Evaluating reachability queries over path collections

Panagiotis Bouros[1], Spiros Skiadopoulos[2], Theodore Dalamagas[3], Dimitris Sacharidis[1], and Timos Sellis[1,3]

[1] National Technical University of Athens
9 Iroon Polytechniou, Athens 157 80, Greece
{pbour,dsachar}@dblab.ntua.gr
[2] University of Peloponnese
Karaiskaki Street, Tripoli 221 00, Greece
spiros@uop.gr
[3] Institute for the Management of Information Systems — "Athena" R.C.
17 G. Mpakou, Athens 115 24, Greece
{dalamag,timos}@imis.athena-innovation.gr

Abstract. Several applications in areas such as biochemistry, GIS, involve storing and querying large volumes of sequential data stored as *path collections*. There is a number of interesting queries that can be posed on such data. This work focuses on reachability queries: given a path collection and two nodes v_s, v_t, determine whether a path from v_s to v_t exists and identify it. To answer these queries, the *path-first search* paradigm, which treats paths as first-class citizens, is proposed. To improve the performance of our techniques, two indexing structures that capture the reachability information of paths are introduced. Further, methods for updating a path collection and its indices are discussed. Finally, an extensive experimental evaluation verifies the advantages of our approach.

1 Introduction

Several applications in various areas involve storing and querying large volumes of sequential data. For instance, the metabolic networks in biochemistry applications deal with large collections of pathways, i.e., series of chemical reactions occurring within a cell [1]. Another example comes from Geographic Information Systems (GIS) and geodata services, where the recent advances in infrastructure, and the proliferation of earth observation applications (e.g., GPS technology), have resulted in the abundance geodata. Path collections are typical in web sites such as ShareMyRoutes.com or TravelByGPS.com, which archive popular touristic routes, i.e., sequences of waypoints or points of interest (POIs), uploaded by users.

These datasets share a common structure in that they involve items that connect with each other to form sequences. In the sequel, we refer to items as nodes and to sequences as paths. There is a number of interesting queries that can be

posed on path collections. The focus of this work is on reachability queries: given a path collection and two nodes v_s, v_t, determine whether a path from v_s to v_t exists and identify it. Note that this path need not be present in the collection, but constructed by combining parts of stored paths. We present two distinct application scenarios. Consider a collection of metabolic pathways. In this context, reachability queries answer whether there exists a cause-effect relationship between two chemical reactions in some pathway, and their intermediates (i.e., the reactants). Furthermore, consider an archive of popular touristic routes. Reachability queries answer whether there exists a meaningful/recommended route between two touristic attractions.

A path collection can be trivially mapped to a graph, where its nodes are those contained in the paths. Hence, reachability queries can be evaluated by standard techniques that fall in three categories: (i) search algorithms, e.g., depth-first search [2], (ii) methods based on the pre-computation of the graph's transitive closure (TC), or (iii) approaches that pre-process the graph to construct a reachability encoding scheme. These techniques share their strengths and weaknesses. Exploiting a search algorithm has minimum space requirements but in the worst case we need to examine all the edges of the graph to answer a query. Considering the TC of the graph uncompressed is very efficient as far as querying is concerned, but the complexity of the construction time and the space requirements make this solution infeasible in practice. Works like 2-hop labels [3] that compress the TC of the graph, or labelling schemes [4–6] have been proposed to encode the reachability information of the graph. These schemes determine whether there exists a path between two nodes and some of the schemes can also identify that path.

It is important to note that techniques of the last two categories require pre-computation and are only efficient for datasets that do not frequently change or when the updates affect a small part of the graph. In our setting, however, there are frequent path insertions in the collection dramatically modifying the associated graph and rendering the pre-processed data useless. Based on this observation, we introduce the *path-first search* (pfs) paradigm for evaluating reachability queries on path collections. Briefly the main idea is to examine entire paths at once rather than single nodes. We present an index structure, termed $\mathcal{P}$-$Index$, that provides efficient access to the paths and devise the pfsP algorithm, which utilizes it. Then, we present $\mathcal{H}$-$graph$, a novel graph-representation of a path collection that captures information about shared nodes among paths, construct an appropriate index, $\mathcal{H}$-$Index$, and introduce the pfsH algorithm. Furthermore, we present methods for updating the index structures when new paths arrive. Finally we present an extensive experimental study verifying the advantages offered by our methods.

Outline. Section 2 reviews the literature on evaluating reachability queries on graphs. Section 3 formally defines the problem of evaluating reachability over path collections. Section 4 introduces the pfsP algorithm, and $\mathcal{P}$-$Index$. Section 5 introduces the graph-based model of $\mathcal{H}$-$graph$, defines $\mathcal{H}$-$Index$ and presents the

pfsH algorithm. Section 6 discusses index maintenance. Section 7 presents the experimental study and Section 8 concludes the paper.

2 Related work

The simplest way to evaluate reachability queries is to traverse the graph at query time exploiting a search algorithm, e.g., depth-first or breadth-first search [2]. This approach has minimum space requirements — we only have to store the adjacency lists of the graph. On the other hand, to answer a query, we need to search all the edges of the graph in the worst case. In our work, we propose a search method, in the spirit of depth-first search, that operates on the paths of a collection instead of the edges of a graph.

Another option is to pre-compute and store the transitive closure (TC) of the graph. Then, we can explore the scheme proposed in [7] to encode the reachability information in the TC. The encoding scheme assigns to each node v a set of triples $\langle destination, via, label \rangle$, where "via" denotes the first hop in the path from v to the destination node. Thus, for each node that is accessible from v they create the corresponding triple. At query time, we can determine the existence of a path between two nodes by a single lookup on the encoding scheme and identify it performing a number of lookups. The problem with this approach lies in computing the TC of the graph. Efficient algorithms for computing the TC in relational databases have been proposed [8, 9]. Even so, the computation time of $O(|V|^3)$ and the space requirements of $O(|V|^2)$ prevent us from applying this solution especially for large graphs. On the other hand, in our work we do not pre-process the path collections to compute all the possible transitions between the nodes. We exploit a graph-representation of the path collections to capture the possible transitions between the paths according to their common nodes.

To reduce the storage cost of the TC, Cohen et al. [3] propose 2-hop labels. They identify a subset of the nodes that best capture the reachability information of graph. Thus, for each node v, they construct a list with part of the nodes that can reach v ($L_{in}[v]$) and another one with part of the nodes reachable from v ($L_{out}[v]$). This scheme requires $O(|V| \cdot \sqrt{|E|})$ space and can determine the existence of a path between two nodes v_s, v_t by checking whether $L_{out}[v_s]$ and $L_{in}[v_t]$ have a common node, the so called center v_{center}. To identify the path from v_s to v_t, we need to repeat the procedure for the paths from v_s to v_{center} and from v_{center} to v_t. The problem with this approach lies in the construction cost. Computing the optimal 2-hop cover is NP-hard and requires the computation of the TC. Therefore after the introduction of 2-hop labels, a number of works proposed methods to avoid the computation of TC and to reduce the construction time, e.g., [10, 11] and [12, 13]. In our work, for each node v, we exploit the common nodes of the paths containing v with the other paths of the collection to capture the connectivity information of v.

In the context of labelling schemes for graphs, [4] proposes an interval labelling scheme. The first step is to compute the spanning tree of the graph, the so-called tree cover. Then, considering both the tree cover and the remaining

edges, they assign to each node v a sequence of intervals $L[v]$. At query time, to determine whether there exists a path between two nodes their sequences of labels are compared. In [6], Wang et al. introduce Dual-Labeling for sparse graphs. They also compute the spanning tree of the graph but then they compute the transitive closure of the edges outside the spanning tree. Each node is assigned two labels: one according to the spanning tree edges and another label for the rest of the edges. In [14], Trißl et al. introduce GRIPP scheme for large graphs. Their work aims at minimizing the time needed to construct the labelling scheme and the space required to store it. They also assign to each node an interval label $[pre, post]$ but unlike the previous works, they do not compute the spanning tree of the graph. Then, to check whether a node $v[a_v, b_v]$ is reachable from another $u[a_u, b_u]$ they consider whether $a_v < a_u < b_v$ holds. If not, they repeat the check for the descendants of v. Finally, the idea in [5] is instead of constructing the tree cover or the spanning tree of the graph, to partition the graph into a set of paths P and then create the so called path-tree cover $G[T]$. The path-tree cover is a graph formed by the paths of P (as nodes) and the edges of the initial graph that are not included in any path. In our work we do not assign labels to the nodes of the collection for encoding their connectivity information. We index the paths that contain each node. The graph-based representation of a collection we present, called *$\mathcal{H}$-graph*, resembles the one proposed in [5]. However, in [5] each node is contained in exactly one path, whereas in our work a node can be included in several paths of the collection. In addition, the edges of *$\mathcal{H}$-graph* are formed by the common nodes between the paths.

3 Problem definition

In this section, we formally define the problem of evaluating reachability queries over a path collection. We introduce the basic aspects of the problem and our notation for the rest of the paper. We begin by defining the notion of a path collection over a set of nodes.

Definition 1 (Path). *Let V be a set of nodes. A* path *$p(v_1, \ldots, v_k)$ over V is a sequence of distinct nodes $(v_1, \ldots, v_k) \in V$. By $nodes(p)$ we denote the set of nodes in p. The length of a path p, denoted by l_p, is the number of contained nodes, i.e., $l_p = |nodes(p)|$.*

Definition 2 (Path collection). *Let V be a set of nodes. A path collection over V, denoted by $\mathbf{P}$, is a set of paths $\{p_1, \ldots, p_m\}$ over V. By $nodes(\mathbf{P})$ we denote the set of nodes in $\mathbf{P}$.*

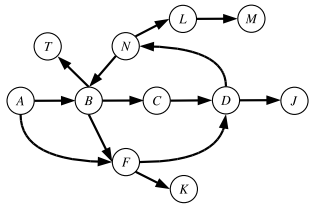
Example 1. Figure 1(a) illustrates an example of a path collection $\mathbf{P} = \{p_1, p_2, p_3, p_4, p_5\}$ over $V = \{A, ..., Z\}$.

Next, we define the family of reachability queries over a path collection.

Definition 3 (Reachability queries). *Let $\mathbf{P}$ be a path collection, and v_s, v_t be two nodes in $nodes(\mathbf{P})$. The family of reachability queries deals with the following problems:*

p_1 (A, B, C, D, J)
p_2 (A, F, D, N, B, T)
p_3 (N, L, M)
p_4 (D, N, B, F, K)
p_5 (A, F, K)

(a)



(b)

Fig. 1. (a) A path collection $\mathbf{P}$, (b) the underlying graph $G_{\mathbf{P}}$ of $\mathbf{P}$.

- *Determine whether there exists a path from v_s to v_t. This query is denoted by* $\texttt{reach}(v_s, v_t)$.
- *Identify a path from v_s to v_t. This query is denoted by* $\texttt{path}(v_s, v_t)$.

In this paper, we deal with the problem of evaluating reachability queries over path collections. To evaluate reachability queries we exploit only the transitions between the nodes contained in the paths of the collection. The collections can be frequently updated with new paths that may involve a number of new nodes. Therefore, we also have to efficiently deal with massive incremental updates.

Given a path collection $\mathbf{P}$, one can construct a graph that contains all the reachability information present in $\mathbf{P}$.

Definition 4 (Underlying graph). *Let $\mathbf{P}$ be a path collection. The underlying graph of $\mathbf{P}$, denoted by $G_{\mathbf{P}}(V, E)$ or simply $G_{\mathbf{P}}$, is a directed graph that contains all the nodes and all the direct transitions of $\mathbf{P}$. Formally:*

- $V = nodes(\mathbf{P})$ *and*
- $E = \{\ (u, v) : (\ldots, u, v, \ldots) \in \mathbf{P}\}$.

It is easy to verify that a path collection $\mathbf{P}$ over set V and the underlying graph $G_{\mathbf{P}}(V, E)$ are equivalent with respect to reachability queries. For example, one can answer $\texttt{path}(F, C)$ over the path collection in Figure 1(a) exploiting $G_{\mathbf{P}}$ graph in Figure 1(b). Therefore, a simple solution to the problem is a search algorithm that exploits the adjacency lists of the graph, with the additional benefits that it imposes minimum construction cost and deals easily with massive updates. In the rest of this work, we consider paths to be first class citizens and propose alternative search-based methods for the task at hand.

4 Evaluating reachability queries over path collections

Section 4.1 introduces the *path-first search* algorithm, termed pfs, for evaluating reachability queries over path collections and Section 4.2 discusses optimizations based on the $\mathcal{P}$-$Index$ structure.

4.1 The path-first search algorithm

The basic idea of the pfs, illustrated in Figure 2, is to examine entire paths at once rather than single nodes. The algorithm takes the collection $\mathbf{P}$, the source v_s and target node v_t as inputs and returns a path connecting them, if one exists, or null, otherwise. The algorithm employs the following data structures: (i) the search stack $\mathcal{Q}$, (ii) the history set $\mathcal{H}$, which contains all nodes that have been pushed in $\mathcal{Q}$, and (iii) the ancestor set $\mathcal{A}$, which stores the direct ancestor of each node in $\mathcal{Q}$. $\mathcal{H}$ is used to avoid cycles and $\mathcal{A}$ to extract answer paths. Note that pfs visits each node once and, thus, there is a single entry per node in $\mathcal{A}$.

The pfs algorithm proceeds similarly to depth-first search as follows. Initially, the stack $\mathcal{Q}$ and $\mathcal{H}$ contain the source node v_s (Lines 1–2). Further, the entry $\langle v_s, \varnothing \rangle$ is inserted in $\mathcal{A}$ (Line 3) denoting that v_s is the source node. The algorithm proceeds examining the contents of the stack (Lines 4–16). The current top node v_n is popped from $\mathcal{Q}$ (Line 5) and checked against target v_t. If they are equal the search terminates and the path is extracted by the ConstructPath method (Line 6). Specifically, starting from v_t, ConstructPath uses the ancestor information of $\mathcal{A}$ to backtrack to the source v_s.

Algorithm pfs
Input: nodes v_s and v_t of a path collection $\mathbf{P}$
Output: a path from v_s to v_t
Parameters:
stack $\mathcal{Q}$: // the search stack
set $\mathcal{H}$: // contains all nodes pushed in $\mathcal{Q}$
set $\mathcal{A}$: // contains the direct ancestor of each node in $\mathcal{H}$

Method:

1. push($v_s, \mathcal{Q}$);
2. **insert** v_s in $\mathcal{H}$
3. **insert** $\langle v_s, \varnothing \rangle$ in $\mathcal{A}$;
4. **while** $\mathcal{Q}$ is not empty **do**
5. **let** v_n = pop($\mathcal{Q}$);
6. **if** v_n is equal to v_t **then return** ConstructPath $(v_s, v_n, \mathcal{A})$;
7. **for** each path $p \in \mathbf{P}$ containing v_n **do**
8. **let** v_p be the node after v_n in p;
9. **while** $v_p \notin \mathcal{H}$ **do** // access each node v_p after v_n in p until the first v_p node // contained in $\mathcal{H}$
10. push($v_p, \mathcal{Q}$);
11. **insert** v_p in $\mathcal{H}$;
12. **insert** $\langle v_p, v_p^- \rangle$ in $\mathcal{A}$, where v_p^- is the direct ancestor of v_p in p;
13. **let** v_p be the next node in p;
14. **end while**
15. **end for**
16. **end while**
17. **return null**;

Fig. 2. Algorithm pfs.

If the target is not found, pfs considers all paths that contain v_n and examines their contents (Lines 7–15). Fix such a path p and let v_p denote the node that follows current top node v_n in p (Line 8). Next, a while loop begins checking if

v_p has never been pushed in $\mathcal{Q}$ (i.e., $v_p \notin \mathcal{H}$). If the check succeeds, v_p is pushed in $\mathcal{Q}$ and inserted in $\mathcal{H}$ (Lines 10–11). In addition, the entry $\langle v_p, v_p^- \rangle$, where v_p^- is the direct ancestor of v_p in path p, is inserted in $\mathcal{A}$ (Line 12). Last, v_p is updated to the next node in p (Line 13) and the while loop continues checking the new v_p. The condition on Line 9 ensures that only nodes that have not been previously enqueued are inserted in $\mathcal{Q}$; hence, pfs avoids cycles.

Example 2. We illustrate the pfs algorithm for the query $\texttt{path}(F, C)$ on the path collection of Figure 1(a). Initially, we have (Lines 1–3):

$$\mathcal{Q} = \{F\},\ \mathcal{H} = \{F\} \text{ and } \mathcal{A} = \{\langle F, \varnothing \rangle\}.$$

At the first iteration of the outer while loop, the algorithm pops F from $\mathcal{Q}$ and identifies paths p_2, p_4 and p_5 that contain F.

- When processing $p_2(A, F, D, N, B, T)$, the algorithm adds to $\mathcal{Q}$ and $\mathcal{H}$, nodes D, N, B and T, and to $\mathcal{A}$ pairs $\langle D, F \rangle$, $\langle N, D \rangle$, $\langle B, N \rangle$ and $\langle T, B \rangle$.
- When processing $p_4(D, N, B, F, K)$, the algorithm adds to $\mathcal{Q}$ and $\mathcal{H}$, node K, and to $\mathcal{A}$ pair $\langle K, F \rangle$.
- When processing $p_5(A, F, K)$, the algorithm does not add anything to $\mathcal{Q}$, $\mathcal{H}$ and $\mathcal{A}$ since there are no new nodes after the current node F (K has been enqueued).

After the first iteration, we have:

$$\mathcal{Q} = \{D, N, B, T, K\},$$
$$\mathcal{H} = \{F, D, N, B, T, K\} \text{ and}$$
$$\mathcal{A} = \{\langle F, \varnothing \rangle, \langle D, F \rangle, \langle N, D \rangle, \langle B, N \rangle, \langle T, B \rangle, \langle K, F \rangle\}.$$

The algorithm proceeds in a similar manner and after the fourth iteration, we have:

$$\mathcal{Q} = \{D, N, C\},$$
$$\mathcal{H} = \{F, D, N, B, T, K, C\} \text{ and}$$
$$\mathcal{A} = \{\langle F, \varnothing \rangle, \langle D, F \rangle, \langle N, D \rangle, \langle B, N \rangle, \langle T, B \rangle, \langle K, F \rangle\, \langle C, B \rangle\}.$$

At the fifth iteration, pfs pops node C (the target) from stack $\mathcal{Q}$ and terminates the search. ConstructPath returns the answer path (F, D, N, B, C) by scanning $\mathcal{A}$.

Algorithm pfs terminates the search when the target node is popped out of stack $\mathcal{Q}$. An alternative approach is to check whether both current search node and the target node are contained in a path of the collection and terminate search without visiting any other node. In the next section, we discuss this improvement and present an extention to pfs called pfsP.

4.2 $\mathcal{P}$-*Index*: indexing path collections

In this section, we describe the *path collection index* $\mathcal{P}$-*Index*, an inverted index on the path collection. We can take advantage of $\mathcal{P}$-*Index* in two ways: (i) for accessing all paths that contain the current search node (Line 7 in Figure 2), and (ii) for enforcing a quick termination condition.

Definition 5 ($\mathcal{P}$-*Index*). *The* path collection index *of* $\mathbf{P}$*, denoted as* $\mathcal{P}$-*Index*($\mathbf{P}$)*, consists of* paths *lists for all nodes in* $\mathbf{P}$*. The list* $paths[v_i]$ *for node* v_i *contains entries* $\langle p_j{:}o_{ij}\rangle$*, where* o_{ij} *indicates the position of* v_i *in* p_j*, for all paths* p_j *that include* v_i*. The entries are stored sorted by their path identifier* p_j*.*

Example 3. Table 1 illustrates the path collection index $\mathcal{P}$-*Index*($\mathbf{P}$) for the collection $\mathbf{P}$ presented in Figure 1(a).

node	*paths* **list**
A	$\langle p_1{:}1\rangle, \langle p_2{:}1\rangle, \langle p_5{:}1\rangle$
B	$\langle p_1{:}2\rangle, \langle p_2{:}5\rangle, \langle p_4{:}3\rangle$
C	$\langle p_1{:}3\rangle$
D	$\langle p_1{:}4\rangle, \langle p_2{:}3\rangle, \langle p_4{:}1\rangle$
F	$\langle p_2{:}2\rangle, \langle p_4{:}4\rangle, \langle p_5{:}2\rangle$
J	$\langle p_1{:}5\rangle$
K	$\langle p_4{:}5\rangle, \langle p_5{:}3\rangle$
L	$\langle p_3{:}2\rangle$
M	$\langle p_3{:}3\rangle$
N	$\langle p_2{:}4\rangle, \langle p_3{:}1\rangle, \langle p_4{:}2\rangle$
T	$\langle p_2{:}6\rangle$

Table 1. $\mathcal{P}$-*Index* for the path collection $\mathbf{P}$ in Figure 1(a).

We introduce pfsP as an extention to pfs algorithm that exploits $\mathcal{P}$-*Index*. Algorithm pfsP identifies all paths that contain a node v_n by performing a linear scan of list $paths[v_n]$.

Furthermore, pfsP exploits $\mathcal{P}$-*Index* to define a fast termination condition. Assume that node v_n has just been popped out (Line 5). The search can be terminated if there exists a path p_c in the collection that contains both v_n and target v_t, such that, v_t comes after v_n. Specifically, pfsP looks for entries $\langle p_c{:}o_{nc}\rangle$, $\langle p_c{:}o_{tc}\rangle$ in lists $paths[v_n]$, $paths[v_t]$ respectively, such that $o_{nc} < o_{tc}$. The procedure is similar to a merge-join that finishes as soon as such a path is found or one of the lists is traversed to the end.

The pfsP is similar to pfs with the exception that it performs the described check. The improved termination condition can be included in Figure 2 by changing Line 6. Figure 3 illustrates the pseudocode of pfsP algorithm.

The ConstructPathP method first calls ConstructPath($v_s, v_n, \mathcal{A}$) to construct the path from source v_s to current search node v_n, i.e., path(v_s, v_n). Then, it concatenates path(v_s, v_n) with the part of p_c from v_n up to v_t. During concatenation the method ensures that each node is contained only once in the answer path. For example, consider path(A, T). After joining $paths[D] = \{\langle p_1{:}4\rangle, \langle p_2{:}3\rangle, \langle p_4{:}1\rangle\}$ and $paths[T] = \{\langle p_2{:}6\rangle\}$ lists we identify common path p_2. Method ConstructPathP first constructs path(A, D) $= (A, B, C, D)$ using set $\mathcal{A}$ and then concatenates it with the part of $p_1(A, F, D, N, B, T)$ from D up to T. Since node B is contained in path(A, D) the answer path is (A, B, T).

Algorithm pfsP
Input: nodes v_s and v_t of a path collection $\mathbf{P}$, $\mathcal{P}$-$Index(\mathbf{P})$
Output: a path from v_s to v_t
Parameters:
stack $\mathcal{Q}$: // the search stack
set $\mathcal{H}$: // contains all nodes pushed in $\mathcal{Q}$
set $\mathcal{A}$: // contains the direct ancestor of each node in $\mathcal{H}$

Method:

1. push(v_s, $\mathcal{Q}$);
2. **insert** v_s in $\mathcal{H}$
3. **insert** $\langle v_s, \varnothing \rangle$ in $\mathcal{A}$;
4. **while** $\mathcal{Q}$ is not empty **do**
5. **let** v_n = pop($\mathcal{Q}$);
6. **if** there is a path $p_c \in \mathbf{P}$ containing v_n before v_t **then**
   return ConstructPathP($v_s, v_n, v_t, \mathcal{A}, p_c$);
7. **for** each path $p \in \mathbf{P}$ containing v_n **do**
8. **let** v_p be the node after v_n in p;
9. **while** $v_p \notin \mathcal{H}$ **do** // access each node v_p after v_n in p until the first v_p node // contained in $\mathcal{H}$
10. push(v_p, $\mathcal{Q}$);
11. **insert** v_p in $\mathcal{H}$;
12. **insert** $\langle v_p, v_p^- \rangle$ in $\mathcal{A}$, where v_p^- is the direct ancestor of v_p in p;
13. **let** v_p be the next node in p;
14. **end while**
15. **end for**
16. **end while**
17. **return null**;

Fig. 3. Algorithm pfsP.

Example 4. We illustrate the pfsP algorithm for the query `path`(F, C) on the path collection $\mathbf{P}$ of Example 2 exploiting the join procedure of the *paths* lists. We use $\mathcal{P}$-$Index(\mathbf{P})$ presented in Table 1.

The first three iterations are identical to the first three iterations of the pfs algorithm presented in Example 2. Summarizing, after these iterations the stack and the sets of pfsP are as follows.

$$\mathcal{Q} = \{D, N, B\},$$
$$\mathcal{H} = \{F, D, N, B, T, K\} \text{ and}$$
$$\mathcal{A} = \{\langle F, \varnothing \rangle, \langle D, F \rangle, \langle N, D \rangle, \langle B, N \rangle, \langle T, B \rangle, \langle K, F \rangle\}.$$

At the fourth iteration of the while loop, Algorithm pfsP pops B. To execute Line 6, we join the *paths* list of current search node B, $paths[B] = \{\langle p_1{:}2 \rangle, \langle p_2{:}5 \rangle, \langle p_4{:}3 \rangle\}$ with the *paths* list for target node C, $paths[C] = \{\langle p_1{:}3 \rangle\}$. The join procedure identifies entries $\langle p_1{:}2 \rangle$, $\langle p_1{:}3 \rangle$ for common path $p_c = p_1$. Since in p_1, B is before C, the search terminates successfully. The answer path (F, D, N, B, C) is the concatenation of (F, D, N, B) (which corresponds to the path from source F to current node B and is constructed using set $\mathcal{A}$) and (B, C) (which corresponds to the part of p_1 that connect B to target node C).

5 Capturing reachability information using $\mathcal{H}$-*graphs*

Section 5.1 introduces the $\mathcal{H}$-*graph* and its associated structure $\mathcal{H}$-*Index*. Section 5.2 discusses the extension of pfs using the $\mathcal{H}$-*Index*.

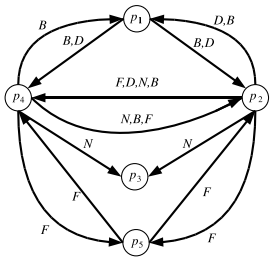
5.1 The $\mathcal{H}$-*graph* and its $\mathcal{H}$-*Index*

The $\mathcal{H}$-*graph* provides additional reachability information by identifying shared nodes and, thus, possible transitions, among paths.

Definition 6 ($\mathcal{H}$-*graph*). *Let* $\mathbf{P} = \{p_1, ..., p_n\}$ *be a path collection. The* $\mathcal{H}$-*graph of* $\mathbf{P}$*, denoted by* $\mathcal{H}$-*graph*$(\mathbf{P})$*, is a labeled directed graph* (V, E) *such that* V *consists of all paths in* $\mathbf{P}$ *and a labeled edge* $(p_i, p_j, v) \in E$ *if paths* p_i*,* p_j *have a common node* $v \in nodes(\mathbf{P})$*, termed* link*, which is neither the first node of* p_i *nor the last of* p_j*.*

Given a path collection $\mathbf{P}$ and $\mathcal{P}$-*Index*$(\mathbf{P})$, the $\mathcal{H}$-*graph*$(\mathbf{P})$ is constructed as follows. For each node $v_k \in nodes(\mathbf{P})$ and each pair of entries $\langle p_i{:}o_{ki}\rangle, \langle p_j{:}o_{kj}\rangle \in paths[v]$, we construct a directed edge from p_i to p_j in $\mathcal{H}$-*graph*$(\mathbf{P})$ and label it with link v. Intuitively, an edge (p_i, p_j, v) denotes that all nodes in p_i before link v_k can reach the nodes after v_k in p_j. If the link lies in the beginning of p_i or at the end of p_j, there is no useful reachability information since no node is contained before v_k in p_i or after v_k in p_j, and hence the edge is omitted from $\mathcal{H}$-*graph*.

Example 5. Figure 4(a) illustrates $\mathcal{H}$-*graph*$(\mathbf{P})$ for the path collection $\mathbf{P}$ of Figure 1(a). To increase readability, multiple edges between the same pair of paths are collapsed into a single edge with multiple labels. For example, the single edge from p_4 to p_2 labeled with N, B, F links corresponds to edges (p_4, p_2, N), (p_4, p_2, B) and (p_4, p_2, F). Note that edge (p_4, p_1, D) is not included since D is the first node in p_4.



(a)

path	*edges* **list**
p_1	$\langle p_2, B{:}2{:}5\rangle, \langle p_2, D{:}4{:}3\rangle, \langle p_4, B{:}2{:}3\rangle, \langle p_4, D{:}4{:}1\rangle$
p_2	$\langle p_1, D{:}3{:}4\rangle, \langle p_1, B{:}5{:}2\rangle, \langle p_3, N{:}4{:}1\rangle, \langle p_4, F{:}2{:}4\rangle, \langle p_4, D{:}3{:}1\rangle,$ $\langle p_4, N{:}4{:}2\rangle, \langle p_4, B{:}5{:}3\rangle, \langle p_5, F{:}2{:}2\rangle$
p_3	
p_4	$\langle p_1, B{:}3{:}2\rangle, \langle p_2, N{:}2{:}4\rangle, \langle p_2, B{:}3{:}5\rangle, \langle p_2, F{:}4{:}2\rangle, \langle p_3, N{:}2{:}1\rangle,$ $\langle p_5, F{:}4{:}2\rangle$
p_5	$\langle p_2, F{:}2{:}2\rangle, \langle p_4, F{:}2{:}4\rangle$

(b)

Fig. 4. (a) $\mathcal{H}$-*graph*$(\mathbf{P})$ of the path collection $\mathbf{P}$ in Figure 1(a), (b) $\mathcal{H}$-*Index* for $\mathcal{H}$-*graph*$(\mathbf{P})$.

The $\mathcal{H}$-*graph* of a path collection $\mathbf{P}$ is stored in a modified adjacency list representation denoted as $\mathcal{H}$-*Index*.

Definition 7 ($\mathcal{H}$-*Index*). *The $\mathcal{H}$-graph* index *of $\mathbf{P}$, denoted as $\mathcal{H}$-Index($\mathbf{P}$), consists of* edges *lists for all paths in $\mathbf{P}$. The list edges[p_i] for path p_i has entries of the form $\langle p_j, v_k{:}o_{ki}{:}o_{kj}\rangle$, for each (p_i, p_j, v_k) edge of $\mathcal{H}$-graph($\mathbf{P}$), where o_{ki} (o_{kj}) denotes the position of the link node v_k in path p_i (p_j). Entries are sorted primarily by the path p_j of the outgoing edge, and secondarily by o_{ki}.*

Example 6. Figure 4(b) illustrates the $\mathcal{H}$-*Index*($\mathbf{P}$) of the $\mathcal{H}$-*graph*($\mathbf{P}$) presented in Figure 4(a).

5.2 The pfsH algorithm

The $\mathcal{H}$-*graph* captures intersections among paths, and hence contains additional information about nodes' reachability compared to that included in the paths alone. To illustrate this, consider node F of path p_2 and node C of path p_1 of the collection in Figure 1(a). The information in $\mathcal{H}$-*Index* suffices to show that a path from F to C exists. In particular, the entry $\langle p_1, B{:}5{:}2\rangle$ of *edges*[p_2] in $\mathcal{H}$-*Index* denotes that there is way from p_2 to p_1 via B. Further, from $\mathcal{P}$-*Index* one derives that B is after F in p_2 and before C in p_1. Hence a path (F, D, N, B, C) can be constructed by combining paths p_2 and p_1.

The above observation is the main idea of the pfsH algorithm. Consider the query path(v_s, v_t) and assume that the current search node is v_n. For each path p_i that contains v_n, the algorithm checks whether an edge (p_i, p_j, v_k) in $\mathcal{H}$-*graph* satisfies three conditions:

(i) p_j contains the target node v_t,
(ii) link v_k is after current search node v_n in p_i, and
(iii) v_k is before v_t in p_j.

If these hold, a path from v_n to target v_t, via v_k exists, and thus a path from source v_s to v_t can be found.

Figure 5 illustrates the pseudocode of pfsH algorithm. The pfsH is similar to pfs with the exception that it introduces two termination conditions in Line 1 and Line 8. First, in Line 1 before initializing stack $\mathcal{Q}$ and sets $\mathcal{H}$, $\mathcal{A}$, the algorithm checks whether there exists a path p_c in the collection containing source v_s before target v_t. To perform this check pfsH exploits the merge-join procedure of *paths*[v_s] and *paths*[v_t] lists introduced for pfsP in Section 4.2. If a path p_c is identified the search terminates and the ConstructPathP method returns the part of p_c from v_s to v_t as the answer path.

Otherwise, in Line 8, as soon as a new path p_i containing the current search node v_n in position o_{ni} is examined, pfsH checks whether there exists an edge (p_i, p_j, v_k) in $\mathcal{H}$-*graph* satisfying the aforementioned three conditions. The algorithm scans lists *edges*[p_i] and *paths*[v_t] from $\mathcal{H}$-*Index*($\mathbf{P}$) and $\mathcal{P}$-*Index*($\mathbf{P}$), respectively, similar to a merge-join as both are sorted on the path identifier. The scan terminates when $\langle p_j, v_k{:}o_{ki}{:}o_{kj}\rangle$ in *edges*[p_i] and $\langle p_j{:}o_{tj}\rangle$ in *paths*[v_t] match. More specifically, the following three conditions are satisfied:

Algorithm pfsH
Input: nodes v_s and v_t of a path collection $\mathbf{P}$, $\mathcal{P}$-$Index(\mathbf{P})$, $\mathcal{H}$-$Index(\mathbf{P})$
Output: a path from v_s to v_t
Parameters:
stack $\mathcal{Q}$: // the search stack
set $\mathcal{H}$: // contains all nodes pushed in $\mathcal{Q}$
set $\mathcal{A}$: // contains the direct ancestor of each node in $\mathcal{H}$

Method:

1. **if** there is a path $p_c \in \mathbf{P}$ containing v_s before v_t **then return** ConstructPathP $(v_s, v_s, v_t, \mathcal{A}, p_c)$;
2. push$(v_s, \mathcal{Q})$;
3. **insert** v_s in $\mathcal{H}$
4. **insert** $\langle v_s, \varnothing \rangle$ in $\mathcal{A}$;
5. **while** $\mathcal{Q}$ is not empty **do**
6. **let** v_n = pop$(\mathcal{Q})$;
7. **for** each path $p \in \mathbf{P}$ containing v_n **do**
8. **if** there is an edge $(p, p_c, v_k) \in \mathcal{H}$-$graph(\mathbf{P})$ such that v_n is before v_k in p and v_k before v_t in p_c **then return** ConstructPathH $(v_s, v_n, v_k, v_t, \mathcal{A}, p, p_c)$;
9. **let** v_p be the node after v_n in p;
10. **while** $v_p \notin \mathcal{H}$ **do** // access each node v_p after v_n in p until the first v_p node // contained in $\mathcal{H}$
11. push$(v_p, \mathcal{Q})$;
12. **insert** v_p in $\mathcal{H}$;
13. **insert** $\langle v_p, v_p^- \rangle$ in $\mathcal{A}$, where v_p^- is the direct ancestor of v_p in p;
14. **let** v_p be the next node in p;
15. **end while**
16. **end for**
17. **end while**
18. **return null**;

Fig. 5. Algorithm pfsH.

(i) the entries correspond to the same path p_j,
(ii) path p_i contains link v_k after current source node v_k, i.e., $o_{ki} > o_{ni}$, and
(iii) path p_j contains link v_k before target node v_t, i.e., $o_{kj} < o_{tj}$

When a qualifying entry $\langle p_j, v_k{:}o_{ki}{:}o_{kj} \rangle$ in $edges[p_i]$ is found, ConstructPathH first constructs path(v_s, v_n), calling ConstructPath$(v_s, v_n, \mathcal{A})$, and then concatenates it with the part of p from v_n to v_k and the part of p_c from v_k to v_t.

Example 7. We illustrate the pfsH algorithm for the query path(F, C) on the path collection of $\mathbf{P}$ of Example 2. Algorithm pfsH exploits $\mathcal{H}$-$Index(\mathbf{P})$ presented in Figure 4(b) and $\mathcal{P}$-$Index(\mathbf{P})$ of Table 1.

First, we check whether exists a path in $\mathbf{P}$ containing source F before target C. The join between $paths[F] = \{\langle p_2{:}2 \rangle, \langle p_4{:}4 \rangle, \langle p_5{:}2 \rangle\}$ and $paths[C] = \{\langle p_1{:}3 \rangle\}$ lists results in no common path, and therefore, we need to further search the collection.

At the first iteration of the outer while loop, Algorithm pfsH pops F. Node F is contained in paths $p_2(A, F, D, N, B, T)$, $p_4(D, N, B, F, K)$ and $p_5(A, F, K)$. Then, we check the termination condition for paths p_2, p_4 and p_5 and perform a join of the corresponding $edges$ list with $paths[C] = \{\langle p_1 : 3 \rangle\}$. The join of $edges[p_2] = \{\langle p_1, D : 3 : 4 \rangle, \langle p_1, B : 5 : 2 \rangle, \langle p_3, N : 4 : 1 \rangle, \langle p_4, F : 2 : 5 \rangle, \langle p_4, D : 3 : 1 \rangle, \langle p_4, N{:}4{:}2 \rangle, \langle p_4, B{:}5{:}3 \rangle, \langle p_5, F{:}2{:}2 \rangle\}$ with $paths[C] = \{\langle p_1{:}3 \rangle\}$ results in common

path p_1 (condition (i)) with the link node B of (p_2, p_1, B) edge contained after F in p_2 (condition (ii)) and before C in p_1 (condition (iii)). Thus, the answer path is (F, D, N, B, C).

6 Updating path collections

Updating a path collection involves adding new paths. To include a new path p_j in a collection, we need (a) to insert the entry $p_j{:}o_{ij}$ in the $paths[v_i]$ for each node v_i contained in p_j (update $\mathcal{P}$-$Index$), and (b) to update the $edges[p_j]$ and the $edges$ lists of the paths containing each node in p_j (update $\mathcal{H}$-$Index$).

In practice, path collections are usually very large to fit in the main memory. Therefore, both $\mathcal{P}$-$Index$ and $\mathcal{H}$-$Index$ of a collection are stored as inverted files on secondary storage and maintained mainly by batch, offline updates. In other words, we usually update the collection with a set of new paths. A requirement for the inverted files to work efficiently is to store the inverted lists, like $paths$ and $edges$ lists, in a contiguous way on secondary storage. Due to this requirement the naïve solution to deal with each new path separately is not efficient for updating the collection. A common approach to this problem is to built a $\mathcal{P}$-$Index$ and an $\mathcal{H}$-$Index$ considering the new paths as inverted indices in main memory and to exploit them for evaluating the queries in parallel with the disk-based $\mathcal{P}$-$Index$ and $\mathcal{H}$-$Index$ of the collection.

Each time a set of new paths arrives, we update only the memory-based indices with minimum cost. Then, to update the disk-based indices, there are three possible strategies: (a) rebuilding them from scratch using both the old and the new paths, (b) merging them with the memory resident ones and (c) lazily updating the $paths$ and the $edges$ lists when they are retrieved from disk during query evaluation. A very helpful survey on update techniques for inverted files can be found at [15]. In our work, we adopt the second strategy for updating the disk-based indices.

Figures 6 and 7 illustrate the procedures required for updating the memory-based indices with the new paths (Procedures updateMP and updateMH) and the merging procedures of the disk-based indices with the memory-based ones (Procedures mergeP and mergeH). Procedures updateMP and updateMH work similarly with the procedures for creating disk-based $\mathcal{P}$-$Index$ and $\mathcal{H}$-$Index$ respectively, from scratch. Especially for updateMH, we also need to create entries considering both the new paths and the existing paths of the collection (Lines 6-9). Finally, Procedures mergeP and mergeH merge disk-based $paths$ and $edges$ lists respectively with the memory resident ones, and then write the new lists on disk.

7 Experiments

We present an experimental evaluation of our methods demonstrating their efficiency. We compare the pfsP and pfsH algorithms against conventional depth-first search which operates on the underlying graph $G_{\mathbf{P}}$, indexed by adjacency lists.

Procedure updateMP
Inputs: memory-based $\mathcal{P}$-$Index(\mathbf{P})$ MP, set of new paths $\mathcal{U}$
Output: updated memory-based $\mathcal{P}$-$Index(\mathbf{P})$ MP
Method:

1. **for** each new path p_j in $\mathcal{U}$ **do**
2. **for** each node v_k in p **do**
3. **append** $\langle p_j{:}o_{kj} \rangle$ entry at the end of $paths[v_k]$ in MP;
4. **end for**
5. **end for**

Procedure mergeP
Inputs: updated memory-based $\mathcal{P}$-$Index(\mathbf{P})$ MP, disk-based $\mathcal{P}$-$Index(\mathbf{P})$ DP
Output: updated disk-based $\mathcal{P}$-$Index(\mathbf{P})$ DP
Method:

1. **for** each node v in MP **do**
2. **append** contents of $paths[v]$ of MP at the end of $paths[v]$ of DP;
3. **write** new $paths[v]$ on DP;
4. **end for**

Fig. 6. Procedures for updating $\mathcal{P}$-$Index$.

All indices, i.e., $\mathcal{P}$-$Index$ and $\mathcal{H}$-$Index$ for the collections, and the adjacency lists for $G_{\mathbf{P}}$, are implemented as inverted files using the Berkeley DB storage engine. All algorithms are implemented in C++ and compiled with gcc. The experimental evaluation was performed on a 3 Ghz Intel Core 2 Duo CPU.

For updating the adjacency lists of $G_{\mathbf{P}}$ graph, we adopt an approach similar to that for $\mathcal{P}$-$Index$ and $\mathcal{H}$-$Index$. In addition, we choose not to check whether the new paths contain a transition between two nodes more than once or if a transition is already included as an edge in $G_{\mathbf{P}}$ graph. Instead, duplicates are removed while merging the disk-based adjacency lists with the memory-based ones. This approach allows for fast updates on the adjacency lists and $G_{\mathbf{P}}$, at the expense of increased main memory utilization.

We generate synthetic path collections to test the methods. We identify five experimental parameters: (a) $|\mathbf{P}|$: the number of paths in the collection, (b) l_{avr}: the average path length, (c) $|V|$: the number of distinct nodes in the paths, (d) $zipf$: the order of Zipfian distribution of node frequency, and (e) U: the update factor. The path collections contain 50000 up to 500000 paths. The average length of each path varies between 5 to 30 nodes. Path collections include 10000 up to 500000 distinct nodes. Note that varying the number of nodes in the collection also affects the number of link (common) nodes and the possible transitions between the paths. Node frequency is a moderately skewed Zipfian distribution of order $zipf$ that varies from 0 up to 0.8. Note that nodes with high frequency are contained in a lot of paths. An update factor U means that there are $U\% \cdot |\mathbf{P}|$ new paths to be added to the collection $\mathbf{P}$. Table 2 summarizes all parameters.

We perform four sets of experiments to show the effects on the size and the construction time of the indices, as well as on the performance of the algorithms for evaluating reachability queries. In each set, we vary one of P, l_{avg}, V, $zipf$ while we keep the remaining three parameters fixed to their default values (see Table 2). In the fifth set of experiments we study the updates of the path col-

Procedure updateMH
Inputs: updated memory-based $\mathcal{P}$-$Index(\mathbf{P})$ MP, disk-based $\mathcal{P}$-$Index(\mathbf{P})$ DP, memory-based $\mathcal{H}$-$Index(\mathbf{P})$ MH
Output: updated memory-based $\mathcal{H}$-$Index(\mathbf{P})$ MH
Method:

1. **for** each node v_k in MP **do**
2. **for** each pair of entries $\langle p_i{:}o_{ki}\rangle, \langle p_j{:}o_{kj}\rangle$ in $paths[v_k]$ of MP **do**
3. **if** $o_{ki} > 1$ **and** $o_{kj} < l_{p_j}$ **then insert** $\langle p_j, v_k{:}o_{ki}{:}o_{kj}\rangle$ in $edges[p_i]$ of MH;
4. **if** $o_{kj} > 1$ **and** $o_{ki} < l_{p_i}$ **then insert** $\langle p_i, v_k{:}o_{kj}{:}o_{ki}\rangle$ in $edges[p_j]$ of MH;
5. **end for**
6. **for** each pair of entries $\langle p_i{:}o_{ki}\rangle, \langle p_j{:}\, o_{kj}\rangle$ where $p_i \in paths[v_k]$ of MP and $p_j \in paths[v_k]$ of DP **do**
7. **if** $o_{ki} > 1$ **and** $o_{kj} < l_{p_j}$ **then insert** $\langle p_j, v_k{:}o_{ki}{:}o_{kj}\rangle$ in $edges[p_i]$ of MH;
8. **if** $o_{kj} > 1$ **and** $o_{ki} < l_{p_i}$ **then insert** $\langle p_i, v_k{:}o_{kj}{:}o_{ki}\rangle$ in $edges[p_j]$ of MH;
9. **end for**
10. **end for**

Procedure mergeH
Inputs: updated memory-based $\mathcal{H}$-$Index(\mathbf{P})$ MH, disk-based $\mathcal{H}$-$Index(\mathbf{P})$ DH
Output: updated disk-based $\mathcal{H}$-$Index(\mathbf{P})$ DH
Method:

1. **for** each path p in MH **do**
2. **merge** $edges[p]$ of DH with $edges[p]$ of MH;
3. **write** new $edges[p]$ on DH;
4. **end for**

Fig. 7. Procedures for updating $\mathcal{H}$-$Index$.

lections and vary only the U parameter while we set the remaining four fixed to their default values.

parameter	values	default value
$\|\mathbf{P}\|$	50000, 100000, 500000	100000
l_{avg}	5, 10, 30	10
$\|V\|$	10000, 50000, 100000, 500000	100000
$zipf$	0, 0.3, 0.6, 0.8	0.6
U	1%, 5%, 10%	-

Table 2. Experimental parameters

7.1 Varying the number of paths in the collection

Figure 8(a) illustrates the effect on the index size. We note that in all cases, $\mathcal{H}$-$Index$ requires at least one order of magnitude more space than the other two indices. $\mathcal{P}$-$Index$ is slightly larger then the size of the adjacency lists. As $|\mathbf{P}|$ increases all indices require more disk space. The size of the adjacency lists increases, because the path collections include more direct transitions between path nodes resulting in more dense $G_{\mathbf{P}}$ graphs. As expected $\mathcal{P}$-$Index$ requires

more space since each node is contained in more paths and therefore, the length of the *paths* lists increases. Finally, as $|\mathbf{P}|$ increases, the paths have more nodes in common, which means that the length of the *edges* lists increases too. Thus, $\mathcal{H}$-*Index* also requires more disk space.

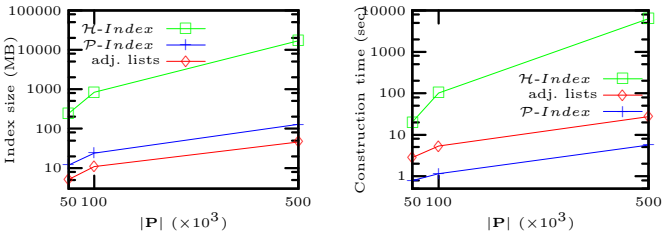


Fig. 8. (a) Index size, and (b) construction time varying $|\mathbf{P}|$, for $l_{avg} = 10$, $|V| = 100000$, $zipf = 0.6$.

Figure 8(b) shows the effect on the construction time of the indices. As $|\mathbf{P}|$ increases the construction of all indices takes more time. We notice that the creation time of the adjacency lists is almost one order of magnitude higher than the time for $\mathcal{P}$-*Index*, in all cases. This is due to the fact that, we first need to construct $G_{\mathbf{P}}$ graph by removing repeated transitions between nodes. On the other hand, the construction time of $\mathcal{H}$-*Index* is always approximately one order of magnitude higher than the time of the other indices.

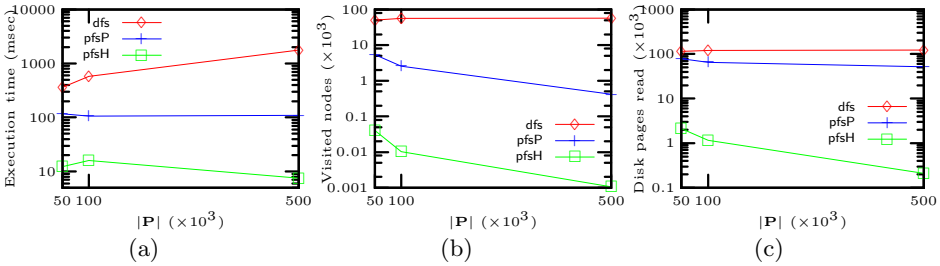


Fig. 9. (a) Average execution time, (b) average number of visited nodes, (c) average number of disk pages read varying $|\mathbf{P}|$, for $l_{avg} = 10$, $|V| = 100000$, $zipf = 0.6$.

We generate 5000 random reachability queries and evaluate them over the three collections and their $G_{\mathbf{P}}$ graphs. Figure 9(a) presents the effects of $|\mathbf{P}|$ on the execution time. In all cases, the average execution time of pfsP and pfsH is lower than that of dfs. pfsP is always almost one order of magnitude faster

than dfs. pfsH is two orders of magnitude faster than dfs. As $|\mathbf{P}|$ increases, the execution time of dfs increases too. This is expected since $G_{\mathbf{P}}$ graph becomes more dense. In contrast, pfsP is less affected by the increase of $|\mathbf{P}|$, whereas the execution time of pfsH decreases. This is because, the length of the *paths* lists in $\mathcal{P}$-*Index* and the *edges* lists in $\mathcal{H}$-*Index* increases and it is very likely that the join procedures in pfsP and pfsH will identify common paths. Thus pfsP and pfsH, in all cases, need to visit fewer nodes to answer a query as Figure 9(b) shows. Figure 9(c) confirms the above observations with respect to the number of I/Os.

7.2 Varying the average length of the paths in the collection

Figure 10(a) shows the effects of l_{avg} on the disk space required to store $\mathcal{P}$-*Index*, $\mathcal{H}$-*Index* and the adjacency lists of $G_{\mathbf{P}}$ graph. Similarly to the case of increasing $|\mathbf{P}|$, we notice that the size of $\mathcal{H}$-*Index* is more than one order of magnitude larger compared to the size of the adjacency lists of $G_{\mathbf{P}}$ graph. In contrast, $\mathcal{P}$-*Index* is slightly larger than the adjacency lists. As l_{avg} increases, there are more direct transitions between path nodes. Thus, $G_{\mathbf{P}}$ graph becomes more dense and its adjacency lists contain more nodes. Finally, as (a) the l_{avg} increases, and (b) $|V|$ remains fixed, the number of occurrences in the paths for each node increases. This results to longer *paths* lists in $\mathcal{P}$-*Index* and to longer *edges* lists in $\mathcal{H}$-*Index* too, because there exist more common nodes between paths. Therefore, the space needed to store $\mathcal{P}$-*Index* and $\mathcal{H}$-*Index* also increases.

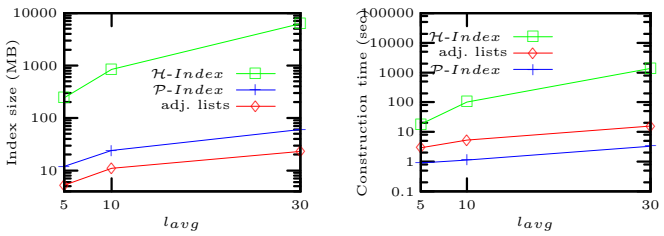


Fig. 10. (a) Index size, and (b) construction time varying l_{avg}, for $|\mathbf{P}| = 100000$, $|V| = 100000$, $zipf = 0.6$.

Figure 10(b) shows the effect on the construction time of the indices. The creation of all indices takes more time as l_{avg} increases. Similarly to the case of varying $|\mathbf{P}|$, the construction time of the adjacency lists is higher than the time of $\mathcal{P}$-*Index*, since we first need to construct $G_{\mathbf{P}}$ graph by removing the repeated transitions between the nodes. The construction time of $\mathcal{H}$-*Index* is always at least one order of magnitude higher than the time of the other indices.

We generate 5000 random reachability queries and evaluate them over the three collections and their $G_{\mathbf{P}}$ graphs. Figure 11(a) presents the effects of varying

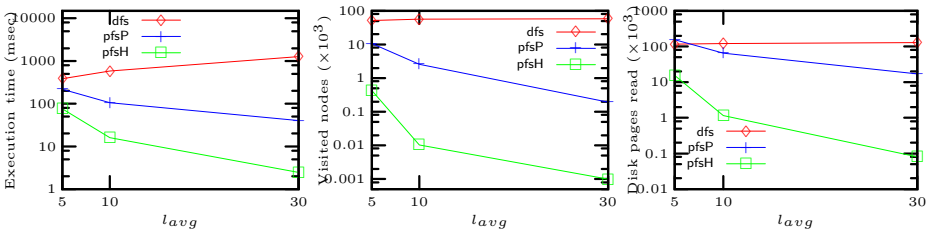


Fig. 11. (a) Average execution time, (b) average number of visited nodes, (c) average number of disk pages read varying l_{avg}, for $|\mathbf{P}| = 100000$, $|V| = 100000$, $zipf = 0.6$.

l_{avg} on the execution time. The experimental results show that the average execution time of pfsP and pfsH is lower than that of dfs in all cases. Moreover, as l_{avg} increases, the execution time of dfs increases, whereas the execution time of pfsP and pfsH decreases. dfs becomes slower because the density of $G_{\mathbf{P}}$ increases. On the other hand, the join procedures in pfsP and pfsH will quickly identify a common path, since *paths* and *edges* lists become longer. Thus, both pfsP and pfsH need to visit fewer nodes to answer a query as Figure 11(b) shows. Figure 9(c) confirms the above findings with respect to the number of I/Os.

7.3 Varying the number of nodes in the path collection

Figure 12(a) illustrates the space requirements for the adjacency lists of $G_{\mathbf{P}}$ graph, and for $\mathcal{P}$-*Index* and $\mathcal{H}$-*Index* of the collection. As $|V|$ increases the adjacency lists and $\mathcal{P}$-*Index* require more disk space. In the case of the adjacency lists, this is because $G_{\mathbf{P}}$ becomes larger and more lists need to be stored. Similarly, the size of $\mathcal{P}$-*Index* also grows as $|V|$ increases since it contains more *paths* lists. On the other hand, $\mathcal{H}$-*Index* requires less disk space as $|V|$ increases. This is due to the fact that the paths have fewer common nodes and thus the *edges* lists become shorter. Note that the total number of *edges* lists does not change as $|\mathbf{P}|$ is fixed to 100000.

Figure 12(b) shows the impact of varying the number of nodes in the collection on the construction time of the indices. As $|V|$ increases the construction of the adjacency lists of $G_{\mathbf{P}}$ and $\mathcal{P}$-*Index* takes more time. Similarly to the previous experiments, the construction time for adjacency lists is higher since we need to construct $G_{\mathbf{P}}$ graph first. On the hand, the construction of $\mathcal{H}$-*Index* takes less time since the *edges* lists become shorter.

We generate 5000 random reachability queries and evaluate them over the collections and their $G_{\mathbf{P}}$ graph. Figure 13(a) illustrates the effect of varying $|V|$ on the query execution time. All three algorithms are affected by the increase of the nodes in the collection. Algorithms pfsP and pfsH are, in all cases, faster than dfs but the difference in the execution time decreases as $|V|$ increases. The performance of dfs is expected because graph $G_{\mathbf{P}}$ becomes larger as $|V|$ increases.

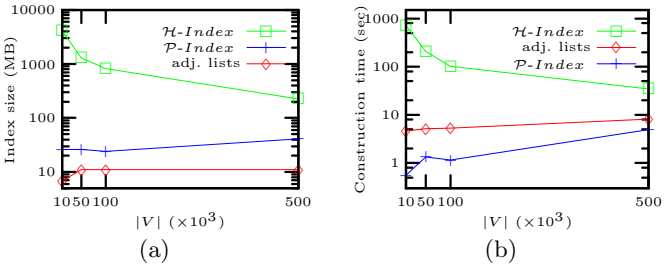


Fig. 12. (a) Index size, and (b) construction time varying $|V|$, for $|\mathbf{P}| = 100000$, $l_{avg} = 10$, $zipf = 0.6$.

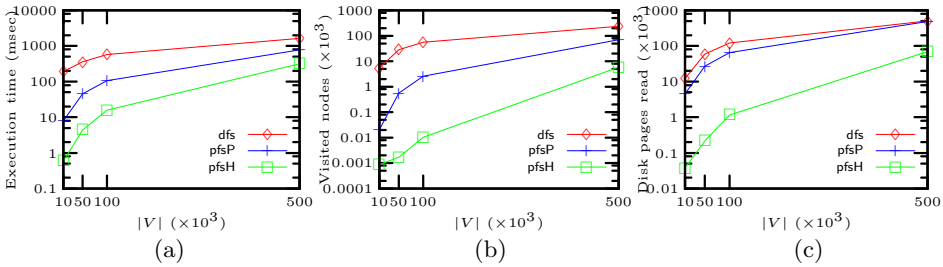


Fig. 13. (a) Average execution time, (b) average number of visited nodes, (c) average number of disk pages read varying $|V|$, for $|\mathbf{P}| = 100000$, $l_{avg} = 10$, $zipf = 0.6$.

Considering pfsP and pfsH, since (a) the collections include more nodes and (b) the number of paths is fixed, the number of node occurrences in paths decreases and in addition the paths have less nodes in common. In other words, since the *paths* lists of $\mathcal{P}$-*Index* and *edges* lists of $\mathcal{H}$-*Index* become shorter, they will likely have fewer common paths. Thus, both pfsP and pfsH need to visit more nodes to answer the queries as Figure 13(b) shows. Figure 13(c) confirms the above observations with respect to the number of I/Os.

7.4 Varying node frequency in the path collection

Figure 14(a) illustrates the space requirements for the adjacency lists of $G_{\mathbf{P}}$ graph, and for $\mathcal{P}$-*Index* and $\mathcal{H}$-*Index*. As expected, the increase in the order of the Zipfian distribution does not affect the size of the adjacency lists. The total number of direct transitions between the nodes of the collection does not change as *zipf* increases. Thus, the total number of edges in the $G_{\mathbf{P}}$ graph remains the same, and so does the size of the adjacency lists. The increase of *zipf* order does not change the total number of entries of the *paths* lists in $\mathcal{P}$-*Index*, and therefore the size of $\mathcal{P}$-*Index* remains the same. On the other hand the size of $\mathcal{H}$-*Index* increases. In fact, the size of $\mathcal{H}$-*Index* is three orders of magnitude

larger compared to the size of the adjacency lists for $zipf = 0.8$. As $zipf$ value increases some nodes can act as link nodes for more paths of the collection. Thus, the *edges* lists become longer and the size of $\mathcal{H}$-$Index$ increases.

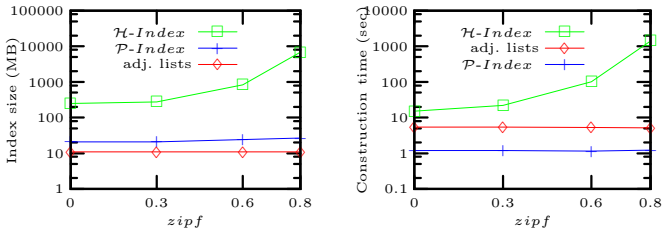


Fig. 14. (a) Index size, and (b) construction time varying $zipf$, for $|\mathbf{P}| = 100000$, $|V| = 100000$, $l_{avg} = 10$.

Figure 14(b) shows the impact of varying the number of nodes in the collection on the construction time of the indices. As expected the construction time for the adjacency lists and $\mathcal{P}$-$Index$ is not affected by the increase of $zipf$. In contrast, as $|V|$ increases the construction time of $\mathcal{H}$-$Index$ increases.

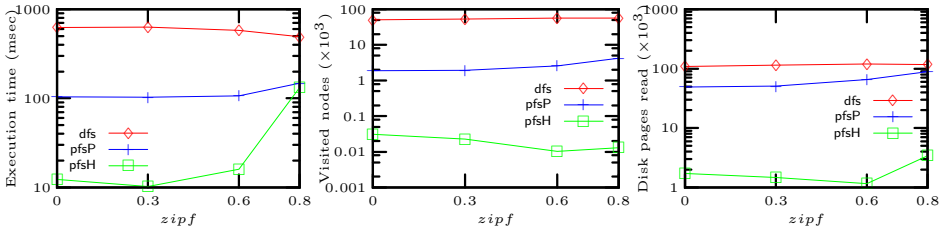


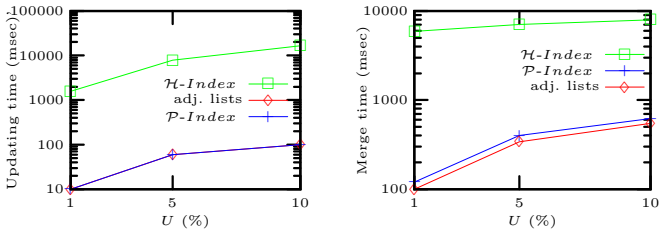
Fig. 15. (a) Average execution time, (b) average number of visited nodes, (c) average number of disk pages read varying $zipf$, for $|\mathbf{P}| = 100000$, $|V| = 100000$, $l_{avg} = 10$.

We generate 5000 random reachability queries and evaluate them over the four collections and their $G_{\mathbf{P}}$ graph. Figure 15(a) shows the effect of varying node frequency on the query execution time. We notice that the execution time of pfsP and pfsH is always lower than the execution time of dfs. Algorithm pfsH is faster than pfsP for approximately one order of magnitude for $zipf < 0.8$. As expected the execution time remains approximately stable since $G_{\mathbf{P}}$ does not change as $zipf$ increases, whereas the execution time of the other two algorithms increases. The increase in the case of pfsP is less intense. Figure 15(b) shows pfsP

needs to visit slightly more nodes as *zipf* increases. On the other hand, pfsH visits fewer nodes as *zipf* increases but retrieving the *edges* lists of the paths that contain very frequent nodes, costs a lot. Figure 15(c) confirms the above observations with respect to the number of I/Os.

7.5 Updating path collections

Finally, we study the methods for updating path collections. We measure: (a) the time required to update memory-based indices considering the new paths, and (b) the time needed to merge the disk-based indices with the memory-based ones.



(a) (b)

Fig. 16. (a) Updating time varying U, (b) index size varying U for $|\mathbf{P}| = 100000$, $l_{avg} = 10$, $|V| = 100000$, $zipf = 0.6$.

Figure 16(a) illustrates the time required to update the memory-based adjacency lists of $G_{\mathbf{P}}$ graph, and $\mathcal{P}$-$Index$ and $\mathcal{H}$-$Index$ of the path collection. We notice that the updating time of $\mathcal{H}$-$Index$ is higher than the time of adjacency lists and $\mathcal{P}$-$Index$ in all cases. This is due to the fact that we need to access the *edges* lists of the disk-based $\mathcal{H}$-$Index$ to update the memory-based one. On the other hand, in all cases $\mathcal{P}$-$Index$ and the adjacency lists are updated in equal time. Similarly, Figure 16(b) shows that the time needed to merge the disk-based $\mathcal{H}$-$Index$ is higher than the time required for the adjacency lists of $G_{\mathbf{P}}$ graph, and $\mathcal{P}$-$Index$.

8 Conclusions

We consider reachability queries on path collections. We proposed the *path-first search* paradigm, which treats paths as first-class citizens, and further discussed appropriate indices that aid the search algorithms. Methods for updating a path collection and its indices were discussed. An extensive experimental evaluation verified the advantages of our approach. Our ongoing work focuses on index compression techniques for $\mathcal{H}$-$Index$. In the future, we plan to extend our indexing methods to other types of queries, such as shortest path, nearest neighbor and constraint queries.

References

1. Critchlow, T., Lacroix, Z.: Bioinformatics: Managing Scientific Data. Morgan Kaufmann (2003)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Second Edition. The MIT Press and McGraw-Hill Book Company (2001)
3. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. In: SODA. (2002) 937–946
4. Agrawal, R., Borgida, A., Jagadish, H.V.: Efficient management of transitive relationships in large data and knowledge bases. In: SIGMOD Conference. (1989) 253–262
5. Jin, R., Xiang, Y., Ruan, N., Wang, H.: Efficiently answering reachability queries on very large directed graphs. In: SIGMOD Conference. (2008) 595–608
6. Wang, H., He, H., Yang, J., Yu, P.S., Yu, J.X.: Dual labeling: Answering graph reachability queries in constant time. In: ICDE. (2006) 75
7. Agrawal, R., Jagadish, H.V.: Materialization and incremental update of path information. In: ICDE. (1989) 374–383
8. Agrawal, R., Jagadish, H.V.: Direct algorithms for computing the transitive closure of database relations. In: VLDB. (1987) 255–266
9. Lu, H.: New strategies for computing the transitive closure of a database relation. In: VLDB. (1987) 267–274
10. Schenkel, R., Theobald, A., Weikum, G.: Hopi: An efficient connection index for complex xml document collections. In: EDBT. (2004) 237–255
11. Schenkel, R., Theobald, A., Weikum, G.: Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In: ICDE. (2005) 360–371
12. Cheng, J., Yu, J.X., Lin, X., Wang, H., Yu, P.S.: Fast computation of reachability labeling for large graphs. In: EDBT. (2006) 961–979
13. Cheng, J., Yu, J.X., Lin, X., Wang, H., Yu, P.S.: Fast computing reachability labelings for large graphs with high compression rate. In: EDBT. (2008) 193–204
14. Trißl, S., Leser, U.: Fast and practical indexing and querying of very large graphs. In: SIGMOD Conference. (2007) 845–856
15. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. **38**(2) (2006)