Fully Dynamic Strongly Connected Components in Planar Digraphs

Adam Karczmarz
University of Warsaw, Poland
IDEAS NCBR, Warsaw, Poland

Marcin Smulewicz
University of Warsaw, Poland

Abstract

In this paper we consider maintaining strongly connected components (SCCs) of a directed planar graph subject to edge insertions and deletions. We show a data structure maintaining an implicit representation of the SCCs within $\widetilde{O}(n^{6/7})$ worst-case time per update. The data structure supports, in $O(\log^2 n)$ time, reporting vertices of any specified SCC (with constant overhead per reported vertex) and aggregating vertex information (e.g., computing the maximum label) over all the vertices of that SCC. Furthermore, it can maintain global information about the structure of SCCs, such as the number of SCCs, or the size of the largest SCC.

To the best of our knowledge, no fully dynamic SCCs data structures with sublinear update time have been previously known for any major subclass of digraphs. Our result should be contrasted with the $n^{1-o(1)}$ amortized update time lower bound conditional on SETH, which holds even for dynamically maintaining whether a general digraph has more than two SCCs.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases dynamic strongly connected components, dynamic strong connectivity, dynamic reachability, planar graphs

Digital Object Identifier 10.4230/LIPIcs.ICALP.2024.95

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://arxiv.org/abs/2406.10420`

Funding *Adam Karczmarz*: Partially supported by the ERC CoG grant TUgbOAT no 772346 and the National Science Centre (NCN) grant no. 2022/47/D/ST6/02184.

1 Introduction

Two vertices of a directed graph $G = (V, E)$ are called strongly connected if they can reach each other using paths in G. Pairwise strong connectivity is an equivalence relation and the strongly connected components (SCCs) of G are its equivalence classes. Computing the SCCs is among the most classical and fundamental algorithmic problems on digraphs and there exists a number of linear-time algorithms for that [14, 33, 35]. Therefore, it is no surprise that *maintaining* SCCs has been one of the most actively studied problems on *dynamic directed graphs* [1, 2, 3, 4, 5, 7, 8, 18, 23, 25, 30, 31].

When maintaining the strongly connected components, the information we care about may vary. First, we could be interested in efficiently answering *pairwise* strong connectivity queries: given $u, v \in V$, decide whether u and v are strongly connected. Pairwise strong connectivity queries, however, cannot easily provide any information about the *global* structure of SCCs (such as the number of SCCs, the size of the largest SCC). Neither they enable, e.g., listing the vertices strongly connected to some $u \in V$. This is why, in the following, we distinguish between *dynamic pairwise strong connectivity* and *dynamic SCCs* data structures which

51st International Colloquium on Automata, Languages, and Programming (ICALP 2024).
Editors: Karl Bringmann, Martin Grohe, Gabriele Puppis, and Ola Svensson;
Article No. 95; pp. 95:1–95:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

provide a more global view. In particular, all the data about the SCCs can be easily accessed if the SCCs are maintained *explicitly*, e.g., if the SCC identifier of every vertex is stored at all times and explicitly updated.

1.1 Previous work

In the following, let $n = |V|$ and $m = |E|$. Dynamic graph data structures are traditionally studies in *incremental*, *decremental* or *fully dynamic* settings, which permit the graph to evolve by either only edge insertions, only deletions, or both, respectively. A decremental data structure maintaining SCCs with near-optimal total update time is known [5]. Very recently, a deterministic data structure with $m^{1+o(1)}$ total update time has been obtained also for the incremental setting [8]. Both these state-of-the art data structures maintain the SCCs explicitly.

The *fully dynamic* variant – which is our focus in this paper – although the most natural, has been studied the least. First of all, there is strong evidence that a non-trivial dynamic SCCs data structure for sparse graphs cannot exist. If the SCCs have to be maintained explicitly, then a single update can cause a rather dramatic $\Omega(n)$-sized amortized change in the set of SCCs[1]. As a result, an explicit update may be asymptotically as costly as recomputing SCCs from scratch. This argument – applicable also for maintaining *connected components* of an undirected graph – does not exclude the possibility of maintaining an *implicit* representation of the SCCs in sublinear time, though. After all, there exist very efficient fully dynamic connectivity data structures, e.g., [20, 21, 38], typically maintaining also an explicit spanning forest which allows retrieving any "global" component-wise information one can think of rather easily. However, Abboud and Vassilevska Williams [1] showed that even for maintaining a single-bit information whether G has *more than two SCCs*, a data structure with $O(n^{1-\epsilon})$ amortized time is unlikely, as it would break the Orthogonal Vectors conjecture implied by the SETH [22, 37].[2] This considerably limits the possible global information about the SCCs that can be maintained within sublinear time per update.

For denser graphs, Abboud and Vassilevska Williams [1] also proved that maintaining essentially any (even pairwise) information about SCCs dynamically within truly subquadratic update time has to rely on fast matrix multiplication. And indeed, that pairwise strong connectivity can be maintained this way follows easily from the dynamic matrix inverse-based dynamic *st*-reachability data structures [36, 32]. More recently, we [25] showed that in fact SCCs can be maintained explicitly in $O(n^{1.529})$ worst-case time per update. They also proved that maintaining whether G has just a single SCC (*dynamic SC*) is easier[3] and can be achieved within $O(n^{1.406})$ worst-case time per update. Both these bounds are tight conditional on the appropriate variants [36] of the OMv conjecture [19].

In summary, the complexity of maintaining SCCs in general directed graphs is rather well-understood now. In partially dynamic settings, the known bounds are near optimal unconditionally, whereas in the fully dynamic setting, the picture appears complete unless some popular hardness conjectures are proven wrong. In particular, for general sparse digraphs, no (asymptotically) non-trivial fully dynamic SCCs data structure can exist.

---

[1] Consider a directed cycle and switching its arbitrary single edge on and off.
[2] In [1], a conditional lower bound of the same strength is also derived for the dynamic #SSR problem where the goal is to dynamically count vertices reachable from a source $s \in V$.
[3] Interestingly, the SETH-based lower bound of [1] does not apply to the dynamic SC problem.

Planar graphs. It is thus natural to ask whether non-trivial dynamic SCCs data structures are possible if we limit our attention to some significant class of sparse digraphs. And indeed, this question has been partially addressed for *planar digraphs* in the past. Since pairwise s,t-strong connectivity queries reduce to two s,t-reachability queries, the known planar dynamic reachability data structures [9, 34] imply that sublinear ($\widetilde{O}(n^{2/3})$ or $\widetilde{O}(\sqrt{n})$-time, depending on whether embedding-respecting insertions are required) updates/queries are possible for *pairwise* strong connectivity. Another trade-off for dynamic pairwise strong connectivity has been showed by Charalampopoulos and Karczmarz [6]. Namely, they showed a fully dynamic data structure for planar graphs with $\widetilde{O}(n^{4/5})$ worst-case update time that can produce an identifier s_v of an SCC of a given query vertex v in $O(\log^2 n)$ time. Whereas this is slightly more general[4], it still not powerful enough to enable efficiently maintaining any of the global data about the SCCs of a dynamic planar digraph such as the SCCs count.

To the best of our knowledge, the question whether a more robust – that is, giving a more "global" perspective on the SCCs beyond only supporting pairwise queries – fully dynamic SCCs data structure for planar digraphs (or digraphs from any other interesting class) with sublinear update time is possible has not been addressed before.

1.2 Our results

In this paper, we address the posed question in the case of planar directed graphs. Specifically, our main result is a dynamic SCCs data structure summarized by the following theorem.

▶ **Theorem 1.** *Let G be a planar digraph subject to planarity-preserving edge insertions and deletions. There exists a data structure maintaining the strongly connected components of G implicitly in $\widetilde{O}(n^{6/7})$ worst-case time per update. Specifically:*

- *The data structure maintains the number of SCCs and the size of the largest SCC in G.*
- *For any query vertex v, in $O(\log^2 n)$ time the data structure can compute the size of the SCC of v, and enable reporting the elements of the SCC of v in $O(1)$ worst-case time per element.*

In particular, Theorem 1 constitutes the first known fully dynamic SCCs data structure with sublinear update time for any significant class of sparse digraphs. It also shows that the conditional lower bound of [1] does not hold in planar digraphs.

The data structure of Theorem 1 is deterministic and does not require the edge insertions to respect any fixed embedding of the graph (this also applies to side results discussed below). Obtaining more efficient data structures for fully dynamic embedding-respecting updates is an interesting direction (see, e.g., [9]) that is beyond the scope of this paper.

Related problems. Motivated by the discrepancies between the known bounds for dynamic SCCs and dynamic SC in general digraphs (both from the lower- [1] and upper bounds [25] perspective), we also complement Theorem 1 with a significantly simpler and faster data structure suggesting that the dynamic SC might be easier (than dynamic SCCs) in planar digraphs as well.[5]

[4] Than answering pairwise strong connectivity queries. Using the SCC-identifiers, one can, e.g., partition any k vertices of G into strongly connected classes in $\widetilde{O}(k)$ time, whereas using pairwise queries this requires $\Theta(k^2)$ queries.

[5] Clearly, one could use Theorem 1 for dynamic SC as well.

▶ **Lemma 2.** *Let G be a planar digraph subject to planarity-preserving edge insertions and deletions. One can maintain whether G has a single SCC in $\widetilde{O}(n^{2/3})$ worst-case time per update.*

Similarly, one could ask how *dynamic #SSR* (i.e., counting vertices reachable from a single source) relates to dynamic SCCs in planar digraphs. Especially since:

(1) in general directed graphs, dynamic SCCs and dynamic #SSR currently have matching lower- [1, 36] and upper bounds [25, 32] (up to polylogarithmic factors);

(2) the former problem is at least as hard as the latter in the sense that dynamic #SSR reduces to dynamic SCCs in general graphs easily[6], whereas an opposite reduction is not known.

Unfortunately, the aforementioned reduction of dynamic #SSR to dynamic SCCs breaks planarity rather badly. Interestingly, the *path net* technique we develop to obtain Theorem 1 does not seem to work for counting "asymmetric" reachabilities from a single source.

Nevertheless, the Voronoi diagram machinery developed for computing the diameter of a planar graph [15] almost immediately yields a more efficient data structure for dynamic #SSR in planar digraphs with $\widetilde{O}(n^{4/5})$ update time. We provide the details of that construction in the full version of this extended abstract.

It is worth noting that Voronoi diagrams-based techniques (as developed for distance oracles [16]) have been used in the pairwise strong connectivity data structure [6]. However, as we discuss later on, it is not clear how to apply those for the dynamic SCCs problem. This is why Theorem 1 relies on a completely different path net approach developed in this paper.

1.3 Organization

We review some standard planar graph tools in Section 2. Then, as a warm-up, we show the data structure for dynamic SC in Section 3. In Section 4 we define a path net data structure and show how it can be used to obtain a dynamic SCCs data structure. Finally, in Section 5 we describe the path net data structure. Due to space limit, some details and proofs are deferred to the full version.

2 Preliminaries

In this paper we deal with *directed* graphs. We write $V(G)$ and $E(G)$ to denote the sets of vertices and edges of G, respectively. We omit G when the graph in consideration is clear from the context. A graph H is a *subgraph* of G, which we denote by $H \subseteq G$, iff $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. We write $e = uv \in E(G)$ when referring to edges of G. By G^{R} we denote G with edges reversed.

A sequence of vertices $P = v_1 \dots v_k$, where $k \geq 1$, is called an $s \to t$ path in G if $s = v_1$, $v_k = t$ and there is an edge $v_i v_{i+1}$ in G for each $i = 1, \dots, k-1$. We sometimes view a path P as a subgraph of G with vertices $\{v_1, \dots, v_k\}$ and (possibly zero) edges $\{v_1v_2, \dots, v_{k-1}v_k\}$. For convenience, we sometimes consider a single edge uv a path. If P_1 is a $u \to v$ path and P_2 is a $v \to w$ path, we denote by $P_1 \cdot P_2$ (or simply P_1P_2) a path obtained by concatenating P_1 with P_2. A vertex $t \in V(G)$ is *reachable* from $s \in V(G)$ if there is an $s \to t$ path in G.

[6] Consider the graph G' obtained from G by adding a supersink t with a single outgoing edge ts and incoming edges vt for all $v \in V$. Then $v \in V$ is reachable from s in G iff s and v are strongly connected in G'. See also [36].

Planar graph toolbox. An r-*division* [13] $\mathcal{R}$ of a planar graph, for $r \in [1, n]$, is a decomposition of the graph into a union of $O(n/r)$ pieces P, each of size $O(r)$ and with $O(\sqrt{r})$ boundary vertices (denoted ∂P), i.e., vertices shared with some other piece of $\mathcal{R}$. We denote by $\partial\mathcal{R}$ the set $\bigcup_{P\in\mathcal{R}} \partial P$. If additionally G is plane-embedded, all pieces are connected, and the boundary vertices of each piece P of the r-division $\mathcal{R}$ are distributed among $O(1)$ faces of P that contain the vertices from ∂P *exclusively* (also called holes of P), we call $\mathcal{R}$ an r-*division with few holes*. Klein [27] showed that an r-division with few holes of a triangulated graph can be computed in linear time.

Fully dynamic r-divisions. Many dynamic algorithms for planar graphs maintain r-divisions and useful piecewise auxiliary data structures under dynamic updates. Let us slightly generalize the definition of an r-division with few holes to non-planar graphs by dropping the requirement that G as a whole is planar but retaining all the other requirements (in particular, the individual pieces are plane-embedded).

▶ **Theorem 3** ([6, 28, 34])**.** *Let $G = (V, E)$ be a weighted planar graph that undergoes edge deletions and edge insertions (assumed to preserve the planarity of G). Let $r \in [1, n]$.*

There is a data structure maintaining an r-division with few holes $\mathcal{R}$ of some G^+, where G^+ can be obtained from G by adding edges[7]*, such that each piece $P \in \mathcal{R}$ is accompanied with some auxiliary data structures that can be constructed in $T(r)$ time given P and use $S(r)$ space.*

The data structure uses $O\left(n + \frac{n}{r} \cdot S(r)\right)$ space and can be initialized in $O\left(n + \frac{n}{r} \cdot T(r)\right)$ time. After each edge insertion/deletion, it can be updated in $O(r + T(r))$ worst-case *time.*

3 Fully dynamic SC data structure

To illustrate the general approach and introduce some of the concepts used for obtaining Theorem 1, in this section we first prove Lemma 2. That is, we show that the information whether a planar graph G is strongly connected can be maintained in $\widetilde{O}(n^{2/3})$ time per update.

We build upon the following general template used previously for designing fully dynamic data structures supporting reachability, strong connectivity, and shortest paths queries in planar graphs, e.g., [34, 28, 11, 6, 24]. As a base, we will maintain dynamically an r-division with few holes $\mathcal{R}$ of G using Theorem 3 with auxiliary piecewise data structures to be fixed later. Intuitively, as long as the piecewise data structures are powerful enough to allow recomputing the requested graph property (e.g., strong connectivity, shortest path between a fixed source/target pair) while spending $r^{1-\epsilon}$ time per piece, for some choice of r we get a sublinear update bound of $\widetilde{O}(n/r^{\epsilon} + r + T(r))$. For example, if $T(r) = O(r^{9.9})$ and $\epsilon = 0.1$, for $r = n^{0.1}$ we get $\widetilde{O}(n^{0.99})$ worst-case update time bound.

Reachability certificates. Subramanian [34] described *reachability certificates* that sparsify reachability between a subset of vertices lying on $O(1)$ faces of a plane digraph G into a (non-necessarily planar) digraph of size near-linear in the size of the subset in question. Formally, we have the following.

[7] Note that G^+ need not be planar.

▶ **Lemma 4** ([34]). *Let H be a plane digraph with a distinguished set $\partial H \subseteq V(H)$ lying on some $O(1)$ faces of H. There exists a directed graph X_H, where $\partial H \subseteq V(X_H)$, of size $\widetilde{O}(|\partial H|)$ satisfying the following property: for any $u, v \in \partial H$, a path $u \to v$ exists in H if and only if there exists a $u \to v$ path in X_H. The graph X_H can be computed in $\widetilde{O}(|H|)$ time.*

▶ Remark 5. For Lemma 4 to hold, it is enough that ∂H lies on $O(1)$ Jordan curves in the plane, each of them having the embedding of H entirely (but not strictly) on one side of the curve. In particular, it is enough that ∂H lies on $O(1)$ faces of some plane supergraph H' with $H \subseteq H'$.

Roughly speaking, Subramanian [34] uses reachability certificates as auxiliary data structures in Theorem 3 in order to obtain a fully dynamic reachability data structure. Crucially, the union of the piecewise certificates preserves pairwise reachability between the boundary vertices $\partial\mathcal{R}$, or more formally (see e.g. [6] for a proof):

▶ **Lemma 6.** *For any $u, v \in \partial\mathcal{R}$, u can reach v in G if and only if u can reach v in $X = \bigcup_{P\in\mathcal{R}} X_P$.*

Strong connectivity data structure. The union of certificates X preserves reachability, and thus strong connectivity between the vertices $\partial\mathcal{R} := \bigcup_{P\in\mathcal{R}} \partial P$. As a result, if G is strongly connected, then so is $\partial\mathcal{R}$ in X. But the reverse implication might not hold. It turns out that for connected graphs, to have an equivalence, it is enough to additionally maintain, for each piece P, whether P is strongly connected conditioned on whether ∂P is strongly connected in G.

In the following, we give a formal description of the data structure. As already said, the data structure maintains a dynamic r-division $\mathcal{R}^+$ of a supergraph G^+ of G (i.e., the input graph), as given by Theorem 3. Since $G \subseteq G^+$, the pieces $\{P^+ \cap G : P^+ \in \mathcal{R}^+\}$ induce an r-division $\mathcal{R}$ of G; however, the boundary ∂P of a piece $P \in \mathcal{R}$ does not necessarily lie on $O(1)$ faces of P, so $\mathcal{R}$ is not technically an r-division with few holes. Nevertheless, ∂P still lies on $O(1)$ faces of a plane supergraph P^+ of P that do not contain vertices outside ∂P. Consequently, by Remark 5, we can still use Lemma 4 to construct a sparse reachability certificate for the piece $P \in \mathcal{R}$. For obtaining Lemma 2, we do not require anything besides beyond that, so for simplicity and wlog. we can assume we work with $\mathcal{R}$ instead of $\mathcal{R}^+$.

While $\mathcal{R}$ evolves, each piece P is accompanied with a reachability certificate X_P of Lemma 4. Note that since $|\partial P| = O(\sqrt{r})$, X_P has size $\widetilde{O}(\sqrt{r})$ and can be constructed in $\widetilde{O}(r)$ time. Moreover, for each P, let $C_{\partial P}$ be a directed simple cycle on the vertices ∂P. We additionally store the (1-bit) information whether the graph $P \cup C_{\partial P}$ is strongly connected. Clearly, this can be computed in $O(|P|) = O(r)$ time. All the accompanying data structures of a piece $P \in \mathcal{R}$ can be thus constructed in $\widetilde{O}(r)$ time. Therefore, by Theorem 3, they are maintained in $\widetilde{O}(r)$ time per update.

Finally, in a separate data structure, we maintain whether G is connected (in the undirected sense). This can be maintained within $n^{o(1)}$ worst-case update time deterministically even in general graphs [17]; in our case, also a less involved data structure such as [12] would suffice.

After $\mathcal{R}$ and the accompanying data structures are updated, strong connectivity of G can be verified as follows. First of all, the union X of all X_P, $P \in \mathcal{R}$, is formed. Note that we can test whether the vertices $\partial\mathcal{R}$ are strongly connected in X in $O(|X|) = \widetilde{O}(n/\sqrt{r})$ time by computing the strongly connected components $\mathcal{S}_X$ of X using any classical linear time algorithm. If G is not connected, or $\partial\mathcal{R}$ is not strongly connected in X, we declare G not strongly connected. If, on the other hand, $\partial\mathcal{R}$ is strongly connected in X, we simply check whether $P \cup C_{\partial P}$ is strongly connected for each $P \in \mathcal{R}$ and if so, declare G strongly connected. This takes $O(n/r)$ time. Thus, testing strong connectivity takes $\widetilde{O}(n/\sqrt{r})$ time. The following lemma establishes the correctness.

▶ **Lemma 7.** *G is strongly connected if and only if G is connected, the vertices $\partial\mathcal{R}$ are strongly connected in X, and for all $P \in \mathcal{R}$, $P \cup C_{\partial P}$ is strongly connected.*

Proof. First suppose that G is strongly connected. Then, G is clearly connected. Moreover, by Lemma 6, $\partial\mathcal{R}$ is strongly connected in X. For contradiction, suppose that for some $P \in \mathcal{R}$ and $u, v \in V(P)$, u cannot reach v in $P \cup C_{\partial P}$. By strong connectivity of G, there exists some path $Q = u \to v$ in G. Since u cannot reach v in P, Q is not fully contained in P. As a result, Q can be expressed as $Q_1 \cdot R \cdot Q_2$, where $Q_1 = u \to a, Q_2 = b \to v$ are fully contained in P, and $a, b \in \partial P$. But there is a path $Z = a \to b$ in $C_{\partial P}$, so there is a $u \to v$ path $Q_1 \cdot Z \cdot Q_2$ in $P \cup C_{\partial P}$, a contradiction.

Now consider the " $\Longleftarrow$ " direction. Suppose G is connected, the vertices $\partial\mathcal{R}$ are strongly connected in X, and for all $P \in \mathcal{R}$, $P \cup C_{\partial P}$ is strongly connected. By Lemma 6, $\partial\mathcal{R}$ is strongly connected in G. Consider any $P \in \mathcal{R}$ and let $x, y \in V(P)$. We first prove that there exists a path $x \to y$ in G. Indeed, if an $x \to y$ path exists in P, it also exists in G. Otherwise, since $P \cup C_{\partial P}$ is strongly connected, there exists a path $Q = x \to y$ in $P \cup C_{\partial P}$ that can be expressed as $Q_1 \cdot R \cdot Q_2$, where $Q_1 = x \to a$ and $Q_2 = b \to y$ are fully contained in P and $a, b \in \partial P$. But since $a, b \in \partial\mathcal{R}$, by strong connectivity of $\partial\mathcal{R}$, there exists a path $R' = a \to b$ in G. Since $Q_1, Q_2 \subseteq G$, $Q_1 \cdot R' \cdot Q_2$ is an $x \to y$ path in G.

Now take arbitrary $u, v \in V(G)$. If there exists a piece in $\mathcal{R}$ containing both u and v, then we have already proved that there exists a path $u \to v$ in G. Otherwise, let P_u, P_v, $P_u \neq P_v$, be some pieces of $\mathcal{R}$ containing u, v, respectively. We have $P_u \neq G$ and $P_v \neq G$. Since G is connected, P_u has at least one boundary vertex $a \in \partial P_u$. Similarly, P_v has at least one boundary vertex $b \in \partial P_v$. We have proved that there exist paths $u \to a$ and $b \to v$ in G. But also $a, b \in \partial\mathcal{R}$, by the strong connectivity of $\partial\mathcal{R}$, there exists a path $a \to b$ in G as well. We conclude that there exists a path $u \to v$ in G. Since u, v were arbitrary, G is indeed strongly connected. ◀

The worst-case update time of the data structure is $\widetilde{O}(r + n/\sqrt{r}) + n^{o(1)}$. By setting $r = n^{2/3}$, we obtain Lemma 2.

4 Dynamic strongly connected components

The approach we take for maintaining strong connectivity in planar graphs does not easily generalize even to dynamic SCCs counting. This is the case for the following reason. Even if the piece P is fixed (static), there can be possibly an exponential number of different assignments of the vertices ∂P to the SCCs in G (when the other pieces are subject to changes), whereas for dynamic SC, a non-trivial situation arises only when all of ∂P lies within a single SCC. In order to achieve sublinear update time, for any assignment we need to be able to count the SCCs fully contained in P in time sublinear in r after preprocessing P in only *polynomial* (and not exponential) time.

The following notion will be crucial for all our developments.

▶ **Definition 8.** *Let $P \in \mathcal{R}$, and let $A \subseteq \partial P$. A* path net *$\Pi_P(A)$ induced by A is the set of vertices of P that lie on some directed path in P connecting some two elements of A.*

In other words, the path net $\Pi_P(A)$ contains vertices $v \in V(P)$ such that v can reach A and can be reached from A in P. We call a path net $\Pi_P(A)$ *closed* if $A = \Pi_P(A) \cap \partial P$, that is, there are no boundary vertices of P outside A that can reach and can be reached from A.

The following key lemma relates a piece's path net to the SCCs of G.

▶ **Lemma 9.** *Let S be an SCC of G containing at least one boundary vertex of P, i.e., $S \cap \partial P \neq \emptyset$. Then the path net $\Pi_P(S \cap \partial P)$ is closed and equals $S \cap V(P)$.*

Proof. Let us first argue that $\Pi_P(S \cap \partial P)$ is closed. If it was not, there would exist $b \in \partial P \setminus S$ such that there exist paths $b \to (S \cap \partial P)$ and $(S \cap \partial P) \to b$ in P. It follows that b can reach and be reached from S in G, i.e., b is strongly connected with S. Hence, $b \in S$, a contradiction.

Let $v \in \Pi_P(S \cap \partial P)$. Since v can reach and can be reached from $S \cap \partial P$ in P, then it is indeed strongly connected with S in G, since the vertices S are strongly connected in G. So $v \in S \cap V(P)$.

Now let $v \in S \cap V(P)$. Pick any $b \in S \cap \partial P$ (possibly $b = v$ if $v \in \partial P$). There exists paths $R = v \to b$ and $Q = b \to v$ in G. Note that R has some prefix $R_1 = v \to a$ that is fully contained in P and $a \in \partial P$. Similarly, Q has a suffix $Q_1 = c \to v$ that is fully contained in P and $c \in \partial P$. Since there exists paths $v \to a$, $a \to b$, $b \to c$, $c \to v$ in G, vertices a, b, c are strongly connected in G. So $a, c \in S \cap \partial P$. The paths R_1, Q_1 certify that v can be reached from and can reach $S \cap \partial P$ in P. Therefore, $v \in \Pi_P(S \cap \partial P)$ as desired. ◀

If an SCC S is as in Lemma 9, then since the vertices ∂P might be shared with other pieces of $\mathcal{R}$, $\Pi_P(S \cap \partial P) \setminus \partial P$ constitutes the vertices of S contained *exclusively in the piece* P. As there are only $\widetilde{O}(n/\sqrt{r})$ boundary vertices through all pieces, their affiliation to the SCCs of G can be derived from the (SCCs of the) certificate graph $X = \bigcup_{P \in \mathcal{R}} X_P$ (defined and maintained as in Section 3), i.e., they may be handled efficiently separately. Consequently, being able to efficiently aggregate labels, or report the elements, of the sets of the form $\Pi_P(A) \setminus \partial P$ (where $\Pi_P(A)$ is closed) is the key to obtaining an efficient implicit representation of the SCCs of G, as claimed in Theorem 1. Our main technical contribution (Theorem 10) is a *path net data structure* enabling precisely that. The data structure requires a rather large $\widetilde{O}(r^3)$ preprocessing time but achieves the goal by supporting queries about $A \subseteq \partial P$ in near-optimal $\widetilde{O}(|A|)$ time. Formally, we show:

▶ **Theorem 10.** *Let $P \in \mathcal{R}$ and let $\alpha : V(P) \to \mathbb{R}$ be a weight function. In $\widetilde{O}(r^3)$ time one can construct a data structure supporting the following queries. Given a subset $A \subseteq \partial P$, such that $\Pi_P(A)$ is closed, in $\widetilde{O}(|A|)$ time one can:*
- *create an* iterator *that enables listing elements of $\Pi_P(A) \setminus \partial P$ in $O(1)$ time per element,*
- *aggregate weights over $\Pi_P(A) \setminus \partial P$, i.e., compute $\sum_{v \in \Pi_P(A) \setminus \partial P} \alpha(v)$.*

▶ Remark 11. We do not require using subtractions to compute the aggregate weights. In fact, the data structure of Theorem 10 can be easily modified to aggregate weights coming from any semigroup, e.g., one can compute the max/min weight in $\Pi_P(A) \setminus \partial P$ within these bounds.

Our high-level strategy is to maintain the certificates and path net data structures accompanying individual pieces along with the r-division. Roughly speaking, to obtain the information about the SCCs of G beyond how the partition of $\partial\mathcal{R}$ into SCCs looks like, we will query the path net data structures for each piece P with the sets A equal to the SCCs of X having non-empty intersection with ∂P. We prove Theorem 10 later on, in Section 5.

In the remaining part of the section, we explain in detail how, equipped with Theorem 10, a dynamic (implicit) strongly connected components data structure can be obtained. As in Section 3, we maintain an r-division $\mathcal{R}$ dynamically, and maintain sparse reachability certificates X_P, along with the set $\mathcal{S}_X$ of SCCs of $X = \bigcup_{P \in \mathcal{R}} X_P$. Moreover, for each P

we store the strongly connected components $\mathcal{S}_P$ of P. Let $\mathcal{S}_{\partial P}$ be the elements of $\mathcal{S}_P$ that contain a boundary vertex, and $\mathcal{S}_{P\setminus\partial P}$ the elements of $\mathcal{S}_P$ that do not. Clearly, we have $\mathcal{S}_P = \mathcal{S}_{\partial P} \cup \mathcal{S}_{P\setminus\partial P}$ and $(\bigcup \mathcal{S}_P) \cap (\bigcup \mathcal{S}_{P\setminus\partial P}) = \emptyset$.

For each piece $P \in \mathcal{R}$, we additionally store a path net data structure $\mathcal{D}_P$ of Theorem 10 with an appropriately defined weight function (to be picked depending on the application later). Note that for a piece P, all the auxiliary data structures accompanying P that we have defined can be computed in $\widetilde{O}(r^3)$ time. We now consider the specific goals that can be achieved this way.

Finding the largest SCC. Denote by S^* the largest SCC of G. To be able to identify S^*, and e.g., compute its size, we additionally store and maintain the following. For each piece P, we also maintain the largest SCC S^*_P of P. The sizes of all the SCCs of P, in particular the size of S^*_P, can be easily found and stored after computing $\mathcal{S}_P$.

Note that if the largest SCC S^* of G is not contained entirely in any individual piece P (and thus is larger than $\max_{P\in\mathcal{R}} |S^*_P|$), it has to intersect $\partial\mathcal{R}$. More specifically, in this case for each piece P such that $S^* \cap V(P) \neq \emptyset$, we have $S^* \cap \partial P \neq \emptyset$.

Recall that by Lemma 6, $X = \bigcup_{P\in\mathcal{R}} X_P$ preserves the strong connectivity relation on the vertices $\partial\mathcal{R}$. Therefore, if S^* intersects $\partial\mathcal{R}$, it has to contain $B \cap \partial\mathcal{R}$ for some SCC B of X. For any such B, we can compute the size of the SCC S_B of G satisfying $B \cap \partial\mathcal{R} \subseteq S_B$ as follows. First of all, $|S_B \cap \partial\mathcal{R}| = |B \cap \partial\mathcal{R}|$ since B is an SCC of X. It is thus enough to compute, for all $P \in \mathcal{R}$, $|S_B \cap (V(P) \setminus \partial P)|$. Since the sets $V(P) \setminus \partial P$ are pairwise disjoint across the pieces, by adding these values, we will get the desired size $|S_B|$.

We have already argued that if $B \cap \partial P = \emptyset$, then $S_B \cap V(P) = \emptyset$. If, on the other hand, $B\cap\partial P$ is non-empty, by Lemma 9, if we use the weight function $\alpha(v) \equiv 1$ in the piecewise data structures $\mathcal{D}_P$ of Theorem 10, we can compute $|S_B \cap (V(P) \setminus \partial P)| = \sum_{v\in\Pi_P(B\cap\partial P)\setminus\partial P} \alpha(v)$ in $\widetilde{O}(|B \cap \partial P|)$ time using the input set $A := B \cap \partial P$. We conclude that the sizes S_B for all $B \in \mathcal{S}_X$ can be computed in time

$$\widetilde{O}\left(\sum_{B\in\mathcal{S}_X} \sum_{\substack{P\in\mathcal{R}\\ B\cap\partial P\neq\emptyset}} |B\cap\partial P|\right) = \widetilde{O}\left(\sum_{P\in\mathcal{R}} \sum_{\substack{B\in\mathcal{S}_X\\ B\cap\partial P\neq\emptyset}} |B\cap\partial P|\right) = \widetilde{O}\left(\sum_{P\in\mathcal{R}} |\partial P|\right) = \widetilde{O}(n/\sqrt{r}). \tag{1}$$

Finally, S^* is either equal to the largest S_B for $B \in \mathcal{S}_X$, or the largest S^*_P (through $P \in \mathcal{R}$). The latter is the case if $\max_{B\in\mathcal{S}_X} |S_B| < \max_{P\in\mathcal{R}} |S^*_P|$. Which case we fall into is easily decided once all the $O(n/\sqrt{r})$ sizes $|S_B|$ are computed.

Accessing the SCC of a specified vertex. Suppose first that we know the SCC S_v of G containing a query vertex v, and additionally whether S_v intersects $\partial\mathcal{R}$. If $S_v \cap \partial\mathcal{R} = \emptyset$, then v is a vertex of a unique piece P, and $S_v \in \mathcal{S}_{P\setminus\partial P}$. In this case, we can clearly compute the size of S_v and report the elements of S_v in $O(1)$ time since $S_v \in \mathcal{S}_P$ is stored explicitly.

Otherwise, if $S_v \cap \partial\mathcal{R} \neq \emptyset$, we reuse the information that we have computed for finding the largest SCC. We have already described how to compute the sizes of all the SCCs S of G intersecting $\partial\mathcal{R}$ (in particular S_v) along with these respective intersections in $\widetilde{O}(n/\sqrt{r})$ time upon update. Moreover, for all $P \in \mathcal{R}$ we have computed $|S \cap (V(P) \setminus \partial P)|$ using the data structure $\mathcal{D}_P$ of Theorem 10. As a result, we can store, for each such S, a subset $L(S) \subseteq \mathcal{R}$ of pieces P such that $S \cap (V(P) \setminus \partial P) \neq \emptyset$. Recall that for all $P \in L(S)$, we can also use $\mathcal{D}_P$ to create an iterator for reporting the elements of $S \cap (V(P) \setminus \partial P)$ in $O(1)$ time per element. So, in order to efficiently report elements of any such S, we first report the vertices of $S \cap \partial\mathcal{R}$, and then the elements of each $S \cap (V(P) \setminus \partial P)$ for subsequent pieces $P \in L(S)$. Indeed, one needs only $O(1)$ worst-case time to find each subsequent vertex of S.

Finally, we are left with the task of of finding the SCC S_v of a query vertex $v \in V$. It is not clear how to leverage path net data structures for that in the case when S_v intersects $\partial\mathcal{R}$. Instead, we use the data structure of [6] in a black-box way. That data structure handles fully dynamic edge updates in $\widetilde{O}(n^{4/5})$ worst-case time, and provides $O(\log^2 n)$ worst-case time access to consistent (for queries issued between any two subsequent updates) SCC identifiers of individual vertices[8]. Using [6], after every update we find the identifiers I of the SCCs of vertices $\partial\mathcal{R}$ in G in $\widetilde{O}(n/\sqrt{r})$ time. Now, to find the SCC of v upon query, we find the identifier i_v of the SCC containing v in $O(\log^2 n)$ time. If $i_v \in I$ (which can be tested in $O(\log n)$ time), we obtain that v is in an SCC of G intersecting $\partial\mathcal{R}$ and some vertex b_v from the intersection. b_v can be in turn used to access $L(S_v)$ and thus enable reporting the elements of S_v. Otherwise, if $i_v \notin I$, $S \cap \partial\mathcal{R} = \emptyset$ and thus S equals the unique SCC from $S_{P\setminus\partial P}$ containing v in the unique piece P containing v.

Counting strongly connected components. Let us separately count SCCs $\mathcal{S}_{\partial\mathcal{R}}$ that intersect $\partial\mathcal{R}$ and those that do not. The former can be counted in $\widetilde{O}(n/\sqrt{r})$ time by counting the SCCs of X that intersect $\partial\mathcal{R}$ (that we maintain). The latter can be computed as follows. Consider the sum $\Phi = \sum_{P\in\mathcal{R}} |\mathcal{S}_{P\setminus\partial P}|$. If we wanted the sum Φ to count the SCCs not intersecting $\partial\mathcal{R}$, then an SCC $S \in \mathcal{S}_{P\setminus\partial P}$ contributes to the sum unnecessarily precisely when S is not an SCC of G. To see that, note that if an SCC S of P is not an SCC of G, it has to be a part of another SCC S' of G that also contains vertices of other pieces, i.e., S' intersects $\partial\mathcal{R}$. From Lemma 9, we conclude:

► **Corollary 12.** *An SCC $S \in \mathcal{S}_{P\setminus\partial P}$ is not an SCC of G iff there exists (precisely one) SCC B of X such that $S \subseteq \Pi_P(B \cap \partial P)$ (or equivalently, such that $S \cap \Pi_P(B \cap \partial P) \neq \emptyset$).*

As a result, we can count the number of SCCs in G that do not intersect $\partial\mathcal{R}$ by subtracting from Φ, for each $P \in \mathcal{R}$, and each $B \in \mathcal{S}_X$ the number $c_{P,B}$ of SCCs in $\mathcal{S}_{P\setminus\partial P}$ that intersect $\Pi_P(B \cap \partial P)$. To this end, we can use the data structure $\mathcal{D}_P$ of Theorem 10 built upon P (and maintained as described before) with a weight function α on $V(P)$ assigning 1 to an arbitrary single vertex v_S of each SCC $S \in \mathcal{S}_{P\setminus\partial P}$, and 0 to all other vertices. By Corollary 12, with such a weight function, $c_{P,B} = \sum_{v\in\Pi_P(B\cap\partial P)\setminus\partial P} \alpha(v)$ can be computed in $\widetilde{O}(|B \cap \partial P|)$ time using $\mathcal{D}_P$. Consequently, similarly as in (1), over all $B \in \mathcal{S}_X$, and $P \in \mathcal{R}$, computing all the values $c_{P,B}$ will take $\widetilde{O}(n/\sqrt{r})$ time. As mentioned before, the SCC count is obtained by subtracting those from Φ and adding the result to the count of $\mathcal{S}_{\partial\mathcal{R}}$.

Depending on the application, the worst-case update time of the data structure is $\widetilde{O}(n/\sqrt{r} + r^3)$ or $\widetilde{O}(n/\sqrt{r} + r^3 + n^{4/5})$. The bound is optimized for $r = n^{2/7}$ and this yields Theorem 1.

5 The path net data structure

This section is devoted to describing the below key component of our dynamic SCCs data structure.

---

[8] The query time of that data structure can be easily reduced to $O(\log n \cdot \log\log n)$ without affecting the $\widetilde{O}(n^{4/5})$ update bound if one simply replaces the classical MSSP data structure [26] used internally for performing point location queries in additively weighted Voronoi diagrams [16] with the MSSP data structure of [29].

▶ **Theorem 10.** *Let $P \in \mathcal{R}$ and let $\alpha : V(P) \to \mathbb{R}$ be a weight function. In $\widetilde{O}(r^3)$ time one can construct a data structure supporting the following queries. Given a subset $A \subseteq \partial P$, such that $\Pi_P(A)$ is closed, in $\widetilde{O}(|A|)$ time one can:*

- *create an* iterator *that enables listing elements of $\Pi_P(A) \setminus \partial P$ in $O(1)$ time per element,*
- *aggregate weights over $\Pi_P(A) \setminus \partial P$, i.e., compute $\sum_{v \in \Pi_P(A) \setminus \partial P} \alpha(v)$.*

5.1 Overview

Charalampopoulos and Karczmarz [6] showed that for any SCC S of G, $V(P) \cap S$ forms an intersection of two cells coming from two carefully prepared additively weighted Voronoi diagrams on P with sites ∂P. As a result, they could use Voronoi diagram point location mechanism [16, 29] for testing in $O(\text{polylog}\, n)$ time whether a query vertex lies in such an intersection. If we tried to follow this approach, we would need to be able to aggregate/report vertices in such intersections of cells coming from two seemingly unrelated Voronoi diagrams. This is very different from just testing membership, and it is not clear whether this can be done efficiently.

Instead, in order to prove Theorem 10, we take a more direct approach. As is done typically, we first consider the situation when A lies on a single face of P. In the single-hole case, the first step is to reduce to the case when the input piece P is acyclic; note that if a vertex lies in the path net $\Pi_P(A)$, its entire SCC in P does. Acyclicity and appropriate perturbation [10] allows us to pick a collection of paths $\pi_{u,v}$, for all $u, v \in \partial P$, such that every two paths in the collection are either disjoint or their intersection forms a single subpath of both. This property makes the paths $\pi_{u,v}$ particularly convenient to use for cutting the piece P into smaller non-overlapping parts.

More specifically, the paths $\pi_{u,v}$ are used to partition – using a polygon triangulation-like procedure – a queried net $\Pi_P(A)$ into regions in the plane with vertices $B \subseteq A$ bounded by either fragments of the face of P containing ∂P or some paths $\pi_{u,v}$ for $u, v \in B$ (so-called *base instances*). A base instance has a very special structure guaranteeing that for a given vertex $v \in V(P) \setminus \partial P$, there are *only* $O(1)$ *pairs* $s, t \in B$ such that an $s \to v \to t$ path exists in P. At the end, this crucial property can be used to reduce a base instance query B even further to looking up $\widetilde{O}(|B|)$ preprocessed answers for base instances with at most 5 vertices from ∂P, of which there are at most $\widetilde{O}(|\partial P|^5) = \widetilde{O}(r^{5/2})$. The precomputation of small base instances can be done in $\widetilde{O}(r^3)$ time.

For efficiently implementing the polygon triangulation-like partition procedure – which repeatedly cuts off base instances from the "core" part of the problem – we develop a dynamic data structure for existential reachability queries on a single face of a plane digraph (Lemma 15). To this end, we leverage the single-face reachability characterization of [23].

As the final step, we show a reduction from the case when A can lie on k faces on P, to the case when A lies on $k-1$ faces. The reduction – which eventually reduces the problem to the single-hole case – can blow up the piece's size by a constant factor. However, since there are only $O(1)$ holes initially, the general case can still be solved only constant factor slower than the single-hole case.

5.2 Reducing to the acyclic case

Recall that P is a piece of an r-division with few holes $\mathcal{R}$ and thus has size $O(r)$. Moreover, ∂P lies on $O(1)$ faces (*holes*) of some supergraph P^+ of P and $|\partial P| = O(\sqrt{r})$. For simplicity we will not differentiate between the faces/holes of P and P^+. Whenever we refer to a hole h of P, we mean a closed curve h in the plane such that whole embedding of P lies weakly on a single side of h, and only vertices ∂P may lie on the curve h.

First of all, we compute the strongly connected components $\mathcal{S}_P$ of P. Observe that from the point of view of supported queries, all vertices of a single SCC $S \in \mathcal{S}_P$ are treated exactly the same: when a single vertex $v \in S \setminus \partial P$ is reported (or its weight $\alpha(v)$ is aggregated), all other vertices from $S \setminus \partial P$ are as well. For each $S \in \mathcal{S}_P$ we precompute the aggregate weight $\alpha(S) = \sum_{v \in S \setminus \partial P} \alpha(v)$ in $O(r)$ time. We then contract each strongly connected component $S \in \mathcal{S}_P$ into a single vertex v_S. Crucially, since each subgraph $P[S]$ is connected in the undirected sense, contractions can be performed in an embedding preserving way (via a sequence of single-edge contractions), so that:

- The obtained graph P' is acyclic.
- The vertices $\partial P' := \{v_S : S \in \mathcal{S}_P, S \cap \partial P \neq \emptyset\}$ lie on $O(1)$ faces of (a supergraph of) P'.

Indeed, to see the latter property, label each $b \in \partial P$ initially each with hole that b lies on and make non-boundary vertices unlabeled. When the vertices merge, label the resulting vertex using one of the involved vertices' labels, if any of them has one. At the end there will be still $O(1)$ labels, each vertex of $\partial P'$ will have one of these labels, and for each label, all the vertices holding that label will be incident to a single face of P'. Clearly, $|P'| = O(r)$ and $|\partial P'| = O(\sqrt{r})$.

The proof of the below lemma can be found in the full version.

▶ **Lemma 13.** *Suppose a data structure $\mathcal{D}'$ of Theorem 10 is built for the acyclic graph P' with the weight function $\alpha'(v_S) := \alpha(S) = \sum_{v \in S \setminus \partial P} \alpha(v)$. Then, a query $A \subseteq \partial P$ from Theorem 10 can be reduced, in $O(|A|)$ time, to a query about $A' = \{v_S : S \in \mathcal{S}_P, S \cap A \neq \emptyset\}$ issued to $\mathcal{D}'$.*

In the following we will assume that P is an acyclic graph. We will no longer need the assumption $\Pi_P(A) \cap \partial P = A$; this was only needed for the efficient reduction to the acyclic case. Consequently, we will design a data structure with query time $\widetilde{O}(|A|)$ even if $A \subsetneq \Pi_P(A) \cap \partial P$.

The single-hole assumption. In the remaining part of this section, we assume we only want to handle queries where the query set A lies on a *single hole* h of an acyclic piece P. Due to space constraints, we present the reduction of the general case to the single-hole case in the full version. Our strategy for the single-hole case will be to efficiently break the problem into subproblems for which we have the answer precomputed.

5.3 Picking non-crossing paths

We first fix, for any $u, v \in V(P)$ such that u can reach v, one particular $u \to v$ path $\pi_{u,v} \subseteq P$. We apply the perturbation scheme of [10] to P, so that the shortest paths in P become unique. For any $u, v \in V(P)$ such that u can reach v in P, we define $\pi_{u,v}$ to be the unique shortest $u \to v$ path in P. Note that each $\pi_{u,v}$ is a simple path in P. We have the following crucial property.

▶ **Lemma 14.** *Let $u, v, x, y \in V(P)$ be such that u can reach v and x can reach y in P. If $V(\pi_{u,v}) \cap V(\pi_{x,y}) \neq \emptyset$, then $\pi_{u,v}$ and $\pi_{x,y}$ share a single (possibly zero-edge) subpath.*

Proof. Let a (b) be the first (last, resp.) vertex on $\pi_{u,v}$ that is also a vertex of $\pi_{x,y}$. $\pi_{u,v}$ can be expressed as $Q_1 \cdot T \cdot Q_2$, where $T = a \to b$. Moreover, $V(\pi_{u,v}) \cap V(\pi_{x,y}) \subseteq V(T)$. If $a \neq b$, the vertex b cannot appear before a on $\pi_{x,y}$, because then a and b would lie on a cycle in P, contradicting acyclicity of P. As a result, $\pi_{x,y}$ has an $a \to b$ subpath as well. Since shortest paths in P are unique and shortest paths have optimal substructure, the $a \to b$ subpath of $\pi_{x,y}$ equals T. So, $\pi_{x,y}$ can be expressed as $R_1 \cdot T \cdot R_2$. Since $\pi_{x,y}$ is simple, and $V(\pi_{u,v}) \cap V(\pi_{x,y}) \subseteq V(T)$, we indeed have $\pi_{u,v} \cap \pi_{x,y} = T$. ◀

Observe that since P is a DAG, for $u, v \in V(P)$, $u \neq v$, $\pi_{u,v}$ and $\pi_{v,u}$ cannot both exist. When one of them exists, we will sometimes write $\pi_{\{u,v\}}$ to denote the one that exists.

5.4 Generalization

Before we proceed, we need to state our problem in a more general way that will enable decomposition into smaller subproblems of similar kind. Let us first fix a counterclockwise order of $V(h)$, as follows. Let $u_1, e_1, u_2, e_2, \ldots, u_r, e_r$ be the sequence of vertices and edges on a counterclockwise bounding walk of h with an arbitrary starting point. The encountered vertices u_i might not be all distinct since h can be a non-simple face in general. The order $\prec_h$ of $V(h)$ that we use is obtained by removing duplicate vertices from the sequence in an arbitrary way, i.e., for each $v \in V(h)$ we pick one index i_v such that $u_{i_v} = v$. If $x, y \in V(h)$ and $i_x < i_y$, then we define $C_{x,y}$ to be the curve $e_{i_x} e_{i_x+1} \ldots e_{i_y-1}$. Otherwise, if $i_x > i_y$, then $C_{x,y} = e_{i_x} e_{i_x+1} \ldots e_r e_1 e_2 \ldots e_{i_y-1}$. If h is simple, then $C_{x,y}$ is a simple curve. However, in general, $C_{x,y}$ can be self-touching but it does not cross itself.

Suppose $B \subseteq \partial P \cap V(h)$ is given in the counterclockwise order on h. Moreover, for each pair of neighboring (i.e., appearing consecutively in the *cyclic* order of B given by $\prec_h$) vertices x, y of B, a curve $\Phi_{x,y}$ connecting x to y is given. The curve $\Phi_{x,y}$ equals either $\pi_{\{x,y\}}$ or $C_{x,y}$. Let Φ be the closed curve formed by concatenating the subsequent curves $\Phi_{x,y}$ for all neighboring $x, y \in B$. For brevity, we also extend the notation $\Phi_{a,b}$ to non-neighboring vertices a, b of B, and define $\Phi_{a,b}$ to be a concatenation of the curves $\Phi_{x,y}$ for all neighboring pairs (x, y) between a and b on Φ. We don't require Φ to be a simple (Jordan) curve; parts of it may be overlapping. Whereas Φ is allowed to be self-touching, it does not cross itself, like e.g., the cycle bounding a non-simple face of a plane graph. Let $P[\Phi]$ denote the region of P that lies *weakly* inside the curve Φ.

The objective of the problem (B, Φ) is to aggregate weights of (or report) the vertices of $\Pi_P(B) \setminus \partial P$ that lie *strictly inside* Φ. More formally, we want to aggregate vertices v that lie, at the same time, strictly on the left side of each of the curves $\Phi_{x,y}$ (seen as a curve directed from x to y) where $x, y \in B$ are neighboring in the counterclockwise order on h.[9] Note that with such a defined problem, the original goal of the query procedure can be rephrased as (A, h) since all the vertices of $\Pi_P(A) \setminus \partial P$ indeed lie strictly inside h.

5.5 Preprocessing

Preprocessing for small instances and subproblems.

- For any tuple $B = (b_1, \ldots, b_q)$ of at most 4 vertices of ∂P appearing in that order on in $\prec_h$, and any out of at most $2^q = O(1)$ possible curves Φ that might constitute the problem (B, Φ), we precompute the aggregate weight and the list of vertices in $\Pi_P(B) \setminus \partial P$ that lie strictly inside Φ. This can clearly be done in $\widetilde{O}(|\partial P|^4 \cdot |P|) = \widetilde{O}(r^3)$ time.
- For any 5-tuple $\tau = (b_0, \ldots, b_4)$ ordered by $\prec_h$, and any out of at most 2^4 possible curves $\Phi_{b_i,b_{i+1}} \in \{\pi_{\{b_i,b_{i+1}\}}, C_{b_i,b_{i+1}}\}$, where $i \in \{0, \ldots, 3\}$, we precompute and store the set X_τ of vertices $x \in V(P) \setminus \partial P$ such that:
  1. x lies on some path connecting b_1 and b_2 in P,
  2. x *does not* lie on any path connecting b_0 and b_1 in P,
  3. x lies strictly to the left of each $\Phi_{b_i,b_{i+1}}$ for $i = \{0, \ldots, 3\}$.

[9] Even more formally, assuming P is embedded in such a way that h is the infinite face of a supergraph of P, we are interested in the vertices lying in the intersection of the regions strictly inside closed curves $\Phi_{x,y} \cdot C_{y,x}$ for all neighboring x, y in B.

We also store the aggregate weight $w(X_\tau)$. This preprocessing can be performed in a brute-force way in $\widetilde{O}(|\partial P|^5 \cdot |P|) = \widetilde{O}(r^{7/2})$ time. It can be also optimized fairly easily to $\widetilde{O}(r^3)$ time and space, as shown in the full version.
- For any $u, v \in \partial P$, we store the path $\pi_{\{u,v\}}$ itself, along with aggregate weights of all its subpaths. This data can be computed in $\widetilde{O}(|\partial P|^2 \cdot |P|^2) = \widetilde{O}(r^3)$ time in a brute-force way.
- Finally, for any two pairs $(u, v), (x, y) \in \partial P \times \partial P$, we compute the intersection of the paths $\pi_{\{u,v\}}$ and $\pi_{\{x,y\}}$. By Lemma 14, that intersection is either empty or is a subpath of both these paths. Hence, it is enough to store the two endpoints of the intersection subpath only. The desired intersections can be found in $\widetilde{O}(|\partial P|^4 \cdot |P|) = \widetilde{O}(r^3)$ time in a brute-force way.

Existential reachability data structure. We also build data structures $\mathcal{L}, \mathcal{L}^R$ of the following lemma for P and P^R, respectively.

▶ **Lemma 15.** *In $\widetilde{O}(r)$ time one can construct a data structure maintaining an (initially empty) set $Z \subseteq \partial P \cap V(h)$ and supporting the following operations in $O(\text{polylog}\, n)$ time:*
- *insert or delete a single $b \in \partial P \cap V(h)$ either to or from Z.*
- *for any query vertex $v \in (\partial P \cap V(h)) \setminus Z$, find any $z \in Z$ (if exists) that v can reach in P.*

Due to space constraints, the proof of Lemma 15 is deferred to the full version.

5.6 Answering queries

We now proceed with the description of our algorithm solving the general problem (B, Φ).

Base case. Let the elements of $B = \{b_1, \ldots, b_k\}$ be sorted according to their order $\prec_h$ on Φ. For convenience, identify b_{k+l} with b_l for any integer l.

We first consider the easier *base case*, with the following additional requirement:

For any $1 \le i < j \le k$, if b_i can reach b_j or can be reached from b_j in P, then either $j = i + 1$ or $(i, j) = (1, k)$.

If $k \le 4$, we will simply return the precomputed aggregate weight, or an iterator to a list of vertices in $\Pi_P(B) \setminus \partial P$ strictly inside Φ. So suppose $k \ge 5$. We start with the following lemma.

▶ **Lemma 16.** *Let $v \in \Pi_P(B) \setminus \partial P$ lie strictly inside Φ. Then:*
- *There exists such i that v lies on some $b_i \to b_{i+1}$ or on some $b_{i+1} \to b_i$ path in P.*
- *Moreover, there is at most one more pair $\{x, y\} \subseteq B$, $\{x, y\} \neq \{b_i, b_{i+1}\}$ such that v lies on some $x \to y$ or $y \to x$ path in P, and either $\{x, y\} = \{b_{i-1}, b_i\}$, or $\{x, y\} = \{b_{i+1}, b_{i+2}\}$.*

Proof. Item (1) follows easily by the additional requirement of the base case. Let v lie on some $b_j \to c$ path in P, where $c \in B$. Since P is acyclic, we have that $b_j \neq c$ and thus $c \in \{b_{j-1}, b_{j+1}\}$. If $c = b_{j+1}$, we put $i = j$ and if $c = b_{j-1}$, we put $i = j - 1$.

For item (2), consider the case when v lies on some $b_i \to b_{i+1}$; the case when v lies on a $b_{i+1} \to b_i$ path is symmetric. Suppose v also lies on some $x \to y$ path in P, where $x, y \in B$, $x \neq y$, $\{x, y\} \neq \{b_i, b_{i+1}\}$. Since b_i can reach y through v, $y \in \{b_{i-1}, b_{i+1}\}$ by the base case requirement. Similarly, since x can reach b_{i+1} through v, $x \in \{b_i, b_{i+2}\}$. We cannot have $x = b_{i+2}$ and $y = b_{i-1}$ since b_{i+2} reaching b_{i-1} is a contradiction with the first base case requirement for $k \ge 5$. As a result, $(x, y) \neq (b_i, b_{i+1})$ implies $(x, y) = (b_i, b_{i-1})$

or $(x, y) = (b_{i+2}, b_{i+1})$. Moreover, v cannot lie on both some $b_i \to b_{i-1}$ path and some $b_{i+2} \to b_{i+1}$ path, since then b_{i+2} could reach b_{i-1}, which is again a contradiction for $k \geq 5$. ◀

By Lemma 16, each $v \in \Pi_P(B) \setminus \partial P$ falls into exactly one of the k sets Y_i, for $i = 1, \dots, k$, such that Y_i contains those v in $V(P) \setminus \partial P$ that lie on a path in P connecting b_i and b_{i+1} (in any direction), but at the same time *do not* lie on any path connecting b_{i-1} and b_i in P (in any direction). Indeed, if v appears only on paths connecting b_i and b_{i+1}, it will be included in Y_i. On the other hand, if v appears both on paths connecting b_i and b_{i+1} and on paths connecting b_{i+1} and b_{i+2}, it will be included only in Y_i, but not in Y_{i+1}.

▶ **Lemma 17.** *Suppose $v \in Y_i$. Then v lies strictly inside Φ if and only if v lies strictly on the left side of Φ_{b_{i-1},b_i}, $\Phi_{b_i,b_{i+1}}$, $\Phi_{b_{i+1},b_{i+2}}$, and $\Phi_{b_{i+2},b_{i+3}}$.*

Proof. Since strictly inside Φ means strictly to the left of all $\Phi_{b_j,b_{j+1}}$, the "$\implies$" direction is trivial.

Consider the "$\impliedby$" direction. For contradiction, suppose v does not lie strictly inside Φ. Then, for some $j \notin \{i-1, i, i+1, i+2\}$, v lies weakly to the right of $\Phi_{b_j,b_{j+1}}$. Since $v \in V(P) \setminus \partial P$ and the hole h contains only vertices of ∂P, this means that $\Phi_{b_j,b_{j+1}} = \pi_{\{b_j,b_{j+1}\}}$. Since b_j and b_{j+1} are consecutive in B, and $i \notin \{j, j+1\}$, b_i lies weakly to the left of $\pi_{\{b_j,b_{j+1}\}}$.

By $v \in Y_i$, the vertex v lies on a path Q connecting b_i and b_{i+1} in P. Since v and b_i lie weakly on different sides of $\pi_{\{b_j,b_{j+1}\}}$, the path Q has to cross $\pi_{\{b_j,b_{j+1}\}}$ at a vertex w appearing not later than v on Q (possibly $v = w$). If $Q = b_i \to b_{i+1}$, the existence of w implies that there exists an $s \to b_{i+1}$ path in P going through v such that $s \in \{b_j, b_{j+1}\}$. By $v \in Y_i$, this implies $s \in \{b_i, b_{i+2}\}$. As a result, $j \in \{i-1, i, i+1, i+2\}$, a contradiction. Similarly, if $Q = b_{i+1} \to b_i$, there exists an $s \to b_i$ path in P going through v such that $s \in \{b_j, b_{j+1}\}$. But on the other hand by $v \in Y_i$ we have $s = b_{i+1}$ so $j \in \{i, i+1\}$, a contradiction. ◀

As a result, we can equivalently aggregate vertices v in each Y_i under a (seemingly) weaker requirement that v lies strictly on the left side of Φ_{b_{i-1},b_i}, $\Phi_{b_i,b_{i+1}}$, $\Phi_{b_{i+1},b_{i+2}}$, and $\Phi_{b_{i+2},b_{i+3}}$. But this is, again, part of the precomputed data for the tuple $(b_{i-1}, b_i, b_{i+1}, b_{i+2}, b_{i+3})$.

Consequently, there are k disjoint sets Y_i to consider. We can thus aggregate weights or construct a list for reporting vertices from $v \in \Pi_P(B) \setminus \partial P$ strictly inside Φ in $O(k)$ time. We thus obtain:

▶ **Lemma 18.** *The base case can be handled in $O(|B|)$ time.*

General case. To solve the general case, we reduce it to a number of base case instances. To this end, we maintain a partition of $B = S \cup T$ such that S precedes T on Φ. Let $S = \{s_1, \dots, s_p\}$ and $T = \{t_1, \dots, t_q\}$. We will gradually simplify the problem while maintaining the following invariants:

(1) For any $u, v \in B$, if u can reach v in P, then $\pi_{u,v} \subseteq P[\Phi]$.
(2) For any $1 \leq i < j \leq p$, if s_i can reach s_j or can be reached from s_j, then $j = i + 1$.
(3) If $x, y \in B$ are neighbors in the counterclockwise order $\prec_h$ on Φ and $\Phi_{x,y} = \pi_{\{x,y\}}$, then $x, y \in S$.
(4) In the data structures $\mathcal{L}, \mathcal{L}^R$ of Lemma 15, the set Z satisfies $Z = S \setminus \{s_p\}$.

The algorithm will gradually modify B, S, T, Φ until we have $S = B$ and $T = \emptyset$. Note that when $T = \emptyset$, $(B, \Phi) = (S, \Phi)$ satisfies the requirement of the base case and can be solved in $\widetilde{O}(|B|)$ time.

Initially we put $\Phi = h$, $S = \{b_1, b_2\}$, and insert b_1 to Z to satisfy the invariants.

The main loop of the procedure runs while $T \neq \emptyset$ and does the following. Using $\mathcal{L}$ and $\mathcal{L}^R$, in $O(\text{polylog}\, n)$ time we test whether t_1 can reach Z or can be reached from Z in P. If this is not the case, we simply move t_1 to the end of S, an update Z accordingly. Note that $|T|$ decreases then.

Otherwise, we can find all vertices $X = \{x_1, \ldots, x_l\} \subseteq Z$ that t_1 can reach or can be reached from in $\widetilde{O}(|X|)$ by repeatedly extracting them from the data structures $\mathcal{L}, \mathcal{L}^R$. Additionally we sort X so that the order $x_1, \ldots, x_l$ matches the order of S on Φ. Let π_i denote the path $\pi_{\{t_1, x_i\}}$ possibly reversed to go from t_1 to x_i, and π_i^R denote the reverse π_i going from x_i to t_1. The vertices X are used to "cut off" l base case instances, as follows. For $i = 0, \ldots, l$, let S_i denote the vertices of S between x_i and x_{i+1} on Φ (inclusive), where we set $x_{l+1} := s_p$, $x_0 := s_1$. We split the problem (B, Φ) into the following subproblems (see Figure 1 for better understanding):

1. For each $i = 1, \ldots, l-1$, an instance $(S_i \cup \{t_1\}, \Phi_{x_i, x_{i+1}} \cdot \pi_{i+1}^R \cdot \pi_i)$.
2. An instance $(S_l \cup \{t_1\}, \Phi_{x_l, t_1} \cdot \pi_l)$.
3. An instance $(S_0 \cup T, \Phi_{s_1, x_1} \cdot \pi_1^R \cdot \Phi_{t_1, s_1})$.

▶ **Lemma 19.** *For each of the above subproblems (B', Φ'), and $u, v \in B'$, if u can reach v in P, then $\pi_{u,v} \subseteq P[\Phi']$.*

Proof. Note that (B', Φ') is obtained from (B, Φ) by cutting it out of (B, Φ) with at most two paths $\pi_{\{t_1, a\}}, \pi_{\{t_1, b\}}$, for some $t_1, a, b \in B' \subseteq B$. By our assumption, $\pi_{u,v} \subseteq P[\Phi]$. As a result, if $\pi_{u,v}$ was not contained in $P[\Phi']$, then $\pi_{u,v}$ would need to cross either $\pi_{\{t_1, a\}}$ or $\pi_{\{t_1, b\}}$. However, this is impossible by Lemma 14 and since $u, v \in V(P[\Phi'])$. ◀

▶ **Lemma 20.** *The obtained instances of types 1 and 2 above fall into the base case.*

Proof. To see that the base case requirement is satisfied, recall that by the invariant posed on S, if two elements of S_i, where $i \in \{1, \ldots, l\}$, are related (wrt. reachability in P), they are neighboring in S_i. By construction, t_1 can be only related to the first and the last element of S_i. ◀

Since the cutting is performed using non-crossing paths in P, the regions $P[\Phi']$ for the obtained subproblems can only share their boundaries, that is, if some v is strictly inside one of the curves Φ', then it it is not strictly inside another obtained curve Φ''. Therefore, if we proceeded with the above subproblems recursively, we would not aggregate or report any vertex of $v \in \Pi_P(B) \setminus \partial P$ twice. However, we still need to report/aggregate vertices that lie on paths $\pi_1, \ldots, \pi_l$ strictly inside the curve Φ. We now discuss how this strategy is implemented. Let us first consider solving the subproblems recursively. We handle all the obtained base case instance of types 1 and 2 as explained before. If $x_1 = s_g$, then by Lemma 18, the total time required for this is $O\left(\sum_{i=1}^{l}(|S_i| + 1)\right) = O((p - g) + l)$. But note that $l \leq p - g$, so in fact the bound is $O(p - g)$.

To handle the instance $(S_0 \cup T, \Phi_{s_1, x_1} \cdot \pi_1^R \cdot \Phi_{t_1, s_1})$, we simply replace (B, Φ) with it and proceed with solving it using the algorithm for the general case. To this end, we set $S := S_0 \cup \{t_1\}$, $T := \{t_2, \ldots, t_q\}$ and update Z in the data structures $\mathcal{L}, \mathcal{L}^R$ to S_0 by removing elements. Then, invariant (1) is satisfied by Lemma 19, and invariants (2), (3) and (4) are satisfied by construction. This way, in $\widetilde{O}(\Delta)$ time we reduce the instance (B, Φ) by $\Delta = |S| - |S_0| = p - g$ vertices. Recall that $Z = S \setminus \{s_p\}$ implies that $g < p$. Thus, $\Delta \geq 1$ and the sizes of B and T strictly decrease.

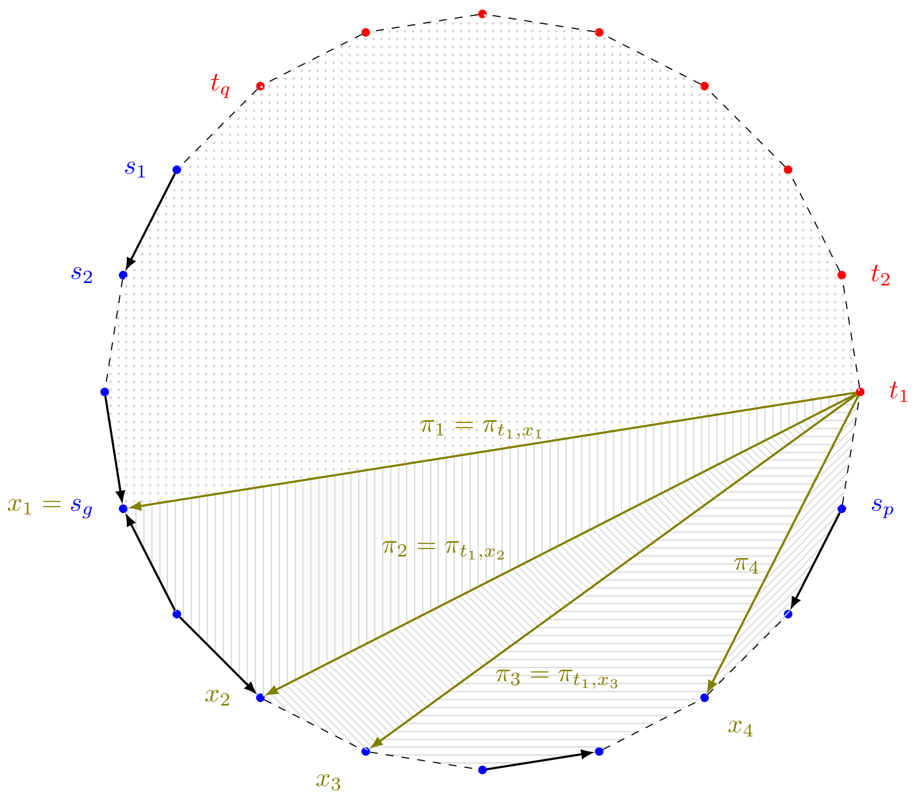

Figure 1 Splitting the instance (B, Φ), where $B = S \cup T$, into 5 smaller instances with paths $\pi_1, \ldots, \pi_l$ (either originating or ending in t_1) for $l = 4$. The vertices $S = \{s_1, \ldots, s_p\}$ are shown in blue, whereas the vertices $T = \{t_1, \ldots, t_q\}$ in red. The black arrows and dashed lines represent the individual parts of the curve Φ: paths of the form $\pi_{u,v}$ or parts of the curve h, respectively. Note that the black arrows appear only on the Φ_{s_1,s_p} part of Φ. The vertices $x_1, \ldots, x_4$, marked green, are precisely all the vertices of $S \setminus \{s_p\}$ that t_1 can reach or can be reached from. The obtained smaller instances are marked with distinct patterns. The instances marked with line patterns (types 1 or 2) are base instances. The instance marked using a dotted pattern (type 3) might constitute the only obtained instance that is not a base instance (for which the algorithm continues).

Let us now discuss how to aggregate/report the vertices of $\Pi_P(B) \setminus \partial P$ that lie on any of the paths $\pi_1, \ldots, \pi_l$ (that are not handled in any of the subproblems), but at the same time lie strictly inside Φ (before altering (B, Φ)). Since Φ is formed of either the edges of the hole h, or from the paths $\pi_{\{u,v\}}$, and each π_i is contained in $P[\Phi]$, equivalently we need to aggregate the vertices of $\Pi_P(B) \setminus \partial P$ on the paths $\pi_1, \ldots, \pi_l$ that do not lie on Φ.

Observe that since $x_1, \ldots, x_l$ lie on Φ in that order, and the paths $\pi_1, \ldots, \pi_l$ all lie in $P[\Phi]$ and are non-crossing by Lemma 14, for any three $i < j < k$, we have $V(\pi_i) \cap V(\pi_k) \subseteq V(\pi_j)$. As a result, any vertex on these paths is included in precisely one of the sets $V(\pi_i) \setminus V(\pi_{i-1})$, for $i = 1, \ldots, l$ and $V(\pi_0) := \emptyset$. Moreover, in Lemma 21 we will show that each π_i can possibly have a non-empty intersection with $O(1)$ parts (between neighboring elements of B) of Φ, that we can also identify in $O(1)$ time. Since, by Lemma 14, for every path $\pi_{\{u,v\}}$, the intersection of $\pi_{\{u,v\}}$ with π_i is either empty or forms a subpath of π_i, aggregating or reporting the required vertices of π_i boils down to aggregating or reporting the vertices of $V(P) \setminus \partial P$ on some $O(1)$ subpaths of π_i that form what remains from π_i after removing $O(1)$ of its intersections with other paths $\pi_{u,v}$.

▶ **Lemma 21.** *Consider the moment when the split into subproblems happens. Suppose* $x_i = s_j$*. Let* $u, v \in B$ *be neighboring on* Φ*, so that* u *comes before* v *in the counterclockwise order* $\prec_h$ *on* Φ*. Then* $(V(\pi_i) \cap V(\Phi_{u,v})) \setminus \partial P \neq \emptyset$ *implies that* $u = s_{j'}$ *for some* $j' \in \{1, \ldots, p\}$ *such that* $|j - j'| \leq 2$*.*

Proof. Recall that the curve is π_i is either the path $\pi_{\{x_i,t_1\}}$ or its reverse. Let us only consider the case when $\pi_{\{x_i,t_1\}} = \pi_{x_i,t_1}$; the case when $\pi_{\{x_i,t_1\}} = \pi_{t_1,x_i}$ is analogous.

By invariant (3), we have that $u, v \in S$ since otherwise $\Phi_{u,v}$ is a part of the curve h and therefore does not contain any vertices from outside $V(h)$. So $\{u, v\} = \{s_{j'}, s_{j'+1}\}$ for some $j' \in \{1, \ldots, p-1\}$. Let $z \in (V(\pi_i) \cap V(\Phi_{u,v})) \setminus \partial P$. If $\Phi_{u,v} = \pi_{s_{j'},s_{j'+1}}$, then the $z \to v$ subpath of $\Phi_{u,v}$ and the $x_i \to z$ subpath of π_i together form an $s_j \to s_{j'+1}$ path in P, which by invariant (2) implies $j' + 1 \in \{j-1, j, j+1\}$, and thus $j' \in [j-2, j]$. If $\Phi_{u,v} = \pi_{s_{j'+1},s_{j'}}$, then, analogously, there exists an $s_j \to s_{j'}$ path in P, so $j' \in [j-1, j+1]$. ◀

By Lemma 21, for each π_i, we need to report all vertices of π_i from outside ∂P, except those on the union of at most 6 subpaths of π_i. Since the subpaths are always intersections of some two paths $\pi_{u,v}$, we can identify these subpaths using precomputed data in $O(1)$ time. Aggregating vertex weights not on at most 6 subpaths of π_i is the same as aggregating weights on at most 7 disjoint subpaths of π_i. Recall that we have precomputed the aggregate weights for all the subpaths of all the possible $\pi_{u,v}$. As a result, aggregating or reporting vertices $\Pi_P(B) \setminus \partial P$ that lie on any of the paths $\pi_1, \ldots, \pi_l$ takes $O(l) = O(p - g)$ time.

▶ **Lemma 22.** *The general case can be handled in* $\widetilde{O}(|B|)$ *time.*

Proof. Recall that when $T = \emptyset$, we have a base instance that can be solved in $O(|B|)$ time.

Every iteration of the main loop that does not involve changing B takes $O(\text{polylog}\, n)$ time and reduces the size of T by one. But the set T can shrink at most $|B|$ times, so such iterations cost $\widetilde{O}(|B|)$ time in total. Every other iteration of the main loop involves reducing the size of B by some $\Delta > 0$ in $\widetilde{O}(\Delta)$ time. Such iterations clearly cost $\widetilde{O}(|B|)$ time in total as well. ◀

References

1. Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014*, pages 434–443. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.53.
2. Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A new approach to incremental cycle detection and related problems. *ACM Trans. Algorithms*, 12(2):14:1–14:22, 2016. doi:10.1145/2756553.
3. Aaron Bernstein, Aditi Dudeja, and Seth Pettie. Incremental SCC maintenance in sparse graphs. In *29th Annual European Symposium on Algorithms, ESA 2021*, volume 204 of *LIPIcs*, pages 14:1–14:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ESA.2021.14.
4. Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental reachability, scc, and shortest paths via directed expanders and congestion balancing. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*, pages 1123–1134. IEEE, 2020. doi:10.1109/FOCS46700.2020.00108.
5. Aaron Bernstein, Maximilian Probst, and Christian Wulff-Nilsen. Decremental strongly-connected components and single-source reachability in near-linear time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, pages 365–376. ACM, 2019. doi:10.1145/3313276.3316335.

6 Panagiotis Charalampopoulos and Adam Karczmarz. Single-source shortest paths and strong connectivity in dynamic planar graphs. *J. Comput. Syst. Sci.*, 124:97–111, 2022. doi:10.1016/j.jcss.2021.09.008.

7 Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Jakub Łącki, and Nikos Parotsidis. Decremental single-source reachability and strongly connected components in õ(m√n) total update time. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016*, pages 315–324. IEEE Computer Society, 2016. doi:10.1109/FOCS.2016.42.

8 Li Chen, Rasmus Kyng, Yang P. Liu, Simon Meierhans, and Maximilian Probst Gutenberg. Almost-linear time algorithms for incremental graphs: Cycle detection, sccs, *s*-*t* shortest path, and minimum-cost flow, 2023. arXiv:2311.18295.

9 Krzysztof Diks and Piotr Sankowski. Dynamic plane transitive closure. In *Algorithms - ESA 2007, 15th Annual European Symposium, Proceedings*, pages 594–604, 2007. doi:10.1007/978-3-540-75520-3_53.

10 Jeff Erickson, Kyle Fox, and Luvsandondov Lkhamsuren. Holiest minimum-cost paths and flows in surface graphs. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018*, pages 1319–1332, 2018. doi:10.1145/3188745.3188904.

11 Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci.*, 72(5):868–889, 2006. doi:10.1016/j.jcss.2005.05.007.

12 Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985. doi:10.1137/0214055.

13 Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput.*, 16(6):1004–1022, 1987. doi:10.1137/0216064.

14 Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Inf. Process. Lett.*, 74(3-4):107–114, 2000. doi:10.1016/S0020-0190(00)00051-X.

15 Pawel Gawrychowski, Haim Kaplan, Shay Mozes, Micha Sharir, and Oren Weimann. Voronoi diagrams on planar graphs, and computing the diameter in deterministic õ($n^{5/3}$) time. *SIAM J. Comput.*, 50(2):509–554, 2021. doi:10.1137/18M1193402.

16 Pawel Gawrychowski, Shay Mozes, Oren Weimann, and Christian Wulff-Nilsen. Better tradeoffs for exact distance oracles in planar graphs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 515–529. SIAM, 2018. doi:10.1137/1.9781611975031.34.

17 Gramoz Goranci, Harald Räcke, Thatchaphol Saranurak, and Zihan Tan. The expander hierarchy and its applications to dynamic graph algorithms. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2212–2228. SIAM, 2021. doi:10.1137/1.9781611976465.132.

18 Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert Endre Tarjan. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Trans. Algorithms*, 8(1):3:1–3:33, 2012. doi:10.1145/2071379.2071382.

19 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015*, pages 21–30. ACM, 2015. doi:10.1145/2746539.2746609.

20 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001. doi:10.1145/502090.502095.

21 Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. Fully dynamic connectivity in o(log n(loglog n)^2) amortized expected time. *TheoretiCS*, 2, 2023. doi:10.46298/THEORETICS.23.6.

22 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001. doi:10.1006/jcss.2000.1727.

23 Giuseppe F. Italiano, Adam Karczmarz, Jakub Łącki, and Piotr Sankowski. Decremental single-source reachability in planar digraphs. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017*, pages 1108–1121. ACM, 2017. doi:10.1145/3055399.3055480.

24 Haim Kaplan, Shay Mozes, Yahav Nussbaum, and Micha Sharir. Submatrix maximum queries in monge matrices and partial monge matrices, and their applications. *ACM Trans. Algorithms*, 13(2):26:1–26:42, 2017. doi:10.1145/3039873.

25 Adam Karczmarz and Marcin Smulewicz. On fully dynamic strongly connected components. In *31st Annual European Symposium on Algorithms, ESA 2023, September 4-6, 2023, Amsterdam, The Netherlands*, volume 274 of *LIPIcs*, pages 68:1–68:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ESA.2023.68.

26 Philip N. Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005*, pages 146–155, 2005. URL: http://dl.acm.org/citation.cfm?id=1070432.1070454.

27 Philip N. Klein, Shay Mozes, and Christian Sommer. Structured recursive separator decompositions for planar graphs in linear time. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 505–514, 2013. doi:10.1145/2488608.2488672.

28 Philip N. Klein and Sairam Subramanian. A fully dynamic approximation scheme for shortest paths in planar graphs. *Algorithmica*, 22(3):235–249, 1998. doi:10.1007/PL00009223.

29 Yaowei Long and Seth Pettie. Planar distance oracles with better time-space tradeoffs. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2517–2537. SIAM, 2021. doi:10.1137/1.9781611976465.149.

30 Jakub Łącki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Trans. Algorithms*, 9(3):27:1–27:15, 2013. doi:10.1145/2483699.2483707.

31 Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.*, 37(5):1455–1471, 2008. doi:10.1137/060650271.

32 Piotr Sankowski. Subquadratic algorithm for dynamic shortest distances. In *Computing and Combinatorics, 11th Annual International Conference, COCOON 2005, Kunming, China, August 16-29, 2005, Proceedings*, volume 3595 of *Lecture Notes in Computer Science*, pages 461–470. Springer, 2005. doi:10.1007/11533719_47.

33 M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981. doi:10.1016/0898-1221(81)90008-0.

34 Sairam Subramanian. A fully dynamic data structure for reachability in planar digraphs. In *Algorithms - ESA '93, First Annual European Symposium, Bad Honnef, Germany, September 30 - October 2, 1993, Proceedings*, pages 372–383, 1993. doi:10.1007/3-540-57273-2_72.

35 Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. doi:10.1137/0201010.

36 Jan van den Brand, Danupon Nanongkai, and Thatchaphol Saranurak. Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019*, pages 456–480. IEEE Computer Society, 2019. doi:10.1109/FOCS.2019.00036.

37 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2-3):357–365, 2005. doi:10.1016/j.tcs.2005.09.023.

38 Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1757–1769. SIAM, 2013. doi:10.1137/1.9781611973105.126.