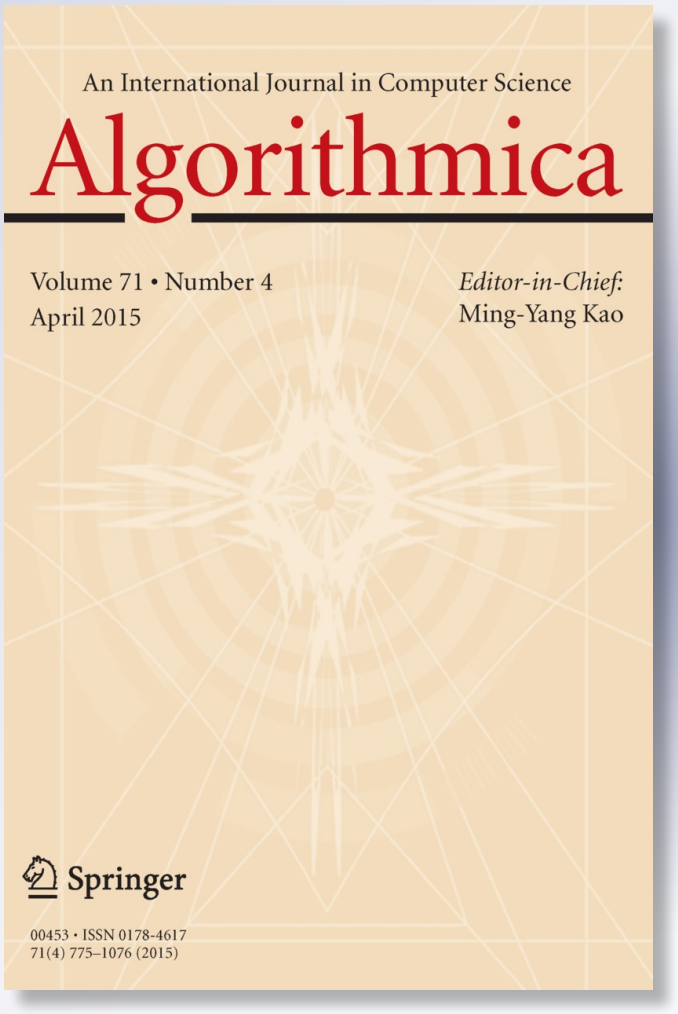
Data Structures on Event Graphs

Bernard Chazelle & Wolfgang Mulzer

Algorithmica

ISSN 0178-4617
Volume 71
Number 4

Algorithmica (2015) 71:1007-1020
DOI 10.1007/s00453-013-9838-4



Springer

Algorithmica (2015) 71:1007–1020
DOI 10.1007/s00453-013-9838-4

Data Structures on Event Graphs

Bernard Chazelle · Wolfgang Mulzer

Received: 24 September 2012 / Accepted: 17 September 2013 / Published online: 26 September 2013

Abstract We investigate the behavior of data structures when the input and operations are generated by an *event graph*. This model is inspired by Markov chains. We are given a fixed graph G, whose nodes are annotated with operations of the type *insert*, *delete*, and *query*. The algorithm responds to the requests as it encounters them during a (random or adversarial) walk in G. We study the limit behavior of such a walk and give an efficient algorithm for recognizing which structures can be generated. We also give a near-optimal algorithm for successor searching if the event graph is a cycle and the walk is adversarial. For a random walk, the algorithm becomes optimal.

Keywords Successor searching · Markov Chain · Low entropy · Data Structure

1 Introduction

In contrast with the traditional adversarial assumption of worst-case analysis, many data sources are modeled by Markov chains (e.g., in queuing, speech, gesture, protein homology, web searching, etc.). These models are very appealing because they are widely applicable and simple to generate. Indeed, locality of reference, an essential

A preliminary version appeared as B. Chazelle and W. Mulzer, *Data Structures on Event Graphs* in Proc. 20th ESA, pp. 313–324, 2012.

W. Mulzer was supported in part by DFG grant MU3501/1.

B. Chazelle
Department of Computer Science, Princeton University, Princeton, USA
e-mail: chazelle@cs.princeton.edu

W. Mulzer (✉)
Institut für Informatik, Freie Universität Berlin, Berlin, Germany
e-mail: mulzer@inf.fu-berlin.de

pillar in the design of efficient computing systems, is often captured by a Markov chain modeling the access distribution. Hence, it does not come as a surprise that this connection has motivated and guided much of the research on self-organizing data structures and online algorithms in a Markov setting [1, 7–11, 15–18]. That body of work should be seen as part of a larger effort to understand algorithms that exploit the fact that input distributions often exhibit only a small amount of entropy. This effort is driven not only by the hope for improvements in practical applications (e.g., exploiting coherence in data streams), but it is also motivated by theoretical questions: for example, the key to resolving the problem of designing an optimal deterministic algorithm for minimum spanning trees lies in the discovery of an optimal heap for constant-entropy sources [2]. Markov chains have been studied intensively, and there exists a huge literature on them (e.g., [12]). Nonetheless, the focus has been on state functions (such as stationary distribution or commute/cover/mixing times) rather than on the behavior of complex objects evolving over them. This leads to a number of fundamental questions which, we hope, will inspire further research.

Let us describe our model in more detail. Our object of interest is a structure $\mathcal{T}(X)$ that evolves over time. The structure $\mathcal{T}(X)$ is defined over a finite subset X of a universe $\mathcal{U}$. In the simplest case, we have $\mathcal{U} = \mathbb{N}$ and $\mathcal{T}(X) = X$. This corresponds to the classic dictionary problem where we need to maintain a subset of a given universe. We can also imagine more complicated scenarios such as $\mathcal{U} = \mathbb{R}^d$ with $\mathcal{T}(X)$ being the Delaunay triangulation of X. An *event graph* $G = (V, E)$ specifies restrictions on the queries and updates that are applied to $\mathcal{T}(X)$. For simplicity, we assume that G is undirected and connected. Each node $v \in V$ is associated with an item $x_v \in \mathcal{U}$ and corresponds to one of three possible requests: (i) $\texttt{insert}(x_v)$; (ii) $\texttt{delete}(x_v)$; or (iii) $\texttt{query}(x_v)$. Requests are specified by following a walk in G, beginning at a designated start node of G and hopping from node to neighboring node. We consider both *adversarial* walks, in which the neighbors can be chosen arbitrarily, and *random* walks, in which the neighbor is chosen uniformly at random. The latter case corresponds to the classic Markov chain model. Let v^t be the node of G visited at time t and let $X^t \subseteq \mathcal{U}$ be the set of *active elements*, i.e., the set of items inserted prior to time t and not deleted after their last insertions. We also call X^t an *active set*. For any $t > 0$, $X^t = X^{t-1} \cup \{x_{v^t}\}$ if the operation at v^t is an insertion and $X^t = X^{t-1} \setminus \{x_{v^t}\}$ in the case of deletion. The query at v depends on the structure under consideration (successor, point location, ray shooting, etc.). Another way to interpret the event graph is as a finite automaton that generates words over an alphabet with certain cancellation rules.

Markov chains are premised on forgetting the past. In our model, however, the structure $\mathcal{T}(X^t)$ can remember quite a bit. In fact, we can define a secondary graph over the much larger vertex set $V \times 2^{\mathcal{U}_{|V}}$, where $\mathcal{U}_{|V} = \{x_v \,|\, v \in V\}$ denotes those elements in the universe that occur as labels in G, see Fig. 1. We call this larger graph the *decorated graph*, $\mathrm{dec}(G)$, since the way to think of this secondary graph is to picture each node v of G being "decorated" with the subsets $X \subseteq \mathcal{U}_{|V}$. (We define the vertex set using $2^{\mathcal{U}_{|V}}$ in order to allow for every possible initial subset X.) Let n be the number of nodes in G. Since $|\mathcal{U}_{|V}| \leq n$, an edge (v, w) in the original graph gives rise to up to 2^n edges $(v, X)(w, Y)$ in the decorated graph, with Y derived from X in the obvious way. A trivial upper bound on the number of states is $n2^n$, which is

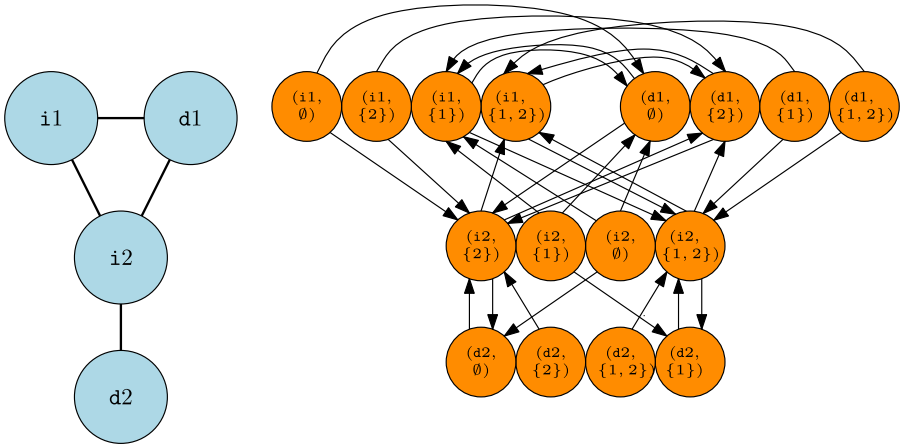


Fig. 1 An event graph over four vertices and the associated decorated graph. Each node of the event graph is replaced by four nodes decorated with the subsets of $\{1, 2\}$

essentially tight. If we could afford to store all of $\mathrm{dec}(G)$, then any of the operations at the nodes of the event graph could be precomputed and the running time per step would be constant. However, the required space might be huge, so the main question is

Can the decorated graph be compressed with no loss of performance?

This seems a difficult question to answer in general. In fact, even counting the possible active sets in decorated graphs seems highly nontrivial, as it reduces to counting words in regular languages augmented with certain cancellation rules. Hence, in this paper we focus on basic properties and special cases that highlight the interesting behavior of the decorated graph. Beyond the results themselves, the main contribution of this work is to draw the attention of algorithm designers to a more realistic input model that breaks away from worst-case analysis.

Our Results The paper has two main parts. In the first part, we investigate some basic properties of decorated graphs. We show that the decorated graph $\mathrm{dec}(G)$ has a unique strongly connected component that corresponds to the limiting phase of a walk on the event graph G, and we give characterizations for when a set $X \subseteq \mathcal{U}_{|V}$ appears as an active set in this limiting phase. We also show that whether X is such an active set can be decided in linear time (in the size of G).

In the second part, we consider the problem of maintaining a dictionary that supports successor searches during a one-dimensional walk on a cycle. We show how to achieve linear space and constant expected time for a random walk. If the walk is adversarial, we can achieve a similar result with near-linear storage. The former result is in the same spirit as previous work by the authors on randomized incremental construction (RIC) for Markov sources [3]. RIC is a fundamental algorithmic paradigm in computational geometry that uses randomness for the construction of certain geometric objects, and we showed that there is no significant loss of efficiency if the randomness comes from a Markov chain with sufficiently high conductance.

2 Basic Properties of Decorated Graphs

We are given a labeled, connected, undirected graph $G = (V, E)$. In this section, we consider only labels of the form $\mathtt{i}x$ and $\mathtt{d}x$, where x is an element from a finite universe $\mathcal{U}$ and $\mathtt{i}$ and $\mathtt{d}$ stand for `insert` and `delete`. We imagine an adversary that maintains a subset $X \subseteq \mathcal{U}$ while walking on G and performing the corresponding operations on the nodes. Since the focus of this section is the evolution of X over time, we ignore queries for now.

Recall that $\mathcal{U}_{|V}$ denotes the elements that appear on the nodes of G. For technical convenience, we require that for every $x \in \mathcal{U}_{|V}$ there is at least one node labeled $\mathtt{i}x$ and at least one node labeled $\mathtt{d}x$. The walk on G is formalized through the *decorated graph* $\mathrm{dec}(G)$. The graph $\mathrm{dec}(G)$ is a directed graph on vertex set $V' := V \times 2^{\mathcal{U}_{|V}}$. The pair $((u, X), (v, Y))$ is an edge in E' if and only if $\{u, v\}$ is an edge in G and $Y = X \cup \{x_v\}$ or $Y = X \backslash \{x_v\}$ depending on whether v is labeled $\mathtt{i}x_v$ or $\mathtt{d}x_v$, see Fig. 1.

By a *walk* W in a (directed or undirected) graph, we mean any finite sequence of nodes such that the graph contains an edge from each node in W to its successor in W (in particular, a node may appear multiple times in W). Let A be a walk in $\mathrm{dec}(G)$. Recall that the nodes in A are tuples, consisting of a node in G and a subset of $\mathcal{U}_{|V}$. By taking the first elements of the nodes in A, we obtain a walk in G, the *projection* of A, denoted by $\mathrm{proj}(A)$. For example, in Fig. 1, the projection of the walk $(\mathtt{i}1, \emptyset), (\mathtt{i}2, \{2\}), (\mathtt{i}1, \{1, 2\}), (\mathtt{d}1, \{2\})$ in the decorated graph is the walk $\mathtt{i}1, \mathtt{i}2, \mathtt{i}1, \mathtt{d}1$ in the event graph. Similarly, let W be a walk in G with start node v, and let $X \subseteq 2^{\mathcal{U}_{|V}}$. Then the *lifting* of W with respect to X is the walk in $\mathrm{dec}(G)$ that begins at node (v, X) and follows the steps of W in $\mathrm{dec}(G)$. We denote this walk by $\mathrm{lift}(W, X)$. For example, in Fig. 1, we have $\mathrm{lift}((\mathtt{i}1, \mathtt{i}2, \mathtt{i}1, \mathtt{d}1), \emptyset) = ((\mathtt{i}1, \emptyset), (\mathtt{i}2, \{2\}), (\mathtt{i}1, \{1, 2\}), (\mathtt{d}1, \{2\}))$.

Since $\mathrm{dec}(G)$ is a directed graph, it can be decomposed into strongly connected components that induce a directed acyclic graph D. We call a strongly connected component of $\mathrm{dec}(G)$ a *sink component* (also called essential class in Markov chain theory), if it corresponds to a sink (i.e., a node with out-degree 0) in D. First, we observe that every node of G is represented in each sink component of $\mathrm{dec}(G)$, see Fig 2.

Lemma 2.1 *Let $\mathcal{C}$ be a sink component of* $\mathrm{dec}(G)$. *For each vertex v of G, there exists at least one subset $Y \subseteq \mathcal{U}_{|V}$ such that (v, Y) is a node in $\mathcal{C}$. In other words, v is the first element of at least one node in $\mathcal{C}$.*

Proof Let (w, X) be any node in $\mathcal{C}$. Since G is connected, there is a walk W in G from w to v, so $\mathrm{lift}(W, X)$ ends in a node in $\mathcal{C}$ whose first element is v. □

Next, we show that to understand the behaviour of a walk on G in the limit, it suffices to focus on a single sink component of $\mathrm{dec}(G)$.

Lemma 2.2 *In* $\mathrm{dec}(G)$ *there exists a unique sink component $\mathcal{C}$ such that for every node $(v, \emptyset)$ in* $\mathrm{dec}(G)$, *$\mathcal{C}$ is the only sink component that $(v, \emptyset)$ can reach.*

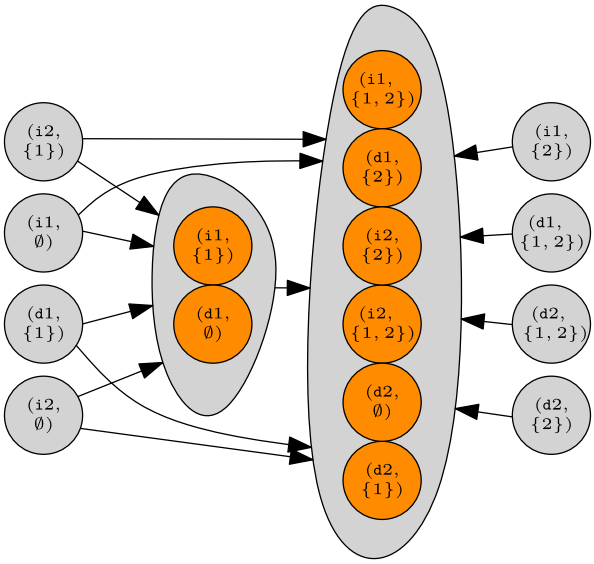


Fig. 2 The decomposition of the decorated graph from Fig. 1 into strongly connected components. There is a unique sink component in which each node from the event graph is represented

Proof Suppose there is a node v in G such that $(v, \emptyset)$ can reach two different sink components $\mathcal{C}$ and $\mathcal{C}'$ in $\mathrm{dec}(G)$. By Lemma 2.1, both $\mathcal{C}$ and $\mathcal{C}'$ must contain at least one node with first element v. Call these nodes (v, X) (for $\mathcal{C}$) and (v, X') (for $\mathcal{C}'$). Furthermore, by assumption $\mathrm{dec}(G)$ contains a walk A from $(v, \emptyset)$ to (v, X) and a walk A' from $(v, \emptyset)$ to (v, X'). Let $W := \mathrm{proj}(A)$ and $W' := \mathrm{proj}(A')$. Both W and W' are closed walks in G that start and end in v, so their concatenations $WW'W$ and $W'W'W$ are valid walks in G, again with start and end vertex v. Consider the lifted walks $\mathrm{lift}(WW'W, \emptyset)$ and $\mathrm{lift}(W'W'W, \emptyset)$ in $\mathrm{dec}(G)$. We claim that these two walks have the same end node (v, X''). Indeed, for each $x \in \mathcal{U}_{|V}$, whether x appears in X'' or not depends solely on whether the label $\mathtt{i}x$ or the label $\mathtt{d}x$ appears last on the original walk in G. This is the same for both $WW'W$ and $W'W'W$. Hence, $\mathcal{C}$ and $\mathcal{C}'$ must both contain (v, X'), a contradiction to the assumption that they are distinct sink components. Thus, each node $(v, \emptyset)$ can reach exactly one sink component.

Now consider two distinct nodes $(v, \emptyset)$ and $(w, \emptyset)$ in $\mathrm{dec}(G)$ and assume that they reach the sink components $\mathcal{C}$ and $\mathcal{C}'$, respectively. Let W be a walk in G that goes from v to w and let $W' := \mathrm{proj}(A)$, where A is a walk in $\mathrm{dec}(G)$ that connects w to $\mathcal{C}'$. Since G is undirected, the reversed walk W^R is a valid walk in G from w to v. Now consider the walks $Z_1 := WW^RWW'$ and $Z_2 := W^RWW'$. The walk Z_1 begins in v, the walk Z_2 begins in w, and they both have the same end node. Furthermore, for each $x \in \mathcal{U}_{|V}$, the label $\mathtt{i}x$ appears last in Z_1 if and only if it appears last in Z_2. Hence, the lifted walks $\mathrm{lift}(Z_1, \emptyset)$ and $\mathrm{lift}(Z_2, \emptyset)$ have the same end node in $\mathrm{dec}(G)$, so $\mathcal{C} = \mathcal{C}'$. The lemma follows. □

Since the unique sink component $\mathcal{C}$ from Lemma 2.2 represents the limit behaviour of the set X during a walk in G, we will henceforth focus on this component. Let us begin with a few properties of $\mathcal{C}$. First, we characterize the nodes in $\mathcal{C}$.

Lemma 2.3 *Let v be a node of G and $X \subseteq \mathcal{U}_{|V}$. We have $(v, X) \in \mathcal{C}$ if and only if there exists a closed walk W in G with the following properties*:

1. *the walk W starts and ends in v*:
2. *for each $x \in \mathcal{U}_{|V}$, there is at least one node in W with label $\mathtt{i}x$ or $\mathtt{d}x$*;
3. *we have $x \in X$ if and only if the last node in W referring to x is an insertion and $x \notin X$ if and only if the last node in W referring to x is a deletion.*

We call the walk W from Lemma 2.3 a *certifying walk* for the node (v, X) of $\mathcal{C}$. For example, as we can see in Fig. 2, the sink component of our example graph contains the node $(\mathtt{d}2, \{2\})$. A certifying walk for this node is $\mathtt{d}2, \mathtt{i}2, \mathtt{d}1, \mathtt{i}2, \mathtt{d}2$.

Proof First, suppose there is a walk with the given properties. By Lemma 2.1, there is at least one node in $\mathcal{C}$ whose first element is v, say (v, Y). The properties of W immediately imply that the walk $\text{lift}(W, Y)$ ends in (v, X), which proves the "if"-direction of the lemma.

Now suppose that (v, X) is a node in $\mathcal{C}$. Since $\mathcal{C}$ is strongly connected, there exists a closed walk A in $\mathcal{C}$ that starts and ends at (v, X) and visits every node of $\mathcal{C}$ at least once. Let $W := \text{proj}(A)$. By Lemma 2.1 and our assumption on the labels of G, the walk W contains for every element $x \in \mathcal{U}_{|V}$ at least one node with label $\mathtt{i}x$ and one node with label $\mathtt{d}x$. Therefore, the walk W meets all the desired properties. □

This characterization of the nodes in $\mathcal{C}$ immediately implies that the decorated graph can have only one sink component.

Corollary 2.4 *The component $\mathcal{C}$ is the only sink component of* $\text{dec}(G)$.

Proof Let (v, X) be a node in $\text{dec}(G)$. By Lemmas 2.1 and 2.3, there exists in $\mathcal{C}$ a node of the form (v, Y) and a corresponding certifying walk W. Clearly, the walk $\text{lift}(W, X)$ ends in (v, Y). Thus, every node in $\text{dec}(G)$ can reach $\mathcal{C}$, so there can be no other sink component. □

Next, we give a bound on the length of certifying walks, from which we can deduce a bound on the diameter of $\mathcal{C}$.

Theorem 2.5 *Let (v, X) be a node of $\mathcal{C}$ and let W be a corresponding certifying walk of minimum length. Then W has length at most $O(n^2)$, where n denotes the number of nodes in G. There are examples where any certifying walk needs $\Omega(n^2)$ nodes. It follows that $\mathcal{C}$ has diameter $O(n^2)$ and that this is tight.*

Proof Consider the reversed walk W^R. We subdivide W^R into *phases*: a new phase starts when W^R encounters a node labeled $\mathtt{i}x$ or $\mathtt{d}x$ for an $x \in \mathcal{U}_{|V}$ that it has not seen before. Clearly, the number of phases is at most n. Now consider the i-th phase and let V_i be the set of nodes in G whose labels refer to the i distinct elements of $\mathcal{U}_{|V}$ that have been encountered in the first i phases. In phase i, the walk W^R can use only vertices in V_i. Since W has minimum cardinality, the phase must consist of a shortest

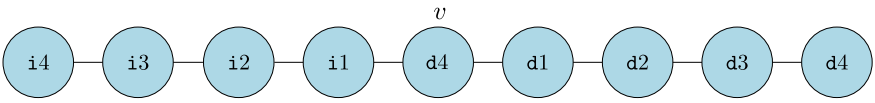


Fig. 3 The lower bound example for $m = 4$. The shortest certifying walk for $(v, \{1, 3\})$ goes from v to i3, then to d2, then to i1, and then back to v

walk in V_i from the first node of phase i to the first node of phase $i + 1$. Hence, each phase consists of at most n vertices and the length of W is $O(n^2)$.

We now describe the lower bound construction. Let $m \geq 2$ be an integer. The event graph P is a path with $n = 2m + 1$ vertices. The first m vertices are labeled im, i$(m-1)$, ..., i1, in this order. The middle vertex is labeled dm, and the remaining m vertices are labeled d1, d2, ..., dm, in this order, see Fig. 3. Let v be the middle vertex of P and $\mathcal{C}$ be the unique sink component of dec(P). First, note that (v, X) is a node of $\mathcal{C}$ for every $X \subseteq \{1, \ldots, m-1\}$. Indeed, given $X \subseteq \{1, \ldots, m-1\}$, we can construct a certifying walk for X as follows: we begin at v, and for $k = m-1$, $m-2, \ldots, 1$, we walk from v to ik or dk, depending on whether k lies in X or not, and back to v. This gives a certifying walk for X with $2(m-1)+2(m-2)+\cdots+2 = \Theta(m^2)$ steps. Now, we claim that the length of a shortest certifying walk for the node $(v, \{2k+1 \mid k = 0, \ldots, \lfloor m/2 \rfloor - 1\})$ is $\Theta(m^2) = \Theta(n^2)$. Indeed, note that the set $Y = \{2k+1 \mid k = 0, \ldots, \lfloor m/2 \rfloor - 1\}$ contains exactly the odd numbers between 1 and $m-1$. Thus, a certifying walk for Y must visit the node i1 after all visits to node d1, the node d2 after all visits to i2, etc. Furthermore, the structure of P dictates that any certifying walk performs these visits in order from largest to smallest, i.e., first comes the last visit to the node for $m-1$, then the last visit to the node for $m-2$, etc. To see this, suppose that there exist $i < j$ such that the last visit to the node for i, w_i, comes before the last visit to the node for j, w_j. Then the parity of i and j must differ, because otherwise the walk must cross w_i on the way from w_j to v. However, in this case, on the way from w_j to v, the certifying walk has to cross the node with the wrong label for i (`insert` instead of `delete`, or vice versa), and hence it could not be a certifying walk. It follows that any certifying walk for (v, Y) has length $\Omega(n^2)$.

We now show that any two nodes in $\mathcal{C}$ are connected by a walk of length $O(n^2)$. Let (u, X) and (v, Y) be two such nodes and let Q be a shortest walk from u to v in G and W be a certifying walk for (v, Y). Then lift(QW, X) is a walk of length $O(n^2)$ in $\mathcal{C}$ from (u, X) to (v, Y). Hence, the diameter of $\mathcal{C}$ is $O(n^2)$. Again, the lower bound example from the previous paragraph applies: the length of a shortest walk in $\mathcal{C}$ between $(v, \emptyset)$ and $(v, \{2k+1 \mid k = 0, \ldots, \lfloor m/2 \rfloor - 1\})$ is $\Theta(n^2)$, as can be seen by an argument similar to the argument for the shortest certifying walk. □

Next, we describe an algorithm that is given G, a node $v \in V$, and a set $X \subseteq \mathcal{U}_{|V}$ and then decides whether (v, X) is a node of the unique sink or not. For $W \subseteq V$, let $\mathcal{U}_{|W}$ denote the elements that appear in the labels of the nodes in W. For $U \subseteq \mathcal{U}$, let $V_{|U}$ denote the nodes of G whose labels contain an element of U.

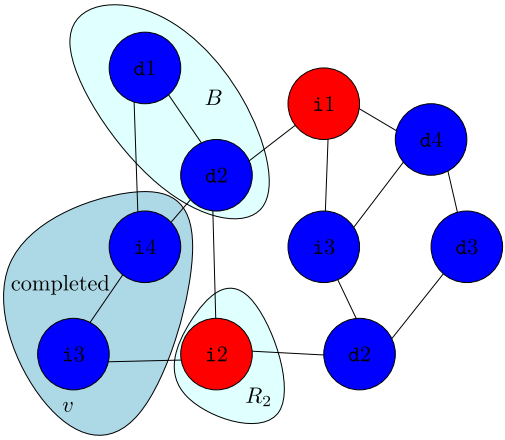


Fig. 4 An intermediate stage of the algorithm while deciding whether the node $(v, \{3, 4\})$ lies in the unique sink of the given event graph. At this point, the nodes v and $\mathtt{i4}$ have been processed. Since the elements 3 and 4 have been encountered, the corresponding nodes have been colored blue. The nodes for the other elements still have the original color. We have $B = \{\mathtt{d1}, \mathtt{d2}\}$, $R_2 = \{\mathtt{i2}\}$, and $R_1 = R_3 = R_4 = \emptyset$. Suppose that in the next step, the algorithm processes $\mathtt{d2}$. Then the node $\mathtt{i2}$ is colored blue and added to B, and $\mathtt{i1}$ is added to R_1

Theorem 2.6 *Given an event graph G, a node v of G and a subset $X \subseteq \mathcal{U}_{|V}$, we can decide in $O(|V| + |E|)$ steps whether (v, X) is a node of the unique sink component $\mathcal{C}$ of* $\mathrm{dec}(G)$.

Proof The idea of the algorithm is to construct a certifying walk for (v, X) through a modified breadth first search.

In the preprocessing phase, we color a vertex w of G *blue* if w is labeled $\mathtt{i}x$ and $x \in X$, or if w is labeled $\mathtt{d}x$ and $x \notin X$. Otherwise, we color w *red*. If v is colored red, then (v, X) cannot be in $\mathcal{C}$, and we are done. Otherwise, we perform a directed breadth first search that starts from v and tries to construct a reverse certifying walk. Our algorithm maintains several queues. The main queue is called the *blue fringe* B. Furthermore, for every $x \in \mathcal{U}_{|V}$, we have a queue R_x, the *red fringe* for x. At the beginning, the queue B contains only v, and all the red fringes are empty.

The main loop of the algorithm takes place while B is not empty. We pull the next node w out of B, and we process w as follows: if we have not seen the element $x_w \in \mathcal{U}_{|V}$ for w before, we color the set $V_{|\{x_w\}}$ of all nodes whose label refers to x_w blue, append all the nodes of R_{x_w} to B, and we delete R_{x_w}. Next, we process the neighbors of w as follows: if a neighbor w' of w is blue, we append it to B if w' has not been inserted into B before. If w' is red and labeled with the element $x_{w'}$, we append w' to $R_{x_{w'}}$, if necessary, see Fig. 4.

The algorithm terminates after at most $|V|$ iterations. In each iteration, the cost is proportional to the degree of the current vertex w and (possibly) the size of one red fringe. The latter cost can be charged to later rounds, since the nodes of the red fringe are processed later on. Let V_{red} be the union of the remaining red fringes after the algorithm terminates.

If $V_{\text{red}} = \emptyset$, we obtain a certifying walk for (v, X) by walking from one newly discovered vertex to the next inside the current blue component and reversing the walk. Now suppose $V_{\text{red}} \neq \emptyset$. Let A be the set of all vertices that were traversed during the BFS. Then $G \setminus V_{\text{red}}$ has at least two connected components (since there must be blue vertices outside of A). Furthermore, $\mathcal{U}_{|A} \cap \mathcal{U}_{|V_{\text{red}}} = \emptyset$. We claim that a certifying walk for (v, X) cannot exist. Indeed, suppose that W is such a certifying walk. Let $x_w \in \mathcal{U}_{|V_{\text{red}}}$ be the element in the label of the last node w in W whose label refers to an element in $\mathcal{U}_{|V_{\text{red}}}$. Suppose that the label of w is of the form $\mathtt{i}x_w$; the other case is symmetric. Since W is a certifying walk, we have $x_w \in X$, so w was colored blue during the initialization phase. Furthermore, all the nodes on W that come after w are also blue at the end. This implies that $w \in A$, because by assumption a neighor of w was in B, and hence w must have been added to B when this neighbor was processed. Hence, we get a contradiction to the fact that $\mathcal{U}_{|A} \cap \mathcal{U}_{|V_{\text{red}}} = \emptyset$, so W cannot exist. Therefore, $(v, X) \notin \mathcal{C}$. □

The proof of Theorem 2.6 gives an alternative characterization of whether a node appears in the unique sink component or not.

Corollary 2.7 *The node (v, X) does not appear in $\mathcal{C}$ if and only if there exists a set $A \subseteq V(G)$ with the following properties*:

1. *$G \setminus A$ has at least two connected components.*
2. *$\mathcal{U}_{|A} \cap \mathcal{U}_{|B} = \emptyset$, where B denotes the vertex set of the connected component of $G \setminus A$ that contains v.*
3. *For all $x \in \mathcal{U}$, A contains either only labels of the form $\mathtt{i}x$ or only labels of the form $\mathtt{d}x$ (or neither). If A has a node with label $\mathtt{i}x$, then $x \notin X$. If A has a node with label $\mathtt{d}x$, then $x \in X$.*

A set A with the above properties can be found in polynomial time.

Lemma 2.8 *Given $k \in \mathbb{N}$ and a node $(v, X) \in \mathcal{C}$, it is* NP*-complete to decide whether there exists a certifying walk for (v, X) of length at most k.*

Proof The problem is clearly in NP. To show completeness, we reduce from Hamiltonian path in undirected graphs. Let G be an undirected graph with n vertices, and suppose the vertex set is $\{1, \ldots, n\}$. We let $\mathcal{U} = \mathbb{N}$ and take two copies G_1 and G_2 of G. We label the copy of node i in G_1 with $\mathtt{i}i$ and the copy of node i in G_2 with $\mathtt{d}i$. Then we add two nodes v_1 and v_2, and we connect v_1 to v_2 and to all nodes in G_1 and G_2, We label v_1 with $\mathtt{i}(n+1)$ and v_2 with $\mathtt{d}(n+1)$. The resulting graph G' has $2n+2$ nodes and meets all our assumptions about an event graph. Clearly, G' can be constructed in polynomial time. Finally, since by definition a certifying walk must visit for each element i either $\mathtt{i}i$ or $\mathtt{d}i$, it follows that G has a Hamiltonian path if and only if the node $(v_1, \{1, \ldots, n+1\})$ has a certifying walk of length at most $n+2$. This completes the reduction. □

3 Successor Searching on Cycle Graphs

We now consider the case that the event graph G is a simple cycle $v_1, \ldots, v_n, v_1$ and the item x_{v_i} at node v_i is a real number. Again, the structure $\mathcal{T}(X)$ is X itself, and we now have three types of nodes: insertion, deletion, and query. A query at time t asks for $\texttt{succ}_{X^t}(x_{v^t}) = \min\{\, x \in X^t \mid x \geq x_{v^t} \,\}$ (or ∞). Again, an example similar to Fig. 3 shows that the decorated graph can be of exponential size: let n be even. For $i = 1, \ldots, n/2$, take $x_{v_i} = x_{v_{n+1-i}} = i$, and define the operation at v_i as $\texttt{i}x_{v_i}$ for $i = 1, \ldots, n/2$, and $\texttt{d}x_{v_{n+1-i}}$ for $i = n/2 + 1, \ldots, n$. It is easy to design a walk that produces *any* subset of $\{1, \ldots, n/2\}$ at either v_1 or v_n, which implies a lower bound of $\Omega(2^{n/2})$ on the size of the decorated graph.

We consider two different walks on G. The *random* walk starts at v_1 and hops from a node to one of its neighbors with equal probability. The main result of this section is that for random walks, maximal compression is possible.

Theorem 3.1 *Successor searching in a one-dimensional random walk can be done in constant expected time per step and linear storage.*

First, however, we consider an *adversarial* walk on G. Note that we can always achieve a running time of $O(\log \log n)$ per step by maintaining a van Emde Boas search structure dynamically [5, 6], so the interesting question is how little storage we need if we are to perform each operation in constant time.

Theorem 3.2 *Successor searching along an n-node cycle in the adversarial model can be performed in constant time per operation, using $O(n^{1+\varepsilon})$ storage, for any fixed $\varepsilon > 0$.*

Before addressing the walk on G, we must consider the following range searching problem (see also [4]). Let $Y = y_1, \ldots, y_n$ be a sequence of n distinct numbers, and consider the points (k, y_k), for $k = 1, \ldots, n$. A query is given by two indices i and j, together with a *type*. The type is defined as follows: the horizontal lines $x \mapsto y_i$ and $x \mapsto y_j$ divide the plane into three unbounded open strips R_1, R_2, and R_3, numbered from top to bottom. For $a = 1, 2, 3$, let $S_a = \{k \in \{1, \ldots, n\} \mid (k, y_k) \text{ lies inside } R_a\}$. The type is specified by the number a together with a direction $\rightarrow$ or $\leftarrow$. The former is called a *right* query, the latter a *left* query. Let us describe the right query: if $S_a = \emptyset$, the result is $\emptyset$. If S_a contains an index larger than i, we want the minimum index in S_a larger than i. If all indices in S_a are less than i, we want the overall minimum index in S_a. The left query is defined symmetrically. See Fig. 5 (left) for an example.

Thus, there are six types of queries, and we specify a query by a triplet (i, j, σ), with σ to being the type. We need the following result, which, as a reviewer pointed out to us, was also discovered earlier by Crochemore et al.[4]. We include our proof below for completeness.

Lemma 3.3 *Any query can be answered in constant time with the help of a data structure of size $O(n^{1+\varepsilon})$, for any $\varepsilon > 0$.*

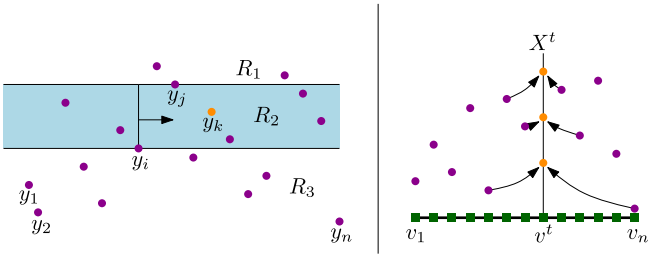


Fig. 5 *Left*: the query $(i, j, (2, \rightarrow))$, we want the leftmost point to the right of y_i in the strip R_2; *right*: the successor data structure. The *squares at the bottom* represent the vertices of the cycle, split at the edge $v_n v_1$ to obtain a better picture. The *dots above the cycle* nodes represent the elements x_{v_i}. The node v^t is the current node, and X^t the active set. We maintain pointers between each element $x \in X^t$ and the closest clockwise and counterclockwise node such that the successor in X^t of the corresponding element is x

Using Lemma 3.3, we can prove Theorem 3.2.

Proof of Theorem 3.2 At any time t, the algorithm has at its disposal: (i) a sorted doubly-linked list of the active set X^t (augmented with ∞); (ii) a (bidirectional) pointer to each $x \in X^t$ from the first node v_k on the circle clockwise from v^t, if it exists, such that $\texttt{succ}_{X^t}(x_{v_k}) = x$ (same thing counterclockwise)—see Fig. 5 (right). Assume now that the data structure of Lemma 3.3 has been set up over $Y = x_{v_1}, \ldots, x_{v_n}$. As the walk enters node v^t at time t, $\texttt{succ}_{X^t}(x_{v^t})$ is thus readily available and we can update X^t in $O(1)$ time. The only remaining question is how to maintain (ii). Suppose that the operation at node v^t is a successor request and that the walk reached v^t clockwise. If x is the successor, then we need to find the first node v_k on the cycle clockwise from v^t such that $\texttt{succ}_{X^t}(x_{v_k}) = x$. This can be handled by two range search queries (i, j, σ): for i, use the index of the current node v^t; and, for j, use the node for x in the first query and the node for x's predecessor in X^t in the second query. An insert can be handled by two such queries (one on each side of v_t), while a delete requires pointer updating, but no range search queries. □

Proof of Lemma 3.3 We define a single data structure to handle all six types simultaneously. We restrict our discussion to the type $(2, \rightarrow)$ from Fig. 5 (left) but kindly invite the reader to check that all other five types can be handled in much the same way. We prove by induction that with $scn^{1+1/s}$ storage, for a large enough constant c, any query can be answered in at most $O(s)$ table lookups. The case $s = 1$ being obvious (precompute all queries), we assume that $s > 1$. Sort and partition Y into consecutive groups $Y_1 < \cdots < Y_{n^{1/s}}$ of size $n^{1-1/s}$ each. We have two sets of tables:

- **Ylinks**: for each $y_i \in Y$, link y_i to the highest-indexed element y_j to the left of i ($j < i$) within each group $Y_1, \ldots, Y_{n^{1/s}}$, wrapping around the strip if necessary (left pointers in Fig. 6 (left)).
- **Zlinks**: for each $y_i \in Y$, find the group Y_{ℓ_i} to which y_i belongs and, for each k, define Z_k as the subset of Y sandwiched between y_i and the smallest (resp. largest) element in Y_k if $k \leq \ell_i$ (resp. $k \geq \ell_i$). Note that this actually defines two sets for Z_{ℓ_i}, so that the total number of Z_k's is really $n^{1/s} + 1$. Link y_i to the lowest-

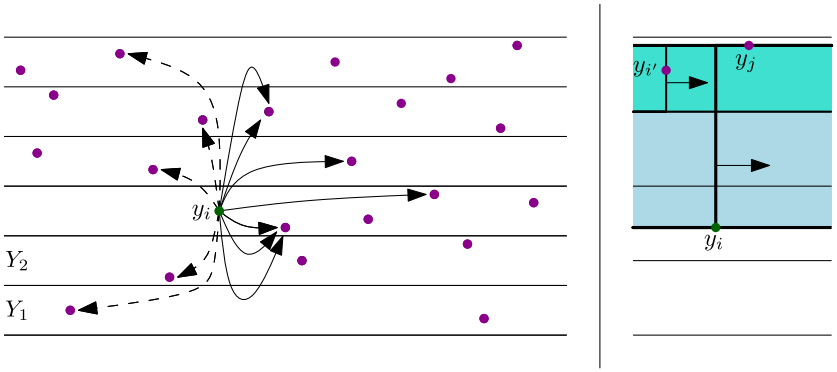


Fig. 6 *Left*: the recursive data structure: The Ylinks (*dashed*) point to the rightmost point to the left of y_i in each strip. The ZLinks point to the leftmost point in each block defined by y_i and a consecutive sequence of strips; *right*: a query (i, j) is decomposed into a part handled by a ZLink and a part that is handled recursively

indexed y_j $(j > i)$ in each Z_k (right pointers in Fig. 6 (left)), again wrapping around if necessary.
– Prepare a data structure of type $s-1$ recursively for each Y_i.

Given a query (i, j) of type $(2, \rightarrow)$, we first check whether it fits entirely within Y_{ℓ_i} and, if so, solve it recursively. Otherwise, we break it down into two subqueries: one of them can be handled directly by using the relevant Zlink. The other one fits entirely within a single Y_k. By following the corresponding Ylink, we find $y_{i'}$ and solve the subquery recursively by converting it into another query (i', j) of appropriate type (Fig. 6 (right)). By induction, it follows that this takes $O(s)$ total lookups and storage

$$dn^{1+1/s} + (s-1)cn^{1/s+(1-1/s)(1+1/(s-1))} = dn^{1+1/s} + (s-1)cn^{1+1/s} \leq scn^{1+1/s},$$

for some constant d and for c large enough, since

$$\left(1 - \frac{1}{s}\right)\left(1 + \frac{1}{s-1}\right) = \frac{s-1}{s}\frac{s}{s-1} = 1. \qquad \square$$

Using Theorem 3.2 together with the special properties of a random walk on G, we can quickly derive the algorithm for Theorem 3.1.

Proof of Theorem 3.1 The idea is to divide up the cycle into $\sqrt{n}$ equal-size paths $P_1, \ldots, P_{\sqrt{n}}$ and prepare an adversarial data structure for each one of them right upon entry. The high cover time of a one-dimensional random walk is then invoked to amortize the costs. De-amortization techniques are then used to make the costs worst-case. The details follow. As soon as the walk enters a new P_k, the data structure of Lemma 3.3 is built from scratch for $\varepsilon = 1/3$, at a cost in time and storage of $O(n^{2/3})$. By merging $L_k = \{ x_{v_i} \mid v_i \in P_k \}$ with the doubly-linked list storing X^t, we can set up all the needed successor links and proceeds just as in Theorem 3.2. This takes $O(n)$

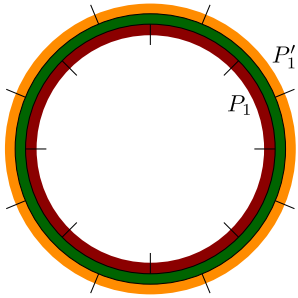


Fig. 7 The parallel tracks on the cycle

time per interpath transition and requires $O(n^{2/3})$ storage. There are few technical difficulties that we now address one by one.

- Upon entry into a new path P_k, we must set up successor links from P_k to X^t, which takes $O(n)$ time. Rather than forcing the walk to a halt, we use a "parallel track" idea to de-amortize these costs (Fig. 7). Cover the cycle with paths P_i' shifted from P_i clockwise by $\frac{1}{2}\sqrt{n}$. and carry on the updates in parallel on both tracks. As we shall see below, we can ensure that updates do not take place simultaneously on both tracks. Therefore, one of them is always available to answer successor requests in constant time.
- Upon entry into a new path P_k (or P_k'), the relevant range search structure must be built from scratch. This work does not require knowledge of X^t and, in fact, the only reason it is not done in preprocessing is to save storage. Again, to avoid having to interrupt the walk, while in P_k we ensure that the needed structures for the two adjacent paths P_{k-1}, P_{k+1} are already available and those for P_{k-2}, P_{k+2} are under construction. (Same with P_k'.)
- On a path, we do not want our range queries to wrap around as in the original structure. Thus, if a right query returns an index smaller than i, or a left query returns an index larger than i, we change the answer to $\emptyset$.
- The range search structure can only handle queries (i, j) for which *both* y_i and y_j are in the ground set. Unfortunately, j may not be, for it may correspond to an item of X^t inserted prior to entry into the current P_k. There is an easy fix: upon entering P_k, compute and store $\mathtt{succ}_{L_k}(x_{v_i})$ for $i = 1, \ldots, n$. Then, simply replace a query (i, j) by (i, j') where j' is the successor (or predecessor) in L_k.

The key idea now is that a one-dimensional random walk has a quadratic cover time [13]; therefore, the expected time between any change of paths on one track and the next change of paths on the other track is $\Theta(n)$. This means that if we dovetail the parallel updates by performing a large enough number of them per walking step, we can keep the expected time per operation constant. This proves Theorem 3.1. □

4 Conclusion

We have presented a new approach to model and analyze restricted query sequences that is inspired by Markov chains. Our results only scratch the surface of a rich body

of questions. For example, even for the simple problem of the adversarial walk on a path, we still do not know whether we can beat van Emde Boas trees with linear space. Even though there is some evidence that the known lower bounds for successor searching on a pointer machine give the adversary a lot of leeway [14], our lower bound technology does not seem to be advanced enough for this setting. Beyond paths and cycles, of course, there are several other simple graph classes to be explored, e.g., trees or planar graphs.

Furthermore, there are more fundamental questions on decorated graphs to be studied. For example, how hard is it to count the number of distinct active sets (or the number of nodes) that occur in the unique sink component of $\mathrm{dec}(G)$? What can we say about the behaviour of the active set in the limit as the walk proceeds randomly? And what happens if we go beyond the dictionary problem and consider the evolution of more complex structures during a walk on the event graph?

Acknowledgements We would like to thank the anonymous referees for their thorough reading of the paper and their many helpful suggestions that have improved the presentation of this paper, as well as for pointing out [4] to us.

References

1. Chassaing, P.: Optimality of move-to-front for self-organizing data structures with locality of references. Ann. Appl. Probab. **3**(4), 1219–1240 (1993)
2. Chazelle, B.: The Discrepancy Method: Randomness and Complexity. Cambridge University Press, Cambridge (2000)
3. Chazelle, B., Mulzer, W.: Markov incremental constructions. Discrete Comput. Geom. **42**(3), 399–420 (2009)
4. Crochemore, M., Iliopoulos, C.S., Kubica, M., Rahman, M.S., Tischler, G., Wale, T.: Improved algorithms for the range next value problem and applications. Theor. Comput. Sci. **434**, 23–34 (2012)
5. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time and linear space. Inf. Process. Lett. **6**(3), 80–82 (1977)
6. van Emde Boas, P., Kaas, R., Zijlstra, E.: Design and implementation of an efficient priority queue. Math. Syst. Theory **10**(2), 99–127 (1976)
7. Hotz, G.: Search trees and search graphs for Markov sources. Elektron. Inf.verarb. Kybern. **29**(5), 283–292 (1993)
8. Kapoor, S., Reingold, E.M.: Stochastic rearrangement rules for self-organizing data structures. Algorithmica **6**(2), 278–291 (1991)
9. Karlin, A.R., Phillips, S.J., Raghavan, P.: Markov paging. SIAM J. Comput. **30**(3), 906–922 (2000)
10. Konneker, L.K., Varol, Y.L.: A note on heuristics for dynamic organization of data structures. Inf. Process. Lett. **12**(5), 213–216 (1981)
11. Lam, K., Leung, M.Y., Siu, M.K.: Self-organizing files with dependent accesses. J. Appl. Probab. **21**(2), 343–359 (1984)
12. Levin, D.A., Peres, Y., Wilmer, E.L.: Markov Chains and Mixing Times. Am. Math. Soc., Providence (2009)
13. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press, Cambridge (1995)
14. Mulzer, W.: A note on predecessor searching in the pointer machine model. Inf. Process. Lett. **109**(13), 726–729 (2009)
15. Phatarfod, R.M., Pryde, A.J., Dyte, D.: On the move-to-front scheme with Markov dependent requests. J. Appl. Probab. **34**(3), 790–794 (1997)
16. Schulz, F., Schömer, E.: Self-organizing data structures with dependent accesses. In: Proceedings of the 23rd International Colloquium on Automata, Languages, and Programming (ICALP), pp. 526–537 (1996)
17. Shedler, G.S., Tung, C.: Locality in page reference strings. SIAM J. Comput. **1**(3), 218–241 (1972)
18. Vitter, J.S., Krishnan, P.: Optimal prefetching via data compression. J. ACM **43**(5), 771–793 (1996)