A Dynamic Data Structure for Maintaining Disjoint Paths Information in Digraphs

Torsten Tholey

07.08.2003

Abstract

In this paper we present the first dynamic data structure for testing - in constant time - the existence of two edge- or quasi-internally vertex-disjoint paths p_1 from s to t_1 and p_2 from s to t_2 for any three given vertices s, t_1, and t_2 of a digraph. By quasi-internally vertex-disjoint we mean that no inner vertex of p_1 appears on p_2 and vice versa. Moreover, for two vertices s and t, the data structure supports the output of all vertices and all edges whose removal would disconnect s and t in a time linear in the size of the output. The update operations consist of edge insertions and edge deletions, where the implementation of edge deletions will be given only in the full version of this paper. The update time after an edge deletion is competitive with the reconstruction of a static data structure for testing the existence of disjoint paths in constant time, whereas our data structure performs much better in the case of edge insertions.

Copyright

A Dynamic Data Structure for Maintaining Disjoint Paths Information in Digraphs

Torsten Tholey

Institut für Informatik
Johann Wolfgang Goethe-Universität Frankfurt
D-60054 Frankfurt am Main, Germany
tholey@ka.informatik.uni-frankfurt.de

Abstract. In this paper we present the first dynamic data structure for testing - in constant time - the existence of two edge- or quasi-internally vertex-disjoint paths p_1 from s to t_1 and p_2 from s to t_2 for any three given vertices s, t_1, and t_2 of a digraph. By quasi-internally vertex-disjoint we mean that no inner vertex of p_1 appears on p_2 and vice versa. Moreover, for two vertices s and t, the data structure supports the output of all vertices and all edges whose removal would disconnect s and t in a time linear in the size of the output. The update operations consist of edge insertions and edge deletions, where the implementation of edge deletions will be given only in the full version of this paper. The update time after an edge deletion is competitive with the reconstruction of a static data structure for testing the existence of disjoint paths in constant time, whereas our data structure performs much better in the case of edge insertions.

1 Introduction

Suppose that in a network represented by a directed graph $G = (V, E)$ we want to send packets from a node v to another node w. Then, for $n \in \mathbb{N}$, it may be useful to know whether there are n edge- or internally vertex-disjoint paths from v to w. This may increase the capacity between the nodes, so that we can send more packets from v to w at the same time. Another aspect is the reliability of the network: If, for $n \geq 2$, there are n edge- (or internally vertex-) disjoint paths from v to w, we can guarantee that w is reachable from v even after the removal of $n-1$ edges (or vertices). Moreover, we can try to check whether some packets are lost by sending the same packets along different routes. Disjoint paths problems also arise in the context of VLSI design.

More generally, given $2k$ vertices $s_1, s_2, \ldots, s_k$ and $t_1, t_2, \ldots, t_k$, one may be interested in testing whether there are k edge-disjoint or quasi-internally vertex-disjoint paths p_i from s_i to t_i ($1 \leq i \leq k$). By *quasi internally vertex-disjoint* (or for short *q.i.disjoint*) we mean that, for all pairs (i, j) with $i, j \in \{1, \ldots, k\}$ and $i \neq j$, no inner vertex of p_i appears on p_j. If the vertices $s_1, \ldots, s_k, t_1, \ldots, t_k$ are pairwise distinct, the problems above are known as the *(vertex-disjoint) k-paths problem* or the *arc-disjoint k-paths problem*, respectively. Unfortunately, both problems are NP-complete [6], even for $k = 2$. Hence, we focus on a special case where $k = 2$ and additionally $s_1 = s_2$. By symmetry, the special case where $t_1 = t_2$ can be handled in the same way.

Previous Results. Using standard network-flow techniques k edge- or q.i.-disjoint paths from a source node s to not necessarily distinct vertices $t_1, \ldots, t_k$ can be computed in $O(|E| + |V|)$ time if $k = O(1)$, and if such paths exist. For a fixed vertex s, Suurballe and Tarjan [12] have shown that, after a preprocessing time of $O(|E| \log_{1+\lceil |E|/|V| \rceil} |V|)$, one can output a pair of edge-disjoint or q.i.disjoint paths p_1 and p_2 from s to an arbitrary vertex t with minimal total length in time linear in the number of edges of the paths output, if such paths exist. It is implicit that - after the preprocessing - we can test the existence of such paths in $O(1)$ time. Lee and Wu [9] consider the problem of finding k edge-disjoint paths between two vertices s and t that - among all edge-disjoint paths between s and t - have a smallest number of common vertices and that, among all such paths, induce the smallest cost, where, for our purposes, we can assume that the cost is equal to the total number of edges of the paths output. They reduce the problem to a minimum-cost network-flow problem that, if $k = O(1)$, can be solved in $O(|V||E| \log |V|)$ time using the algorithm of Ahuja et al. [1].

Concerning dynamic algorithms, where edges may be inserted or deleted dynamically, there was a great development for undirected graphs, see e.g. [8], [13], and [14]. Much less is known about dynamic algorithms for digraphs. The most important results concerning digraphs are about the maintenance of the transitive closure ([3], [10] and [11]) and of shortest paths ([4], [5], [10] and [11]). These results imply a quadratic or super-quadratic amortized update time, respectively.

New results. In this paper we present the first dynamic algorithm for testing the existence of two edge- or q.i.disjoint paths p_1 from s to t_1 and p_2 from s to t_2, for all possible $\Theta(|V|^3)$ values of (s, t_1, t_2). If no edge- or q.i.disjoint paths between two vertices s and t exist, our data structure can output the set $V_G(s, t)$ of all vertices or the set $E_G(s, t)$ of all edges whose removal would disconnect s and t. More precisely, to avoid ambiguity, $V_G(s, t)$ and $E_G(s, t)$ denote the set of all vertices or edges, respectively, that lie on every path from s to t, i. e. in particular $s, t \in V_G(s, t)$. In more detail, we develop a data structure supporting update operations Initialize, Insert, and Delete, and queries EdgeDis, VertexDis, CEdges, and CVertices defined as follows: Given an $n \in \mathbb{N}$, Initialize(n), initializes the data structure for the graph G with vertex set $V = \{1, \ldots, n\}$ and an empty edge set E. Given two vertices $r, s \in V$ with $r \neq s$, Insert(r, s) adds edge (r, s) to E, Delete(r, s) deletes edge (r, s) from E, CEdges(r, s) returns $E_G(r, s)$, and CVertices(r, s) returns $V_G(r, s)$. Finally, given three vertices s, t_1, t_2, EdgeDis and VertexDis return "yes" if there exist two paths p_1 from s to t_1 and p_2 from s to t_2 that are edge-disjoint or q.i.disjoint, respectively, and "no" otherwise.

Our data structure is *output-linear*. By that we mean that it supports all queries in a time linear in the complexity of the output. For example, the running time of CEdges is linear in the number of edges output. Initialize and Insert require $O(|V|^2)$ time, whereas Delete is supported in $O(\min\{|V|^2|E|, |V|^3\})$ time.

The update time for an edge deletion in our dynamic data structure is competitive with the best known construction time for a static output-linear data structure supporting our four queries (in the full version of this paper we will show that a construction time of $O(\min\{|V|^2|E|, |V|^3\})$ is possible), whereas in the case of insertions the dynamic data structure performs much better.

In the full version of this paper we present an incremental version of our dynamic data structure supporting an additional query MDisPath (see Theorem 11). This operation, given three vertices s, t_1, and t_2, supports the output of two simple paths from s to t_1 and from s to t_2 that are simultaneously *maximally edge-disjoint* and *maximally vertex-disjoint*. By that we mean that the paths, among all pairs of paths leading from s to t_1 and s to t_2, have the smallest number of common vertices or edges, respectively.

2 The Main Idea

As shown later, for two vertices v and w of a digraph $G = (V, E)$, there are two q.i.disjoint paths from v to w if and only if $V_G(v, w) = \{v, w\}$ and $(v, w) \notin E_G(v, w)$. Hence, if we restrict our attention to the special case $t_1 = t_2$, one way to support VertexDis(s, t_1, t_2) in $O(1)$ time is to maintain the sets $V_G(v, w)$ and $E_G(v, w)$ for all pairs of vertices (v, w). But, if we store each such set separately, an edge insertion can add $\Omega(|V|)$ vertices or edges to nearly all such sets so that our update time may exceed the promised performance bound of $O(|V|^2)$.

Fortunately, one can show that all simple directed paths from v to w visit the vertices of $V_G(v, w)$ in the same order and that, for the last vertex $u \in V_G(v, w)$ visited before reaching w, $V_G(v, u) = V_G(v, w) - \{w\}$ holds. For each fixed vertex $v \in V$, we can therefore store the sets $V_G(v, w)$, for all vertices w reachable from v, in one tree T_v such that the nodes on the tree path from v to w are exactly the vertices of $V_G(v, w)$. This reduces the space for representing the sets $V_G(v, w)$ for all pairs (v, w) to $O(|V|^2)$. As we will show later, the time needed for the update of all trees T_v after an edge insertion can be reduced to $O(|V|^2)$ as well.

We will prove that, if we mark all tree arcs (x, y) of T_v with $(x, y) \notin E_G(v, y)$ and x is the father of y, then the set of unmarked tree arcs on the tree path from v to another node w of T_v is exactly $E_G(v, w)$. Moreover, there are two edge-disjoint paths from v to w if and only if all tree arcs on the tree path from v to w are marked. Hence, if we let $a_{T_v}(w)$ be the ancestor of w in T_v with the farthest distance from w that is connected to w by a tree path without any unmarked tree arcs, then $E_G(v, w) = \emptyset$ iff $a_{T_v}(w) = v$. Thus, given the value $a_{T_v}(w)$, we can answer EdgeDis(v, w, w) in constant time. As we will see later, the same is true for all queries of the form VertexDis(v, w_1, w_2) and EdgeDis(v, w_1, w_2).

3 Representing Disjoint Paths Information

We call the tree T_v introduced in the last section the *bottleneck tree* of v or, for short, the *B-tree* of v. In this section we study some basic properties of the

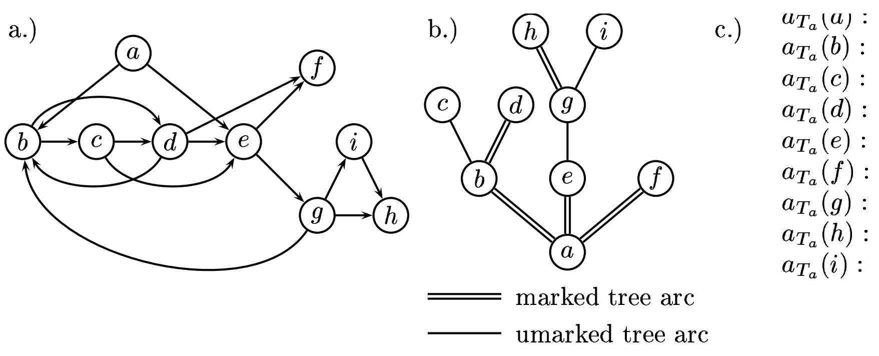

Fig. 1. a) a digraph, b) the B-tree T_a of a, c) the function a_{T_v}.

bottleneck tree, and of some special kind of lists that we will call *bottleneck lists.* But first, let us make some further definitions:

Let $G = (V, E)$ be a digraph. A path p in G is *simple* if no vertex appears on p more than once. We use the notation $p : a \to b$ to denote a path p leading from $a \in V$ to $b \in V$. A vertex $w \in V$ is *doubly-edge-reachable* from a vertex $v \in V$ if there are at least two edge-disjoint paths from v to w, and we denote this fact by writing $v \underset{2}{\Rightarrow} w$. If there are also at least two different q.i.disjoint paths from v to w, we call w *doubly-vertex-reachable* from v and write $v \underset{2}{\circ\!\!\Rightarrow} w$. Let us say that $v \underset{2}{\Rightarrow} v$ and $v \underset{2}{\circ\!\!\Rightarrow} v$ hold by definition. In Sect. 2 we mentioned that, if v $\underset{2}{\circ\!\!\Rightarrow}$ w, i. e. if v and w are the only common vertices of two maximally vertex-disjoint paths from v to w, then we have $V_G(v, w) = \{v, w\}$, and that all simple paths from v to w visit the vertices of $V_G(v, w)$ in the same order. This follows from Lemma 1:

Lemma 1. *The set of common edges of two arbitrary maximally edge-disjoint paths from v to w is exactly $E_G(v, w)$ and, similarly, the set of common vertices of two arbitrary maximally vertex-disjoint paths from v to w is exactly $V_G(v, w)$. Moreover, all simple paths from v to w visit the edges of $E_G(v, w)$, as well as the vertices of $V_G(v, w)$, in the same order.*

We call the list $(v_1, v_2, \ldots)$ of all vertices from $V_G(v, w)$, sorted in the order of their appearance on every simple path from v to w, the *bottleneck list* of (v, w), and we refer to it as $bl(v, w)$ (take $bl(v, v) = (v)$). For the graph in Fig. 1.a we have $bl(a, h) = (a, e, g, h)$. We write $x \in bl(v, w)$ if x is a vertex that appears in the list $bl(v, w)$. By $last(v, w)$ we denote the last vertex in $bl(v, w)$ before w, or w itself if $v = w$. For bottleneck lists the following holds:

Lemma 2. *Let $v, w \in V$ with $v \neq w$ and $(v_1, \ldots, v_k) = bl(v, w)$. Then, for all $i \in \mathbb{N}$ with $0 < i < k$, either $v_i \underset{2}{\circ\!\!\Rightarrow} v_{i+1}$ or $(v_i, v_{i+1}) \in E_G(v, w)$, but not both.*

In the last section we already mentioned that $V_G(v, last(v, w)) \subseteq V_G(v, w)$. We can show something more:

Lemma 3. *Let v and w be two distinct vertices of V, and $(v_1, \dots, v_k) = bl(v, w)$. Then, for every vertex u on a simple path from v to v_i with $i \in \{1, \dots, k\}$,*

- *$(v_i, v_{i+1}, \dots, v_k)$ is a suffix of $bl(u, w)$,*
- *$E_G(u, w) \supseteq \{(r, s) \in E_G(v, w) \mid r = v_j, i \leq j < k\}$.*

Moreover, if $u = v_i$, then $bl(u, w) = (v_i, \dots, v_k)$, and the inclusion above for $E_G(u, w)$ holds with equality. By symmetry, if u is a vertex of a simple path from v_i to w for some $i \in \{1, \dots, k\}$, we have

- *$(v_1, \dots, v_i)$ is a prefix of $bl(v, u)$,*
- *$E_G(v, u) \supseteq \{(r, s) \in E_G(v, w) \mid r = v_j, 1 \leq j < i\}$,*

and, if $u = v_i$, then $bl(v, u) = (v_1, \dots, v_i)$, and the inclusion above for $E_G(v, u)$ holds with equality.

The data structure presented in this paper maintains for each $v \in V$ the *B-tree* T_v of v. Recall that the nodes of T_v are those vertices of G that are reachable from v, its root is v, and the father of an inner node u in T_v, denoted by $f_{T_v}(u)$, is $last(v, u)$. We mark a tree arc $(f_{T_v}(u), u)$ iff $(f_{T_v}(u), u) \notin E_G(v, u)$ which according to Lemma 2 and Lemma 3 is equivalent to $f_{T_v}(u) \underset{2}{\Rightarrow} u$. Moreover, we define a function m_{T_v} that maps $(f_{T_v}(u), u)$ to 1 if $(f_{T_v}(u), u)$ is marked, and to 0 otherwise. The B-tree for vertex a of the graph in Fig. 1.a is given in Fig. 1.b. From the B-tree one can read off the sets $V_G(v, w)$ and $E_G(v, w)$:

Lemma 4. *Let u be an ancestor of a node w in the B-tree T_v of v. Then the ordered list of nodes on the tree path from u to w is exactly $bl(u, w)$, and the set of unmarked arcs on this path is exactly $E_G(u, w)$.*

For determining the set $E_G(v, w)$ more efficiently, we store with each node w of the B-tree T_v a value $a_{T_v}(w)$, which we have already defined as the ancestor of w in T_v with the farthest distance from w that is connected to w by a tree path without any unmarked tree arcs (see Fig. 1.c). Then, given an unmarked tree arc (x, y) with $x = f_{T_v}(y)$ and $a_{T_v}(x) \neq v$, the next unmarked tree arc on the tree path from x to v is $(f_{T_v}(a_{T_v}(x)), a_{T_v}(x))$. We can also conclude that in the case $a_{T_v}(x) \neq v$ vertex x is not doubly edge-reachable from v since $(f_{T_v}(a_{T_v}(x)), a_{T_v}(x)) \in E_G(v, x)$. This means that $v \underset{2}{\Rightarrow} x$ implies $a_{T_v}(x) = v$. The following lemma shows that the reverse is also true:

Lemma 5. *If x is an ancestor of a node y in the B-tree T_v of v such that all tree arcs on the path from x to y are marked, then $x \underset{2}{\Rightarrow} y$. In particular, for all nodes w in tree T_v, we have $a_{T_v}(w) \underset{2}{\Rightarrow} w$.*

Another important property of B-trees is given in the following lemma:

Lemma 6. *Let x, y, and z be three distinct nodes of a B-tree T_v such that x is the lowest common ancestor of y and z in T_v. Then there exist two q.i.disjoint paths, one going from x to y, and the other one going from x to z.*

In the following we will use the notation $lca_T(u, v)$ to denote the lowest common ancestor of two nodes u and v in a tree T.

4 The Dynamic Data Structure

Our dynamic data structure maintains, for each vertex $v \in V$, the B-tree T_v of v, (the values of) the functions a_{T_v} and m_{T_v}, and a data structure L_{T_v}, where for a tree T the data structure L_T allows us to compute $lca_T(u, w)$ for all nodes u, w of T in constant time. Such a data structure is given by Harel and Tarjan [7].

For every vertex $v \in V$, Initialize defines T_v to be the tree consisting of only one node, namely v, and lets $a_{T_v}(w) = v$ if $w = v$, and $a_{T_v}(w) = f_{T_v}(w) = nil$ if $w \neq v$. VertexDis(v, w, w) returns "yes" if $f_{T_v}(w) = v$ and the tree arc (v, w) is marked, and "no" in all other cases. If $w_1 \neq w_2$ VertexDis(v, w_1, w_2) outputs "yes" if and only if $lca_{T_v}(w_1, w_2) = v$ (Lemma 4 and 6). EdgeDis(v, w, w) returns "yes" iff w is a node of T_v and $a_{T_v}(w) = v$. As shown in Sect. 3, this is correct. EdgeDis(v, w_1, w_2) outputs "yes" if and only if $a_{T_v}(lca_{T_v}(w_1, w_2)) = v$. See the full version of this paper for the correctness of this step.

CVertices(v, w) returns the set of all nodes on the tree path from v to w in T_v, whereas CEdges(v, w) has to return the set of unmarked tree arcs on this tree path (Lemma 4). It is not hard to see that the implementations above support VertexDis and EdgeDis in constant time, Initialize in $O(|V|^2)$ time, and CVertices and CEdges in a time linear in the number of vertices or edges output, respectively (for supporting CEdges we make use of function a_{T_v}).

We still have to implement Insert, which will be more complicated. Suppose that an edge (r, s) is inserted in G, and we want to update T_v, m_{T_v}, a_{T_v}, and L_{T_v} for an arbitrary, but fixed, vertex $v \in V$ with $bl(v, r) = (v_1, \ldots, v_k)$. Before describing the update in more detail, we will make some further definitions:

For all $x \in V$, we let $R(x)$ be the set of vertices that are reachable from x before the insertion of (r, s). The vertices of $bl(v, r)$ and their descendants in T_s will be handled in a special way[1]. Thus, for brevity, we denote the set of these vertices by S. Finally, we call a vertex $w \in V$ *broken* if $w \in (R(s) \cap R(v)) - S$ and either $w = s$ or, before the insertion of (r, s), the father of w in T_s does not coincide with the father of w in T_v. The broken vertices will be the only vertices for which we possibly have to "break" a given tree edge in the old B-tree T_v before the update and to connect them to a new father after the update.

Now Insert(r, s) updates T_v, m_{T_v}, a_{T_v}, and L_{T_v} as follows:

If r is not reachable from v, i. e. $r \neq v$ and $f_{T_v}(r) = nil$, Insert(r, s) leaves T_v, m_{T_v}, and a_{T_v} unchanged. Otherwise, Insert(r, s) determines the index $ind(w)$ with $v_{ind(w)} := lca_{T_v}(r, w)$ for all $w \in R(v)$. Then a depth-first search in T_s is started at node s, and for each visited node w, the following is done:

If $w = v_{ind(w)}$ we skip w and its descendants in T_s and continue the depth-first search with the next node of T_s that does not lie in the subtree rooted at w. Otherwise, we compute a vertex u and a value $mark \in \{0, 1\}$ such that

- $u = r$ and $mark = 0$, if $w = s$ and $w \notin R(v)$,
- $u = v_{ind(w)}$ and $mark = 1$, if $w \in R(s) \cap R(v)$ and w is broken,
- $u = f_{T_s}(w)$ and $mark = m_{T_s}(u, w)$ in the remaining cases,

[1] For the following, keep in mind that T_s and - for all $w \in V$ - $bl(s, w)$ cannot be changed by the insertion of an edge (r, s).

and then we set $f_{T_v}(w) = u$, and $m_{T_v}(f_{T_v}(w), w) = mark$. Moreover, we define $a_{T_v}(w) = a_{T_v}(u)$, if $mark = 1$, and $a_{T_v}(w) = w$, if $mark = 0$. After this update of T_v, m_{T_v}, and a_{T_v} we recompute L_{T_v}. Using the data structure of Harel and Tarjan [7] or the simpler implementation of Bender and Farach-Colton [2] this takes $O(|V|)$ time.

With the above implementation the update of T_v, m_{T_v}, a_{T_v}, and L_{T_v} takes $O(|V|)$ time, so that Insert(r, s) runs in $O(|V|^2)$ time. We still have to show that, for each vertex $v \in V$, the functions f_{T_v}, m_{T_v}, and a_{T_v} are correctly updated:

The implementation of Insert does not change any of the values $f_{T_v}(w)$, $m_{T_v}(f_{T_v}(w), w)$ and $a_{T_v}(w)$, if $r \notin R(v), w \notin R(s)$, or $w \in S$. In the full version of this paper we show that this is correct. Thus, in the following, let us assume that $r \in R(v), w \in R(s) - S$, and that we have already shown that, for all ancestors u of w in T_s, the values $f_{T_v}(u), m_{T_v}(f_{T_v}(u), u)$, and $a_{T_v}(u)$ were updated correctly.

If $w \in R(s) - (R(v) \cup \{s\})$, it follows from $f_{T_s}(w) \in bl(s, w)$ that, after the insertion of (r, s), every path from v to w visits vertex $f_{T_s}(w)$. Thus, we have $f_{T_s}(w) \in V_G(v, w)$ for the updated graph, and $f_{T_s}(w)$ must be an ancestor of w in the updated B-tree T_v. Moreover, before and hence after the update we have $bl(f_{T_s}(w), w) = (f_{T_s}(w), w)$, and it is easy to see that there are two q.i.disjoint paths from $f_{T_s}(w)$ to w in the updated graph iff this also holds before the update. Hence, the new values of $f_{T_v}(w)$ and $m_{T_v}(f_{T_v}(w), w)$ are correctly defined to be equal to the (old) values of $f_{T_s}(w)$ and $m_{T_s}(f_{T_s}(w), w)$, respectively. Finally, since $a_{T_v}(f_{T_s}(w))$ was already updated correctly, Insert(r, s) correctly lets $a_{T_v}(w)$ be equal to w if $m_{T_v}(f_{T_v}(w), w) = 0$, and equal to $a_{T_v}(f_{T_s}(w))$ if $m_{T_v}(f_{T_v}(w), w) = 1$. Similar arguments show that if for a vertex $w \in R(s) \cap R(v)$ the parents of w in T_s and T_v exist and coincide, the values $f_{T_v}(w)$, $m_{T_v}(f_{T_v}(w), w)$, and $a_{T_v}(w)$ are also correctly updated in the same way.

If $w = s$ and $s \notin R(v)$ it is easy to see that $f_{T_v}(w) = r$, $m_{T_v}(f_{T_v}(w), w) = 0$, and $a_{T_v}(w) = w$ must hold after the update.

Finally, let w be broken, i. e. $w \in (R(s) \cap R(v)) - S$ and either $w = s$ or the father of w in T_s does not coincide with the father of w in T_v. It is clear that every path from v to w in the updated graph not using edge (r, s) has to visit $lca_{T_v}(w, r)$ - where $lca_{T_v}(w, r)$ should denote the corresponding value before the update - since this was true before the update. But every new path using edge (r, s) has also to visit node $lca_{T_v}(w, r)$, since $lca_{T_v}(w, r) \in bl(v, r)$. Hence, $lca_{T_v}(w, r) \in V_G(v, w)$, and $lca_{T_v}(w, r)$ must be an ancestor of w in the updated B-tree T_v. As we will show, there exist two q.i.disjoint paths from $lca_{T_v}(w, r)$ to w in the updated graph, and thus Insert(r, s) correctly defines $f_{T_v}(w) = lca_{T_v}(w, r)$, $m_{T_v}(f_{T_v}(w), w) = 1$, and $a_{T_v}(w) = a_{T_v}(lca_{T_v}(w, r))$.[2]

It remains to show that there exist indeed two q.i.disjoint paths from $v_{ind(w)}$ $(= lca_{T_v}(w, r))$ to w after the update. Let us consider two different cases:

Case 1. $w \in (R(s) \cap R(v)) - S$, and either $w = s$ or $f_{T_s}(w) \notin R(v)$. In this case, before and hence after the update, there exist two q.i.disjoint paths $p_1 : v_{ind(w)} \to w$ and $p_2 : v_{ind(w)} \to r$ (Lemma 6).

[2] Since $lca_{T_v}(w, r) \in bl(v, r)$ the value $a_{T_v}(lca_{T_v}(w, r))$ does not change, so that w.l.o.g. we can assume that $a_{T_v}(lca_{T_v}(w, r))$ is already updated correctly.

If $w \neq s$ we replace p_2 by a new path that follows the old path p_2 between $v_{ind(w)}$ and r, then follows edge (r, s), and finally follows an arbitrary simple path from s to $f_{T_s}(w)$. This new path p_2 is also q.i.disjoint to p_1, since $f_{T_s}(w)$ and therefore all vertices x on the new path between s and $f_{T_s}(w)$ do not belong to $R(v)$.[3] We know that $bl(f_{T_s}(w), w) = (f_{T_s}(w), w)$ before and hence also after the update. According to the following lemma $v_{ind(w)} \underset{2}{\circ\!\!\Rightarrow} w$ holds.

Lemma 7. *Let $q_1 : a \to b$ and $q_2 : a \to c$ be q.i.disjoint paths in a digraph. Then, if $bl(b, c) = (b, c)$, there exist two q.i.disjoint simple paths from a to c.*

Similar considerations show that $v_{ind(w)} \underset{2}{\circ\!\!\Rightarrow} w$ also holds if $w = s$.

Case 2. $w, f_{T_s}(w) \in (R(s) \cap R(v)) - S$ and $f_{T_s}(w) \neq f_{T_v}(w)$ before the insertion of (r, s). Let $t = f_{T_s}(w)$ and z be the first broken node on the tree path from t to s in T_s in this direction. Such a node exists: If there is a node $a \in R(v)$ on the tree path from t to s with $f_{T_s}(a) \notin R(v)$, the first such node is broken, otherwise s is broken. We consider the situation before the insertion of edge (r, s): It follows from the definition of z that node t is a descendant of z in T_v (and T_s) - possibly equal to z - but w cannot be a descendant of z in T_v. There exist two simple q.i.disjoint paths $p_1 : lca_{T_v}(t, w) \to w$ and $q_1 : lca_{T_v}(t, w) \to t$ (Lemma 6). Let r_1 be a third simple path from v to $lca_{T_v}(t, w)$. r_1 is q.i.disjoint to p_1 and q_1. Otherwise, there would be a common vertex x of r_1, and say p_1, with $x \neq lca_{T_v}(t, w)$, and hence a path from v to w not using node $lca_{T_v}(t, w)$, a contradiction to $lca_{T_v}(t, w) \in bl(v, w)$. Figure 2.a shows the different cases that can occur if $ind(w) > ind(t)$, $ind(t) > ind(w)$, or $ind(w) = ind(t)$.

The sub-paths of p_1, q_1 or r_1 between any two consecutive vertices of Fig. 2.a are pairwise q.i.disjoint in all five cases. Thus, we can construct new q.i.disjoint simple paths as indicated in Fig. 2.b. Since z is broken and z is an ancestor of w in T_s we conclude that $f_{T_v}(z) = v_j$ and $v_j \underset{2}{\circ\!\!\Rightarrow} z$ hold for some $j \in \{1, \ldots, k\}$ after the update. $j \leq ind(z)$, since $v_i \notin V_G(v, z)$ for all $i > ind(z)$ before and hence after the update. Moreover, $j \geq ind(w)$. Otherwise, since $w \notin S$, there is a path from z to w that does not visit $v_{ind(w)}$ and hence a path via z from v to w that does not visit $v_{ind(w)}$. But we already know that $v_{ind(w)} \in V_G(v, w)$ after the insertion of (r, s). Altogether, there are two q.i.disjoint paths t_1 and t_2 from some v_j with $ind(w) \leq j \leq ind(z)$ to z. We distinguish between different cases where $ind(w) > j$, or $ind(w) = j$ as shown in Fig. 2.c.

In Case α of Fig. 2.c v_j is a descendant of $v_{ind(w)}$ in T_v. Then, since $j \leq ind(z) = ind(t)$, we are in Case A of Fig. 2.b, and we choose s_1 to be the path following q_3 from $v_{ind(w)}$ to v_j. It follows that p_2, q_2, and s_1 are pairwise q.i.disjoint. In both cases of Fig. 2.c we can apply the following lemma to construct two q.i.disjoint paths $p_w : v_{ind(w)} \to w$ and $p_t : v_{ind(w)} \to t$.

Lemma 8. *Let $a, b, c, d,$ and e be vertices of a digraph G with $b \notin \{c, d\}$. Then, if there exist five paths $p_1 : a \to b$, $p_2 : a \to c$, $p_3 : d \to e$, p_4 and p_5 leading from c to d in G such that p_1 is q.i.disjoint to both p_2 and p_3, and p_4 is q.i.disjoint to p_5 (see Fig. 3), there exist two q.i.disjoint simple paths $p_6 : a \to b$ and $p_7 : a \to e$.*

[3] Otherwise, $x \in R(v)$ and $f_{T_s}(w) \in R(x)$ would lead to $f_{T_s}(w) \in R(v)$.

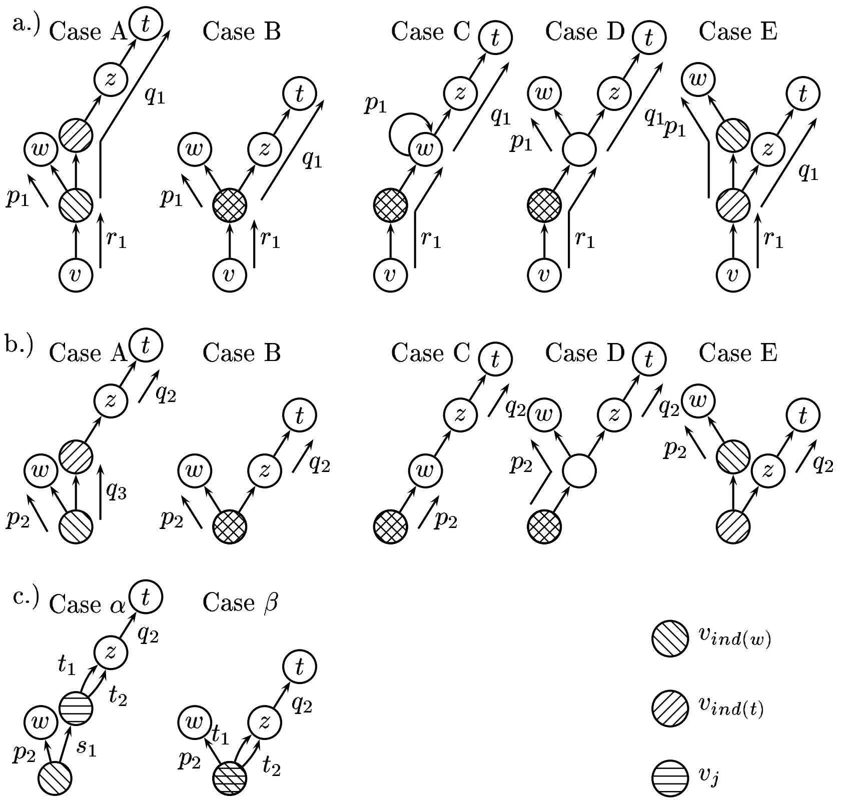

Fig. 2. Construction of disjoint paths in Case 2.

From $t = f_{T_s}(w)$ it follows that $bl(t, w) = (t, w)$. According to Lemma 7 we can construct two q.i.disjoint paths from $v_{ind(w)}$ to w. This finishes our proof for Case 2. We just have shown:

Theorem 9. *There exists an output-linear data structure using $O(|V|^2)$ space and supporting the operations Initialize, Insert, VertexDis, EdgeDis, CEdges, and CVertices such that Initalize and Insert have a running time of $O(|V|^2)$.*

In the full version of this paper we will also show:

Theorem 10. *The data structure of Theorem 9 can be extended such that it supports Delete in $O(\min\{|V|^2|E|, |V|^3\})$ time.*

Theorem 11. *There exists an output-linear data structure using $O(|V|^3)$ space and supporting the operations Initialize, Insert, VertexDis, EdgeDis, CVertices, CEdges, and MDisPath such that Initialize has a running time of $O(|V|^2)$, and*

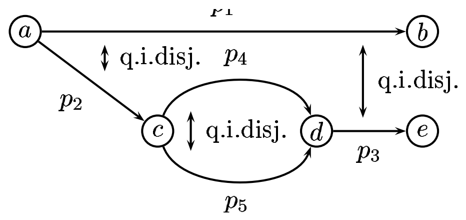

Fig. 3. The paths of Lemma 8.

Insert runs in $O(|V|^3)$ worst case time and $O(\min\{|V|^4/m + |V|^2, |V|^3\})$ amortized time, where m denotes the total number of edge insertions.

References

1. R. K. Ahuja, A. V. Goldberg, J. B. Orlin, and R. E. Tarjan, Finding minimum-cost flows by double scaling. *Math. Programming, Series A* **53** (1992), pp. 243–266.
2. M. A. Bender, and M. Farach-Colton, The LCA problem revisited. Proc. 4th Latin American Theoretical Informatics Symposium (LATIN 2000), Lecture Notes in Computer Science, Vol. 1776, Springer, Berlin, 2000, pp. 88–94.
3. C. Demetrescu, and G. F. Italiano, Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier. Proc. 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS 2000), pp. 381-389.
4. C. Demetrescu, and G. F. Italiano, Fully dynamic all pairs shortest paths with real edge weights. Proc. 42nd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2001), pp. 260–267.
5. C. Demetrescu, and G. F. Italiano, Improved bounds and new trade-offs for dynamic all pairs shortest paths. Proc. 29th International Colloquium on Automata, Languages, and Programming (ICALP 2002), Lecture Notes in Computer Science, Vol. 2380, Springer, Berlin, 2002, pp. 633–643.
6. S. Fortune, J. E. Hopcroft, and J. Wyllie, The directed subgraph homeomorphism problem. *Theoretical Computer Science* **10** (1980), pp. 111–121.
7. D. Harel, and R. E. Tarjan, Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.* **13** (1984), pp. 338–355.
8. J. Holm, K. de Lichtenberg, and M. Thorup, Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM* **48** (2001), pp. 723–760.
9. S.-W. Lee, and C.-S. Wu, A K-best paths algorithm for highly reliable communication networks. *IEICE Trans. Commun.* **E82-B** (1999), pp. 586–590.
10. V. King, Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. Proc. 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1999), pp. 81-89.
11. V. King, and M. Thorup, A space saving trick for directed dynamic transitive closure and shortest path algorithms. 7th Annual International Computing and Combinatorics Conference (COCOON 2001), Lecture Notes in Computer Science, Vol. 2108, Springer, Berlin, 2001, pp. 268–277.

12. J. W. Suurballe, and R. E. Tarjan, A quick method for finding shortest pairs of disjoint paths. *Networks* **14** (1984), pp. 325-336.
13. M. Thorup, Near-optimal fully-dynamic graph connectivity. Proc. 32nd Annual ACM Symposium on Theory of Computing (STOC 2000), pp. 343–350.
14. M. Thorup, Fully-dynamic min-cut. Proc. 33rd Annual ACM Symposium on Theory of Computing (STOC 2001), pp. 224–230.