Санкт–Петербургский государственный университет
Факультет математики и компьютерных наук

Антон Игоревич Парамонов

Выпускная квалификационная работа

Классы графов для саморегулирующихся сетей

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5005.2018 «Прикладная математика, фундаментальная информатика и программирование»
Профиль «Современное программирование»

Научный руководитель:
профессор, А.С. Куликов

Рецензент:
Пацут Мачиек, научный сотрудник
Берлинский Технический университет

Санкт-Петербург
2022 г.

Saint–Petersburg State university
Faculty of math and computer science

Anton Igorevich Paramonov

Bachelor thesis

Self-Adjusting Linear Networks with Ladder Demand Graph

Academic degree: bachelor
Program 01.03.02 «Applied Mathematics and Computer Science»
Main program CB.5005.2018 «Applied mathematics, fundamental informatics and programming»
Specialization «Modern programming»

Scientific director:
Professor A.S. Kulikov

Reviewer:
Maciej Pacut, researcher
Technical University Berlin

Saint-Petersburg
year 2022

Contents

1. Introduction

Networked and distributed systems are becoming increasingly flexible. In particular, it has become possible to dynamically migrate communication endpoints [2] in a self-adjusting and demand-aware manner, allowing these networks to adapt to the ongoing traffic pattern. In particular, more frequently communicating nodes can be moved topologically closer to each other, thus, reducing bandwidth taxes [11] and improving performance.

We consider the following problem. The network is a line and the graph built by pairwise requests is G, i.e., the *demand graph*. The requests between nodes are coming online, i.e. one by one, revealing G. Before performing a request, we can re-adjust the line graph by performing several swaps of two neighbouring nodes and by paying one for each swap. Please, note that it does not matter when we perform adjustments, before or after requests, the total cost will differ by a constant. Then, to serve a request itself, we should follow the path between the corresponding nodes in the line network and pay each time we pass a link, i.e., in total, we pay the distance between the nodes per request. Thus, the total cost of an algorithm consists of two parts: the total cost of adjustments, i.e., swaps, plus the total cost of the requests.

In the previous work [7], 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. This problem was inspired by Itinerant List Update Problem [14].

Contributions. In this work, we go further: we consider the same line network topology while trying to generalize the demand graph. As a result, we provide three separate results. At first, the generic algorithm for arbitrary demand graphs with the best complexity per request (in a theoretical sense). Secondly, we present an algorithm for the case when the demand graph is a cycle. And, finally, as the main result, we present an algorithm for a generalization of a line demand graph: a $2 \times n$ grid graph, later called the n-ladder graph. Note that this algorithm can be applied to any demand graph that is a subgraph of the n-ladder graph. Most importantly, both algorithms, for a cycle and a ladder graph, match the lower bound for the line demand graph: $n^2 \log n$ cost for adjustments in total

plus $O(1)$ per request being the lower bound.

Our work is an important improvement of the previous work. A solution for the n-ladder grid is the first step towards a more generic problem where the demand graph is a generic grid, $n \times m$. One of the main arising problems is that even an embedding of a tree onto an infinite grid is NP-hard [12], while it is polynomial in our case leading to a reasonably simple algorithm. Moreover, a ladder (and a cycle) has a constant *bandwidth*, i.e., a minimum value over all embeddings onto a target line graph 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. However, the calculation of the bandwidth, in general, is also NP-hard [15].

Related work. Self-adjusting networks (SANs) have been made possible due to novel network technologies [10]. They were introduced from an algorithmic point of view in [5], [16]. Existing works have studied linear networks [7], bounded degree networks [6], skip list networks [4] and skip graphs [13]. Moreover, recent work has also focused in more advanced cost models [1], [9]. The most relevant work to ours [7] studied linear networks with linear demand graphs. Furthermore, [3] studied optimal network topologies, when the demand is known.

Roadmap. Section 2 contains the description of the model in which we are working. Section 3 contains the list of our three results and their high-level proofs. This list contains the generic result, the result for the cycle, and the results for the ladder. In Section 4 we introduce all the necessary notions for our algorithm on the grid. Section 5 presents an algorithm for the ladder. In Section 6, we calculate the complexity of the presented algorithm and show that it achieves the desired cost. Section 9 concludes the paper. Due to space constraints, some technical details are deferred to the appendix. The pseudocode of our algorithm for the ladder appears in Appendix 7.2 and 7.1, while proofs appear in Appendix 8.

2. Model

Let us introduce the notation that we are going to use throughout the paper.

- Let $d_G(u, v)$ be the distance between u and v in graph G.
- A configuration (or an embedding) of V in a graph N (the host network) is

an injection of V into the vertices of N; $C_{V \to N}$ denotes the set of all such configurations.

- A configuration $h \in C_{V \to N}$ is said to serve a communication request $(u, v) \in V \times V$ at cost $d_N(h(u), h(v))$.
- A finite communication sequence $\sigma = (\sigma_1, \ldots, \sigma_m)$ is served by a sequence of configurations $h_0, h_1, \ldots, h_m \in C_{V \to N}$.
- The cost of serving σ is the sum of serving each σ_i in h_i plus the reconfiguration cost between subsequent configurations h_i and h_{i+1}.
- The reconfiguration cost between h_i and h_{i+1} is the number of migrations necessary to change from h_i to h_{i+1}; a migration swaps the images of two neighbouring nodes u and v under h in N.
- $E_i = \{\sigma_1, \ldots, \sigma_i\}$ denotes the first i requests of σ interpreted as a set of edges on V, and $D(\sigma) = (V, E_m)$ denotes the demand graph of σ.

In particular, we study the problem of designing an online self-adjusting linear network: a network whose topology forms a 1-dimensional grid, i.e., a line.

Definition 2.1 (Working Model)**.** Let V, h, and σ be as before, with $N = (\{1, \ldots, n\}, \{(1,2), (2,3), \ldots, (n-1, n)\}$ representing a list graph L_n. The cost of serving $\sigma_i = (u, v) \in \sigma$ is given by $|h(u) - h(v)|$, 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.

Let us talk a little bit about the previous results from [7]. In that work, the demand graph was the line graph L_n, as the network graph. 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 costs $O(1)$ amortized time. Moreover, $n^2 \log n$ appears to be the lower bound: if there are $\Theta(n^2)$ requests then the total cost is $\Omega(n^2 \log n)$, meaning that the static-optimality factor is $\log n$.

Theorem 2.0.1. *Consider a linear network L_n and the linear demand graph $D(\sigma)$ where σ is the sequence of requests. 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.

Corollary 2.0.1. *If $|\sigma| = \Omega(n^2 \log n)$ then the amortized cost of serving each request is $O(1)$ amortized time.*

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 2.0.2 (Lower Bound)**.** *For every online algorithm ON there is a sequence of requests σ_{ON} of length $\Theta(n^2)$ with $D(\sigma_{ON})$ being a line such that $cost(ON(\sigma_{ON})) = \Omega(n^2 \log n)$.*

That implies $\Omega(\log n)$ static-optimality factor since any offline algorithm knowing a whole request sequence σ in advance can simply reconfigure a network to match $D(\sigma)$ paying $\Theta(n^2)$ in the worst case.

3. List of results

In this work, we take into consideration demand graphs different from the line, considered in [7]. We present the general result for arbitrary graphs and more tight results for the generalizations of the line graph: the cycle and the ladder. The result for the cycle follows from [7] almost directly. However the result for the ladder is non-trivial and very technical — 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.

3.1. Preliminaries

Definition 3.1. An *embedding* of a graph G into graph H is an injective mapping $\varphi : V(G) \to V(H)$. The set of all embeddings of G into H is denoted as $C_{G \to H}$.

Definition 3.2. A *correct embedding* of a graph G into graph H is an injective mapping $\varphi : V(G) \to V(H)$ that preserves edges, i.e.

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

Definition 3.3 (Bandwidth)**.** Given a graph G, the *Bandwidth* of an embedding $h \in C_{G \to L_n}$ is equal to the maximum over all edges $(u, v) \in E$ of $|h(u) - h(v)|$, i.e., the distance between u and v on L_n.

$\mathrm{Bandwidth}(G)$ is the minimum bandwidth over all embeddings from $C_{G \to L_n}$.

Remark 3.1.1. *The computation of a* $\mathrm{Bandwidth}$ *of an arbitrary graph is an NP-hard problem [8].*

To save the space, we typically omit the proofs of lemmas and theorems in this paper and put them in Appendix 8.

Lemma 3.1.1. *Let C_n be a cycle graph on n vertices, i.e., $E = \{(1, 2), \ldots, (n - 1, n), (n, 1)\}$. Then,* $\mathrm{Bandwidth}(C_n) = 2$.

Here we define the $2 \times n$ grid or ladder graph for which we get the main results of our paper.

Definition 3.4. A graph $Grid_n = (V, E)$ 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 3.1.2. $\mathrm{Bandwidth}(Grid_n) = 2$.

Lemma 3.1.3. *For each subgraph S of a graph G,* $\mathrm{Bandwidth}(S) \leq \mathrm{Bandwidth}(G)$.

3.2. General result

At first, we present a general result — when the demand graph is an arbitrary graph G.

Theorem 3.2.1. *Suppose we are given a graph G and an algorithm B, that for any subgraph S of G outputs an embedding $h \in C_{S \to L_n}$ with the bandwidth less than or equal to $\lambda \cdot \mathrm{Bandwidth}(G)$ 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 $O(|E(G)| \cdot |V(G)|^2 + \lambda \cdot \mathrm{Bandwidth}(G) \cdot |\sigma|)$. In particular, if the number of requests is $\Omega(|E(G)| \cdot |V(G)|^2)$ each request has $O(\lambda \cdot \mathrm{Bandwidth}(G))$ amortized cost.*

Proof. Assume we have processed i requests so far. We get a demand graph built on edges $E_i = \{\sigma_0, \ldots, \sigma_i\}$. It induces a subgraph S_i of G.

We want to maintain the invariant that each S_i is embedded via $h \in C_{S_i \to L}$ such that bandwidth of h is no greater than $\lambda \cdot \mathrm{Bandwidth}(G)$.

Suppose now the embedding h_{i-1} of S_{i-1} respects the invariant and we get a new request σ_i. σ_i is an edge in G, say (u, v). We have two possibilities: either σ_i is already in S_{i-1} or not.

If $(u, v) \in E(S_{i-1})$ then $S_{i-1} = S_i$ and since h_{i-1} respects the invariant we know that $|h_{i-1}(u) - h_{i-1}(v)| \leq \lambda \cdot \mathrm{Bandwidth}(G)$ and hence we take $h_i = h_{i-1}$.

If on the opposite $(u, v) \notin E(S_{i-1})$ we take $h_i = B(S_i)$, as an embedding of S_i to the line, and reconfigure the network from scratch.

Now, we analyse the cost. We perform adjustments only for a new revealed edge and, thus, there would be no more than $|E(G)|$ reconfigurations. For each reconfiguration we make at most $|V(G)^2|$ migrations, meaning that the total cost of reconfigurations is at most $|E(G)| \cdot |V(G)|^2$. Since we serve the request after performing a reconfiguration and each configuration has a bandwidth of at most $\lambda \cdot \mathrm{Bandwidth}(G)$ we state that we pay no more than $\lambda \cdot \mathrm{Bandwidth}(G) \cdot |\sigma|$ for serving all the requests. □

Lemma 3.2.1. *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* $\mathrm{Bandwidth}(G)$.

Proof. Consider the resulting numeration φ of $V(G)$ done by ON after serving $r \geq 0$ requests. By the definition of bandwidth, there are such $u, v \in V(G)$ that $|\varphi(u) - \varphi(v)| \geq \mathrm{Bandwidth}(G)$. So, we let the next request $\sigma_{ON}[r+1]$ be (u, v) making ON pay at least $\mathrm{Bandwidth}(G)$ for that request. □

Remark 3.2.2. *Using the previous lemma, we can show that λ is a static-optimality factor for the case when $|\sigma| = \Omega(|E(G)| \cdot |V(G)|^2)$.*

3.3. Cycle

Now, we consider a problem where the demand graph is the cycle on n vertices.

Theorem 3.3.1. *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.*

Proof. The idea of the algorithm is to act as in the algorithm described in [7] for the list demand graph until revealed edges do not form a cycle. Once they do we perform a total reconfiguration enumerating nodes of C_n with

$$\begin{cases} i \to 2i - 1, \text{ if } i \leq \lceil \frac{n}{2} \rceil \\ i \to 2(n - i + 1), \text{ otherwise} \end{cases}$$

so, for each pair of adjacent nodes the difference of their numbers is at most 2. The enumeration can be seen on Figure 1.

More formally. Let $G_i = (V, E_i)$ — the demand graph after i requests, and $G_0 = (V, \emptyset)$. We want to maintain the invariant that each G_i is embedded in a way that all adjacent nodes are at a distance of at most 2. Moreover, if there is a line subgraph of G_i then it is embedded as a line, i.e., the embedding preserves edges. We present an algorithm that maintains this invariant by induction.

This invariant holds for G_0. We assume that the invariant holds for G_{i-1} and a new request σ_i arrives. If σ_i is already present in $E(G_{i-1})$ then the invariant holds, we do not reconfigure, and pay at most 2.

If now G_i is a cycle we perform a total reconfiguration with the enumeration with bandwidth 2 described above. For that we pay $O(n^2)$ that is less than $O(n^2 \log n)$, and, thus, our complexity lies inside our bounds. Note that once G_i becomes a cycle we need no further reconfigurations since all the edges are known and the invariant is maintained.

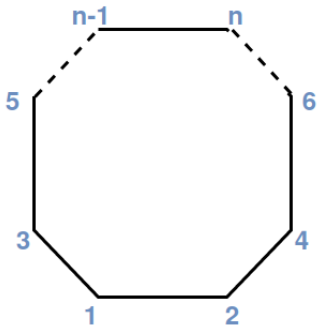


Figure 1: Cycle enumeration with Bandwidth 2

The last case is when σ_i is a new edge and G_i still consists of several connectivity components. We use the algorithm presented in [7]. σ_i connects two

different connectivity components, say L_1 and L_2 forming a new list subgraph L. Suppose that $|V(L_1)| \leq |V(L_2)|$. Our strategy would be to "drag" L_1 towards L_2 that is if $L_1 = \{u_1, \ldots, u_l\}$, $V(L_2) = \{v_1, \ldots, v_k\}$, $\sigma_i = (v_k, u_1)$. By the invariant L_2 is embedded with v_p at $q+p$ for some q and we want nodes of L_1 to be embedded with $u_p \to q + k - 1 + p$. So, we bring each node of L_1 to its position performing required number of swaps. Note, that this reconfiguration brings the embedding that supports the invariant. Now, we analyze the cost of the algorithm processing the requests.

Due to the invariant each request is served with a cost of at most 2. As for the reconfiguration cost: a node can move a distance $\Theta(n)$ during processing one request and it moves no more than $O(\log n)$ times: we either form a cycle or merge two components. Thus, the total cost does not exceed $O(n^2 \log n)$. □

Remark 3.3.2. *Please note that the lower bound with $\Omega(n^2 \log n)$ that was presented for a line graph still holds in the case of a cycle, since the cycle contains the line as the subgraph. Thus, our algorithm is optimal.*

3.4. Ladder

Finally, we consider a case where the demand graph is a ladder.

Theorem 3.4.1. *Suppose a demand graph is a ladder. There is an algorithm such that the total time 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 present all the necessary notion in the next Section, then, in Section 5 we present an overview of the algorithm and, finally, we calculate its cost in Section 6.

As for the cycle, the algorithm matches a lower bound, since the ladder G_n contains L_n as a subgraph.

4. Ladder: notation

Definition 4.1. A line-graph on n vertices is a graph with $V = [1, \ldots, n]$ and $E = \{(i, i+1) \mid i \in [1, \ldots, n-1]\}$.

We refer to the i-th node of a line-graph l as $l[i]$.

Definition 4.2. A ladder or $2 \times n$ grid graph consists of two line-graphs on n vertices l_1 and l_2 with additional edges between the lines: $\{(l_1[i], l_2[i]) \mid i \in [n]\}$.

Further, we denote $2 \times n$ grid as $Grid_n$.

We call the set of two vertices, $\{l_1[i], l_2[i]\}$, the i-th level of the grid and denote it as $level_{Grid_n}(i)$ 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 $v \in V(Grid_n)$ if $v \in level_{Grid_n}(i)$.

We call l_1 and l_2 as the sides of the grid.

Before going into details, we have to note that we will embed the graph not onto $Grid_n$, but onto $Grid_N$, where N is large but does not exceed $2 \cdot n$. This does not affect the bandwidth of the graph (probably, with constant difference) and the map of a graph from the grid onto the line, but significantly simplifies the construction of our embedding.

4.1. Tree embedding

Definition 4.3. Consider some correct embedding φ of a tree T into $Grid_n$. Let $t = \arg\max_{v \in V(T)} level\langle \varphi(v) \rangle$ be the "rightmost" node of the embedding and $b = \arg\min_{v \in V(T)} level\langle \varphi(v) \rangle$ be the "leftmost" node of the embedding. The trunk of T is a path in T connecting b and t. The trunk of a tree T for the embedding φ is denoted with $trunk_\varphi(T)$.

Definition 4.4. Let T be a tree and φ be its correct embedding into $Grid_n$. The level i of $Grid_n$ is called *occupied* if there is a vertex $v \in V(T) : \varphi(v) \in level_{Grid_n}(i)$.

Statement 4.1.1. *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 an useful concept to define since the following holds for it.

Lemma 4.1.1. *Let T be a tree correctly embedded into $Grid_n$ by some embedding φ. Then, all the connectivity components in $T \setminus trunk_\varphi(T)$ are line-graphs.*

Lemma 4.1.2. *For the tree T and any correct embedding φ we know for each node of degree three (except for maximum two of them) if $trunk_\varphi(T)$ passes through it or not.*

Definition 4.5. *Support nodes* are the nodes of two types: either a node of degree three without the neighbours of degree three, or a node that is located on some path between two nodes with degree three. It can be proven that the support nodes appear in the trunk of every correct embedding.

The path through the support nodes is called *trunk core*. We denote this path for a tree T as $trunkCore(T)$.

To give an intuition: the trunk core consists of vertices that lie on a trunk of any embedding.

Definition 4.6. Let T be a tree. All the connectivity components in $T \setminus trunkCore(T)$ are called *simple-graphs* of tree T.

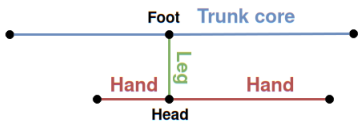
Lemma 4.1.3. *Simple-graphs of a tree T are line-graphs.*

Definition 4.7. 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.



Definition 4.8. A simple-graph connected to some end node of the trunk core is called *exit-graph*.

Definition 4.9. 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 $Grid_N$ rather than $Grid_n$. N should be approximately equal to $2 \cdot n$.

Definition 4.10. An embedding $\varphi : V(G) \to V(Grid_N)$ of a graph G into $Grid_N$ is called *quasi-correct* if:

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

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

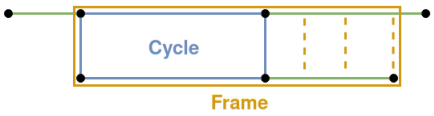
Lemma 4.1.4. *For a graph mapped into the line network with a use of its quasi-correct embedding as described above (level by level) any pair of adjacent nodes are embedded at the distance of at most five.*

4.2. Cycles included

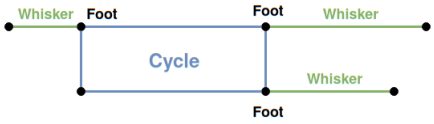
Definition 4.11. A *maximal* cycle C of a graph G is a cycle in G that cannot be enlarged, i.e., there is no other cycle C' in G such that $V(C) \subsetneq V(C')$.

Definition 4.12. Consider a graph G and a maximal cycle C of G. The whisker W of C is a line graph inside G such that:

- $V(W) \neq \emptyset$, and $V(W) \cap V(C) = \emptyset$.



- 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.
- W is maximal, i.e., there is no W' in G such that W' satisfies previous properties and $V(W) \subsetneq V(W')$.



Definition 4.13. Suppose we have a graph G that can be correctly embedded into $Grid_n$ by φ and a cycle C in G. Whiskers W_1 and W_2 of C are called adjacent for the embedding φ if

$$\forall i \in [\min(|V(W_1)|, |V(W_2)|] \ (\varphi(W_1[i]), \varphi(W_2[i])) \in E(Grid_n)$$

Lemma 4.2.1. *Suppose we have a graph G that can be correctly embedded into $Grid_n$ 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 $Grid_N$.*

Definition 4.14. 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 $\{W_1[i], W_2[i] \mid i \in [\min(|V(W_1)|, |V(W_2)|)]\}$ for each pair of adjacent whiskers W_1 and W_2. Adding all the edges $\{(W_1[i], W_2[i]) \mid i \in [\min(|V(W_1)|, |V(W_2)|)]\}$ for each pair of adjacent whiskers W_1 and W_2 makes frame *completed*.

5. Ladder: algorithm

5.1. Static quasi-embedding

We start with one of the basic algorithms — how to quasi-embed on Grid $2 \times N$ with large N any graph that can be embedded in Grid $2 \times n$. The whole algorithm is presented in Appendix 7.1.

5.1.1 Tree embedding

Assume, we are given a tree T that can be embedded into $Grid_n$. Furthermore, there are two special nodes in the tree: one is marked as *R* (for right) and another one is marked as *L* (left). It is known that there exists a correct embedding of T into $Grid_n$ 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 onto $Grid_N$ with *R* being the rightmost node and *L* being the leftmost one while L is mapped to $ImageL$. Moreover, our embedding obeys the following invariant.

Invariant 5.1 (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 onto $Grid_n$. The pseudocode is in Appendix Algorithm 1.

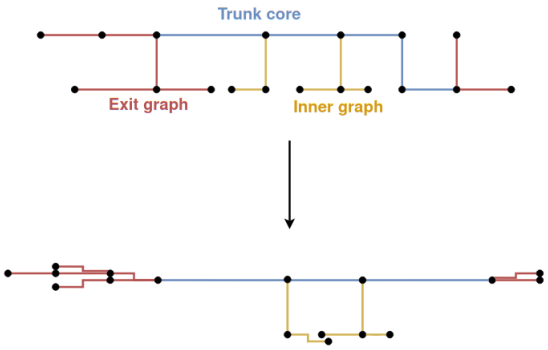


Figure 2: Example of a quasi-correct embedding

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. It can be done simply by following the definition.

Now 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 2.

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 is really simple (straightforwardly) to embed without conflicts.

The pseudocode appears in Appendix Algorithm 2.

5.1.2 Cycle embedding

Now, we are going to talk about how to embed a cycle into $Grid_N$.

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 $Grid_N$ with $level\langle L\rangle \leq level\langle u\rangle \leq level\langle R\rangle \; \forall u \in V(C)$ 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 Appendix Algorithm 3.

Now, suppose that not all information, such as R, L, and $ImageL$, is provided. We will reduce this problem to the case when the missing variables are known. Though the 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. The pseudocode appears in Appendix Algorithm 4

5.1.3 Component embedding

Right now we explain how to embed onto $Grid_N$ a connectivity component S that can be embedded onto $Grid_n$.

We start with an algorithm on how to make a cycle-tree decomposition chain of S assuming no uncompleted 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 glue the results together maintaining the correct order of the chain components. The decomposition pseudocode appears in Appendix Algorithm 5.

Now, 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 uncompleted frames with vertical edges. After this preprocessing, 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 Appendix Algorithm 6.

5.2. Dynamic algorithm

In the previous subsection, we presented an algorithm on how to quasi-embed a static graph. Now, we will explain on how to operate for the requests coming in online manner. The full version of the algorithm is presented in Appendix 7.2.

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 the constant bandwidth. Thus, further, we will 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 we have some edges of a $Grid_n$ already revealed forming connectivity components. After an edge reveal we should reconfigure the target line graph. For that, instead of line reconfiguration we reconfigure our embedding to $Grid_N$ that is then embedded to the line level by level and introduce some constant factor. So, we can consider the reconfiguration only of $Grid_N$ and forget about the target line graph at all. When doing the reconfiguration of an embedding we want to maintain the following invariants:

1. The embedding of any connectivity component is quasi-correct.
2. For each tree in the cycle-tree decomposition its embedding respects Septum invariant 5.1.
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 $Grid_N$.

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

5.2.1 Edge in one component

The pseudocode appears in 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.

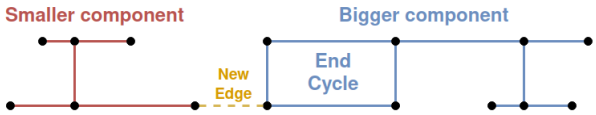
5.2.2 Edge between two components

The pseudocode appears in Appendix Algorithm 9.

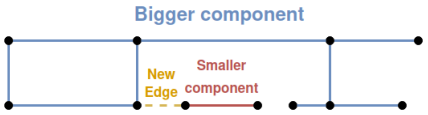
In order to obtain an amortization in the cost, we always "move" the smaller component to the bigger one. Thus, the main question here is how to glue a component to the existing embedding of another component.

The idea is to consider several cases of where the smaller component will be connected to the bigger one. There are three possibilities:

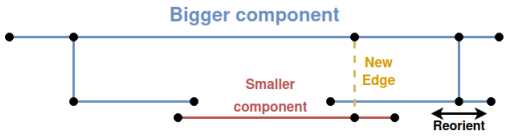
1. *It connects to a cycle node.* In this case there are again two possibilities. Either it "points away" from the bigger component meaning that the cycle to which we connect is the one of the ends in the cycle-tree decomposition of the bigger component. Here, we just simply embed it to the end of the 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 5.1 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.

6. Ladder: cost of the algorithm

Now, we calculate the cost of our dynamic algorithm: how many swaps we should do and how much we should pay for the routing requests itself. Please, remind 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 on the target line network pretty close, by no more than 12 nodes apart.

This bound holds because the nodes are quasi-correctly embedded on $Grid_N$, 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 graph, 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 take our dynamic algorithm and count the total cost of each case 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 become on 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 5.1, thus, paying $O(n^2)$ cost in total — each node can move by at most $O(n)$.

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 $O(\log n)$ time 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 some other modification of two 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 5.1, 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.

Theorem 6.0.1. *We presented an online algorithm for the demand ladder graph of size n with total cost $O(n^2 \log n + |\sigma|)$ for requests σ.*

Remark 6.0.2. *The same result holds for any demand graph that is the subgraph of the ladder of size n.*

7. Full algorithm

7.1. Static quasi-embedding

7.1.1 Tree embedding

Assume, we are given a tree T that can be embedded into $Grid_n$. Furthermore, there are two marked nodes in the tree: one is marked *right* and the *left*. It is known that there is a correct embedding of T with *right* being the right-most node, meaning no node is embedded higher or to the same level, and *left* being the left-most node.

We now describe how to obtain a quasi-correct embedding of T with *right* being the right-most node and *left* being the left-most one and $left$ mapped to $leftImage$. Moreover, this embedding obeys the following invariant:

Invariant 7.1 (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 the $left - right$ path strictly vertically and then we orient line-graphs connected to it in the way that they do not violate septum invariant.

See Algorithm 1.

We now proceed with an embedding of a tree T where $right$, $left$ and $leftImage$ might or might not be given. In the case the variable is not given we denote its value with None.

The idea here would be to first embed the trunk core and its inner line-graphs using Left-Right tree embedding and then to embed exit-graphs strictly vertically "away" from the trunk core. That means that the hands of exit-graphs that are connected to to the right of the trunk core are embedded increasingly and hands of those exit-graphs which are connected to the left of the trunk core are embedded decreasingly.

Algorithm 1 Left-Right tree embedding

procedure RightLeftTreeEmbedding(T, $left$, $right$, $leftImage$)
$P \leftarrow$ path from *left* to *right*
$leftSide \leftarrow side(leftImage)$
$leftLevel \leftarrow level\langle leftImage\rangle$
Embed $P[i] \to level(leftLevel - 1 + i)[leftSide]$
$L \leftarrow$ line-graphs connected to P
$Septa \leftarrow \{level\langle foot(l)\rangle \mid l \in L\} \cup \{level\langle left\rangle, level\langle right\rangle\}$
for $l \in L$ **do**
$i \leftarrow level\langle foot(l)\rangle$
Embed $head(l) \to level(i)[other(leftSide)]$
Orient l ensuring no nodes of $l \setminus \{head(l)\}$ are embedded to any of levels from $Septa$

If a tree does not have a trunk core, that means that its structure is rather simple (in particular it has no more than two nodes of degree three), so we do not care about conflicts.

See Algorithm 2.

Algorithm 2 Tree quasi-correct embedding

procedure TreeQuasiCorrectEmbedding(T, $left$, $right$, $leftImage$)
if $leftImage = \text{None}$ **then**
$leftImage \leftarrow level(1)[1]$
if $(left \neq \text{None}) \wedge (right \neq \text{None})$ **then**
LeftRightTreeEmbedding(T, $left$, $right$, $leftImage$)
return
else if $(left \neq \text{None}) \wedge (right = \text{None})$ **then**
if T has a trunk core **then**
$u, v \leftarrow$ ends of a trunk core
if u between v and $left$ **then**
$trunkRight \leftarrow v$
else
$trunkRight \leftarrow u$
$et \leftarrow$ exit-graphs connected to $trunkRight$
$S' \leftarrow S \setminus et$
LeftRightTreeEmbedding(S', $left$, $trunkRight$, $leftImage$)

for $e \in et$ **do**

$lvl \leftarrow level\langle trunkRight \rangle$

$side \leftarrow side(trunkRight)$

Embed $head(e) \rightarrow level(lvl + 1)[side]$

for $h \in hands(e)$ **do**

for $i \in [length(h)]$ **do**

Embed $h[j] \rightarrow level(lvl + 1 + i)[side]$

else

$right \leftarrow$ arbitrary node of degree 1

LeftRightTreeEmbedding(T, $left$, $right$, $leftImage$)

else if $(left = \text{None}) \wedge (right \neq \text{None})$ **then**

if T has a trunk core **then**

$u, v \leftarrow$ ends of a trunk core

if u between v and $right$ **then**

$trunkLeft \leftarrow v$

else

$trunkLeft \leftarrow u$

$eb \leftarrow$ exit-graphs connected to $trunkLeft$

$S' \leftarrow S \setminus eb$

$leftImageLevel \leftarrow level\langle leftImage \rangle$

$leftImageSide \leftarrow side(leftImage)$

$vShift \leftarrow \max_{e \in eb} \max_{h \in hands(e)} length(h)$

$trunkLeftImage \leftarrow level(leftImageLevel + vShift)[leftImageSide]$

LeftRightTreeEmbedding(S', $left$, $right$, $trunkLeftImage$)

for $e \in eb$ **do**

$lvl \leftarrow level\langle trunkLeft \rangle$

$side \leftarrow side(trunkLeft)$

Embed $head(e) \rightarrow level(lvl - 1)[side]$

for $h \in hands(e)$ **do**

for $i \in [length(h)]$ **do**

Embed $h[j] \rightarrow level(lvl - 1 - i)[side]$

else

$left \leftarrow$ arbitrary node of degree 1

LeftRightTreeEmbedding(T, $left$, $right$, $leftImage$)

else

if T has a trunk core **then**

$trunkRight,\ trunkLeft \leftarrow$ ends of a trunk core

$et \leftarrow$ exit-graphs connected to $trunkRight$

$eb \leftarrow$ exit-graphs connected to $trunkLeft$

$S' \leftarrow S \setminus et \setminus eb$

$leftImageLevel \leftarrow level\langle leftImage\rangle$

$leftImageSide \leftarrow side(leftImage)$

$vShift \leftarrow \max_{e \in eb} \max_{h \in hands(e)} length(h)$

$trunkLeftImage \leftarrow level(leftImageLevel + vShift)[leftImageSide]$

LeftRightTreeEmbedding(S', $left$, $right$, $trunkLeftImage$)

for $e \in et$ **do**

$lvl \leftarrow level\langle trunkRight\rangle$

$side \leftarrow side(trunkRight)$

Embed $head(e) \rightarrow level(lvl + 1)[side]$

for $h \in hands(e)$ **do**

for $i \in [length(h)]$ **do**

Embed $h[j] \rightarrow level(lvl + 1 + i)[side]$

for $e \in eb$ **do**

$lvl \leftarrow level\langle trunkLeft\rangle$

$side \leftarrow side(trunkLeft)$

Embed $head(e) \rightarrow level(lvl - 1)[side]$

for $h \in hands(e)$ **do**

for $i \in [length(h)]$ **do**

Embed $h[j] \rightarrow level(lvl - 1 - i)[side]$

else

$right, left \leftarrow$ furthest nodes of degree 1

$P \leftarrow$ path from $left$ to $right$

Embed P strictly monotone placing $left$ to $leftImage$

if there is a line-graph left **then**

$l \leftarrow$ left line-graph

$(u, v) \leftarrow (u, v) \in E(T)$ s.t. $u \in P, v \in l$

Embed v to the same level, opposite side to u.

Embed l to the opposite side to the side of P preserving connectivity and correctness

7.1.2 Cycle embedding

Given a cycle C of length ≥ 6 and nodes $left, right \in V(C)$ we construct a correct embedding of C into $Grid_\infty$ with $level\langle left\rangle \leq level\langle u\rangle \leq level\langle right\rangle \; \forall u \in V(C)$ and $left$ placed to $leftImage$.

For convenience we assume that for every node v there is a local consecutive numeration starting at v. The number of node $u \in V(C)$ in this numeration is referenced with $number_v(u)$. The node with number i in local numeration of v is referenced with $C_v[i]$.

We first check if it is possible to satisfy the given constraints of placing the $left$ node to left and a $right$ node to the right. If it is indeed possible, we place $left$ to desired place and then choose an orientation (clockwise or counterclockwise) following which we would embed the rest of the nodes, keeping in mind that $right$ must stay on the highest level. See Algorithm 3.

Now, suppose that not all information, such as $right$, $left$, and $LeftImage$, is provided. We will reduce this problem to the case when the missing variables are known. Though the subtlety might occur due to the fact that there are inner edges in the cycle. In this case we choose missing $left/right$ more precisely in order to embed inner edge vertically. See Algorithm 4.

7.1.3 Component embedding

Right now we explain on how to embed onto $Grid_N$ a connectivity component S that can be embedded onto $Grid_n$.

We start with an algorithm on how to make a cycle-tree decomposition chain of S assuming no uncompleted 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

Algorithm 3 Left-Right cycle embedding

procedure LeftRightCycleEmbedding(C, $left$, $right$, $leftImage$)
$h \leftarrow \frac{length(C)}{2}$
Ensure: $number_{left}(right) \in \{h, h+1, h+2\}$
$leftLevel \leftarrow level\langle leftImage\rangle$
$leftSide \leftarrow side(leftImage)$
if $(number_{left}(right) = h) \vee (number_{left}(right) = h+1)$ **then**
for $i \in [h]$ **do**
Embed $C_{left}[i] \to level(leftLevel + i - 1)[leftSide]$
for $i \in [h]$ **do**
Embed $C_{left}[h+i] \to level(leftLevel + h - i)[other(leftSide)]$
else
Embed $left \to leftImage$
for $i \in [h]$ **do**
Embed $C_{left}[i+1] \to level(leftLevel + i - 1)[other(leftSide)]$
for $i \in [h-1]$ **do**
Embed $C_{left}[h+1+i] \to level(leftLevel + h - i)[leftSide]$

glue the results together maintaining the correct order of the chain components. See the Algorithm 5.

Now, 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 uncompleted frames with vertical edges. After this preprocessing, 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.

As before, we have additional variables $left$, $right$ and $leftImage$ which might or might not be given.

Algorithm 6 Connectivity component quasi-correct embedding

procedure Preprocess(S)
$C \leftarrow$ maximal cycles of length 4 in S
for $c \in C$ **do**
remove arbitrary edge of c from S
$F \leftarrow$ cycle frames in S
for $f \in F$ **do**

Algorithm 4 Cycle embedding

procedure CycleEmbedding(C, $left$, $right$, $leftImage$)
$h \leftarrow \frac{length(C)}{2}$
if $(left \neq \text{None}) \wedge (right \neq \text{None})$ **then**
Ensure: $number_{left}(right) \in \{h, h+1, h+2\}$
if $leftImage = \text{None}$ **then**
$leftImage \leftarrow level(1)[1]$
if $(left = \text{None}) \wedge (right = \text{None})$ **then**
$left \leftarrow$ arbitrary node of C
if C has an inner edge **then**
Choose $right$ out of $\{C_{left}[h], C_{left}[h+2]\}$ to respect the inner edge
else
Choose $right$ out of $\{C_{left}[h], C_{left}[h+2]\}$ arbitrary
else if $(left \neq \text{None}) \wedge (right = \text{None})$ **then**
if C has an inner edge **then**
Choose $right$ out of $\{C_{left}[h], C_{left}[h+2]\}$ to respect the inner edge
else
Choose $right$ out of $\{C_{left}[h], C_{left}[h+2]\}$ arbitrary
else if $(left = \text{None}) \wedge (right \neq \text{None})$ **then**
if C has an inner edge **then**
Choose $left$ out of $\{C_{left}[h], C_{left}[h+2]\}$ to respect the inner edge
else
Choose $left$ out of $\{C_{left}[h], C_{left}[h+2]\}$ arbitrary
LeftRightCycleEmbedding(C, $left$, $right$, $leftImage$)

Algorithm 5 Cycle-Tree decomposition chain

function CycleTreeDecompositionChain(S)
Ensure: S has no uncompleted frames
  if S is empty **then**
    return []
  else if S is a tree **then**
    return $[S]$
  else
    $C \leftarrow$ arbitrary maximal cycle in S
    $S_1, S_2 \leftarrow$ connectivity components of $S \setminus C$
    $C_1 \leftarrow CycleTreeDecompositionChain(S_1)$
    $C_2 \leftarrow CycleTreeDecompositionChain(S_2)$
    if C_2 is empty **then**
      return $[C] + C_1$
    else
      if $\exists u \in V(C_2[0]),\ v \in V(C),\ s.t.\ (u, v) \in E(S)$ **then**
        return $C_1 + [S] + C_2$
      else
        return $C_2 + [S] + C_1$

complete F

procedure ComponentEmbeddingLeftFixed(S, $left$, $right$, $leftImage$)
  if S is a tree **then**
    TreeQuasiCorrectEmbedding(S, $left$, $right$, $leftImage$)
    return
  if S is a cycle **then**
    CycleEmbedding(S, $left$, $right$, $leftImage$)
    return
  Preprocess(S)
  $C \leftarrow CycleTreeDecompositionChain(S)$
  if $left \neq$ None **then**
    Reverse C in the way that $left \in C[1]$
  if $right \neq$ None **then**
    Reverse C in the way that $right \in C[length(C)]$
  for $i \in [length(C)]$ **do**
    if $i = 1$ **then** $u, v \leftarrow (u, v) \in E(S)$, s.t. $(u \in V(C[1])) \wedge (v \in$

$V(C[i+1])$

if $C[1]$ is a tree **then**

$cur \leftarrow C[1] \cup (u, v)$

TreeQuasiCorrectEmbedding(cur, $left$, v, $leftImage$)

else

CycleEmbedding($C[1]$, $left$, u, $leftImage$)

else if $i = length(C)$ **then**

$u, v \leftarrow (u, v) \in E(S)$, s.t. $(u \in V(C[i-1])) \wedge (v \in V(C[i])$

$leftLevel \leftarrow level\langle u \rangle + 1$

$leftSide \leftarrow side(u)$

$localLeftImage \leftarrow level(leftLevel)[leftSide]$

if $C[i]$ is a cycle **then**

CycleEmbedding($C[i]$, v, $right$, $localLeftImage$)

else

TreeQuasiCorrcetEmbedding($C[i]$, v, $right$, $localLeftImage$)

else

$u_1, v_1 \leftarrow (u, v) \in E(S)$, s.t. $(u \in V(C[i-1])) \wedge (v \in V(C[i])$

$leftLevel \leftarrow level\langle u_1 \rangle + 1$

$leftSide \leftarrow side(u_1)$

$localLeftImage \leftarrow level(leftLevel)[leftSide]$

$u_2, v_2 \leftarrow (u, v) \in E(S)$, s.t. $(u \in V(C[i])) \wedge (v \in V(C[i+1])$

if $C[i]$ is a cycle **then**

CycleEmbedding($C[i]$, v_1, u_2, $localLeftImage$)

else

$cur \leftarrow C[i] \cup (u_2, v_2)$

TreeQuasiCorrcetEmbedding($C[i]$, v_1, v_2, $localLeftImage$)

We finally notice that having a procedure to embed a component with a fixed $leftImage$ it is easy to obtain a procedure which embeds with $rightImage$ fixed. We simply apply the "$left$" procedure and then flip the result.

Algorithm 7 Component embedding right fixed

procedure ComponentEmbeddingRightFixed(S, $left$, $right$, $rightImage$)
ComponentEmbeddingLeftFixed(S, $right$, $left$, $rightImage$)
Flip the image of S over horizontal axis maintaining the position of $right$

7.2. Dynamic algorithm

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

1. The embedding of any connectivity component is quasi-correct.
2. For each tree in the cycle-tree decomposition its embedding respects Septum invariant 5.1.
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., two nodes of a cycle do not map to same node of $Grid_N$.

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

7.2.1 Edge in one component

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.

Algorithm 8 Process Edge In One Component

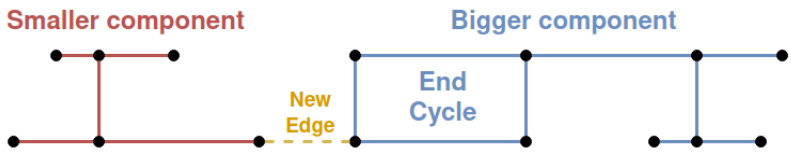
procedure ProcessEdgeOneComponent(S, (u, v))
  if Edge (u, v) already exists **then**
    return
  if Edge (u, v) forms a maximal cycle of length 4 **then**
    return
  $C \leftarrow$ maximal cycle containing u, v
  $F \leftarrow$ completed frame of C
  $S_1, S_2 \leftarrow$ connectivity components of $S \setminus F$
  $u_1, v_1 \leftarrow u, v : \ u \in V(F), v \in V(S_1), (u, v) \in E(S)$
  $u_2, v_2 \leftarrow u, v : \ u \in V(F), v \in V(S_2), (u, v) \in E(S)$
  if $level\langle u_1 \rangle > level\langle u_2 \rangle$ **then**
    $Swap(S_1, S_2),\ Swap(u_1, u_2),\ Swap(v_1, v_2)$
  CycleEmbedding(F, u_1, u_2, None)
  ComponentEmbeddingTopFixed(S_1, None, u_1, $image(u_1)$)
  ComponentEmbeddingBotFixed(S_2, u_2, None, $image(u_2)$)

7.2.2 Edge between two components

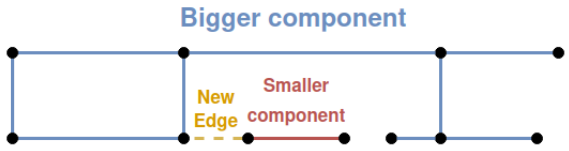
In order to obtain an amortization in the cost, we always "move" the smaller component to the bigger one. Thus, the main question here is how to glue a component to the existing embedding of another component.

The idea is to consider several cases of where the smaller component will be connected to the bigger one. There are three possibilities:

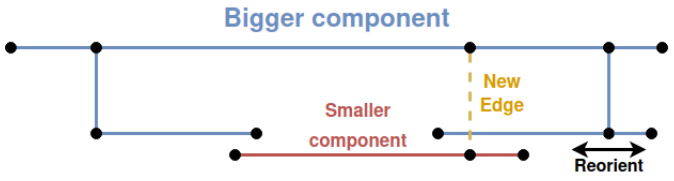
1. *It connects to a cycle node.* In this case there are again two possibilities. Either it "points away" from the bigger component meaning that the cycle to which we connect is the one of the ends in the cycle-tree decomposition of the bigger component. Here, we just simply embed it to the end of the 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 5.1 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.

Algorithm 9 Process edge between two components

procedure AddInnerWhisker(S, C, W, (u, v))

$S \leftarrow S \cup W \cup (u, v)$

$F \leftarrow$ completed frame of C

$S_1, S_2 \leftarrow$ connectivity components of $S \setminus F$

if S_1 is embedded above S_2 **then**

$Swap(S_1, S_2)$

$s_1, t_1 \leftarrow s, t : \ s \in V(F), t \in V(S_1), (s, t) \in E(S)$

$s_2, t_2 \leftarrow s, t : \ s \in V(F), t \in V(S_2), (s, t) \in E(S)$

CycleEmbedding(F, s_1, s_2, None)

ComponentEmbeddingTopFixed($S_1 \cup (s_1, t_1)$, None, s_1, $image(s_1)$)

ComponentEmbeddingBotFixed($S_2 \cup (s_2, t_2)$, s_2, None, $image(s_2)$)

procedure ProcessEdgeTwoComponents(S_1, S_2, (u, v))

Ensure: $u \in V(S_1), v \in V(S_2)$

if $V(S_1) < V(S_2)$ **then**

$Swap(S_1, S_2), \ Swap(u, v)$

$DC_1 \leftarrow CycleTreeDecomposition(S_1)$

Reverse DC_1 in a way $DC[i]$ is embeded under $DC[i + 1] \ \forall i$

$A, i \leftarrow DC_1[i], i : \ u \in V(DC_1[i])$

if A is a cycle **then**

if $length(DC_1) = 1$ **then**

if A has an inner edge **then**

if u is a top node **then**

$bot \leftarrow$ arbitrary bottom node of A

ComponentEmbeddingBotFixed($S_1 \cup S_2 \cup (u, v)$, bot, None, None)

else

$top \leftarrow$ arbitrary top node of A

ComponentEmbeddingTopFixed($S_1 \cup S_2 \cup (u, v)$, None, top, None)

else

ComponentEmbeddingBotFixed($S_1 \cup S_2 \cup (u, v)$, None, None, None)

else if $i = 1$ **then**

$p, q \leftarrow p, q : \ p \in V(DC_1[i]), q \in V(DC_1[i+1]), (p, q) \in E(S_1)$

if $(u, p) \in E(S_1)$ **then** AddInnerWhisker(S_1, A, S_2, (u, v))

else

if u is not a bottom node of A **then**

Flip A over diagonal containing p

ComponnetEmbeddingTopFixed(S_2, None, u, $image(u)$)

else if $i = length(DC_1)$ **then**

$p, q \leftarrow p, q : \ p \in V(DC_1[i]), q \in V(DC_1[i-1]), (p, q) \in E(S_1)$

if $(u, p) \in E(S_1)$ **then** AddInnerWhisker(S_1, A, S_2, (u, v))

else

if u is not a top node of A **then**

Flip A over diagonal containing p

ComponnetEmbeddingBotFixed(S_2, u, None, $image(u)$)

elseAddInnerWhisker(S_1, A, S_2, (u, v))

if A is a tree **then**

if $u \in$ extended trunk core of A **then**

Embed $v \rightarrow opposite(u)$

$l_1, l_2 \leftarrow u$ neighbouring inner simple-graphs

Orient S_2, l_1, l_2 to maintain Septum invariant in A

else if $u \in$ inner simple-graph **then**

$S_1 \leftarrow S_1 \cup S_2 \cup (u, v)$

$l \leftarrow$ inner simple graph containing u

Orient l to maintain Septum invariant in A

else if $u \in$ exit-graph **then**

if $i = 1$ **then**

if $length(DC_1) = 1$ **then**

$p \leftarrow$ arbitrary highest node of S_1

$q \leftarrow$ additional temporary node

else

$p, q \leftarrow p, q :\ p \in V(DC_1[i]), q \in V(DC_1[i+1]), (p, q) \in E(S_1)$

ComponentEmbeddingTopFixed($A \cup S_2 \cup (p, q)$, None, q, $image(q)$)

else if $i = length(DC_1)$ **then**

$p, q \leftarrow p, q :\ p \in V(DC_1[i]), q \in V(DC_1[i-1]), (p, q) \in E(S_1)$

ComponentEmbeddingBotFixed($A \cup S_2 \cup (p, q)$, q, None, $image(q)$)

---

8. Proofs and Analysis

8.1. Strategy

At the very beginning there are no requests and we don't know any request-edges. Requests come one at a time, possibly, revealing new edges. Known edges form connectivity components which are all subgraphs of the request graph. Our strategy would be to maintain such enumeration σ of vertices that for each connectivity component S

$$\max_{(u,v) \in E(S)} |\sigma(u) - \sigma(v)| \leq 12$$

We call this property of an enumeration the *proximity property*.

So, if we receive the request which was already known, we do nothing since the property persists. But if the new edge comes, we might perform a re-enumeration σ on the vertices to maintain the property.

8.2. Bandwidth of subgraphs

Observation 8.2.1. *Bandwidth($Grid_n$) = 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. □

Definition 8.1. Consider two connected graphs S and G. The correct embedding of S into G is a mapping $\varphi : V(S) \to V(G)$ such that:

- φ is injective

- $(u, v) \in E(S) \rightarrow (\varphi(u), \varphi(v)) \in E(G)$

If φ is not injective, i.e. there are nodes u, v, s.t. $\varphi(u) = \varphi(v)$, we say that there is a conflict between u and v.

Lemma 8.2.1. *If there is a correct embedding of S into G then $bandwidth(S) \leq bandwidth(G)$.*

Proof. Let φ be a correct embedding of S into G. And let σ be the enumeration on G with which the $bandwidth(G)$ is achieved.

Let U be the finite set of unique natural numbers. For $v \in U$ we define $ord_U(v) = |\{u \mid u \in U, u \leq v\}|$.

We now define the enumeration σ_S of S as follows:

$$U = \{\sigma(\varphi(v)) \mid v \in S\}$$
$$\sigma_S(v) = ord_U(\sigma(\varphi(v)))$$

We state that $\max\limits_{(u,v)\in E(S)} |\sigma_S(u) - \sigma_S(v)| \leq bandwidth(G)$. This follows from two facts:

- For every $(u, v) \in E(S)$

  $$|\sigma(\varphi(u)) - \sigma(\varphi(v))| \leq bandwidth(G)$$

  since $(\varphi(u), \varphi(v)) \in E(G)$

- If U is a set of unique natural numbers than for every $u, v \in U$

  $$|ord_U(u) - ord_U(v)| \leq |u - v|$$

Then, for each edge $(u, v) \in E(S)$ we get the following inequalities:

$$|\sigma_S(u) - \sigma_S(v)| = |ord_U(\sigma(\varphi(u))) - ord_U(\sigma(\varphi(v)))| \leq |\sigma(\varphi(u)) - \sigma(\varphi(v))| \leq bandw$$

□

We know that all the graphs that appear during the requests processing (revealing the edges) are subgraphs of $Grid_n$. Thus, by Lemma 8.2.1 we conclude that their $bandwidth \leq 2$. And we can use the embedding function from this Lemma to enumerate each subgraph S of $Grid_n$ with σ_S.

Remark 8.2.1. *We do not need to worry about the top and bottom bounds of $Grid_n$ when performing an embedding. In fact, we can perform an embedding of S into the $Grid_\infty$ and, since S is connected and the embedding preserves connectivity, the whole image of S will be within some $Grid_m$ (for $m \geq n$) which is enough to obtain a requested $bandwidth \leq 2$.*

8.3. Connectivity component structure

Requests come with time possibly revealing new edges of a request graph and forming connectivity components which are subgraphs of the request graph.

One connectivity component can be decomposed into cycles and trees. Let us now provide some statements about tree and cycle embedding.

Tree

Definition 8.2. Consider some correct embedding φ of a tree T into $Grid_n$. Let $t = \arg\max_{v \in V(T)} level\langle\varphi(v)\rangle$ be the "topmost" node of the embedding and $b = \arg\min_{v \in V(T)} level\langle\varphi(v)\rangle$ be the "bottommost" node of the embedding. The trunk of T is a path in T connecting b and t. The trunk of a tree T for the embedding φ is denoted with $trunk_\varphi(T)$.

Definition 8.3. Let T be a tree, φ be a correct embedding of T into $Grid_n$. The level i of $Grid_n$ is called *occupied* if there is a vertex $v \in V(T) :\ \varphi(v) \in level_{Grid_n}(i)$.

Statement 8.3.1. *For every occupied level i there is $v \in trunk_\varphi(T)$ such that $v \in level(i)$.*

Proof. By the definition 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 is an useful concept to define since the following holds for it.

Lemma 8.3.1. *Let T be a tree correctly embedded into $Grid_n$ by some embedding φ. Then all the connectivity components in $T \setminus trunk_\varphi(T)$ are line-graphs.*

Proof. Suppose that it is not true and then there should exist a subgraph S of T such that $V(S) \cap V(trunk_\varphi(T)) = \emptyset$ and S contains a node of degree three. Since there is a node of degree three in S we can state that there are two nodes of S, say u and v with the same level ($level\langle\varphi(v)\rangle = level\langle\varphi(u)\rangle$). But the image of the tree trunk passes through all occupied levels of the grid by Statement 8.3.1. Hence, either u or $v \in trunk_\varphi(T)$ which contradicts the assumption. □

The bad thing about the trunk is that it depends on the embedding. And there can be several correct embeddings of the same tree giving different trunks. So, we introduce the concept of a *trunk core* which alleviates this issue. But at first, we prove some technical statements.

Statement 8.3.2. *For the tree T, disregarding the correct embedding φ, the $trunk_\varphi(T)$ must pass through a node of degree three if it has no neighbours of degree three. If there are two adjacent nodes with degree three, the trunk must pass through at least one of them.*

Proof. First, consider the case of a node with no neighbours of degree three. Let's call it a. To prove by contradiction we assume that the trunk does not pass through a. Let's call a-s neighbours b, c and d. W.l.o.g assume that

$$\begin{aligned} \varphi(a) &= level(i)[1] \\ \varphi(b) &= level(i-1)[1] \\ \varphi(c) &= level(i)[2] \\ \varphi(d) &= level(i+1)[1] \end{aligned}$$

Since the trunk does not pass through a and by Statement 8.3.1 it passes through the level i it should pass through c. If c has degree one, the trunk contains one node from level i, and does not contain any node from $i+1$, thus, this trunk cannot contain the topmost node. If c has degree two, we say that its second neighbour is mapped to $level(i-1)[2]$. The case when it is mapped to

$level(i+1)[2]$ is symmetric. But then trunk does not pass through the $i+1$ level which contradicts the Statement 8.3.1.

Now, coming to the case with two adjacent nodes of degree three, we have two adjacent nodes a and b of degree three. And let c, d be the rest neighbours of a and e, f be the rest neighbours of b. If a and b are embedded to the same level, then by Statement 8.3.1 the trunk passes through at least one of them. Suppose now that a and b are on different levels, say

$$\begin{aligned}\varphi(a) &= level(i)[1]\\ \varphi(b) &= level(i+1)[1]\\ \varphi(c) &= level(i)[2]\\ \varphi(d) &= level(i-1)[2]\\ \varphi(e) &= level(i+1)[2]\\ \varphi(f) &= level(i+2)[1]\end{aligned}$$

Since the edge (a, b) is a bridge between two connected components of a tree and the trunk contains nodes form both components the trunk should pass through the edge (a, b), so it passes through both a and b.

□

Lemma 8.3.2. *For the tree T and any correct embedding φ we know for each node of degree three (except maximum two) if the $trunk_\varphi(T)$ passes through it or not.*

Proof. We call a pair of adjacent nodes of degree three *"paired"* nodes. We call a node of degree three with no neighbours of degree three *"single"*.

If the tree contains not more than two nodes of degree three, the statement is trivial. So, we suppose that there exist at least three nodes of degree three.

The trunk passes through the single nodes by Statement 8.3.2. Thus we are interested in paired nodes. Consider such pair. Let's call its nodes a and b. By the Statement 8.3.2 we know that either a or b is in the trunk.

By the assumption there exist either another single node or other paired nodes. If there is a single node, let's call it c, we know that it is in the trunk. c is reachable from a and b and since we have tree either a is on path from b to c or b is on path from a to c. W.l.o.g. assume b is on a path from a to c. But this

implies that b is in the trunk, because if not, a is, and, thus, there are two paths from a to c — the trunk and the one containing b. Thus, we have a cycle, which is impossible since we have a tree.

If there are no single nodes, there are paired nodes. We denote them with u and v. u and v are reachable from a, thus, w.l.o.g. we can assume that u is on the path from a to v. If now b is on the path from a to u, we have the following: $a \to b \rightsquigarrow u \to v$. By Statement 8.3.2 we know that the trunk must pass through either u or v. Denote the one the trunk passes through with c. We can reduce this case to the previous one, if we take any $c = u$ or $c = v$. Applying the same reasoning we deduce that b must be in the trunk.

Support nodes are the nodes of two types: either it is a single node, or it is a node that is located on the path between two other nodes with degree three. This Lemma shows that the support nodes appear in the trunk of every correct embedding.

We make a path P through support nodes. For any inner node of this path which is paired there is no chance for its pair to be in a trunk if it is not in P already, because the trunk is a path. So, the only uncertainty remains about at most one node in the pairs of end nodes. □

Definition 8.4. Path P constructed in Lemma 8.3.2 is called *trunk core*. We denote this path for a tree T as $trunkCore(T)$. Note that it can be embedded into $Grid_n$.

Definition 8.5. The embedding φ of a line-graph l on the grid is called monotone if the nodes $\varphi(l[i])$ and $\varphi(l[j])$ are on the same level of the grid only when they are adjacent on T.

Lemma 8.3.3. *If a line graph is embedded preserving edges into $Grid_n$ with no self-intersections non-monotonically then one of the end-points shares a level with a node of a path it is not adjacent with.*

Proof. Denote a line graph l. Let's say i is the smallest index such that $l[i]$ shares level with some other non-adjacent node $l[j]$, $|i - j| > 1$. W.l.o.g. let's assume that $l[i]$ is embedded to $level(k)[1]$. Since i was chosen the smallest $j > i$. Let us assume that $l[i-1]$ is embedded to $level(k-1)[1]$. Then, since $l[j]$ is embedded to $level(k)[2]$, $l[i+1]$ is embedded into $level(k+1)[1]$. We also state that $l[j-1]$

is embedded to $level(k+1)[2]$, since if it is embedded into $level(k-1)[2]$, the path should go from $l[i+1]$ to $l[j-1]$ (note that $i+1 < j-1$) without passing through level k which is impossible. So we have the following embeddings:

$$\begin{aligned} l[i] &\to level(k)[1] \\ l[i-1] &\to level(k-1)[1] \\ l[j] &\to level(k)[2] \\ l[j+1] &\to level(k-1)[2] \end{aligned}$$

It is easy to see that $l[j+2]$ has no other options but to be embedded to $level(k-2)[2]$. But then $l[i-3]$ should be embedded to $level(k-3)[1]$ and so on $l[j+t] \to level(k-t)[2]$ and $l[i-t] \to level(k-t)[2]$ in general. We now take $t = \min(i-1, length(p)-j)$ so either $l[i-t]$ or $l[j+t]$ is an end nodes and they both exist. They share level, so the lemma is proved. □

Lemma 8.3.4. *The trunk core of a tree T is always embedded in the monotone manner.*

Proof. Trunk core connects nodes of degree three which cannot be embedded with any other nodes of the trunk to the same level since then either a cycle appears or we obtain a conflict. Thus, by the Lemma 8.3.3 trunk core must be embedded monotonically. □

From now on we assume that every mentioned tree can be embedded into the $Grid_n$.

Definition 8.6. Let T be a tree. All the connectivity components in $T \setminus trunkCore(T)$ are called *simple-graphs* of tree T.

Lemma 8.3.5. *Simple-graphs of a tree T are line-graphs.*

Proof. Note that all the nodes of degree three in T are either in the trunk core or they are adjacent to the trunk core. Hence after removing the nodes of the trunk core no nodes of degree three are left and, thus, all the graphs left are line-graphs. □

Definition 8.7. The edge via between a simple-graph and the trunk core is called a *leg*.
The end of the leg in the simple-graph is called a *head* of the corresponding simple-graph.
The end of the leg in the trunk core is called a *foot* of the corresponding simple-graph.
If you remove the head of the simple-graph and it falls apart into two connectivity connectivity, such simple-graph is called two-handed and those parts are called its hands. Otherwise, one connectivity component remains and it is called a hand of the simple-graph. The simple-graph in this case is named called one-handed. If there are no nodes in the simple-graph but just a head it is called zero-handed.

Definition 8.8. A simple-graph connected to the end nodes of the trunk core is called *exit-graph*.

Definition 8.9. A simple-graphs connected to the inner nodes of the trunk core is called *inner-graph*.

Definition 8.10. An embedding $\varphi : V(T) \to V(Grid_n)$ of a tree T into $Grid_n$ is called *quasi-correct* if:

- $(u, v) \in E(T) \to (\varphi(u), \varphi(v)) \in E(Grid_n)$
- There are no more than **three** nodes mapped into each level of $Grid_n$

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

Lemma 8.3.6. *For a graph embedded into line with a use of its quasi-correct embedding as described above any adjacent nodes are embedded at the distance of at most 5.*

Proof. Since two adjacent vertices are embedded to the levels with a number difference of at most 1, we can state that there are no more than 4 nodes between them in the line, since there are no more than 3 nodes per level. □

8.3.1 Tree embedding strategy

We start with the discussion on how to embed a tree with $|V(trunkCore(T))| \leq 1$. Such tree can have ≤ 3 nodes of degree three, since, otherwise, there are at least four nodes of degree three and the trunk core has at least two nodes:

- ≥ 2 single nodes. In this case, they are both in the trunk core.
- at least one single node and at least one paired nodes. In this case, one node from a pair and a single node are in the trunk core.
- at least two disjoint paired nodes. In this case, for each pair we know for certain the member who is in the trunk, thus we again have at least two nodes in the trunk core.

Further, we analyse the cases depending on the number of nodes of degree three. We need the following technical Lemma.

Lemma 8.3.7. *If there is a tree with three nodes of degree three a, b, and c and there are edges (a, b) and (b, c), then the third neighbour of b is of degree one and for any correct embedding a, b, and c are embedded to the different levels.*

Proof. Consider a correct embedding φ. Say $\varphi(b) = level(i)[1]$. If now $\varphi(a) = level(i)[2]$, both $level(i+1)[2]$ and $level(i-1)[2]$ are occupied by neighbours of a, so no matter where we embed c, say to $level(i+1)[1]$ there would be only one spare slot, $level(i+2)[1]$ in this case, for two neighbours of c. Recall that we have a tree so a and c can't share more then one neighbour.

So the only possible embedding up to the symmetry is

$$\begin{cases} \varphi(b) = level(i)[1] \\ \varphi(a) = level(i-1)[1] \\ \varphi(c) = level(i+1)[1] \end{cases}$$

In this case $level(i \pm 1)[2]$ are occupied by neighbours of a and c, so the third neighbour of b cannot have any neighbours except for b since there is no place for them.

□

Now, we consider the possible cases for the amount of nodes with degree three.

- There are no nodes of degree three. In this case our tree is just a line-graph l and we embed it the following way:

$$\begin{aligned} \varphi : V(l) &\to Grid_n \\ \varphi(l[i]) &= level(i)[1] \end{aligned}$$

  Remember that right now we allow to choose any n, arbitrary large.

- There is one node of degree three. We can think of it as two line graphs l_1 and l_2 with additional edge $(l_1[i], l_2[1])$ for some i. We embed the tree in the following way:

$$\begin{aligned} \varphi : V(l_1) \cup V(l_2) &\to Grid_n \\ \varphi(l_1[j]) &= level(j)[1] \\ \varphi(l_2[j]) &= level(i + j - 1)[2] \end{aligned}$$

- There are two nodes of degree three. Since $|V(trunkCore(T)| \leq 1$, we conclude that those two nodes are paired, since otherwise they would be single nodes and therefore be in the trunk core by Lemma 8.3.2. So, in this case we can present T as two line-graphs l_1 and l_2 with additional edge $(l_1[i], l_2[j])$ for some i and j. We embed T in the following way:

$$\begin{aligned} \varphi : V(l_1) \cup V(l_2) &\to Grid_n \\ \varphi(l_1[k]) &= level(k)[1] \\ \varphi(l_2[k]) &= level(i + k - j)[2] \end{aligned}$$

- There are three nodes of degree three. Since $|V(trunkCore(T)| \leq 1$, we conclude that there is no single node, otherwise, it is in the trunk core and one of the other two is also in the trunk core, contradicting the assumption.

  So, with our three nodes of degree three, say a, b and c we must have edges (a, b) and (b, c). By the Lemma 8.3.7 none of a, b, c can be embedded into the same level, and the third neighbour of b is of degree one. Denote the line-graphs connected to a with l_a^1 and l_a^2 (they are line-graphs since we have only three nodes of degree three) and the line-graphs connected to c with l_c^1 and l_c^2. Denote the third neighbour of b with d. We embed as follows:

$$\begin{aligned}
\varphi(b) &= level(2)[1] \\
\varphi(d) &= level(2)[2] \\
\varphi(a) &= level(1)[1] \\
\varphi(c) &= level(3)[1] \\
\varphi(l_a^1[i]) &= level(1-i)[1] \\
\varphi(l_a^2[i]) &= level(2-i)[2] \\
\varphi(l_c^1[i]) &= level(3+i)[1] \\
\varphi(l_c^2[i]) &= level(2+i)[2]
\end{aligned}$$

- There are no other cases, since we showed that if there are four nodes with degree three then the size of the trunk core should be bigger than one.

Now, we discuss how to embed a more generic tree with $|V(trunkCore(T))| \geq 2$ into the grid. We call our embedding as $\tilde{\varphi}$.

1. $\tilde{\varphi}(trunkCore(T)[i]) = level(i)[1]$

2. Suppose l is a simple-graph connected to the inner node with number i of the trunk core by its j-th node, so the leg of l is $(trunkCore[i], l[j])$. We embed $l[j]$ to the opposite of $trunkCore[i]$, i.e. $\tilde{\varphi}(l[j]) = level(i)[2]$.

   We also want to reserve nodes $level(|V(trunkCore(T))|)[2]$ and $level(1)[2]$ for exit-graphs, so we say we embed phantom nodes there for algorithm not to use them on Step 3.

3. We now want to embed hands of simple-graphs connected to the inner trunk core nodes. Suppose we have such simple graph l and we've embedded its head to $level(i)[2]$ on Step 2.

   If l is zero-handed, it is already embedded on Step 2.

   If l is two-handed, denote its hands with h_1 and h_2 and choose the one of embeddings from

$$\left[\begin{array}{l} \begin{cases} \tilde{\varphi}(h_1[j]) = level(i+j)[2] \\ \tilde{\varphi}(h_2[j]) = level(i-j)[2] \end{cases} \\ \begin{cases} \tilde{\varphi}(h_1[j]) = level(i-j)[2] \\ \tilde{\varphi}(h_2[j]) = level(i+j)[2] \end{cases} \end{array}\right.$$

   which does not map nodes from $V(h_1) \cup V(h_2)$ to the place nodes were mapped to on step 2.

   If l is one-handed, denote its hand with h and consider two cases:

   - $$\begin{cases} trunkCore(T)[i+1] \text{ is an inner node and it is a foot of another} \\ \qquad\qquad\qquad\qquad \text{one-handed or zero-handed simple-graph } l_2 \\ \qquad\qquad\qquad\qquad \text{with hand (possibly empty) } h_2 \\ m(h[j]) = level(i-j)[2] \text{ maps some nodes of } h \text{ to the place} \\ \qquad\qquad\qquad\qquad\qquad \text{where nodes were placed on step 2} \end{cases}$$

     In this case we define $\tilde{\varphi}$ for l and l_2 at a time the following way:

$$\begin{cases} \tilde{\varphi}(h[j]) = level(i+j)[2] \\ \tilde{\varphi}(h_2[j]) = level(i+1-j)[2] \end{cases}$$

- The symmetric case is when

$$\begin{cases} trunkCore(T)[i-1] \text{ is an inner node and it is a foot of another} \\ \qquad\qquad\qquad\qquad \text{one-handed or zero-handed simple-graph } l_2 \\ \qquad\qquad\qquad\qquad \text{with hand (possibly empty) } h_2 \\ m(h[j]) = level(i+j)[2] \text{ maps some nodes of } h \text{ to the place} \\ \qquad\qquad\qquad\qquad \text{where nodes were placed on step 2} \end{cases}$$

  In this case we define $\tilde{\varphi}$ for l and l_2 at a time the following way:

$$\begin{cases} \tilde{\varphi}(h[j]) = level(i-j)[2] \\ \tilde{\varphi}(h_2[j]) = level(i-1+j)[2] \end{cases}$$

- If the previous two cases don't come true we act pretty much the similar as we did for two-handed simple-graph, namely denote hand of l with h and choose one of the following definitions of $\tilde{\varphi}$ which doesn't map nodes of h to the places already used on step 2:

$$\left[\begin{array}{l} \tilde{\varphi}(h[j]) = level(i+j)[2] \\ \tilde{\varphi}(h[j]) = level(i-j)[2] \end{array} \right.$$

4. The last case is to consider an exit-graph.

   Denote the end-node of the trunk core, to which the exit-graph is connected by i. i is either $|V(trunkCore(T))|$ or 1.

   - If $i = |V(trunkCore(T))|$. There are two line-graphs connected to i, say l_1 and l_2. Note that they can't both be two-handed, since that means we have three nodes of degree three in a row, the middle one is the end node of the trunk core, but then the middle one by Lemma 8.3.7 must have the third neighbour of degree one which is not the case since it is a trunk core node and trunk core nodes are all of degree ≥ 2.

So let's assume that l_1 is one-handed. We embed it with

$$\tilde{\varphi}(l_1[j]) = level(i+j)[1]$$

If l_2 is one-handed we embed it with

$$\tilde{\varphi}(l_2[j]) = level(i+j-1)[2]$$

If l_2 instead has two hands h_1 and h_2 and it connects to i with the node $l_2[j]$. Then we define

$$\begin{aligned}\tilde{\varphi}(l_2[j]) &= level(i)[2]\\ \tilde{\varphi}(h_1[k]) &= level(i+k)[2]\\ \tilde{\varphi}(h_2[k]) &= level(i+k)[2]\end{aligned}$$

- If $i = 1$ we do everything symmetrically. Remember we don't care if we go out $Grid_n$ top or bottom borders, if it happens, we can just enlarge our grid to $Grid_m$ for some large enough m to accommodate the image.

Definition 8.11. The definition of φ on the hand(s) of the simple-graph connected to the inner trunk core node is called the orientation of that simple-graph.

Definition 8.12. We say that two inner simple-graphs are neighbours if there are no other simple-graphs connected to the trunk core in between their foots.

Lemma 8.3.8. *The resulting embedding of this strategy exists and it is quasi-correct.*

Before diving into prove let us discuss what does the Lemma give to us. There are three key points about the described quasi-correct embedding.

First of all, we should emphasise that such embedding can be efficiently computed.

Not only that, but it also can be recomputed easily while remaining quasi-correctness when the new vertices come, which is relevant to the online scenario.

And last but not least, recall that each two adjacent nodes are embedded at the distance at most five (see Lemma 8.3.6) so we are not worried about serving the same request many times.

Proof. The lemma is obvious for the trees with $|V(trunkCore(T))| \leq 1$, since we can do a correct embedding.

Denote the resulting embedding with $\tilde{\varphi}$. We know that a correct embedding exists, denote it φ.

- The described embedding of the trunk core meets no constraints, so it always exists.

- Let $top := |V(trunkCore(T))|$.

  The described embedding of the exit-graphs does not have any constraints, so it exists. Let us now focus on the exit-graphs connected to $trunkCore(T)[top]$. For each node u of those exit-graphs it is true that $\tilde{\varphi}(u)$ is on the $level(top)$ or higher. There are no more than three nodes of exit-graphs per level. Simple-graphs connected to the inner trunk core nodes are not allowed to pass through $level(top)$, so since they are connected, no nodes from simple graphs are embedded into levels $\geq top$. There are no nodes of the trunk core higher than top and on $level(top)$ there are only one node from our exit-graphs. The exit graphs connected to the $trunkCore(T)[1]$ are all embedded to the levels ≤ 1, so they can't interfere with the exit-graphs connected to the $trunkCore(T)[top]$. Thus we conclude that nodes of exit-graphs connected to the $trunkCore(T)[top]$ do not violate quasi-correctness since there are no more than three nodes on their levels. The same for the exit-graphs connected to the $trunkCore(T)[1]$.

- Now to the two-handed inner simple-graphs. The leg of each such simple-graph for any correct embedding must be embedded horizontally, i.e. $\varphi(foot)$ and $\varphi(head)$ must be on the same level. This is since we know that the trunk core image is monotone by Lemma 8.3.4 and it can't be if $\varphi(foot)$ and $\varphi(head)$ are on the different levels:

Say $\varphi(foot) = level(i)[1]$ and $\varphi(head) = level(i+1)[1]$. Then $level(i+1)[2]$ and $level(i+2)[1]$ are occupied by $head$ neighbours since it is of degree three. The $foot$ is also of degree three because it is an inner trunk core node with a simple-graph connected to it. Thus $level(i-1)[1]$ and $level(i)[2]$ are occupied with its neighbours, trunk core nodes. But the node mapped to $level(i)[2]$ can't be the end node of the trunk core since then it is of degree three and the node mapped to $level(i+1)[2]$ is its neighbour thus we obtain a cycle $\varphi^{-1}(\{level(i)[1], level(i+1)[1], level(i+1)[2], level(i)[2]\})$. So there is another trunk core node after it and it is inevitably mapped to $level(i-1)[2]$ violating monotone property of the trunk core embedding.

Now denote the hands of our two-handed graph with h_1 and h_2 and let's say that $\varphi(foot) = level(i)[1]$, $\varphi(head) = level(i)[2]$ and $\varphi(h_1[1]) = level(i+1)[2]$. Then $\varphi(h_1[2])$ must be $level(i+2)[2]$ since $level(i+1)[1]$ is occupied by the $foot$ trunk core neighbour $next$ (remember $foot$ is inner). If now $next$ is of degree three we obtain a conflict or a cycle, since $next's$ neighbour occupies $level(i+2)[2]$. If not, $next$ is an inner trunk core node and we continue with the $level(i+3)[2]$ for the $h_1[3]$ and $level(i+3)[1]$ for the next trunk core node of $next$. So we do until we ran out of h_1 nodes. We now say that there are no nodes of degree three in

$$\{trunkCore(T)[i+j] \mid j \in [|V(h_1)|]\}$$

since if there is j such that $trunkCore(T)[i+j]$ is of degree three, we obtain a conflict between the third neighbour of $trunkCore(T)[i+j]$ and $h_1[j]$. That means that $\tilde{\varphi}(h_1[j]) = level(i+j)$ will not place a node to the slot already occupied on step 2 of the strategy. The same for h_2. We call this line of reasoning the *inductive argument*.

So we proved that for each two-handed simple-graph connected to the inner node of a trunk core one of its orientations will not face conflicts with a neighbours of a trunk core nodes of degree three. Or in other words two-handed inner graphs can't violate the existence of the described embedding.

- We've shown that the quasi-correctness can't be violated on the levels $\geq top$

and ≤ 1. So now we need to proof that it is not violated in between.

To violate the quasi-correctness we need to obtain at least four nodes per level. Since on each level between top and 1 there is a node from the trunk core and there are no nodes from exit-graphs we conclude that there must be at least three nodes of an inner-simple graphs. And note also that they must be from the different simple-graphs since we don't embed more than one node from one inner simple-graph per level. Denote those simple-graphs with a, b, c. Their foots are somehow ordered in the trunk core, say $foot(b)$ is between $foot(a)$ and $foot(c)$. Since simple-graphs hands conflict at some node, we conclude that either hand of a crosses the $head(b)$ or a hand of c crosses the $head(b)$, otherwise a and c just don't share nodes. W.l.o.g. hand of a crosses $head(b)$. But this is only possible when b and a are one-handed graphs with adjacent foots and in this case their hands are oriented contrary and they only have two conflicts: $hand(a)[1]$ is embedded to the same node as $head(b)$ and $hand(b)[1]$ is embedded to the same node as $head(a)$. So c can possibly participate in that conflict only if c is a one-handed graph with a foot adjacent to $foot(b)$. That is because by our strategy two-handed simple-graphs do not cross other simple-graphs heads at all and the one-handed do only if their foots are adjacent. Our goal now is to show that in such setting c can be oriented the other direction to avoid conflict with b.

Note that we have three nodes of degree three and edges $(foot(a), foot(b))$, $(foot(b), foot(c))$. This is exactly the statement of Lemma 8.3.7, so we conclude that we have the following structure up to symmetry:

$$\begin{aligned} \varphi(foot(b)) &= level(i)[1] \\ \varphi(foot(a)) &= level(i-1)[1] \\ \varphi(foot(c)) &= level(i+1)[1] \\ \varphi(head(b)) &= level(i)[2] \end{aligned}$$

and we know that b in fact consists of one node.

We still have two possibilities for $head(c)$, namely $level(i+1)[2]$ or $level(i+2)[1]$.

If $\varphi(head(c)) = level(i+1)[2]$, then $\varphi(c[2]) = level(i+2)[2]$ and by the inductive argument applied to the $hand(c)$ there are no nodes of degree three in

$$\{trunkCore(T)[i+1+j] \mid j \in [|V(hand(c))|]\}$$

so

$$\tilde{\varphi}(hand(c)[j]) = level(i+1+j)[2]$$

won't place nodes of c to the slots already occupied on step 2 of the strategy.

If on the other hand $\varphi(head(c)) = level(i+2)[1]$ then $\varphi(trunkCore(i+2))$ must be $level(i+1)[2]$. Thus $trunkCore(i+2)$ can't be the end of the trunk core since then it is of degree three but $level(i)[2]$ is occupied by $head(b)$. So we say that $trunkCore(i+3)$ exists and $\varphi(trunkCore(i+3)) = level(i+2)[2]$ and it is also not the end node since it can't be of degree three since $level(i+2)[1]$ is occupied by the assumption by the $head(c)$. We now apply the inductive argument obtaining that there are no nodes of degree three in

$$\{trunkCore(T)[i+2+j] \mid j \in [|V(c)|]\}$$

We also showed that $trunkCore(T)[i+2]$ is not of degree three, so we state that $\tilde{\varphi}(hand(c)[j]) = level(i+1+j)$ won't place nodes of c to the slots already occupied on step 2 of the strategy.

This completes the proof of quasi-correctness of the embedding.

- So the last thing to show is that the described embedding exists for one-handed inner graphs.

  Suppose we have a one-handed inner graph l with hand h connected to the i-th node of the trunk core. Suppose also that w.l.o.g. $\varphi(foot(l)) = level(i)[1]$.

The only constrainted case in our strategy is when the following doesn't hold:

$$\left[\begin{array}{l}\left\{\begin{array}{l} trunkCore(T)[i+1] \text{ is an inner node and it is a foot} \\ \qquad \text{of another one-handed or zero-} \\ \qquad \text{handed simple-graph } l_2 \text{ with} \\ \qquad \text{hand (possibly empty) } h_2 \\ m(h[j]) = level(i-j)[2] \text{ maps some nodes of } h \text{ to the} \\ \qquad \text{place where nodes were placed} \\ \qquad \text{on step 2} \end{array}\right. \quad (1) \\ \left\{\begin{array}{l} trunkCore(T)[i-1] \text{ is an inner node and it is a foot} \\ \qquad \text{of another one-handed or zero-} \\ \qquad \text{handed simple-graph } l_2 \text{ with} \\ \qquad \text{hand (possibly empty) } h_2 \\ m(h[j]) = level(i+j)[2] \text{ maps some nodes of } h \text{ to the} \\ \qquad \text{place where nodes were placed} \\ \qquad \text{on step 2} \end{array}\right. \quad (2)\end{array}\right.$$

For the proof by contradiction assume now that both $\tilde{\varphi}(h[j]) = level(i+j)[2]$ and $\tilde{\varphi}(h[j]) = level(i-j)[2]$ map the nodes of h to the places already used on step 2. By the inductive argument that means that there exist such $j_1, j_2 \leq |V(h)|$ that $trunkCore(i+j_1)$ and $trunkCore(i-j_2)$ are of degree three. But that means that $\varphi(head(l)) \neq level(i)[2]$ since in that case by the inductive argument $\varphi(h[j])$ must be either $level(i-j)[2]$ or $level(i+j)[2]$ but in the first case we obtain a conflict with a neighbour of $trunkCore(i+j_1)$ and in the second case we obtain a conflict with $trunkCore(i-j_2)$.

So, $\varphi(head(l))$ is either $level(i+1)[1]$ or $level(i-1)[1]$. Let's consider the case $level(i+1)[1]$, the second is totally symmetric.

The trunk core nodes adjacent to $foot(l)$ are $trunkCore(T)[i-1]$ and $trunkCore(T)[i+1]$ they are mapped by φ to $level(i-1)[1]$ and $level(i)[2]$. We consider the case where $\varphi(trunkCore(T)[i-1]) = level(i-1)[1]$ and $\varphi(trunkCore(T)[i+1]) = level(i)[2]$ and we show that in this case (1) holds. Symmetrically $\varphi(trunkCore(T)[i-1]) = level(i)[2]$ and $\varphi(trunkCore(T)[$

$1]) = level(i-1)[1]$ will lead to (2).

We now prove that $trunkCore(T)[i+1]$ can't be the end node of the trunk core.

In Lemma 8.3.2 we make a path through the support nodes. If $trunkCore(T)[i+1]$ is not a support node, it can't be the end node of the trunk core since the trunk core connects to support nodes. It is neither a single node, since it a has a neighbour $trunkCore(T)[i]$ of degree three. So, since it is in the trunk core, we deduce that it is of degree three and there are nodes a and c of degree three s.t. there is a path $a \to trunkCore(T)[i+1] \rightsquigarrow c$. If a is different from $trunkCore(T)[i]$ then we have three consecutive nodes $trunkCore(T)[i]$, $trunkCore(T)[i+1]$ and a of degree three, so by the Lemma 8.3.7 $\varphi(trunkCore(T)[i])$ is on the same side as $\varphi(trunkCore(T)[i+1])$, which contradicts the assumption of $\varphi(trunkCore(T)[i]) = level(i)[1]$ and $\varphi(trunkCore(T)[i+1]) = level(i)[2]$. So there is a node c of degree three which is not adjacent to $trunkCore(T)[i+1]$ and there is a path $trunkCore(T)[i] \to trunkCore(T)[i+1] \rightsquigarrow c$. But then either c or its pair (if it is paired) is in the trunk core meaning that the trunk core passes through $trunkCore(T)[i+1]$ so it is inner.

Thus the $trunkCore(T)[i+2]$ exists and it has no other options but to be embedded to $level(i+1)[2]$ since if it is embedded to $level(i-1)[2]$ it is embedded to the same level as $trunkCore(T)[i-1]$ and it violates the trunk core monotone property stated by Lemma 8.3.4. So we are now able to apply the inductive argument deducing that there are no nodes of degree three in $\{trunkCore(T)[i+1+j] \mid j \in [|V(l)|]\}$. Recall that by our proof by contradiction assumption we have that mapping $m : m(h[j]) = level(i+j)[2]$ maps some nodes of h to the place where nodes were placed on step 2 meaning that there is a node of degree three in $\{trunkCore(T)[i+j] \mid j \in [|V(h)|]\}$. But this implies that the node $trunkCore(T)[i+1]$ is of degree three, it is inner, so we have an inner simple-graph connected to it, moreover this simple graph is a one-handed graph since otherwise we have three nodes of degree three: $trunkCore(T)[i]$, $trunkCore(T)[i+1]$ and

$head$ of that simple-graph, implying by Lemma 8.3.7 that $trunkCore(T)[i]$ and $trunkCore(T)[i+1]$ are mapped to the same side of the grid which as we know is not the case.

So we have that $m : m(h[j]) = level(i+j)[2]$ maps some nodes of h to the place where nodes were placed on step 2 and that there is a one-handed inner simple-graph connected tot he $trunkCore(T)[i+1]$ which is exactly the case (1).

□

8.3.2 Cycle embedding strategy

Definition 8.13. A *maximal* cycle C of a graph G is a cycle in G which cannot be enlarged, i.e., there is no other cycle C' in G such that $V(C) \subsetneq V(C')$.

Definition 8.14. Consider a graph G and a maximal cycle C of G. The whisker W of C is a path in G such that:

- $V(W) \neq \emptyset$
- $V(W) \cap V(C) = \emptyset$
- There exists only one edge between the cycle and the whisker (w, c) in G 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.
- There are no nodes of degree three in $V(W)$
- W is maximal, i.e., there is no W' in G such that W' satisfies previous properties and $V(W) \subsetneq V(W')$

Definition 8.15. Suppose we have a graph G that can be correctly embedded into $Grid_n$ by φ and a cycle C in G. Whiskers W_1 and W_2 of C are called adjacent for the embedding φ if

$$\forall i \in [\min(|V(W_1)|, |V(W_2)|] \ (\varphi(W_1[i]), \varphi(W_2[i])) \in E(Grid_n)$$

Statement 8.3.3. *For any correct embedding of a cycle C into $Grid_n$ each level of $Grid_n$ is either occupied with two nodes of C or not occupied at all.*

Proof. Suppose the contradictory, and there exists a correct embedding φ of C such that there is only one node of C, say a, on some level i, i.e., $level(i)[1]$. a has two neighbours in C, which we call b and c. W.l.o.g. we say that $\varphi(b) = level(i-1)[1]$ and $\varphi(c) = level(i+1)$. We define $next_{ab}(x)$ for the node $x \in V(C) \setminus \{a\}$ as the next node in C for x in the direction ab. It is easy to see that if $level\langle\varphi(x)\rangle > i$ then $level\langle\varphi(next_{ab}(x))\rangle > i$ since it cannot be less than i and due to the connectivity of the cycle image and it cannot be equal to i since then $next_{ab}(x) = a$ and then $x = c$ but $level\langle\varphi(c)\rangle = i - 1$. $level\langle\varphi(b)\rangle = i + 1 \to level\langle\varphi(next_{ab}(b))\rangle > i \to level\langle\varphi(next_{ab}(next_{ab}(b)))\rangle > i \to \ldots \to level\langle\varphi(c)\rangle > i$ which is a contradiction. □

Lemma 8.3.9. *Suppose we have a graph G which can be embedded into $Grid_n$. Suppose there exist a maximal cycle C in G with $V(C) \geq 6$ 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 $Grid_n$.*

Proof. At first, we show that for every correct embedding $foot(W_1)$ and $foot(W_2)$ are embedded to the same level of the grid. Suppose not. So there exists a correct embedding φ of G s.t. $\varphi(foot(W_1)) = level(i)[1]$ and $\varphi(foot(W_2)) = level(i-1)[1]$. By the Statement 8.3.3 $level(i)[2]$ and $level(i-1)[2]$ are also occupied with nodes from cycle. So we deduce that $\varphi(W_1[1]) = level(i+1)[1]$ and $\varphi(W_2[1]) = level(i-2)[1]$. But by Statement 8.3.3 it means that there are no nodes of C mapped to the levels $i+1$ and $i-2$ and so due to connectivity of the cycle image there are no more nodes of the cycle, but then there are only four nodes in C.

Now we want to show that $W_1[1]$ and $W_2[1]$ are embedded to the same level of the grid for any correct embedding of G. Suppose not. So there exists a correct

embedding φ of G s.t.

$$\varphi(foot(W_1)) = level(i)[1]$$
$$\varphi(foot(W_2)) = level(i)[2]$$
$$\varphi(W_1[1]) = level(i+1)[1]$$
$$\varphi(W_2[1]) = level(i-1)[2]$$

But this by Statement 8.3.3 implies that there are no nodes of C mapped to levels $i+1$ and $i-1$ and thus there are no more nodes of C at all due to the connectivity of the cycle image. Contradiction, since there are at least 6 nodes in C, not 2.

So for every correct mapping φ of G we know that up to symmetry it does the following:

$$\varphi(foot(W_1)) = level(i)[1]$$
$$\varphi(foot(W_2)) = level(i)[2]$$
$$\varphi(W_1[1]) = level(i+1)[1]$$
$$\varphi(W_2[1]) = level(i+1)[2]$$

So there is no other option for $W_1[2]$ and $W_2[2]$ but to be embedded to $level(i+2)[1]$ and $level(i+2)[2]$ respectively and so until we reach the end of either W_1 or W_2. In other words for any correct embedding φ

$$\forall i \in [\min(|V(W_1)|, |V(W_2)|] \ (\varphi(W_1[i]), \varphi(W_2[i])) \in E(Grid_n)$$

□

Remark 8.3.1. *Due to Lemma 8.3.9 if the cycle is of length ≥ 6 we can forget about an embedding while talking about adjacent whiskers.*

Definition 8.16. Assume we have a graph G and a maximal cycle C in G of length ≥ 6. The frame for C is a subgraph of G induced by vertices of C and $\{W_1[i], W_2[i] \mid i \in [\min(|V(W_1)|, |V(W_2)|)]\}$ for each pair of adjacent whiskers W_1 and W_2.

Nodes $W_1[\min(|V(W_1)|, |V(W_2)|)]$ and $W_2[\min(|V(W_1)|, |V(W_2)|)]$ are called end nodes of the frame.

Lemma 8.3.10. *If we have a cycle of length at least* 6 *in a graph which is a subgraph of the request graph then its end nodes of the frame are adjacent in the request graph.*

Proof. This is because by Lemma 8.3.9 they are adjacent for every embedding and in particular for the original embedding of the cycle into the request graph. □

Remark 8.3.2. *Due to Lemma 8.3.10 we can "extend" each maximal cycle to the ends of its frame, so, we we do not have any adjacent whiskers, i.e., one whisker is embedded fully.*

Lemma 8.3.11. *Assume we have a graph G which can be embedded into $Grid_n$ and a maximal cycle C of length at least* 6 *of G has no adjacent whiskers. Then, there are at most two nodes connected to C (i.e. $(v, c) \in E(G)$, such that $v \in V(G) \setminus V(C) \wedge c \in V(C)$).*

Moreover, these two connecting nodes are not adjacent.

Proof. Consider a correct embedding φ of G into $Grid_n$. The cycle occupies level from i to j, $i < j$ (it can't make a gap due to the connectivity of the image and by the Lemma 8.3.3 it occupies the whole level). So the possible places for v are $level(i-1) \cup level(j+1)$.

For the proof by contradiction assume that there are at least three nodes connected to C. Then by the pigeon hole principle there are two of them on the same level, say v_1 and v_2. There can't be an edge between v_1 and v_2 since then the cycle can be extended by adding v_1 and v_2 and thus is not maximal. But if there is no edge between v_1 and v_2 they form whiskers and those whiskers are adjacent. Contradiction. □

Trees and cycles

Definition 8.17. 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

- $\bigcup_{i \in [n]} V(C_i) \cup \bigcup_{i \in [m]} V(T_i) = V(G)$

- $V(C_i) \cap V(C_j) = \emptyset \; \forall i \neq j$
- $V(T_i) \cap V(T_j) = \emptyset \; \forall i \neq j$
- $V(T_i) \cap V(C_j) = \emptyset \; \forall i \in [m], j \in [n]$
- $\forall i \neq j \; \forall u \in V(T_i) \; \forall v \in V(T_j) \; (u, v) \notin E(G)$

Lemma 8.3.12. *Assume we have a graph G which can be embedded into $Grid_n$. Suppose that there is a cycle C and a tree T from cycle-tree decomposition of G, such that C and T are connected by an edge $(c, t) \in E(G)$, where $t \in V(T)$ and $c \in V(C)$. Then, for any correct embedding φ $t \in trunk_\varphi(T)$.*

Proof. By Lemma 8.3.3 there is another node of C on the $level\langle\varphi(c)\rangle$. Let's say this level has number i. If $\varphi(t) \in level(i+1)$ then we know that no nodes of T can be embedded to the $level(i)$ (and thus, due to the connectivity of T-s image, below it) so t is the bottom most node and thus it is in the trunk. Symmetrically, it is the top most node of T if $\varphi(t) \in level(i-1)$. □

Definition 8.18. We call a node t from Lemma 8.3.12 an *end-node* of a tree T, and a node c a *foot* of a T.

Remark 8.3.3. *If the tree T is connected to a cycle, then $trunkCore(T)$ can be extended to an end-node of T.*

We call the path connecting the end-node of the trunk core and an end-node of a tree an extension of the trunk core.

We call a trunk core with two of its possible extensions an extended trunk core.

The exit-graphs are now simple-graphs connected to the end-nodes of an extended trunk core.

Note that the end-nodes of the tree might not exist while the end-nodes of the trunk core are just the end-nodes of the path.

We now define how to embed a tree T from a cycle-tree decomposition.

We include possible foots of a tree with their neighbours in that tree, making them the end-nodes of the trunk core. We then apply strategy 8.3.1 to the obtained tree.

Definition 8.19. We say that such an embedding of a tree respects the strategy 8.3.1.

8.4. Dynamic algorithm

Now, we talk about how we update the embedding with respect to new requests.

In our strategy of edge processing, if an already known edge is requested we do nothing since the requested nodes are already at the distance at most 12, because by the assumption the enumeration preserves the proximity property (see the strategy plan 8.1).

But if we obtain a new edge, our enumeration may no longer maintain the proximity property , so we perform a re-enumeration.

There are two possible cases for the new edge. It may be within the connectivity component or it may connect two different connectivity components. We analyse these cases separately.

We want to maintain the following five invariants:

1. The embedding of any connectivity component is quasi-correct.
2. For each tree in the cycle-tree decomposition the embedding of that tree matches the strategy 8.3.1
3. We do not have maximal cycles of length 4
4. Each maximal cycle does not have adjacent whiskers
5. There are no conflicts with cycle nodes

8.4.1 New edge within one connectivity component

Assume we have a connectivity component S with at least one cycle (call it C_S) and a quasi-correct embedding φ' of S preserving all the invariants from 8.4.

Assume the new edge connects nodes u and v. Since u and v are already in a one connectivity component we conclude that there is now a cycle C' containing u and v. We consider a maximal cycle C containing C'.

We call a graph S with an edge (u, v) as S_+.

If C is of length 4, for every two nodes a and b of C $|level\langle\varphi'(a)\rangle - level\langle\varphi'(b)\rangle| \leq 3$ since the distance in S between a and b is at most 3 and φ'

preserves connectivity. Thus, since there are no more than 3 nodes per level, we conclude that the difference between numbers of a and b is at most 12, so, the proximity property is maintained and we do nothing.

If C is now of length at least 6 (note that if a cycle can be embedded into $Grid_n$ its length must be even) we consider its frame F stating that it is in fact a cycle by Lemma 8.3.10 and that it has at most two nodes connected to it by Lemma 8.3.11.

Lemma 8.4.1. *Consider a graph G embedded into $Grid_n$ with some embedding φ that respects invariants 8.4. Consider a frame of a maximal cycle C in G embedded by φ into levels from i to j ($i < j$). Then $\{v \mid v \in V(G),\ level\langle\varphi(v)\rangle > j\}$ form a connectivity component.*

Proof. By the invariant 4 8.4 and Lemma 8.3.11 there is only one node u connected to C with $level\langle\varphi(u)\rangle > j$. So, if some path from a to b ($level\langle\varphi(a)\rangle > j$, $level\langle\varphi(b)\rangle > j$) goes through a node v with $level\langle\varphi(v)\rangle \leq j$ it must pass through u twice, meaning we can replace $a \rightsquigarrow u \rightsquigarrow v \rightsquigarrow u \rightsquigarrow b$ with $a \rightsquigarrow u \rightsquigarrow b$. □

We say that a group of nodes is on the same side from a cycle if they are in a one connectivity component when the cycle is removed.

Lemma 8.4.2. *If after the removal of F, a connectivity component S_i $i \in \{1, 2\}$ is a line-graph and it connects to F via its end-node then all of its nodes belong to an exit-graph in S.*

Proof. Since by our assumption S has a cycle, we state that each tree has a non-empty extended trunk core and, thus, each node of the component has only four options where to belong. It is either in a cycle, an extended trunk core, an inner-graph, or an exit-graph. So, we now prove that the first three do not happen here:

- The node remains on a cycle when adding a new edge and the nodes from S_i are not on the cycle in S_+, thus, they were not in S.
- Each node in the extended trunk core is either of degree three or has two edge-disjoint paths to nodes of degree three. Those properties can't disappear

when adding a new edge and none of them hold in S_+ for nodes in l thus did not them in S. So the nodes from l were not in the extended trunk core.

- As shown in 8.3.1 for every correct embedding nodes of a simple-graphs always have a node embedded to the same level, namely a node from the trunk core. But S_+ has an embedding with all the nodes from S_i being single on their level, and since every embedding for S_+ induces an embedding for S the same holds for them in S thus none of them is a node of an inner-graph in S.

□

Lemma 8.4.3. *If a connectivity component l left when removing F is a line-graph and it connects to F via its inner node u then all of its nodes belong to an exit-graph in S except possibly u.*

Proof. The proof is almost the same as in 8.4.2 but with to adjustments:

- The second bullet does not hold for u
- l has an embedding where a node from l is single on its level or with another node of l and since nodes of l are not the inner trunk core nodes that means that they are not nodes of inner simple-graphs.

□

By Lemma 8.4.1 there are at most two connectivity components left when removing F. Let's call them S_1 and S_2. We now describe how we embed F, S_1 and S_2.

So, imagine that we formed F and it has node f of degree three. We first discuss the case of S_1 being a line-graph connected with f with its end-node. By Lemma 8.4.2 we deduce that S_1 was a part of an exit-graph in S and thus it was embedded strictly monotonically. In other words, (let's set an enumeration of S_1 such that $(S_1[1], f) \in E(S_+))$ $level\langle\varphi(S_1[1])\rangle < level\langle\varphi(l[2])\rangle < \ldots$. We then embed F in the way that f is embedded higher then any other node of F, say to $level(i)[1]$. And we embed $S_1[j]$ to $level(i+j)[1]$. If the levels of nodes of S_1

were decreasing we act the same way but embedding f lower then other nodes of F and embedding S_1 in decreasing order.

What if now S_1 is a line-graph connected to F via its inner node. By the Lemma 8.4.3 we know that its hands (call them h_1 and h_2) were exit-graphs and thus were embedded monotonically, assume increasingly numerating from head. Assume head of S_1 was connected to $f \in V(F)$. We then embed F in a way that f is embedded higher then any other node of F say to $level(i)[1]$. We then embed

$$\begin{aligned} head(S_1) &\to level(i+1)[1] \\ h_1[j] &\to level(i+1+j)[1] \\ h_2[j] &\to level(i+j)[2] \end{aligned}$$

We act symmetrically if the order on h_i was decreasing.

We now want to show that we cannot face incompatibility namely that we have two line-graphs connected (no matter via their inner or end nodes) to F both having increasing (decreasing) order on their hands in S. By Lemma 8.4.2 those line-graphs were both whiskers in S, so we denote them W_1 and W_2. If they both had the same order that means they were in the same tree. To see this we first prove that there are at most two trees from the cycle-tree decomposition of the component can have exit-graphs. Consider the embedding satisfying invariants 8.4. It induces an order on the maximal cycle of the component, since maximal cycles do not intersect and there for can be enumerated from bottom to top for example. If the tree is embedded between two consecutive cycles (say, C_1 and C_2) it must be connected to both of them. This is because they are connected with some path connecting nodes c_1 and c_2 of cycles. We consider such path that contains just two nodes from cycles, it is straightforward to see that such path can be obtained if we have an arbitrary one. This path (except c_1 and c_2) belongs to some tree T in a cycle-tree decomposition. If now some tree different from T (say, T_2) from a cycle-tree decomposition is embedded between cycles, since the component is connected it has two options: either to be connected to T or to one of the cycles. It can't be connected to T by the definition of the cycle-tree decomposition. Neither it can be connected to C_1 or C_2 since by the Lemma 8.4.1 if that cycle is removed since T and T_2 are on the one side of that cycle they are in the one connectivity

component. But this implies that T_2 is connected to other cycle which is impossible due to the Lemma 8.4.1.

The only way for a tree to have exit-graphs is to be embedded below the lowest cycle or above the highest. But exit-graphs in the highest tree are embedded increasingly and the whiskers in the lowest tree are embedded decreasingly.

This means thatW_1 and W_2 are in the same tree in S, or in other words, they are both on the same side from C_S. But now this cycle is contained in F (since we only have F, W_1, W_2) and since W_1 and W_2 by Lemma 8.3.11 are now on the opposite sides of F they are on the opposite sides to the C_S, which is impossible since then removing C_S leaves W_1 and W_2 in the different connectivity components which was not the case in S.

Lemma 8.4.4. *Assume that S_i $i \in \{1, 2\}$ connects to F via a node u, $u \in V(T)$ where T is a tree from the cycle-tree decomposition of S_i and S_i has a node of degree three. Then all the nodes from an extended trunk core of T are embedded monotonically in φ'.*

Proof. All the nodes from a trunk core of T belong to a trunk core in S. This is because T is contained in some tree T_S from a cycle-tree decomposition of S and thus a trunk core of T is contained in a trunk core of T_S since all the support nodes remain support nodes when extending a tree.

So we consider two extensions of T-s trunk core ext_1 and ext_2. Let's say ext_1 is the one which extends to F. Note that ext_2 was an extension in S and thus we already have that $trunkCore(T) \cup ext_2$ is embedded monotonically by φ'.

The ext_1 was either a part of an extended trunk core of T_S or an exit-graph in T_S. This is because it is obviously couldn't have been a part of a cycle, since nodes on cycle remain on cycle when the new edge is added. Neither could it have been a part of the inner simple-graph since then the end-node of the $trunkCore(T)$ to which ext_1 is connected was an inner node of the trunk core meaning that there was two edge-disjoint paths from it to two support nodes. If those support nodes are in T now the end-node of $trunkCore(T)$ is an inner trunk core node of T which is nonsense. Otherwise it is not in S_i meaning the path to it goes through ext_1 since it is the only path connecting T and F. But then ext_1 belongs to a trunk core in S. So ext_1 was either a part of an exit-graph or a part of the $trunkCore(T_S)$. In both

cases it is embedded monotonically with $trunkCore(T) \cup ext_2$. □

So assume we have S_i $i \in \{1, 2\}$ which connects to $f \in V(F)$ via a node u, $u \in V(T)$ where T is a tree from the cycle-tree decomposition of S_i and S_i has a node of degree three. Lemma 8.4.4 tells us that T-s extended trunk core was embedded monotonically, say, increasingly starting from u. In this case we embed F the way that f is embedded higher then every other node from F, say, to $level(i)[1]$, and we embed j-th node of an extended trunk core of T to the $level(i + j)[1]$. All the other nodes from S_i we embed the same as they were embedded by φ' relatively to the nodes of the extended trunk core of T.

We now analyze if we obtained a conflict of nodes from S_i and F and if T is embedded respecting strategy 8.3.1.

Consider inner simple-graphs of T in order they are connected to the extended trunk core of T going through the extended trunk core from u. We state that all of them except possibly the first two were inner simple-graphs in S. This is because for a foot of such simple-graph there is a support node before it (one of the foots of first two simple-graphs) and a support node or an end-node after it (because it is inner in S_+). So those simple-graphs can not conflict with nodes of F since they did not pass through the levels of foots of first two simple-graphs. They also respect the strategy 8.3.1.

So we only need to orient first two simple-graphs in the way they don't conflict with nodes of F and they respect the embedding strategy. This can be done, since the strategy can be applied to $T \cup \{f$ and its neighbours$\}$.

Our last goal in analyzing an embedding of a component with a node of degree three which starts with a tree is to show that we can't face incompatibility. If say, S_1 is starting with a tree T with increasing order on its extended trunk core in S then the order on S_2 (if one exists) is decreasing (meaning that S_2 has a decreasing order in S on an extended trunk core of a tree it starts with or a decreasing order in S on its hands if S_2 is just a line-graph). Let's say S_1 and S_2 are connected to F with nodes $c_1 \in V(S_1)$ and $c_2 \in V(S_2)$ respectively.

First assume we have S_1 starting with a tree T_1 and S_2 starting with a tree T_2. Both S_1 and S_2 have a node of degree three. For T_i take a node u_i which is a foot of T_i other then the one in F if such exists or the end of the extended trunk

core other then the one connected to F. Since S_i has a node of degree three, one of those must exist. A shortest path connecting u_1 and u_2 contain both extended trunk cores of T_1 and T_2. Since no path in $\varphi'(S)$ is self-crossing the path $u_1 - u_2$ must be monotone by Lemma 8.3.3 since no nodes of the path can be embedded to the same level with u_1 and u_2 (the neighbour of u_i is either a cycle node or an exit-graph node). Thus paths $u_1 - c_1$ and $c_2 - u_2$ have the same order in $\varphi'(S)$ so the extended trunk cores of T_1 and T_2 which are contained in $c_1 - u_1$ and $c_2 - u_2$ have opposite order.

Now consider the case when S_1 has a node of degree three and starts with a tree T and S_2 is just a line-graph. Let's say that S_1 connects to $f_1 \in V(F)$ and S_2 connects to $f_2 \in V(F)$. For T take a node u_1 which is a foot of T other then the one in F if such exists or the end of the extended trunk core other then the one connected to F. Since S_1 has a node of degree three, one of those must exist. And let's say we have an increasing order on the hands of S_2. Consider the top-most node of a hand of S_2 in $\varphi'(S)$, let's call it u_2. Note that u_2 is the top-most node among all S and can share level only with nodes from exit-graphs. Now consider the shortest path $u_1 - u_2$. It contains an extended trunk core of T. Our goal now is to proof that this path is monotone, that would imply as in the previous case that T-s extended trunk core and S_2-s hands have different order. To see that it is monotone recall that φ' produces no self-crossing paths and thus, if the path is not monotone, by Lemma 8.3.3 we either have u_1 sharing level with some other node from path which is impossible since it is either a cycle node or a trunk-core end-node. Or u_2 shares level with some other node from path which is also impossible since u_2 is the top-most node with only nodes from other exit-graphs hands possibly being on its level.

So we discussed what to do if S_i connects to the F with a node from tree from its cycle-tree decomposition. The last case is when it connects to F with a node of a cycle from its cycle-tree decomposition.

So assume S_1 connects to a node $f_1 \in V(F)$ with a cycle C-s node, say u_1. And let's assume was the top-most node in C the case when it is the bottom-most is totally symmetric. We embed F the way f_1 becomes a bottom-most node we embed C the way u_1 is the top most node and it is under f_1. We embed the rest of S_1 relatively to C as it was embedded by φ'.

Nothing changed in S_1, so the invariants maintain for it. Therefor our only goal is to show that we don't face incompatibility with S_2.

If S_2 starts with a cycle C_2 which connects to F via a node u_2 then we conclude that u_2 was the bottom-most node of C_2 since there is a path in which connects u_1 and u_2 without passing through other nodes of C_1 and C_2.

If S_2 starts with a tree we act similarly as we did in previous cases take such node u_2 that the path $u_1 - u_2$ contains tree-s trunk core/extended trunk core/hand of an exit graph and we chow that by Lemma 8.3.3 path $u_1 - u_2$ should be monotone, since no nodes of it can be embedded to the same level as u_2 for the same reasons as before and neither can they be embedded to the same level with u_1 since there is a cycle node embedded to the same level as u_1. So if, say, u_1 was the top-most node of C in $\varphi'(S)$ then the path is increasing and thus the trunk core/extended trunk core/hand of an exit graph is also increasing in $\varphi'(S)$ which is consistent to the fact that it connects to the top-most node of F.

All the actions described above assume that there was a cycle in S already. If there wasn't we act as described below.

The new edge (u, v) is in one connectivity component so there is a maximal cycle containing u and v. Let's take a frame F of that cycle. There are possibly two connectivity components left when removing F from S_+, denote them S_1 and S_2. Let's say that S_1 connects to $f_1 \in V(F)$ and S_2 connects to $f_2 \in V(F)$. Moreover we know that S_1 and S_2 are trees.

We embed F the way that f_1 is the top-most node and f_2 is the bottom-most node. We then embed S_1 and S_2 the way they match strategy 8.3.1 orienting the trunk not to conflict with cycle nodes.

We now want to analyze the cost of actions performed when serving a new edge within one component. Note that since the resulting embedding is quasi-correct the cost of serving the request is $O(1)$.

To make an amortized analysis we introduce the concept of *scenarios*. The scenario is a reason for node to move in the embedding and thus change its number. Each scenario has two main properties: the number of times it can happen to a certain node (denoted with SC_N) and a cost payed for that node movement in this

scenario (denoted with SC_C). So the total cost payed for node movements in this terms is bounded with

$$2n \cdot \sum_{SC \in \text{SCENARIOS}} SC_N \cdot SC_C$$

Where the first factor $2n$ arises due to the fact that SC_N and SC_C are defined for one particular node, but we want the total cost.

We propose that we only need to focus on the relative order changes.

Lemma 8.4.5. *If we have two enumerations h_1 and h_2 of graph G then the cost of obtaining h_2 from h_1 via swaps is no more than*

$$|\{(u, v) \mid u, v \in V(G),\ h_1(u) < h_1(v) \wedge h_2(u) > h_2(v)\}|$$

Proof. We can order nodes of G by h_2. h_1 can then be viewed as a permutation so the statement of the Lemma can be reformulated as "the swap distance between a permutation p and an identity permutation is less or equal the number of inversions in p".

We proof this by induction. The induction would be among the number of elements in permutations and among the number of inversions in permutations.

The induction base is 0 inversions for each number of elements which is trivial. We also notice that if there is just one element in the permutation then this permutation can't have inversions.

We now assume that our permutation p has n elements and k inversions, with $k > 0$ and $n > 1$.

If now $p[1] = 1$ then the distance between p and $identity_n$ is the distance between p' and $identity_{n-1}$ where $p'[i] = p[i + 1] - 1$. And since p' has the same number of inversions as p then by the inductive assumption it is less or equal to k which is what we desire.

If $p[1] = i$, $i \neq 1$ then we first spend $i - 1$ swaps to bring i to the position 1 reducing the number of inversions by $i - 1$. And then apply the same idea with p' obtaining that the distance from p' to an $identity_{n-1}$ is $k - (i - 1)$ and thus we provided the series of swaps to obtain an $identity_n$ from p with $\leq k$. □

Lemma 8.4.6. *Suppose S is a connectivity component with a cycle embedded into $Grid_n$ via φ' which respects invariants 8.4. Suppose we have a new edge in the connectivity component S. We denote S with a new edge by S_+. Let's call the frame of a maximal cycle containing the ends of the new edge F. The connectivity components left when removing F from S_+ are S_1 and S_2. Suppose that our algorithm embeds S_1 above F.*

1. *If there is a cycle in F that was present in S then all the nodes from S_1 were above that cycle in $\varphi'(S)$.*

2. *If there is a cycle in F that was present in S then there is a node of cycle that is top-most for both new and old embeddings.*

3. *For each node $u \in V(S_1)$ and for each node $v \in V(S_2)$ $level\langle\varphi'(u)\rangle > level\langle\varphi'(v)\rangle$ except possibly the first two inner simple-graph nodes of T_i if S_i connects to F with a tree T_i from a cycle-tree decomposition of S_i.*

Proof. Assume S_1 connects to F via edge $(f_1, c_1),\ f_1 \in V(F), c_1 \in V(S_1)$ and S_2 connects to F via edge $(f_2, c_2),\ f_2 \in V(F), c_2 \in V(S_2)$

1. We want to prove the following fact: consider component A is connected to cycle C with edge (a, c), $a \in V(A)$ $c \in V(C)$ and it is embedded above C by φ_1 and below C by φ_2. Then for each path in A starting from a which is monotone for both φ_1 and φ_2 it has changed its orientation i.e. if it was increasing it is now decreasing and vice versa. To see this note that a was a top-most node and became the bottom-most node of the path or vice versa.

   For each type of S_1 our new edge processing strategy maintained an orientation of some monotone path in S_1 which can be extended remaining monotone to the node connected to the cycle. Thus by the fact above it must remain on the same side of the cycle.

2. Denote the cycle in statement by C,. Denote by A the component that was above C in $\varphi'(S)$ which contains S_1. Say it is connected to C via edge (a, c) $a \in V(A)$ $c \in V(C)$. Then by item 1 node c is top-most in both embeddings of S and S_+ since there is a monotonically increasing path starting from a which does not pass through nodes of C.

3. If S_1 and S_2 are both line-graphs from the proof of compatibility for line-graphs we know that the must be in the different trees in a cycle-tree decomposition of S meaning they are separated by cycle and thus their nodes don't share levels. Moreover nodes form S_1 were in the top-most tree and nodes from S_2 were in the bottom-most tree, so indeed every node from S_1 was embedded higher than every node of S_2.

   For all the other cases on S_1 and S_2 in the proof of their compatibility we build a monotone path which passes through c_2, f_2, f_1, c_1. This path is monotonically increasing levels in $\varphi'(S)$ if S_1 is embedded above F by our algorithm. So $level\langle\varphi'(c_2)\rangle < level\langle\varphi'(c_1)\rangle$ so our goal now is to analyze what nodes from S_1 can go under $level\langle\varphi'(c_1)\rangle$ (the analysis for S_2 is symmetric).

   If S_1 connects to F with a cycle or S_1 is a line-graph no nodes can go under c_1.

   To analyze the last case on S_1 we need the following fact: if S_2 exists then $V(F)$ contains a node of degree three in S. This is because $V(F)$ contains two nodes of degree three in S_+ namely the foots of opposite (non adjacent) whiskers and those nodes are not adjacent since the cycle is of length at least 6. So one of them must have been of degree three before the new edge.

   So assume now S_1 connects to F with a tree and contains a node of degree three. From the analysis of compatibility for such S_1 we know that c_1 is either an extended trunk core node or an exit-graph node in S. If it is an exit-graph node, then S_2 does not exists since if it does there is a node of degree three in $V(F)$ in S and thus there are two disjoint paths from c_1 to nodes of degree three which can't be for an exit-graph node.

   If now c_1 is in an extended trunk core node of a tree T. If F contains a cycle that was present in S then nodes from S_1 and S_2 were separated by this cycle then by item 1 all the nodes from S_1 were above that cycle and all the nodes from S_2 were bellow so the proposal holds. All the nodes from an extended trunk core of T were above c_1 in S so the only possible nodes are the nodes to go below c_1 are nodes from inner simple-graphs of T. And

the inner simple-graphs starting from the third one can't cross the first two graphs heads so they can't cross c_1 as well.

□

We are now ready to analyze the cost payed by each node when a new edge in the connectivity component appears.

Scenario 8.1 (Inner simple-graph reorienting)**.** The node falls into this scenario if it is a part of an inner simple-graph which is being reoriented.

Lemma 8.4.7. *The Inner simple-graph reorienting scenario costs $O(n)$ for a node and happens only once for a given node.*

Proof. Node can't pay more than $2n$ so $O(n)$ bound is trivial.

The inner simple-graph has only two possible orientations. We change its orientation only if the current one violates the invariants 8.4 thus we will never get back to it. □

Scenario 8.2 (First time on a cycle)**.** The node falls into this scenario if it was not on a cycle before the new edge and after the new edge it is.

Lemma 8.4.8. *The First time on a cycle scenario happens at most once for each node and costs $O(n)$.*

Proof. Trivial. □

Scenario 8.3 (Cycle in the frame)**.** The node falls in this scenario if it is in the cycle which is a part of F and is present in S.

Lemma 8.4.9. *The Cycle in the frame scenario happens $O(n)$ times for each node and costs $O(1)$.*

Proof. Since we have only $O(n)$ edges the $O(n)$ upper bound is trivial.

If the cycle is embedded respecting invariants 8.4 with a node specified to be the top most (which is the case due to the lemma 8.4.6) then we have only two four possibilities for a cycle to be embedded and for each to of them one can be transformed to another making each node changing the relative order only with $O(1)$ nodes from cycle which by Lemma 8.4.5 gives us $O(1)$ cost per node.

Note also that by Lemma 8.4.6 nodes from cycle do not change relative order with nodes from S_1 or S_2 and thus the cost of their inner relative change is the only cost they need to pay. □

So the nodes of F fall into either *First time on a cycle* scenario or to the cycle in the frame scenario.

As for the S_i nodes we need to consider 2 cases. The nodes are either a part of first two inner simple-graphs when S_i starts from a tree or not.

If yes, and those node change their relative order to the nodes of S_i this means that the reorientation of simple-graphs had been performed thus they fall into *Inner simple-graph reorienting* scenario. If they did not change relative order to S_i this means they didn't change the order with S_j, $j \in \{1, 2\} \setminus \{i\}$ neither with F so they pay nothing.

As for the nodes from S_i that are not the part of first two inner simple-graphs by the Lemma 8.4.6 they didn't change the relative order with S_j, by the embedding strategy they didn't change the relative order with the nodes from S_i (// consider mirroring? //) accept possibly first two inner-simple graphs and also by the Lemma 8.4.6 they didn't change the relative order with the cycle contained in F. As for the nodes of F which were not on the cycle we say that they pay for all the relative order changes.

8.4.2 New edge between two components

We now define and analyse the behaviour of the algorithm when the edge between two connectivity components is revealed. The strategy would be to bring the larger component towards the smaller one.

Scenario 8.4 (Connectivity component movement)**.** . The node falls in this scenario if its component is a smaller of two between which the new edge is revealed.

Lemma 8.4.10. *The Connectivity component movement scenario happens* $O(\log n)$ *times and cost* $O(n)$.

Proof. The size of the component is at least doubled. □

We now dive into the case analysis.

We first want to distinguish to cases: either the bigger component has a tree with a non-empty trunk core or not.

If not that means that there are at most two nodes of degree three. If the new edge increases the number of degree three nodes from 0 to one, from 1 to 2 or from 2 to three we say that it is an individual scenario and we can allow the total reconfiguration according to the 8.3.1.

Scenario 8.5 (New degree three node.)**.** The node falls into this scenario if the new edge increases the number of the nodes of degree three in the component from 0 to 1, from 1 to 2 or from 2 to 3.

Lemma 8.4.11. *The New degree three node scenario can happen* $O(1)$ *times and costs* $O(n)$.

Proof. Trivial. □

We now assume that there is a tree in the bigger component with a non-empty trunk core or a cycle with an inner edge. The smaller component can then connect to:

1. The inner node of the trunk core.
2. The inner simple-graph node.
3. The cycle node.
4. The exit-graph node.

In the following analysis the new scenario appear

Scenario 8.6 (No more an exit-graph.)**.** The node falls into this scenario if it is no longer a part of an exit-graph.

Lemma 8.4.12. *The No more an exit-graph scenario happens at most once and costs* $O(n)$.

Proof. The closest node of degree three to the exit-graph node has only one path to another degree three node. □

Let us discuss all the possible cases.

1. If the smaller connectivity component connects to the inner trunk core node then by Lemma 8.3.5 it must be a line-graphs and there for to maintain the quasi-correctness of the embedding we only to choose its orientation and reorient its trunk core neighbours. The scenarios engaged here are the *connectivity component movement* and *inner simple-graph reorienting*.

2. For the inner simple-graph node the analysis is basically the same as in the previous case.

3. Cases:

   - Only cycle
   - Cycle and a line-graph
   - Cycle and a tree with a degree three node

4. Cases:

   - Smaller component does not make new nodes of degree three
   - It does. Then we reorient the exit-graphs as they are no longer exit-graphs. This is the no more an exit graph scenario.

9. Conclusion

In this paper, we presented three results: 1) the upper bound cost on any algorithm for an arbitrary demand graph; 2) an online algorithm with its cost for a cycle demand graph; and, finally, 3) an online algorithm with its cost for a $Grid_n$ demand graph. In the last two cases, we presented algorithms that match the lower bound. We think this is the first important step towards the tight bound for more generic graphs such as arbitrary grids that we are going to research in the future work.

Bibliography

[1] Chen Avin, Chen Griner, Iosif Salem, and Stefan Schmid. An online matching model for self-adjusting tor-to-tor networks. *CoRR*, abs/2006.11148, 2020.

[2] Chen Avin, Andreas Loukas, Maciej Pacut, and Stefan Schmid. Online balanced repartitioning. In *International Symposium on Distributed Computing*, pages 243–256. Springer, 2016.

[3] Chen Avin, Kaushik Mondal, and Stefan Schmid. Demand-aware network design with minimal congestion and route lengths. *IEEE/ACM Transactions on Networking*, 2022.

[4] Chen Avin, Iosif Salem, and Stefan Schmid. Working set theorems for routing in self-adjusting skip list networks. In *39th IEEE Conference on Computer Communications, INFOCOM 2020, Toronto, ON, Canada, July 6-9, 2020*, pages 2175–2184. IEEE, 2020.

[5] Chen Avin and Stefan Schmid. Toward demand-aware networking: a theory for self-adjusting networks. *Comput. Commun. Rev.*, 48(5):31–40, 2018.

[6] Chen Avin and Stefan Schmid. Renets: Statically-optimal demand-aware networks. In Michael Schapira, editor, *2nd Symposium on Algorithmic Principles of Computer Systems, APOCS 2020, Virtual Conference, January 13, 2021*, pages 25–39. SIAM, 2021.

[7] Chen Avin, Ingo van Duijn, and Stefan Schmid. Self-adjusting linear networks. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 368–382. Springer, 2019.

[8] Josep Díaz, Jordi Petit, and Maria Serna. A survey of graph layout problems. *ACM Computing Surveys (CSUR)*, 34(3):313–356, 2002.

[9] Evgeniy Feder, Ichha Rathod, Punit Shyamsukha, Robert Sama, Vitaly Aksenov, Iosif Salem, and Stefan Schmid. Toward self-adjusting networks for the matching model. In Kunal Agrawal and Yossi Azar, editors, *SPAA '21: 33rd*

ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021, pages 429–431. ACM, 2021.

[10] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil R. Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel C. Kilper. Projector: Agile reconfigurable data center interconnect. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 216–229. ACM, 2016.

[11] Chen Griner, Johannes Zerwas, Andreas Blenk, Stefan Schmid, Manya Ghobadi, and Chen Avin. Cerberus: The power of choices in datacenter topology design (a throughput perspective). In *Proc. ACM SIGMETRICS*, 2021.

[12] Ralf Heckmann, Ralf Klasing, Burkhard Monien, and Walter Unger. Optimal embedding of complete binary trees into lines and grids. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 25–35. Springer, 1991.

[13] Sikder Huq and Sukumar Ghosh. Locally self-adjusting skip graphs. In Kisung Lee and Ling Liu, editors, *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 805–815. IEEE Computer Society, 2017.

[14] Neil Olver, Kirk Pruhs, Kevin Schewior, René Sitters, and Leen Stougie. The itinerant list update problem. In *International Workshop on Approximation and Online Algorithms*, pages 310–326. Springer, 2018.

[15] Ch H Papadimitriou. The np-completeness of the bandwidth minimization problem. *Computing*, 16(3):263–270, 1976.

[16] Stefan Schmid, Chen Avin, Christian Scheideler, Michael Borokhovich, Bernhard Haeupler, and Zvi Lotker. Splaynet: Towards locally self-adjusting networks. *IEEE/ACM Trans. Netw.*, 24(3):1421–1433, 2016.