} \times \mathcal{P} \to \mathcal{S} \times (\mathcal{O}' \times \mathcal{P}')^{\leq k}$ for a constant $k \in \mathbb{N}$ and some state space $\mathcal{S}$ with unique starting state $S_0 \in \mathcal{S}$. Take any valid $\mathcal{D}$-dynamic update sequence $\mathrm{op}_1, \ldots, \mathrm{op}_l$ with parameters $P_1, \ldots, P_l$, and $D_0, \ldots, D_l$ the sequence of resulting data structures. Let $(S_i, (\mathrm{op}'_{i,1}, P'_{i,1}), \ldots, (\mathrm{op}'_{i,j_i}, P'_{i,j_i})) = \phi(S_{i-1}, \mathrm{op}_i, P_i)$ for $i \in [l], j_i \leq k$. Define $D'_i = \mathrm{op}'_{i,j_i}(\mathrm{op}'_{i,j_i-1}(\ldots \mathrm{op}'_{i,1}(D'_{i-1}, P'_{i,1}) \ldots, P'_{i,j_i-1}), P'_{i,j_i})$ if all intermediary results are not $\perp$, otherwise $D'_i = \perp$. Then ϕ is an emulation routine if $D'_l \neq \perp$ and $E(D_l) = E(D'_l)$ for any valid $\mathcal{D}$-dynamic update sequence.

Note that the emulation may hold a state between translating different updates using the state space $\mathcal{S}$. This is required in most cases as it allows saving information about the past updates executed, which can influence which updates and parameters are used in the translated updates. It is important to note that the translation has to happen online, otherwise the fully dynamic update sequence and priority queue dynamic update sequence would be equivalent: With every insertion one can check when the edge will be removed again and set the lifetime of the edge accordingly. That each update operation needs to be translated to at most a constant number of updates is also a requirement to ensure the emulation is not too powerful: If one would allow $2n^2$ emulated updates per update, the stack dynamic update sequence could emulate the fully dynamic update sequence as a removal could be emulated by removing all edges from the stack and inserting back everything except the removed edge. We note that one could also require a different bound on the number of emulated update operations per update, for example a logarithmic bound, or even allow amortizing this count. We barely use this in this thesis, but highlight a few places where such an emulation can be used, and we note that this could be an interesting research area for future work in Chapter 8.

An emulation therefore allows translating an online sequence of updates from one update sequence to another update sequence. If there is an emulation routine emulating $\mathcal{D}$-dynamic update sequences using $\mathcal{D}'$-dynamic update sequences, we call the $\mathcal{D}'$-dynamic update sequence a *generalization* of the $\mathcal{D}$-dynamic update sequence, and the $\mathcal{D}$-dynamic update sequence a *specialization* of the $\mathcal{D}'$-dynamic update sequence. If the $\mathcal{D}$-dynamic update sequence is a generalization of the $\mathcal{D}'$-dynamic update sequence and vice versa, we call the update sequences *equivalent*. On the other hand, if the $\mathcal{D}$-dynamic update sequence is not a generalization of the $\mathcal{D}'$-dynamic update sequence and vice versa, we call the update sequences *incomparable*. Based on these terms, a notion of complexity can be defined, where more general update sequences are more complex compared to specialized update sequences. It is easy to see that the notion of generalization is transitive, and that the notion of equivalence is an equivalence relation. An emulation can directly be used to translate $\mathcal{D}'$-dynamic algorithms to $\mathcal{D}$-dynamic algorithms. The runtime directly follows from the runtime of the emulated updates, and the overhead required by the emulation itself.

The priority queue dynamic update sequence can emulate the stack dynamic update sequence

To show that the $\mathcal{D}'$-dynamic update sequence can emulate the $\mathcal{D}$-dynamic update sequence, it suffices to give an emulation routine. The main difficulty lies in proving that the emulation routine indeed emulates every valid $\mathcal{D}$-dynamic update sequence correctly as specified in Definition 4.2. To show this, induction is the tool of choice: One assumes to have shown correctness up to a specific length of the emulated update sequence, and then shows that emulating a single operation leads to the same changes in the edge set of the emulated and emulating data structure. Note that this in many cases does not suffice: For example when removing an edge in the stack dynamic update sequence it is not clear from the edge set contained in the stack which edge is removed; one requires knowledge of the entire stack to know the removed edge. One therefore also needs to know the internal state of the emulated data structure. This can either be stored by the emulating data structure, or in the state space $\mathcal{S}$. We for example show that for the priority queue dynamic update sequence emulating the stack dynamic update sequence the state of the stack is contained in the emulating priority queue: The topmost stack edge has the minimum lifetime, with the lifetime of edges strictly increasing for edges further down in the stack. For correct induction, one then also needs to show that this state is correctly updated by the emulated updates.

As an example, we now formally show that the priority queue dynamic update sequence is a generalization of the stack dynamic update sequence. Most further emulation results are only informally described in this thesis.

Lemma 4.3. *The priority queue dynamic update sequence is a generalization of the stack dynamic update sequence. The emulation of each update can be implemented in constant time.*

Proof. The emulation routine maintains an additional integer i, which is initialized to 0. The emulation then works as follows:

- Insertion of edge uv into the stack is emulated by decrementing i and inserting uv into the priority queue with lifetime i.
- Removal of an edge from the stack is emulated by the removal of the shortest lifetime edge from the priority queue and incrementing i.

By decrementing i on insertion and incrementing on deletion, $i = -|E|$ at any point in time. The lifetime of an edge therefore corresponds to the number of edges of the graph after its edge insertion. By structural induction, it is easy to see that the lifetime of edges in the data structures is $\{-|E|, \dots, -1\}$, and is in particular injective. It therefore directly follows that edges are always inserted in the front of the priority queue. As the priority queue also always removes edges from the front, this emulation therefore turns the priority queue into a stack. The top of the stack is the front of the priority queue, and the bottom is the back. An update therefore leads to the same changes in the edge set of the stack and the emulating priority queue, and furthermore maintains the current state of the stack in the priority queue correctly. By induction, this emulation is therefore correct. As i is bounded by n^2, incrementing and decrementing i can be done in constant time. It is therefore easy to see that this emulation can be implemented with a constant runtime per update. □

The same proof also works for the queue dynamic update sequence by incrementing i for every insertion and not modifying it for deletions. Note that in this case i is unbounded, one therefore needs to decrease i and the lifetime of every edge every $O(n^2)$ insertions such that it can be incremented in constant time. This is done by removing every edge from the priority queue and inserting them back with decreased lifetime. This leads to an emulation with an amortized constant number of priority queue dynamic updates per emulated queue dynamic update. We note that a worst-case constant amortization is not possible, at least not with a bounded runtime for the emulation in the WORD-RAM model: Assume otherwise that it would be possible with at most k priority queue dynamic updates per emulated queue dynamic update. Insert $k + 1$ elements into the queue and emulate the operations in the priority queue. Let x be the last element in the emulating priority queue. Repeatedly insert and remove one more element into the queue, and emulate these operations in the priority queue. Then after at most $k + 1$ insertions and removals, x was removed from the queue, and therefore also needs to be removed from the emulating priority queue. This means that x must have moved by at least one position in the priority queue, otherwise x could not have been removed in k priority queue dynamic updates. This means an element needs to either be inserted with or have changed its lifetime to some lifetime bigger than $\tau(x)$. One can repeat this indefinitely, increasing the maximum lifetime arbitrarily. It follows that the lifetime of an element cannot be represented by a bounded number of cells in the WORD-RAM model, the emulation therefore requires an unbounded runtime.

It is furthermore easy to see that the deque dynamic queue dynamic update sequence can emulate both the stack and queue dynamic update sequences: For emulating the stack dynamic update sequence, it suffices to always use the same end for insertion and deletion of edges. For emulating the queue dynamic update sequence, use one end for insertion and the other end for removal of edges.

The stack and queue dynamic update sequences are incomparable

We now discuss results that some update sequences cannot emulate others. To prove that, we show that for any emulation with at most k emulating update operations for emulating a single update there is a sequence of updates after which there is some element that is removable in at most x emulated updates, but the same element requires more than kx emulating updates in the emulating update sequence to be removed, which is a contradiction. As an introductory result, we show that the stack and queue dynamic update sequences are incomparable. Later in this section, we also show other results, in particular that the priority queue and deque dynamic update sequences are incomparable.

Lemma 4.4. *The queue dynamic update sequence cannot emulate the stack dynamic update sequence.*

Proof. Assume otherwise that an emulation with k queue dynamic updates for each stack dynamic update exists. Insert $2k$ elements into the stack. These elements are also required to be in the emulating queue. Insert one more element x into the stack. When emulating this insertion, one of the k emulating queue dynamic operations needs to insert x at the end of the queue. For all other emulating operations, at most k elements can be removed from the start of the queue. This therefore means that x cannot be in the first k elements in the emulating queue as the queue contained $2k$ elements before inserting x, at most k of these elements were removed, and as x was inserted at the back of the queue. But then x can be removed from the stack in one operation, as it was the last inserted edge, but not from the queue in k operations, as there are at least k elements before it in the queue. This contradicts our assumption that k queue dynamic updates suffice to translate every stack dynamic update. □

Lemma 4.5. *The stack dynamic update sequence cannot emulate the queue dynamic update sequence.*

Proof. Assume otherwise that an emulation with k stack dynamic updates for each queue dynamic update exists. Insert $k+1$ elements into the queue, which are also contained in the emulating stack. Name the element at the bottom of the emulating stack x. When inserting more elements into the queue and emulating the operations in the stack, x needs to stay at the bottom of the emulating stack as each emulation can access at most the topmost k elements. Insert $(k+1)^2$ elements into the queue and emulate the operations in the stack. Remove the first $k+1$ elements from the queue, which in particular contains x. Emulating these remove operations in the stack, the emulation must therefore reach x in at most $k(k+1)$ stack dynamic updates. But this is a contradiction, as there are at least $(k+1)^2$ elements on top of x in the stack. □

Corollary 4.6. *The stack and queue dynamic update sequences are incomparable.*

Proof. This directly follows from Lemmas 4.4 and 4.5. □

From Lemma 4.3 and 4.4, it directly follows that the queue dynamic update sequence cannot emulate the priority queue dynamic update sequence. The same idea does not directly show that the stack dynamic update sequence cannot emulate the priority queue dynamic update sequence as the priority queue dynamic update sequence can only emulate the queue dynamic update sequence in an amortized constant emulation. We nevertheless show later that also the stack dynamic update sequence cannot emulate the priority queue dynamic update sequence.

$\mathcal{D}$-with-j-slots-dynamic update sequence

We now introduce modifications of the basic restricted update sequences. It is shown that these modifications can in some cases make a significant difference to the complexity of the update sequences, while in other cases these modifications can easily be emulated by the basic data structures.

For any j and any $\mathcal{D}$-dynamic update sequence, define the *$\mathcal{D}$-with-j-slots-dynamic update sequence* as the $\mathcal{D}$-dynamic update sequence that supports as additional update operations inserting and removing edges from j slots. In a slot, any edge can be freely added (if there is not an edge in the slot already) and removed (if there is an edge in the slot) at any point in time. We call edges that are stored in slots *slot edges*. The edge set represented by an instance of this data structure is then the union of edges contained in the instance of $\mathcal{D}$ as well as all slot edges.

It is easy to see that adding $j+1$ slots to an update sequence is more general compared to adding just j slots. If the $\mathcal{D}'$-dynamic update sequence is a generalization of the $\mathcal{D}$-dynamic update sequence, then the $\mathcal{D}'$-with-j-slots-dynamic update sequence is also a generalization of the $\mathcal{D}$-with-j-slots-dynamic update sequence. Furthermore, adding $\frac{n(n-1)}{2}$ slots for undirected underlying graphs or $n(n-1)$ slots for directed underlying graphs to any update sequence makes the update sequence equivalent to the fully dynamic update sequence: One can put every edge into its own slot.

Although this enhancement to a data structure intuitively makes the update sequences strictly more general, the following lemmas show that a constant number of slots can be emulated in the case of the stack, priority queue and deque dynamic update sequences.

Lemma 4.7. *The stack dynamic update sequence can emulate the stack-with-j-slots dynamic update sequence for any constant $j \in \mathbb{N}$. Each stack-with-j-slots dynamic update is emulated using at most $2j+1$ stack dynamic updates, and the emulation requires constant time per update.*

Proof. Keep the slot edges on top of the stack. Then, emulating updates works as follows:

- Insertion of uv onto the stack: Remove all slot edges from the top of the stack and memorize them. Push uv to the top of the stack and push all slot edges back.
- Insertion of uv into a slot: Push uv on top of the stack.
- Remove an edge from the stack: Remove all slot edges from the top of the stack and memorize them. Remove the topmost edge of the stack and push all slot edges back.
- Remove an edge from a slot: Remove all slot edges from the top of the stack and memorize them. Insert all of them back except for the one in the removed slot.

For correct translation, the emulation routine needs to maintain the number of slot edges, and the order of slot edges on top of the stack. Furthermore, it temporarily needs to memorize at most j edges.

The correctness and runtime of the algorithm is easy to see. Furthermore, each update is emulated by at most $2j + 1$ updates, with equivalence if all slots have edges and when adding or removing an edge from the stack. □

The same lemma also directly applies to the deque dynamic update sequence. Here, one can choose which end of the deque requires overhead for the emulation. A similar lemma is now shown for the priority queue dynamic update sequence. While one can apply exactly the same proof of removing and adding back all slot edges on every priority queue dynamic update, it is shown how this overhead can be removed for inserting edges.

Lemma 4.8. *The priority queue dynamic update sequence can emulate the priority-queue-with-j-slots dynamic update sequence for any constant $j \in \mathbb{N}$. Insertion into the priority-queue-with-j-slots is emulated using one priority queue dynamic update, while removal is emulated using at most $2j + 1$ updates. Emulation requires a constant time per update operation.*

Proof. Emulate all priority-queue-with-j-slots dynamic update operations such that the lifetime of edges in the priority queue is positive, and the lifetime of edges in slots is negative. This can be done by modifying the lifetime of priority queue edges uv from $\tau(uv)$ to $\tau'(uv) = e^{\tau(uv)}$, and for an edge xw in slot i setting $\tau'(xw) = -i$. Then, insertions into the priority queue and slots can be emulated via one update. Removal of edges requires removal and reinsertion of all slot edges, similar to Lemma 4.7.

As the exponential function has a positive image, slot edges always have lower lifetime compared to edges in the emulating priority queue. Furthermore, as the exponential function is strictly monotonically increasing, the order of edges in the emulated and emulating priority queue stay the same. Correctness then follows analogously to Lemma 4.7 and using the above two properties of the emulation. □

Note that a similar methodology as described in Lemmas 4.7 and 4.8 is not directly applicable to the queue dynamic update sequence: Both lemmas required storing the slot edges where they can easily be removed and added back. Such a storage position does not exist for the queue dynamic update sequence. It is in fact easy to see that the queue dynamic update sequence can indeed not emulate the queue-with-one-slot dynamic update sequence, as noted in the following lemma:

Lemma 4.9. *The queue dynamic update sequence cannot emulate the queue-with-one-slot dynamic update sequence.*

Proof. Assume otherwise that an emulation with k queue dynamic updates per emulated queue-with-one-slot dynamic update exists. Insert $2k$ elements in the queue, then insert an edge x in the slot. Emulate these operations in the queue dynamic update sequence. Then, analogous to Lemma 4.4, x cannot be removed in k operations from the emulating queue, but can be removed in one operation from the queue-with-one-slot. □

Other modifications to basic update sequences

Similar results are now described for other changes to the already described update sequences. The first modification is allowing the stack dynamic update sequence to remove any of the j topmost edges of the stack. For a constant j, this can again be emulated using just a stack dynamic update sequence. This can be proven similar to Lemma 4.7: Removal of an edge is emulated by the removal of at most j edges and inserting all except the removed edge

back. Another modification is lifting the restriction for the priority queue dynamic update sequence that the mapping τ is injective: One allows up to j edges to map to the same value, and a remove operation can choose any of the edges with minimum lifetime to be removed. This can be shown to be equivalent to the priority queue dynamic update sequence if a value for $\min\{|\tau(uv) - \tau(xy)| \mid uv, xy \in E, \tau(uv) \neq \tau(xy)\}$ is known in advance. The emulation works by mapping the lifetime $\tau(uv)$ of an edge uv to $\tau(uv) + |\{xy \in E \mid \tau(xy) = \tau(uv)\}| \cdot \varepsilon$ for small enough ε. Depending on how $|\{xy \in E \mid \tau(xy) = \tau(uv)\}|$ is maintained, this emulation has different runtime characteristics. Storing it as a binary tree, each update requires $O(\log n)$ time to be emulated. If randomization is allowed, one can use a hash table with expected $O(1)$ time per update.

j-$\mathcal{D}$-dynamic update sequences

After showing that some modifications to update sequences do in fact not make some update sequences more general, another modification is now given that is shown to be strictly more general than its base update sequence. For any j and any data structure holding edges $\mathcal{D}$, define the *j-$\mathcal{D}$-dynamic update sequence* analogous to the $\mathcal{D}$-dynamic update sequence, but j instances of $\mathcal{D}$ are maintained, and every update operation has an additional parameter $i \in [j]$, specifying which instance is changed. The edge set represented by the data structure is then the union of the edges contained in the j instances of $\mathcal{D}$. It is easy to see that for any $\mathcal{D}$ and any $j > 1$, the j-$\mathcal{D}$-dynamic update sequence is a generalization of the $(j-1)$-$\mathcal{D}$-dynamic update sequence. This also shows that the 2-stack dynamic update sequence is a generalization of the stack dynamic update sequence. Furthermore, the 2-stack dynamic update sequence is a specialization of the deque dynamic update sequence by interpreting each of the two ends of the deque as two stacks. As the queue dynamic update sequence cannot emulate the stack dynamic update sequence by Lemma 4.4, it directly follows that the queue dynamic update sequence also cannot emulate the 2-stack dynamic update sequence or the deque dynamic update sequence.

It is now shown that the 2-stack dynamic update sequence is strictly more general than the stack dynamic update sequence. This in particular shows that the deque dynamic update sequence is strictly more general than the stack dynamic update sequence.

Lemma 4.10. *It is not possible to emulate the 2-stack dynamic update sequence using a stack dynamic update sequence.*

Proof. Assume otherwise that such an emulation routine exists. Let k be the maximum number of stack dynamic update operations each 2-stack dynamic update is emulated with. Let A and B be the emulated stacks, and S be the stack used in the emulation. Choose $n > k$.

Insert n edges into A and B each and emulate the operations in S. Let uv be the edge at the bottom of S after the insertions. Let without loss of generality $uv \in A$.

Assume uv is the topmost edge of A. Then removing uv from A would require at least $n > k$ updates being executed: All edges in B would first need to be removed as they are above uv in S. This contradicts the assumption that the emulation routine can translate every 2-stack dynamic update to at most k stack dynamic updates.

Therefore, assume uv is not the topmost edge of A. Remove the topmost edge of A. As the emulation routine can do at most k removals and insertions from the stack S, there are at least the n edges of B on top of uv on the stack and $n > k$, uv must remain the bottommost edge after removal of the topmost edge of A. Therefore, one can iteratively remove the topmost edge of A until uv is the topmost edge of A and uv remains the bottommost edge

of S. But after the remove operations, it is now not possible to remove uv in a constant number of updates as already shown. □

One can also show that the queue dynamic update sequence cannot emulate the 2-queue dynamic update sequence: This directly follows as the 2-queue dynamic update sequence can emulate the queue-with-one-slot dynamic update sequence, but the queue dynamic update sequence cannot do that by Lemma 4.9. We show later that the priority queue dynamic update sequence also cannot emulate the 2-priority-queue dynamic update sequence.

The priority queue and deque dynamic update sequences are incomparable

We now show something even stronger than Lemma 4.10: Not even the priority queue dynamic update sequence can emulate the 2-stack dynamic update sequence, which implies that it also cannot emulate the deque or 2-priority-queue dynamic update sequences. We also show that the deque dynamic update sequence cannot emulate the priority queue dynamic update sequence, which makes these two update sequences incomparable.

Lemma 4.11. *The priority queue dynamic update sequence cannot emulate the 2-stack dynamic update sequence.*

Proof. Assume otherwise that a 2-stack dynamic update can be emulated with at most k priority queue dynamic updates. We call the emulated two stacks A and B, and the emulating priority queue Q. Insert n elements into both stacks and emulate that in Q. Let without loss of generality the backmost element e of Q be in stack A. We first investigate how repeatedly removing an element from A changes the position of e in Q over time. We show that the elements in B significantly influence how fast e can move forward in Q. In particular, it is shown that for large enough number of elements n in both stacks, it takes more than n remove operations from A until e is in the first k elements in Q. This is a contradiction, as e can be removed in n 2-stack dynamic operations, but cannot be removed in nk priority queue dynamic operations from the priority queue.

Emulating a remove operation in A can remove at most k elements from the front of Q, and reinserting them all except for the removed element back into Q with different lifetime. Restrict the view of Q to only elements in $B \cup \{e\}$, which we call $Q_{B\cup\{e\}}$. A remove operation in $A \setminus \{e\}$ then just reinserts the at most first k elements in $Q_{B\cup\{e\}}$ with new lifetime. We call this a *shuffle*. We allow shuffles, and therefore also the emulation, to be even stronger by allowing reinsertions of exactly the first k elements in $Q_{B\cup\{e\}}$. Define the *drift* d_j^t of position $j > k$ in $Q_{B\cup\{e\}}$ at shuffle t as the number of positions the element in position j moves forward in this shuffle. As an extreme example, assume a shuffle t moves the first k elements in $Q_{B\cup\{e\}}$ to the back of $Q_{B\cup\{e\}}$. The elements at positions $j > k$ then move forward exactly by k elements. Therefore, $d_j^t = k$. As another extreme example, assume a shuffle t keeps the first k elements in $Q_{B\cup\{e\}}$ at the front of $Q_{B\cup\{e\}}$. Then the elements at position $j > k$ do not move forward in the shuffle, therefore $d_j^t = 0$. It is easy to see that $0 \leq d_j^t \leq k$ for all $j > k$ and t. Furthermore, d_j^t is non-increasing over j for all t. For a sequence of shuffles of length l, define the *average drift over time* for position $j > k$ in the priority queue as $d_j = \frac{1}{l}\sum_{t=1}^{l} d_j^t$.

Fix the state of $Q_{B\cup\{e\}}$ after inserting n elements into both A and B, where $e \in A$ is the backmost element in $Q_{B\cup\{e\}}$. Fix a sequence of shuffles that minimizes the time until e is shuffled once. We now investigate for how many positions j there holds $d_j \geq \frac{1}{2}$. Let x be maximal with $d_x \geq \frac{1}{2}$. We show that x is bounded from above by a function depending on k, but not n. Consider the ith topmost stack element in B, with $i \leq \lfloor \frac{x}{k} \rfloor$. This element then needs to be in the first $ik \leq x$ elements in Q, otherwise it could be removed from B

in i operations but not from Q in ik operations. Therefore, a shuffle operation can move this item to position at most x in $Q_{B\cup\{e\}}$. As the average drift over time is non-increasing, this means that this element is always at positions in $Q_{B\cup\{e\}}$ with average drift over time of at least $\frac{1}{2}$. On average, it therefore moves forward by at least $\frac{1}{2}$ positions for each shuffle. As it has position at most ik, this means that on average the element is shuffled at most every $2ik$ shuffles. Let t be a shuffle where the ith topmost stack element in B is shuffled, and let p_i^t be the position this element is shuffled to. Then $d_{k+1}^t \geq d_{p_i^t}^t \geq d_{p_i^t+1}^t + 1 \geq d_{ik+1}^t + 1$, where the first and third inequality follows as the drift is non-increasing, and the second inequality follows as the ith topmost element is inserted at position p_i^t at shuffle t. As this happens every $2ik$ shuffles, $d_{k+1} \geq d_{ik+1} + \frac{1}{2ik}$, and therefore $d_x \leq d_{k+1} - \frac{1}{2ik}$. This accumulates for every i with $i \leq \lfloor \frac{x}{k} \rfloor$. Combining this reduced average drift over time for all $i \in [1, \ldots, \lfloor \frac{x}{k} \rfloor]$ leads to $d_x \leq k - \sum_{i=1}^{\lfloor \frac{x}{k} \rfloor} \frac{1}{2ik} = k - \frac{1}{2k} \sum_{i=1}^{\lfloor \frac{x}{k} \rfloor} \frac{1}{i}$. It is well-known that $\lim_{j\to\infty} \sum_{i=1}^{j} \frac{1}{i} = \infty$. Therefore, for large enough x, this would upper bound the average drift over time by a negative number. This is a contradiction as the drift is always positive. Therefore, x must be bounded from above by a function dependent on k but not n. As k is constant, this means that at most a constant number of positions in the priority queue have an average drift over time of at least $\frac{1}{2}$.

Show now that for large enough n the backmost element e requires more than n shuffles until it is shuffled exactly once. Assume otherwise that at most n shuffles suffice. If strictly less than n shuffles are needed, add intermediate shuffles that do not move the elements at all. Therefore, we can assume that after exactly n shuffles e is at position at most k in $Q_{B\cup\{e\}}$. Let p_e^t be the position of element e before shuffle t, and $d_{p_e^t}^t$ the drift of element e at shuffle t. For the computation of d_j, set $d_p^t = 0$ for all $p > p_e^t$ for all shuffles t. This only decreases the average drift over time as the drift is non-negative. Furthermore, set $d_p^t = d_{p_e^t}^t$ for all $k < p \leq p_e^t$ for all shuffles t. This again only decreases the average drift over time as the drift is non-increasing over $Q_{B\cup\{e\}}$. Due to these changes, note that for the assumed sequence of n shuffles after which e is at position at most k in $Q_{B\cup\{e\}}$ there holds $\sum_{t=1}^{n} d_j^t \geq n - j$ for all $j > k$: Each positive d_j^t signifies that e was after position j in the priority queue and moved forward by d_j^t positions due to shuffle t. Therefore, if for some $j > k$ there holds $\sum_{t=1}^{n} d_j^t < n - j$, then e would have moved forward by at most $n - j$ positions. As e started at position $n + 1$, it would end up at position at least j in $Q_{B\cup\{e\}}$. This is a contradiction to e being at position at most k in $Q_{B\cup\{e\}}$ after the n shuffles. Investigate the average drift over time of position $\frac{n}{2}$ in $Q_{B\cup\{e\}}$. For this position, $\sum_{t=1}^{n} d_{\frac{n}{2}}^t \geq \frac{n}{2}$. This leads to an average drift over time of $d_{\frac{n}{2}} \geq \frac{1}{2}$. As the drift is non-increasing over positions in $Q_{B\cup\{e\}}$, it also means that $d_p \geq \frac{1}{2}$ for all $k < p \leq \frac{n}{2}$. This is a contradiction to what was shown above that at most a constant number of positions in $Q_{B\cup\{e\}}$ have an average drift over time of at least $\frac{1}{2}$. Therefore, for large enough n, the backmost element e requires more than n shuffles until it is shuffled exactly once.

Expanding our view of Q to both elements in A and B again, we have therefore shown that for n remove operations in A, e ended up at position at least k in $Q_{B\cup\{e\}}$ and therefore also Q. But this is then a contradiction as e can be removed from the 2-stack in at most n operations, but that it is not possible to remove it from Q when emulating these operations. Therefore, the priority queue dynamic update sequence cannot emulate the 2-stack dynamic update sequence. □

Lemma 4.12. *The deque dynamic update sequence cannot emulate the priority queue dynamic update sequence.*

Proof. Assume otherwise that a priority queue dynamic update can be emulated by at most k deque dynamic updates.

Take a fixed list of n edges $e_1, \ldots, e_n$. We consider update sequences first inserting edges $e_1, \ldots, e_n$ in that order but with potentially different lifetime, followed by n remove operations. Compute how many different update sequences of such form exist for the priority queue dynamic update sequence and the deque dynamic update sequence emulating it. For the priority queue dynamic update sequence, we ignore the exact lifetime of edges and instead focus on the order of edges in the priority queue. This ensures that different priority queue dynamic update sequences have a different edge set at some point in the update sequence.

We start by describing how many different new states an edge insertion can lead to for the priority queue and deque. Let first a priority queue and deque state already be given with i elements. Then an edge insertion in the priority queue leads to one of $i+1$ new states, depending on where in the priority queue the edge was inserted. For the emulation using a deque, we allow the emulation to remove k elements from both ends of the deque, and insert these edges, including the edge to be inserted, in any order at any of two ends again. Then, there can be at most $C_1 = 2^{2k+1}(2k+1)!$ many possible next states, the 2^{2k+1} indicating the end of the deque where an edge is inserted, and the $(2k+1)!$ in which order they are inserted. For n insertions on the empty priority queue and deque dynamic states, there are therefore $n!$ many priority queue states, and at most C_1^n many deque states.

We now discuss deleting edges. For the priority queue, deleting an edge can only lead to a single possible next state, the same priority queue but with the shortest lifetime edge removed. For the deque, one can do a similar computation as above, and get at most $C_2 = 2^{2k-1}(2k-1)!$ possible next states for a single edge deletion. For n edge deletions and a fixed starting deque state, there therefore are C_2^n possible states.

Combining both results, there are therefore $n!$ different priority queue dynamic update sequences consisting of n insertions of fixed edges followed by n deletions, while there are only $(C_1C_2)^n$ different sequences for the deque emulating the priority queue. For large enough n, there are therefore at least two priority queue dynamic update sequences that are emulated with the same sequence of deques. This is a contradiction as the priority queue dynamic update sequences are distinct, therefore the lifetime of edges is ordered differently and at some point in the removal of the edges the update sequences contain a different set of edges. □

This in particular also shows that the 2-stack and stack dynamic update sequences cannot emulate the priority queue dynamic update sequence.

Theorem 4.13. *The priority queue dynamic and deque dynamic update sequences are incomparable. Furthermore, the priority queue dynamic and 2-stack dynamic update sequences are also incomparable.*

Proof. This directly follows from Lemmas 4.11 and 4.12. □

The 2-stack dynamic update sequence cannot emulate the queue dynamic update sequence

We now show a stronger version of Lemma 4.5: Not even the 2-stack dynamic update sequence can emulate the queue dynamic update sequence with a worst-case constant number of 2-stack dynamic updates per emulated queue dynamic update. Note that this is in particular interesting as it is well-known that a queue can be implemented using two stacks with an amortized constant time per update operation: Insert elements into one stack and remove elements from the other stack; if the stack for removal is empty, move all

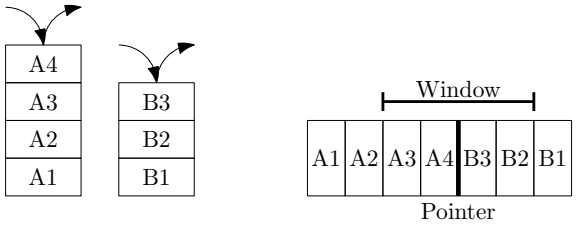

Figure 4.2: A 2-stack state and its representation in a single list.

elements from the insertion stack to the removal stack one-by-one. This methodology can also be used to get an emulation of a queue dynamic update sequence using an amortized constant number of 2-stack dynamic updates.

Lemma 4.14. *The 2-stack dynamic update sequence cannot emulate the queue dynamic update sequence.*

Proof. Assume otherwise that such an emulation exists that emulates each queue dynamic update with at most k 2-stack dynamic updates. We allow the emulation to be even stronger but simpler like we have done in Lemma 4.12 by allowing access to the topmost k elements of both stacks. Our proof proceeds as follows: We first show that it requires many emulated queue dynamic update operations to move items between the emulating stacks. Afterwards, we give a sequence of queue dynamic updates that forces many items to move between the stacks. We then show a contradiction using the above two results, by showing that the length of the given queue dynamic update sequence is significantly shorter than required.

Fix a 2-stack state. To show that moving d items between the stacks requires many emulated queue dynamic update operations, interpret a 2-stack state as follows: Concatenate the two stacks at their top, producing a single list where the elements at the ends of the list are the bottommost elements of the stack, and a *pointer* into the list separating the two stacks from each other. We call the at most $2k$ items around the pointer where the two stacks meet the *window*. A visualization of this representation with $k = 2$ is given in Figure 4.2. We show later that we are only interested in inserting items into the queue. In that interpretation, an insertion is implemented by shuffling the window, inserting the element into the window, and moving the pointer at most k elements in either direction. Then, moving items between stacks means moving the pointer around in the list. To be more precise, as every queue dynamic update inserts an element, we are interested in shrinking one of the stacks by d elements. Note that similar to the proof of Lemma 4.11 the ith item in the queue needs to have a distance of at most ik elements from the pointer. We allow the emulation to be even stronger: The ith item in the queue needs to have distance of at most ik elements from the pointer while only counting elements that were in the original 2-stack state, ignoring any newly inserted edges. This allows us to completely ignore the insertion of new elements into the queue, in particular allowing us to skip representing them in our list and interpreting insertions as shuffles of the window and moving the pointer. Assume that the pointer is initially at position j, and we want to compute a lower bound on how many queue dynamic updates it takes for the pointer to move to position $j + d$. The first item in the queue was then at position at most $j + k$, and ended up at position at least $j + d - k$, moving at least $d - 2k$ positions. Likewise, the second item in the queue was at position at most $j + 2k$, and ended up at position at least $j + d - 2k$, moving at least $d - 4k$ positions. This works analogously for the first $\lfloor \frac{d}{2k} \rfloor$ items. Summing everything up, we get the total movement required for these items:

$$\sum_{i=1}^{\lfloor \frac{d}{2k} \rfloor} (d - 2ik) \in \Omega\left(\frac{d^2}{k}\right)$$

Movement of items is done by insertions when shuffling the window. As an upper bound, we can approximate the number of items moved by a single shuffle of the window by the number of pairs that swapped their order due to the shuffle, which is $O(k^2)$ due to the window size of $2k$. Therefore, it takes at least $\Omega\left(\frac{d^2}{k^3}\right)$ queue dynamic updates for moving d items between the stacks.

We now give an update sequence of $O(k^3 d)$ queue dynamic updates, that is split into three batches. It is shown that in at least one batch, at some point during execution of the batch, d items were moved between the stacks, compared with the 2-stack state when the batch started. This leads to a contradiction for constant k and large enough d, as $\Omega\left(\frac{d^2}{k^3}\right)$ queue dynamic updates are required as computed above. Insert $3(k + d)$ elements, and call it the 1st batch. Insert $6k(k + d)$ more elements, and call it the 2nd batch. Looking only at elements from the 2nd batch, at least one of the emulating stacks A contains $3k(k + d)$ elements. In A, the bottommost element must be from the 2nd batch: Otherwise it would be in the first batch and therefore removable in $3(k + d)$ queue dynamic updates, but not in $3k(k + d)$ 2-stack dynamic updates as there are at least $3k(k + d)$ elements from the 2nd batch on top of it. Assume that after insertion of the 1st batch the stack A contained more than $k + d$ elements. Then, as an element at the bottom of A was inserted, at least d of the elements in the 1st batch must have been moved to the other stack while inserting the 2nd batch. In this case we are therefore finished, as we wanted to show that moving d elements is required in the sequence of updates. It therefore remains to discuss the case where after insertion of the 1st batch A contained at most $k + d$ elements. Therefore, the other stack B contained at least $2(k + d)$ items. Assume that after inserting the 2nd batch B contained at most $k + d$ items. Then d items would have been moved to stack A while inserting this batch. In this case, we are therefore already finished. It therefore remains to consider the case where B contains at least $k + d$ items after insertion of the 2nd batch. Then both stacks A and B contain at least $k + d$ items: A has at least $3k(k + d)$ items from the 2nd batch and B has at least $k+d$ items from the 1st batch. Insert $2k(3(k+d)+6k(k+d))$ more items and call it the 3rd batch. Again, at least one element in the 3rd batch would need to end up at the bottom of at least one stack. But this means that d elements from that stack would need to be moved to the other stack. Therefore, no matter how the emulation is implemented, in at least one batch during the emulation d items moved between the stacks. As already mentioned, this leads to a contradiction as this sequence of updates only has length $O(k^3 d)$, while $\Omega(\frac{d^2}{k^3})$ would be required. □

We note that Lemma 4.14 also shows that the 2-stack dynamic update sequence cannot emulate the deque dynamic update sequence: The deque dynamic update sequence can emulate the queue dynamic update sequence, while the 2-stack dynamic update sequence cannot do that. Therefore, the deque dynamic update sequence is a strict generalization of the 2-stack dynamic update sequence.

Finishing Figure 4.1

We now give the last remaining proofs required to show that Figure 4.1 is indeed the transitive reduction of all possible emulations (assuming a bounded runtime for the emulation). After that, for every pair of update sequences, we have either shown that one can (potentially transitively) emulate the other, or that they are indeed incomparable.

Furthermore, for amortized emulations, we have shown that a worst-case emulation is indeed not possible.

Lemma 4.15. *The following hold:*

1. *The fully dynamic update sequence can emulate every other update sequence mentioned in this thesis.*
2. *No update sequence mentioned in this thesis, except for the fully dynamic update sequence itself, can emulate the fully dynamic update sequence.*
3. *The incremental and decremental update sequences cannot emulate any update sequences mention in this thesis, except themselves.*

Proof. To show statement 1, we give a simple emulation: The emulation just maintains the underlying data structure and therefore knows which edges are added or removed in an update. As all of our update operations add or remove exactly one edge, this can then be translated into a fully dynamic update sequence.

Regarding statement 2, note that for every restricted update sequence, for insertion of enough edges, there is at least one edge that cannot be removed in a constant number of operations. But this edge can be removed in the fully dynamic update sequence in one update.

Statement 3 is obvious as all presented update sequences except for the incremental and decremental ones can both insert and remove edges. □

4.3 Initial Results

In this section, we give some of the easy first results for our model of restricted update sequences. We use results introduced in related work and see how they can be used in our model. We first describe how the concepts in the literature relate to restricted update sequences to enable using results from the literature in our model.

It was already described above how the commonly used incremental, decremental and fully dynamic update sequences can be interpreted as a $\mathcal{D}$-dynamic update sequence. The sliding window model [CMS13] can be modelled as a $\mathcal{D}$-dynamic update sequence, where $\mathcal{D}$ has a single update operation that allows inserting an edge and that also removes the oldest edge from the graph if the number of edges exceeds the size of the sliding window. It can also be emulated by the queue dynamic update sequence, by explicitly removing the oldest edge after each insertion if the number of edges would exceed the size of the sliding window. Batch dynamic algorithms [AABD19] are $\mathcal{D}$-dynamic update sequences, where insertion and deletion operations take a set of edges. Furthermore, if queries are done in batches, they can instead be replaced by one query operation taking a list of parameters. Algorithms for emergency planning [HN16] can be modelled analogous to batch dynamic algorithms, but any update after the first one always return $\perp$. Offline dynamic graphs [Epp94] can be translated to a priority queue dynamic update sequence, as such a translation can check at which point in time an edge is removed and use that information to populate the lifetime of the edge. Finally, the deletion-look-ahead setting introduced by Peng and Rubinstein [PR23] directly corresponds to the priority queue dynamic update sequence. They in particular showed a general reduction from incremental algorithms to priority queue dynamic algorithms. As this reduction is used a few times in this thesis, we state their main result. We also give a sketch of the proof, refer to the paper by Peng and Rubinstein [PR23] for details.

Lemma 4.16 ([PR23])**.** *Let $T \geq 1$ be the total number of updates. Suppose there exists a dynamic algorithm for the incremental update sequence with query time t_q and worst-case update time t_u. Then there is a dynamic algorithm for the priority queue dynamic update sequence with query time t_q and amortized update time $O(t_u \cdot \log(T))$.*

Proof sketch. The proof works by regularly rewinding computation and redoing it in an order such that elements with small lifetime were done more recently compared to elements with longer lifetime. This is done by distributing elements into buckets $B_0, \ldots, B_{\lceil \log(T) \rceil}$ of exponentially growing size. The reduction then rewinds the computation of bucket B_i every 2^i operations, and executes the operations again in order of decreasing lifetime. This guarantees that B_0 always contains the element with minimum lifetime, and in general that the $2^{i+1} - 1$ elements with smallest lifetime are in $B_0 \cup \ldots \cup B_i$. The correctness of this reduction then follows from B_0 always containing the element with minimum lifetime, which allows deleting the element with minimum lifetime efficiently. The amortized runtime follows from there being $O(\log T)$ buckets, and from a bucket with 2^i elements being recomputed every 2^i updates. □

While the runtime depends on the total number of updates done, which is independent of n, it is easy to see by inspecting the proof that one can also use n^2 instead of T for graphs. This was also mentioned by van den Brand et al. [vdBFNP24]. One can therefore convert incremental algorithms with worst-case runtime guarantee to amortized priority queue dynamic algorithms with only a logarithmic update time overhead.

Inspecting the proof for this reduction, it is also easy to see that it also applies to the stack dynamic case with amortized runtime. In this case, the proof can also be interpreted as an emulation with an amortized logarithmic number of stack dynamic updates per priority queue dynamic update. This leads to the following lemma:

Lemma 4.17. *Suppose there exists a dynamic algorithm for the stack dynamic update sequence with query time t_q and amortized update time t_u. Then there is a dynamic algorithm for the priority queue dynamic update sequence with query time t_q and amortized update time $O(t_u \cdot \log(n))$.*

By reverting changes done by an incremental algorithm, as also done by Peng and Rubinstein [PR23] in their reduction, one can also convert a worst-case incremental algorithm to a stack dynamic algorithm without runtime overhead. We note that an algorithm with amortized insertion time does not work: One can incrementally build a graph until one hits an expensive update. Repeatedly undoing and executing the same update again would lead to the expensive insertion operation being repeated. This can lead to almost all updates executed being expensive.

Lemma 4.18. *Suppose there exists a dynamic algorithm for the incremental update sequence with query time t_q and worst-case update time t_u. Then there is a dynamic algorithm for the stack dynamic update sequence with query time t_q and update time $O(t_u)$.*

We can directly apply Lemma 4.18 and Lemma 4.17 to get initial runtimes for many problems. We now list a few of these results.

Computing a maximal matching in a static, undirected graph is easy to do using a greedy algorithm in linear time. When the graph is unweighted, the order of the edges the greedy algorithms considers does not matter, the result is always a maximal matching. This can

equivalently be viewed as an incremental algorithm with constant update and query time. This directly leads to a stack dynamic algorithm with the same runtime, and a priority queue dynamic algorithm with amortized logarithmic update time and constant query time. Note that Solomon [Sol16] already gave a fully dynamic algorithm for maintaining a maximal matching in amortized expected constant time with high probability. Our priority queue dynamic algorithm is therefore worse than the algorithm presented by Solomon.

When inserting vertices v with incident edges, a similar technique can also be used to maintain a maximal independent set in $O(\deg(v))$ update time for the incremental and stack dynamic update sequences. Note that in this case, Lemma 4.17 cannot directly be applied as the runtime of a single vertex insertion depends on the degree of the vertex. Removing, reordering and reinserting a set of vertices as done by Peng and Rubinstein [PR23] would therefore lead to a different runtime per update based on the order of edge insertion. One could of course assume a worst-case time of $O(n)$ per vertex insertion, but that would lead to a runtime worse than recomputing a maximal independent set from scratch. Note that the runtime of this methodology cannot be compared with the runtime of incremental and fully dynamic algorithms as given by Gupta and Khan [GK18] as they also allow insertion of edges between already inserted vertices.

When incrementally inserting simplicial vertices — vertices whose neighborhood form a clique — it is easy to compute the size of the largest clique of the graph as maximum cliques are always formed by a simplicial vertex and its neighborhood at time of insertion. It only takes $O(\deg(v))$ time to compute the size of the clique created by inserting a simplicial vertex v. In case one also requires verification that the inserted vertex is indeed simplicial, an additional $O(\deg(v)^2)$ time is required. Regardless of whether one needs to verify that the inserted vertex is simplicial, this methodology can be made stack dynamic. Note that undoing a vertex insertion with verification can actually be sped up to $O(\deg(v))$, which is required to update the underlying graph, if we store for each vertex the maximum clique size the graph had before it was inserted and restore that value on removal. Lemma 4.17 again cannot be applied similar to the maximal independent set problem.

We can also recognize interval graphs in the stack dynamic update sequence when inserting vertices with adjacent edges. This is again a simple application of Lemma 4.18 to the algorithm by Hsu [Hsu96], and has update time $O(\deg(v) + \log n)$ and constant query time. Lemma 4.17 again does not apply. We also note that the methodology cannot be applied to the incremental algorithm by Korte and Möhring [KM89], as they assume that the vertices are ordered by a lexicographic breadth-first search.

Finally, we also note that Lemma 4.18 can be applied to the incremental reachability algorithm developed by Bulteau et al. [BDHTG25] for directed acyclic graphs and when inserting sinks. Refer to their paper for details regarding the runtime of the algorithm. Lemma 4.17 is again not applicable.

5. Connectivity

Connectivity is one of the most common problems investigated for dynamic graphs. Its queries require two vertices as parameters, and return a Boolean value that indicates whether the two vertices are connected in the underlying undirected graph. The current best fully dynamic algorithm for connectivity has an expected amortized $O(\log n(\log\log n)^2)$ update time and $O(\log n/\log\log\log n)$ query time [HHKP17]. Incremental connectivity can be solved using the union-find data structure in amortized $O(\alpha(n))$ time for updates and queries [EGI99]. In the special case of plane underlying graphs, Eppstein et al. [EIT+92] gave an algorithm with amortized $O(\log n)$ update and query time. If the graph is always guaranteed to be a forest, top trees can be used to solve the problem in $O(\log n)$ time per operation. The current best lower bound per operation for this problem is $\Omega(\log n)$ for fully dynamic update sequences, as shown by Pătraşcu and Demaine [PD05]. This lower bound even holds with amortization and randomization. They in particular also showed optimality of the algorithms in the plane case or if the graph is always a forest. Some related work on connectivity problems is summarized in Table 5.1, a survey by Hanauer et al. [HHS22] provides a more complete reference of already existing dynamic connectivity algorithms.

In this chapter, algorithms with improved update time are given for restricted update sequences. It is shown that a logarithmic time for updates and queries for the priority queue and deque dynamic update sequences is possible. We show that this is optimal for the priority queue dynamic update sequence. For the stack dynamic update sequence, this can be improved to amortized $O(\log n/\log\log n)$ time per operation. Based on these

Update Sequence	Update Time	Query Time	Reference
incremental	$O(\alpha(n))^a$	$O(\alpha(n))^a$	[EGI99]
fully dynamic	$O(\sqrt{n}\log(m/n))$	$O(1)$	[EGIN92]
fully dynamic	$O(\sqrt{n})$	$O(1)$	[EGI93]
fully dynamic	$O((\log n)^3)^{a\,r(l)}$	$O(\log n)$	[HK99]
fully dynamic	$O(\log n(\log\log n)^3)^{a\,r(l)}$	$O(\log n/\log\log\log n)$	[Tho00]
fully dynamic	$O(\log n(\log\log n)^2)^{a\,r(l)}$	$O(\log n/\log\log\log n)$	[HHKP17]
fully dynamic	insert: $O(\alpha(n))^a$, remove: $O(n\log(m/n))$	$O(\alpha(n))^a$	[EGIN92]
fully dynamic; planar	$O(\log n)$	$O(\log n)$	[EIT+92]

Table 5.1: Summary of dynamic connectivity algorithms. For the runtimes, a superscript of a refers to amortization, and $r(l)$ refers to Las Vegas randomization.

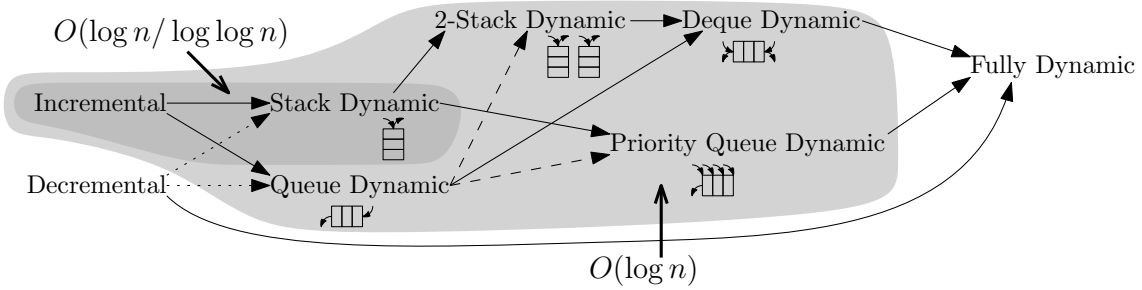

Figure 5.1: Connectivity Results

improvements, further problems are solved as a direct consequence of the results given in this chapter. A summary of achieved runtimes is given in Figure 5.1.

5.1 Stack Dynamic

The first update sequence for connectivity this chapter investigates is the stack dynamic update sequence. While one could directly apply Lemma 4.18 to the incremental algorithm presented by Eppstein et al. [EGI99], this would lead to an $O(\log n)$ time algorithm. This is because a standard union-find implementation has the worst-case time of $O(\log n)$ for finding the representative of the set containing an element [WT89].

In order to improve on this runtime, a union-find data structure is required that already has undoing the most recent union operation built in. Such a data structure is given by Westbrook and Tarjan [WT89], which has amortized $O(\log n/\log\log n)$ time required to compute the union of two sets, finding the representative of a vertex, and undoing the latest union-operation. The authors have also shown that this runtime is optimal for any separable pointer-based algorithm. Using this data structure, one can easily construct an algorithm for stack dynamic connectivity, as shown in Theorem 5.1. Such an algorithm was already mentioned by Henzinger and King [HK99] as well as Henzinger and Fredman [HF98].

Theorem 5.1. *There is an algorithm for the connectivity problem in the stack dynamic update sequence with amortized $O(\log n/\log\log n)$ update and query time.*

Proof. This uses the union-find data structure with backtracking introduced by Westbrook and Tarjan [WT89]. A set corresponds to the vertices of a connected component of the graph. Furthermore, maintain a stack that contains information about which edge insertion merged two sets and which did not.

Whenever an edge uv is inserted, query whether the endpoints of uv are already connected using the union-find data structure. If they are already connected, push $false$ onto the stack. If they are in different connected components, push *true* onto the stack and merge the sets containing u and v.

Whenever an edge is removed, remove the topmost item of the stack. If it is $false$, nothing needs to be done. If it is *true*, backtrack the union-find data structure.

The correctness follows directly from the correctness of the incremental algorithm [EGI99], and the fact that undoing an edge insertion backtracks the union-find data structure if and only if the endpoints of the edge were in different connected components before insertion. The runtime is dominated by the union-find data structure, which is amortized $O(\log n/\log\log n)$. □

Notice that this algorithm does not require knowing which edge is removed by a remove operation. In fact, this algorithm does not even require maintaining the underlying graph. Furthermore, a minor modification of this algorithm also allows maintaining a spanning forest for stack dynamic update sequences in amortized $O(\log n / \log \log n)$ time: Add an edge to the spanning forest whenever the two endpoints were in to different sets in the union-find data structure and remove an edge from the spanning forest if it is removed from the graph.

5.2 Longest Lifetime Spanning Forest

We introduce the concept of longest lifetime substructures now. This is a regularly used concept in the thesis, in particular maintaining the longest lifetime spanning forest is the key for the proof of a priority queue dynamic connectivity algorithm with a logarithmic runtime per operation. Longest lifetime substructures require a total order on the edges, which allows defining a total order on sets edges. This total order on the edges is defined based on the lifetime of an edge in most cases.

Definition 5.2 (Longest Lifetime Substructures)**.** *Totally order edges based on an injective mapping $\tau : E \to \mathbb{R}$. Compare sets of edges lexicographically, which means that for $S_1, S_2 \subseteq E$ with $S_1 \neq S_2$ there holds $S_1 < S_2$ if and only if one of the following holds:*

- $S_1 = \emptyset, S_2 \neq \emptyset$.
- $\min\{S_1\} < \min\{S_2\}$.
- $\min\{S_1\} = \min\{S_2\}$ *and* $S_1 \setminus \{\min\{S_1\}\} < S_2 \setminus \{\min\{S_2\}\}$.

Otherwise, $S_2 < S_1$.

Given a set $\mathcal{S} \subseteq 2^E$ of subsets of the edges, define the longest lifetime substructure *as the maximum element in $\mathcal{S}$.*

We also define the *shortest lifetime substructure* as the longest lifetime substructure for the total order $-\tau$. While intuitively the lifetime of a substructure is only based on the minimum lifetime of any edge contained in it, this thesis requires a stronger notion of lifetime for substructures. This restriction is helpful as we show later that when removing the shortest lifetime edge from the underlying graph and its longest lifetime spanning forest, the resulting forest is also the longest lifetime spanning forest of the updated underlying graph. It furthermore makes the longest lifetime spanning forest unique. This thesis takes a closer look at longest and shortest lifetime substructures in Chapter 7.

Note that in particular, the spanning forest constructed by the stack dynamic update sequence is the longest lifetime spanning forest. This is easy to see with the following lemma:

Lemma 5.3. *A spanning forest F of $G = (V, E)$ is the longest lifetime spanning forest based on the mapping $\tau : E \to \mathbb{R}$ if and only if for all non-forest edges $uv \in E \setminus F$ there holds $\tau(uv) < \tau(xw)$ for all $xw \in \pi(u, v)$.*

Proof. Let F be a spanning forest such that there exists an edge $uv \in E \setminus F$, and an $xw \in \pi_F(u, v)$ such that $\tau(uv) > \tau(xw)$. Set $F' = F \setminus \{xw\} \cup \{uv\}$. Then F' is a forest. Assume otherwise. A cycle would either only use edges in F or use uv. The first case would contradict F being a forest. In the second case, one could use $\pi_F(u, v)$ instead of uv, which again contradicts F being a forest. It is a spanning forest as F' has the same number

of edges as F. Furthermore, F has shorter lifetime than F', as F includes xw, F' does not, and they have the same edges with lifetime shorter than xw. Therefore, F is not the longest lifetime spanning forest.

Let now F be a spanning forest such that for all $uv \in E \setminus F$, $\tau(uv) < \tau(xw)$ for all $xw \in \pi_F(u, v)$. Assume $F' \neq F$ is the longest lifetime spanning forest. Let xw be the edge with minimum τ that is included in F but not in F'. Then the path $\pi_{F'}(x, w)$ exists and does not use xw, otherwise xw would in F'. Let F_x, F_w be the spanning forests containing x and w if xw is removed from F. Following the path $\pi_{F'}(x, w)$ from x to w, there exists some edge uv with one endpoint $u \in F_x$ and the other in $v \in F_w$, otherwise $\pi_{F'}(x, w) \cup \{xw\}$ would be a cycle in F. Then, $\pi_F(u, v) = \pi_F(u, x) \cup \{xw\} \cup \pi_F(w, u)$. By the definition of F, this means $\tau(uv) < \tau(xw)$. But as $uv \in F'$ and $uv \notin F$ and as xw was assumed to have minimum lifetime, this means that F' does in fact have a shorter lifetime than F. □

In fact, from Lemma 5.3, it directly follows that the longest lifetime spanning forest is exactly the maximum spanning forest of the graph weighted with τ:

Corollary 5.4. *The longest lifetime spanning forest of a graph G is the maximum spanning forest of G weighted with τ.*

Proof. By Lemma 5.3, the minimum lifetime edge in each cycle is not in the longest lifetime spanning forest: For cycles containing exactly one non-forest edge, this is a direct consequence of the lemma. For cycles containing more than one non-forest edge, all other non-forest edges can directly be replaced by the path between their endpoints in the longest lifetime spanning forest instead, including only longer lifetime edges into the cycle and creating a cycle with only one non-forest edge. This method can be used to iteratively remove edges from the graph until a forest remains, which is the longest lifetime spanning forest. This is equivalent to applying the red coloring rule by Tarjan [Tar83] to the graph weighted by τ. All remaining edges of the spanning forest can then be colored blue by the blue coloring rule. Tarjan [Tar83] has then shown that the maximum spanning forest then contains all the blue edges but none of the red ones. This means that the maximum spanning forest is exactly the longest lifetime spanning forest. □

5.3 Priority Queue Dynamic

Similarly to how the algorithm for the stack dynamic update sequence can maintain the longest lifetime spanning forest, this thesis now shows that this is also possible for the priority queue dynamic update sequence. Note that while a forest edge in the stack dynamic update sequence can't become a non-forest edge, this is not the case in the priority queue dynamic update sequence. It is therefore first shown how to compute the new longest lifetime spanning forest after an edge insertion and deletion.

Lemma 5.5. *Let F be the longest lifetime spanning forest of G, where the edges are totally ordered by τ. When inserting $uv \notin E$ with $\tau(uv) \in \mathbb{R} \setminus \tau(E)$ into G producing G', the longest lifetime spanning forest F' of G' is given by:*

1. *If u and v are in different connected components of G, then $F' = F \cup \{uv\}$.*
2. *Else, if $\tau(uv) < \tau(xw)$ for all $xw \in \pi_F(u, v)$, then $F' = F$.*
3. *Else, let xw be the edge in $\pi_F(u, v)$ with minimum τ, then $F' = F \setminus \{xw\} \cup \{uv\}$.*

Proof. Assume first that u and v are in different connected components of G. Then F' must include uv, as it would otherwise not be a spanning forest. On all other edges $E \setminus \{uv\}$, F' must match F as both are required to be spanning forests and therefore have the same number of edges, and due to Lemma 5.3 have the same set of non-forest edges.

Assume now that u, v are already in the same connected components of G. Therefore, the insertion of uv must create exactly one non-forest edge, which is either uv or in $\pi(u, v)$. By Lemma 5.3, the edge that is not in the spanning forest is the edge in $\pi(u, v) \cup \{uv\}$ with minimum τ. This is exactly what happens in the cases 2 and 3. Similar to what was shown above, the forest F' must match F for all edges not in the cycle $\pi(u, v) \cup \{uv\}$. □

Note that Lemma 5.5 is analogous to how the minimum spanning forest can be maintained incrementally [EGI99]. This is due to Corollary 5.4 showing that the longest lifetime spanning forest is just the maximum spanning forest where edges are weighted with τ.

We now show that the longest lifetime forest can also be maintained when removing an edge:

Lemma 5.6. *Let F be the longest lifetime spanning forest of G, where the edges are totally ordered by τ. When deleting the edge $uv \in E$ from G with minimum τ, producing G', the longest lifetime spanning forest F' of G' is given by:*

1. *If $uv \notin F$, then $F' = F$.*
2. *If $uv \in F$, then $F' = F \setminus \{uv\}$.*

Proof. Assume first that $uv \notin F$. Note that in this case, the longest lifetime forest F' of G' is also a spanning forest of G, and F is also a spanning forest of G'. Assume $F \neq F'$, and F' has longer lifetime than F. But then F' would be a longer lifetime spanning forest than F in G. Analogously, if F has longer lifetime than F', then F would be a longer lifetime spanning forest than F' in G'. Therefore, $F = F'$.

Assume now that $uv \in F$. Assume first that $F' = F \setminus \{uv\}$ would not be a spanning forest of G'. Let F'_u and F'_v be the connected components of F' containing u and v. Due to the assumption that F' is not a spanning forest of G', there is an edge in $e \in G'$ connecting F'_u and F'_v. This edge also exists in G and is a non-forest edge. It also has a longer lifetime than uv, as uv is currently being removed while e still is in the graph. But then, swapping out uv with e in F leads to another spanning forest, which has longer lifetime compared to F. This contradicts F being the longest lifetime spanning forest. Therefore, F' is a spanning forest of G'. Similar to what was shown in Lemma 5.5, assuming that F' is not the longest lifetime spanning forest would lead to the conclusion that F was not the longest lifetime spanning forest. Therefore, F' is the longest lifetime spanning forest of G'. □

Given Lemmas 5.5 and 5.6, the main result of this chapter can now be shown.

Theorem 5.7. *There is a priority queue dynamic algorithm for the connectivity problem with $O(\log n)$ update and query time.*

Proof. The data structure maintains the underlying graph, the edges as a balanced binary search tree ordered by τ, and the longest lifetime spanning forest as a top tree.

Whenever an edge is inserted, it updates the underlying graph, inserts the edge into the binary search tree in $O(\log n)$ time, and updates the longest lifetime spanning forest as given in Lemma 5.5. Whenever an edge is removed, it removes the first element of the

binary search tree in $O(\log n)$ time, removes that edge from the underlying graph, and updates the longest lifetime spanning forest as given in Lemma 5.6. For every query whether two vertices u and v are connected, check whether u and v are in the same tree of the longest lifetime spanning forest.

The correctness of insertion and deletion directly follows from Lemmas 5.5 and 5.6. The correctness of queries directly follows as the forest this algorithm maintains is a spanning forest. It therefore suffices to show that the algorithms mentioned in Lemmas 5.5 and 5.6, and querying whether two vertices are in the same tree of the spanning forest can be implemented in logarithmic time.

Insertion requires finding out whether the two endpoints of an edge are in the same spanning tree, finding the edge on a tree path with the shortest lifetime, and removing and inserting an edge into the tree. Edge removal requires removing an edge from the spanning forest. Queries require checking whether two vertices are in the same spanning forest. By Lemma 2.1, all this can be done in logarithmic time using top trees. □

This in particular also shows an amortized logarithmic update and query time for queue dynamic connectivity. An algorithm for the priority-queue-with-j-slots dynamic update sequence for constant $j \in \mathbb{N}$ also directly follows from Lemma 4.8.

5.4 Priority Queue Dynamic Lower Bound

As already previously mentioned, Pătraşcu and Demaine [PD05] have shown a logarithmic lower bound per operation for fully dynamic connectivity. In particular, they showed that $\max\{t_u, t_q\} = \Omega(\log n)$, where t_u is the update and t_q is the query time of any algorithm. They have shown this lower bound in the cell-probe computation model, which is a generalization of the WORD-RAM model this thesis assumes. In this section, we argue that their lower bound proof also applies to the priority queue dynamic update sequence. Our priority queue dynamic connectivity methodology developed in Section 5.3 is therefore optimal in regard to $\max\{t_u, t_q\}$.

We first give an overview of the proof for the lower bound for fully dynamic connectivity as presented by Pătraşcu and Demaine [PD05], refer to their paper for details regarding the proof. We then show how the proof needs to be modified to work for the priority queue dynamic update sequence.

Their proof of the lower bound for fully dynamic connectivity is based on a similar lower bound for the partial-sums problem. This problem requires maintaining an array $A[1 \dots n]$ of integers with support for the following operations [PD05, Section 2.1]:

- update(k, Δ): Set $A[k] = \Delta$.
- sum(k): Return the partial sum $\sum_{i=1}^{k} A[i]$.
- select(Σ): Return the index i such that $\text{sum}(i-1) < \Sigma \leq \text{sum}(i)$.

For the proof, an additional operation verifySum(i, Σ) is required, which checks whether $i = \text{select}(\Sigma) = \text{select}(\Sigma - 1) + 1$ [PD05, Section 6.1]. Their proof for the partial-sums problem works for any additive group with a sufficient number of elements. This partial-sums problem can then be translated to a connectivity problem of a graph with $n+\sqrt{n}$ nodes and n edges, arranged in $\sqrt{n}+1$ columns and $\sqrt{n}$ rows, where the group of the partial-sums problem is the permutation group on $\sqrt{n}$ elements. A permutation π_x is represented in the

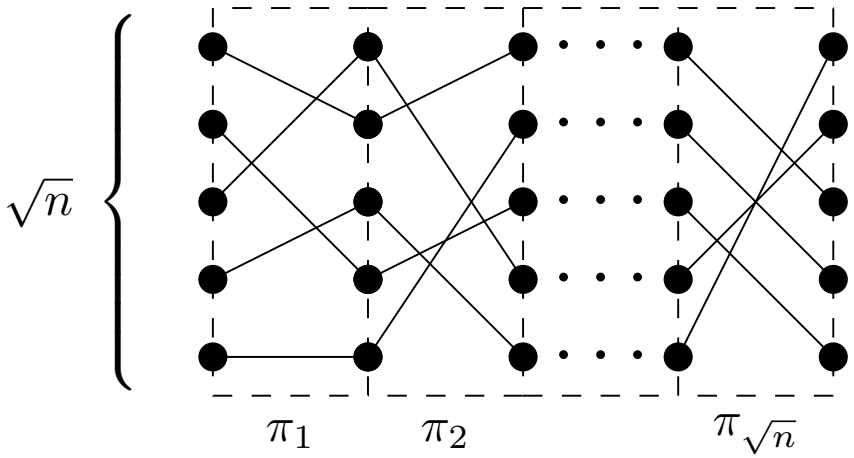

Figure 5.2: Visualization of the graph from a partial-sum instance. [PD05]

graph as a permutation box between column x and $x+1$, which means the vertex (x, y_1) in the grid is connected to the vertex $(x+1, y_2)$ if and only if $\pi_x(y_1) = y_2$. See Figure 5.2 for a visual example. An update(x, π) of the partial-sums problem is then done by removing all $\sqrt{n}$ edges of the permutation box x, and inserting new edges corresponding to the updated permutation. A verifySum(x, π) is emulated by $\sqrt{n}$ connectivity checks between node $(1, y)$ and $(x+1, \pi(y))$ for all $y \in \{1, \ldots, \sqrt{n}\}$.

To show the lower bound for the partial-sums problem, the sequence of operations is alternated between update and verifySum. The index chosen by both operations is always chosen uniformly at random, as is the element for the update. The element for the verifySum is chosen such that the result is always true. This does not trivialize the problem, as the algorithm has no guarantee that verifySum should indeed return true, and the cells the algorithm reads need to certify the result. They then show for a long enough sequence of operations a lower bound on the expected total time. This is proven by considering two adjacent intervals of operations, and bounding the number of reads done in the second interval that were last written in the first interval. The lower bound proof is in particular given a random variable G that contains all indices touched by every operation and the value for every update operation except for the values in the first interval.

We now argue that this proof also works for the priority queue dynamic update sequence. This leads to the following result:

Theorem 5.8. *Any data structure in the WORD-RAM model for the priority queue dynamic connectivity problem with update time t_u and query time t_q satisfies $\max\{t_u, t_q\} = \Omega(\log n)$. This holds under amortization, Las Vegas style randomization, or Monte Carlo randomization with error probability $n^{-\Omega(1)}$.*

Proof. We modify the proof given by Pătraşcu and Demaine [PD05]. We can use most of their proof, but there are a few steps that break for the priority queue dynamic update sequence.

The first problem we have is that the indices used for an update-operation are chosen uniformly at random. But as we have a lower bound on expected runtime over all sequences of operations, there in particular is a sequence of operations where this lower bound holds. We fix this sequence of operations. This sequence of operations on the partial-sum can then be translated to a fully dynamic update sequence for the connectivity problem. As we

have the entire fully dynamic update sequence offline, this can easily be converted into a priority queue dynamic update sequence with the same number of insertions and deletions.

The second problem is that having the priority queue dynamic update sequence gives the algorithm additional information compared to the fully dynamic one, which could potentially be used to improve the runtime. But as the proof of the lower bound for the partial-sum problem actually has access to the entire sequence of update indices given in the random variable G, the lower bound proof still holds. We note that Pătraşcu and Demaine [PD05, Section 5.6] similarly argued that the sequence of update indices can be assumed to be given to the algorithm in its entirety for their lower bound on the partial-sum problem where queries are done using the sum-operation instead of the verifySum-operation. □

Our algorithm developed in Section 5.3 is therefore optimal. We finally note that this proof does not apply to stack and queue dynamic connectivity. While the priority queue dynamic update sequence is able to do the same edge insertions and deletions given the entirety of the update sequence in advance, this does not hold for the stack and queue dynamic update sequences. For the stack dynamic update sequence, we have already shown in Section 5.1 that the lower bound presented in this section can be beaten. It is unclear whether this lower bound can also be modified to hold for the queue dynamic update sequence, or whether there exists a more specialized queue dynamic algorithm that can be implemented in $o(\log n)$ update and query time.

5.5 Deque Dynamic

We present deque dynamic connectivity next. As a priority queue already allows insertion of edges at the front and back, as well as deletion at the front, the main issue to resolve is removal of edges from the back of the deque. When maintaining the longest lifetime spanning forest, the backmost edge of the deque is always a forest edge. This is problematic, as there may be further non-forest edges in the deque that can reconnect the disconnected components after removal of the backmost edge. To resolve this issue, maintain for every forest-edge a fallback edge, which is included into the forest in case the forest-edge is removed. The new edge of course also possibly requires a fallback edge, which therefore leads to maintaining a list of fallback edges for each forest edge. Correctly maintaining the list of fallback edges is the main issue to resolve for deque dynamic connectivity. As Theorem 5.9 shows, this is possible to do without increasing the update or query time.

Theorem 5.9. *There is a deque dynamic algorithm for the connectivity problem with an amortized $O(\log n)$ time per update and query.*

Proof. Name the maintained deque Q. Define one end of Q as the front and one as the back. The main idea for correctly maintaining the list of fallback edges is maintaining two spanning forests, one as far in the front as possible, and one as far in the back as possible. When a new edge replaces an old one in a spanning forest, the old one is then the fallback edge for the new edge.

More specifically, maintain Q and maintain a numbering of edges $\tau : E \to \mathbb{R}$ that is strictly monotonically increasing in Q. This is easy to do by numbering an element inserted into the empty queue with 0, and when inserting an edge in the front of a non-empty queue, assigning it the number of the previous first deque element minus one, and when inserting in the back, assign it the previous last number plus one. In order to bound the values of τ such that operations can be done in constant time in the WORD-RAM model, reassign

all values of τ after $O(n^2)$ operations such that the minimum τ is 0 afterwards. This only takes amortized constant time.

Maintain furthermore the shortest lifetime spanning forest F and longest lifetime spanning forest B of G based on the numbering τ. We show later that we can maintain a partition of the edge set into *fallback deques*. These fallback deques have the property that the edges at the front of any fallback deque are exactly the edges in F, and analogously that the edges at the back of any fallback deque are exactly the edges in B. As every edge is contained in exactly one fallback deque, we can store a reference from each edge to the fallback deque it is contained in.

It is now described how to insert an edge uv at the front of Q. Inserting edges from the back is analogous, but uses the back spanning forest and the back of the fallback deques instead.

1. If u and v are in different connected components of F, then add uv to both F and B. Create a new fallback deque only containing uv.
2. If u and v are in the same connected component of F, find the edge $xw \in \pi(u, v)$ with maximal τ. Swap out xw with uv in the front spanning forest, and add uv in the front of the fallback deque that started with xw.

Regarding deleting an edge uv, it is also only described how to do that in the front of Q. The back works analogously.

1. If the fallback deque starting with uv only contains uv, remove uv from F and B and delete the fallback deque containing uv.
2. Otherwise, remove uv from its fallback deque. Let xw be the next element. Swap out uv with xw in F.

For querying whether two vertices are connected, just query whether they are in the same spanning tree in F. As both F and B are spanning forests of G, B could be used for querying as well. It is easy to see that all operations described here can be implemented using top trees in logarithmic time. The remainder of this proof is therefore focused on the correctness. For this, the following invariants are shown to always hold for the data structure:

1. F and B are the shortest and longest lifetime spanning forests regarding τ.
2. Let e and e' be a pair of adjacent vertices in a fallback deque, with e being before (after) e'. Assume one executes remove operations from the front (back) of Q until e is the first (last) element of Q. Let the forests F' and B' be the spanning forests after the remove operations. If we execute another remove operation from the front (back) of Q, removing e, then the resulting front (back) spanning forest is $F' \setminus \{e\} \cup \{e'\}$ ($B' \setminus \{e\} \cup \{e'\}$).
3. Each edge is in exactly one fallback deque. The set of edges at the front (back) of the fallback deques are exactly the edges in F (B).

Only insertions and deletions in the front of Q are shown, the back works analogously. Invariant 3 follows trivially by inspection of the algorithm.

Assume first that an edge is inserted that connects two different spanning trees in F. Invariant 1 follow analogously to the proof of Theorem 5.7. Invariant 2 is trivial, as the only changed fallback deque contains only one element.

Assume an edge is inserted, where both endpoints are in the same spanning tree in F. Invariant 1 again directly follows from the proof of Theorem 5.7. Regarding invariant 2,

the only pair of edges that does not follow from the invariant holding in the previous data structure are the newly inserted edge e, and the edge e' that was removed from F due to insertion of e. One needs to show that both removal of e would include e' in the front spanning forest and the other direction that removal of e' and all edges in Q after e' would include e in the back spanning forest. Replacing e with e' would just undo the changes done to F by inserting e. By invariant 1 holding before insertion of e, the resulting forest from replacing e with e' would therefore also be the shortest lifetime spanning forest. For the other direction, note that removing all edges after e' in Q, and then removing e' leads to e being the only edge in its fallback deque. By invariant 3, it therefore has to be in the back spanning forest. This therefore shows invariant 2.

Assume an edge uv is deleted from F that is the only edge in its fallback deque. By invariant 3, this means the edge uv was in both F and B. By invariant 1, uv is the only edge connecting F_u and F_v in G, where F_u and F_v are the connected components of F containing u and v after removal of uv. This therefore shows that removal of uv also disconnects a connected component G. Removing uv from both F and B therefore maintains invariant 1 similar to Theorem 5.7. Invariant 2 follows trivially.

Assume an edge e is deleted from F that has a fallback edge e'. By invariant 2 holding before removal of e, invariant 1 holds after removal of e. Invariant 2 again is trivial as it also held before removal of e. □

This in particular shows that 2-stack dynamic connectivity can be solved in logarithmic time per operation. By interpreting the deque as either a stack or a queue, this also shows that it is possible to maintain the shortest lifetime spanning forest in logarithmic time for both of these update sequences. It is not clear whether maintaining the shortest lifetime spanning forest is similarly possible for a priority queue dynamic update sequence, as it could require inserting edges in the middle of the fallback deques, and could even move edges between fallback deques.

Note that the only place in the proof of Theorem 5.9 that uses amortization is maintaining τ. One can alternatively use a worst-case constant time order data structure either by Bender et al. [BCD+02] or by Dietz and Sleator [DS87]. This also requires modifying the maintained top trees to maintain the maximum and minimum edges on the cluster path based on the order data structure directly, instead of maintaining the maximum and minimum edge lifetime. This leads to a worst-case logarithmic connectivity algorithm for the deque dynamic update sequence.

5.6 Corollaries

This chapter finishes by giving corollaries that directly follow from the ability to maintain a spanning forest in restricted update sequences efficiently.

Cycle Detection

We first note that it is easy to check whether a given graph contains a cycle, by just checking for the existence of non-forest edges.

Corollary 5.10. *When maintaining a spanning forest, one can check whether the graph contains a cycle in constant query time, and update in the same time as required to maintain the spanning forest. One can additionally return a cycle in $O(n)$, or if the spanning forest is maintained as a top tree in $O(l + \log n)$ where l is the length of the largest cycle.*

Update Time	Reference
$O(\sqrt{n})$	[EGI93]
$O((\log n)^3)^{a\,r(l)}$	[HK99]
insert: $O(\alpha(n))^a$, remove: $O(n \log(m/n))$	[EGIN92]

Table 5.2: Summary of dynamic bipartiteness algorithms. For the runtimes, a superscript a refers to amortization, and $r(l)$ refers to Las Vegas randomization.

Proof. This follows directly from the fact that there exists a cycle in the graph if and only if there is an edge not in the spanning forest. One can easily keep track of the total number of edges in the graph compared to the number of edges in the spanning forest.

Returning a cycle can be done by taking an arbitrary non-forest edge uv and returning $\pi(u, v) \cup \{uv\}$. Finding $\pi(u, v)$ for general forests can be done using a depth-first search. For top trees, this is possible in $O(\log n + l)$ time using Lemma 2.1. □

The improved runtime bound for returning a cycle in Corollary 5.10 therefore applies to the priority queue dynamic and the deque dynamic update sequences. It does not directly apply to the stack dynamic update sequence in amortized $O(\log n / \log \log n)$ time per operation. But by emulating the stack dynamic update sequence using the priority queue dynamic update sequence, it again applies while worsening the runtime to $O(\log n)$.

Connected Components

One can furthermore return all vertices connected to a queried vertex efficiently, as well as maintain the number of connected components:

Corollary 5.11. *When maintaining a spanning forest, one can return all nodes in the same connected component as an arbitrary vertex v in time linear in the number of vertices returned.*

Proof. This can easily be done by a depth-first search in the spanning forest starting at v. □

Corollary 5.12. *When maintaining a spanning forest, one can maintain the number of connected components of the graph in constant query time, and the same update time as required to maintain the spanning forest.*

Proof. This can be done by initially having n connected components, if two components merge, decrease the count by one, and if a forest-edge is removed without replacement, increase the count by one. □

One can therefore maintain the number of connected components in $O(\log n / \log \log n)$ amortized time per operation for the stack dynamic update sequence, as well as $O(\log n)$ time per operation for the priority queue and queue dynamic update sequences.

Bipartiteness

By maintaining the number of connected components, and using results found in related work, we can directly give an algorithm for checking whether the underlying graph is bipartite. We give related work in this area of research in Table 5.2. As only a single Boolean value is maintained, the query time of such algorithms is constant and is omitted.

Problem	Update Time	Reference
incremental MSF	$O(\log n)$	[EGI99]
MSF	$O(\sqrt{n}\log(m/n))$	[EGIN92]
MSF	$O(\sqrt{n})$	[EGI99]
MSF	$O(\sqrt{n})$	[EGI93]
MSF	$O((\log n)^4/\log\log n)^a$	[HRW14]
MSF	insert: $O(\log n)$, remove: $O(n\log(m/n))$	[EGIN92]
MSF in plane graphs	$O(\log n)^a$	[EIT+92]
k-weight MSF	$O(k(\log n)^3)^{a\,r(l)}$	[HK99]
$(1+\varepsilon)$-approximate MST	$O(((\log n)^3\log U)/\varepsilon)^{a\,r(l)}$	[HK99]

Table 5.3: Summary of dynamic minimum spanning forest algorithms. For the runtimes, a superscript a refers to amortization, and $r(l)$ refers to Las Vegas randomization. U is the maximum weight of an edge.

Corollary 5.13. *When maintaining the number of connected components of the graph, one can maintain whether the graph is bipartite in constant query time, and the same update time as required to maintain the number of connected components.*

Proof. This follows directly as shown by Crouch et al. [CMS13], Ahn et al. [AGM12] and McGregor [McG14]. They show that a graph is bipartite if and only if the number of connected components of the cycle double cover[1] of the graph is twice the number of connected components of the graph. For a graph $G = (V, E)$, its cycle double cover $D(G) = (V', E')$ contains vertices $v_1, v_2 \in V'$ for all $v \in V$, and $u_1v_2, u_2v_1 \in E'$ for all $uv \in E$. It is easy to see that one can maintain the cycle double cover and also run the algorithm for maintaining the number of connected components on the cycle double cover. □

This therefore allows maintaining a Boolean value indicating whether the current underlying graph is bipartite in an amortized $O(\log n/\log\log n)$ time for the stack dynamic update sequence, and $O(\log n)$ time for the priority queue and deque dynamic update sequences.

Minimum Spanning Forest

We now investigate maintaining a minimum spanning forest (MSF). We give related literature in this area of research in Table 5.3. Note that when maintaining a minimum spanning forest, the query is now whether a specific edge is in the forest or not. All related work in this area can query this property in constant time, we therefore omit query time. We use results by Henzinger and King [HK99] to maintain k-weight and $(1+\varepsilon)$-approximate minimum spanning forests for restricted update sequences. Their idea for maintaining the k-weight minimum spanning forest is maintaining k connectivity instances, one for each weight. Each instance contains the minimum spanning forest F along with all edges of one weight. To insert an edge e, insert it into its corresponding connectivity instance based on its weight. If it connects previously disconnected components, add it to the minimum spanning forest and to all other connectivity instances. Otherwise, find the maximum weight edge e' in the minimum spanning forest between its two endpoints. If the weight of the e is greater than the weight of e', e is not inserted into the minimum spanning forest. Otherwise, the e replaces e' in the minimum spanning forest, and all connectivity instances require an update. For deletion of edge uv, delete uv in its corresponding connectivity

[1] McGregor [McG14] calls this graph the bipartite double cover.

instance. If uv was in the spanning forest, delete uv from all connectivity instances. We call the two resulting trees from the deletion of uv from F F_u and F_v. To check whether another edge replaces uv, search through all connectivity instances and try to find one in which u and v are still connected. If no such instance is found, deletion of the edge splits one connected component into two and no other changes need to be done. If there is such an instance, take the one with minimum weight. To find the edge replacing uv in that instance, binary search the path between u and v to find the edge with one endpoint in F_u and the other in F_v.

We can directly use this algorithm to also maintain a minimum spanning forest with k different edge weights in our model of restricted update sequences. The runtime improvements for connectivity we get from the restricted update sequences directly improve the runtime of this algorithm compared to the fully dynamic case:

Corollary 5.14. *One can maintain a minimum spanning forest, if only k different weights are used, in time $O((\log n)^2 + k \log n)$ for the priority queue and deque dynamic update sequences.*

Proof. The same proof as presented by Henzinger and King [HK99] also works here. We only analyze the runtime of their methodology using our improvements for the priority queue and deque dynamic update sequences, the correctness is analogous to their algorithm.

Inserting an edge takes $O(k \log n)$ time, as in the worst-case an edge is inserted into the minimum spanning forest and all connectivity instances require an update. Deletion of a non-MSF edge is done in $O(\log n)$ time, as only a single instance required an update. When deleting an MSF edge uv, the edge can be deleted from all instances in $O(k \log n)$ time. One can furthermore search through all instances to find the one containing a replacement edge in $O(k \log n)$ time. To actually find the replacement edge, the binary search can not be sped up and the runtime remains $O((\log n)^2)$ like in the algorithm by Henzinger and King [HK99]: One $O(\log n)$ factor due to the depth of the binary search and another $O(\log n)$ factor to check whether the endpoints of the currently considered edge are connected to u or v in the minimum spanning forest after deleting uv. Adding the replacement edge to the spanning forest again requires updating all connectivity instances in $O(k \log n)$ time. □

Henzinger and King [HK99] also showed how this algorithm could be used to maintain an $(1+\varepsilon)$-approximate MSF. The same method of rounding edge weights to powers of $(1+\varepsilon)$ also works for our algorithm:

Corollary 5.15. *One can maintain an $(1+\varepsilon)$-approximate minimum spanning forest in time $O((\log n)^2 + \frac{\log U}{\varepsilon} \log n)$ for priority queue and deque dynamic update sequences, where U is the maximum weight.*

Proof. This can be derived from the algorithm for maintaining the minimum spanning forest with k different edge weights, by rounding weights to powers of $(1+\varepsilon)$. This was in particular shown by Henzinger and King [HK99], Ahn et al. [AGM12], and Crouch et al. [CMS13]. □

In fact, for the stack dynamic update sequence, it is even possible to maintain the exact minimum spanning forest in time $O(\log n)$. This directly follows from applying Lemma 4.18 to the incremental algorithm given by Eppstein et al. [EGI99].

6. 2-Edge Connectivity and Biconnectivity

We now investigate the 2-edge and biconnectivity problems. Queries for these problems take two vertices as parameters and return whether the vertices are 2-edge connected or biconnected. Two vertices are 2-edge connected if there are two edge-disjoint paths, and biconnected if there are two vertex-disjoint paths connecting them.

We start by introducing some lemmas that describe when two edges are 2-edge connected or biconnected. These lemmas are also commonly used in related work to give algorithms for the fully dynamic case. We then separately discuss 2-edge connectivity and biconnectivity in more detail.

6.1 Preliminary Lemmas

We now give two similar lemmas, one for 2-edge connectivity and one for biconnectivity, that describe sufficient and necessary conditions for vertices to be 2-edge connected or biconnected. These lemmas were in particular given by Hershberger et al. [HRS92] for 2-edge connectivity, and by Henzinger and King [HK95], Rauch[1] [Rau92] and Holm et al. [HdLT98] for biconnectivity. For completeness, we prove the 2-edge connectivity lemma. We also give direct consequences of this lemma, which simplify our proofs later. The proof of biconnectivity works analogously and is omitted.

We first introduce some notation. Let F be a spanning forest of a graph G. Let $xw \in F$. We call xw *covered* if there is a path from x to w in $G \setminus \{xw\}$. We call a path from x to w in $G \setminus \{xw\}$ an *alternative path* from x to w. If xw is not covered, we call xw a *bridge*. We now show that two connected vertices u and v are 2-edge connected if and only if all edges in $\pi(u, v)$ are covered:

Lemma 6.1 ([HRS92])**.** *For G a graph, and F a spanning forest of the graph, two vertices u and v are 2-edge connected if and only if there is a path $\pi(u, v)$ between them in the spanning forest, and all edges in the path are covered.*

Proof. $\Rightarrow$ As u and v are 2-edge connected, they are in particular connected and $\pi(u, v)$ exists. Assume there is a bridge $xw \in \pi(u, v)$. Removing xw therefore splits the connected

[1] Note that Monika H. Rauch is the maiden name of Monika R. Henzinger [HLP95]

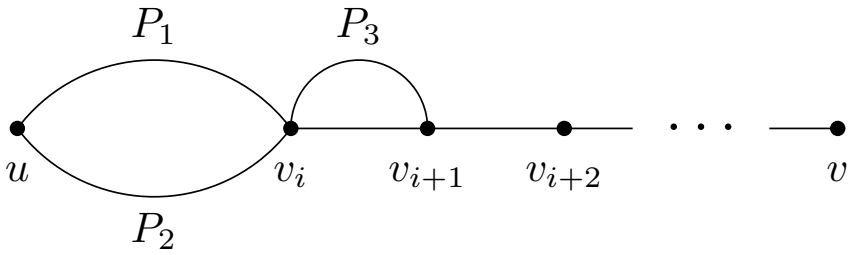

Figure 6.1: Visualization of the proof setup for Lemma 6.1.

component containing x and w into two connected components. Removing xw also separates u and v: Otherwise there exists a path from x to u, from u to v, and from v to w, contradicting that xw is a bridge. Every path between u and v must therefore go through edge xw. This means that there are no two edge-disjoint paths from u to v, u and v are therefore not 2-edge connected.

$\Leftarrow$ Show that if u and v are not 2-edge connected then they are either not connected at all or there is bridge $xw \in \pi(u, v)$. Obviously, if u and v are not connected at all, then they are also not 2-edge connected. It therefore suffices to consider the case where u and v are connected, but there are no two edge-disjoint paths. Let $v_0, ..., v_l$ be the path in the spanning forest from $u = v_0$ to $v = v_l$. Let i be the maximal index such that u and v_i are 2-edge connected, via paths P_1, P_2, like shown in Figure 6.1. Assume one of the paths, say without loss of generality P_1, contains $v_{i+1}v_i$. Then P_1 followed until v_{i+1} and P_2v_{i+1} would be two edge-disjoint paths to v_{i+1}. This contradicts our assumption that i is maximal with u and v_i being 2-edge connected. Therefore, neither P_1 nor P_2 contains the edge v_iv_{i+1}

Show now that there is no path from v_i to v_{i+1} in $G \setminus \{v_iv_{i+1}\}$. Assume otherwise, and call that path P_3. Assume P_1 and P_3 are edge-disjoint. Then P_1v_{i+1} and P_2P_3 are two edge-disjoint paths to v_{i+1}. Analogously, if P_2 and P_3 are edge-disjoint, then P_2v_{i+1} and P_1P_3 would be edge-disjoint. These cases therefore contradict our assumption that v_0 and v_{i+1} are not 2-edge connected. P_3 must therefore share at least one edge with P_1 and P_2. Let without loss of generality P_1 be the path that intersects last with P_3 in the direction from v_i to v_{i+1}. Name this edge the paths intersect at e. Construct paths P_1' and P_2' as follows: Path P_1' follows P_1 until e, at which point it follows P_3. Path P_2' follows P_2 until v_i, then goes to v_{i+1}. Both paths go to v_{i+1}. They are furthermore edge disjoint, as P_2 does not intersect with P_1 or P_3 after the edge e, and v_iv_{i+1} is contained in neither P_1 nor P_3. This again contradicts our assumption that i is maximal with u and v_i being 2-edge connected. Therefore, the path P_3 cannot exist. The edge $v_iv_{i+1} \in \pi(u, v)$ is therefore a bridge. □

We now restrict the set of paths in $G \setminus \{xw\}$ between x and w for a forest edge xw. This helps to simplify further proofs.

Lemma 6.2. *A covered forest edge xw has a path between x and w in $G \setminus \{xw\}$ that contains exactly one non-forest edge.*

Proof. Let F_x, F_w be the trees of the forest containing x and w if the edge xw is removed. Let P be a path from x to w in the $G \setminus \{xw\}$. Then there exist at least one pair of vertices $u \in F_x$ and $v \in F_w$ adjacent in P, otherwise w would be contained in F_x and the forest would contain a cycle consisting of $\pi_{F_x}(x, w)$ and xw. Then $\pi_{F_x}(x, u)$, uv and $\pi_{F_w}(v, w)$ forms a path from x to w only using one edge not in the spanning forest, uv. □

Update Sequence	Update Time	Query Time	Reference
incremental	$O(\alpha(n))^a$	$O(\alpha(n))^a$	[WT92]
fully dynamic	$O(\sqrt{n}\log(m/n))$	$O(\log n)$	[EGIN92]
fully dynamic	$O(\sqrt{n})$	$O(\log n)$	[EGI93]
fully dynamic	$O((\log n)^4)^a$	$O((\log n)^4)^a$	[HdLT98]
fully dynamic	insert: $O(\alpha(n))^a$, remove: $O(n\log(m/n))$	$O(\alpha(n))^a$	[EGIN92]
fully dynamic; planar	$O((\log n)^2)$	$O(\log n)$	[HRS92]

Table 6.1: Summary of dynamic 2-edge connectivity algorithms. For the runtimes, a superscript a refers to amortization.

Therefore, an edge uv is covered if and only if there exists a non-forest edge e with $uv \in \pi(e)$. From this, it directly follows that a non-forest edge e covers exactly the edges in $\pi(e)$.

Lemma 6.3. *Let G be a graph and F a spanning forest of G. When inserting an edge uv between already connected vertices u and v, all edges in $\pi(u,v)$ become covered. For all other edges in $F \setminus \pi(u,v)$, they are covered if and only if they were covered before insertion of uv.*

Proof. Let $xw \in \pi(u,v)$. Then xw is covered after insertion of uv due to the alternative path $\pi(u,v) \cup \{uv\} \setminus \{xw\}$. If an edge in $xw \in F \setminus \pi(u,v)$ was covered before insertion of uv, it is also covered after insertion of uv with the same path. If an edge in $xw \in F \setminus \pi(u,v)$ was not covered before insertion of uv, it is also not covered after insertion of uv: Assume otherwise, then by Lemma 6.2 there would be a path from x to w in $G \setminus \{xw\}$ using exactly one non-forest edge. As the path did not exist before insertion of uv, the non-forest edge would be uv. But then $xw \in \pi(u,v)$, contradicting that $xw \in F \setminus \pi(u,v)$. □

We now give a lemma analogous to Lemma 6.1 for biconnectivity. The proof of this lemma is analogous and is omitted.

We again first require some notation. For a graph G and a spanning forest F of G, we call a subpath of length two in F a *triple*. We say that the triple xyz *covered* if there is a path from x to z in $G \setminus \{y\}$. Then two connected vertices u and v are covered if and only if all triples in $\pi(u,v)$ are covered:

Lemma 6.4 ([HK95, HdLT98, Rau92])**.** *For G a graph, and F a spanning forest of the graph, two vertices u and v are biconnected if and only if there is a path between them in the spanning forest, and every triple xyz on $\pi(u,v)$ is covered.*

Note that for a fixed vertex y, the relation specifying the pairs of neighbors of y that are connected in $G \setminus \{y\}$ is transitive. This means that if xyz is covered and $x'yz$ is covered, then xyx' is also covered [HdLT98]. We therefore informally say that being covered is transitive, even though it is strictly speaking not a binary relation.

6.2 2-Edge Connectivity

We now investigate the 2-edge connectivity problem. In the literature, the current best fully dynamic 2-edge connectivity algorithm has an amortized time of $O((\log n)^4)$ per operation [HdLT98]. Incremental 2-edge connectivity can be solved in amortized $O(\alpha(n))$ [WT92].

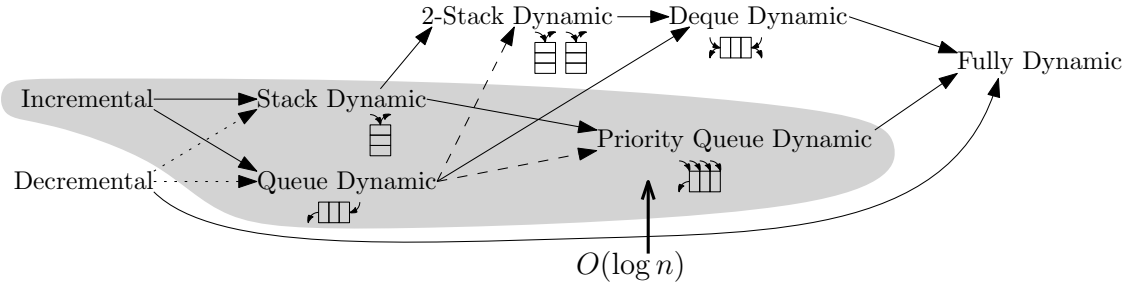

Figure 6.2: 2-Edge Connectivity Results

In the case of planar graphs, fully dynamic $O((\log n)^2)$ update and $O(\log n)$ query time is possible [HRS92]. We summarize related work for this problem in Table 6.1. This problem furthermore has a lower bound of $\Omega(\log n / \log \log n)$ per operation for the fully dynamic update sequence [HF98].

We show that a dynamic 2-edge connectivity algorithm can be implemented for the stack dynamic update sequence in logarithmic time per operation. This algorithm is then modified for the queue and priority queue dynamic update sequences, also with logarithmic time per operation. A summary of resulting runtimes is given Figure 6.2.

We first note that we cannot directly convert the incremental algorithm given by Westbrook and Tarjan [WT92] to a stack dynamic one similar to what we have done in Theorem 5.1 using the union-find data structure with backtracking. This is because their data structure does not only use a union-find data structure, but also does operations on other data structures that have amortized constant time. These amortized constant time operations lead to similar issues as discussed for Lemma 4.18.

Instead, we show how Lemmas 6.1 and 6.3 can be used in a novel algorithm for stack dynamic 2-edge connectivity in $O(\log n)$ time per update and query. The basic idea is maintenance of a spanning forest, and additional data for every forest edge on whether it is covered or not. We have already seen how the spanning forest can be maintained in Chapter 5. It therefore suffices to shown that this additional data can effectively be maintained for edge insertions and deletions.

Theorem 6.5. *There is a data structure that answers 2-edge connectivity queries during stack dynamic updates in $O(\log n)$ time per update and query.*

Proof. Maintain the spanning forest data structure via top trees, like done in Theorem 5.7. We show that we can maintain a weight for every forest edge uv that is greater than 0 if and only if uv is covered. By Lemma 6.1, two vertices u, v are therefore 2-edge connected if and only if they are connected and the minimum weight of the path in the spanning forest is greater than 0. This is easy to check via top trees by Lemma 2.1. It therefore remains to show how the weight is maintained. This follows from Lemma 6.3: When an edge is inserted into the spanning forest, it connects previously disconnected components and is therefore a bridge. The weight of such an edge should therefore be initialized to 0. Whenever an edge uv is inserted that is not in the forest, this edge is included in a fallback path for every edge in $\pi(u, v)$. Therefore, it suffices to add 1 to every edge in $\pi(u, v)$. This is possible to do via top trees in logarithmic time by Lemma 2.1. Deleting an edge that is not in the forest just reverts this operation by subtracting 1 from every edge on the path between the two endpoints in the tree. Deleting a forest edge is analogous to Theorem 5.7 and just removes the edge from the spanning forest. □

A similar result is now given for the queue dynamic update sequence. Note that applying the same methodology as for the stack dynamic update sequence in Theorem 6.5 does not directly work, as the longest lifetime spanning forest changes during the runtime of the algorithm. This in particular causes the paths between the two endpoints of non-forest edges to change over time. This leads to the weight of some edges not being decreased when removing a non-forest edge. Instead, we store for each forest-edge what the latest inserted non-forest edge is that proves that there is an alternative path.

Theorem 6.6. *There is a data structure that answers 2-edge connectivity queries under queue dynamic updates in amortized $O(\log n)$ time per update and query.*

Proof. The data structure assigns each $e \in E$ the time $\tau(e)$ it was inserted into the queue, starting with 1. Furthermore, $\min_{e \in E} \tau(e)$ is maintained. In order to bound $\tau(e)$ to allow for constant time operations, reduce $\tau(e)$ for each edge and $\min_{e \in E} \tau(e)$ by n^2 every n^2 insertions. Both $\tau(e)$ and $\min_{e \in E} \tau(e)$ are easy to maintain in amortized constant time. The algorithm furthermore maintains the longest lifetime spanning forest F with regard to τ. This can be done as described in Theorem 5.7 in $O(\log n)$ update time. Note that due to the longest lifetime property of the spanning forest, a newly inserted edge e is always inserted into the spanning forest, and ejects at most one edge e' from the spanning forest. The final data structure, for which we show that it can be maintained efficiently, is a function $s : F \to \mathbb{N}_0$ that either maps a forest edge e to 0 if e is a bridge between two components and the edge was never covered while it existed, or to the insertion time of the (possibly already removed) non-forest edge with maximum τ that at some point showed that there was an alternative path between the two endpoints of e not using e. Note that s also needs to be reduced by n^2 every n^2 edge insertions.

It is now described how s is updated on edge insertion. Assume e is inserted that connects two previously separated connected components. This means that e is in particular a bridge. Initialize $s(e) = 0$. For all other edges, s remains unchanged.

Assume now that an edge e is inserted and ejects another edge e' from the spanning forest. In that case, set $s(xw) = \max\{s(xw), \tau(e')\}$ for all $xw \in \pi(e')$. This is possible to do in logarithmic time using top trees by Lemma 2.1.

It remains to show the correctness of this algorithm. For that, it suffices to show that a forest-edge uv is covered if and only if $s(uv) \geq \min_{e \in E} \tau(e)$. When that is shown, it directly follows from Lemma 6.1 that two nodes u and v are 2-edge connected if and only if $\min_{e \in \pi(u,v)} s(e) \geq \min_{e \in E} \tau(e)$. This can be verified in $O(\log n)$ time using top trees by Lemma 2.1.

Assume first that $s(uv) \geq \min_{e \in E} \tau(e)$. Let e' be the edge with $\tau(e') = s(uv)$. Let F' be the spanning forest directly after e' was ejected from the forest. Then $uv \in \pi_{F'}(e')$, and $\tau(e') < \tau(xw)$ for all $xw \in \pi_{F'}(e')$. As $\tau(e') = s(uv) \geq \min_{e \in E} \tau(e)$, e' and all edges of $\pi_{F'}(e')$ are still in G. The edges $\{e'\} \cup \pi_{F'}(e') \setminus \{uv\}$ therefore form an alternative path for uv.

Assume now that $s(uv) < \min_{e \in E} \tau(e)$, but there exists an alternative path for uv in the current graph. By Lemma 6.2, there is at least one alternative path using exactly one non-forest edge. Let e' be the non-forest edge with maximal τ that induces such an alternative path. By the assumption, we have $s(uv) < \min_{e \in E} \tau(e) \leq \tau(e')$. Let F' again be the spanning forest directly after e' was ejected from the spanning forest. Then $uv \notin \pi_{F'}(e')$, otherwise $s(uv)$ would have been updated such that $s(uv) \geq \tau(e')$. This means that the path $\pi_{F'}(e')$ was changed over time into the path $\pi_F(e')$. One of this change introduced uv into the path. Let e'' be the edge that got ejected from the path when uv was inserted,

and F'' be the spanning forest directly after e'' got ejected. There holds $\tau(e'') > \tau(e')$, as e'' was in $\pi(e')$ at some point of the algorithm. Then, $uv \in \pi_F(e'')$ as no updates ejected uv from F. This contradicts the choice of e' as the edge with maximum τ with $uv \in \pi_F(e')$. Therefore, uv is not covered. □

The same proof is also applicable to the priority queue dynamic update sequence, with the restriction that once an edge e with lifetime $\tau(e)$ has been removed, no edge with lifetime at most $\tau(e)$ may be inserted again. In that case, an edge insertion may either replace an edge in the longest lifetime forest, or end up being a non-forest edge itself. In both cases, set $s(xw) = \max\{s(xw), \tau(e')\}$ for all edges $xw \in \pi(e')$, where e' is the edge that was either ejected from the spanning forest, or the newly inserted edge if it was not inserted into the spanning forest.

Removing the above-mentioned restriction for priority queue dynamic update sequences is possible. This leads to the following theorem:

Theorem 6.7. *There is a data structure that answers 2-edge connectivity queries under a priority queue dynamic update sequence in amortized $O(\log n)$ time per update and query.*

Proof. As described above, Theorem 6.6 can be modified to work for the priority queue dynamic update sequence if $\min_{e \in E} \tau(e)$ is monotonically increasing over the runtime of the algorithm. It therefore suffices to find an emulation of the priority queue dynamic update sequence using such a restricted priority dynamic queue dynamic update sequence. We call the *real lifetime* of an edge the lifetime it was inserted with in the priority queue, and the *emulating lifetime* the lifetime used in the restricted priority dynamic queue dynamic update sequence.

To implement this emulation, maintain a balanced binary search tree. This tree contains an entry for every edge, which contains its real and emulating lifetime. It furthermore contains a dummy entry, which has real lifetime $-\infty$ and emulating lifetime equal to the emulating lifetime of the last removed edge. When a new edge e is inserted, insert it into the balanced search tree based on its real lifetime. Populate its emulating lifetime based on the neighbors of e in the binary search tree as follows: Note that e cannot be inserted as the first edge in the priority queue, as the dummy element always has minimum real lifetime. If it is inserted into the end of the binary search tree, set its emulating lifetime to some value bigger than the previously last edge. Otherwise, set its emulating lifetime as the average of its two neighbors in the binary search tree.

For an edge removal, remove the first edge from the tree and update the emulating lifetime of the dummy entry.

This methodology obviously maintains the order of elements of the priority queue. Furthermore, it can never happen that the emulating lifetime of an inserted edge is smaller than the emulating lifetime of an already removed edge. This therefore shows correctness of the emulation.

The logarithmic runtime directly follows from the runtime of the queue dynamic algorithm and the usage of a balanced binary search tree data structure for the emulation. □

The above proof assumes storage and computation with arbitrarily precise real numbers in constant time. This restriction can be lifted when using the order data structure by Bender et al. [BCD+02] or by Dietz and Sleator [DS87]. Applying their data structure would also require updating the 2-edge connectivity algorithm to store for each edge not

Update Sequence	Update Time	Query Time	Reference
incremental	$O(\alpha(n))^a$	$O(\alpha(n))^a$	[WT92]
fully dynamic	$O(m^{\frac{2}{3}})^a$	$O(1)$	[Rau92]
fully dynamic	$O((\log n)^4)^a$	$O((\log n)^4)^a$	[HdLT98]
fully dynamic	$O((\log n)^4)^{a\,r(l)}$	$O((\log n)^2)$	[HK95]
fully dynamic	insert: $O(\alpha(n))^a$, remove: $O(n \log(m/n))$	$O(\alpha(n))^a$	[EGIN92]
fully dynamic; planar	$O(\sqrt{n \log n})^a$	$O((\log n)^2)$	[Rau92]

Table 6.2: Summary of dynamic biconnectivity algorithms. For the runtimes, a superscript a refers to amortization, and $r(l)$ refers to Las Vegas randomization.

the maximum lifetime of an edge covering it, but instead the edge itself. This also requires taking care of the special case when an edge is added and removed multiple times.

Note that for deque dynamic 2-edge connectivity, a similar methodology as applied in the proof of Theorem 6.6 does not easily combine with the ideas given in Theorem 5.9. The problem is that a non-forest edge, which may be used by forest edges to guarantee alternative paths, can become a forest edge. In this case, some forest edges do not have the correct information on whether there is an alternative path between its endpoints or not.

To end this section, we note that the algorithms presented in this section can also be used to provide more detailed queries, instead of just a Boolean answer:

Corollary 6.8. *The queries from Theorems 6.5, 6.6 and 6.7 can also be modified to do exactly one of the following, depending on whether the queried vertices are 2-edge connected, connected or disconnected:*

- *Answer that the vertices are 2-edge connected.*
- *Give a single edge whose removal would disconnect the two vertices.*
- *Answer that the vertices are not connected at all.*

Proof. It is easy to find out via top trees whether the vertices are connected at all. If they are connected, but the path between them contains an edge uv with weight 0 for the stack dynamic update sequence, or an edge with $s(uv) < \min_{e \in E} \tau(e)$ for the queue or priority queue dynamic update sequences, then uv is a bridge between the queried vertices and would disconnect them if it was removed. Such an edge uv can be found using Lemma 2.1. If they are connected and there is no such edge, then the vertices are 2-edge connected. □

6.3 Biconnectivity

We now consider the biconnectivity problem. For the fully dynamic update sequence, this can again be solved in amortized $O((\log n)^4)$ per operation [HdLT98], or with randomization in expected amortized $O((\log n)^4)$ update and $O((\log n)^2)$ query time [HK95]. In the incremental case, amortized $O(\alpha(n))$ time per operation is possible [WT92]. Related work for this problem is summarized in Table 6.2. A lower bound of $\Omega(\log n / \log \log n)$ per operation for the fully dynamic biconnectivity problem was shown by Henzinger and Fredman [HF98].

We first give an incremental algorithm for biconnectivity in amortized $O(\alpha(n) \log n)$ time per operation. While this is worse compared to the amortized runtime of $O(\alpha(n))$ given by Westbrook and Tarjan [WT92], we show that this algorithm can be converted into a stack

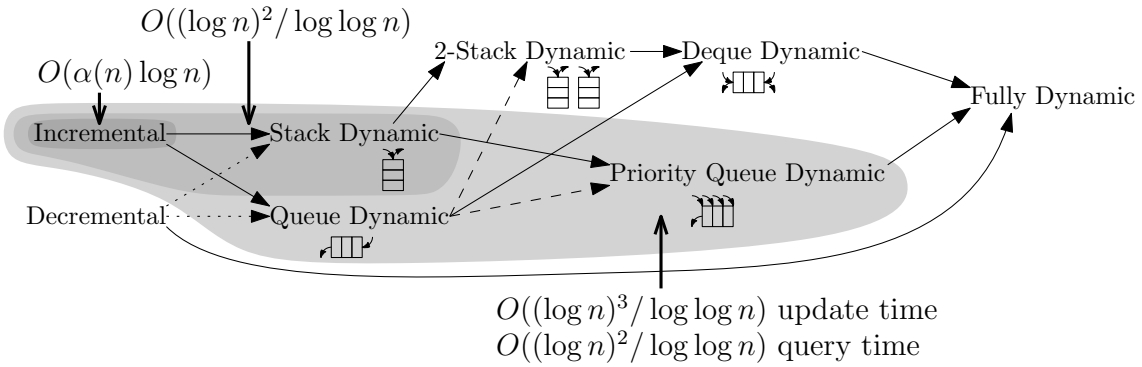

Figure 6.3: Biconnectivity Results

dynamic algorithm and a priority queue dynamic algorithm with an improved runtime compared to expected amortized $O((\log n)^4)$ update time and $O((\log n)^2)$ query time for fully dynamic biconnectivity given by Henzinger and King [HK95]. A summary of the achieved runtimes is given in Figure 6.3.

We again first note that converting the incremental amortized $O(\alpha(n))$ algorithm from Westbrook and Tarjan [WT92] using union-find with backtracking similar to Theorem 5.1 is not directly possible, as changes to their data structures are made that cannot be undone quickly in an obvious way. Instead, we give another novel algorithm based on similar ideas as presented for 2-edge connectivity in Section 6.2.

The basis of our algorithm is Lemma 6.4. As being covered is now not a property of a single edge like it was in 2-edge connectivity, it is unclear how information about being covered can be stored efficiently. But we can use transitivity to store for each vertex a union-find data structure of adjacent vertices, where vertices are in the same set if they are biconnected. This was similarly done by Holm et al. [HdLT98]. All in all, this leads to the following incremental algorithm:

Lemma 6.9. *Biconnectivity in an incremental dynamic graph can be answered in amortized $O(\alpha(n)\log n)$ query and update time.*

Proof. We maintain a spanning forest of the graph as a top tree, like done in Theorem 5.7. We show that we can maintain for each path cluster C a bit c_C indicating whether all triples on the cluster path are covered. Furthermore, maintain for each path cluster the second and second-to-last vertex in the cluster path. This can be done similar to Lemma 2.1 by maintaining the cluster path as a linked list for each path cluster.

We maintain for each vertex y a union-find data structure of adjacent vertices in the forest U_y. Two vertices x, z are in the same set in U_y if xyz is covered. We allow x and z to be in different sets of y even though the triple is covered if there is a covered path cluster containing the triple xyz in its cluster path. Whenever a forest edge uv is inserted, add u as an element to U_v and v as an element to U_u.

Overall, the algorithm maintains two invariants:

1. The bit c_C of a path cluster C is set if and only if all triples on the cluster path of C are covered.
2. If xyz is not contained in a covered cluster path, then it is a covered triple if and only if x and z are in the same set of U_y.

One can then query whether two vertices are biconnected by exposing them and checking c_C of the root cluster of the resulting top tree. Correctness directly follows from invariant 1.

When an edge is inserted connecting two different connected components of the graph, just update the spanning forest as done in Theorem 5.7. Otherwise, the inserted edge uv is a non-forest edge and therefore covers all triples on $\pi(u, v)$. Expose u and v and set c_C on the root cluster of the resulting top tree.

We now describe how c_C and the union-find data structures are maintained when the top tree changes. As described by Alstrup et al. [AHLT05], it suffices to describe how to split and join clusters.

Let us first consider splitting a cluster. Make a case distinction based on whether c_C is set and whether the split and resulting clusters are path clusters. Assume a path cluster C with c_C set is split into path clusters A and B, where $\partial C \neq \partial A$ and $\partial C \neq \partial B$. Then set the bits c_A and c_B. We furthermore merge the sets containing the vertices that are adjacent to $y \in \partial A \cap \partial B$ on the cluster path of C in U_y. These vertices are just the second-to-last vertex of the cluster path of A and the second vertex of the cluster path of B. If a path cluster C with c_C set is split into a path cluster A with $\partial C = \partial A$ and either a path or a point cluster B, set c_A. If a path cluster C with c_C not set, or a point cluster C is split, nothing needs to be done. It is easy to see how invariants 1 and 2 are maintained by splitting nodes.

It is now described how two clusters are joined. One again differentiates between the types the joined clusters have. If two path clusters A and B are joined into a path cluster C with $\partial C \neq \partial A$ and $\partial C \neq \partial B$, set c_C if and only if c_A is set, c_B is set, and the adjacent vertices x and z of $y \in \partial A \cap \partial B$ in the cluster path of C are in the same set in U_y. If at least one of the conditions does not hold, either a triple in the cluster path of A or B is not covered, or the triple centered at y is not covered. In that case, at least one triple in the cluster path of C is also not covered. If all conditions hold, the triples in the cluster path of A and B are covered due to invariant 1 holding for A and B. Furthermore, the triple centered at y is covered from invariant 2, while noting that xy and yz are in different clusters when merged, the triple xyz is therefore not already in a covered cluster path. Therefore, invariant 1 holds. Invariant 2 obviously still holds as the union-find data structures are unchanged. When joining a path cluster A with $\partial C = \partial A$ and either a path or a point cluster B into a path cluster C, set $c_C = c_A$. When joining two clusters into a point cluster, nothing needs to be done. In these two cases, invariants 1 and 2 obviously still hold.

It therefore remains to show the runtime of the algorithm. By Alstrup et al. [AHLT05], all top tree operations required for maintaining the spanning forest and querying biconnectivity of two nodes can be implemented in $O(\log n)$ joins and splits of top tree clusters. The described algorithms for splitting and joining a single cluster require a single operation for the union-find data structure in amortized $O(\alpha(n))$, and further constant time operations. The claimed runtime of amortized $O(\alpha(n) \log n)$ therefore directly follows. □

We now describe how this data structure can be modified for stack dynamic biconnectivity. This leads to the following theorem:

Theorem 6.10. *Biconnectivity in a stack dynamic update sequence can be answered in amortized $O((\log n)^2 / \log \log n)$ query and update time.*

Proof. Modify the proof of Lemma 6.9. Instead of a union-find implementation with an amortized runtime of $O(\alpha(n))$ per operation, we use the implementation allowing

backtracking by Westbrook and Tarjan [WT89] with an amortized time of $O(\log n / \log\log n)$ per operation. We furthermore store for each edge insertion which union operations are done in the union-find data structures, for which clusters c_C was changed, and how the top tree was modified using splits and joins. As only $O(\log n)$ joins and splits are done per operation, and each join and split modifies at most one c_C and U_y, this data requires $O(\log n)$ time to be stored. This leads to an overall insertion time of $O((\log n)^2 / \log\log n)$, $O(\log n / \log\log n)$ per join and split due to the choice of the union-find data structure and $O(\log n)$ many joins and splits are required to insert an edge. The correctness of insertion directly follows from the correctness of the incremental algorithm.

When removing an edge, we undo the changes done to the data structure in reverse order. This requires amortized time $O(\log n / \log\log n)$ per split and join for undoing unions using a de-union operation, and for the changes to the top tree and c_C only a constant time. We undo at most $O(\log n)$ splits and joins, leading to an amortized runtime of $O((\log n)^2 / \log\log n)$. We do the same for query operations, which undoes any changes to the top tree done by queries. The correctness of the operations is obvious as the state after undoing changes is exactly the same as the state of the data structure before the changes were done. □

Applying Lemma 4.17, this leads directly to a priority queue dynamic algorithm with an additional $O(\log n)$ factor for the update time:

Corollary 6.11. *Biconnectivity in a priority queue dynamic graph can be answered in amortized $O((\log n)^2 / \log\log n)$ query time and $O((\log n)^3 / \log\log n)$ update time.*

We can again modify the information returned by a query, similar to what we have done for 2-edge connectivity in Corollary 6.8. This time, a more detailed description is needed for how to find the vertex that would disconnect the queried vertices as the information is not stored within an edge anymore but within a vertex. Nevertheless, we can develop a similar search algorithm like done by Alstrup et al. [AHLT05].

Corollary 6.12. *The queries from Lemma 6.9, Theorem 6.10 and Corollary 6.11 can be modified to do exactly one of the following, depending on whether the queried vertices are biconnected, connected or disconnected:*

- *Answer that the vertices are biconnected.*
- *Give a single vertex, whose removal would disconnect the two queried vertices.*
- *Answer that the vertices are not connected at all.*

Proof. The cases where the vertices are biconnected or not connected at all are analogous to Corollary 6.8. We therefore just show how to find a vertex whose removal would disconnect the two queried vertices if the two queried vertices are connected but not biconnected.

We first expose the two queried vertices in the top tree. As the vertices are not biconnected, the bit c_C is not set for the resulting cluster. By invariants 1 and 2 from the proof of Lemma 6.9, this means either the triple centered at the shared boundary vertex of the two child clusters is not covered, or the cluster path of a child path cluster contains an uncovered triple. In the first case, an uncovered triple is found, and the middle vertex can be returned. In the second case, one can recurse on a child path cluster containing an uncovered triple. As the depth of the top tree is logarithmic, the recursion is bounded by $O(\log n)$. Each recursion step requires querying a union-find data structure, and further constant time operations. This was also required when exposing the two queried nodes, the runtime of this modification is therefore unchanged compared to the runtimes of the previous results. □

7. Lifetime Problems

We now define a new class of dynamic problems based on the lifetime of edges as defined for the priority queue dynamic update sequence and any of its specializations. When queried, dynamic problems in this class need to return the longest or shortest lifetime substructure as given in Definition 5.2. It is shown that some of the already given methods can be used to solve lifetime problems. Furthermore, some results given in the literature regarding dynamic substructure counting can also be translated to solving the corresponding lifetime problems.

We start by defining lifetime problems. An example of such a problem is the longest lifetime path between vertices. For that problem, each query receives two vertices u and v as parameters. In case u and v are not connected in the underlying graph, the query should return a value indicating that no path between u and v exists in G. If they are connected, the answer to the query should be the path between u and v that has the longest lifetime amongst all paths from u to v in G.

Definition 7.1. Lifetime problems *are dynamic graph problems for the priority queue dynamic update sequence or its specializations where queries imply a set $\mathcal{S} \subseteq 2^E$, and where the answer to a query is the longest or shortest lifetime substructure amongst $\mathcal{S}$.* Longest lifetime problems *are lifetime problems returning the longest lifetime substructure, while* shortest lifetime problems *are the problems returning the shortest lifetime substructure.*

As an example, a query for the longest lifetime path between u and v in the underlying graph implies $\mathcal{S}$ as the set of all paths between u and v in G. An algorithm solving the longest lifetime path problem then needs to return the longest lifetime element in $\mathcal{S}$.

Note that unlike most other problems considered for now, lifetime problems require returning a graph substructure instead of a Boolean value. As the substructures can be quite large, the size of the returned substructure can have a significant effect on the runtime of algorithms solving such problems. To unify notation, let l be the size of the returned substructure for queries.

7.1 Introductory Problems

We now investigate problems based on the longest and shortest lifetime spanning forests as an introductory example for lifetime problems. We can use our results from Chapter 5 for

maintaining the longest and shortest lifetime spanning forests to solve such problems. To be more specific, we have shown that the longest lifetime spanning forest can be maintained in a priority queue dynamic update sequence in Theorem 5.7, and the shortest lifetime spanning forest can be maintained in the stack and queue dynamic update sequences, which follows from the proof of Theorem 5.9. Returning the edges contained in a top tree can be done by Lemma 2.1. This leads to the following corollary:

Corollary 7.2. *There is an algorithm for the priority queue dynamic update sequence that can return the longest lifetime spanning forest in $O(l)$ time, while supporting updates in $O(\log n)$ time.*

The same holds for the shortest lifetime spanning forest in the stack and queue dynamic update sequences.

As already noted in Chapter 5, it is unclear whether maintaining the shortest lifetime forest is possible for the priority queue dynamic update sequence in a logarithmic runtime per operation.

We now show how we can use the longest and shortest lifetime spanning forests to return the longest and shortest lifetime path between any two vertices. In particular, it is shown that the longest and shortest lifetime path between two vertices is just the path between the vertices in the longest and shortest lifetime forests. Top trees can then be used to efficiently maintain the spanning forest and return the path between the vertices, leading to the following corollary:

Corollary 7.3. *There is an algorithm for the priority queue dynamic update sequence that can return the longest lifetime path between any two vertices (if one exists) in $O(\log n + l)$ time, while supporting updates in $O(\log n)$ time.*

The same holds for the shortest lifetime path in the stack and queue dynamic update sequences.

Proof. This is just shown for the longest lifetime spanning path for the priority queue dynamic update sequence. The shortest lifetime path for the stack and queue dynamic update sequences follow analogously from maintaining the shortest lifetime spanning forests in those update sequences. Furthermore, we have already seen how to check whether two vertices are connected in Chapter 5. This allows us to quickly answer queries between disconnected vertices. The remainder of this proof therefore considers only connected vertices.

We maintain the longest lifetime spanning forest F as described in Theorem 5.7. We show that the longest lifetime path between any two connected vertices is the path between them in F. Then, the runtime for updates directly follows from Theorem 5.7, while the runtime for queries directly follows from Lemma 2.1.

Assume otherwise that P is the longest lifetime path between u and v, containing a non-forest edge $xw \in P \setminus F$. By Lemma 5.3, taking $\pi(xw)$ instead of xw in the path P (which possibly requires simplifying the path, but this does not decrease the lifetime of the path) would only introduce longer lifetime edges into the path. This would therefore have a longer lifetime than P, contradicting the assumption that P is the longest lifetime path. Therefore, all edges in P must be forest edges. As there is only one simple path between u and v in F, P must be that path. □

Substructure(s)	Update Time	Query Time	Reference
△	$O(\sqrt{m})^a$	$O(1)$	[KNN+18]
∴ ∴ ∧ △	$O(h)^{a\,r(l)}$	$O(1)$	[ES09]
⋮ ⋮ ∣ ⋮ ∣ ∣ ⌜ ⌜ ◸ ⌜ □ ◺ ◹ ⊠	$O(h^2)^{a\,r(l)}$	$O(1)$	[EGST12]
△	$O(\sqrt{m})^a$	$O(1)$	[HHH21]
△ containing queried edge	$O(\sqrt{m})^a$	$O(\sqrt{m})$	[HHH21]
△ containing queried vertex	$O(m^{2/3})^a$	$O(m^{2/3})$	[HHH21]
⌜ containing fixed vertex	$O(1)$	$O(1)$	[HHH21]
△ ⌜ containing fixed vertex	$O(\sqrt{m})$	$O(1)$	[HHH21]
◺ □ ◹ containing fixed vertex	$O(m^{2/3})$	$O(1)$	[HHH21]
⊠ containing fixed vertex	$O(m)$	$O(1)$	[HHH21]
non-induced ⌜	$O(\sqrt{m})^a$	$O(1)$	[HHH21]
non-induced ⌜ containing queried edge	$O(\sqrt{m})^a$	$O(\sqrt{m})$	[HHH21]
non-induced ◺ □ ◹	$O(m^{2/3})^a$	$O(1)$	[HHH21]
non-induced ◺ □ ◹ containing queried edge	$O(m^{2/3})^a$	$O(m^{2/3})$	[HHH21]

Table 7.1: Summary of fully dynamic substructure counting algorithms. For the runtimes, a superscript a refers to amortization, and $r(l)$ refers to Las Vegas randomization. h refers to the h-index of the graph. "Containing queried edge" and "containing queried vertex" refers to an edge or vertex given in a query, while "containing fixed vertex" refers to the vertex being fixed before running the algorithm. Substructures are counted induced, unless noted otherwise.

We note that the methodology given in Corollary 7.3 is optimal for priority queue dynamic update sequences: It can be used to check whether two vertices are connected, which is lower bounded by $\Omega(\log n)$ per operation from Section 5.4. It furthermore needs to return l elements for queries in case they are connected, which is lower bounded by $\Omega(l)$.

Analogously, returning the longest and shortest lifetime cycles is also possible. From Corollary 5.4, it directly follows that the shortest lifetime edge in the cycle is always an edge not in the longest lifetime spanning forest. As we defined the lifetime of substructures lexicographically, comparing the lifetime of cycles therefore first compares the lifetime of the shortest lifetime edge included in the cycle. The longest lifetime cycle is therefore the cycle induced by the longest lifetime non-forest edge e in the longest lifetime spanning forest: All other cycles contain at least one non-forest edge that has a shorter lifetime than e. The same holds for the shortest lifetime cycle. This therefore directly shows the following corollary:

Corollary 7.4. *There is an algorithm for the priority queue dynamic update sequence that can return the longest lifetime cycle of the graph in $O(\log n + l)$ time, while supporting updates in $O(\log n)$ time.*

The same holds for the shortest lifetime cycle in the stack and queue dynamic update sequences.

7.2 Longest Lifetime Triangle and 4-Clique

We now investigate returning the longest lifetime subgraph matching a pattern, like returning the longest lifetime triangle or 4-clique. Relevant related work in this area is

focused on dynamically counting such subgraphs [ES09, EGST12, HHH21, KNN+18], and is summarized in Table 7.1. We first give a trivial algorithm for maintaining the longest lifetime triangle, which can then be improved using a similar idea like used by Eppstein and Spiro [ES09] for counting triangles. A similar method can also be used for maintaining the longest lifetime 4-clique, with similar modifications as done by Eppstein et al. [EGST12] for counting 4-cliques.

It is easy to get an algorithm with runtime $O(\deg(u) + \deg(v))$ for inserting an edge uv and $O(1)$ for deleting an edge when maintaining the longest lifetime triangle: For an insertion of the edge uv, one can check for any neighbor w of u or v whether it creates a new triangle and compare its lifetime with the current longest lifetime triangle. For edge deletion, note that an edge from the longest lifetime triangle is only removed if no other triangle exists after deletion.

We can speed up this trivial algorithm with ideas similar to what was presented by Eppstein and Spiro [ES09] for dynamic triangle counting. We first give an overview of their methodology. They maintain a partition of vertices into high-degree and low-degree vertices, and for each edge insertion count the number of newly created triangles differently based on whether the third vertex is a high degree or a low degree vertex. For high degree vertices, one can directly count the newly created triangles like done in the trivial algorithm. For low degree vertices, the number of newly created triangles can be looked up by maintaining for every pair of vertices the count of low degree vertices connecting to both of them. The runtime of their algorithm is then amortized $O(h)$, where h is defined as the maximum h such that there are h vertices with degree at least h, which is also called the *h-index* of the graph. This measurement is a way to describe how uniformly edges are distributed in the graph. It was shown by Eppstein and Spiro [ES09] that $h \in O(\sqrt{m})$ and in many real-world graphs $h \in O(n^{\frac{1}{3}})$ or $h \in O(n^{\frac{1}{4}})$.

To give an intuition about this measurement, we compute the h-index for two example graphs with the same number of vertices and edges: The star graph with n edges has an h-index of 1, as there is a vertex that has degree at least 1, but as there are no two vertices with degree 2. On the other hand, a $\sqrt{n}$-clique with additional $n - \sqrt{n}$ isolated vertices has an h-index of $\sqrt{n} - 1$, as all $\sqrt{n}$ vertices in the clique have degree $\sqrt{n} - 1$. Note in particular that a runtime of amortized $O(h)$ means that the algorithm can easily tolerate many low-degree vertices, as well as a few very high-degree vertices. Instead, the runtime mainly depends on the number of intermediate-degree vertices [ES09].

We now use similar ideas as also presented by Eppstein and Spiro [ES09] for counting triangles to show that the longest lifetime triangle can be maintained in amortized $O(h)$ time per update.

Theorem 7.5. *There is an algorithm that can maintain the longest lifetime triangle of the graph under a priority queue dynamic update sequence in amortized $O(h)$ update time.*

Proof. Maintain a partition of the vertex set into high-degree H and low-degree $V \setminus H$ vertices as done by Eppstein and Spiro [ES09] in amortized constant time. Note that $|H| \in O(h)$, and that vertices in $V \setminus H$ have $O(h)$ neighbors. Furthermore, vertices move between high-degree and low-degree on average $O(1/h)$ times per update. Our proof proceeds as follows: We first investigate how the longest lifetime triangle changes by edge insertion and deletion. For edge insertion, we enumerate all triangles that could possibly be

the new longest lifetime triangle. It is shown that enumerating this list requires maintaining a mapping P, which is described in further detail below. We then show that P can be efficiently maintained when inserting or removing edges, as well as vertices moving between H and $V \setminus H$.

We first investigate how the deletion of the shortest lifetime edge uv updates the longest lifetime triangle. Assume uv is not contained in the longest lifetime triangle. Then obviously the old longest lifetime triangle is still the longest lifetime triangle. Assume now that uv is contained in the longest lifetime triangle. Then, after deletion of uv, the graph contains no more triangles, otherwise such a triangle would have longer lifetime than the previous longest lifetime triangle.

We now consider how an insertion of the edge uv can update the longest lifetime triangle. For that, we compute a list of $O(h)$ many triples of vertices, which we call *potential longest lifetime triangles*. We show that the new longest lifetime triangle is included in this list of potential longest lifetime triangles. One can then compute the new longest lifetime triangle by removing all triples in the list that do not form a triangle, and from the remaining triangles find the one with the longest lifetime. Due to the length of the list, one can then compute the longest lifetime triangle given this list in $O(h)$ time. It therefore suffices to show how this list can be computed for inserting an edge uv.

Consider first all triangles that do not contain the edge uv. These triangles were also present before insertion of uv, and did not change their lifetime due to insertion of uv. This implies that it suffices to add the previous longest lifetime triangle as the only potential longest lifetime triangle not containing uv, as all other triangles not containing uv have a shorter lifetime.

All other potential longest lifetime triangles therefore contain the edge uv. Note that in that case, due to the lifetime of substructures being defined lexicographically, the lifetime of a triangle containing the edge uv follows directly from the lifetime of the other two edges in the triangle. Therefore, the longest lifetime triangle containing the edge uv (if it exists) is induced by the longest lifetime 2-path between u and v. It therefore suffices to enumerate vertices w such that we can guarantee that the longest lifetime 2-path between u and v is the path uwv for one w. Like done by Eppstein and Spiro [ES09], we make a case distinction based on whether $w \in H$ or $w \in V \setminus H$. Note that as $|H| \in O(h)$, we can just try all $w \in H$: Add (u, w, v) as a potential longest lifetime triangle for all $w \in H$. For $w \in V \setminus H$, we show later that we can maintain a mapping $P[u, v]$, that maps vertices u, v to the vertex $w \in V \setminus H$ that induces the longest lifetime 2-path between u and v using a vertex in $V \setminus H$, if such a vertex w exists. Therefore, add $(u, v, P[u, v])$ as a potential longest lifetime triangle if $P[u, v]$ is set. As mentioned above, this already suffices, as all other vertices $w \in V \setminus (H \cup \{P[u, v]\})$ would induce a shorter lifetime triangle.

We have therefore already shown how we can compute a list of length $O(h)$ of potential longest lifetime triangles, and therefore also the longest lifetime triangle after edge insertion, assuming one can maintain P. It therefore remains to show how P can be maintained efficiently. To allow maintaining P, we relax its definition: If it can be guaranteed that there exists a longer lifetime path between the vertices using a vertex in H, then we allow $P[u, v]$ to not be set even though there exists such a path, or even be set to the wrong vertex. In that case, the methodology described above still obviously correctly determines the longest lifetime triangle after the update. We need to describe edge insertion and deletion, as well as how moving vertices between H and $V \setminus H$ changes P.

To update P for edge insertion uv where $u \in V \setminus H$, investigate all neighbors $w \neq v$ of u and set $P[w, v] = u$ if the path over u has longer lifetime than the previous one, or if $P[w, v]$ was previously not set. Analogously, if $v \in V \setminus H$, potentially update $P[u, w]$ for all neighbors $w \neq u$ of v. We now show that this correctly updates P. We only consider the case where $u \in V \setminus H$, the case $v \in V \setminus H$ follows analogously. Obviously for all pairs of vertices this methodology does not consider for an update, the longest lifetime path between the vertices remains unchanged as u is not connected with at least one of the vertices. We therefore only need to consider $P[w, v]$ for all neighbors w of u. We need to consider both the case where $P[w, v]$ was previously set correctly, and also the case where it was set wrongly if we could guarantee that there was a longer lifetime path between w and v using a vertex in H. In case it was set correctly, this method obviously only updates the longest lifetime path between w and v if the path over u has longer lifetime. In this case, there is also no other path with even longer lifetime, as insertion of edge uv only creates a single new 2-path from w to v. We now consider the case where $P[w, v]$ was set wrongly as we could guarantee that a longer lifetime 2-path between w and v exists using a vertex in H. If the lifetime of the path over u is longer than all paths over vertices in H, our algorithm correctly updates $P[w, v]$ as the path over u then also has longer lifetime than the path over the previous $P[w, v]$. In case a path over a vertex in H still has a longer lifetime than over u, it does not matter that $P[w, v]$ is set to the wrong value. This therefore shows correctness of this methodology. As vertices in $V \setminus H$ have degree at most $O(h)$, these steps can be done in $O(h)$ time. In case $u \in H$ (analogously $v \in H$), we do not need to update $P[w, v]$ for any vertex w as u cannot be used as the intermediate low-degree vertex in that case.

To update P for an edge deletion uv if $u \in V \setminus H$ (and analogously v), again investigate every neighbor $w \neq v$ of u and invalidate $P[w, v]$ if $P[w, v] = u$. In case $P[w, v]$ was previously correctly set, no other path between w and v using a vertex in $V \setminus H$ can exist as $P[w, v]$ was the vertex inducing the longest lifetime path. In case it was incorrectly set, the longer lifetime path using a vertex in H still exists, it is therefore valid to keep it set incorrectly.

When moving a vertex u from H to $V \setminus H$, consider $P[w, v]$ for all pairs of neighbors w, v of u and potentially update it to u if that leads to a longer lifetime path. Correctness follows analogous to inserting an edge. When moving a vertex u from $V \setminus H$ to H, also consider $P[w, v]$ for all pairs of neighbors w, v of u and invalidate it if $P[w, v] = u$. Note that in the second case, we can guarantee that there is a longer lifetime path from w to v using a vertex in H, it is therefore valid for $P[w, v]$ to not be set even though there may exist another path using a vertex from $V \setminus H$. As in both cases u has $O(h)$ neighbors, this takes $O(h^2)$ time. But as vertices move between H and $V \setminus H$ only on average every $\Omega(h)$ operation, it amortizes to $O(h)$. □

Kara et al. [KNN+18] have used the online matrix-vector multiplication (OMv) conjecture, as given in the Conjecture 7.6, to compute a lower bound for maintaining the triangle count of the graph:

Conjecture 7.6 (OMv Conjecture [KNN+18])**.** *For any $\gamma > 0$, there is no algorithm that multiplies an $n \times n$ Boolean matrix online with n column vectors of size n in time $O(n^{3-\gamma})$.*

They have shown that maintaining the triangle count incrementally is not possible in amortized $O(m^{\frac{1}{2}-\gamma})$ update time and $O(m^{1-\gamma})$ query time for any $\gamma > 0$, unless the OMv conjecture fails. Their proof even showed that detecting whether the graph contains a single triangle is not possible under these conditions. As $h \in O(\sqrt{m})$, this shows in particular that the algorithm presented in Theorem 7.5 cannot be improved by a polynomial factor,

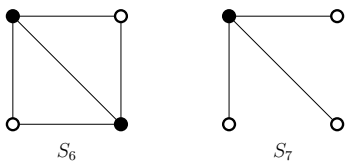

Figure 7.1: The non-induced patterns for which we maintain the longest lifetime subgraphs. This mapping is indexed by the white vertices, and needs to return the black vertices such that the substructure induced by the black and white vertices is the longest lifetime depicted substructure containing the white vertices and that have low-degree vertices for the black vertices, if such a substructure exists. [EGST12]

unless the OMv conjecture fails. We also note that this data structure also allows querying the longest lifetime triangle containing a specific edge uv in time $O(h)$. This is simple to do by considering all vertices in H, as well as the vertex $w = P[u, v]$, and returning the longest lifetime triangle created by any of these vertices.

An analogous proof to Theorem 7.5 is also possible for 4-cliques with amortized $O(h^2)$ update time. This requires similar modifications to the algorithm as also proposed by Eppstein et al. [EGST12] for counting 4-cliques.

Theorem 7.7. *There is an algorithm that can maintain the longest lifetime 4-clique of the graph under a priority queue dynamic update sequence in amortized $O(h^2)$ update time.*

Proof. We again maintain a partition H of vertices into high-degree and low-degree vertices. The proof proceeds as follows: We first describe how deletion and insertion of an edge changes the longest lifetime 4-clique. For edge insertion, this is done by enumerating all possible 4-cliques that could possibly be the new longest lifetime 4-clique after the update. Similar to how enumerating potential longest triangles required the mapping P in the proof of Theorem 7.5, enumerating such 4-cliques requires maintaining two mappings S_6 and S_7 as described below[1]. It is then shown how these mappings are updated on edge insertion, removal, and vertices moving between H and $V \setminus H$.

It is easy to see again the deleting an edge only changes the longest lifetime 4-clique if the edge is contained in the 4-clique. If the edge is indeed contained in the 4-clique, no other 4-cliques exist.

Regarding insertion of an edge uv, we define *potential longest lifetime 4-cliques* analogous to potential longest lifetime triangles in the proof of Theorem 7.5. We show that we can compute a list of $O(h^2)$ potential longest lifetime 4-cliques that contains the longest lifetime 4-clique after the update. Computing the longest lifetime 4-clique from this list is then analogous to Theorem 7.5, and takes $O(h^2)$ time. It is again easy to see that it suffices to add the previous longest lifetime 4-clique as the only potential longest lifetime 4-clique not containing the edge uv. It therefore remains to consider the case where uv is in the potential longest lifetime 4-clique. Make a case distinction again on whether the other vertices w, x of the potential longest lifetime 4-clique are in H or $V \setminus H$.

[1]We use the naming S_6 and S_7 as introduced by Eppstein et al. [EGST12]. Note that they also introduced subgraphs S_0 to S_5, but we do not need these.

For $w, x \in H$, one can just enumerate them like also done in Theorem 7.5, and add (u, v, w, x) as a potential longest lifetime 4-clique for all $w, x \in H$. This adds $O(h^2)$ potential longest lifetime 4-cliques.

For $w, x \in V \setminus H$, we maintain a mapping S_6 that maps two vertices u, v to vertices w, x such that the four vertices form the longest lifetime subgraph as depicted in Figure 7.1. We give details later on how this mapping can be maintained. One then adds (u, v, w, x) for $(w, x) = S_6[u, v]$ as a potential longest lifetime 4-clique if $S_6[u, v]$ is set. Similar to how it sufficed to consider $P[u, v]$ as the only low degree vertex for potential longest lifetime triangles in Theorem 7.5, it also suffices to consider only this single potential longest lifetime 4-clique where both other vertices are $V \setminus H$.

Consider the case now that one of the additional vertices is in H, and the other is in $V \setminus H$. Let without loss of generality $x \in H$ and $w \in V \setminus H$. For that case, we maintain another mapping S_7 that maps three vertices u, v, x to a vertex w such that the four vertices form the longest lifetime subgraph depicted in Figure 7.1. We again defer details on how this mapping can be maintained. Then, to find all potential longest lifetime 4-cliques containing uv with one additional high-degree and one additional low-degree vertex, we iterate over every $x \in H$ and add $(u, v, x, S_7[u, v, x])$ as a potential longest lifetime 4-clique if $S_7[u, v, x]$ is set. This therefore adds $O(h)$ potential longest lifetime 4-cliques. It is again easy to see that these $O(h)$ cases suffice as potential longest lifetime 4-cliques that contain uv and that have one additional high-degree and one additional low-degree vertex.

We have therefore already shown how we can compute the list of potential longest lifetime 4-cliques, and therefore also the longest lifetime 4-clique after edge insertion, assuming one can maintain S_6 and S_7 as defined above. It therefore suffices again to describe how S_6 and S_7 can be efficiently maintained. Like done in the proof of Theorem 7.5, we allow these mappings to not be set, or set to the wrong vertices, if we can guarantee that a longer lifetime 4-clique is possible using at least one vertex more from H. One again needs to consider edge insertion and deletion, as well as vertices moving between high and low degree.

We start with S_6 and inserting an edge uv. Consider the different cases based on whether u and v are in H or in $V \setminus H$. If both are in H, nothing needs to be done as S_6 does not contain such an edge. If without loss of generality $v \in H$, and $u \in V \setminus H$. Then uv could be one of the border edges of S_6. Iterate over all pairs of distinct neighbors w, x of u with $v \notin \{w, x\}$ and check whether the correct substructure is formed. If the correct substructure is formed for some pairs of neighbors, potentially update the longest lifetime substructure of S_6 for the correct pair of vertices. This step takes $O(h^2)$ as u has low degree. If both vertices are in $V \setminus H$, uv could also be the additional diagonal edge. Do the same as also done if only one vertex was in $V \setminus H$, but also check for the additional case that uv is the diagonal edge. For edge deletion, invalidate every longest lifetime substructure containing uv. Those substructures can be found analogous to edge insertion. The correctness of this method follows analogous to edge insertion and deletion when maintaining the longest lifetime triangle in Theorem 7.5.

Regarding keeping S_7 updated for edge insertion uv, do something similar: If both vertices are in H, nothing needs to be done as S_7 does not contain such an edge. If $u \in V \setminus H$, then u could be the low degree vertex in S_7. Iterate over all pairs of distinct neighbors w, x of u with $v \notin \{w, x\}$ and potentially update the longest lifetime substructure of S_7 for the triple (v, w, x). Do the same for $v \in V \setminus H$. Edge deletion for S_7 works analogously to edge deletion for S_6, where finding the correct substructures to invalidate is analogous to edge

insertion in S_7. This step obviously takes $O(h^2)$ again. The proof of correctness is again similar to Theorem 7.5.

When a vertex u is moved from H to $V \setminus H$, update the longest lifetime S_6 and S_7 like done for an edge insertion of uv for every neighbor v of u. This takes $O(h^3)$, but is amortized over $\Omega(h)$ steps again and therefore takes amortized $O(h^2)$ time. When a vertex u is moved from $V \setminus H$ to H, analogously update S_6 and S_7 like done for an edge deletion of uv for every neighbor v of u. This again takes amortized $O(h^2)$. The correctness of both cases is again similar to Theorem 7.5. □

By Hanauer et al. [HHH21], the runtime given in Theorem 7.7 cannot be improved by a polynomial factor unless the 4-clique conjecture as given in Conjecture 7.8 fails.

Conjecture 7.8 (4-Clique Conjecture [HHH21])**.** *For any $\gamma > 0$ and any n-vertex graph, there is no $O(n^{4-\gamma})$ time combinatorial algorithm for 4-clique detection with error probability at most $\frac{1}{3}$.*

As a final discussion for maintaining subgraphs, we now consider other patterns with 3 or 4 vertices. Here, one needs to differentiate between induced and non-induced subgraph counting [HHH21]. For induced subgraph counting, a pattern matches a subgraph if they exactly match, including missing edges. On the other hand, non-induced subgraph counting allows for missing edges in the pattern to be present in the subgraph.

We first note that non-induced subgraphs for three vertices are trivial to maintain: If the pattern is the independent set with three vertices, any three vertices may be returned. If the pattern is one edge and one independent vertex, return the longest lifetime edge and an arbitrary third vertex. If the pattern is the 2-edge path, maintain for each vertex the two adjacent edges with the longest lifetime, and return the vertex with the two adjacent edges that have the longest lifetime.

We now consider the induced case. While Eppstein and Spiro [ES09], Eppstein et al. [EGST12] and Hanauer et al. [HHH21] were able to extend their algorithms to allow for counting other substructures with 3 or 4 vertices for the induced case, their improvements do not extend to the longest lifetime substructures. The main problem in the induced case is that an insertion of an edge may invalidate the current longest lifetime subgraph. In that case, one needs to efficiently find the new longest lifetime subgraph. This seems hard to do efficiently.

8. Future Work

In this section, we discuss some areas of work that could be investigated in the future. This includes the simple question whether our model of restricted update sequences can also be applied to other dynamic graph problems to achieve a speedup compared to fully dynamic algorithms. We discuss such problems, highlight relevant related work in these areas and ideas on how they could be improved using restricted update sequences. Furthermore, we consider ideas how the algorithms presented in this thesis could be generalized to other update sequences. We also talk about general open problems for restricted update sequences.

8.1 Applying Update Sequences to Different Problems

We have already seen connectivity in Chapter 5 and 2-edge connectivity and biconnectivity in Chapter 6. The next logical step would be asking for k-edge and k-vertex connectivity between two vertices for some constant $k \geq 3$, or even finding the minimum edge or vertex cut between two vertices. A slight simplification of this question could be asking whether the entire graph is k-edge or k-vertex connected, instead of querying this property for two vertices. The related work in this area of research is summarized in Table 8.1. One potential tool for solving these problems are k-skeletons, as introduced by McGregor [McG14], and also used by Crouch et al. [CMS13]. The k-skeleton is a sparse subgraph of the graph, where two vertices are k-edge connected in the original graph if and only if they are k-edge connected in the k-skeleton. One can modify the proof by Crouch et al. [CMS13] for maintaining a k-skeleton in the sliding window model to also apply to the queue and priority queue dynamic update sequences similar to Section 5.2. One can then run static algorithms on the sparse skeletons like also done by Eppstein et al. [EGI93], leading to the same runtimes as presented in their paper. We however note that sparsification as introduced by Eppstein et al. [EGIN92, EGI93] is an alternative method of getting a sparse representation of a graph in the fully dynamic case without runtime overhead. Therefore, maintaining the k-skeleton of the graph in the priority queue dynamic update sequence does not directly lead to an improvement compared to the fully dynamic case.

A commonly investigated problem for directed underlying graphs is reachability. Similar to connectivity, it asks whether there is a path from one vertex to another, but for reachability this path has to adhere to the directions of the edges. Related work for this problem is summarized in Table 8.2. Simply applying a union-find data structure does not work as reachability is, unlike connectivity, not symmetric. A special union-find data structure,

Problem	Update Time	Query Time	Reference
3-edge connectivity	$O(n^{\frac{2}{3}})$	$O(n^{\frac{2}{3}})$	[EGIN92]
3-edge connectivity	insert: $O(\alpha(n))^a$, remove: $O(n\log(m/n))$	$O(\alpha(n))^a$	[EGIN92]
3-vertex connectivity	$O(n)$	$O(1)$	[EGI93]
3-vertex connectivity	insert: $O(\alpha(n))^a$, remove: $O(n\log(m/n))$	$O(\alpha(n))^a$	[EGIN92]
4-edge connectivity	$O(n\alpha(n))$	$O(1)$	[EGIN92]
4-vertex connectivity	$O(n\log(m/n)+n\alpha(n))$	$O(1)$	[EGIN92]
4-vertex connectivity	$O(n\alpha(n))$	$O(1)$	[EGI93]
4-vertex connectivity	insert: $O(\log n)$, remove: $O(n\log n)$	$O(1)$	[EGI93]
k-edge connectivity	$O(n\log n)$	$O(1)$	[EGIN92]
Graph $\sqrt{2+o(1)}$-approximate edge connectivity	$O(\text{poly}(\log n))^a$	$O(\text{poly}(\log n))^a$	[TK00]
Graph edge connectivity	$\tilde{O}(m^{1-\frac{1}{16}})^a$	$O(1)$	[GHN+23]
Graph edge connectivity	$\tilde{O}(n)$	$O(1)$	[GHN+23]
Polylogarithmic graph edge connectivity	$\tilde{O}(\sqrt{n})$	$O(1)$	[Tho01]
Graph minimum edge-cut	$\tilde{O}(\sqrt{m})$	$O(\log n)$ per returned edge	[Tho01]

Table 8.1: Summary of further dynamic connectivity algorithms. For the runtimes, a superscript a refers to amortization, and poly is some polynomial function. If the problem refers to a graph, then the queries ask about the problem for the entire graph and not pairs of vertices.

Update Sequence	Update Time	Query Time	Reference
incremental	$O(n)^a$	$O(1)$	[LPvL88]
incremental; acyclic	$O(1)$ – $O(\log n)$	$O(n^2\log n)$ – $O(n^3)$	[BDHTG25]
decremental	$O(n(\log n)^2)^{a\,r(m)}$	$O(n/\log n)$	[HK95]
decremental; acyclic	$O(n)^a$	$O(1)$	[LPvL88]
fully dynamic	$O(n^2)^a$	$O(1)$	[Rod03]
fully dynamic	$O(m+n\log n)^a$	$O(n)$	[RZ04]
fully dynamic	$O(m\sqrt{n})^a$	$O(\sqrt{n})$	[RZ08]
fully dynamic	$O(m^{0.58}n)^a$	$O(m^{0.43})$	[RZ08]
fully dynamic	$O(\hat{m}\sqrt{n}(\log n)^2+n)^{a\,r(m)}$	$O(n/\log n)$	[HK95]
fully dynamic	$O(n\hat{m}^{0.58}(\log n)^2)^{a\,r(m)}$	$O(n/\log n)$	[HK95]

Table 8.2: Summary of dynamic reachability algorithms. For the runtimes, a superscript a refers to amortization, and $r(m)$ refers to Monte Carlo randomization. $\hat{m}$ refers to the average number of edges in the graph during the entire update sequence. Note that the runtime by Bulteau et al. [BDHTG25] can be varied, choosing some desired update time leads to a fixed query time and vice versa; furthermore, the query time depends on many different properties of the graph, we use worst-case values depending on n for these properties. Furthermore, unlike the other algorithms in the table, their algorithm inserts sinks instead of edges.

Problem	Update Time	Query Time	Reference
Incremental SSSP up to distance k	$O(k)^a$	$O(1)$	[RZ11]
SSSP	$O(j \log n)^a$	$O(1)$	[FMSN96]
$(1+\varepsilon)$-approximate SSSP	$\tilde{O}(n^{1.823}/\varepsilon^2)^{r(m)}$	$O(1)$	[vdBN19]
APSP	$\tilde{O}(m\sqrt{n})^{a\,r(m)}$	$O(n^{\frac{3}{4}})$	[RZ11]
APSP	$\tilde{O}(n^2)^a$	$O(1)$	[DI03]
$(1+\varepsilon)$-approximate, undirected, unweighted APSP	$\tilde{O}(n^2/\varepsilon^{1+\omega})^{r(m)}$	$O(1)$	[vdBN19]
$(1+\varepsilon)$-approximate, directed, positively weighted APSP	$\tilde{O}(n^{2.045}/\varepsilon^2)^{r(m)}$	$O(1)$	[vdBN19]

Table 8.3: Summary of dynamic shortest path problems algorithms. For the runtimes, a superscript a refers to amortization, and $r(m)$ refers to Monte Carlo randomization. $\omega \approx 2.373$ is the matrix multiplication exponent. The algorithms are for directed, unweighted graphs unless noted otherwise. For the result by Frigion et al. [FMSN96], j is chosen such that there exists a j-bounded accounting function, a function which maps each edge to one of its endpoints, called its owner, such that every edge owns at most j edges. There holds $j \in O(\sqrt{m})$ in general, and j is constant for planar or near-planar graphs, and can also be bounded based on the treewidth or genus of the graph.

which merges two sets, but where the old sets would still be valid, could work. In that case, when adding an edge from u to v, one could merge the reachable set from v into the reachable set from u, while still keeping the previous reachable set for v. Note that in such a union-find data structure, a vertex could be in multiple distinct sets, for example a vertex could be reachable from both u and v, but u is not reachable from v and vice versa. If one shows that such a union-find data structure exists, this automatically leads to a stack dynamic algorithm as an undo can be implemented by restoring the set for u. Lemma 4.17 can then be used for a priority queue dynamic algorithm.

Another common problem is the shortest path problem. For this problem, one considers different cases: In the *all-pairs shortest paths* (APSP) problem one can query the shortest path distance between any pair of vertices, while for the *single-source shortest paths* (SSSP) problem one vertex is fixed before running the algorithm and the other vertex can be freely chosen in the query. Related work for the shortest path problems is summarized in Table 8.3. We first note that Roditty and Zwick [RZ11] showed that a general polynomial improvement for the incremental weighted single-source shortest paths over the trivial algorithm recomputing everything from scratch after each update would lead to improvements to the static all-pairs shortest paths problem to $o(mn)$, which seems unlikely. Similar but weaker results are also given for the unweighted variant. As these reductions hold for the incremental update sequence, they therefore also hold for all update sequences mentioned in this thesis. Note however that for special kinds of graphs, such as planar or near-planar graphs, as well as for graphs with bounded treewidth or genus, improvements are possible, as shown by Frigion et al. [FMSN96]. One can also investigate the incremental single-source shortest paths problem only for paths up to a certain distance, like done by Roditty and Zwick [RZ11]. While their algorithm is amortized and therefore cannot be trivially made stack dynamic, it would be interesting to check whether an alternative approach could allow solving the single-source shortest paths up to a specific distance for more general update sequences. Regarding the all-pairs shortest paths problem, note that the update time presented by Demetrescu and Italiano [DI03] is only a polylogarithmic factor away from the worst-case lower bound $\Omega(n^2)$ for constant query time, which even holds in the

Problem	Update Time	Query Time	Reference
$O(1)$-approximate vertex cover and fractional matching	$O(1)^a$	$O(1)$	[BCH17]
$(2+\varepsilon)$-approximate vertex cover and maximum matching	$O((\log n)^3)$	$O(1)$	[BHN17]
$(3/2+\varepsilon)$-approximate vertex cover	$O(m^{\frac{1}{4}}/\varepsilon^2)$	$O(1)$	[BS16]
$(1+\varepsilon)$-approximate matching and vertex cover	$O(\sqrt{m}/\varepsilon^2)$	$O(1)$	[GP13]
2-approximate maximum matching and vertex cover	$O(1)^{a\,r(l)\,r(m)}$	$O(1)$	[Sol16]
Incremental maximal independent set	$O(\min\{\Delta, \sqrt{m}\})^a$	$O(1)$	[GK18]
Maximal independent set	$O(\min\{\Delta, m^{3/4}\})^a$	$O(1)$	[GK18]
Maximal independent set	$O(\min\{\Delta, \sqrt{m}\})^a$	$O(\sqrt{m})$	[GK18]

Table 8.4: Summary of dynamic vertex cover, matching and independent set algorithms. For the runtimes, a superscript a refers to amortization, $r(l)$ refers to Las Vegas randomization, and $r(m)$ refers to Monte Carlo randomization. Δ refers to the maximum node degree in the graph.

Problem	Update Time	Query Time	Reference
Parameterized vertex cover	$O(\text{poly}(k)\log n)$	$f(\text{poly}(k), k)$	[IO14]
Parameterized cluster vertex deletion	$O(\text{poly}(k)\log n)$	$f(\text{poly}(k), k)$	[IO14]
Parameterized chromatic number in the solution size of cluster vertex deletion *	$O(\log n)$	$O(1)$	[IO14]
Parameterized bounded-degree feedback vertex set *	$O(\log n)$	$O(1)$	[IO14]

Table 8.5: Summary of parameterized algorithms. For the runtimes, k is the parameter, $f(n, k)$ is the time complexity of the best static parameterized problem, and poly is some polynomial function. For the problems marked with *, the parameter is assumed to be constant.

incremental case. An improvement in this area with a constant query time therefore seems unlikely. Improvements are potentially possible if one allows for a non-constant query time.

We have already talked about the maximal matching and independent set problems in Section 4.3. We have seen that trivially converting an incremental to a priority queue dynamic algorithm leads to runtimes worse than already existing fully dynamic algorithms. Can other methods be used to get better queue or priority queue dynamic results? What about their (approximate) maximum variants, or vertex cover? We have summarized related work in this area in Table 8.4.

One can also consider parameterized problems, as done by Iwata and Oka [IO14]. Their results are summarized in Table 8.5. As the exponential factors depending on the parameter are the same as in the best static algorithms, it seems hard to significantly improve on the results, even for restricted update sequences.

Problem	Update Time	Query Time	Reference
Incremental planarity	$O(\alpha(n))^a$	$O(1)$	[LP94]
Planarity	$O((\log n)^3)^a$	$O(1)$	[HR20]
Cycle equivalence	$O(\sqrt{n}\log n)$	$O((\log n)^2)$	[Hen94]
Betweenness centrality	$O(\nu^{*2}(\log n)^2)^a$	$O(1)$	[PR15]
Incremental dominator tree	$O(n)^a$	$O(1)$	[AL70]
Decremental dominator tree	$O(n)^a$	$O(1)$	[GGI+19]
Decremental low-high orders	$O(n)^a$	$O(1)$	[GGI+19]

Table 8.6: Summary of other miscellaneous dynamic graph algorithms. For the runtimes, a superscript a refers to amortization. ν^* bounds the number of distinct edges that lie on shortest paths through any single vertex. Algorithms presented by Georgiadis et al. [GGI+19] only work for reduced graphs.

Last but not least, we summarize some miscellaneous dynamic graph algorithms in Table 8.6. Refer to a survey from Hanauer et al. [HHS22] for more inspiration about problems that could be improved for restricted update sequences in the future.

8.2 Generalize Update Sequences for Investigated Problems

In this section, we discuss some ideas whether our algorithms presented in this thesis could possibly be modified to allow for more general update sequences.

In Chapter 5 we investigated connectivity. It was shown that in particular deque dynamic connectivity, and therefore also 2-stack dynamic connectivity could be solved in logarithmic time per operation. How about j-stack dynamic connectivity, for $j \geq 3$? The algorithm we described maintained two spanning forests, which used edges as close to the top of each stack as possible, and only used bottommost edges of the other stack if strictly required. This could be done similarly for the j-stack dynamic update sequence, by having j spanning forests using the topmost edges of its own stack each and using bottommost edges of the other stacks only if required. The problem here is specifying from which stack the bottommost edges is chosen. Furthermore, one somehow needs to handle the case in which a bottommost edge is removed from one stack and needs to be transitioned to another stack.

For something even harder, one could look into j-priority-queue dynamic connectivity for $j \geq 2$. For $j = 2$, this could possibly again be handled by maintaining the shortest lifetime forests for each of the priority queues and only using edges from the other priority queue if required. This first needs some way to maintain the shortest lifetime forests, which as discussed in Section 5.5 is not as easy as with the stack and queue dynamic update sequences. For $j \geq 3$, the same problems arise as also discussed with j-stack dynamic connectivity.

One further interesting problem is whether the results for 2-edge connectivity and biconnectivity from Chapter 6 could be extended to work for the deque dynamic update sequence. As already discussed in Section 6.2, our algorithm relies on the fact that a non-forest edge never becomes a forest edge during the runtime of the algorithm. This does not hold for the deque dynamic update sequence. It is therefore unclear how our work can be extended to allow for deque dynamic 2-edge connectivity, or whether new insights are required.

The last open research question based on our results can be found in Chapter 7. Here, we have shown how the longest lifetime triangle and 4-clique can be maintained using algorithms inspired by dynamic subgraph counting. We have also seen that also the longest lifetime non-induced 3-vertex patterns can be efficiently maintained, which directly follow

by simple algorithms. For the non-induced 4-vertex patterns this is harder though, as these patterns are not trivial like the 3-vertex ones. Such non-trivial patterns include for example the paw graph — the triangle with one additional edge to a fourth vertex. Searching for the longest lifetime paw graph is at least as hard as checking whether the graph contains at least one triangle: One can reduce the problem of searching for a triangle to the problem searching for a paw by adding a new vertex v' for each vertex v and adding the edges vv'. It may be possible to extend the proof by Eppstein et al. [EGST12] for maintaining the longest lifetime 4-vertex patterns. For induced 3- and 4-vertex patterns, maintaining the longest lifetime substructures seems a lot harder, as an insertion of an edge can now invalidate the longest lifetime substructure. This means that we need an efficient way to find the replacement substructure. While one can maintain a sorted list of all matched substructures, this intuitively seems inefficient as one edge insertion can create and invalidate many substructures at once. It would also be interesting to see what further lifetime problems can be formulated and solved.

8.3 Miscellaneous Open Problems

We now discuss miscellaneous open problems related to restricted update sequences.

In Chapter 4, we have given examples of pairs of update sequences that cannot emulate each other. It would be nice to have some example problems showing this difference. This would require two problems: One that can be solved for example in $O(t_u(n,m))$ update time and $O(t_q(n,m))$ query time for the stack dynamic update sequence, but requires $\omega(t_u(n,m))$ update time for $O(t_q(n,m))$ query time in the queue dynamic update sequence, and a second problem showing the inverse direction. While the related work on dynamic lower bounds is currently sparse compared to upper bounds, the additional information from having a restricted update sequences could possibly simplify some lower bounds. Lower bounds, for example for stack and priority queue dynamic update sequences, would also be interesting on their own. Note however that such lower bounds would be weaker compared to fully dynamic lower bounds.

As mentioned in Section 4.2, our emulation is required to emit at most a worst-case constant number of operations per emulated operations. We already discussed a few places where an amortized emulation or even a logarithmic emulation would make sense. This includes the 2-stack dynamic update sequence being able to emulate the queue dynamic update sequence in amortized constant number of operations, but not in a worst-case constant number of operations. We have also seen how the algorithm converting a stack dynamic to a priority queue dynamic update sequence can be seen as an emulation with an amortized logarithmic number of operations per emulated operations. Are there any other results we can achieve with such kinds of emulations? Can some of our incomparability results be extended to also show incomparability under amortized constant emulations?

Like already mentioned in Chapter 3, it would be interesting to consider batch dynamic algorithms with restricted update sequences. As an example, one could insert a set of edges at once, and all edges inserted at the same time are also removed at the same time in a stack, queue or priority queue dynamic update sequence. Other restrictions are of course also possible, like inserting edges in a batch, but always removing a single edge in the batch according to some of the above-mentioned update sequences. As a starting point, one could investigate the connectivity problem in that model, like done by Acar et al. [AABD19], but other problems are potentially also worth looking at.

We also note that it could be interesting to not only restrict updates, as given in this thesis, but also restrict queries. As a trivial example, one could investigate fully dynamic connectivity where one can query connectivity between vertices u and v only directly after

the edge uv is inserted into the graph. Obviously, the result of such queries is always "yes", and the problem can therefore be answered in $O(1)$ update and query time. Are there any non-trivial problems worth looking at?

It would also be interesting to evaluate the algorithm developed in this thesis in practice. As our algorithms are relatively simple, the difficulty would mainly lie in an implementation of top trees. For this, one could for example apply an implementation of splay top trees by Holm et al. [HRR23].

9. Conclusion

In this thesis, we considered dynamic graphs with restricted update sequences. Such restricted update sequence were defined based on commonly used data structures, like the stack, queue or priority queue. We have seen how different update sequences can emulate each other, which allowed us to define a notion of complexity for different update sequences. It was furthermore discussed how such restrictions can be used to create dynamic graph algorithms with better runtime compared to fully dynamic algorithms found in the literature. We saw that dynamic connectivity can be solved for the priority queue and deque dynamic update sequences in logarithmic time per operation, while the current best fully dynamic algorithm requires an amortized expected $O(\log n(\log\log n)^2)$ update and $O(\log n/\log\log\log n)$ query time [HHKP17]. For the priority queue dynamic update sequence, this runtime was shown to be optimal. Stack dynamic connectivity had an even better runtime of amortized $O(\log n/\log\log n)$ per operation, using a union-find data structure with backtracking [WT89]. From the connectivity results, further algorithms like checking whether the underlying graph is bipartite or maintaining an approximate minimum spanning forest directly follow. We have furthermore shown that 2-edge connectivity can be solved in logarithmic time per operation for the priority queue dynamic update sequence, compared to amortized $O((\log n)^4)$ per operation for the current best fully dynamic algorithm [HdLT98]. For biconnectivity, we achieved amortized $O((\log n)^2/\log\log n)$ per operation for the stack dynamic update sequence, which implied a priority queue dynamic algorithm with an additional logarithmic factor for updates. Those results were based on an incremental algorithm with an amortized runtime of $O(\alpha(n)\log n)$ per operation. This improves the runtime for the current best fully dynamic result with expected amortized $O((\log n)^4)$ update and $O((\log n)^2)$ query time [HK95]. For priority queue dynamic update sequences and its specializations, we have also defined lifetime problems. Using our connectivity results, we have for example shown how to efficiently compute the longest lifetime path between any two vertices. Using related work in the area of dynamic subgraph counting, we have also shown how the longest lifetime triangle and 4-clique can be maintained. The runtime of those algorithms cannot be improved by a polynomial factor unless some popular conjectures fail.

We have already highlighted future work in this area of research, and a few ideas how they could be resolved, in Chapter 8. One major open area is applying our model to further dynamic problems, potentially improving their runtime compared to fully dynamic algorithms. Orthogonal to this research effort, one can also try to find algorithms for problems this thesis investigated for more general update sequences.

Bibliography

[AABD19] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, and Laxman Dhulipala. Parallel batch-dynamic graph connectivity. In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'19)*, page 381–392. ACM, 2019.

[AEGH98] Pankaj K. Agarwal, David Eppstein, Leonidas J. Guibas, and Monika R. Henzinger. Parametric and kinetic minimum spanning trees. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science (FOCS'98)*, pages 596–605. IEEE, 1998.

[AGM12] Kook J. Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the 23rd Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'12)*, pages 459–467. SIAM, 2012.

[AHLT05] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms*, 1(2):243–264, 2005.

[AL70] Stephen Alstrup and Peter Lauridsen. A simple dynamic algorithm for maintaining a dominator tree. Technical Report 93–03, Department of Computer Science, University of Copenhagen, 1970.

[BCD+02] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA'02)*, volume 2461 of *LNCS*, pages 152–164. Springer Berlin Heidelberg, 2002.

[BCH17] Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika R. Henzinger. Deterministic fully dynamic approximate vertex cover and fractional matching in $O(1)$ amortized update time. In *Proceedings of the 19th Integer Programming and Combinatorial Optimization Conference (IPCO'17)*, volume 10328 of *LNCS*, pages 86–98. Springer International Publishing, 2017.

[BDHTG25] Laurent Bulteau, Pierre-Yves David, Florian Horn, and Euxane Tran-Girard. Incremental reachability. hal-04929088, 2025.

[BHN17] Sayan Bhattacharya, Monika R. Henzinger, and Danupon Nanongkai. Fully dynamic approximate maximum matching and minimum vertex cover in $O(\log^3 n)$ worst case update time. *CoRR*, abs/1704.02844, 2017.

[BS16] Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *Proceedings of the 27th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'16)*, pages 692–711. SIAM, 2016.

[CFQS10] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *CoRR*, abs/1012.0009, 2010.

[CMS13] Michael S. Crouch, Andrew McGregor, and Daniel Stubbs. Dynamic graphs in the sliding-window model. In *Proceedings of the 21st Annual European Symposium on Algorithms (ESA'13)*, volume 8125 of *LNCS*, pages 337–348. Springer Berlin Heidelberg, 2013.

[DI03] Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. In *Proceedings of the 35th Annual ACM Symposium on the Theory of Computing (STOC'03)*, page 159–166. ACM, 2003.

[DS87] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on the Theory of Computing (STOC'87)*, page 365–372. ACM, 1987.

[EGI93] David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Improved sparsification. Technical Report 93–20, Department of Information and Computer Science University of California, Irvine, 1993.

[EGI99] David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Dynamic graph algorithms. *Algorithms and Theory of Computation Handbook*, 1, 1999.

[EGIN92] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science (FOCS'92)*, pages 60–69. IEEE, 1992.

[EGST12] David Eppstein, Michael T. Goodrich, Darren Strash, and Lowell Trott. Extended dynamic subgraph statistics using h-index parameterized data structures. *Theoretical Computer Science*, 447:44–52, 2012.

[EIT+92] David Eppstein, Giuseppe F. Italiano, Roberto Tamassia, Robert E. Tarjan, Jeffery Westbrook, and Moti Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. *Journal of Algorithms*, 13(1):33–54, 1992.

[Epp94] David Eppstein. Offline algorithms for dynamic minimum spanning tree problems. *Journal of Algorithms*, 17(2):237–250, 1994.

[ES09] David Eppstein and Emma S. Spiro. The h-index of a graph and its application to dynamic subgraph statistics. In *Proceedings of the 11th International Workshop on Algorithms and Data Structures (WADS'09)*, volume 5664 of *LNCS*, pages 278–289. Springer Berlin Heidelberg, 2009.

[ES22] Thomas Erlebach and Jakob T. Spooner. Exploration of k-edge-deficient temporal graphs. *Acta Informatica*, 59(4):387–407, 2022.

[FMSN96] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully dynamic output bounded single source shortest path problem. In *Proceedings of the 7th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'96)*, page 212–221. SIAM, 1996.

[GGI+19] Loukas Georgiadis, Konstantinos Giannis, Giuseppe F. Italiano, Aikaterini Karanasiou, and Luigi Laura. Dynamic dominators and low-high orders in DAGs. In *Proceedings of the 27th Annual European Symposium on Algorithms (ESA'19)*, volume 144, pages 50:1–50:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.

[GHN+23] Gramoz Goranci, Monika R. Henzinger, Danupon Nanongkai, Thatchaphol Saranurak, Mikkel Thorup, and Christian Wulff-Nilsen. Fully dynamic exact edge connectivity in sublinear time. In *Proceedings of the 34th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'23)*, pages 70–86. SIAM, 2023.

[GK18] Manoj Gupta and Shahbaz Khan. Simple dynamic algorithms for maximal independent set and other problems. *CoRR*, abs/1804.01823, 2018.

[GP13] Manoj Gupta and Richard Peng. Fully dynamic $(1+\epsilon)$-approximate matchings. *CoRR*, abs/1304.0378, 2013.

[HdLT98] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proceedings of the 13th Annual ACM Symposium on the Theory of Computing (STOC'98)*, page 79–89. ACM, 1998.

[Hen94] Monika R. Henzinger. Fully dynamic cycle-equivalence in graphs. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science (FOCS'94)*, pages 744–755. IEEE, 1994.

[HF98] Monika R. Henzinger and Michael L. Fredman. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica*, 22(3):351–362, 1998.

[HFL15] Silu Huang, Ada Wai-Chee Fu, and Ruifeng Liu. Minimum spanning trees in temporal graphs. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of data (SIGMOD'15)*, page 419–430. ACM, 2015.

[HHH21] Kathrin Hanauer, Monika R. Henzinger, and Qi Cheng Hua. Fully dynamic four-vertex subgraph counting. *CoRR*, abs/2106.15524, 2021.

[HHKP17] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully dynamic connectivity in $O(\log n(\log\log n)^2)$ amortized expected time. In *Proceedings of the 28 Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'17)*, pages 510–520. SIAM, 2017.

[HHS22] Kathrin Hanauer, Monika R. Henzinger, and Christian Schulz. Recent advances in fully dynamic graph algorithms – a quick reference guide. *ACM Journal of Experimental Algorithmics*, 27(1), 2022.

[HK95] Monika R. Henzinger and Valerie King. Fully dynamic biconnectivity and transitive closure. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science (FOCS'95)*, pages 664–672. IEEE, 1995.

[HK99] Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.

[HLP95] Monika R. Henzinger and Han La Poutré. Certificates and fast algorithms for biconnectivity in fully-dynamic graphs. In *Proceedings of the 3rd Annual European Symposium on Algorithms (ESA'95)*, volume 979 of *LNCS*, pages 171–184. Springer Berlin Heidelberg, 1995.

[HN16] Monika R. Henzinger and Stefan Neumann. Incremental and fully dynamic subgraph connectivity for emergency planning. In *Proceedings of the 24th Annual European Symposium on Algorithms (ESA'16)*, volume 57, pages 48:1–48:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016.

[HR20] Jacob Holm and Eva Rotenberg. Fully-dynamic planarity testing in polylogarithmic time. In *Proceedings of the 52nd Annual ACM Symposium on the Theory of Computing (STOC'20)*, page 167–180. ACM, 2020.

[HRR23] Jacob Holm, Eva Rotenberg, and Alice Ryhl. Splay top trees. In *2023 Symposium on Simplicity in Algorithms (SOSA)*, pages 305–331. SIAM, 2023.

[HRS92] John Hershberger, Monika Rauch, and Subhash Suri. Fully dynamic 2-edge-connectivity in planar graphs. In *Proceedings of the 3rd Scandinavian Workshop on Algorithm Theory (SWAT'92)*, volume 621 of *LNCS*, pages 233–244. Springer Berlin Heidelberg, 1992.

[HRW14] Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. Faster fully-dynamic minimum spanning forest. *CoRR*, abs/1407.6832, 2014.

[Hsu96] Wen-Lian Hsu. On-line recognition of interval graphs in $O(m + n \log n)$ time. In *Combinatorics and Computer Science*, volume 1120 of *LNCS*, pages 27–38. Springer Berlin Heidelberg, 1996.

[IO14] Yoichi Iwata and Keigo Oka. Fast dynamic graph algorithms for parameterized problems. *CoRR*, abs/1404.7307, 2014.

[KM89] Norbert Korte and Rolf H. Möhring. An incremental linear-time algorithm for recognizing interval graphs. *SIAM Journal on Computing*, 18(1):68–81, 1989.

[KNN+18] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Counting triangles under updates in worst-case optimal time. *CoRR*, abs/1804.02780, 2018.

[LP94] Johannes A. La Poutré. Alpha-algorithms for incremental planarity testing (preliminary version). In *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing (STOC'94)*, page 706–715. ACM, 1994.

[LPvL88] Johannes A. La Poutré and Jan van Leeuwen. Maintenance of transitive closures and transitive reductions of graphs. In *Proceedings of the 14th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'88)*, volume 314 of *LNCS*, pages 106–120. Springer Berlin Heidelberg, 1988.

[LS13] Jakub Lacki and Piotr Sankowski. Reachability in graph timelines. In *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science (ITCS'13)*, page 257–268. ACM, 2013.

[McG14] Andrew McGregor. Graph stream algorithms: A survey. *SIGMOD Record*, 43(1):9–20, 2014.

[Mic15] Othon Michail. An introduction to temporal graphs: An algorithmic perspective. In *Algorithms, Probability, Networks, and Games*, volume 9295 of *LNCS*, pages 308–343. Springer International Publishing, 2015.

[NTM+12] Vincenzo Nicosia, John Tang, Mirco Musolesi, Giovanni Russo, Cecilia Mascolo, and Vito Latora. Components in time-varying graphs. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 22(2), 2012.

[PD05] Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *CoRR*, abs/cs/0502041, 2005.

[PR15] Matteo Pontecorvi and Vijaya Ramachandran. Fully dynamic betweenness centrality. In *Proceedings of the 26th International Symposium on Algorithms and Computation (ISAAC'15)*, volume 9472 of *LNCS*, pages 331–342. Springer Berlin Heidelberg, 2015.

[PR23] Binghui Peng and Aviad Rubinstein. Fully-dynamic-to-incremental reductions with known deletion order (e.g. sliding window). In *Proceedings of the 34th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'23)*, pages 261–271. SIAM, 2023.

[Rau92] Monika Rauch. Fully dynamic biconnectivity in graphs. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science (FOCS'92)*, pages 50–59. IEEE, 1992.

[Rod03] Liam Roditty. A faster and simpler fully dynamic transitive closure. In *Proceedings of the 14th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'03)*, page 404–412. SIAM, 2003.

[RZ04] Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proceedings of the 36th Annual ACM Symposium on the Theory of Computing (STOC'04)*, page 184–191. ACM, 2004.

[RZ08] Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM Journal on Computing*, 37(5):1455–1471, 2008.

[RZ11] Liam Roditty and Uri Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011.

[SM10] Piotr Sankowski and Marcin Mucha. Fast dynamic transitive closure with lookahead. *Algorithmica*, 56(2):180–197, 2010.

[SMS+20] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB Journal*, 29(2):595–618, 2020.

[Sol16] Shay Solomon. Fully dynamic maximal matching in constant update time. In *Proceedings of the 57th Annual IEEE Symposium on Foundations of Computer Science (FOCS'16)*, pages 325–334. IEEE, 2016.

[ST81] Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. In *Proceedings of the 13th Annual ACM Symposium on the Theory of Computing (STOC'81)*, page 114–122. ACM, 1981.

[Tar83] Robert E. Tarjan. Minimum spanning trees. In *Data Structures and Network Algorithms*, pages 71–83. SIAM, 1983.

[Tho00] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the 32nd Annual ACM Symposium on the Theory of Computing (STOC'00)*, page 343–350. ACM, 2000.

[Tho01] Mikkel Thorup. Fully-dynamic min-cut. In *Proceedings of the 33rd Annual ACM Symposium on the Theory of Computing (STOC'01)*, page 224–230. ACM, 2001.

[TK00] Mikkel Thorup and David R. Karger. Dynamic graph algorithms with applications. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory (SWAT'00)*, volume 1851 of *LNCS*, pages 1–9. Springer Berlin Heidelberg, 2000.

[TSM+10] John Tang, Salvatore Scellato, Mirco Musolesi, Cecilia Mascolo, and Vito Latora. Small-world behavior in time-varying graphs. *Physical Review E*, 81(5), 2010.

[vdBFNP24] Jan van den Brand, Sebastian Forster, Yasamin Nazari, and Adam Polak. On dynamic graph algorithms with predictions. In *Proceedings of the 35th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'24)*, pages 3534–3557. SIAM, 2024.

[vdBN19] Jan van den Brand and Danupon Nanongkai. Dynamic approximate shortest paths and beyond: Subquadratic and worst-case update time. In *Proceedings of the 60th Annual IEEE Symposium on Foundations of Computer Science (FOCS'19)*, pages 436–455. IEEE, 2019.

[WCH+14] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path problems in temporal graphs. *Proceedings of the VLDB Endowment*, 7(9):721–732, 2014.

[WT89] Jeffery Westbrook and Robert E. Tarjan. Amortized analysis of algorithms for set union with backtracking. *SIAM Journal on Computing*, 18(1):1–11, 1989.

[WT92] Jeffery Westbrook and Robert E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7(1):433–464, 1992.