Nordic Journal of Computing 3(1996), 238–255.

PARTIALLY PERSISTENT DATA STRUCTURES OF BOUNDED DEGREE WITH CONSTANT UPDATE TIME

GERTH STØLTING BRODAL
BRICS, Department of Computer Science, University of Aarhus
Ny Munkegade, DK-8000 Århus C, Denmark
gerth@brics.dk

Abstract. The problem of making bounded in-degree and out-degree data structures partially persistent is considered. The node copying method of Driscoll *et al.* is extended so that updates can be performed in *worst-case* constant time on the pointer machine model. Previously it was only known to be possible in amortised constant time.

The result is presented in terms of a new strategy for Dietz and Raman's dynamic two player pebble game on graphs.

It is shown how to implement the strategy and the upper bound on the required number of pebbles is improved from $2b+2d+O(\sqrt{b})$ to $d+2b$, where b is the bound of the in-degree and d the bound of the out-degree. We also give a lower bound that shows that the number of pebbles depends on the out-degree d.

CR Classification: E.1, F.2.2

Key words: data structures, partial persistence, pebble game, lower bounds

1. Introduction

This paper describes a method to make data structures *partially persistent.* A partially persistent data structure is a data structure in which old versions are remembered and can always be inspected. However only the latest version of the data structure can be modified.

An interesting application of a partially persistent data structure is given in [4] where the planar point location problem is solved by an elegant application of partially persistent search trees. The method given in [4] can be generalised to make arbitrary bounded in-degree data structures partially persistent [2].

As in [2], the data structures we consider will be described in the pointer machine model, i.e. they consist of records with a constant number of fields each containing a unit of data or a pointer to another record. The data structures can be viewed as graphs with bounded out-degree. In the following let d denote this bound.

The main assumption is that the data structures also have *bounded in-degree.* Let b denote this bound. Not all data structures satisfy this con-

Received March 9, 1995; accepted June 28, 1996.

straint — but they can be converted to do it: Replace nodes by balanced binary trees, so that all original pointers that point to a node now instead point to the leafs in the tree substituted into the data structure instead of the node, and store the node's original information in the root of the tree. The assumption can now be satisfied by letting at most a constant number of pointers point to the same leaf. The drawback of this approach is that the time to access a node v is increased from $O(1)$ to $O(\log b_v)$ where b_v is the original in-degree of v.

The problem with the method presented in [2, 4] is that an update of the data structure takes *amortised* time $O(1)$, in the worst case it can be $O(n)$ where n is the size of the current version of the data structure.

In this paper we describe how to extend the method of [2, 4] so that an update can be done in *worst case* constant time. The main result of this paper is:

THEOREM 1. *It is possible to implement partially persistent data structures with bounded in-degree (and out-degree) such that each update step and access step can be performed in* worst case *time* $O(1)$.

The problem can be restated as a dynamic two player pebble game on dynamic directed graphs, which was done by Raman and Dietz in [1]. In fact, it is this game we consider in this paper.

The central rules of the game are that player **I** can add a *pebble* to an arbitrary node and player **D** can remove all pebbles from a node provided he places a pebble on all of the node's predecessors. For further details refer to Sect. 3. The goal of the game is to find a strategy for player **D** that can guarantee that the number of pebbles on all nodes are bounded by a constant M. Dietz and Raman gave a strategy which achieved $M \leq 2b + 2d + O(\sqrt{b})$ — but they were not able to implement it efficiently which is necessary to remove the amortisation from the original persistency result.

In this paper we improve the bound to $M = d+2b$ by a simple modification of the original strategy. In the static case (where the graph does not change) we get $M = d + b$.

We also consider the case where the nodes have different bounds on their in- and out-degree. In this case we would like to have $M_v = f(b_v, d_v)$ where $f : N^2 \to N$ is a monotonically increasing function. Hence only nodes with a high in-degree should have many pebbles. We call strategies with this property for *locally adaptive*. In fact, the strategy mentioned above satisfies that $M_v = d_v + 2b_v$ in the dynamic game and $M_v = d_v + b_v$ in the static game.

By an *efficiently implementable strategy* we mean a strategy that can be implemented such that the move of player **D** can be performed in time $O(1)$ if player **D** knows where player **I** performed his move. In the following we call such strategies implementable.

The implementable strategies we give do not obtain such good bounds. Our first strategy obtains $M = 2bd+1$, whereas the second is locally adaptive and obtains $M_v = 2b_v d_v + 2b_v - 1$.

The analysis of our strategies are all tight — we give examples which match the upper bounds. The two efficiently implementable strategies have simple implementations with small constant factors.

We also give lower bounds for the value of M which shows that M depends both on b and d for all strategies. More precisely we show that (we define $\log x = \max\{1, \log_2 x\}$):

$$M \geq \max\{b+1, \lfloor \alpha + \sqrt{2\alpha - 7/4} - 1/2 \rfloor, \left\lceil \frac{\log \frac{2}{3}d}{\log\log \frac{2}{3}d} - 1 \right\rceil\},$$

where $\alpha = \min\{b, d\}$.

The paper is organised as follows. In Sect. 2 we describe the method of [2, 4] and in Sect. 3 we define the dynamic graph game of [1]. In Sect. 4 we give the new game strategy for player **D** which is implementable. The technical details which are necessary to implement the strategy are described in Sect. 5 and the strategy is analysed in Sect. 6. In Sect. 7 we give a locally adaptive strategy and in Sect. 8 we give a locally adaptive strategy which is implementable. Finally, the lower bound for M is given in Sect. 9.

2. The node copying method

In this section we briefly review the method of [2, 4]. For further details we refer to these articles. The purpose of this section is to motivate the game that is defined in Sect. 3, and to show that if we can find a strategy for this game and implement it efficiently, then we can also remove the amortisation from the partially persistency method described below.

The *ephemeral data structure* is the underlying data structure we want to make partially persistent. In the following we assume that we have access to the ephemeral data structure through a finite number of entry pointers. For every update of the data structure we increase a version counter which contains the number of the current version.

When we update a node v we cannot destroy the old information in v because this would not enable us to find the old information again. The idea is now to add the new information to v together with the current version number. So if we later want to look at an old version of the information, we just compare the version numbers to find out which information was in the node at the time we are looking for. This is in very few words the idea behind the so called *fat node* method.

An alternative to the previous approach is the *node copying* method. This method allows at most a constant number (M) of additional information in each node (depending on the size of b). When the number of different copies of information in a node gets greater than M we make a copy of the node and the old node now becomes *dead* because new pointers to the node has to point to the newly created copy. In the new node we only store the information of the dead node which exists in the current version of the ephemeral data structure. We now have to update all the nodes in the current version of the data structure which have pointers to the node that has now become dead. These pointers should be updated to point to the newly created node instead — so we recursively add information to all the predecessors of the node that we have copied. The copied node does not contain any additional information.

3. The dynamic graph game

The game Dietz and Raman defined in [1] is played on a directed graph $G = (V, E)$ with bounded in-degree and out-degree. Let b be the bound of the in-degree and d the bound of the out-degree. W.l.o.g. we do not allow self-loops or multiple edges. To each node a number of pebbles is associated, denoted by P_v. The *dynamic graph game* is now a game where two players **I** and **D** alternate to move. The moves they can perform are:

Player **I**:

- **a)** add a pebble to an arbitrary node v of the graph or
- **b)** remove an existing edge (v, u) and create a new edge (v, w) without violating the in-degree constraint on w, and place a pebble on the node v.

Player **D**:

- **c)** do nothing or
- **d)** remove all pebbles from a node v and place a new pebble on all the predecessors of v. This is denoted by ZERO(v).

The goal of the game is to show that there exists a constant M and a strategy for player **D** such that, whatever player **I** does, the maximum number of pebbles on any node after the move of player **D** is bounded by M. In the static version of the game player **I** can only do moves of type a).

The relationship between partially persistent data structures and the pebble game defined is the following. The graph of the pebble game corresponds to the current version of an ephemeral data structure. A pebble corresponds to additional information stored in a node. A move of player **I** of type a) corresponds to updating a data field in the ephemeral data structure and a move of type b) corresponds to updating a pointer field in the ephemeral data structure. A ZERO operation performed by player **D** corresponds to the copying of a node in the node copying method. The pebbles placed on the predecessor nodes correspond to updating the incoming pointers of the corresponding node copied in the persistent data structure.

The existence of a strategy for player **D** was shown in [1], but the given strategy could not be implemented efficiently (i.e. the node v in d) could not be located in time O(1)).

THEOREM 2. (DIETZ AND RAMAN [1]) *A strategy for player* **D** *exists that achieves* $M = \mathrm{O}(b + d)$.

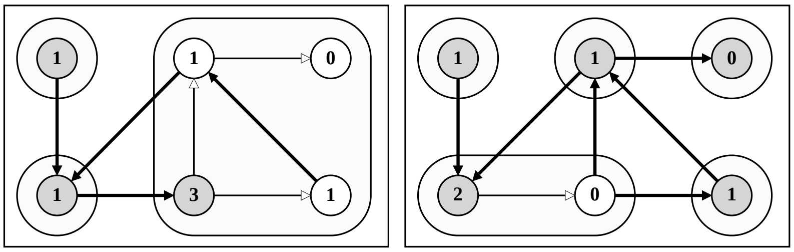

Fig. 1: The effect of performing a BREAK operation. The numbers are the number of pebbles on the nodes.

4. The strategy

We now describe our new strategy for player **D**. We start with some definitions. We associate the following additional information with the graph G.

- Edges are either *black* or *white*. Nodes have at most one incoming white edge. There are no white cycles.
- Nodes are either *black* or *white*. Nodes are white if and only if they have an incoming white edge.

The definitions give in a natural way rise to a partition of the nodes into components: two nodes connected by a white edge belong to the same component. It is easily seen that a component is a rooted tree of white edges with a black root and all other nodes white. A single black node with no adjacent white edge is also a component. We call a component consisting of a single node a *simple component* and a component with more than one node a *non simple component*. See Fig. 1 (on the left) for an example of a graph with two simple components and one non simple component.

To each node v we associate a queue Q_v containing the predecessors of v. The queue operations used in the following are:

- $\mathrm{ADD}(Q_u, v)$ adds v to the back of Q_u.
- $\mathrm{DELETE}(Q_u, v)$ removes v from Q_u.
- $\mathrm{ROTATE}(Q_u)$ moves the front element v of Q_u to the back of Q_u, and returns v.

The central operation in our strategy is now the following BREAK operation. The component containing v is denoted C_v.

procedure $\mathrm{BREAK}(C_v)$
  $r \leftarrow$ the root of C_v
  colour all nodes and edges in C_v black
  if $Q_r \neq \emptyset$ **then**
    colour r and $(\mathrm{ROTATE}(Q_r), r)$ white
  endif
  $\mathrm{ZERO}(r)$
end.

The effect of performing BREAK on a component is that the component is broken up into simple components and that the root of the original component is appended to the component of one of its predecessors (if any). An example of the application of the BREAK operation is shown in Fig. 1.

A crucial property of BREAK is that all nodes in the component change colour (except for the root when it does not have any predecessors, in this case we per definition say that the root changes its colour twice).

Our strategy is now the following (for simplicity we give the moves of player **I** and the counter moves of player **D** as procedures).

```
procedure AddPebble(v)
    place a pebble on v
    Break(C_v)
end.

procedure MoveEdge((v,u),(v,w))
    place a pebble on v
    if (v,u) is white then
        Break(C_v)
        Delete(Q_u, v)
        replace (v,u) with (v,w) in E
        Add(Q_w, v)
    else
        Delete(Q_u, v)
        replace (v,u) with (v,w) in E
        Add(Q_w, v)
        Break(C_v)
    endif
end.
```

In MOVEEDGE the place where we perform the BREAK operation depends on the colour of the edge (v, u) being deleted. This is to guarantee that we only remove black edges from the graph (in order not to have to split components).

Observe that each time we apply ADDPEBBLE or MOVEEDGE to a node v we find the root of C_v and zero it. We also change the colour of all nodes in C_v — in particular we change the colour of v. Now, every time a black node becomes white it also becomes zeroed, so after two **I** moves have placed pebbles on v, v has been zeroed at least once. That the successors of a node v cannot be zeroed more than O(1) times and therefore cannot place pebbles on v without v getting zeroed is shown in Sect. 6. The crucial property is the way in which BREAK colours nodes and edges white. The idea is that a successor u of v cannot be zeroed more than O(1) times before the edge from (v, u) will become white. If (v, u) is white both v and u belong to the same component, and therefore u cannot change colour without v changing colour.

In Sect. 5 we show how to implement BREAK in worst case time O(1) and in Sect. 6 we show that the approach achieves that $M = O(1)$.

5. The new data structure

The procedures in Sect. 4 can easily be implemented in worst case time O(1) if we are able to perform the BREAK operation in constant time. The central idea is to represent the colours indirectly so that all white nodes and edges in a component points to the same variable. All the nodes and edges can now be made black by setting this variable to black.

A *component record* contains two fields. A colour field and a pointer field. If the colour field is white the pointer field will point to the root of the component.

To each node and edge is associated a pointer *cr* which points to a component record. We will now maintain the following invariant.

- The *cr* pointer of each black edge and each node forming a simple component will point to a component record where the colour is black and the root pointer is the null pointer. Hence, there is a component record for each non simple component, but several black edges and nodes forming a simple component can share the same component record.
- For each non simple component there exist exactly one component record where the colour is white and the root pointer points to the root of the component. All nodes and white edges in this component point to this component record.

An example of component records is shown in Fig. 2. Notice that the colour of an edge e is simply $e.cr.colour$ so the test in MOVEEDGE is trivial to implement. The implementation of BREAK is now:

procedure BREAK(v)
    if $v.cr.colour = black$ **then**
        $r \leftarrow v$
    else
        $r \leftarrow v.cr.root$
        $v.cr.colour \leftarrow black$
        $v.cr.root \leftarrow \perp$
    endif
    if $r.Q \neq \emptyset$ **then**
        $u \leftarrow$ ROTATE($r.Q$)
        if $u.cr.colour = black$ **then**
            $u.cr \leftarrow$ *new-component-record*($white, u$)
        endif
        $r.cr \leftarrow (u, r).cr \leftarrow u.cr$
    endif
    ZERO(r)
end.

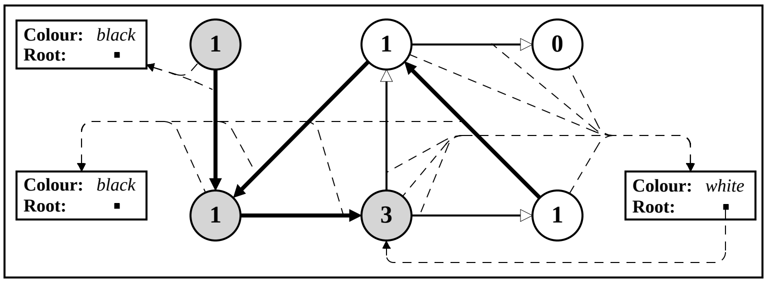

Fig. 2: A graph with component records.

From the discussion of the node copying method in Sect. 2 it should be clear that the above described data structure also applies to this method.

6. The analysis

THEOREM 3. *The player* **D** *strategy given in Sect. 4 achieves* $M = 2bd + 1$.

PROOF. A direct consequence of Lemmas 1 and 2. □

LEMMA 1. *The player* **D** *strategy given in Sect. 4 achieves* $M \leq 2bd + 1$.

PROOF. Let the first operation (either an ADDPEBBLE or MOVEEDGE operation) be performed at time 1, the next at time 2 and so on.

Assume that when the game starts all nodes are black and there are no pebbles on any node.

Fix an arbitrary node v at an arbitrary time t_{now}. Let t_{last} denote the last time before t_{now} when v was zeroed (if v has never been zeroed let t_{last} be 0). In the following we want to bound the number of pebbles placed on v in the interval $]t_{last}, t_{now}[$. In this interval v cannot change its colour from black to white because this would zero v.

Assume without loss of generality that v is white at the end of time t_{last}, that at time $t_{break} \in]t_{last}, t_{now}[$ a BREAK(C_v) is performed and (therefore) at time t_{now} v is black (it is easy to see that all other cases are special cases of this case).

Note that the only time an ADDPEBBLE(v) or MOVEEDGE((v,u), (v,w)) operation can be performed is at time t_{break} because these operations force the colour of v to change. Therefore, v's successors are the same in the interval $]t_{last}, t_{break}[$, and similarly for $]t_{break}, t_{now}[$.

We will handle each of the two intervals and the time t_{break} separately. Let us first consider the interval $]t_{last}, t_{break}[$. Let w be one of v's successors in this interval. w can be zeroed at most b times before it will be blocked by a white edge from v (w cannot change the colour without changing the colour of v), because after at most $b-1$ ZERO(w) operations, v will be the first element in Q_w.

So a successor of v can be zeroed at most bd times throughout the first interval which implies that at most bd pebbles can be placed on v during the first interval. For $]t_{break}, t_{now}[$ we can repeat the same argument so at most bd pebbles will be placed on b during this interval too.

We now just have to consider the operation at time t_{break}. The colour of v changes so a BREAK(C_v) is performed. There are three possible reasons for that: a) An ADDPEBBLE(v) operation is performed, b) a MOVEEDGE $((v,u),(v,w))$ is performed or c) one of the operations is performed on a node different from v. In a) and b) we first add a pebble to v and then perform a BREAK(C_v) operation and in c) we first add a pebble to another node in C_v and then do BREAK(C_v). The BREAK operation can add at most one pebble to v when we perform a ZERO operation to the root of C_v (because we do not allow multiple edges) so at most two pebbles can be added to v at time t_{break}.

We have now shown that at time t_{now} the number of pebbles on v can be at most $2bd+2$. This is nearly the claimed result. To decrease this bound by one we have to analyse the effect of the operation performed at time t_{break} more carefully.

What we prove is that when two pebbles are placed on v at time t_{break} then at most $bd-1$ pebbles can be placed on v throughout $]t_{break}, t_{now}[$. This follows if we can prove that there exists a successor of v that cannot be zeroed more than $b-1$ times in the interval $]t_{break}, t_{now}[$.

In the following let r be the node that is zeroed at time t_{break}. We have the following cases to consider:

i) ADDPEBBLE(v) and BREAK(r) places a pebble on v. Now r and one of its incoming edges are white. So r can be zeroed at most $b-1$ times before (v,r) will become white and block further ZERO(r) operations.

ii) MOVEEDGE($(v,u),(v,w)$) and ZERO(r) places a pebble on v. Depending on the colour of (v,u) we have two cases:

- **a)** (v,u) is white. Therefore u is white and $r \neq u$. Since we perform BREAK(r) before we modify the pointers we have that $r \neq w$. So as in i) r can be zeroed at most $b-1$ times throughout $]t_{break}, t_{now}[$.
- **b)** (v,u) is black. Since BREAK is the last operation we do, the successors of v will be the same until after t_{now}, so we can argue in the same way as i) and again get that r can be zeroed at most $b-1$ times throughout $]t_{break}, t_{now}[$.

We conclude that no node will ever have more than $2bd+1$ pebbles. □

LEMMA 2. *The player* **D** *strategy given in Sect. 4 achieves* $M \geq 2bd+1$.

PROOF. Let $G=(V,E)$ be the directed graph given by $V=\{r, v_1, \ldots, v_b, w_1, \ldots, w_d\}$ and $E=\{(r,v_b)\} \cup \{(v_i,w_j) | i \in \{1,\ldots,b\} \wedge j \in \{1,\ldots,d\}\}$. The graph is shown in Fig. 3. Initially all nodes in V are black and all queues

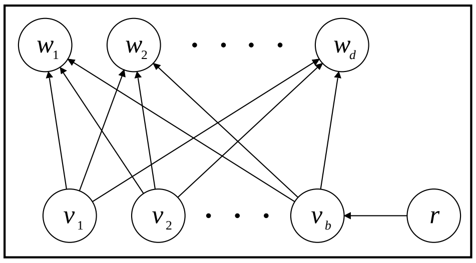

Fig. 3: A graph which can force M to become $2bd+1$.

Q_{w_i} contain the nodes $(v_1, \ldots, v_b)$. We will now force the number of pebbles on v_b to become $2bd+1$.

First place one pebble on v_b — so that v_b becomes white. Then place $2b-1$ pebbles on each w_j. There will now be bd pebbles on v_b and all the edges (v_b, w_j) are white. Place one new pebble on v_b and place another $2b-1$ pebbles on each w_j. Now there will be $2bd+1$ pebbles on v_b. □

7. A simple locally adaptive strategy

In this section we present a simple strategy that is adaptive to the local in- and out-degree bounds of the nodes. It improves the bound achieved in [1]. The main drawback is that the strategy cannot be implemented efficiently in the sense that the node to be zeroed cannot be found in constant time. In Sect. 8 we present an implementable strategy that is locally adaptive but does not achieve as good a bound on M.

Let d_v denote the bound of the out-degree of v and b_v the bound of the in-degree. Define M_v to be the best bound player **D** can guarantee on the number of pebbles on v. We would like to have that $M_v = f(b_v, d_v)$ for a monotonic function $f : N^2 \to N$.

The strategy is quite simple. To each node v we associate a queue Q_v containing the predecessors of v and a special element ZERO. Each time the ZERO element is rotated from the front of the queue the node is zeroed.

The simple adaptive strategy
    if the **I**-move deletes (v, u) and adds (v, w) **then**
        DELETE(Q_u, v)
        ADD(Q_w, v)
    endif
    while $(v' \leftarrow$ ROTATE$(Q_v))\neq$ZERO **do** $v \leftarrow v'$ **od**
    ZERO(v)
end.

Notice that the strategy does not use the values of b_v and d_v explicitly. This gives the strategy the nice property that we can allow b_v and d_v to change dynamically.

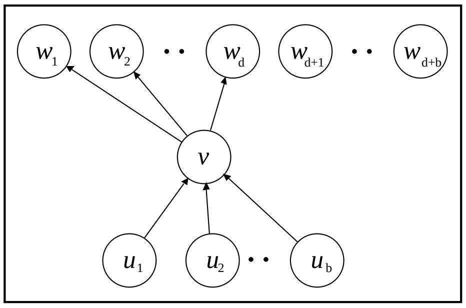

Fig. 4: A graph which can force M to become $d_v + 2b_v$.

The best bound Dietz and Raman could prove for their strategy was $M \leq 2b + 2d + \mathrm{O}(\sqrt{b})$. The next theorem shows that the simple strategy above achieves a bound of $M_v = d_v + 2b_v$. If the graph is static the bound improves to $M_v = d_v + b_v$.

THEOREM 4. *For the simple adaptive strategy we have that* $M_v = d_v + 2b_v$. *In the static case this improves to* $M_v = d_v + b_v$.

PROOF. Each time we perform ADDPEBBLE(v) or MOVEEDGE$((v,u), (v,w))$ we rotate Q_v. It is possible to rotate Q_v at most b_v times without zeroing v. This implies that between two ZERO(v) operations at most b_v MOVEEDGE operations can be performed on outgoing edges of v. Therefore, v can have had at most $b_v + d_v$ different successors between two ZERO(v) operations. Between two zeroings of a successor w of v, Q_v must have been rotated because ROTATE(Q_w) returned v in the while-loop, this is because the ZERO element is moved to the back of Q_w when w is being zeroed. So except for the first zeroing of w all zeroings of w will be preceded by a rotation of Q_v.

For each operation performed on v we both place a pebble on v and rotate Q_v. So the bound on the number of rotations of Q_v gives the following bound on the number of pebbles that can be placed on v: $M_v \leq (d_v + b_v) + b_v$.

In the static case the number of different successors between two ZERO (v) operations is d_v so in the same way we get the bound $M_v \leq d_v + b_v$.

It is easy to construct an example that matches this upper bound. Let $G = (V, E)$ where

$$\begin{aligned} V &= \{v, u_1, \ldots, u_{b_v}, w_1, \ldots, w_{d_v}, w_{d_v+1}, \ldots, w_{d_v+b_v}\} \text{ and} \\ E &= \{(u_i, v) | i \in \{1, \ldots, b_v\}\} \cup \{(v, w_i) | i \in \{1, \ldots, d_v\}\}. \end{aligned}$$

The graph is shown in Fig. 4.

At the beginning all nodes are black and the ZERO element is at the front of each queue. The sequence of operations which will force the number of pebbles on v to become $d_v + 2b_v$ is the following: ADDPEBBLE on $v, w_1, \ldots, w_{d_v}$, followed by $\text{MOVEEDGE}((v, w_{i-1+d_v}), (v, w_{i+d_v}))$ and $\text{ADDPEBBLE}(w_{i+d_v})$ for $i = 1, \ldots, b_v$. The matching example for the static case is constructed in a similar way. □

8. A locally adaptive data structure

We will now describe a strategy that is both implementable and locally adaptive. The data structure presented in Sect. 4 and Sect. 5 is *not* locally adaptive, because when redoing the analysis with local degree constraints we get the following bound for the static version of the game:

$$M_v = 1 + 2 \sum_{\{w \mid (v,w) \in E\}} b_w.$$

The solution to this problem is to incorporate a ZERO element into each of the queues Q_v as in Sect. 7 and then only zero a node when ROTATE returns this element. We now have the following BREAK operation:

procedure BREAK(C_v)
    $r \leftarrow$ the root of C_v
    colour all nodes and edges in C_v black
    $w \leftarrow$ ROTATE(Q_r)
    if w =ZERO **then**
        ZERO(r)
        $w \leftarrow$ ROTATE(Q_r)
    endif
    if $w \neq$ZERO **then**
        colour r and (w, r) white
    endif
end.

The implementation is similar to the implementation of Sect. 5.

The next theorem shows that the number of pebbles on a node v with this strategy will be bounded by $M_v = 2b_v d_v + 2b_v - 1$, so only nodes with large in-degree (or out-degree) can have many pebbles.

THEOREM 5. *The above strategy for player* **D** *achieves* $M_v = 2b_v d_v + 2b_v - 1$.

PROOF. The proof follows the same lines as in the proof of Theorem 3. A node v can change its colour at most $2b_v - 1$ times between two zeroings. We then have that the number of ADDPEBBLE and MOVEEDGE operations performed on v is at most $2b_v - 1$.

The time interval between two ZERO(v) operations is partitioned into $2b_v$ intervals and that v changes its colour only on the boundary between two intervals. In each of the intervals each successor w of v can be zeroed at most once before it will be blocked by a white edge from v.

So when we restrict ourselves to the static case we have that each successor gets zeroed at most $2b_v$ times. Hence the successors of v can place at most $2b_v d_v$ pebbles on v.

Each AddPebble operation places a pebble on v, so for the static case, the total number of pebbles on v is bounded by $M_v = 2b_v d_v + 2b_v - 1$.

We now only have to show that a MoveEdge$((v,u),(v,w))$ operation does not affect this analysis. We have two cases to consider. If u has been zeroed in the last interval then u will either be blocked by a white edge from v or v appears before the Zero element in Q_u and therefore none of the Break operations in MoveEdge can result in a Zero(u). If u has not been zeroed then it is allowed to place a pebble on v in the MoveEdge operation. If the Break operation forces a Zero(w) to place a pebble on v then w cannot place a pebble on v during the next time interval. So we can conclude that the analysis still holds.

The matching lower bound is given in the same way as in Theorem 4. □

9. A lower bound

In this section we will only consider the static game.

Raman states in [3] that "the dependence on d of M appears to be an artifact of the proof (for the strategy of [1])". Theorem 6 shows that it is not an artifact of the proof, but that the value of M always depends on the value of b and d.

It is shown in [2] that $M \leq b$ holds in the amortised sense, so in that game M does *not* depend of d.

Theorem 6. *For $b \geq 1$ and all player $\mathbf{D}$ strategies we have:*

$$M \geq \max\{b+1, \lfloor \alpha + \sqrt{2\alpha - 7/4} - 1/2 \rfloor, \left\lceil \frac{\log \frac{2}{3}d}{\log\log \frac{2}{3}d} - 1 \right\rceil\},$$

where $\alpha = \min\{b, d\}$.

Proof. Immediate consequence of Lemma 3 and 4 and Corollary 1. □

Lemma 3. *For $b, d \geq 1$ and all player $\mathbf{D}$ strategies we have $M \geq b+1$.*

Proof. We will play the game on a convergent tree with l levels where each node has exactly b incoming edges. The player $\mathbf{I}$ strategy is simple, it just places the pebbles on the root of the tree.

The root has to be zeroed at least once for each group of $M+1$ AddPebble operations. So at least a fraction $\frac{1}{M+1}$ of the time will be spent on zeroing the root. At most M pebbles can be placed on any internal node before the next Zero operation on that node, because we do not perform AddPebble on internal nodes. So a node on level 1 has to be zeroed at least once for every M ZERO operation on the root. Zeroing a node at level 1 takes at least $\frac{1}{M(M+1)}$ of the time, and in general, zeroing a node at level i takes at least $\frac{1}{M^i(M+1)}$ of the time.

Because the number of nodes in each level of the tree increases by a factor b we now have the following constraint on M:

$$\sum_{i=0}^{l} \frac{b^i}{M^i(M+1)} = \frac{1}{M+1} \sum_{i=0}^{l} \left(\frac{b}{M}\right)^i \leq 1.$$

By letting $l \gg M$ we get the desired result $M \geq b+1$. If $d=1$, it follows from Theorem 4 that this bound is tight. □

LEMMA 4. *For $b, d \geq 1$ and all player* **D** *strategies we have:*

$$M \geq \left\lceil \frac{\log \frac{2}{3}d}{\log\log \frac{2}{3}d} - 1 \right\rceil.$$

PROOF. We will play the game on the following graph $G = (V, E)$ where $V = \{r, v_1, \ldots, v_d\}$ and $E = \{(r, v_1), \ldots, (r, v_d)\}$. The adversary strategy we will use for player **I** is to cyclically place pebbles on the subset of the v_i's which have not been zeroed yet. The idea is that for each cycle at least a certain fraction of the nodes will not be zeroed.

We start by considering how many nodes cannot be zeroed in one cycle. Let the number of nodes not zeroed at the beginning of the cycle be k. Each time one of the v_i's is zeroed a pebble is placed on r, so out of $M+1$ zeroings at least one will be a ZERO(r). So we have that at least $\lfloor \frac{k}{M+1} \rfloor$ of the nodes are still not zeroed at the end of the cycle. So after i cycles we have that the number of nodes not zeroed is at least (the number of floors is i):

$$\left\lfloor \cdots \left\lfloor \left\lfloor \frac{d}{M+1} \right\rfloor \frac{1}{M+1} \right\rfloor \cdots \frac{1}{M+1} \right\rfloor.$$

By the definition of M, we know that all nodes will be zeroed after $M+1$ cycles, so we have the following equation (the number of floors is $M+1$):

$$\left\lfloor \cdots \left\lfloor \left\lfloor \frac{d}{M+1} \right\rfloor \frac{1}{M+1} \right\rfloor \cdots \frac{1}{M+1} \right\rfloor = 0.$$

Lemma 3 gives us that $M \geq 2$. By induction on the number of floors is it easy to show that omitting the floors increases the result at most 3/2. Hence, we have

$$\frac{d}{(M+1)^{M+1}} \leq 3/2.$$

So the minimum solution of M for this inequality will be a lower bound for M. It is easy to see that this minimum solution has to be at least $\frac{\log \frac{2}{3}d}{\log\log \frac{2}{3}d} - 1$.
□

LEMMA 5. *For all* $\mathbf{D}$ *strategies where* $b = d$ *we have:*

$$M \geq \lfloor b + \sqrt{2b - 7/4} - 1/2 \rfloor.$$

PROOF. For $b = d = 0$ the lemma is trivial. The case $b = d = 1$ is true by Lemma 3. In the following we assume $b = d \geq 2$.

Again, the idea is to use player $\mathbf{I}$ as an adversary that forces the number of pebbles to become large on at least one node.

The graph we will play the game on is a clique of size $b + 1$. For all nodes u and v both (u, v) and (v, u) will be edges of the graph and all nodes will have in- and out-degree b. Each ZERO operation of player $\mathbf{D}$ will remove all pebbles from a node of the graph and place one pebble on all the other nodes.

At a time given $P_0, P_1, \ldots, P_b$ will denote the number of pebbles on each of the $b + 1$ nodes — in increasing order, so P_b will denote the number of pebbles on the node with the largest number of pebbles.

Let c_1, c_2 and c_3 denote constants characterising the adversary's strategy. The following invariants will hold from a certain moment of time to be defined later:

$$\begin{array}{ll}
I_1: & i \leq j \Rightarrow P_i \leq P_j, \\
I_2: & P_i \geq i, \\
I_3: & \left\{ \begin{array}{ll} P_{c_1+c_2-i} \geq c_1 + c_2 - 1 & \text{for } 1 \leq i \leq c_3, \\ P_{c_1+c_2-i} \geq c_1 + c_2 - 2 & \text{for } c_3 < i \leq c_2, \end{array} \right. \\
I_4: & 1 \leq c_3 \leq c_2 \quad \text{and} \quad c_1 + c_2 \leq b + 1.
\end{array}$$

I_1 is satisfied per definition. I_2 is not satisfied initially but after the first b ZERO's will be satisfied. This is easily seen. The nodes that have not been zeroed will have at least b pebbles and the nodes that have been zeroed can be ordered according to the last time they were zeroed. A node followed by i nodes in this order will have at least i pebbles because each of the following (at least) i zeroings will place a pebble on the node.

We can now satisfy I_3 and I_4 by setting $c_1 = c_2 = c_3 = 1$ so now we have that all the four invariants are satisfied after the first b ZERO operations.

Fig. 5 illustrates the relationship between c_1, c_2 and c_3 and the number of pebbles on the nodes. The figure only shows the pebbles which are guaranteed to be on the nodes by the invariants. The idea is to build a *block* of nodes which all have the same number of pebbles. These nodes are shown as a dashed box in Fig. 5. The moves of player $\mathbf{I}$ and $\mathbf{D}$ affect this box. A player $\mathbf{I}$ move will increase the block size whereas a player $\mathbf{D}$ move will push the block upwards. In the following we will show how large the block can be forced to be.

We will first consider an ADDPEBBLE operation. If $c_3 < c_2$ we know that on node $c_1 + c_2 - c_3 - 1$ (in the current ordering) there are at least $c_1 + c_2 - 2$ pebbles so by placing a pebble on the node $c_1 + c_2 - c_3 - 1$ we can increase c_3 by one and still satisfy the invariants $I_1, \ldots, I_4$. There are

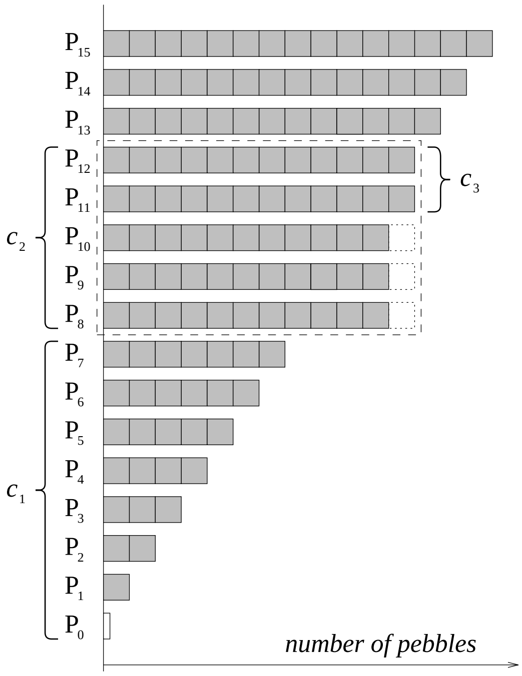

Fig. 5: The adversary's strategy.

three cases to consider. If the node $c_1 + c_2 - c_3 - 1$ already has $c_1 + c_2 - 1$ pebbles we increase c_3 by one and try to place the pebble on another node. If $c_3 = c_2$ and $c_1 + c_2 < b + 1$ we can increase c_2 by one and set $c_3 = 1$ and then try to place the pebble on another node. If we have that $c_2 = c_3$ and $c_1 + c_2 = b + 1$ we just place the pebble on an arbitrary node — because the block has reached its maximum size.

Whenever player **D** does a ZERO operation we can easily maintain the invariant by just increasing c_1 by one — as long as $c_1 + c_2 < b + 1$. Here we have three cases to consider. Let i denote the number of the node that player **D** zeroes. We will only consider the case when $c_1 \leq i < c_1 + c_2$, the cases $0 \leq i < c_1$ and $c_1 + c_2 \leq i \leq b$ are treated in a similar way. The values of the Ps after the ZERO operation are: $P'_0 = 0, P'_1 = P_0 + 1, \ldots, P'_i = P_{i-1} + 1, P'_{i+1} = P_{i+1} + 1, \ldots, P'_b = P_b + 1$. So because I_2 and I_3 were satisfied before the ZERO operation it follows that when we increase c_1 by one the invariant will still be satisfied after the ZERO operation.

We will now see how large the value of c_2 can become before $c_1 + c_2 = b + 1$. We will allow the last move to be a player **I** move.

We let x denote the maximum value of c_2 when $c_1 + c_2 = b + 1$. At this point we have that $c_1 = b + 1 - x$. Initially we have that $c_1 = 1$. Each ZERO operation can increase c_1 by at most one so the maximum number of ADDPEBBLE operations we can perform is $1 + ((b + 1 - x) - 1) = b + 1 - x$.

It is easily seen that the worst case number of pebbles we have to add to bring c_2 up to x is $1+\sum_{i=2}^{x-1}(i-1)$ — because it is enough to have two pebbles in the last column of the block when we are finished.

So the size of $x \geq 0$ is now constrained by:

$$1+\sum_{i=2}^{x-1}(i-1) \quad \leq \quad b+1-x.$$

Hence, we have $x \geq \lfloor 1/2+\sqrt{2b-7/4} \rfloor$. Let $i \in \{0,1,\ldots,x-1\}$ denote the number of ZERO operations after the block has reached the top. By placing the pebbles on node $b-1$ it is easy to see that the following invariants will be satisfied (I_3 and I_4 will not be satisfied any longer):

$$\begin{array}{ll} I_5: & P_b \geq b+i, \\ I_6: & P_{b-j} \geq b+i-1 \qquad \text{for } j=1,\ldots,x-i-1. \end{array}$$

So after the next $x-1$ zeroings we see that $P_b \geq b+(x-1)$ which gives the stated result. □

COROLLARY 1. *For all* **D** *strategies we have* $M \geq \lfloor \alpha+\sqrt{2\alpha-7/4}-1/2 \rfloor$ *where* $\alpha = min\{b,d\}$.

10. Conclusion

In the preceding sections we have shown that it is possible to implement partially persistent bounded in-degree (and out-degree) data structures where each access and update step can be done in worst case constant time. This improves the best previously known technique which used amortised constant time per update step.

It is a further consequence of our result that we can support the operation to delete the current version and go back to the previous version in constant time. We just have to store all our modifications of the data structure on a stack so that we can backtrack all our changes of the data structure.

11. Open problems

The following list states open problems concerning the dynamic two player game.

- Is it possible to show a general lower bound for M which shows how M depends on b and d?
- Do better (locally adaptive) strategies exist?
- Do implementable strategies for player **D** exist where $M \in \mathrm{O}(b+d)$?

Acknowledgements

I want to thank Dany Breslauer, Thore Husfeldt and Lars A. Arge for encouraging discussions. Especially I want to thank Peter G. Binderup for the very encouraging discussions that lead to the proof of Lemma 5 and Erik M. Schmidt for comments on the presentation.

This work was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 7141 (project ALCOM II).

References

[1] Dietz, Paul F. and Raman, Rajeev. 1991. Persistence, Amortization and Randomization. In *Proc. 2nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 78–88.

[2] Driscoll, James R., Sarnak, Neil, Sleator, Daniel D., and Tarjan, Robert E. 1989. Making Data Structures Persistent. *Journal of Computer and System Sciences 38*, 86–124.

[3] Raman, Rajeev. 1992. *Eliminating Amortization: On Data Structures with Guaranteed Response Time*. PhD thesis, University of Rochester, New York.

[4] Sarnak, Neil and Tarjan, Robert Endre. 1986. Planar Point Location Using Persistent Search Trees. *Communications of the ACM 29*, 669–679.