Self-Adjusting Linear Networks with Ladder Demand Graph

Vitaly Aksenov[1][0000−0001−9134−5490], Anton Paramonov[2], Iosif Salem[3][0000−0003−2810−2781], and Stefan Schmid*[3][0000−0002−7798−1711]

[1] ITMO University, St. Petersburg, Russia aksenov.vitaly@gmail.ru
[2] EPFL, Switzerland anton.paramonov2000@gmail.com
[3] TU Berlin, Berlin, Germany
iosif.salem@inet.tu-berlin.de, stefan.schmid@tu-berlin.de

Abstract. Self-adjusting networks (SANs) have the ability to adapt to communication demand by dynamically adjusting the workload (or demand) embedding, i.e., the mapping of communication requests into the network topology. SANs can reduce routing costs for frequently communicating node pairs by paying a cost for adjusting the embedding. This is particularly beneficial when the demand has structure, which the network can adapt to. Demand can be represented in the form of a demand graph, which is defined by the set of network nodes (vertices) and the set of pairwise communication requests (edges). Thus, adapting to the demand can be interpreted by embedding the demand graph to the network topology. This can be challenging both when the demand graph is known in advance (offline) and when it revealed edge-by-edge (online). The difficulty also depends on whether we aim at constructing a static topology or a dynamic (self-adjusting) one that improves the embedding as more parts of the demand graph are revealed. Yet very little is known about these self-adjusting embeddings.
In this paper, the network topology is restricted to a line and the demand graph to a ladder graph, i.e., a $2 \times n$ grid, including all possible subgraphs of the ladder. We present an online self-adjusting network that matches the known lower bound asymptotically and is 12-competitive in terms of request cost. As a warm up result, we present an asymptotically optimal algorithm for the cycle demand graph. We also present an oracle-based algorithm for an arbitrary demand graph that has a constant overhead.

Keywords: Ladder graph · Self-adjusting networks · Traffic patterns · Online algorithms.

1 Introduction

Traditional networks are static and demand-oblivious, i.e., designed without considering the communication demand. While this might be beneficial for all-to-

* Supported partially by the Austrian Science Fund (FWF) project I 4800-N (ADVISE) and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, grant agreement No. 864228 "Self-Adjusting Networks (Adjust-Net)".

all traffic, it doesn't take into account temporal or spatial locality features in demand. That is, sets of nodes that temporarily cover the majority of communication requests may be placed diameter-away from each other in the network topology. This is a relevant concern as studies on datacenter network traces have shown that communication demand is indeed bursty and skewed [3].

Self-adjusting networks (SANs) are optimized towards the traffic they serve. SANs can be static or dynamic, depending on whether it is possible to reconfigure the embedding (mapping of communication requests to the network topology) in between requests, and offline or online, depending on whether the sequence of communication requests is known in advance or revealed piece-wise. In the online case, we assume that the embedding can be adjusted in between requests at a cost linear to the added and deleted edges, thus, bringing closer frequently communicating nodes. Online algorithms for SANs aim to reduce the sum of routing and reconfiguration (re-embedding) costs for any communication sequence.

We can express traffic in the form of a demand graph that is defined by the set of nodes in the network and the set of pairwise communication requests (edge set) among them. Knowing the structure of the demand graph could allow us to further optimize online SANs, even though the demand is still revealed online. That is, by re-embedding the demand graph to the network we optimize the use of network resources according to recent patterns in demand.

To the best of our knowledge, the only work on demand graph re-embeddings to date is [2], where the network topology is a line and the demand graph is also a line. The authors presented an algorithm that serves $m = \Omega(n^2)$ requests at cost $O(n^2 \log n + m)$ and showed that this complexity is the lower bound. The problem is inspired by the Itinerant List Update Problem [12] (ILU). To be more precise, the problem in [2] appears to be the restricted version of the online Dynamic Minimum Linear Arrangement problem, which is another reformulation of ILU.

Contributions. In this work, we take the next step towards optimizing online SANs for more general demand graphs. We restrict the network topology to a line, but assume that the demand graph is a ladder, i.e., a $2 \times n$ grid. We assume that before performing a request, we can re-adjust the line topology by performing several swaps of two neighbouring nodes, paying one for each swap. We present a 12-competitive online algorithm that embeds a ladder demand graph to the line topology, thus, asymptotically matching the lower bound in [2]. This algorithm can be applied to any demand graph that is a subgraph of the ladder graph and that when all edges of the demand graph are revealed the topology is optimal and no more adjustments are needed. We also optimally solve the case of cycle demand graphs, which is a simple generalization of the line demand graph, but is not a subcase of the ladder due to odd cycles. Finally, we provide a generic algorithm for arbitrary demand graphs, given an oracle that computes an embedding with the cost of requests bounded by the bandwidth.

A solution for the ladder is the first step towards the $k \times n$ grid demand graph where k is an arbitrary constant. Moreover, a ladder (and a cycle) has a constant *bandwidth*, i.e., a minimum value over all embeddings in a target line topology of a maximal path between the ends of an edge (request). It can be

shown that given a demand graph G the best possible complexity per request is the bandwidth.

Related work. Avin et al. [2], consider a fixed line (host) network and a line demand graph. Their online algorithm re-embeds the demand graph to the host line topology with minimum number of swaps on the embedding. Both [1,6] present constant-competitive online algorithms for a fixed and complete binary tree, where nodes can swap and the demand is originating only from the source. However, these two works do not consider a specific demand graph. Moreover, [5] studied optimal but static and bounded-degree network topologies, when the demand is known. Self-adjusting networks have been formally organized and surveyed in [7]. Other existing online SAN algorithms consider different models. The most distinct difference is our focus on online re-embedding while keeping a fixed host graph (i.e., a line) compared to other works that focus on changing the network topology. The latter is what, for example, SplayNet [14] is proposing, where tree rotations change the form of the binary search tree network, without optimizing for a specific family of demand patterns.

Online demand graph re-embedding also relates to dynamically re-allocating network resources to follow traffic patterns. In [4], the authors consider a fixed set of clusters of bounded size, which contain all nodes and migrate nodes online according to the communication demand. But more broadly, [8] assumes a fixed grid network and migrates tasks according to their communication patterns.

Online embedding of metric spaces is studied in [11]. Authors consider the problem in which elements of some metric space are exposed one after another and the goal is to map them into another metric space while preserving the smallest expansion possible. There are several differences with our problem: 1) they care about all pairs of elements, while we consider a special demand graph; 2) nodes can not be re-embedded after being placed.

Also, relevant problems, from a migration point of view, are the classic list update problem (LU) [15], the related Itinerant List Update (ILU) problem [12], and the Minimum Linear Arrangement (MLA) problem [10]. In contrast to those problems, we study an online problem where requests occur between nodes.

Roadmap. Section 2 describes the model and background. Section 3 contains the summary of our three contributions (ladder, cycle, general demand graph) and their high-level proofs. Section 4 presents the algorithm and the analysis for ladder demand graphs. Some technical details can be found in the technical report [13].

2 Model and Background

Let us introduce the notation that we are going to use throughout the paper. Let $V(H)$ and $E(H)$ be the sets of vertices and edges in graph H, respectively. Sometimes, we just use V and E if the graph H is obvious from the context. Let $d_H(u, v)$ be the distance between u and v in graph H.

Let N be the network topology and σ be a sequence of pairwise communication requests between nodes in N. Let the demand graph G be the graph

built over the nodes in N and the pairs of nodes that appear in σ, i.e. $G = (V(N), \{\sigma_i = (s_i, d_i) \mid \sigma_i \in \sigma\})$. We assume that the demand graph is of a certain type and our overall goal will be to embed the demand graph G in the actual network topology N at a minimum cost. This is non-trivial as requests are selected from G by an online adversary and G is not known in advance. In the following, we formalize demand graph embedding and topology reconfiguration.

A configuration (or an embedding) of G (the demand graph) in a graph N (the host network) is a bijection of $V(G)$ to $V(N)$; $C_{G \to N}$ denotes the set of all such configurations. A configuration $c \in C_{G \to N}$ is said to serve a communication request $(u, v) \in E(G)$ at the cost $d_N(c(u), c(v))$. A finite communication sequence $\sigma = (\sigma_1, \ldots, \sigma_m)$ is served by a sequence of configurations $c_0, c_1, \ldots, c_m \in C_{G \to N}$. The cost of serving σ is the sum of serving each σ_i in c_i plus the reconfiguration cost between subsequent configurations c_i and c_{i+1}. The reconfiguration cost between c_i and c_{i+1} is the number of *migrations* necessary to change from c_i to c_{i+1}; a *migration* swaps the images of two neighbouring nodes u and v under c in N. Moreover, $E_i = \{\sigma_1, \ldots, \sigma_i\}$ denotes the first i requests of σ interpreted as a set of edges on V. We present algorithms for an online self-adjusting linear network: a network whose topology forms a 1-dimensional grid, i.e., a line.

Definition 1 (Working Model). *Let G be the demand graph, n be the number of vertices in G, $N = (\{1, \ldots, n\}, \{(1,2), (2,3), \ldots, (n-1, n)\}$ be a line (or list) graph L_n (host network), c be a configuration from $C_{G \to N}$, and σ be a sequence of communication requests. The cost of serving $\sigma_i = (u, v) \in \sigma$ is given by $|c(u) - c(v)|$, i.e., the distance between u and v in N. Migrations can occur before serving a request and can only occur between nodes configured on adjacent vertices in N.*

In the following we introduce notions relevant to our new results.

Definition 2 (Bandwidth). *Given a graph G, the* Bandwidth *of an embedding $c \in C_{G \to L_n}$ is equal to the maximum over all edges $(u, v) \in E$ of $|c(u) - c(v)|$, i.e., the distance between u and v on L_n.* Bandwidth(G) *is the minimum bandwidth over all embeddings from $C_{G \to L_n}$.*

Remark 1 *The* Bandwidth *computation of an arbitrary graph is an NP-hard problem [9].*

To save the space, we typically omit the proofs of lemmas and theorems in this paper and put them in [13, Appendix C]. Here we define the $2 \times n$ grid or ladder graph for which we get the main results of our paper.

Definition 3. *A graph $Ladder_n = (V, E)$ is represented as follows. The vertices V are the nodes of the grid $2 \times n$ — $\{(1,1), (1,2), \ldots, (1,n), (2,1), (2,2), \ldots, (2,n)\}$. There is an edge between vertices (x_1, y_1) and (x_2, y_2) iff $|x_1 - x_2| + |y_1 - y_2| = 1$.*

Lemma 1. Bandwidth($Ladder_n$) = 2.

Proof. The bandwidth is greater than 1, because there are nodes of degree three. The bandwidth of 2 can be achieved via the "level-by-level" enumeration as shown on the figure.

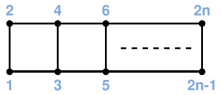


Fig. 1: Optimal ladder numeration.

Lemma 2. *For each subgraph S of a graph G,* Bandwidth(S) $\leqslant$ Bandwidth(G).

2.1 Background

Let us overview the previous results from [2]. In that work, both the demand and the host graph (network topology) were the line L_n on n vertices. It was shown that there exists an algorithm that performs $O(n^2 \log n)$ migrations in total, while serving the requests themselves in $O(1)$. By that, if the number of requests is $\Omega(n^2 \log n)$ then each request has $O(1)$ amortized cost.

Theorem 1 (Avin et al. [2]). *Consider a linear network L_n and a linear demand graph. There is an algorithm such that the total time spent on migrations is $O(n^2 \log n)$, while each request is performed in $O(1)$ omitting the migrations.*

We give an overview of this algorithm. At each moment in time, we know some subgraph of the line demand graph. For each new communication request, there are two cases: 1) the edge from the demand graph is already known — then, we do nothing; 2) the new edge is revealed. In the second case, this edge connects two connected components. We just move the smaller component on the line network closer to the larger component. The move of each node in one reconfiguration does not exceed n. Since, the total number of reconfigurations in which the node participates does not exceed $\log n$, we have $O(n^2 \log n)$ upper bound on the algorithm. From [2], $\Omega(n^2 \log n)$ is also the lower bound on the total cost. Thus, the algorithm is asymptotically optimal in the terms of complexity.

Corollary 1. *If $|\sigma| = \Omega(n^2 \log n)$ the amortized service cost per request is $O(1)$.*

The algorithms are not obliged to perform migrations at all, but the sum of costs for $\Theta(n^2)$ requests can be lower-bounded with $\Omega(n^2 \log n)$.

Theorem 2 (Lower bound, Avin et al. [2]). *For every online algorithm ON there is a sequence of requests σ_{ON} of length $\Theta(n^2)$ with the demand graph being a line, such that cost$(ON(\sigma_{ON})) = \Omega(n^2 \log n)$.*

That implies $\Omega(\log n)$ optimality (or competitive) factor since any offline algorithm knowing the whole request sequence σ in advance can simply reconfigure the network to match the (line) demand graph by paying $\Theta(n^2)$ in the worst case.

3 Summary of contributions

In this work we present self-adjusting networks with a line topology for a demand graph that is either a cycle, or a $2 \times n$ grid (ladder), or an arbitrary graph. We study offline and online algorithms on how to best embed the demand graph on the line, such that the total cost is minimized. The online case is more challenging, as the demand graph is revealed edge-by-edge and the embedding changes, with a cost. The result for the cycle follows from [2] almost directly. However, the result for the ladder is non-trivial and requires new techniques; it is not simple to reconfigure a subgraph on a $2 \times n$ grid after revealing a new edge in order to get $O(n^2 \log n)$ cost of modifications in total. We give an overview of each case below.

3.1 Cycle demand graph

We start with the following observation. Let C_n be a cycle graph on n vertices, i.e., $E(C_n) = \{(1,2), \ldots, (n-1,n), (n,1)\}$. Then, Bandwidth$(C_n) = 2$. We give a brief description of how the algorithm works. We start with the same algorithm as for the line (Section 2.1): while the number of revealed edges is not more than $n-1$, we can emulate the algorithm for the line. When the last edge appears we restructure the whole embedding in order to get bandwidth 2, which is the cycle bandwidth. For the last-step restructuring using swaps, we pay no more than $O(n^2)$. This cost is less than the total time spent on the reconstruction $\Omega(n^2 \log n)$.

Theorem 3. *Suppose the demand graph is C_n. There is an algorithm such that the total cost spent on the migrations is $O(n^2 \log n)$ and each request is performed in $O(1)$. In particular, if the number of requests is $\Omega(n^2 \log n)$ each request has $O(1)$ amortized cost.*

The full proof appears in [13, Appendix A]

Remark 2 *The lower bound with $\Omega(n^2 \log n)$ that was presented for a line demand graph still holds in the case of a cycle, since the cycle contains the line as the subgraph. Thus, our algorithm is optimal.*

3.2 Ladder demand graph

Now, we state the main result of the paper — the algorithm for the case when the demand graph is a ladder.

Theorem 4. *Suppose a demand graph is a ladder. There is an algorithm such that the total cost spent on the migrations is $O(n^2 \log n)$ and each request is performed in $O(1)$. In particular, if the number of requests is $\Omega(n^2 \log n)$ each request has $O(1)$ amortized cost.*

We provide a brief description of the algorithm. We say that a ladder has n levels from left to right: i.e., the nodes $(1, y)$ and $(2, y)$ are on the same level y (see Figure 1). On a high-level, we use the same algorithmic approach as in Theorem 1 for the line demand graph. The main difference is that instead of embedding the demand graph right away in the line network, at first, we "quasi-embed" the graph in the $2n$-ladder graph, which then we embed in the line. By "quasi-embedding" we mean a relaxation of the embedding defined earlier: at most **three** vertices of the demand graph are mapped to each level of the ladder.

Suppose for a moment that we have a dynamic algorithm that quasi-embeds the graph in the $2n$-ladder. Given this quasi-embedding we can then embed the $2n$-ladder in the line L_n. We sequentially go through from level 1 to level $2n$ of our ladder and map (at most three) vertices from the level to the line in some order (see Theorem 1). Such a transformation from the ladder to the line costs only a constant factor in bandwidth.

We explain briefly how to design a dynamic quasi-embedding algorithm with the desired complexity. At first, we present a static quasi-embedding algorithm, i.e., we are given a subgraph of the ladder and we need to quasi-embed it. This algorithm consists of three parts: embed a tree, embed a cycle, embed everything together. To embed a tree we find a special path in it, named trunk. We embed this trunk from left to right: one vertex per level. All the subgraphs connected to trunk are pretty simple and can be easily quasi-embedded in parallel to the trunk (see Figure 2). To embed a cycle we just have to decide which orientation it should have. To simplify the algorithm we embed only the cycles of length at least 6, omitting the cycles of length 4. This decision introduces just the multiplicative constant of the cost. Finally, we embed the whole graph: we construct its cycle-tree decomposition and embed cycles and trees one by one from left to right.

Now, we give a high-level description of our dynamic algorithm. We maintain the invariant that all the components are quasi-embedded. When an already served request (edge) appears, we do nothing. The complication comes from a newly revealed edge-request. There are two cases. The first one is when the edge connects nodes in the same component — thus, there is a cycle. We redo only the part of the quasi-embedding of the component around the new cycle; the rest of the component remains. In the second case, the edge connects two components. We move the smaller component to the bigger one as in Theorem 1. The bigger component does not move and we redo the quasi-embedding of the smaller one.

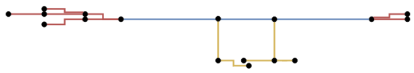


Fig. 2: Quasi-correct embedding of a tree

Now, we briefly calculate the complexity of the dynamic algorithm. For the requests of the first case, if the nodes are on the cycle for the first time (this event happens only once for each node), we pay $O(n)$ for it. Otherwise, there are already nodes in the cycle. In this case we make sure to re-embed the existing cycle in a way that all the nodes are moved for a $O(1)$ distance. As for the neighboring nodes, it can be shown that each node is moved only once as a part of the cycle neigh-

borhood, so we also bound this movement with $O(n)$ cost. This gives us $O(n^2)$ complexity in total — each node is moved by at most $O(n)$. For the requests of the second case, we always move the smaller component and, thus, we pay $O(n^2 \log n)$ in total: each node can be moved by $O(n)$ at most $O(\log n)$ times, i.e., any node can be at most $\log n$ times in the "smaller" component. Our algorithm matches the lower bound, since the ladder contains L_n as a subgraph.

3.3 General graph

We finish the list of contributions with a general result; the case where the demand graph is an arbitrary graph G. The full proofs are available in [13, Appendix D].

Theorem 5. *Suppose we are given a (demand) graph G and an algorithm B, that for any subgraph S of G outputs an embedding $c \in C_{S \to L_{|V(G)|}}$ with bandwidth less than or equal to $\lambda \cdot \text{Bandwidth}(G)$ for some λ. Then, for any sequence of requests σ with a demand graph G there is an algorithm that serves σ with a total cost of $O(|E(G)| \cdot |V(G)|^2 + \lambda \cdot \text{Bandwidth}(G) \cdot |\sigma|)$. In particular, if the number of requests is $\Omega(|E(G)| \cdot |V(G)|^2)$ each request has $O(\lambda \cdot \text{Bandwidth}(G))$ amortized cost.*

Here we give a brief description of the algorithm. Suppose that the current configuration c_i is the embedding of the current demand graph G_i in $L_{|V(G)|}$ after i requests. Now, we need to serve a new request in $\lambda \cdot \text{Bandwidth}(G_i) \leqslant \lambda \cdot \text{Bandwidth}(G)$. If the corresponding edge already exists in the demand graph, we simply serve the request without the reconfiguration. Now, suppose the request reveals a new edge and we get the demand graph G_{i+1}. Using the algorithm B we get the configuration (embedding) c_{i+1} that has $\lambda \cdot \text{Bandwidth}(G_{i+1}) \leqslant \lambda \cdot \text{Bandwidth}(G)$. Please note that we do not put any constraints on the algorithm B: typically this problem is NP-complete. To serve the request fast, we should rebuild the configuration c_i into the configuration c_{i+1}. By using the swap operations on the line we can get from c_i to c_{i+1} in $O(|V(G)|^2)$ operations: each vertex moves by at most $V(G)$. After the reconfiguration we can serve the request with the desired cost.

A new edge appears at most $|E(G)|$ times while the reconfiguration costs $|V(G)|^2$. Each request is served in $\lambda \cdot \text{Bandwidth}(G)$. Thus, the total cost of requests σ is $O(|E(G)| \cdot |V(G)|^2 + \lambda \cdot \text{Bandwidth}(G) \cdot |\sigma|)$.

Lemma 3. *Given a demand graph G. For each online algorithm ON there is a request sequence σ_{ON} such that ON serves each request from σ_{ON} for a cost of at least* $\text{Bandwidth}(G)$.

4 Embedding a ladder demand graph

We present our algorithms for embedding a demand graph that is a subgraph of the ladder graph ($2 \times n$-grid) on the line. We first present the offline case,

where the demand graph is known in advance (Section 4.1). Then we present the dynamic case, where requests are revealed online, revealing also the demand graph and thus possibly changing the current embedding (Section 4.2). Finally, we discuss the cost of the dynamic case in Section 4.3.

Though our final goal is to embed a demand graph into the line, we will first focus on how to embed a partially-known demand graph into $Ladder_N$, where N is large enough to make the embedding possible, i.e., no more than $2n$. When we have such an embedding one might construct an embedding from $Ladder_N$ into $Line_n$, simply composing it with a level by level (see the proof of Lemma 1) embedding of $Ladder_N$ to $Line_{2N}$ and then by omitting empty images we get $Line_n$. Such a mapping of $Ladder_N$ to $Line_{2N}$ enlarges the bandwidth for at most a factor of 2, but significantly simplifies the construction of our embedding.

Definition 4. *A ladder graph l consists of two line-graphs on n vertices l_1 and l_2 with additional edges between the lines: $\{(l_1[i], l_2[i]) \mid i \in [n]\}$, where $l_j[i]$ is the i-th node of the line-graph l_j. We call the set of two vertices, $\{l_1[i], l_2[i]\}$, the i-th level of the ladder and denote it as $level_{Ladder_n}(i)$ or just $level(i)$ if it is clear from the context. We refer to $l_1[i]$ and $l_2[i]$ as $level(i)[1]$ and $level(i)[2]$, respectively. We say that $level\langle v\rangle = i$ for $v \in V(Ladder_n)$ if $v \in level_{Ladder_n}(i)$. We refer to l_1 and l_2 as the sides of the ladder.*

Definition 5. *A* correct embedding *of a graph A into a graph B is an injective mapping $\varphi : V(A) \to V(B)$ that preserves edges, i.e.*

$$\begin{cases} \forall u, v \in V(A) \text{ with } u \neq v \Rightarrow \varphi(u) \neq \varphi(v) \\ (u, v) \in E(A) \Rightarrow (\varphi(u), \varphi(v)) \in E(B) \end{cases}$$

4.1 Static quasi-embedding

We start with one of the basic algorithms — how to quasi-embed any graph that can be embedded in $Ladder_n$ onto $Ladder_N$ with large N . We present a tree and cycle embedding and then we show how to to combine them in an embedding of a general component (by first doing a cycle-tree decomposition). The whole algorithm is presented in [13, Appendix B.1].

Tree embedding In this case, our task is to embed a tree on a ladder graph. We start with some definitions and basic lemmas.

Definition 6. *Consider some correct embedding φ of a tree T into $Ladder_n$. Let $r = \arg\max_{v \in V(T)} level\langle\varphi(v)\rangle$ and $l = \arg\min_{v \in V(T)} level\langle\varphi(v)\rangle$ be the "rightmost" and "leftmost" nodes of the embedding, respectively. The trunk of T is a path in T connecting l and r. The trunk of a tree T for the embedding φ is denoted with $trunk_\varphi(T)$.*

Definition 7. *Let T be a tree and φ be its correct embedding into $Ladder_n$. The level i of $Ladder_n$ is called* occupied *if there is a vertex $v \in V(T)$ on that level, i.e., $\varphi(v) \in level_{Ladder_n}(i)$.*

Statement 1 *For every occupied level i there is $v \in trunk_\varphi(T)$ such that $v \in level(i)$.*

Proof. By the definition of the trunk, an image goes from the minimal occupied level to the maximal. It cannot skip a level since the trunk is connected and the correct embedding preserves connectivity.

The trunk of a tree in an embedding is a useful concept to define since the following holds for it. The proofs for the lemmas in this section appear in [13, Appendix C].

Lemma 4. *Let T be a tree correctly embedded into $Ladder_n$ by some embedding φ. Then, all the connected components in $T \setminus trunk_\varphi(T)$ are line-graphs.*

Lemma 5. *For the tree T and for each node v of degree three (except for maximum two of them) we can verify in polynomial time if for any correct embedding φ, $trunk_\varphi(T)$ passes through v or not.*

Support nodes are the nodes of two types: either a node of degree three without neighbours of degree three or a node that is located on some path between two nodes with degree three. The path through passing through all support nodes is called *trunk core*. We denote this path for a tree T as $trunkCore(T)$. Intuitively, the trunk core consists of vertices that lie on a trunk of any embedding. It can be proven that the support nodes appear in the trunk of every correct embedding (proof appears in [13]).

Definition 8. *Let T be a tree. All the connected components in $T \setminus trunkCore(T)$ are called* simple-graphs *of tree T.*

Lemma 6. *The simple-graphs of a tree T are line-graphs.*

Definition 9. *The edge between a simple-graph and the trunk core is called a* leg*. The end of a leg in the simple-graph is called a* head *of the simple-graph. The end of a leg in the trunk core is called a* foot *of the simple-graph. If you remove the head of a simple-graph and it falls apart into two connected components, such simple-graph is called* two-handed *and those parts are called its* hands*. Otherwise, the graph is called* one-handed*, and the sole remaining component is called a* hand*. If there are no nodes in the simple-graph but just a head it is called zero-handed.*

Fig. 3: Hands, Legs, and Trunk core.

Definition 10. *A simple-graph connected to some end node of the trunk core is called* exit-graph*. A simple-graph connected to an inner node of the trunk core is called* inner-graph*.*

Please note that the next definition is about a much larger ladder graph, $Ladder_N$, rather than $Ladder_n$. Here, N is equal to $2n$ to make sure that we have enough space to embed.

Definition 11. *An embedding $\varphi : V(G) \to V(Ladder_N)$ of a graph G into $Ladder_N$ is called* quasi-correct *if:*

- $(u, v) \in E(G) \Rightarrow (\varphi(u), \varphi(v)) \in E(Ladder_N)$*, i.e., images of adjacent vertices in G are adjacent in the ladder.*
- *There are no more than* ***three*** *nodes mapped into each level of $Ladder_N$, i.e., the two ladder nodes on each level are the images of no more than three nodes.*

We can think of a quasi-correct embedding as an embedding into levels of the ladder with no more than three nodes embedded to the same level. Then, we can compose this embedding with an embedding of a ladder into the line which is the enumeration level by level. More formally if a node u is embedded to level i and a node v is embedded to level j and $i < j$ then the resulting number of u on the line is smaller than the number of v, but if two nodes are embedded to the same level, we give no guarantee.

Lemma 7. *Any graph mapped into the ladder graph by the quasi-correct embedding described above can be mapped to the line level by level with the property that any pair of adjacent nodes are embedded at the distance of at most five.*

Assume, we are given a tree T that can be embedded into $Ladder_n$. Furthermore, there are two special nodes in the tree: one is marked as R (right) and another one is marked as L (left). It is known that there exists a correct embedding of T into $Ladder_n$ with R being the rightmost node, meaning no node is embedded more to the right or to the same level, and L being the leftmost node.

We now describe how to obtain a quasi-correct embedding of T in $Ladder_N$ with R being the rightmost node and L being the leftmost one while L is mapped to $ImageL$ — some node of the $Ladder_N$. Moreover, our embedding obeys the following invariant.

Invariant 1 (Septum invariant) *For each inner simple-graph, its foot and its head are embedded to the same level and no other node is embedded to that level.*

We embed a path between L and R simply horizontally and then we orient line-graphs connected to it in a way that they do not violate our desired invariant. It can be shown that it is always possible if T can be embedded in $Ladder_n$. The pseudocode is in [13, Appendix, Algorithm 1].

Suppose now that not all information, such as R, L, and $ImageL$, is provided. We explain how we can embed a tree T. We first get the *trunk core* of the given tree. This can be done by following the definition. Now

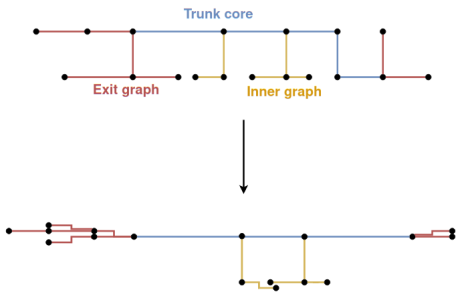


Fig. 4: Example of a quasi-correct embedding

the idea would be to first embed the trunk core and its inner line-graphs using a tree embedding presented earlier with R and L to be the ends of the trunk core. Then, we embed exit-graphs strictly horizontally "away" from the trunk core. That means, that the hands of exit-graphs that are connected to the right of the trunk core are embedded to the right, and the hands of those exit-graphs that are connected to the left of the trunk core are embedded to the left. An example of the quasi-correct embedding is shown in Figure 4.

If a tree does not have a trunk core, then its structure is quite simple (in particular it has no more than two nodes of degree three). Such a tree can be embedded without conflicts. The pseudocode appears in [13, Appendix, Algorithm 2].

Cycle embedding Now, we show how to embed a cycle into $Ladder_N$. First, we give some important definitions and lemmas.

Definition 12. *A* maximal *cycle C of a graph G is a cycle in G that cannot be enlarged, i.e., there is no other cycle C' in G such that $V(C) \subsetneq V(C')$.*

Definition 13. *Consider a graph G and a maximal cycle C of G. A whisker W of C is a line inside G such that: 1) $V(W) \neq \varnothing$ and $V(W) \cap V(C) = \varnothing$. 2) There exists only one edge between the cycle and the whisker (w, c) for $w \in V(W)$ and $c \in V(C)$. Such c is called a* foot *of W. The nodes of W are enumerated starting from w. 3) W is maximal, i.e., there is no W' in G such that W' satisfies previous properties and $V(W) \subsetneq V(W')$.*

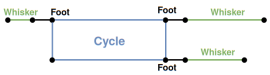


Fig. 5: Cycle and its Whiskers.

Definition 14. *Suppose we have a graph G that can be correctly embedded into $Ladder_n$ by φ and a cycle C in G. Whiskers W_1 and W_2 of C are called adjacent (or neighboring) for the embedding φ if $\forall i \leqslant \min(|V(W_1)|, |V(W_2)|)$ $(\varphi(W_1[i]), \varphi(W_2[i])) \in E(Ladder_n)$.*

Lemma 8. *Suppose we have a graph G that can be correctly embedded into $Ladder_n$ and there exists a maximal cycle C in G with at least 6 vertices with two neighbouring whiskers W_1 and W_2 of C, i.e., $(foot(W_1), foot(W_2)) \in E(G)$. Then, W_1 and W_2 are adjacent in any correct embedding of G into $Ladder_N$.*

Definition 15. *Assume, we have a graph G and a maximal cycle C of length at least 6. The frame for C is a subgraph of G induced by vertices of C and $\{W_1[i], W_2[i] \mid i \leqslant \min(|V(W_1)|, |V(W_2)|)\}$ for each pair of adjacent whiskers W_1 and W_2. Adding all the edges $\{(W_1[i], W_2[i]) \mid i \leqslant \min(|V(W_1)|, |V(W_2)|)\}$ for each pair of adjacent whiskers W_1 and W_2 makes a frame* completed.

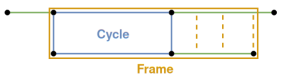
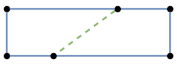


Fig. 6: Cycle, its frame, and edges (dashed) to make the frame completed

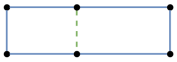
Given a cycle C of length at least six and its special nodes $L, R \in V(C)$, we construct a correct embedding of C into $Ladder_N$ with $level\langle L\rangle \leqslant level\langle u\rangle \leqslant level\langle R\rangle$ for all u in $V(C)$, while L is mapped into the node $ImageL$.

We first check if it is possible to satisfy the given constraints of placing the L node to the left and a R node to the right. If it is indeed possible, we place L to the desired place $ImageL$ and then we choose an orientation (clockwise or counterclockwise) following which we could embed the rest of the nodes, keeping in mind that R must stay on the rightmost level. The pseudocode appears in [13, Appendix, Algorithm 3].

Now, suppose that not all the information, such as R, L, and $ImageL$, is provided. We reduce this problem to the case when the missing variables are known. This subtlety might occur since there are inner edges in the cycle. In this case, we choose missing L/R more precisely in order to embed an inner edge vertically. For more intuition, please see Figures 7a and 7b. A dashed line denotes an inner edge. The pseudocode appears in [13, Appendix, Algorithm 4].



(a) Incorrect cycle embedding



(b) Correct cycle embedding

Fig. 7: Cycle embeddings.

Embedding a connected component of the demand graph Combining the previous results, we can now explain how to embed in $Ladder_N$ a connected component S that can be embedded in $Ladder_n$.

Definition 16. *By the cycle-tree decomposition of a graph G we mean a set of maximal cycles $\{C_1, \dots C_n\}$ of G and a set of trees $\{T_1, \dots, T_m\}$ of G such that*

- $\bigcup_{i\in[n]} V(C_i) \cup \bigcup_{i\in[m]} V(T_i) = V(G)$
- $V(C_i) \cap V(C_j) = \varnothing \ \forall i \neq j$
- $V(T_i) \cap V(T_j) = \varnothing \ \forall i \neq j$
- $V(T_i) \cap V(C_j) = \varnothing \ \forall i \in [m], j \in [n]$
- $\forall i \neq j \ \forall u \in V(T_i) \ \forall v \in V(T_j) \ (u, v) \notin E(G)$

We start with an algorithm on how to make a cycle-tree decomposition of S assuming no incomplete frames. To obtain a cycle-tree decomposition of a graph: 1) we find a maximal cycle; 2) we split the graph into two parts by logically removing the cycle; 3) we proceed recursively on those parts, and, finally, 4) we combine the results together maintaining the correct order between cycle and two parts (first, the result for one part, then the cycle, and then the result for the second part). Since we care about the order of the parts, we say that it is

a *cycle-tree decomposition chain*. The decomposition pseudocode appears in [13, Appendix, Algorithm 5].

We describe how to obtain a quasi-correct embedding of S. We preprocess S: 1) we remove one edge from cycles of size four; 2) we complete incomplete frames with vertical edges. Then, we embed parts of S from the cycle-tree decomposition chain one by one in the relevant order using the corresponding algorithm (either for a cycle or for a tree embedding) making sure parts are glued together correctly. The pseudocode appears in [13, Appendix, Algorithm 6].

4.2 Online quasi-embedding

In the previous subsection, we presented an algorithm on how to quasi-embed a static graph. Now, we will explain how to operate when the requests are revealed in an online manner. The full version of the algorithm is presented in [13, Appendix B.2].

There are two cases: a known edge is requested or a new edge is revealed. In the first case the algorithm does nothing since we already know how to quasi-correctly embed the current graph and, thus, we already can embed into the line network with constant bandwidth. Thus, further, we consider only the second case.

We describe how one should change the embedding of the graph after the processing of a request in an online scenario. At each moment some edges of the demand graph $Ladder_n$ are already revealed, forming connected components. After an edge reveal we should reconfigure the target line topology. For that, instead of line reconfiguration we reconfigure our embedding onto $Ladder_N$ that is then embedded to the line level by level and introduces a constant factor. So, we can consider the reconfiguration only of $Ladder_N$ and forget about the target line topology at all. When doing the reconfiguration of an embedding we want to maintain the following invariants:

1. The embedding of any connected component is quasi-correct.
2. For each tree in the cycle-tree decomposition its embedding respects Septum invariant 1.
3. There are no maximal cycles of length 4.
4. Each cycle frame is completed with all "vertical" edges even if they are not yet revealed.
5. There are no conflicts with cycle nodes, i.e., each cycle node is the only node mapped to its image in the embedding to $Ladder_N$.

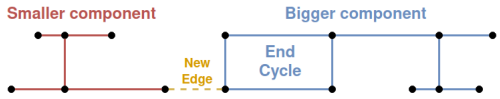
For each newly revealed edge there are two cases: either it connects two nodes from one connected component or not. We are going to discuss both of them.

Edge in one component The pseudocode appears in [13, Appendix, Algorithm 8]. If the new edge is already known or it forms a maximal cycle of length four, we simply ignore it. Otherwise, it forms a cycle of length at least six, since two connected nodes are already in one component. We then perform the following steps:

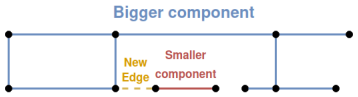
1. Get the completed frame of a (possibly) new cycle.
2. Logically "extract" it from the component and embed maintaining the orientation (not twisting the core that was already embedded in some way).
3. Attach two components appeared after an extraction back into the graph, maintaining their relative order.

Edge between two components The pseudocode appears in [13, Appendix, Algorithm 9]. In order to obtain an amortization in the cost, we always "move" the smaller component to the bigger one. Thus, the main question here is how to glue a component to the existing embedding of another component. The idea is to consider several cases of where the smaller component will be connected to the bigger one. There are three possibilities:

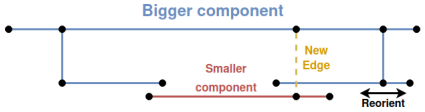
1. *It connects to a cycle node.* In this case, there are again two possibilities. Either it "points away" from the bigger component meaning that the cycle to which we connect is the one of the ends in the cycle-tree decomposition of the bigger component. Here, we just simply embed it to the end of the cycle-tree decomposition while possibly rotating a cycle at the end.



   Or, the smaller component should be placed somewhere between two cycles in the cycle-tree decomposition. Here, it can be shown that this small graph should be a line-graph, and we can simply add it as a whisker, forming a larger frame.



2. *It connects to a trunk core node of a tree in the cycle-tree decomposition.* It can be shown that in this case the smaller component again must be a line-graph. Thus, our only goal is to orient it and possibly two of its inner simple-graphs neighbours to maintain the Septum invariant 1 for the corresponding tree from the decomposition.



3. *It connects to an exit graph node of an end tree of the cycle-tree decomposition.* In this case, we straightforwardly apply a static embedding algorithm of this tree and the smaller component from scratch. Please, note that only the exit graphs of the end tree will be moved since the trunk core and its inner graphs will remain.

4.3 Complexity of the online embedding

Now, we calculate the cost of our online algorithm (a more detailed discussion on the cost of the algorithm appears in [13, Appendix C.5]): how many swaps we should do and how much we should pay for the routing requests. Recall that we first apply the reconfiguration and, then, the routing request.

We start with considering the routing requests. Their cost is $O(1)$ since they lie pretty close on the target line network, i.e., by no more than 12 nodes apart. This bound holds because the nodes are quasi-correctly embedded on $Ladder_N$, two adjacent nodes at G are located not more than four levels apart (in the worst case, when we remove an edge of a cycle with length four) where each level of the quasi-correct embedding has at most three images of nodes of G. Thus, on the target line, if we enumerate level by level, the difference between any two adjacent nodes of G is at most 12.

Then, we consider the reconfiguration. We count the total cost of each case of the online algorithm before all the edges are revealed.

In the first case, we add an edge in one component. By that, either a new frame is created or some frame was enlarged. In both cases, only the nodes, that appear on some frame for the first time, are moved. Since, a node can be moved only once to be mapped to a frame and it is swapped at most $N = O(n)$ times to move to any position, the total cost of this type of reconfiguration is at most $O(n^2)$. Also, there are several adjustments that could be done: 1) the "old" frame can rotate by one node, and 2) possibly, we should flip the first inner-graphs of two components connected to the frame. In the first modification, each node at the frame can only be "rotated" once, thus, paying $O(n)$ cost in total. In the second modification, inner-graph can change orientation at most once in order to satisfy the Septum invariant (Invariant 1), thus, paying $O(n^2)$ cost in total — each node can move by at most $N = O(n)$.

In the second case, we add an edge in between two components. At first, we calculate the time spent on the move of the small component to the bigger one: each node is moved at most $O(\log n)$ times since the size of the component always grows at least two times, the number of swaps of a vertex is at most $N = O(n)$ to move to any place, thus, the total cost is $O(n^2 \log n)$. Secondly, there are two more modification types: 1) a rotation of a cycle, and 2) some simple-graphs can be reoriented. The cycle can be rotated only once, thus, we should pay at most $O(n)$ there. At the same time, each simple-graph can be reoriented at most once to satisfy the Septum invariant (Invariant 1), thus, the total cost is $O(n^2)$ for that type of a reconfiguration.

To summarize, the total cost of requests σ is $O(n^2 \log n)$ for the whole reconfiguration plus $O(|\sigma|)$ per requests. This matches the lower bound that was obtained for the line demand graph. The same result holds for any demand graph that is the subgraph of the ladder of size n.

Theorem 6. *The online algorithm for embedding the ladder demand graph of size n on the line has total cost $O(n^2 \log n + |\sigma|)$ for a sequence of communication requests σ.*

5 Conclusion

We presented methods for statically or dynamically re-embedding a ladder demand graph (or a subgraph of it) on a line, both in the offline and online case. As side results, we also presented how to embed a cycle demand graph and a meta-algorithm for a general demand graph. Our algorithms for the cycle and the ladder cases match the lower bounds. Our work is a first step towards a tight bound on dynamically re-embedding more generic demand graphs, such as arbitrary $k \times n$ grids.

References

1. Avin, C., Bienkowski, M., Salem, I., Sama, R., Schmid, S., Schmidt, P.: Deterministic self-adjusting tree networks using rotor walks. In: 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS). pp. 67–77. IEEE (2022)
2. Avin, C., van Duijn, I., Schmid, S.: Self-adjusting linear networks. In: International Symposium on Stabilizing, Safety, and Security of Distributed Systems. pp. 368–382. Springer (2019)
3. Avin, C., Ghobadi, M., Griner, C., Schmid, S.: On the complexity of traffic traces and implications. Proceedings of the ACM on Measurement and Analysis of Computing Systems **4**(1), 1–29 (2020)
4. Avin, C., Loukas, A., Pacut, M., Schmid, S.: Online balanced repartitioning. In: International Symposium on Distributed Computing. pp. 243–256. Springer (2016)
5. Avin, C., Mondal, K., Schmid, S.: Demand-aware network design with minimal congestion and route lengths. IEEE/ACM Transactions on Networking (2022)
6. Avin, C., Mondal, K., Schmid, S.: Push-down trees: optimal self-adjusting complete trees. IEEE/ACM Transactions on Networking **30**(6), 2419–2432 (2022)
7. Avin, C., Schmid, S.: Toward demand-aware networking: a theory for self-adjusting networks. ACM SIGCOMM Computer Communication Review **48**(5), 31–40 (2019)
8. Batista, D.M., da Fonseca, N.L.S., Granelli, F., Kliazovich, D.: Self-adjusting grid networks. In: 2007 IEEE international conference on communications. pp. 344–349. IEEE (2007)
9. Díaz, J., Petit, J., Serna, M.: A survey of graph layout problems. ACM Computing Surveys (CSUR) **34**(3), 313–356 (2002)
10. Hansen, M.D.: Approximation algorithms for geometric embeddings in the plane with applications to parallel processing problems. In: 30th Annual Symposium on Foundations of Computer Science. pp. 604–609. IEEE Computer Society (1989)
11. Newman, I., Rabinovich, Y.: Online embedding of metrics. arXiv preprint arXiv:2303.15945 (2023)
12. Olver, N., Pruhs, K., Schewior, K., Sitters, R., Stougie, L.: The itinerant list update problem. In: International Workshop on Approximation and Online Algorithms. pp. 310–326. Springer (2018)
13. Paramonov, A., Salem, I., Schmid, S., Aksenov, V.: Self-adjusting linear networks with ladder demand graph. arXiv preprint arXiv:2207.03948 (2022)
14. Schmid, S., Avin, C., Scheideler, C., Borokhovich, M., Haeupler, B., Lotker, Z.: Splaynet: Towards locally self-adjusting networks. IEEE/ACM Trans. Netw. **24**(3), 1421–1433 (2016)
15. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Communications of the ACM **28**(2), 202–208 (1985)